FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Webová aplikace pro spolehlivostní modely
Student:	Daniel Vrátil
Vedoucí:	Ing. Martin Daňhel, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Nastudujte teorii spolehlivosti omezte se na Markovské a RBD modely.

Analyzujte možnosti, jak uživatelsky přívětivě pracovat se spolehlivostními modely (graficky vytvářet/editovat/mazat) pomocí GUI webové aplikace. Na základě vytvořených modelů aplikace vygeneruje soubor ve formátu Wolfram Mathematica, kde se dále budou provádět nezávislé výpočty. Předpokládá se spolupráce více uživatelů v rámci projektu v reálném čase.
Projektem se rozumí popis konkrétního modelovaného systému spolehlivostním modelem či modely, včetně podružné dokumentace.
Dle analýzy navrhněte a vytvořte webovou aplikaci:
Serverová část:
Bude umožňovat autentizaci uživatelů a správu jejich rolí. Veškeré informace budou uloženy v databázi. Umožňuje realtime komunikaci s klientem.
Klientská část:
Bude mít přehledné GUI, pro snadnou manipulaci s projekty, modely a uživateli. Zvažte použití vhodných technologií React, knihovnu React Flow, TypeScript.
Výsledenou webovou aplikaci otestujte.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 17. prosince 2021 v Praze.

Bakalářská práce

WEBOVÁ APLIKACE PRO SPOLEHLIVOSTNÍ MODELY

Daniel Vrátil

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Martin Daňhel, Ph.D.
9. května 2022

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat všem, kdo mi při vypracování práce jakkoliv pomáhal. Především pak vedoucímu práce Ing. Martinovi Daňhelovi za vedení práce, odbornou pomoc při studiu tématiky a rady, které mi práci v mnohém usnadnily. Poté RNDr. Alešovi Němečkovi za konzultace o aktivaci programu Wolfram Mathematica. Dále chci poděkovat rodině za plnou podporu 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. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 9. května 2022 .....

Abstrakt

Cílem práce bylo vytvoření nového nástroje pro modelování spolehlivostních modelů ve formě webové aplikace. Práce je zaměřena především na Markovské a blokové spolehlivostní modely, pro které je aplikace navržena. Dále je práce navrhována tak, aby mohlo vícero uživatelů pracovat současně v rámci jednoho projektu, který může čítat několik spolehlivostních modelů. Projekty mohou být popsány formou dokumentace, kterou lze napsat nebo vygenerovat v závislosti na konkrétním projektu. Analytická část modelů je nezávisle zpracována matematickým systémem Wolfram Mathematica.

Klíčová slova blokový spolehlivostní model, Markovský spolehlivostní model, návrh spolehlivostního modelu, PHP, React, real-time synchronizace, webová aplikace, Wolfram Mathematica

Abstract

The goal of the thesis was to create a new tool to build reliability models as a web application. Thesis is mainly focused on Markov and block reliability models. Furthermore, the work is designed in a way that allows multiple users to work simultaneously within a single project, which may consist of several reliability models. Projects can be described in the form of documentation, which can be written or generated depending on the specific project. The analytical part of the models is independently processed by Wolfram Mathematica.

Keywords block reliability model, Markov reliability model, PHP, React, real-time synchronization, reliability model design, web application, Wolfram Mathematica

Seznam zkratek

API	Application Programming Interface
FPMH	Failures per million hours
HTML	Hypertext Markup Language
JWT	JSON Web Token
MTBF	Mean time between failures
MTTF	Mean time to failure
MTTR	Mean time to repair
MVC	Model View Controller
ORM	Object–relational mapping
PHP	PHP: Hypertext Preprocessor
RBD	Reliability block diagram
REST	Representational state transfer
SHARPE	Symbolic Hierarchical Automated Reliability and Performance Evaluator

Kapitola 1
Úvod

Úspěch naší každodenní práce je závislý na využití mnoha zařízení a technologií. Spoléháme na jejich správnou funkci, často navzdory jejich omezené životnosti. Výrobci neustále produkují nová sofistikovanější a komplexnější zařízení a zákazníci požadují čím dál ekonomičtější a spolehlivější provoz. Z toho plyne, že doba po kterou produkt správně funguje, je důležitá nejen pro výrobce, ale především pro jejich zákazníky. Ti si vybírají mimo jiné vhodné produkty právě dle jejich spolehlivostních parametrů jakým je například bezporuchovost. Výrobci se naopak snaží efektivně optimalizovat návrh a výrobu svých produktů. Právě pro tyto účely je důležité, aby se výrobci zabývali otázkami spolehlivosti.

Problematika spolehlivosti je často abstraktní a velmi rozsáhlá. Výrobce se v ní však musí orientovat, aby nabídl zákazníkovi takový produkt, který pro něj bude atraktivní. K tomuto účelu slouží několik nástrojů, jedním z nich jsou spolehlivostní modely, které jsou hlavním tématem této práce. Existuje několik druhů modelů, které se využívají při návrhu zařízení, pro určení spolehlivostních parametrů. Pokud bychom chtěli s takovými modely pracovat, zjistíme, že dostupné nástroje se dělí na tři základní skupiny. Do první z nich lze zařadit veřejně přístupné programy, které jsou však převážně morálně zastaralé a nejsou z dnešního pohledu kompatibilní se současnými a běžně používanými operačními systémy. Na poměry dnešních aplikací nemají ani vhodné uživatelské rozhraní a pro práci se složitějšími či většími modely nejsou příliš použitelné. Druhá skupina zahrnuje zejména privátní či neveřejný sektor, kam spadá podnikový software, který je často „tajně“ vyvíjen právě pro použití v rámci jedné společnosti. A poslední skupina zahrnuje software či nástroje, které jsou sice dostupné a robustní, avšak nepřiměřeně drahé a nevhodné pro použití v menších projektech, školních institucích či pro jednotlivce zabývající se touto tematikou. Není proto divu, že pro studentské a podobné účely se spolehlivostní modely často navrhují pouze ručně na papír či za pomoci grafických editorů a výpočty jsou manuálně přepisovány do výpočetních aplikací třetích stran.

Tato práce je soustředěna na nahrazení nevyhovujících nástrojů první kategorie. Jejich problémy jsou vyřešeny použitím webového prostředí a nových technologií. S příchodem nové aplikace mohou být malé projekty zastřešeny webovou službou a spravovány vícero uživateli současně. Zjednoduší se tím i jejich dostupnost, jelikož pro správu projektů stačí uživateli prohlížeč a internetové připojení. Aplikace je zamýšlena tak, aby obecně tvořila základ pro budoucí práce a rozšíření, které by ji posunuly na úroveň univerzálního nástroje pro uživatele, kteří spadají do její cílové skupiny. Hlavní částí je návrh spolehlivostních modelů. Existuje jich mnoho typů, například blokové, Markovské, stromy poruch a další. V této práci se zaměřuji pouze na blokové a Markovské spolehlivostní

modely. I přesto je návrh a implementace zpracována tak, aby bylo možné aplikaci jednoduše rozšířit o další modely a to bez nutnosti změny již stávajícího kódu.

Jednotlivé modely jsou z důvodu přehlednosti členěny pod projekty. Jelikož se jedná o webovou aplikaci s přístupem vícero uživatelů, je projekt úrovní, ve které je možné nastavit uživatelům přístup a oprávnění. Ke spolehlivostním modelům je v tom případě důležité znát kontext, jaké součástky jsou modelem vyjádřeny, k čemu se používají a další užitečné informace. Ty jsou zaznamenány formou dokumentace projektu. Referenční nástroje podporují jen jednoduchý popis modelu nebo jeho částí. Souhrn těchto poznámek však není možné nijak vyexportovat do jednoho dokumentu. To vede k vytváření souborů v externích aplikacích, které jsou ukládány mimo projekt. Z toho důvodu je v aplikaci možné vytvořit pro každý projekt dokumentaci. Sepsat ji lze v textovém editoru, který je připraven speciálně pro tyto účely. Díky přítomnosti dokumentace přímo v aplikaci je možné základní podobu dokumentace vygenerovat pomocí předem připravených šablon.

Důvodem k tvorbě modelů je pro výrobce optimalizace spolehlivostních parametrů výsledného produktu. K jejich dalšímu zkoumání je ovšem nutné je nejprve získat. To lze provést mnoha způsoby, veřejně dostupné nástroje z první kategorie volí především výpis samostatných výsledků. V této práci využiji export nezávislých výpočtů pro externí výpočetní software, konkrétně pro program Wolfram Mathematica. V rámci takto vygenerovaného notebooku budou uživatelé schopni vyzkoušet různé hodnoty samostatně bez zásahu do modelu. Budou moci tak lépe analyzovat parametry navržených modelů a ty poté optimalizovat.

První kapitoly práce se nejprve věnují vysvětlení základních termínů a uvedení do problematiky spolehlivosti. Představeny jsou základní spolehlivostní modely a podrobně vysvětleny jejich parametry, výpočty a grafy. V kapitole *Analýza* jsou rozebrány dostupné aplikace se všemi svými přínosy a nevýhodami. Pro potřeby implementace jsou popsány technologie a platformy, které lze použít při implementaci. Následuje kapitola *Návrh a implementace*, ta má za cíl s pomocí informací získaných v analýze navrhnout a následně vytvořit webovou aplikaci. Na počátku návrhu jsou zvoleny používané technologie. Návrh pokračuje doménovým a relačním modelem pro databázi. Dále návrhem rozhraní pro komunikaci s klienty. V klientské části stojí za zmínku design a uživatelské rozhraní pro sestavení modelu. V neposlední řadě je popsána synchronizace mezi uživateli a její použití ze strany serveru i klienta. Práce je zakončena kapitolou *Testování a nasazení*, v ní jsou popsány všechny typy testů obou částí aplikace a nasazení včetně konfigurace a licencování některých nástrojů.

1.1 Motivace a cíle práce

Pokud nastane situace, kdy uživatelé musí při návrhu zařízení či systému pracovat s jeho spolehlivostí, potřebují k tomu příslušné nástroje. V případě spolehlivostních modelů je malý výběr aplikací, které slouží k tomuto účelu. Všechny mně dostupné aplikace jsou více než 15 let staré, jejich vývoj je již zastaven a to má dopad na jejich použití. Ostatní aplikace, které nejsou dostupné typicky vyvíjejí velké společnosti, které je využívají hlavně pro interní vývoj. Pokud jsou přístupné veřejnosti, tak jsou často velmi drahé a složité, tím nejsou vhodné pro malé a středně velké projekty.

Mojí motivací je umožnit cílové skupině uživatelů, v podobě menších týmů, studijních institucí nebo jednotlivců, používat moderní formu nástroje pro práci se spolehlivostními modely.

Cílem práce je naprogramování dostupné webové aplikace, která bude vhodná pro využití při práci s malými či středně velkými projekty. Dále by měla být levná a přístupná také pro studijní účely. Oproti dosavadním aplikacím by měla být inovativní, využívat moderní technologie a využívat moderní prvky, které pomůžou aplikace na straně uživatelské přívětivosti. Aplikace by měla umožnit vhodnou práci v celého týmu v rámci celého projektu, nejen tedy při práci se spolehlivostním modelem, ale také související při práci na souvisejících úkolech, jako je například dokumentace.

Kapitola 2
Spolehlivostní analýza

Abych byl schopen provést analýzu potřebnou pro další postup při tvorbě aplikace, musím zde nejprve přiblížit a vysvětlit termíny jako spolehlivost, spolehlivostní model a další terminologii. Kapitola popisuje veškerou teorii potřebnou pro analýzu a návrh aplikace.

2.1 Význam spolehlivosti

Spolehlivost je obecný termín a jeho význam není příliš intuitivní. Podle definice převzaté z [1] se jedná o „obecnou vlastnost objektu spočívající ve schopnosti plnit požadované funkce při zachování hodnot stanovených provozních ukazatelů v daných mezích a v čase podle stanovených technických podmínek“. Nyní je vhodné si vysvětlit některé části této definice. Spolehlivost sama o sobě není jednoznačná, jedná se spíše o komplexní vlastnost nějakého objektu (zařízení). Ta zahrnuje další specifické vlastnosti jako je bezporuchovost, životnost, skladovatelnost a další. S některými z nich dále pracuji a proto budou později v této kapitole blíže vysvětleny. V definici jsou uvedeny technické podmínky, ty představují specifické technické vlastnosti pro správnou funkci objektu. Ty jsou předem stanoveny a jedná se například o provoz, údržbu nebo opravu. Dále musejí být zachovány hodnoty stanovených provozních ukazatelů. Zde se jedná například o ukazatele rychlosti, spotřeby, produktivity a podobné. [1]

Za objekt z definice spolehlivosti považujeme libovolný celek, který je možné zkoumat najednou. Při určování objektů záleží na kontextu zkoumání. Například můžeme jako objekt označit součástku, obvod nebo celý systém. Při zkoumání spolehlivosti se objekty typicky nachází v bezporuchovém a poruchovém stavu. V bezporuchovém stavu se objekt nachází pokud je (dle definice spolehlivosti) schopen plnit požadované funkce, v opačném případě jde o poruchový stav. [1]

V souvislosti s tím rozdělujeme objekty na dvě kategorie, obnovované a neobnovované. Jak naznačuje název, obnovované objekty lze změnit z poruchového do bezporuchového stavu. Tento přechod se nazývá oprava. Neobnovovaný objekt je takový, nemůže být po poruše znovu opraven. Důvody mohou být různé, například situace, kdy je nepřístupný nebo jen není oprava prováděna. [1]

2.1.1 Neobnovované objekty

Při měření spolehlivosti není předem známo, kdy přejde objekt do poruchového stavu. Vyvolání této akce obsahuje míru náhody, proto budeme při predikci počítat s mírou pravděpodobnosti přechodu. Mluvíme-li o neobnovovaných objektech znamená to, že pokud nastane takový přechod, objekt již opravit nelze. Pravděpodobnost výskytu poruchy je v čase nutně rostoucí. Náhodná veličina je tedy charakterizována distribuční funkcí. Označme náhodnou veličinu x reprezentující poruchu. Pak je distribuční funkce $F_x(t)$ vypočítána jako

$$F_x(t) = P(x < t). \tag{2.1}$$

Při zkoumání neobnovovaných objektů je zajímavá především informace, kdy nastane přechod, proto zkoumáme hodnotu t od počátku do doby, kdy dojde k poruše. V souvislosti s tím definujeme pravděpodobnost poruchy objektu, která se značí $Q(t)$.[1]

Již známe výpočet pravděpodobnosti, že porucha nastane, z praktických důvodů je zajímavější opačná pravděpodobnost, že se objekt nachází v bezporuchovém stavu. Tuto veličinu nazýváme pravděpodobnost bezporuchového stavu a značíme $R(t)$. Jde o převrácenou hodnotu již zmiňované $F_x(t)$ a její výpočet je následující

$$R(t) = 1 - Q(t). \tag{2.2}$$

Za předpokladu, že je náhodná veličina spojitá, má smysl počítat její hustotu pravděpodobnosti (značíme $f(t)$). Pokud mluvíme o hustotě pravděpodobnosti bezporuchového stavu nazýváme ji hustotou poruch a získáme vztahem

$$f(t) = \frac{dQ(t)}{dt}. \tag{2.3}$$

Pomocí hustoty poruch vypočítáme intenzitu náhodné veličiny. Ta udává "podmíněnou hustotu poruch v čase t za předpokladu, že k poruše dosud nedošlo". Jde se o důležitou veličinu často používanou v praxi. Označujeme ji $\lambda(t)$ a její hodnota je vypočtena podílem hustoty poruch a pravděpodobností bezporuchového stavu, tedy

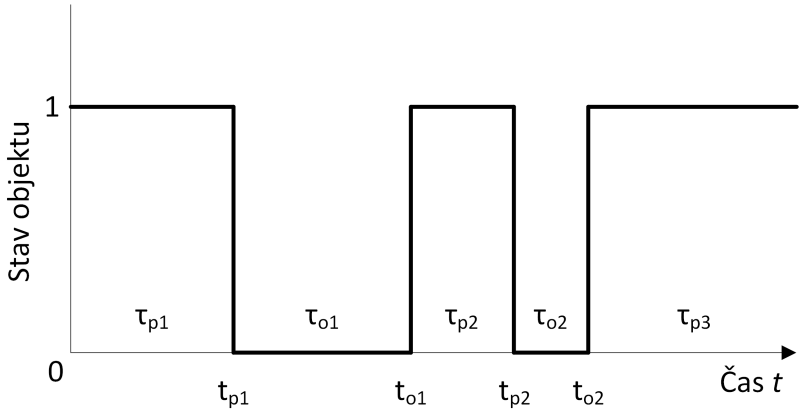
$$\lambda(t) = \frac{f(t)}{R(t)}. \tag{2.4}$$

Ve statistice se běžně při zkoumání pravděpodobností počítá střední hodnota náhodné veličiny. Ta se pro spojité veličiny počítá jako integrál distribuční funkce od 0 do ∞. Zde má smysl definovat střední dobu bezporuchového stavu (T_s). Ta představuje střední dobu do poruchy objektu (MTTF).[1]

$$T_s = \int_0^\infty R(t)dt \tag{2.5}$$

2.1.2 Obnovované objekty

Obnovované objekty v čase přechází mezi bezporuchovým a poruchovým stavem. Průběh v čase je znázorněn na obrázku 2.1.



Obrázek 2.1 Stav obnovovaného objektu v čase - Obnovované objekty v čase mění svůj stav. Na svislé ose jsou promítnuty stavy objektu. Hodnota 1 značí bezporuchový stav, hodnota 0 značí poruchový stav.

Na ose x jsou zakresleny události t_{pi}, které reprezentují i-tou poruchu, a t_{oi} reprezentující i-tou opravu. Stav objektu zobrazujeme v závislosti na čase. Pokud je hodnota v čase t rovna 1, objekt je v bezporuchovém stavu, naopak hodnota rovna 0 označuje stav poruchový. Na uvedeném grafu je počáteční stav bezporuchový. Délka trvání i-tého úseku poruchy je τ_{pi} a délka trvání od i-té poruchy do i-té opravy je τ_{oi}.[1]

Při zkoumání obnovovaných objektů je důležitá informace jak dlouho se v průměru bude objekt nacházet v bezporuchovém stavu, neboli jak dlouho bude objekt funkční. Tuto informaci získáme podílem součtu všech časů poruch a jejich celkového počtu. Výsledek nazýváme střední dobou mezi poruchami (MTBF).[2]

$$T_s = \frac{\sum_{i=1}^{n} \tau_{pi}}{n} \tag{2.6}$$

Střední doba mezi poruchami značí střední dobu, po kterou je objekt funkční. Pokud bychom potřebovali získat střední dobu od opravy i do opravy $i+1$, tedy včetně doby po kterou objekt funkční není, musíme nahradit časy poruch za časy oprav. Označme tuto informaci jako střední dobu cyklu (T_c).[2] Triviálně lze říct, že hodnota T_c bude vetší nebo rovna (za předpokladu instantní opravy) než T_s.

Nyní máme definované střední doby a chceme získat pravděpodobnost, že v určitý čas t bude objekt funkční. To vyjadřuje okamžitý součinitel pohotovosti ($K_p(t)$). K němu zadefinujme stacionární součinitel pohotovosti, který je jeho limitou pro t jdoucí do nekonečna. Ten je určen jednoduchým podílem

$$K_p = \frac{t_p}{t_p + t_o} = \frac{\sum_{i=1}^{n} t_{pi}}{\sum_{i=1}^{n} t_{pi} + \sum_{i=1}^{n} t_{oi}}. \tag{2.7}$$

Pro další výpočty zavedeme střední dobu opravy (MTTR), ta se počítá obdobně jako T_s.[1]

$$T_o = \frac{\sum_{i=1}^{n} \tau_{oi}}{n} \tag{2.8}$$

V praktickém využití figuruje střední frekvence oprav reprezentující frekvenci, se kterou jsme schopni v daných podmínkách periodicky provádět opravy objektu (značíme μ). S jejím využitím lze střední dobu oprav spočítat jako

$$T_o = \frac{1}{\mu}. \tag{2.9}$$

a po dosazení se stacionární součinitel pohotovosti zjednoduší na

$$K_p = \frac{T_s}{T_s + T_o} = \frac{\mu}{\mu + \lambda}. \tag{2.10}$$

K_p je pravděpodobnost, triviálně tedy můžeme získat její doplněk nazývaný součinitel prostoje.[2] Okamžitý součinitel prostoje získáme doplňkem okamžitého součinitele pohotovosti

$$K_n(t) = 1 - K_p(t) \tag{2.11}$$

a obdobně pro stacionární součinitel prostoje

$$K_n = 1 - K_p. \tag{2.12}$$

2.2 Spolehlivostní model

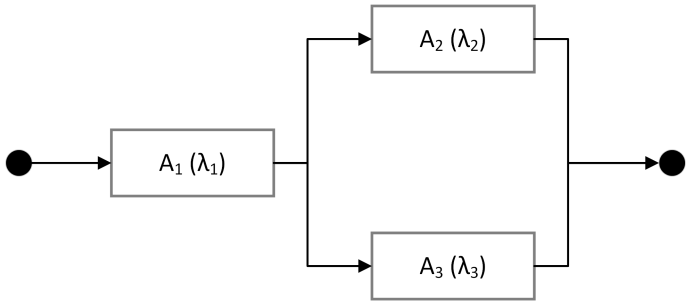
Již máme zadefinovány základní definice pro výpočet některých faktorů spolehlivosti. Pokud navrhujeme nějaké zařízení nebo systém, potřebujeme tyto faktory spolehlivosti předvídat. K dispozici máme obvykle objekty (součástky), s jejichž pomocí navrhujeme nový výsledný objekt (zařízení, systém, ...). Pro zhodnocení výsledných faktorů spolehlivosti používáme modely. Existuje jich mnoho a každý z nich je zaměřen na odlišné faktory a závislosti mezi jednotlivými objekty. Zde popíši jen ty modely, které jsou důležité pro výslednou aplikaci. Těmi jsou blokový a Markovský model.

2.2.1 Blokový model

Blokový model, neboli model s nezávislými prvky se využívá v situaci, kdy chceme zjistit, jak každý objekt přispívá k nespolehlivosti celého systému. Jeho výhoda je především v jednoduchosti zápisu modelu i jeho základních výpočtů.

Prvky jsou vyobrazeny jako bloky, které jsou dle jejich závislostí propojeny orientovanými hranami. Prvek reprezentuje jeden objekt výsledného systému a zpravidla je označen především názvem a intenzitou poruch, může však obsahovat další doplňující informace. Vyhodnocení zapojení probíhá po směru orientace hran – ty se nesmějí vracet do předchozího prvku (nesmí se zacyklit). Každé zapojení má právě jeden vstup a výstup, jejich reprezentace není předem určena, může to být speciální prvek nebo vstupní (výstupní) hrana.

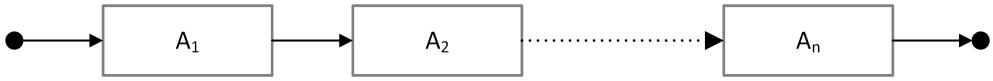
Při vyhodnocení se prochází všechny cesty ze vstupního do výstupního bodu a pokud existuje cesta, na které jsou všechny prvky v bezporuchovém stavu, pak je celý objekt také v bezporuchovém stavu. Model na obrázku 2.2 bude v bezporuchovém stavu pokud jsou v bezporuchovém stavu také prvky A_1 a zároveň A_2 nebo A_3, avšak provozuschopný, pokud budou v bezporuchovém stavu prvky A_1 a A_2 nebo A_1 a A_3. Vyhodnocení a následný výpočet je závislý na typu zapojení, ty jsou celkem tři a nyní si je podrobně popíšeme.



Obrázek 2.2 Jednoduchý blokový model - Model se třemi prvky: A_1, A_2, A_3. V tomto případě jsou počáteční a koncové prvky zobrazeny bodem a intenzity prvků jsou psané v závorkách. Například prvek A_1 má intenzitu poruch λ_1.

Sériový model

Funguje-li několik prvků tak, že výpadek jednoho z nich způsobí výpadek celého zapojení, nazýváme jejich propojení sériové.



Obrázek 2.3 Sériový model - Model s n sériovými prvky.

Výsledná pravděpodobnost bezporuchového stavu ($R_s(t)$) tohoto zapojení je rovná součinu pravděpodobností bezporuchového stavu ($R_i(t)$) všech prvků.[2]

$$R_s(t) = \prod_{i=1}^{n} R_i(t) \tag{2.13}$$

Za předpokladu, že má každý prvek konstantní intenzitu poruch λ_i a výsledná intenzita poruch je

$$\lambda = \sum_{i=1}^{n} \lambda_i, \tag{2.14}$$

získáme výslednou $R_s(t)$ pomocí zjednodušení na

$$R_s = \prod_{i=1}^{n} e^{-\lambda_i} = e^{-\lambda_i}. \tag{2.15}$$

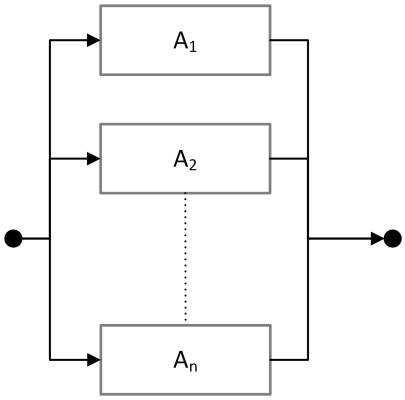
Jelikož známe celkovou intenzitu poruch λ, jednoduše získáme střední dobu bezporuchového provozu (T_s) jako

$$T_s = \frac{1}{\lambda}. \tag{2.16}$$

V blokovém modelu může jeden prvek reprezentovat libovolně velkou část celého systému. Pro výslednou hodnotu záleží pouze na jeho parametrech (intenzitě poruch) a zapojení vzhledem k ostatním prvkům. Máme-li sériové zapojení s n prvky a intenzitami poruch $\lambda_1, \lambda_2, ..., \lambda_n$ s celkovou intenzitou $\lambda_s = \sum_{i=1}^{n} \lambda_i$, můžeme těchto n prvků nahradit jedním prvkem s intenzitou poruch právě λ_s.[1]

Paralelní model

Druhou možností je paralelní zapojení. To využijeme v případě, kdy pro poruchu celého objektu musí nastat porucha na všech paralelně zapojených prvcích.[2]



Obrázek 2.4 Paralelní model - Model s n paralelně zapojenými prvky.

V případě paralelního zapojení lze jednoduše získat pravděpodobnost poruchového stavu ($Q_s(t)$) součinem všech $Q_i(t)$ pro všechny prvky zapojení.

$$Q_p(t) = \prod_{i=1}^{n} Q_i(t) \tag{2.17}$$

Pomocí doplňku získáme pravděpodobnost bezporuchového stavu.

$$R_p(t) = 1 - \prod_{i=1}^{n} Q_i(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t)) \tag{2.18}$$

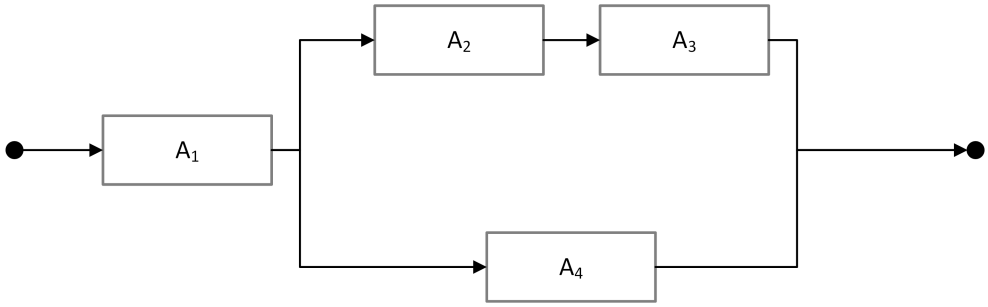
Střední dobu bezporuchového stavu lze po součtu intenzit spočítat jako

$$T_s^p = \frac{1}{\lambda_p} \sum_{i=1}^{n} \frac{1}{i} \tag{2.19}$$

Podobně jako u sériového modelu i paralelní lze sloučit do jednoho jediného bloku. Máme-li paralelní zapojení s n prvky a intenzitami poruch $\lambda_1, \lambda_2, ..., \lambda_n$ s celkovou intenzitou $\lambda_p = \prod_{i=1}^{n} \lambda_i$, můžeme těchto n prvků nahradit jedním prvkem s intenzitou poruch právě λ_p.[1]

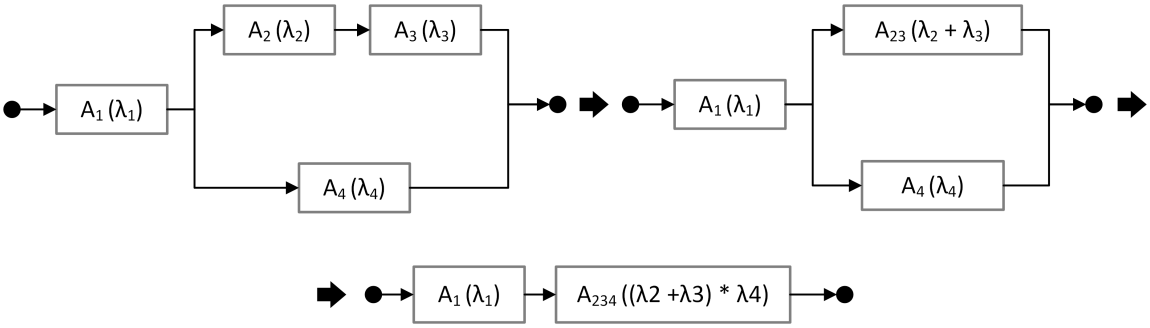
Kombinovaný model

Posledním základním typem zapojení blokových modelů je kombinovaný. Model je kombinovaný pokud obsahuje prvky které jsou zapojeny sériově a zároveň prvky, které jsou zapojeny paralelně.[2]



Obrázek 2.5 Kombinovaný model - Sériové zapojení je mezi prvky A2 a A3 nebo A1 a trojicí prvků (A2, A3, A4). Paralelní zapojení nalezneme mezi dvojicí prvků (A2, A3) a prvkem A4.

Zapojení kombinovaných modelů může být velmi obecné a proto pro něj neexistuje univerzální vzorec výpočtu výsledných hodnot. Víme však jakým způsobem získat hodnoty jako $R(t)$ nebo T_s pro paralelní a sériové modely – kombinovaný model není nic jiného než několik sériových a paralelních podsystémů. Všechny prvky, které jsou sériově či paralelně zapojeny lze sloučit do jednoho jediného prvku s celkovou intenzitou poruch. Postupným slučováním prvků vždy nutně dostaneme samotné sériové nebo paralelní zapojení, pro která již máme postup výpočtu. Na obrázku 2.6 je po krocích znázorněno postupné slučování prvků s počátečním modelem z obrázku 2.5. V závorkách můžeme vidět, jak sloučení ovlivní intenzitu poruch.



Obrázek 2.6 Zjednodušení kombinovaného modelu - V prvním kroku se sloučily prvky A_2 a A_3 do prvku A_{23} a intenzita jejich poruch se sečetla. Ve druhém kroku se sloučily prvky A_{23} a A_4 a jejich intenzity se vynásobily. Třetí model je sériový a lze řešit standardním způsobem.

Ve výše rozepsaném případě postupujeme obdobně i pro výpočet pravděpodobnosti bezporuchového stavu a střední doby bezporuchového stavu. Po prvním sloučení prvků A_2 a A_3 získáme prvek A_{23} a s ním hodnotu $R_{23} = R_2 \cdot R_3$. V druhém kroku slučujeme prvky A_{23} a A_4. Nově vzniklým prvkem je A_{234} s $R_{234} = 1 - (1 - R_{23}) \cdot (1 - R_4)$. Zbývá jen vyhodnotit zbylý sériový model a výsledná pravděpodobnost bezporuchového stavu je $R = R_1 \cdot R_{234}$.

2.2.2 Markovský model

Předchozí kapitola se zabývá jednoduchým modelem využívaným k zjištění nespolehlivosti každého prvku systému, kde se předpokládá nezávislost jednotlivých prvků. Pokud narazíme na systém, který nelze dále dělit na nezávislé prvky, jsme nuceni použít složitější model. Jedním z nich

je právě Markovský model, který se používá pro „určování hodnot spolehlivostních ukazatelů systému se závislými prvky“. Funguje na principu stavů a přechodů. K přechodům z jednoho stavu do druhého dochází v náhodném čase a s nějakou pravděpodobností.

Markovský náhodný proces

Náhodný proces označme funkcí $X(t)$, jejíž hodnota v každé hodnotě t je náhodná veličina. Předpokládejme, že je náhodný proces diskrétní, množina argumentů t je spojitá (jedná se o čas) a obor hodnot je diskrétní. Prvky oboru hodnot nazvěme stavy a jejich počet označme n. Ke změně stavu může dojít v libovolném t, tedy v libovolném časovém bodě. Takový náhodný proces nazýváme diskrétní Markovský proces. Při zjišťování pravděpodobnosti, že v čase t dojde k přechodu ze stavu X_i do stavu X_j, závisí výsledek pouze na původním stavu (X_i). Díky této vlastnosti můžeme sestavit matici, která má v i-tém řádku a j-tém sloupci pravděpodobnost přechodu ze stavu X_i do stavu X_j. Nazýváme ji maticí přechodů (angl. Rate Matrix).[1]

$$P = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,n} \end{pmatrix}$$

Obrázek 2.7 Obecná matice přechodů pro Markovský model s n stavy. Hodnota $p_{i,j}$ označuje pravděpodobnost přechodu ze stavu X_i do stavu X_j.

Matice se skládá z podmíněných pravděpodobností přechodů ze stavu X_i do stavu X_j. Nyní označme elementární časový interval jako dt. Pokud se v čase t nacházíme ve stavu X_i, tedy $X(t) = i$, pak pravděpodobnost, že se po uplynutí času dt nacházíme ve stavu X_j, neboli $X(t + dt) = j$, je rovna $p_{i,j}$. Pokud jsou všechny pravděpodobnosti přechodů ($p_{i,j}$ pro $i <= n, j <= n$) konstantní (v čase neměnné) jedná se o homogenní markovský proces.

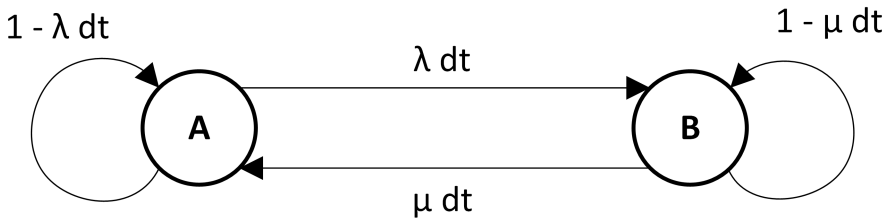
Vzhledem k tomu, že k přechodu vždy musí dojít (doba mezi přechody není omezena) je součet všech pravděpodobností v řádku matice přechodů roven 1.[1]

$$\sum_{j=1}^{n} P_{i,j} = 1 \tag{2.20}$$

Při počítání s diskrétním markovským procesem zadefinujme důležitou vlastnost. Tou je intenzita přechodu ze stavu X_i do stavu X_j, značíme $\lambda_{i,j}$. Je-li proces homogenní, pak je $\lambda_{i,j}$ konstantní pro všechna t. Intenzita přechodu úzce souvisí s intenzitou poruch. Pokud máme nějaký stav X_i, reprezentující bezporuchový stav součástky, a přechod do jiného stavu X_j, který značí poruchový stav součástky, pak je intenzita přechodu ekvivalentní k intenzitě poruchy této součástky. Pomocí intenzity přechodu můžeme získat podmíněnou pravděpodobnost téhož přechodu vztahem

$$p_{i,j} = \lambda_{i,j} \cdot dt. \tag{2.21}$$

Na následujícím příkladu Markovského modelu dvou stavů je blíže ukázána práce s maticí přechodů, pravděpodobnostmi přechodů a dalšími vlastnostmi. Na obrázku 2.8 máme příklad Markovského modelu se stavy A a B, ten použijeme jako předlohu pro další ukázku. [1]



Obrázek 2.8 Markovský model se dvěma stavy, který je popsán v příkladu.

Model reprezentuje funkci jedné součástky, stav A znázorňuje bezporuchový stav součástky, stav B naopak její poruchový stav. Mějme pravděpodobnost poruchy, tedy pravděpodobnost přechodu ze stavu A do stavu B, rovnou $\lambda \cdot dt$. Přechod opačným směrem reprezentuje opravu součástky, tudíž bude intenzita přechodu rovna intenzitě opravy, označíme ji $\mu \cdot dt$. Jelikož se součet pravděpodobností přechodů vycházejících z libovolného stavů musí rovnat 1, doplňme přechod ze stavu A zpět do stavu A roven $1 - \lambda \cdot dt$ a obdobně pro stav B. Z takto sestaveného modelu můžeme vytvořit následující matici přechodů.[1]

$$P = \begin{pmatrix} p_{A,A} & p_{A,B} \\ p_{B,A} & p_{B,B} \end{pmatrix} = \begin{pmatrix} 1-\lambda\,dt & \lambda\,dt \\ \mu\,dt & 1-\mu\,dt \end{pmatrix}$$

Obrázek 2.9 Matice přechodů pro Markovský model - Model má dva stavy, proto má matice přechodů rozměry 2x2. První řádek odpovídá výstupním přechodům ze stavu A, druhý řádek odpovídá výstupním přechodům stavu ze B. V druhé matici jsou dosazeny hodnoty těchto pravděpodobností.

Nyní máme vytvořený model i matici přechodů a dále chceme získat pravděpodobnost, zda se v náhodném čase t součástka nachází ve stavu A (funkční) nebo ve stavu B (porucha). Zadefinujme tedy $p_A(t)$ a $p_B(t)$, které značí, že je součástka v čase t ve stavu A resp. B. Víme, že toto jsou jediné dva stavy, ve kterých se součástka může nacházet, můžeme tedy říct, že $p_A(t) + p_B(t) = 1$. Z matice přechodů získáme následující soustavu rovnic.[1]

$$\begin{aligned} p_A(t+dt) &= (1-\lambda \cdot dt) \cdot p_A(t) + \mu \cdot dt \cdot p_B(t) \\ p_B(t+dt) &= (1-\mu \cdot dt) \cdot p_B(t) + \lambda \cdot dt \cdot p_A(t) \end{aligned} \tag{2.22}$$

Vysvětlení významu těchto rovnic si ukážeme na první z nich. Součástka se v nějakém čase $t+dt$ nachází ve funkčním stavu tehdy, pokud se v předchozím časovém bodě t nacházela ve funkčním stavu a její stav se nezměnil a nebo se nacházela ve stavu poruchy a byla opravena. Obdobně lze vysvětlit význam druhé rovnice.[1]

Tyto rovnice můžeme dále upravit vydělením dt a získat tento tvar.

$$\frac{p_B(t+dt) - p_B(t)}{dt} = -\mu \cdot p_B(t) + \lambda \cdot p_A(t) \tag{2.23}$$

Levé strany jsou rovny derivaci $p_X(t)$, proto je můžeme následně zjednodušit na

$$\begin{aligned} p'_A(t) &= -\lambda \cdot p_A(t) + \mu \cdot p_B(t) \\ p'_B(t) &= -\mu \cdot p_B(t) + \lambda \cdot p_A(t). \end{aligned} \tag{2.24}$$

Výsledkem těchto úprav jsou dvě lineární diferenciální rovnice s konstantními koeficienty. Pokud známe počáteční podmínky ($p_A(0)$, $p_B(0)$) lze tyto soustavy dále řešit nahrazením jedné z nich normalizační podmínkou.

Před další kapitolou musíme definovat pojem **absorpční stav**. Stav je absorpčním právě tehdy, když nemá žádné výstupní přechody. Jedná se tedy o stav, ve kterém považujeme průběh za skončený. V kontextu systémů se většinou jedná o situaci, kdy nastala chyba, kterou již nelze opravit. Toto bude mít v následujících částech kapitoly několik důsledků na celkovou spolehlivost modelovaného systému.[1]

Markovské modely s absorpčními stavy

Tento typ Markovských modelů se používá pro určení spolehlivostních ukazatelů neobnovovaných systémů. Absorpční stavy zde reprezentují poruchy, které již není možné opravit a pokud nastanou, celý systém již nepřejde do funkčního stavu.

Pro výpočet bezporuchového stavu nejprve sloučíme všechny absorpční stavy do jednoho a hrany vedoucí do původních stavů upravíme tak, aby vedli do nově vytvořeného jediného absorpčního stavu. Dále vyjádříme soustavu lineárních diferenciálních rovnic, podobně jako v 2.24. Výslednou pravděpodobnost bezporuchového stavu ($R(t)$) získáme součtem pravděpodobností všech bezporuchových stavů. Pravděpodobnost nevratné poruchy (Q(t)) je rovna pravděpodobnosti absorpčního stavu.[1]

$$R(t) = \sum_{i=1}^{n} k_i \cdot p'_i(t) \tag{2.25}$$

Střední doba bezporuchového provozu T_s se stejně jako v předchozích případech vypočítá integrací $R(t)$.

$$T_s = \int_0^{\infty} R(t)dt \tag{2.26}$$

Markovské modely bez absorpčních stavů

Je-li systém ve všech svých stavech obnovovaný, při každé poruše části systému může nastat oprava. Systém se tedy může vrátit zpět do funkčního stavu. Z toho vyplývá, že Markovský model tohoto systému nemá žádný absorpční stav. Markovský model bez absorpčních stavů se využívá právě pro takové systémy, u kterých nikdy nenastane stav trvalé poruchy. V teoretické rovině uvažujeme nekonečnou délku běhu systému.

Jelikož průběh přechodovým grafem (modelem) nikdy neskončí a z každého stavu existuje cesta do dalších stavů, zavádíme střední frekvenci přechodů, značenou $f_{i,j}$, jako průměrný počet přechodů ze stavu X_i do stavu X_j za jednotku času. Ta je rovna součinu ustálené pravděpodobnosti stavu X_i a intenzitou přechodu z X_i do X_j.[1]

$$f_{i,j} = p_i \cdot \lambda_{i,j} \tag{2.27}$$

Přesuňme se k výpočtu stacionárního součinitele pohotovosti ($K_p(t)$) a prostoje ($K_p(t)$). Tyto dvě hodnoty získáme ze soustavy lineárních diferenciálních rovnic (podobně jako $R(t)$ a $Q(t)$ u modelu s absorpčními stavy). V soustavě jednu z rovnic nahradíme normalizační podmínkou a poté dopočítáme ustálené pravděpodobnosti všech stavů. Stacionární součinitele získáme následujícími vztahy.[1]

$$K_p = \sum_i p_i, \tag{2.28}$$

kde i prochází všechny bezporuchové stavy.

$$K_n = \sum_j p_j, \tag{2.29}$$

kde j prochází všechny poruchové stavy.

Kapitola 3
Analýza

Mým cílem pro tuto práci je vytvořit webovou aplikaci. Primárně se jedná o práci se spolehlivostními modely. Problematika kolem spolehlivosti je poměrně rozsáhlá a příslušných funkcí pro aplikaci tohoto typu můžeme vymyslet mnoho. Například existuje mnoho typů různých spolehlivostních modelů. Aplikace by tedy měla být navržena dostatečně obecně tak, aby bylo možné v budoucnu přidávat nové typy spolehlivostních modelů. To by mělo probíhat pouze implementací nových funkcí bez přepisování starého kódu. V rámci cílů nejsou zadány pouze modely, ale také dokumentace, která je důležitou součástí každého návrhu zařízení.

V dnešní době je výhodné mít práci přístupnou odkudkoliv bez komplikovaných přenosů souborů. Tento přístup je platný nejen ve chvíli, kdy uživatel změní zařízení, ale také při práci týmu na jednom projektu či modelu. V průběhu analýzy počítám s touto myšlenkou a kladu důraz na popsání řešení těchto problémů.

3.1 Existující řešení

Prvním krokem této analýzy by měl být rozbor podobných řešení. Následuje tedy podrobný popis aplikací, které řeší stejnou problematiku. Jak již bylo zmíněno v úvodu, problém spolehlivosti je velmi široký a obsáhlý. Existují řady typů spolehlivostních modelů. Nicméně nástrojů, které by uživatelům umožnily s těmito modely pracovat, vytvářet dokumentaci nebo projekty interaktivně sdílet s ostatními členy týmu, není mnoho. Většina z nich je navíc dostupná pouze pro firmy nebo po nákupu licence. Proto jsou zde uvedeny pouze mě dostupné nástroje.

Aplikace rozdělme do dvou kategorií. Do první z nich patří ty, které opravdu umožňují navrhnout spolehlivostní model a na základě návrhu vyhodnotit výsledky. Takové aplikace přímo souvisí s řešením této práce a některé funkce se nutně musí překrývat. Zde je důležité identifikovat výhody, které mohou být využity, a nevýhody, kterým se vyvarovat.

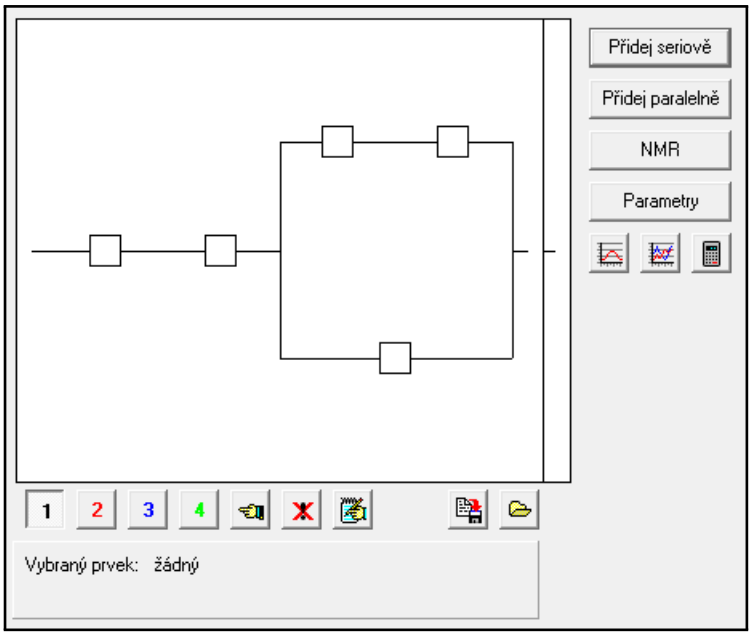
Do druhé kategorie zařaďme aplikace, které neslouží k samotnému návrhu, ale s touto prací souvisí. Jedná se hlavně o aplikace či služby, které řeší obdobnou problematiku nebo pouze její část. Od analýzy těchto nástrojů si slibuji identifikaci zajímavých funkcí, které se běžně v obdobných aplikacích nevyskytují. Mohou být využity pro rozšíření rozsahu problematiky, který aplikace pokrývá. Například přidání zajímavých výpočtů nebo seznam základních součástek s jejich vlastnostmi.

3.1.1 Program Spolehlivost

První vybranou aplikací je program Spolehlivost. Jedná se o jednoduchý program vytvořen jako semestrální práce studentem Radkem Jelínkem. Vytvořen je v grafickém prostředí Delphi a programovacím jazyce Object Pascal pro operační systém Windows 2000/XP. Delphi díky svému prostředí mohlo být v době, kdy byl program vytvořen, dobrým nástrojem pro rychlý vývoj podobné aplikace. Dnes má pochopitelně problémy s kompatibilitou a na jiných operačních systémech než je Windows nelze ani spustit. Já jsem pro účely analýzy aplikaci zkoušel na Windows 10 a i zde jsem narazil na chyby primárně grafické rázu, které by mohly být způsobeny rozdílem v operačním systému. Po obsahové stránce je program omezen pouze na tvorbu blokových spolehlivostních modelů. Z vytvořeného modelu umí vygenerovat základní výsledky a grafy.

Návrh blokového modelu

Návrh blokového spolehlivostního modelu probíhá v hlavní části okna aplikace. Bloky jsou značeny malými čtverci, které jsou spojeny linkami reprezentujícími hrany. Značné omezení nastává při přidávání nového bloku. To je řízeno dvěma tlačítky, *Přidat sériově* a *Přidat paralelně*. Při přidání nového bloku potřebujeme nejprve nějaký zvolit. Poté pomocí těchto tlačítek přidáme nový dle zvoleného typu zapojení. Druhým velkým problémem je skutečnost, že stávající bloky nelze smazat, uděláme-li chybu při návrhu, musíme začít znovu. Z toho vyplývá, že pokud vytváříme model, měli bychom ho mít už připravený a aplikaci využít převážně pro výpočty.



Obrázek 3.1 Blokový model v programu Spolehlivost - Hlavní okno je rozděleno na vizualizaci modelu, kontrolní tlačítka pro ovládání modelu (po pravé straně) a kontrolní tlačítka pro přepínání mezi modely, ukládání a další funkce (na spodní části okna). Ve vizualizaci modelu je patrný grafický problém – svislá čára na pravém okraji modelu.

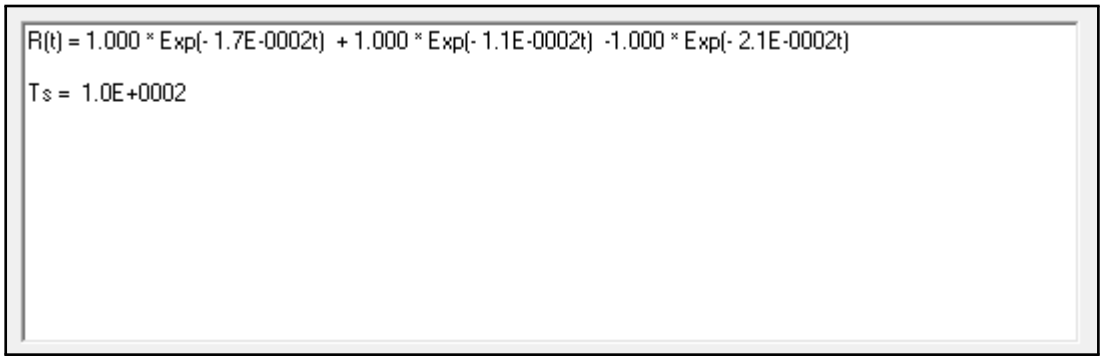
Výpočetní část

Jelikož program umí pracovat pouze s blokovým spolehlivostním model, výsledné výstupy v podobě grafů a výpočtů závisí na dvou faktorech. Prvním je sestavení modelu, neboli konkrétní

zapojení bloků. Každý blok má nastavený jeden parametr – intenzitu poruch. Ta představuje pravděpodobnost, zda dojde k poškození bloku a tím jeho vyřazení ze systému.

Tuto funkci v aplikaci zajišťuje tlačítko *Parametry*. Po výběru některého z bloků a kliku na tlačítko (lze použít i pravé tlačítko na blok) se otevře okno s nastavení parametru λ.

Aplikace zobrazuje výstupy textově i graficky. Textový výstup obsahuje dva základní výpočty: na prvním řádku je intenzita poruch ($R(t)$) celého modelu. Na druhém řádku je střední doba bezporuchového provozu (T_s). Grafický výstup podporuje pouze zobrazení závislosti intenzity poruch v čase t. Výchozí nastavení t je 1000, je ale možné ho po kliknutí na okno s grafem libovolně upravit.



Obrázek 3.2 Textový výstup - Výsledky sestaveného blokového modelu. $R(t)$ značí intenzitu poruch a T_s představuje střední dobu bezporuchovosti.

Dokumentační část

Aplikace Spolehlivost je zaměřena spíše na tvorbu menších modelů a pouze základních funkcí. Jedná se o školní projekt a autor zřejmě nepočítal s použitím pro modelování složitých modelů. Dokumentace je pojata formou klasického okna s textovým polem a slouží spíše jako poznámky a jednoduchý popis. Popis funguje pouze jako součást aplikace bez možnosti exportu do externích formátů.

Podpora práce v týmu

Jednoduchost aplikace neumožňuje vhodnou práci v týmu. Spustit ji lze pouze na lokálním stroji, veškerá práce musí být manuálně uložena a načtena do/ze souboru. Výstupní formát je binární, což pravděpodobně vede k menší velikosti souborů. Nevýhodou tohoto formátu je verzování. Binární formát nelze slučovat s jinými verzemi, proto soubor slouží pouze jako záloha předchozí práce.

Další zajímavé funkce

V této části jsou shrnuty dodatečné funkce, které mohou zlepšit práci s aplikací nebo sloužit jako předloha funkcí pro praktickou část.

Při návrhu modelu je vhodné mít nástroj pro porovnání podobných modelů. V aplikaci Spolehlivost je toto zařízeno čtyřmi pozicemi pro modely. Spolu s funkcí pro duplikování modelu je možné porovnat rozdíly mezi podobnými modely, nebo stejnými modely s různými parametry. Na tuto situaci je připraven i grafický výstup, který zobrazuje průchod všech modelů zároveň.

NMR je funkce, která pomocí jednoho bloku v návrhu nasimuluje n stejných bloků v sériovém zapojení. Po výběru bloku zadáme n, které značí počet takových bloků a jejich parametr λ. Při použití NMR může být výsledný model mnohem lépe čitelný.

3.1.2 SHARPE

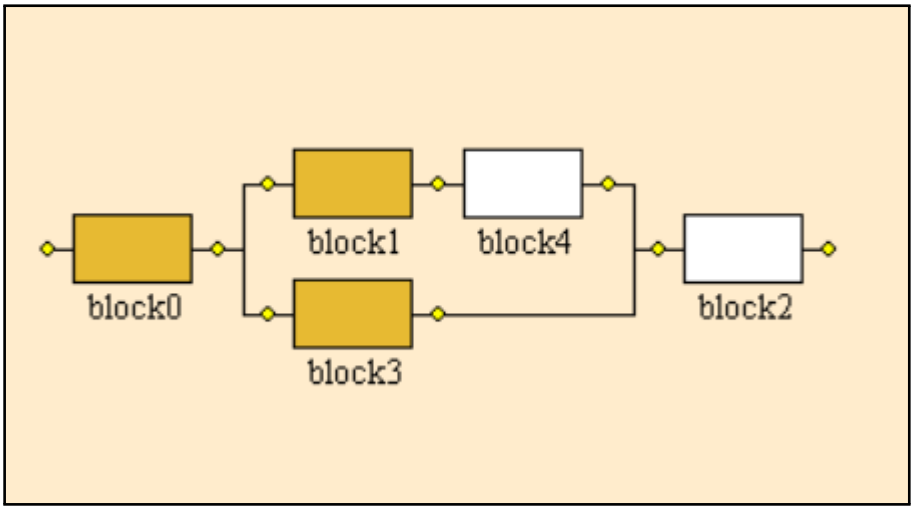
Nyní se dostávám k mnohem propracovanější aplikaci vydávané pod záštitou Duke University v USA. Pro potřeby analýzy byla použita verze 1.3.1 ze září roku 2002. SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) není veřejně distribuovaná a získání kopie je možné pouze za pomoci univerzity nebo pro firemní účely po kontaktu s vydavatelem.

SHARPE je desktopová aplikace vyvíjena v jazyce Java. Vzhledem k datu vydání je patrné, že se stále jedná o aplikaci staršího data, což se samozřejmě projevuje i na uživatelském rozhraní. I přesto je aplikace využívána pro studijní i praktické účely, zřejmě právě pro její širokou škálu typů modelů a dalších funkcí. V následujících sekcích jsou rozebrány ty z nich, které jsou důležité pro tuto práci.

Návrh blokového modelu

Aplikace je koncipována tak, že na každou akci existuje vlastní okno. Mezi hlavní okna se řadí řídící lišta, ta obsahuje základní nástroje pro práci s aktuálním modelem a projektové okno, kde můžeme vidět náhled modelů.

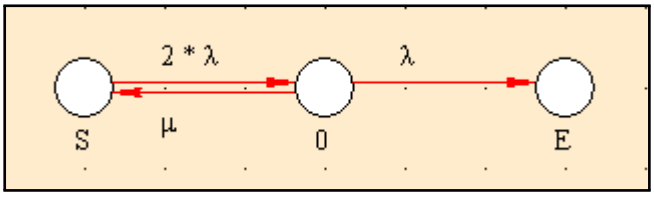
Při práci s RBD modelem se řídící lišta skládá z 10 tlačítek. Podobně jako v programu Spolehlivost se zde nachází nástroje pro přidání nových bloků a kontrolní tlačítka (pro úpravu bloku, mazání, vrácení poslední změny).



Obrázek 3.3 RBD model v SHARPE - Příklad kombinovaného zapojení.

Návrh Markovského modelu

Zvolí-li uživatel návrh Markovského modelu, změní se nástroje v řídící liště. Stav vytvoříme pomocí tlačítka *Node* a přechod v podobě orientované hrany tlačítkem *Arc*. Užitečnou funkcí je *Rate Matrix*, která otevře nové okno s maticí stavů. Zde lze nastavit pravděpodobnost přechodu mezi stavy nebo také vytvořit nový stav. Touto tabulkou je možné nahradit většinu interaktivní tvorby modelu. Dále můžeme využít pomocné nástroje jako je zalomení přechodu nebo posun stavu.



Obrázek 3.4 Markovský model v SHARPE - Příklad modelu pro 2 součástky s možností opravy. Intenzita přechodu pro selhání jedné součástky je rovna λ a intenzita opravy jedné součástky je rovna μ.

Dokumentační část

Při tvorbě složitějších RBD modelů může být dosaženo několika stovek nebo i více propojených bloků. Bez dalších textů může být velmi složité se v takovém modelu vyznat. Tuto situaci umí SHARPE zpřehlednit pomocí vnoření modelu, ale ani to ovšem nemusí v některých situacích příliš pomoci. Navíc je důležité, evidovat další informace o jednotlivých blocích. Proto SHARPE kromě názvu umožní ke každému bloku přidat textový popis. Ten není nijak formátován, jde o typické textové pole. Tímto dokumentace modelu končí, SHARPE neumožňuje vygenerovat žádný souhrn všech popisků nebo jinou formu dokumentace.

Podpora práce v týmu

SHARPE je desktopová aplikace spustitelná na lokálním stroji. Neumožňuje uložení dat na cloud ani online synchronizaci s ostatními instancemi aplikace, proto podpora pro práci v týmu spočívá pouze v přenosu souborů.

SHARPE definuje formát souboru zvlášť pro každý typ modelu (.rbd, .mkv), navíc definuje separátní formát pro informace o projektu (.rgl). Soubory nejsou jako v případě programu Spolehlivost binární, nýbrž textové. Jednotlivé části modelu jako jsou bloky nebo hrany jsou přehledně odděleny a uloženy se všemi metadaty. Tento způsob ukládání souborů má zpravidla horší poměr uložených informací ku velikosti souboru, ovšem také má řadu výhod. Soubor se dá jednoduše analyzovat i bez použití SHARPE, což může být v určitých situacích výhodné. Po širší analýze může být do ostatních aplikací integrována podpora pro načtení nebo uložení ve stejném formátu. Takovýto transparentní formát může vést ke kompatibilitě s ostatními aplikacemi, ať už se jedná o aplikace stejného druhu (tvorba modelu) nebo aplikace pro další výpočty.

3.1.3 Reliability analytics toolkit

V tomto případě se jedná o webový nástroj původně vytvořen v roce 2010 Reliability Analytics Corporation. Web se skládá z 35 nezávislých nástrojů zabývajících se výpočty a vizualizacemi souvisejících s problematikou spolehlivosti. Patří do druhé skupiny analyzovaných aplikací (souvisí s touto prací, ale nejedná se o podobnou aplikaci). Není to klasická aplikace, která umožní navrhnout model a vypočítat výsledky. I přesto zde ale nalezneme nástroje, které s touto prací souvisí.

Popis nástrojů

V této části budou popsány ty nástroje, které přímo souvisí s prací a jejich analýza může být využita v praktické části práce.

První z nich slouží k výpočtu bezporuchovosti v závislosti na n paralelně zapojených blocích, kde m z nich musí být funkčních pro správnou funkci celého systému [3]. Nástroj nepočítá s opravou jednotlivých bloků. To úzce souvisí s RBD spolehlivostním modelem. Vezmeme-li RBD model a zapojíme n paralelních bloků je výpočet stejný jako když do nástroje zadáme $m = 1$.

Dále se nastaví intenzita poruch. V tomto výpočtu se počítá se stejnou intenzitou poruchovosti pro všech n bloků. Intenzita je zadávána v jednotkách „počet chyb za milion hodin“ (FPMH). Po stisku *Calculate* stránka přesměruje na výsledek. Nástroj vrátí dvě vypočítané hodnoty – celkovou intenzitu poruch zapojení v FPMH a střední dobu mezi poruchami v hodinách.

Intenzita poruch dvou paralelně zapojených bloků s odlišnými intenzitami poruch [4] přijímá dva parametry. λ_A představuje intenzitu poruch bloku A, to stejné pro λ_B. Tento výpočet nepočítá s opravou a odpovídá standardní závislosti dvou paralelních bloků RBD spolehlivostního modelu. Za pomoci tohoto výpočtu je možné díky skládání zapojení spočítat intenzitu poruch libovolného paralelního zapojení. Výsledek je znovu vrácen ve dvou hodnotách – intenzita poruch a střední hodnota bezporuchovosti celého zapojení.

Dosavadní uvedené nástroje slouží jako alternativa ke klasickým aplikacím, následující lze využít společně s nimi. Dle normy predikce spolehlivosti součástek MIL-HDBK-217F(N2) dokáže odhadnout intenzitu poruch, která je následně zadávána jako jeden ze vstupů RBD spolehlivostního modelu. [5] Nástroj vytvoří odhad dle zadaného typu součástky, její kvality, prostředí a počtu těchto součástek. Výsledné hodnoty představují intenzitu poruch (v FPMH) a střední dobu bezporuchovosti (v hodinách). Počet různých součástek k výběru činí asi 200.

3.2 Požadavky projektu

V této kapitole jsou shrnuté požadavky vyplývající ze zadání a následných konzultací s vedoucím práce. Jedná se o rozšíření cílů aplikace na podrobnější popis funkcí a rozdělení pro konkrétní analýzu v následujících kapitolách.

3.2.1 Funkční požadavky

F1 - Registrace nových uživatelů
Uživatelé se zaregistrují pomocí svojí e-mailové adresy, aplikace jim vytvoří účet, pomocí kterého mohou dále s aplikací pracovat.

F2 - Vytvoření a správa projektu
Každý uživatel má možnost vytvořit vlastní projekt a dále ho spravovat. Správou projektu je myšleno nastavení přístupu dalším uživatelům a změna informací o projektu.

F3 - Vytvoření nového modelu
Uživatel vytvoří jako součást již existujícího projektu nový model. Během vytváření modelu uživatel zvolí jeho typ a název.

F4 - Modelování schématu
Po vytvoření modelu má každý uživatel s oprávněním alespoň pro úpravy (nebo vyšší) možnost sestavit model pomocí grafického uživatelského prostředí. Dále je možné nastavovat parametry nebo popis modelu.

F5 - Generování výpočtů do Wolfram Notebook
Po sestavení modelu a nastavení všech povinných parametrů je možné vygenerovat Wolfram Mathematica Notebook se základními výpočty a grafy. Před generováním je možné nastavit základní nastavení obsahu.

F6 - Vytvoření nové dokumentace
Mimo modely umožňuje aplikace vytvořit textový dokument, který slouží k dokumentování

celého projektu či jednotlivých modelů. Tento dokument je možné upravovat přímo v aplikaci a následně exportovat do formátu pro Microsoft Word.

F7 - Generování dokumentace dle šablon
Dokumentaci není nutné psát ručně, ale lze vygenerovat její základní podobu. Obsah dokumentace je sestaven z vytvořených modelů v rámci projektu. Vzhled i obsah lze nastavit pomocí výběru jedné z připravených šablon.

F8 - Synchronizace dat při práci s modelem a dokumentací
Při modelování či psaní dokumentace umožňuje aplikace interagovat s ostatními uživateli. Více uživatelů může tedy upravovat modely v jednu chvíli a všechny změny se automaticky projeví i u ostatních uživatelů.

3.2.2 Nefunkční požadavky

N1 - Výkon pro desítky modelů s až stovkami objektů
V aplikaci může každý z uživatelů vytvořit libovolný počet projektů, které se skládají z několika modelů a dokumentací. Aplikace musí být schopna zajistit vhodné podmínky pro práci s modely pro nejméně desítky uživatelů zároveň.

N2 - Vhodný přístup pro implementaci nových modelů, šablon a dalších funkcí
Aplikace má potenciál pro práci s mnoha typy modelů, proto je důležité zajistit vhodnou implementaci dalších typů spolehlivostních modelů. To stejné platí pro šablony dokumentace nebo výpočty spolehlivostních parametrů.

3.2.3 Případy užití

UC1 - Registrace uživatele
1. Uživatel začíná na úvodní stránce webu. Ta se skládá z formuláře pro přihlášení uživatele a odkazu na registraci nového uživatele. Uživatel přejde na registraci.

2. Na stránce s registrací je zobrazen jednoduchý formulář se základními prvky. Obsahuje pole pro email a heslo spolu s potvrzovacím tlačítkem. Po vyplnění a potvrzení registrace je formulář validován. Pokud účet s emailem již existuje nebo heslo nesplňuje podmínky, je uživatel upozorněn na chybu ve formuláři.

3. Po úspěšné kontrole údajů je účet vytvořen se základní rolí pro uživatele a přesměrován na základní stránku v přihlášeném režimu (seznam projektů).

UC2 - Vytvoření nového projektu a nastavení přístupů
1. Tento případ užití začíná na seznamu projektů. V tomto seznamu jsou zobrazeny všechny projekty, ke kterým má přihlášeny uživatel přístup. Každý uživatel, bez ohledu na roli, má možnost vytvořit vlastní projekt.

2. Po kliku na ikonu pro přidání projektu web otevře dialog, který obsahuje příslušný formulář s názvem projektu a potvrzením pro jeho vytvoření. *Název* je povinné pole a musí být vyplněn. Pokud není, web zobrazí chybovou hlášku a čeká na opětovné odeslání. Po správném vyplnění je projekt vytvořen a uživatel přesměrován na detail.

3. V detailu projektu je možnost přiřadit nové uživatele k projektu a nastavit jim oprávnění. Tato oprávnění jsou trojího typu, *Admin* (administrace projektu), *Edit* (úprava projektu) a *View*

(prohlížení projektu). Přidané uživatele lze zpětně odstranit. Samotný vlastník projektu má automaticky oprávnění *Admin*. K této části projektu mají přístup pouze jeho administrátoři a vlastník.

UC3 - Sestavení blokového modelu

1. V seznamu projektu zvolí přihlášený uživatel daný projekt. Pokud nemá k žádnému přístup, může vytvořit vlastní (viz UC2) a poté přejde na detail projektu. Nový model lze vytvořit pouze s oprávněním pro úpravu projektu. Pokud nemá uživatel dostatečné oprávnění, může požádat vlastníka či administrátora projektu o jeho zvýšení.

2. V seznamu modelů stiskne uživatel ikonu pro vytvoření nového modelu a otevře se dialog s formulářem. Ten je sestaven z názvu modelu a jeho typu. Pro tento případ užití vybereme *Block model*. Poté je formulář zkontrolován a v případě chyby je uživatel požádán o jeho úpravu.

3. Po úspěšném vytvoření modelu je uživatel přesměrován na detail modelu. Sestavení blokového modelu probíhá pomocí nabídky bloků. Bloky se přidají do modelu a následně propojí hranami. Každý blok, hrana a parametr mají vlastní nastavení v nabídce. Parametru lze nastavit popisek a hodnotu. Blok má nastavitelnou barvu, parametr a název. Posledním objektem je hrana se svým názvem.

4. Po sestavení modelu je možné vygenerovat základní výpočty do notebooku pro Wolfram Mathematica. Před exportem je model zkontrolován – bloky musí být správně spojeny, model nesmí obsahovat orientované cykly a každý blok musí mít nějakou intenzitu poruch. Pokud je v modelu chyba, aplikace zobrazí chybovou hlášku. Pokud je naopak model v pořádku, aplikace spustí stahování vygenerovaného notebooku.

5. Notebook obsahuje základní výpočty. Prvním z nich je výpočet pravděpodobnosti bezporuchového provozu $R(t)$ i s postupem. Pokud to uživatel povolil při nastavení, v notebooku bude i graf projekce $R(t)$ v čase. Postup výpočtu je postupně rozepsán, aby byla umožněna jednoduchá úprava hodnot v rámci notebooku.

UC4 - Sestavení Markovského modelu

1. Samotné vytvoření Markovského modelu funguje podobně jako v UC3. Pro jeho vytvoření je nutné oprávnění pro úpravu projektu.

2. Když v seznamu modelů stiskne uživatel ikonu pro vytvoření nového modelu a otevře se dialog s formulářem. Ten je sestaven z názvu modelu a jeho typu. Pro tento případ užití vybereme *Markov model*. Poté je formulář zkontrolován a v případě chyby je uživatel požádán o jeho úpravu.

3. Po úspěšném vytvoření modelu je uživatel přesměrován na detail modelu. Stavy Markovského modelu jsou opět v seznamu prvků. V tomto případě se jedná o jediný typ bloku, který je možné použít. Dalším objektem jsou hrany, kterým je nutné nastavit intenzitu přechodu. Dostupné intenzity přechodu můžeme přidávat, mazat a upravovat v samostatné nabídce. Stavu můžeme nastavit název a barvu zobrazení.

UC5 - Duplikace projektu

1. Nový model nemusí vzniknout pouze přidáním nového prázdného modelu, ale také duplikací již existujícího. Po přechodu na detail libovolného modelu může uživatel s oprávněním pro alespoň úpravu projektu vytvořit nový model jeho duplikací. Výslednému modelu je nastaven nový název a všechny ostatní vlastnosti (včetně bloků, parametrů i popisu) jsou zkopírovány.

UC6 - Vygenerování dokumentace

1. V seznamu projektu zvolí přihlášený uživatel projekt, jehož dokumentaci chce vygenerovat. Podmínkou je oprávnění pro úpravu vybraného projektu.

2. Uživatel je přesměrován na stránku s úpravou dokumentace. Zde je možné libovolně zapisovat do dokumentu s využitím základních textových nástrojů (např. tučný text, kurzíva, vložení obrázku, ...). V nabídce funkcí je na výběr export dokumentace a vygenerování obsahu.

3. Volba generace zobrazí výběr šablony. Šablona udává, jaký obsah a v jakém rozložení bude vygenerován. Obecně platí, že generovaná dokumentace má jako předlohu vytvořené modely, jejich parametry a popis.

4. Takto vygenerovanou (i ručně vytvořenou) dokumentaci je možné pomocí nástroje pro export uložit formou dokumentu pro Microsoft Word nebo ve formátu PDF.

3.3 Technologie

Analýza technologií je věnována jednotlivým částem celé aplikace a popisuje technologie, které v nich lze využít. Výsledek této podkapitoly vede k výběru vhodných nástrojů a přípravě pro návrh aplikace. Aplikace pro spolehlivostní modely není jednoduchým webem, proto se vyplatí při analýze počítat s robustní klientskou aplikací. Analýzu proto rozděluji na server a klient.

3.3.1 Server

Stěžejní serverové technologie pro analýzu jsou: databáze, programovací jazyk a webový framework. Tyto tři části mají přímý vliv na výkon i stavbu aplikace. V mnoha případech mohou ovlivnit i obecný návrh, proto je vhodné je zde popsat.

3.3.1.1 Programovací jazyk

PHP (PHP: Hypertext Preprocessor) je široce využívaný open source skriptovací jazyk využívaný především pro vývoj webových aplikací. [6] Původně by vytvořen v roce 1994 Rasmusem Lerdorfem v programovacím jazyce C a sloužil pouze jako souhrn podpůrných skriptů na webové stránce. Později byl stejným autorem přepsán a rozšířen o komunikaci s databází a nástroje pro vývoj dynamických webových stránek. První oficiálně vydanou verzí bylo PHP 3.0 v roce 1998, tentokrát již týmem několika vývojářů (PHP Development Team). Do dnešní podoby prošlo několika dalšími verzemi – aktuální nese označení 8.1. Dle webu *w3techs.com*[7], který pravidelně vydává statistiky týkající se webových technologií, je k březnu roku 2022 PHP používáno asi na 77% všech webových stránek.

Vzhledem k mému rozdílu ve zkušenostech mezi jazykem PHP a ostatními jazyky využívanými pro webové aplikace jsem se rozhodl použít pro server právě jazyk PHP. V době, kdy jsem začal na projektu pracovat, byla oficiální verze 8.0, proto je pro vývoj použita tato verze.

3.3.1.2 Framework

PHP je nejpoužívanější jazyk pro webové aplikace. Mnoho vývojářů s ním umí pracovat a opakované části vývoje jsou již mnohokrát vyřešeny. Správná volba frameworku nejen usnadní a urychlí vývoj stále opakovaných funkcionalit, ale díky širokému využití (testování) při správném použití zvýší bezpečnost celé aplikace.

Mezi často používané frameworky patří například Laravel ([8]), Symfony ([9]) nebo Nette ([10]). Pro aplikaci tohoto typu není jeden nejvhodnější framework – všechny tři jmenované jsou pro projekt tohoto typu připraveny. Osobně mám největší zkušenosti se Symfony, proto jsem ho zvolil také pro tuto aplikaci.

Symfony

Framework Symfony je rozšířený PHP open-source framework s rozsáhlou komunitou vývojářů. Je vyvíjen společnostmi a komunitou přesahující 600 tisíc vývojářů z několika zemí. Vzhledem k tomu je schopen reagovat na nové tendence v rámci webových aplikací. Docílí toho pomocí přídavných modulů, které může pro Symfony vytvořit jakýkoliv vývojář.[11]

Nejefektivnější způsob použití Symfony je strukturování aplikace podle vzoru MVC (Model View Controller) architektury. Tento vzor pomáhá vývojáři rozdělit webovou aplikaci na menší části, podle jejich zodpovědností. Tyto části jsou tři. První z nich je model, který se stará o práci s daty, definování jejich struktury nebo ukládání do databáze. View poskytuje vyhodnocená data uživateli a Controller zpracuje akce vyvolané uživatelem.[11]

Rozdělení Symfony do mnoha komponent nevyžaduje po vývojáři použití všech funkcí frameworku, ale pouze těch, které opravdu použije. I přesto lze v Symfony vytvořit hodně komplexní. Pokud je cílem pouze jednoduchá aplikace, může být projekt mnohem menší a tím i přehlednější a rychlejší pro vývoj.[11]

Laravel

Laravel je multiplatformní PHP framework pro tvorbu webových aplikací. Byl navržen jako nástroj pro zjednodušení běžných částí vývoje například routování, autentizace, migrace a další. Umožňuje jednoduchou integraci připravených modulů do aplikace pomocí intuitivního rozhraní pro příkazovou řádku. Pro Laravel existuje rozsáhlá online dokumentace pro začínající i pokročilé vývojáře. [12]

Aplikace vyvíjené v Laravelu jsou vysoce škálovatelné a velmi dobře udržovatelné. S nástrojem pro správu knihoven a modulů lze přidávat nebo aktualizovat širokou škálu rozšíření základního frameworku. Pro rozdělení odpovědností mezi části aplikace používá Laravel podobně jako Symfony MVC architekturu. Pro definici struktur dat je většinou využíváno objektově relační modelování (ORM). To dokáže převádět entity mezi PHP a SQL databází.[12]

Nette

Nette stojí za zmínění mimo jiné z toho důvodu, že se jedná o framework spravovaný českým týmem vývojářů a také využívá MVP architekturu, kdežto předchozí frameworky využívají architekturu MVC. Nette je také open-source a každý vývojář se muže podílet na jeho vývoji. Stavěný je tak, aby byl co nejpoužitelnější a nejvstřícnější. Poskytuje srozumitelnou a úspornou syntaxi a vychází vstříc při programování i debugování.

MVP (Model-View-Presenter) je architektura, která se na prví pohled podobá architektuře MVC, *controller* vrsta tady je však nahrazena vrstvou *presenter*. Myšlenka této architektury zní „nechť je odkaz totéž, co zavolání funkce“. Presenter tedy zpracovává akce od uživatele, váže je na dané metody a aktualizuje model i view.

Součástí Nette je také Latte, což je šablonovací systém nejen pro HTML šablony. I přesto je Nette v dnešní době méně využívaný, než Symfony nebo Laravel. Jedním z důvodů může být i méně podrobná dokumentace.

3.3.1.3 Databáze

Cílová aplikace je podle zadání a požadavků webová aplikace, která má data uložená na serveru. Server proto musí obsahovat databázi, se kterou bude back-end komunikovat. Čtyři z pěti nejpoužívanějších databází jsou relační (SQL)[13]. Ty jsou pro aplikace tohoto typu vhodnou volbou. Jednou z výhod je podpora ze strany téměř všech frameworků pro PHP. Analyzovat zde budu tři z těchto databázových strojů.

PostgreSQL

PostgreSQL je silný open-source objektově-raleční databázový systém využívající jazyk SQL. K základní podobě SQL přidává rozšiřující funkce pro bezpečné a škálovatelné uložení strukturovaných dat. Rozšíření jsou koncipována tak, aby pomáhala při vývoj a administraci dat libovolného rozsahu.[14]

Vzhledem k tomu, že je PostgreSQL volně dostupný a open-source, je také vysoce rozšiřitelný. Vyvojář si může zadefinovat vlastní datové typy, vytvořit vlastní funkce nebo pracovat s různými programovacími jazyky. Ke všem těmto možnostem je poskytnuta rozsáhlá online dokumentace ke každé verzi databázového systému. Hlavními výhodami je volná dostupnost a široká komunita aktivně vyvíjející nové funkce a verze databázového systému.[14]

Architektura PostgreSQL je rozdělena na clustery, ty dále obsahují uživatele, databáze a tabulkové prostory (*tablespace*). Toto umožňuje jednomu uživateli pracovat s více databázemi v rámci jednoho uživatelského účtu, což tvoří výhodu například oproti systému Oracle.

MySQL

MySQL je jeden z nejpopulárnějších databázových strojů. Je široce a efektivně používán napříč všemi typy aplikací. Jedná se o open-source relační databázový systém vyvinut a podporován společností Oracle, který je založen na jazyce SQL.[15] MySQL je rychlým, spolehlivým a jednoduchým řešením jako databáze webových aplikací. Po nasazení nevyžaduje databáze téměř žádnou údržbu. V kontextu požadavků aplikace je MySQL databáze plně schopna naplnit všechny požadavky projektu. Výhodou jeho využití může být, díky jeho popularitě, skutečnost, že všechny analyzované frameworky mají plnou podporu pro práci s MySQL.

3.3.2 Klient

Běžné webové stránky fungují většinou na principu tenkého klienta. To znamená, že na stroji koncového uživatele není žádná složitější logika aplikace, ale pouze vykreslení stránky získané jedním HTTP požadavkem. Tento postup je výhodný ve své jednoduchosti implementace.

Cílová aplikace je zaměřena na uživatelskou přívětivost a měla by být pro uživatele plynulá bez zbytečného načítání stránek při každé akci. Rozhodl jsem se tedy implementovat robustní klientskou aplikaci, ve které lze toto chování eliminovat.

Pro tyto účely existují JavaScript frameworky, které jsou k tomuto požití určeny. Zde jsou uvedeny dva z nich, React[16] a Vue.js[17].

React

React je open-source knihovna v jazyce JavaScript vyvíjena společností Facebook a komunitou vývojářů. Její hlavní případ užití je pro vývoj webových aplikací, kde je jednou z nejpoužívanějších front-end knihoven.

Pomocí knihovny React lze jednoduše vyvíjet dynamické aplikace, jelikož již obsahuje mnoho funkcí, které lze jednoduše použít. Oproti samotnému jazyku JavaScript, kde je zdrojovým kód brzy komplexní a nepřehledný. Výhodnou vlastností je použití komponent, kterými lze zdrojový kód rozdělit a každé přidělit zodpovědnost za malou část aplikace. Tyto komponenty jsou snadno udržovatelné a testovatelné a navíc je lze použít opakovaně. Nespornou výhodou je velká škála vývojářských nástrojů, které vznikají hlavně díky popularitě knihovny.

Komponenty využívají syntaktické rozšíření JSX, které se strukturou podobá HTML. Toho je využito při aktualizaci DOM webové stránky. React při akci aktualizuje pouze takové části DOM, kterých se změna týká, tím zrychluje chod aplikace.

Výpis kódu 3.1 Ukázka JSX kódu - Kód obsahuje tlačítko s akcí reagující na klik. Dále seznam se dvěma prvky. Implementace tlačítka, seznamu a jeho prvků je oddělena v samostatných komponentách (často i samostatných souborech).

```
<div>
    <Button
        variant="primary"
        onClick={() => console.log("Adding␣item!")}
    >
        Add item
    </Button>
    <List>
        <Item>A</Item>
        <Item>B</Item>
    </List>
</div>
```

Vue.js

Vue je framework v jazyce JavaScript pro vývoj uživatelského rozhraní.[18] Jedná se o druhý nejpopulárnější front-end framework. Při tvorbě Vue byla hlavní myšlenka vytvoření silného, ale velmi lehkého a flexibilního frameworku, který lze použít bez závislosti na velkém množství podpůrných knihoven. Součástí celého balíčku je také nástroj pro příkazovou řádku, který zjednoduší nastavení.[19]

Vue.js používá podobně jako React architekturu založenou na komponentách, tím rozděluje velké části kódu na malé komponenty. Každá je napsána pouze pomocí HTML, CSS a jazyku JavaScript. Díky tomu je se práci s Vue rychle naučí každý vývojář, který zná alespoň základy jazyku JavaScript. Nevýhodou může být přílišná flexibilita a nekonzistentní udržování kódy mezi vývojáři jednoho týmu nebo také implementační problémy spojené s aktualizací dat mezi komponentami.[19]

3.3.3 Synchronizace

V požadavcích webové aplikace najdeme synchronizaci dat mezi všemi aktivními uživateli. Tato funkce je použita pouze na stránkách s tvorbou modelu a dokumentace. Abych tento termín uvedl do kontextu, později popíšu případ užití této funkce. Na návrhu systému bude často pracovat několik lidí zároveň. V rámci jednoho projektu může každý vytvářet svůj vlastní model, ale pokud mají pracovat na jednom modelu zároveň, nastane problém synchronizace jejich verzí. V podkapitole 3.1 jsou rozebrány dva programy, které slouží k návrhu spolehlivostního modelu – jedná se však o desktopové aplikace, které nevyužívají připojení k internetu a tudíž není možné, aby více lidí v jednu chvíli pracovalo na stejném modelu. Pokud tak činí, musejí svou práci nakonec sloučit do jednoho modelu, což je obecně komplikované a náchylné k chybám.

Cílová aplikace má díky výběru platformy (web) dobré předpoklady k elegantnímu řešení tohoto problému. Existuje již mnoho veřejně dostupných aplikací (např. Google Docs[20], Whimsical[21]), které tuto synchronizaci umožňují. Ve chvíli, kdy mají dva uživatelé otevřen detail modelu a jeden z nich například vytvoří nový stav, klientská aplikace druhého uživatele, získá informaci o přidání nového stavu a automaticky ho přidá. V ideálním případě tato změna probíhá ihned a bez nutnosti vyvolání akce na straně příjemce, různé implementace se ale těmito vlastnostmi liší a proto je postupně popíši.

Periodická aktualizace

Prvním a přímočarým způsobem, jak lze dosáhnout synchronizace klientů je periodická aktualizace. Princip této techniky je jednoduchý. Definujeme nějaké časové okno (například 1 minuta) a spustíme interval. Po jeho uplynutí je spuštěn požadavek pro získání změn a ty se následně provedou.

Tento postup není nutné implementovat – lze použít „Web Periodic Background Synchronization API“[22]. Nicméně pro jeho správnou funkci vyžadují některé prohlížeče oprávnění a některé ji nepodporují vůbec. API poskytuje funkce pro registraci úloh, které jsou periodicky spouštěny. Tyto úlohy se nazývají „periodic background sync requests“. Při registraci se nastaví název úlohy a minimální interval aktualizace. Následně se definuje funkce, která je při každé aktualizaci spuštěna. Nakonec se úloha odhlásí a další aktualizace neprobíhá. [22]

Tento postup je výhodný pro svou jednoduchou implementaci, na druhou stranu má ale řadu nevýhod. Jednou z nich je efektivita požadavků. Pokud zvolíme časové okno příliš velké, bude se často stávat, že nedostaneme aktuální data a bude docházet k nekonzistenci. Nastavíme-li naopak časové okno příliš krátké, aplikace bude posílat zbytečné požadavky a zatěžovat tím síť. Tento způsob aktualizace může být vhodný například při aktualizaci příspěvků na sociální síti. Pro případy užití v této aplikaci, kde jsou v reálném čase synchronizována data všech uživatelů, není dobrým řešením.

WebSocket

Další nástroj se nazývá je *The WebSocket API*. Jedná se o pokročilou technologii, která otevře obousměrné spojení mezi klientem a serverem. Pomocí tohoto API lze kdykoliv během spojení zasílat zprávy jak z prohlížeče na server, tak i opačným směrem. [23]

Implementace na straně serveru spočívá ve vytvoření služby, která poslouchá na daném portu. Aplikaci lze napsat v libovolném programovacím jazyce, který má podporu„Barkeley sockets“ – mezi ně patří například Python, PHP nebo JavaScript. V předchozí části analýzy byl vybrán jazyk PHP a Symfony framework, proto se zaměřím na tuto kombinaci technologií. Konkrétně pro Symfony existuje rozšíření, které podporuje komunikaci pomocí *WebSockets*.

Mercure

Mercure je otevřený protokol pro komunikaci v reálném čase napsaný v programovacím jazyce Go. Dle specifikace na stránkách nástroje Mercure se jedná o mechanismus pro publikování a odebírání dat z veřejných i privátních zdrojů. Umožňuje posílat jakýkoliv webový obsah klientům rychlou a spolehlivou cestou. [24]

Na serveru je spuštěn Mercure Hub, který představuje hlavní bod komunikace – server, přes který veškerá komunikace probíhá. V rámci komunikace definujeme *topic* (téma) zprávy (nejčastěji URI adresu zdroje). Při přihlášení klient požádá o token pro tuto komunikaci, server vygeneruje JWT (JSON Web Token)[25], ve kterém nastaví oprávnění na ten *topic*, ke kterému má přístup. Uživatel se poté zaregistruje pro odběr některého tématu (může i více najednou) a čeká na jejich aktualizaci. Serverová PHP aplikace při zaznamenání změny od jednoho z klientů upozorní Mercure Hub o aktualizaci a ten následně odešle stejnou zprávu všem odebírajícím klientům.

Mercure funguje obdobně jako WebSocket, rozšiřuje ho však o Mercure Hub a standardizaci komunikace. Tvoří tak velmi šikovný nástroj pro tento případ užití. Navíc Symfony, které je již pro aplikaci vybráno, má pro Mercure oficiální podporu. Proto jsem tuto technologii vybral jako způsob synchronizace uživatel v aplikaci pro spolehlivostní modely.

3.3.4 Docker

Přesuňme se od implementace k nasazení. Dosud jsme v analýze našli několik technologií, které musí být správně nasazeny. Mezi služby, které musí být při spuštění aplikace dopředu správně nakonfigurovány jsou PHP server, databáze, Mercure server a instalace nástrojů pro Wolfram Script i Wolfram Mathematica. Prvním způsobem nasazení je vytvořit definované prostředí na lokálním stroji a tam aplikaci vyvíjet. Ve chvíli, kdy dojde k prvnímu nasazení se musí stejným způsobem nakonfigurovat produkční prostředí. Při realizaci tohoto postupu mohou jednoduše nastat chyby, které vedou například k instalací odlišných verzí a tudíž i odlišného chování. Abych těmto chybám předešel, využiji pro nasazení Docker – konkrétně bude na produkčním serveru i na vývojovém lokálním stroji probíhat nasazení a běh aplikace pomocí stejných předem nakonfigurovaných Docker kontejnerů.

Docker je otevřená platforma pro vývoj a nasazení aplikací, která poskytuje systém pro zabalení izolovaného prostředí do kontejneru. Kontejner je formou virtualizován a sestaven dle konfigurace tak, aby nebyl závislý na konkrétním hostujícím stroji. [26]

Kapitola 4

Návrh a implementace

Tato kapitola je věnována návrhu všech částí aplikace. Webové aplikace se typicky rozdělují na serverovou a klientskou část. Tato práce se bude tohoto postupu držet. Server se stará o definici databáze, validaci hodnot a autentizaci. Na druhou stranu klientská část se soustředí na vytvoření uživatelsky přívětivého nástroje pro správu projektů, tvorbu modelů nebo dokumentace. Proto je i návrh těchto dvou částí oddělen. Kapitola vychází z předchozích analýz existujících řešení a dostupných technologií. Získané informace jsou promítnuty do jednotlivých částí kapitoly a vhodně využity tak, aby byl výsledný koncept co nejlepší pro implementaci a následné použití koncovým uživatelem.

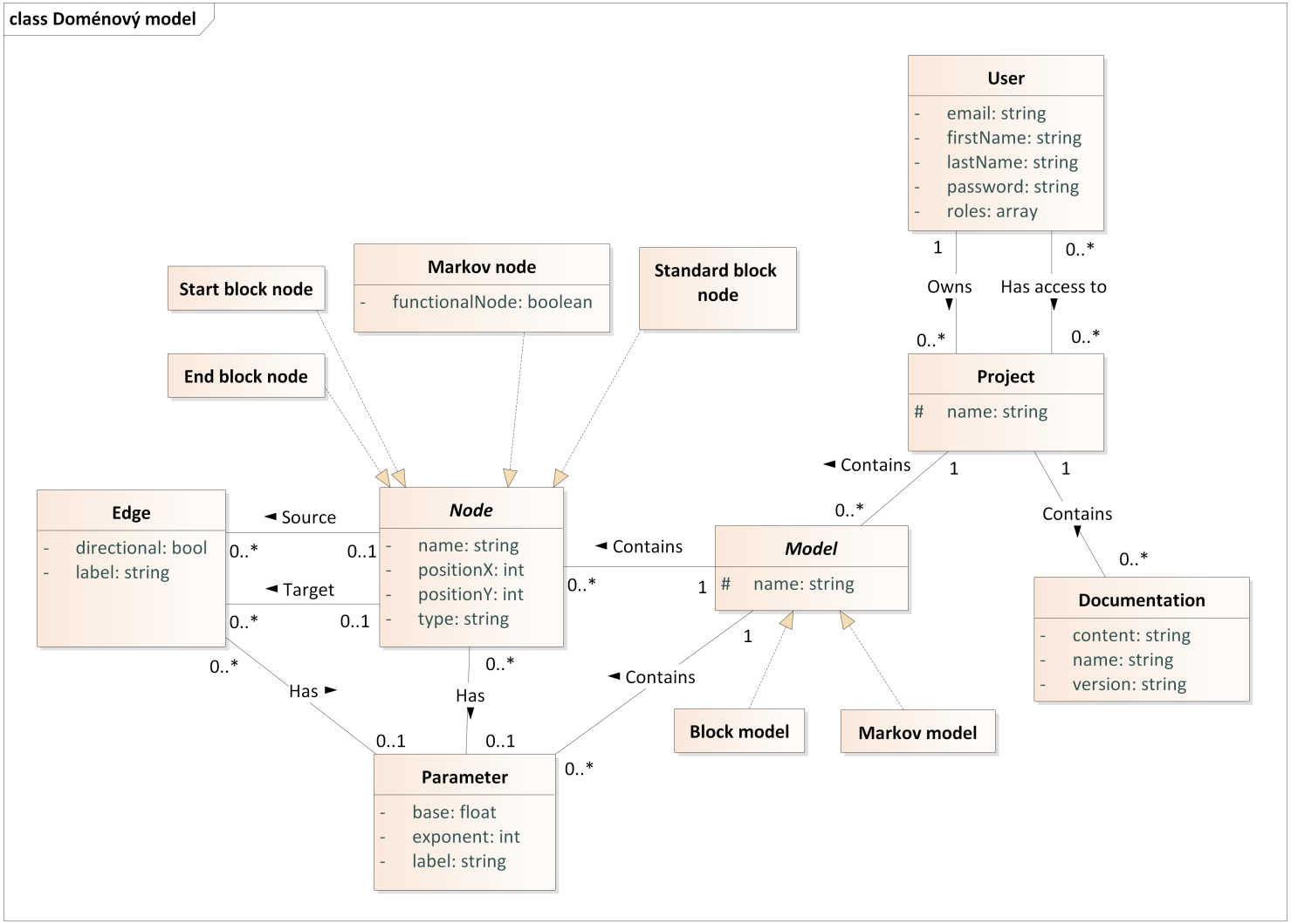
4.1 Serverová část

V kapitole Analýza (3) byly popsány a vybrány technologie, které budu při implementaci používat. Programovací jazyk je PHP s webovým frameworkem Symfony. Jako databáze byla zvolena relační databáze, konkrétně PostgreSQL.

V požadavcích projektu (3.2) jsou zadefinované funkční požadavky a případy užití. V tuto chvíli je vhodné identifikovat entity v rámci doménového modelu, který bude později sloužit jako předloha k definici relační databáze.

4.1.1 Doménový model

Doménový model je sestaven ze sedmi základních entit a dvou specializací entity Model. Výsledek vychází z analýzy funkčních požadavků a případů užití.



Obrázek 4.1 Doménový model - Export doménového modelu z programu Enterprise Architect[27].

User

User je entita, která představuje jednoho uživatele aplikace. Pro udržení jednoduchosti není potřeba mít více informací než jméno, příjmení a přihlašovací údaje. To zajistí jednoduchou a rychlou registraci uživatelů. Dále jsou doplněny role, které slouží pro odlišení administrátora a běžného uživatele. Libovolný uživatel může vytvořit a poté vlastnit projekty. K práci ale není potřeba vytvořit nový projekt, stačí být přidán k nějakému již existujícímu.

Projekt - Model - Dokumentace

Projekt slouží jako zastřešující entita nad všemi logicky souvisejícími náležitostmi (model, dokumentace). Každý projekt má právě jednoho vlastníka, který automaticky drží maximální práva na správu projektu. K němu poté může přiřadit další uživatele, kterým nastaví jeden ze tří stupňů oprávnění.

Jednou ze součástí projektu je model. Aplikace umí pracovat se dvěma typy modelů, blokový (RBD) a Markovský. Aby bylo zajištěno snadné přidání nových typů mají tyto entity generalizace vztah na abstraktní entitu Model. Tyto entity reprezentují základní informace o spolehlivostním modelu a asociace na další entity, které reprezentují jejich obsah.

Dokumentace, podobně jako model, spadá pod projekt a obsahuje název, verzi i obsah dokumentu. Ten je tvořen HTML schématem, které zajistí zobrazení na webové stránce tak, aby působilo jako fulltextový formát. Slouží k popisu projektu nebo jeho části. Díky přístupu k informacím o modelu a programově připravených šablonách je možné základní dokumentaci automaticky vygenerovat. Každý projekt může mít libovolný počet dokumentací.

Node - Edge - Parameter

Zbývající větev doménového modelu zobrazuje všechny objekty, které společně tvoří obsah každého modelu. Node je entita reprezentující jeden blok/stav ve schématu – vždy je pojmenován a umístěn na určitou pozici v modelu. Atribut typ udává, jakým způsobem má být blok zobrazen v klientské aplikaci. Například v RBD modelu mají zvláštní význam bloky, které uvádí a ukončují model, proto by měly být od ostatních bloků vzhledově odlišeny. Toto rozdělení může hrát roli i při výpočtu, jelikož úvodní a ukončující blok slouží pouze jako označení začátku a konce modelu a nemají tedy žádné číselné parametry.

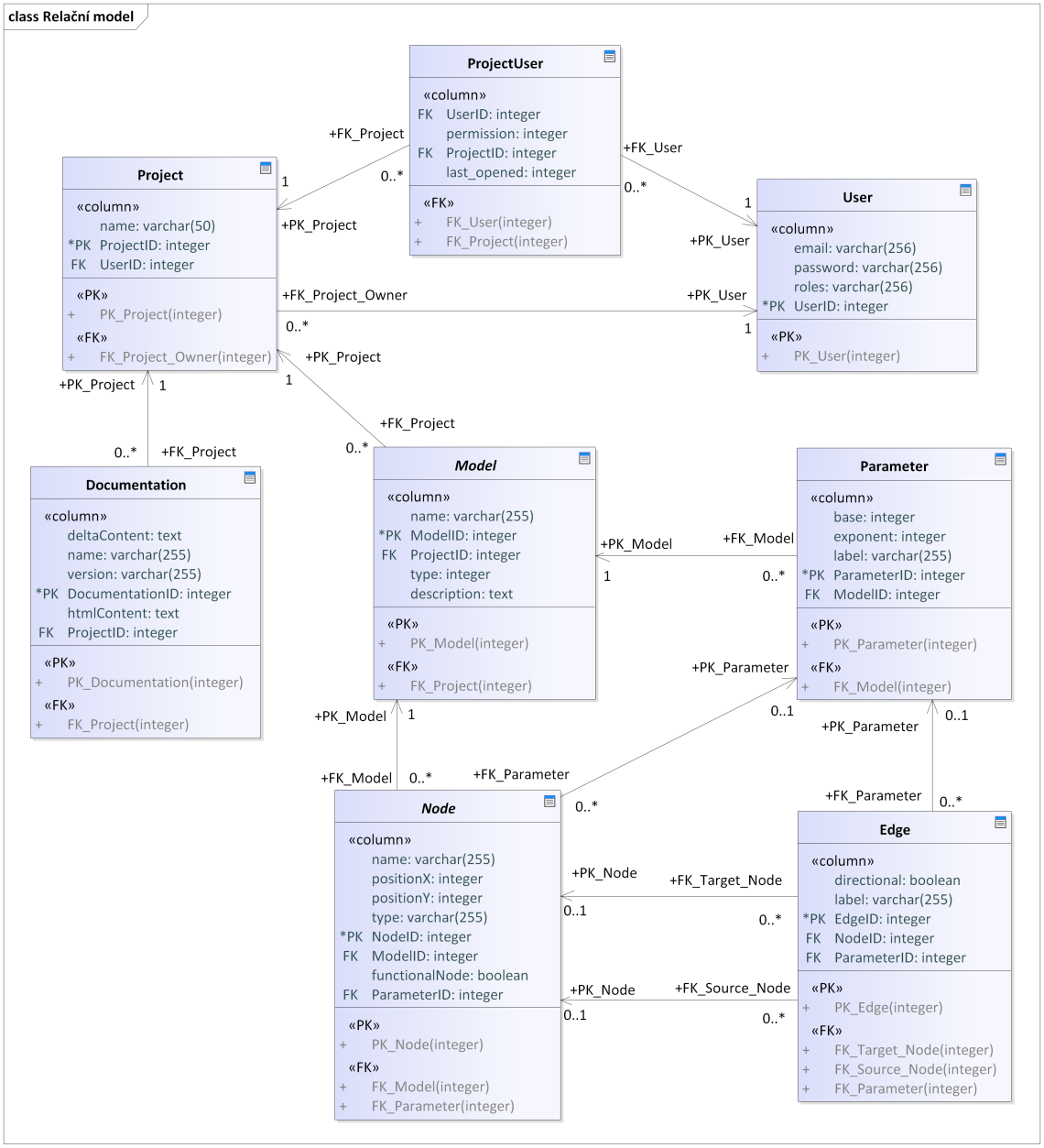
Entita Edge představuje hranu mezi dvěma bloky. Hrana vždy vede ze zdrojového do cílového bloku. Atributy tvoří popisek a informace o tom, zda se má hrana zobrazit jako orientovaná. Směr hrany je vždy ze zdroje.

Entita Parameter značí číselné hodnoty používané při výpočtech. Každý parametr je definován na úrovni modelu. Pro každý model tedy existuje seznam parametrů, které mohou být přiřazeny k některým objektům stejného modelu. Pro nynější typy modelů jsou parametry přiřazeny různým objektům. V případě blokového modelu musí mít pro výsledek výpočtu každý blok přiřazen právě jeden parametr. V případě Markovského modelu jsou parametry přiřazeny ke každé hraně. Parametr funguje jako definice – po zadání názvu a hodnoty může být použit na několika místech v modelu. V případě obou typů modelů jsou parametry typicky velmi nízká čísla. Z důvodu jednoduššího zápisu je ukládán ve tvaru *base* a *exponent*, kde je výsledná hodnota rovna $base \cdot 10^{exponent}$.

4.1.2 Relační model

Relační model reprezentuje finální podobu relační databáze, její tabulky a vztahy mezi nimi. Model standardně vychází z doménového modelu a vztahy mezi entitami upravuje tak, aby byly vhodné pro příslušnou databázi. Zde jsem zvolil databázi PostgreSQL, což má vliv na výsledný model

hlavně u datových typů atributů. V této podkapitole jsou shrnuty odlišnosti oproti předchozímu doménovému modelu.



Obrázek 4.2 Relační model - Model relační databáze vytvořený v programu Enterprise Architect[27].

User - Project

Tabulka s uživateli má dvě vazby na projekty, první je přímá, kde každý projekt obsahuje cizí klíč uživatele, který je vlastníkem projektu. Projekty však nejsou veřejné a uživatelé k nim musí být přiřazeni s některou úrovní oprávnění. K tomu slouží druhá vazba přes tabulku *ProjectUser*.

Jedná se o vazební tabulku mezi uživatelem a projektem, která mimo pouhého spojení dvou entit představuje také úroveň oprávnění a časové razítko posledního přístupu.

Model

Tabulka *Model* reprezentuje spolehlivostní modely všech typů. V doménovém modelu jsou typy rozděleny pomocí generalizace. To musí být ve fyzické podobě databáze nahrazeno nějakým vzorem. Pro tento případ jsem využil vzor jedné tabulky (Single table). Výsledkem je jedna společná tabulka (*Model*), která obsahuje všechny sloupce všech typů. Pro každý řádek následně platí, že jsou vyplněny právě ty sloupce, které daný typ využívá (ostatní jsou nastaveny na *null*). Rozpoznání typu konkrétního řádku tabulky je řízeno sloupcem *type*. Do něj lze zapsat číselnou hodnotu, která identifikuje daný typ modelu.

Node

U této tabulky nastává obdobná situace jako v případě modelu. V aplikaci jsou nyní přítomny čtyři typy bloků/stavů pro různé modely a všechny jsou zapisovány do jedné tabulky s názvem *Node*. Rozdílem je, že datovým typem sloupce *type* – rozlišující blok/stav – je textová hodnota datového typu *string*.

4.2 Architektura aplikace

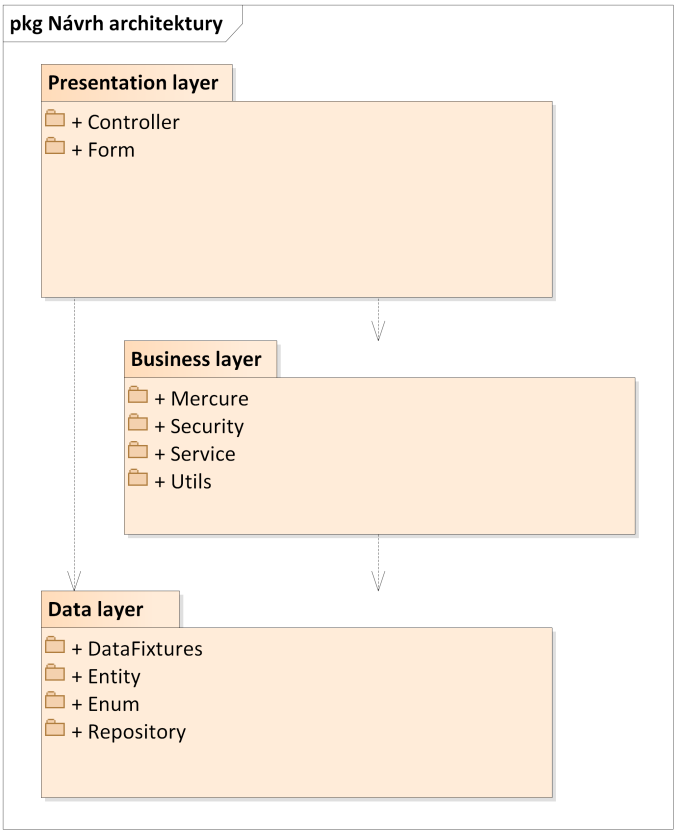
Serverová část aplikace je často rozdělována do více vrstev. Tyto vrstvy oddělí logické části aplikace a umožní mi pracovat s některými z nich nezávisle na ostatních. Vzhledem k vlastnostem backend části aplikace jsem se rozhodl použít architekturu třívrstvou. Skládá se z prezentační, business a datové vrstvy. Každá z nich představuje jiný logický segment aplikace. Základem je datová vrstva, ta se obecně stará o definování struktury dat (entity) a operace nad databází. Business vrstva obsahuje veškeré logické prvky. Například to, jak se pomocí blokového modelu vypočte jeho pravděpodobnost bezporuchového stavu. Poslední vrstva je prezentační – vyřizuje komunikaci směrem k uživateli a zpracovává jeho akce. V této aplikaci se typicky jedná o zpracování komunikace v REST API. Pokud budu chtít v průběhu implementace poskytovat i jiný druh komunikace s uživatelem než je REST API (například klasické HTML stránky), stačí jednoduše vyměnit pouze prezentační vrstvu, datové a business vrstvy se taková úprava nemusí týkat.

Na začátku podkapitoly jsem uvedl, že se jedná o třívrstvou architekturu, ta obecně obsahuje pouze závislost prezentační vrstvy na business vrstvě a business vrstvy na datové vrstvě. REST API však obsahuje některé zdroje, ve kterých se pouze vrací samostatná data bez další akce. Prezentační vrstva by se tedy musela dotázat do business vrstvy a ta následně do datové s tím, že přechod přes business vrstvu by neměl žádnou přidanou hodnotu. Z tohoto důvodu využívám přímočarý typ třívrstvé architektury, který se nazývá relaxovaná třívrstvá architektura. V tomto případě je možné z prezentační vrstvy rovnou zavolat metodu v datové vrstvě a vrátit data uživateli. Výsledný kód se tím zjednoduší a zpřehlední.

Na obrázku 4.3 jsou jednotlivé vrstvy propojeny orientovanými hranami, které značí závislost jedné vrstvy na druhé (například business vrstva je závislá na datové vrstvě). Je tomu tak z toho důvodu, že businesss vrstva potřebuje pro výpočty získat vstupní data, ta se typicky nachází v databázi a o jejich získání se stará právě datová vrstva.

Prezentační vrstva je dále rozdělena na dva balíčky. Jedním je *Controller*, který přímo zpracovává a definuje koncové body API rozhraní. *Form* obsahuje pomocné formulářové třídy, se kterými Symfony pracuje. Business vrstva je složena ze čtyř balíčků, kde do *Mercure* patří pomocné třídy pro zpracování Mercure tokenu, *Security* obsahuje naopak třídy pro práci s *JWT*. Do balíčku *Service* patří obecné služby (například pro převod modelu na výraz pro program Wolfram Mathematica) a *Utils* slouží k obecně definovaným třídám pro práci v ostatních balíčcích business vrstvy. Poslední

datová vrstva je rozdělena na *DataFixtures*, které se používají pro nahrání výchozích dat do databáze, *Entity*, které mapují PHP třídy na databázové tabulky, *Repository*, které slouží pro přímou práci s databází a nakonec *Enum* pro výčtové typy.



Obrázek 4.3 Návrh architektury - Rozdělen na tři vrstvy, které obsahují balíčky tříd. Šipky značí závislosti mezi vrstvami. Model je zpracován v programu Enterprise Architect[27].

4.3 Spojení server-klient

Klientská aplikace pracuje za běhu s daty z databáze. Do ní však nemůže mít z bezpečnostních důvodů přímý přístup a proto probíhá komunikace prostřednictvím serveru. Chce-li si uživatel zobrazit seznam svých projektů, klientská aplikace odešle požadavek na server a očekává odpověď. Server má v tuto chvíli několik zodpovědností. Musí zkontrolovat, o jakého jde uživatele a zda má přístup k projektům, následně získat informace o projektech z databáze a odpovědět uživateli. Takto popsaná komunikace probíhá podle serverem definovaných pravidel (API). Pro tyto typy aplikací je nejpoužívanější architektonický styl REST.

REST (Represantational State Transfer) je založený na protokolu HTTP. Signifikatní vlastností pro REST je práce se zdroji, kde zdrojem může být libovolná informace, kterou lze pojmenovat. V kontextu této aplikace může být zdrojem uživatel, projekt, blok, parametr a další. Tyto zdroje jsou následně identifikovány pomocí URI. Pro práci s těmito zdroji jsou využívány HTTP metody (GET, POST, DELETE, ...). Každá metoda určuje akci, kterou chceme požadavkem vyvolat. Chceme-li

například získat informace o projektu, použijeme metodu GET na URI daného projektu (např. */projects/1*). Formát dat, který klient od serveru získá může být různý – nejčastěji se používá JSON nebo XML, ale lze kombinovat i více formátů. Důležitým bodem je, že komunikace přes REST API musí být bezstavová, server nezná kontext předchozích požadavků a každý požadavek musí obsahovat všechny informace potřebné pro jeho zpracování.

Při návrhu REST API vycházím z předchozí definice požadavků. Rozhraní představuje kompletní a jedinou komunikaci mezi uživatelem a serverem. Nejedná se tedy o pouhý doplněk pro externí aplikace, nýbrž o plnohodnotné rozhraní. Jako první krok identifikuji všechny zdroje a přiřadím příslušné URI. Ke každému zdroji nastavuji povolené metody a stručný popis, ve kterém je vysvětlen význam dotazu a podmínky pro jeho úspěšné splnění.

URI zdroje	**Metoda**	**Popis**
/api/project	GET	Vrátí informace o všech projektech.
	PUT	Vytvoří nový projekt.
/api/project/{id}	GET	Vrátí informace o projektu.
	PUT	Upraví projekt identifikovaný podle id.
	DELETE	Smaže projekt identifikovaný podle id.
/api/project/{id}/model	GET	Vrátí modely patřící projektu identifikované podle *id*.
/api/project/{id}/model/{type}	PUT	Vytvoří model typu *type* pro projekt identifikovaný *id*.
/api/project/{id}/documentation	GET	Vrátí seznam dokumentací projektu identifikovaného podle id.
	PUT	Vytvoří novou dokumentaci.
/api/project/{id}/user	GET	Vrátí seznam uživatelů přiřazených k projektu.

Tabulka 4.1 Schéma REST API

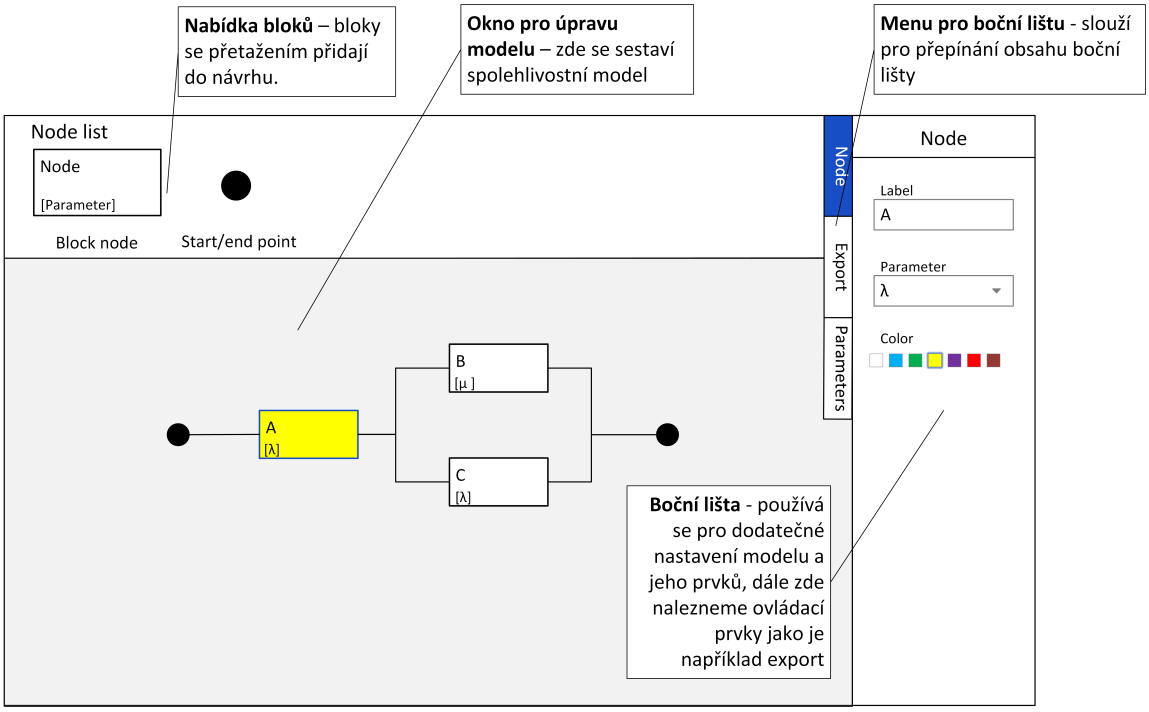
V tabulce 4.1 je dle schématu REST API patrné, že se množina zdrojů velmi podobá množině stavů doménového modelu 4.1 (kompletní dokumentace je ve fyzické příloze práce). Je to logické rozdělení množiny zdrojů, protože nad všemi stavy doménového modelu můžeme běžně provádět právě operace, které REST umožňuje. Většina zdrojů umožňuje metody GET, PUT a DELETE, které mají pro každý zdroj podobný význam. Liší se návratové kódy. V případě kódu pro úspěch (2xx) jsou ve schématu kódy 200 a 201 podle provedené akce. Chybové kódy pro chybu na straně uživatele najdeme tři, 404 pro neexistující zdroj, 403 pro neoprávněnou akci a 400 pro chybu v některé části požadavku (např. selhání validace vytvářeného objektu).

4.4 Návrh spolehlivostního modelu

Vytvoření spolehlivostního modelu je stěžejní částí aplikace. Pravděpodobně se jedná o nejčastěji využívanou funkci, na které stráví uživatelé mnoho času. Je pochopitelné, že tvorba libovolného

typu spolehlivostního modelu musí být uživatelsky velmi přívětivá a navržena tak, aby uživatelům usnadnila práci. Z analýzy dosavadních řešení vyplývá, že výsledná forma by měla být co nejvíce interaktivní. Z rozboru aplikace *Spolehlivost* (3.1.1) jasně vyplynulo, že využití tlačítek jako ovládacích prvků vede k nevhodnému ovládání modelu.

O dosažení požadovaného chování se stará klientská část aplikace. Server pouze poskytuje REST API v takovém schématu, aby byl klient schopný s modelem správně pracovat. Rozsah práce je sice omezen pouze na dva typy spolehlivostních modelů, avšak návrh by měl do budoucna počítat s myšlenkou implementace dalších modelů, které zatím nejsou součástí práce. Implementace nových rozšíření by měla probíhat bez větších zásahu do stávajícího kódu.



Obrázek 4.4 Návrh stránky pro návrh spolehlivostního modelu

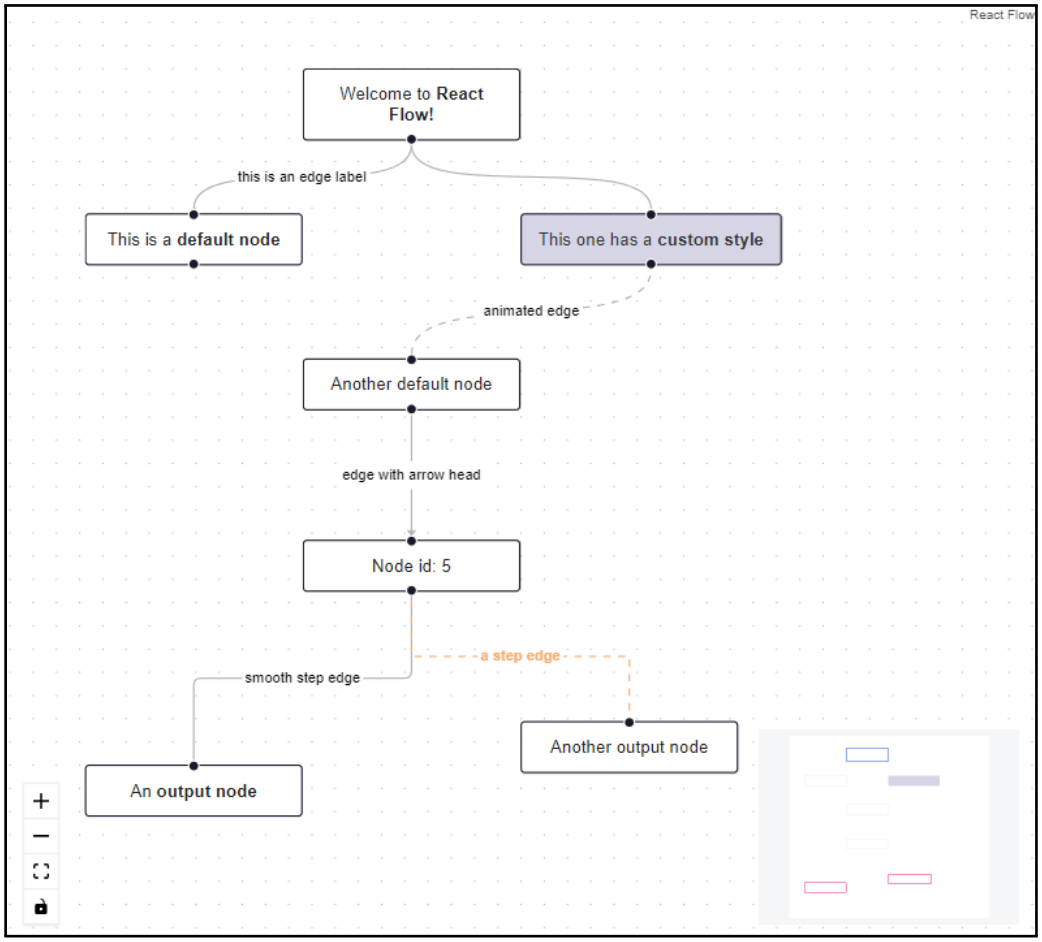
Návrh by měl být velmi obecný a musí podporovat *customizaci* všech prvků. Již v rámci doménového modelu je využita obecná vlastnost modelů – všechny se skládají ze dvou základních objektů, bloků (node) a hran (edge). Tyto objekty můžeme dále dělit na různé typy, protože blok pro Markovský model má jiné vlastnosti a chování než blok pro blokový model. Stále se však jedná o bloky a z hlediska návrhu budou mít některé shodné vlastnosti. Při rozšíření o další modely to umožní aplikaci škálování těchto typů objektu, obecně se však bude jednat o stejný objekt.

Z analýzy existujících aplikací vychází, že k dosažení rychlého a intuitivního rozhraní je důležité mít model co nejvíce interaktivní. Jednotlivé bloky by měly být zobrazeny v nabídce a jednoduchým přetažením do modelu by měl být vytvořen nový blok. Přičemž jakýkoliv existující blok musí jít jednoduše přesunout v rámci plochy pro návrh modelu. Obdobné podmínky by měly platit u natažení hran z jedno bloku do druhého.

Všechny ovládací prvky není vhodné integrovat přímo do okna modelu, jelikož by to mohlo vést ke zhoršenému ovládání modelu, proto jsou dodatečná nastavení v boční liště. Tam najdeme nastavení pro bloky, parametry i celý model. V horní části lišty je navigační menu, které slouží

pro výběr obsahu. Po výběru některé z nich se lišta vyplní příslušným nastavením. Mezi možné funkce určitě patří úprava bloku, úprava hrany, definice parametrů nebo export Wolfram notebooku.

K výše popsanému postupu velmi dobře slouží knihovna React Flow. Model vytvořený knihovnou pracuje právě se dvěma objekty, blokem a hranou. Oba mají předdefinované parametry a výchozí styl zobrazení. Pokud se tento styl nehodí do designu aplikace, lze jednoduše vytvořit vlastní vzhled komponenty včetně odlišného chování objektu. Vlastnosti knihovny se shodují s požadavky pro návrh modelu. [28]



Obrázek 4.5 Ukázkové schéma knihovny React Flow[29]

Model vykreslený knihovnou React Flow je možné rozšířit o ovládací prvky. Na obrázku 4.5 se jedná o ikony vlevo dole. Základní prvky jsou již připraveny k použití bez další implementace, Jedním z nich je vycentrování modelu na obrazovce nebo přiblížení či oddálení okna. Knihovna je připravena i možnost implementace vlastních ovládacích prvků. Minimapa, která je na obrázku 4.5 vpravo dole je také již připravena jako součást knihovny a pro použití ji stačí jednoduše zapnout.

4.4.1 Blokový model

Blokový model je prvním typem spolehlivostních modelů, se kterým aplikace pracuje. Příklad takového modelu je součástí obrázku 4.4 a skládá se z bloků, které reprezentují nezávislé části systému. Každý tento blok má stejný význam a nepracuje se zde s významově různými typy bloků. Jednotlivé bloky se liší pouze v intenzitě poruch. Propojeny jsou hranami, jejichž význam ovlivňuje pouze zdrojový a cílový blok. Hrana sama o sobě nenese žádnou informaci (pro výsledek modelu). Blok a hrana tedy tvoří množinu prvků, které jsou nutné pro vytvoření blokového modelu.

Na tento typ modelu velmi dobře vystačí prvky knihovny React Flow. Pokud bych postupoval pouze podle základní specifikace modelu, nastane problém při zobrazení počátečního a koncového bloku. Modeluji-li čistě paralelní model 2.4, existuje několik počátečních i koncových bloků a to může být problém při validaci modelu i zpracování výsledku. Z tohoto důvodu přidám do množiny prvků ještě další typ bloku. Ten slouží právě jako označení počátku i konce modelu a reprezentován je jako černý plný kruh. Sám o sobě nepřináší žádný význam ani pro něj nedává smysl nastavovat intenzitu poruch. Zde už se situace s paralelním modelem zlepší a zvýší se tím i přehlednost pro uživatele. Stále se ale nejedná o ideální situaci, narážíme zde totiž na problém s knihovnou React Flow. Ta umožňuje propojení bloků pomocí tzv. *handlerů*. Ty lze standardně přidat na jakýkoliv blok a natažením hrany z jednoho *handleru* do druhého vznikne nová hrana (projde-li validací). *Handler* však může být pouze vstupní nebo výstupní, nikoliv obojí. To není problém pro běžné bloky, jelikož je standardní postup modelovat závislosti zleva doprava (levý *handler* je vstupní, pravý je výstupní). Počáteční a koncový blok má z logiky věci pouze jeden *handler*. Proto rozděluji tento prvek na dva, jedním bude počáteční s výstupním *handlerem* vpravo a druhý koncový se vstupním *handlerem* vlevo. Takto jsou bloky vnímány i při validaci a výpočtu, není tedy možné začít model koncovým stavem a naopak.

Obrázek 4.6 Množina prvků blokového modelu - Počáteční blok, koncový blok a standardní blok představující část modelovaného systému.

Model již mohu sestavit, nyní je nutné nastavit správné parametry a další informace. Aby měl model smysl, musí být nutně pro všechny standardní bloky (ne počáteční ani koncové) nastavena intenzita poruch. Ta musí být vyplněna v nějakém číselném poli, proto se toto nastavení provádí v boční liště (zobrazena na obrázku 4.4). Pro přehlednost a opakované použití parametrů jsem se rozhodl neumisťovat číselné hodnoty přímo v nastavení bloku. Důvodem pro toto rozhodnutí je lepší škálovatelnost při definování nových typů parametrů pro stávající i budoucí typy modelů. Parametry jsou tedy definovatelné ve vlastním seznamu a tento seznam je použitelný v rámci celého modelu. U každého bloku pak uživatel pouze vybere jeden z definovaných parametrů pro intenzitu poruch. Název tohoto parametru se následně zobrazí ve spodní části bloku v hranatých závorkách.

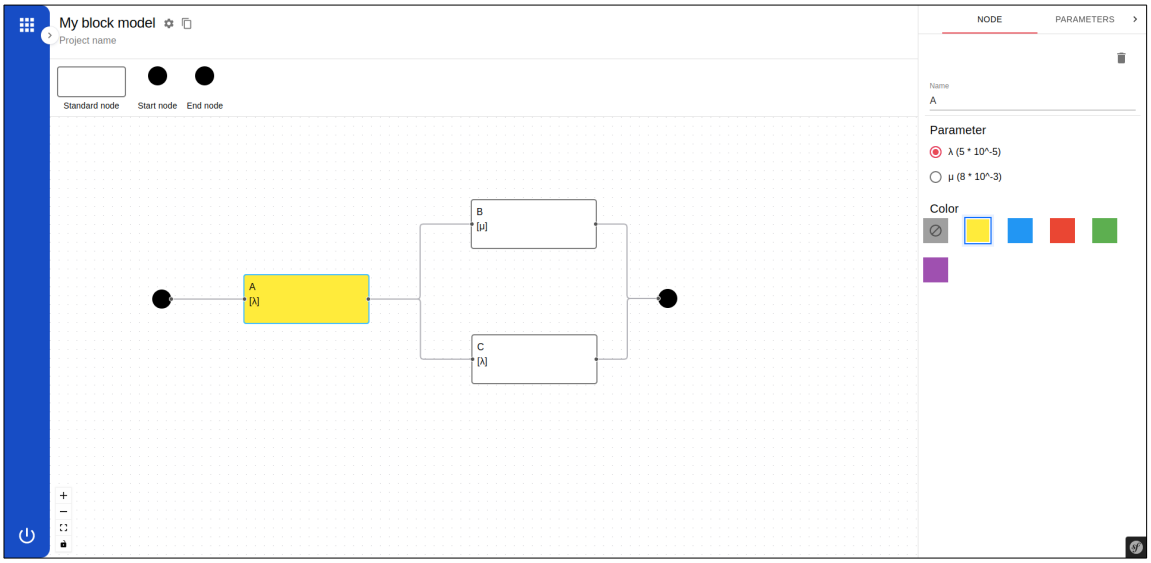
Intenzita poruch se běžné pohybuje v malých zlomcích. Proto je hodnota zadávána ve formátu základu (base) a exponentu, kde $\lambda = base \cdot 10^{exponent}$.

Další nastavení pro standardní blok tvoří jeho název, ten je následně zobrazen v horní části bloku. Uživatel si může také přizpůsobit barvu bloku, ta však slouží pouze jako označení pro uživatele a implicitně nenese žádný význam. Klientská aplikace nabízí pro zjednodušení pár základních barev, server i databáze dokáže ale zpracovat libovolnou barvu z RGB spektra.

Blokový model pro hrany nemá žádné parametry. Pro tuto aplikaci jsem přidal možnost hranu

pojmenovat – toto nastavení je čistě volitelné, uživatel ho nemusí využít, ale v některých případech může zvýšit přehlednost modelu.

Samotnému modelu může uživatel také upravit základní informace, jedná se hlavně o název a popis. Obě slouží k identifikaci, k lepšímu pochopení významu modelu a nevíc jsou využity jako podklady pro dokumentaci.



Obrázek 4.7 Blokový model v aplikaci pro spolehlivostní modely - Jedná se o stejný model jako v návrhu 4.4 vytvořený v aplikaci pro spolehlivostní modely

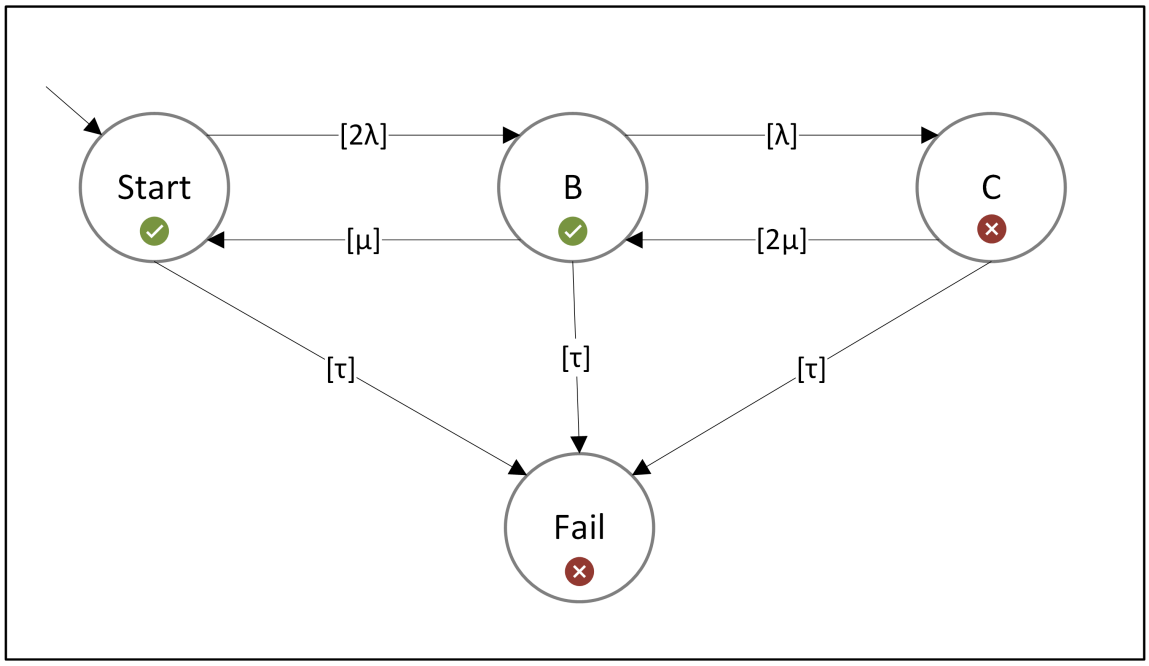
Modelování je připraveno a po úspěšném sestavení modelu se očekává, že aplikace připraví v nějaké formě výstup. Ten je generován v podobě notebooku pro Wolfram Mathematica. V notebooku jsou připraveny výpočty pro bezporuchovost celého systému ($R(t)$) a střední doby bezporuchového provozu (T_s). Spolu s číselným výpočtem notebook obsahuje také graf pravděpodobnosti bezporuchového stavu v čase. Výhodou přítomnosti výstupu v externí aplikaci jako je Wolfram Mathematica je možnost nezávislých úprav. Uživatel po vygenerování notebooku může jeho obsah libovolně upravovat, například měnit parametry některých bloků a zkoumat výsledek. Více informací rozebírám v následující podkapitole 4.5.

4.4.2 Markovský model

Model Markovského typu je návrhem odlišný od blokového modelu, oba ale mají společné znaky. I zde platí obecné rozdělení prvků na uzly a hrany. Markovský model má však pouze jeden typ uzlu, ten reprezentuje stav, ve kterém se systém může nacházet. Jedinou povinnou vlastností je jeho jedinečná identifikace v rámci všech uzlů, které tentokrát nenesou žádnou číselnou hodnotu, jako je to v případě blokového modelu. Hrany jsou vždy orientované (neplatí zde implicitní směr zleva doprava) a také představují přechod z jednoho stavu do druhého. Tento přechod může nastat s nějakou pravděpodobností. I v Markovském modelu lze hranu, pro lepší orientaci, pojmenovat.

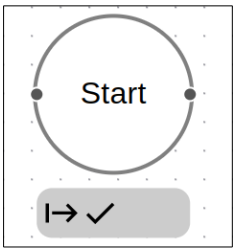
Pro Markovský model existuje více notací a každý nástroj si může notaci lehce upravit tak, aby vyhovovala při práci s daným nástrojem. Já pracuji s notací, kde jsou všechny stavy modelu tvořeny uzly stejného typu, liší se však jejich vnitřním nastavením. Obecně je zvykem tyto stavy

zobrazovat jako kruhy, kde uvnitř kruhu je zapsán název stavu. Stavy Markovského modelu mají více nastavení než v případě blokového modelu. Tato nastavení jsou samozřejmě viditelná v boční liště, to je ale problematické v případě, že chceme model pouze číst. Na první pohled nejsou tato důležitá nastavení viditelná, což může vést k chybné interpretaci modelu. Proto jsem se pod každý uzel rozhodl přidat panel ikon, díky kterému budou důležité parametry viditelné přímo v modelu.



Obrázek 4.8 Návrh Markovského modelu - Model je tvořen čtyřmi stavy, samostatná vstupní šipka určuje počáteční model a ikony pod názvy stavů označují funkční či poruchový stav

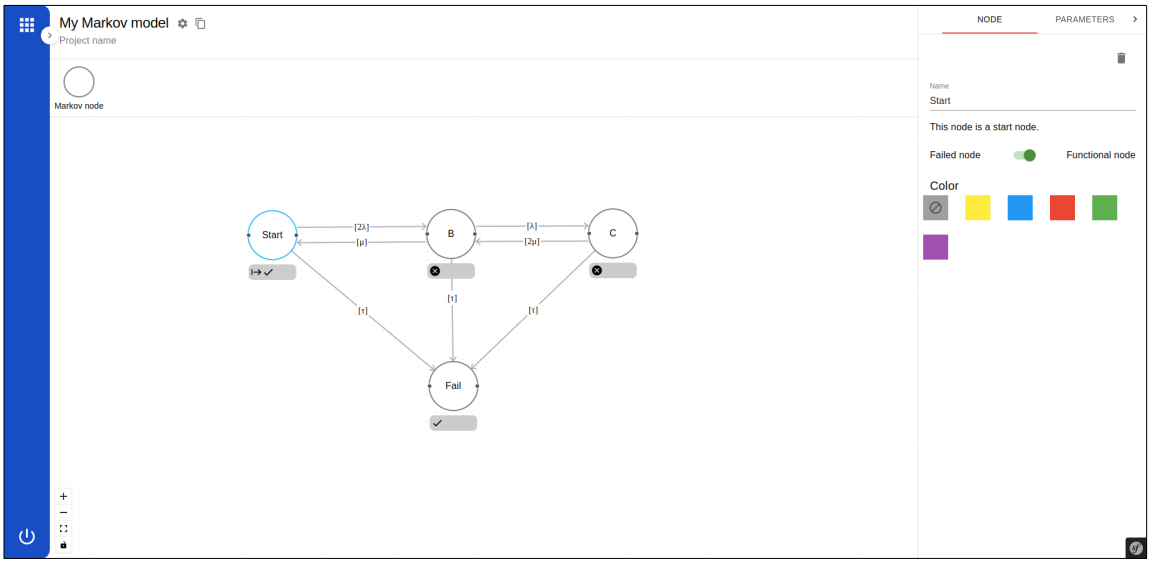
Aby dával model smysl, musí v počátečním čase začít v některém ze stavů. Tento stav se označuje jako počáteční. To tvoří první atribut. Blok reprezentuje nějaký stav modelovaného systému, musíme tedy označit, zda je v tomto stavu systém funkční či poruchový. Posledním čistě grafickým prvkem je obarvení bloku ze seznamu barev. Všechna tato nastavení lze, po výběru daného bloku, řídit v boční liště.



Obrázek 4.9 Blok Markovského modelu s lištou atributů - Lišta se skládá ze dvou ikon – první značí, že se jedná o startovní blok, druhá stav, ve kterém je systém funkční

Po sestavení modelu je i zde možnost vygenerování výsledků do notebooku. Výpočty pro Mar-

kovský model jsou rozděleny na dva typy v závislosti na existenci absorpčních stavů. V modelu s absorpčními stavy se může model dostat do stavu, ze kterého nevede žádná hrana, proto v tomto stavu zůstane. To má značný vliv na formu výsledku. Hledané veličiny jsou pravděpodobnost bezporuchového stavu ($R(t)$) a střední doba bezporuchového stavu (T_s). Pravděpodobnost bezporuchového stavu lze podobně jako u blokového modelu zobrazit do grafu v závislosti na čase. Pokud sestavený model neobsahuje žádné absorpční stavy, vypadá notebook jinak. Jelikož se zařízení nikdy nedostane do absorpčního stavu, zajímá uživatele hlavně stacionární součinitel bezporuchového stavu (resp. poruchového stavu) (K_p, resp. K_n).



Obrázek 4.10 Markovský model v aplikaci pro spolehlivostní modely

4.5 Wolfram Notebook

Od všech aplikací tohoto typu se očekává, že po vytvoření modelu a nastavení parametrů vrátí v nějaké formě alespoň základní výpočet. Tento výsledek může mít několik forem. Například aplikace *Spolehlivost* vrací výsledek ve formě jednoduchého textového pole, kde jsou vypočteny základní veličiny. V případě aplikace, by bylo možné podobný přístup vytvořit také a výsledek uživateli jednoduše zobrazit. Pokud bych chtěl integrovat tento postup přímo do aplikace, je důležité myslet na to, že modely pracují běžně s velmi malými čísly (např. $3 \cdot 10^{-5}$) a to může vytvořit problém s přesností výsledku. Navíc každá hodnota musí být počítána zvlášť a uživatel nemá možnost dále tento výsledek upravit. Proto jsem zvolil externí výpočetní aplikaci, která bude výpočty zpracovávat. Důvodem je také skutečnost, že aplikace může sloužit (mimo jiné) pro výukové účely a Wolfram Mathematica je v tomto prostředí často používanou aplikací, která umožňuje mnoho zajímavých funkcí. Nevýhodou tohoto řešení je samozřejmě nutnost vlastnit aplikaci Wolfram Mathematica, která je veřejně dostupná pouze v placené verzi. I to je důvodem, proč by se výsledná aplikace neměla omezovat na jeden typ výstupu a měla by být připravena na budoucí rozšíření. Jedním z nich může být například využití veřejných API, které umí zpracovat libovolné matematické výrazy.

```
Print["Output for 'Model A' of 'Project 1'"]
timeY = 24 × 365;

(*Parameters*)
parameters = {P1 → 5/10^4, P2 → 1/10^4, P3 → 8/10^4};
(*Nodes*)

BlockA[t_] := e^(-P1 t) /. parameters;
Print["BlockA[t]: ", BlockA[t]];
BlockB[t_] := e^(-P2 t) /. parameters;
Print["BlockB[t]: ", BlockB[t]];
BlockC[t_] := e^(-P3 t) /. parameters;
Print["BlockC[t]: ", BlockC[t]];

(*Result equation*)
R[t_] := BlockA[t] (1 - (1 - BlockB[t]) * (1 - BlockC[t]));
Print["R[t]: ", R[t]]

Ts[t_] := ∫_0^∞ R[t] dt;
Print["Ts[t]: ", Ts[t]]

(*Result Plot*)
Plot[{R[t], BlockA[t], BlockB[t], BlockC[t]}, {t, 0, timeY},
 PlotRange → All, PlotLabel → "Spolehlivostní funkce R(t)",
 AxesLabel → {"t", "R(t)"}, AxesLabel → {"t", "R(t)"},
 PlotLabels → {"R(t) - celková", "BlockA[t]", "BlockB[t]", "BlockC[t]"}]
```

Obrázek 4.11 Vygenerovaný notebooku v programu Wolfram Mathematica - Jedná se o výstup pro blokový spolehlivostní model skládající se ze třech bloků. Stejný model je znázorněn na obrázku 4.7.

Output for 'Model A' of 'Project 1'

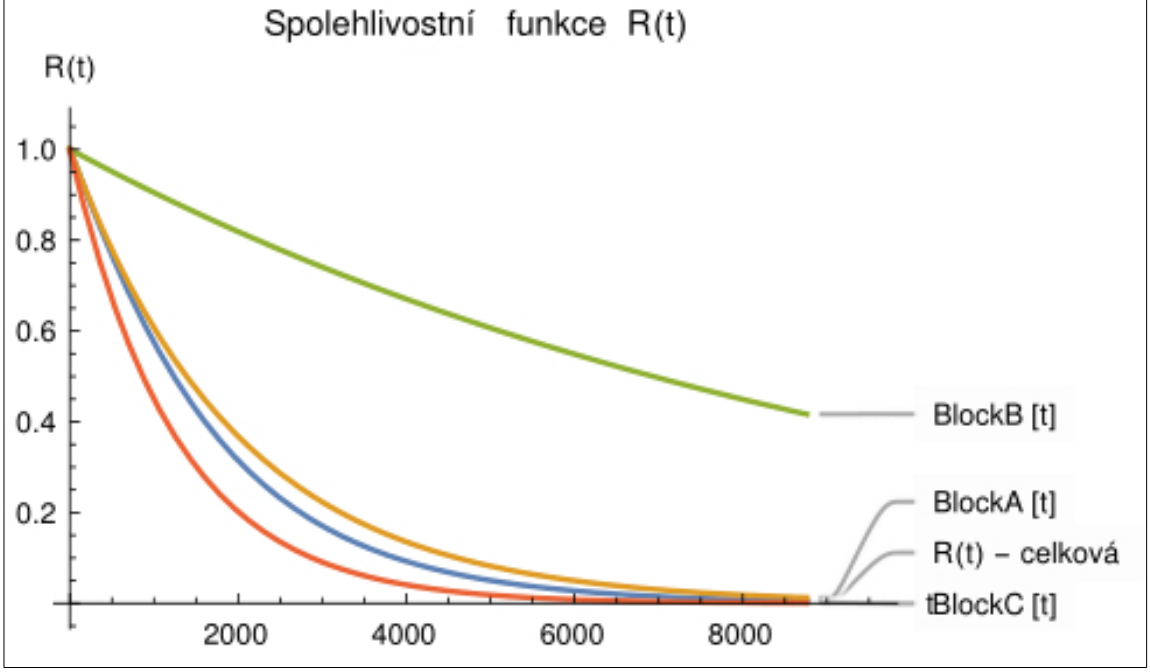
BlockA[t]: $e^{-t/2000}$

BlockB[t]: $e^{-t/10\,000}$

BlockC[t]: $e^{-t/1250}$

R[t]: $e^{-t/2000}\left(1-\left(1-e^{-t/1250}\right)\times\left(1-e^{-t/10\,000}\right)\right)$

Ts[t]: $\frac{470\,000}{273}$

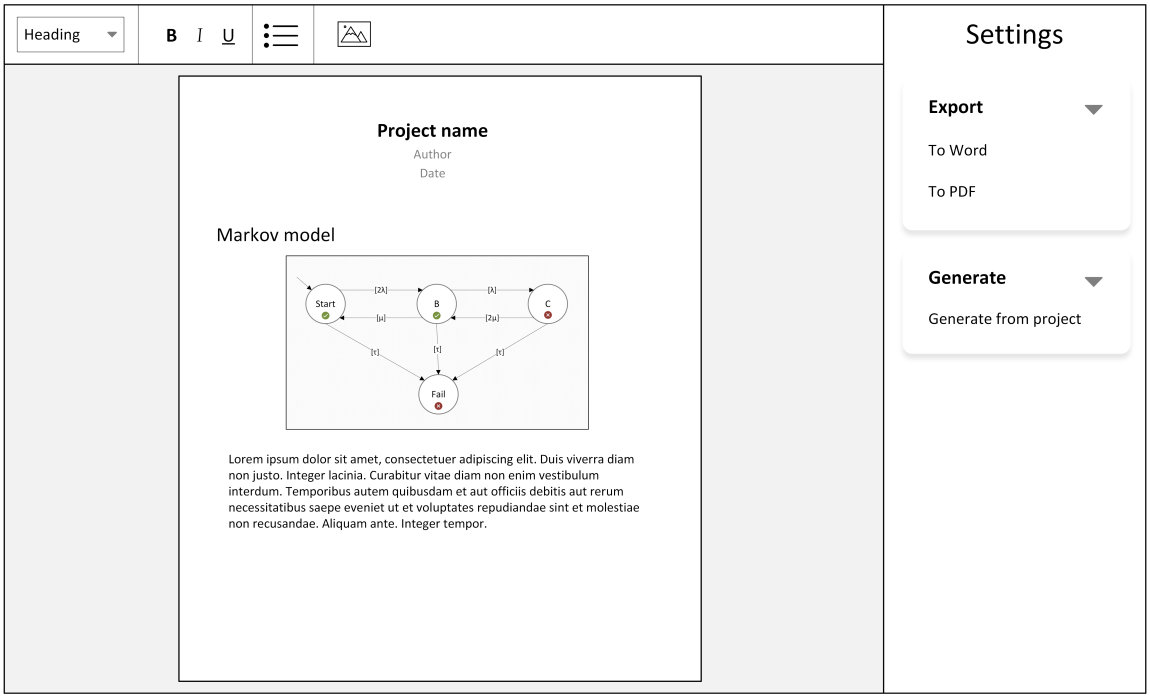


Obrázek 4.12 Vyhodnocený Wolfram Notebook - Jedná se o výstup, který je vyhodnocen notebookem 4.11.

4.6 Dokumentace projektu

Je možné, že projekty vytvořené v aplikaci budou často robustní a poskládané z mnoha modelů. Pokud mají zároveň i modely mnoho bloků jednoduše se uživatelé mohou dostat do situace, kdy nebude projekt přehledný. Z analýzy aplikací *Spolehlivost* a *SHARPE* je patrné, že komplexní dokumentace

musí být vytvořena mimo tyto aplikace. To přináší mnoho nevýhod, jako je nutnost dokumentaci stále aktualizovat a také nutnost použití externích textových editorů. Dokumentace je v oblasti spolehlivosti nedílnou součástí projektu a integrace alespoň základní dokumentace může mít velkou přidanou hodnotu.



Obrázek 4.13 Návrh dokumentace - Zpracován je základ stránky, která slouží k úpravě dokumentace spolu s boční lištou pro generování a export dokumentace. Obsah dokumentace v obrázku je návrhem základní šablony dokumentace.

Dokumentace je zařazena na úrovni projektu, to umožňuje obsáhnout v ní více modelů, které jsou významově propojeny. V klientské aplikaci má stránka s dokumentací podobu textového editoru, v hlavní části se nachází strana textu a nad ní nabídka pro jeho formátování. Text umožňuje základní formátování jako je nabídka typů písma (nadpis, normální text, popisek), tučné písmo, barva písma, seznamy a další. Kromě textových formátů lze vložit také libovolný obrázek, což může být využito například ve spojení s exportem modelu do PNG formátu.

Skutečnost, že je textový editor pro dokumentaci přímo součástí aplikace, přináší výhodu v možnostech využití existujícího projektu a jeho modelu. Při práci s modelem můžeme přidávat popisky pro model i pro jeho prvky. Ty poté slouží pro lepší orientaci při sestavování modelu. Tyto texty se budou v dokumentaci často opakovat, proto jsem vytvořil funkci pro automatické generování dokumentace, které na základě již sestavených modelů a jejich popisu vygeneruje základ dokumentace. Postup je velmi jednoduchý, v boční liště uživatel stiskne tlačítko pro generování a otevře se dialog s nastavením (ten obsahuje především šablonu dokumentu). Nakonec stačí výběr pouze potvrdit a dokumentace se vytvoří. Po vygenerování je vždy přepsán dosavadní obsah, pokud chce uživatel obsah zachovat, musí vytvořit novou instanci dokumentace a v ní vygenerovat nový obsah.

Stále zbývá vysvětlit termín šablony. Jedná se o připravené formáty, ve kterých bude dokumen-

tace projektu vygenerována. Například základní šablona obsahuje všechny modely a ke každému z nich vygeneruje popis modelu, seznam prvků. Pro modely, které projdou validací vypíše také základní vzorec pro výpočet pravděpodobnosti bezporuchového stavu. Aplikace počítá s následným rozšířením šablon a jejich dodatečným nastavením pro konkrétní situaci. Například může existovat šablona, ve které uživatel zvolí pouze některé modely.

4.7 Synchronizace

Návrh popsaný v této kapitole je z velké části kompletní, zbývá již navrhnout způsob, jakým podpořit práci více uživatelů na jednom projektu. Aplikace *Spolehlivost* a *SHARPE* mají v tomto směru nevýhodu, protože jsou obě desktopové aplikace, které nejsou připojeny na žádný server. Výsledné modely jsou pouze exportovány do souborů, které musí uživatelé sdílet, což vylučuje paralelní práci na jednom projektu. Webová aplikace má v tomto směru větší potenciál, protože již z principu musí mít alespoň základní server. Při vymýšlení jakou formou bude spolupráce na jednom modelu fungovat jsem se inspiroval několika existujícími aplikacemi používajícími podobnou funkci. Mezi ně patří hlavně Google Docs[20] nebo Whimsical[21]. Tyto systémy fungují na principu synchronizace uživatelů. Pokud nějaký uživatel upraví dokument (model), změna je uložena na server a ten zašle zprávu všem připojeným klientům, kteří provedou stejnou změnu. Takto je uživatelům téměř ihned aktualizován obsah modelu a oni tak mohou reagovat na změny bez složitých komplikací.

Z analýzy vyšlo jako nejlepší řešení použití technologie Mercure[24]. Hlavní částí je Mercure Hub, aplikace, která je spuštěna na serveru a vyřizuje požadavky. Pro práci s ním jsem využil oficiální knihovnu pro Symfony framework. Ta poskytuje základní nastavení spojení s Mercure Hub a třídy, které abstrahují celou komunikaci na pár jednoduše použitelných metod.

4.7.1 Průběh synchronizace

Veškerá komunikace mezi centrem komunikace (Mercure Hub) a PHP serverem probíhá na protokolu HTTP. Hub pouze poslouchá na určitém portu, v aplikaci je port nastaven na hodnotu 3000, lze ho však jednoduše přenastavit. Před popisem konkrétního API je potřeba vysvětlit několik termínů, které Mercure definuje.

Základem je nějaký zdroj dat, který je nazýván ***topic***. K určitému zdroji existuje ***publisher*** a ***subscriber***. *Publisher* může být server nebo klient a jedná se o vlastníka nějakého zdroje, který na tento zdroj zasílá zprávy (***Update***). *Subscriber* je typicky uživatel, který tyto zprávy přijímá. Aby mohl zprávy odebírat, musí se nejdříve k tomuto odběru zaregistrovat pomocí ***Subscription***.

Topic dle specifikace představuje zdroj, což je podobně jako v REST libovolná informace, kterou lze pojmenovat. Pojmenován může být libovolným způsobem, autoři Mercure protokolu však doporučují použít URI formát, čímž vzniknou stejné identifikátory zdrojů pro Mercure i REST. U REST API je zvykem rozlišit URI od ostatních. K tomu je nejčastěji použita nějaká předpona jako například `/api` nebo `/api/v1`, ale není to podmínkou. Pro Mercure platí, že URI ke každému zdroji musí mít předponu `/.well-known/mercure`.

Prvním logickým krokem pro zahájení komunikace je odběr jednoho ze zdrojů, ze kterého chci odebírat informace. Tuto akci vykonává klient a stává se tím odběratelem zpráv (*subscriber*). Poté server nebo klient, který je v daném zdroji *publisher* zašle zprávu na Mercure Hub a ten následně zašle tutéž zprávu odběrateli.

Komunikaci v kontextu aplikace pro spolehlivostní modely popíšu na příkladu vytvoření nového bloku modelu. Máme dva uživatele, kteří pracují na daném modelu (označeny jako A a B). Uživatel A právě pracuje na blokovém spolehlivostním modelu. Uživatel B se taktéž připojí na stránku

Topic	Popis
/model/{id}/node/{type}	Vytvoří nový blok určitého typu.
/model/{id}/node/{id}	Upraví blok identifikovaný pomocí *id*.
/model/{id}/edge	Vytvoří novou hranu.
/model/{id}/edge/{id}	Upraví hranu identifikovanou pomocí *id*.
/model/{id}/parameter	Vytvoří nový parametr.
/model/{id}/parameter/{id}	Upraví parametr identifikovaný pomocí *id*.

Tabulka 4.2 Schéma Mercure API

s modelem a přihlásí se k odběru zpráv na zdroji `/model/1/node/10`. Uživatel A poté přidá nový blok a přes REST API odešle tuto informaci na server, ten následně vytvoří zprávu, kterou jako *publisher* na zdroji `/node/10` odešle na Mercure Hub. Nakonec odešle Hub zprávu o změně všem příjemcům včetně uživatele B a klientská aplikace mu tuto změnu ihned bez nutnosti manuální aktualizace zobrazí.

Kapitola 5

Testování a nasazení

Aplikace je již navrhnuta a v základu implementována, nyní je nutné aplikaci otestovat. Testování webové aplikace může probíhat na několika úrovních. V této kapitole jsou podrobněji popsány ty z nich, které jsou v aplikaci používány. Jelikož je v aplikaci implementován dostatek funkcí alespoň k jejímu základnímu použití, je vhodné připravit ji na skutečné nasazení. To je nejen navrhnuto a popsáno, ale také provedeno na školním webovém prostředí *Cloud FIT* [30]. V kapitole je sepsán podrobný postup nasazení pomocí služby Docker včetně konfigurace celého projektu.

5.1 Unit testy

Jak již bylo zmíněno, webová aplikace může být otestována na více úrovních. Tou nejnižší jsou unit testy. Jedná se o typ testů, při kterém se kontroluje správnost nejmenších částí kódu, většinou funkce nebo metody. Hlavním cílem je oddělit jednotlivé metody a zjistit zda fungují samostatně. Zvykem při psaní unit testů je začít od těch nejmenších a nejnižších částí a postupovat ke složitějším, které mohou být závislé právě na již otestovaných menších metodách. [31] Průběh unit testu se typicky skládá ze tří částí. Každý test musí být připraven v závislosti na požadavcích. Následně je vytvořen skript, který otestuje požadavek a vyhodnotí výsledek. Nakonec je test spuštěn a vrátí informaci o tom, zda se skript zdařil nebo ne. Tento postup přiblížím na příkladu metody, která zjistí, zda má Markovský model nějaký absorpční stav.

Výpis kódu 5.1 Metoda *hasAbsorptionNode* - Implementace metody pro zjištění, zda má model absorpční stav

```
public function hasAbsorptionNode(): bool {
    foreach ($this->getNodes() as $node){
      if($node->isAbsorption()){
        return true;
      }
    }
    return false;
  }
```

Výpočty pro Markovský model jsou závislé na tom, zda model obsahuje absorpční stavy. Vznikne proto metoda, která tuto informaci zjistí v závislosti na daném modelu. Tím je splněn první bod. Následně se vytvoří test obsahující všechny situace, které mohou nastat. Test spustí danou metodu

a očekává správný výstup. Tím je splněna druhá část. Nakonec zbývá pouze spustit testy, které otestují správnost metody *hasAbsorptionNode* 5.1.

Tímto postupem jsem otestoval všechny důležité metody, které vykonávají nějakou logiku aplikace. Pro správu testů a přehled o tom, které části aplikace jsou již otestovány jsem použil příslušné nástroje. K tomuto účelu jsou pro jazyk PHP rozšířené knihovny PHPUnit[32] a PHP Code Coverage [33] (obě knihovny jsou součástí balíčku symfony/test-pack).

5.1.1 PHPUnit

Úkolem unit testů je tedy otestovat ideálně všechny průchody programu a zjistit, zda vrácené výsledky odpovídají výsledkům požadovaným. V jazyce PHP toto řeší knihovna PHPUnit. Jedná se o knihovnu, která má mnoho funkcí a pomocných nástrojů, sloužících ke kompletní práci s unit testy. Součástí testovacího balíčku Symfony jsou i další pomocné nástroje, jako je například generování testovací souborů.

PHPUnit poskytuje základní třídu *TestCase*, která obsahuje metody pro vyhodnocování testů. Každá testující třída by tak měla být instancí třídy *TestCase*. Dále je zvykem takové třídy pojmenovávat podle pravidla `<NazevTestovaneTridy>Test`. Každá metoda představující jeden test by měla být pojmenována `test<NazevTestu>`. Symfony má pro tvorbu testujících tříd nástroj pro příkazovou řádku, který se spouští příkazem `php bin/console make:test`. Podobně funguje samotné spuštění testů, které se spouští jednotlivě (`php bin/phpunit <TestujiciTrida>`) nebo najednou (`php bin/phpunit`).

Aplikace se může skládat z desítek nebo stovek tříd, které mají několik metod. Může to tedy znamenat desítky tříd obsahující unit testy. To může vytvořit problém, jelikož testování tolika tříd nemusí být přehledné a snadno zůstane některá z nich neotestována. Proto má knihovna PHPUnit nástroj zvaný *Code Coverage*. Po spuštění testů vytvoří výstup zadaného formátu (výchozí HTML), který obsahuje zprávu o proběhlém testování. Výsledná zpráva se skládá ze seznamu všech tříd rozdělených dle struktury adresáře a pro každý celek zdrojového kódu zaznamenává informace o tom, které části kódu byly během testování spuštěny a které vynechány. Díky této zprávě má programátor přehled o tom, pro které části kódu stále testy chybí a měly by být přidány.

5.2 Testy REST API

Z pohledu klientské části přistupujeme k serveru pouze jako k rozhraní pro práci s daty. Nepochybně by dalším krokem mělo být otestování správné funkce požadavků na REST API. Otestováno může být několika způsoby. Jedním z nich je využití externího nástroje *Postman*, který dokáže zaslat HTTP požadavek a vyhodnotit odpověď serveru. Nevýhodou tohoto přístupu je nepřímá integrace do zdrojového kódu aplikace, což nutí vývojáře při testování jedné aplikace využívat dva různé přístupy. Vzhledem ke zmíněné nevýhodě jsem zvolil formu testování obdobnou jako v případě unit testů. Symfony nabízí vygenerování testující třídy s názvem *WebTestCase*. Po vytvoření této třídy je připravena implementace pro zasílání požadavků na libovolné URI aplikace.

Pro každý požadavek jsem vytvořil jeden či více testů tak, abych otestoval veškerou jejich funkci. V rámci jednoho testu jsou poslány požadavky, které splňují i nesplňují podmínky pro úspěch a na základě odpovědi se test vyhodnotí. Oproti unit testům však pracuji s celou aplikací, nikoliv pouze s odstíněnými metodami. Při implementaci jsem počítal s tím, že stejné požadavky může vykonávat klientská aplikace, proto testy pracují například s databází, autentizací či autorizací.

Prvním krokem je načtení *DataFixtures*, což jsou speciální třídy, které načítají výchozí obsah do databáze. Po jejich nahrání jsou spuštěné samostatné testy. Pokud požadavek testuje situaci,

kdy musí být uživatel přihlášen (většina požadavků), provede se nejdříve požadavek na přihlášení (5.2), kterým se získá autentizační token a ten se nastaví jako cookie. Poté již server pozná, že se jedná o daného uživatele. V úryvku zdrojového kódu 5.3 je nejdříve použita metoda pro získání tokenu a následně jeho použití při testování tvorby nového projektu.

Výpis kódu 5.2 Metoda pro přihlášení - Metoda získá autentizační token, dle zadaných uživatelských údajů a následně token vrátí

```
protected function getToken(string $email, string $password): string {
  if ($this->token) {
    return $this->token;
  }

  $this->client->jsonRequest('POST', '/api/login_check', [
    'email' => $email,
    'password' => $password,
  ]);

  $this->assertResponseIsSuccessful();
  $responseContent = this->client->getResponse()->getContent();
  $content = json_decode($responseContent, true);
  $this->token = $content['token'];
  return $this->token;
}
```

Výpis kódu 5.3 Metoda testující vytvoření nového projektu - Nejprve je získán autentizační token, následuje série chybných požadavků a nakonec jeden úspěšný požadavek, který vede k vytvoření nového projektu

```
public function testProjectCreate() {
  $this->getToken('test@test.cz', 'testPass');

  $this->client->jsonRequest('PUT', '/api/project');
  $this->assertResponseStatusCodeSame(401);

  $this->client->getCookieJar()
    ->set(new Cookie('oauthToken', $this->token, path: '/'));
  $this->client->jsonRequest('PUT', '/api/project');
  $this->assertResponseStatusCodeSame(400);

  $this->client->jsonRequest('PUT', '/api/project', [
    'name' => 'Project␣A',
  ]);
  $this->assertResponseIsSuccessful();
}
```

5.3 Konfigurace nástroje Docker

Přesuňme se od testování k nasazení. Celá aplikace se vzhledem k jejím požadavkům skládá z několika částí. Jednou je samotná serverová aplikace, dále je to Wolfram Mathematica, Mercure server pro synchronizaci a další nastavení. Mým cílem je připravit nasazení tak, aby administrátor

systému pouze jednoduše upravil výchozí konfiguraci a následně se všechny komponenty sami nainstalovaly v požadovaném stavu. K tomu slouží orchestrační nástroj Docker (jeho krátký popis je součástí analýzy 3.3.4). Ten virtualizuje celý systém ve zvoleném nastavení, který není závislý na hostujícím stroji.

Docker používá pro virtualizaci systém oddělených kontejnerů. Každá komponenta systému tvoří jeden kontejner – ten je dále nastaven tak, aby bylo v případě potřeby možné komunikovat s ostatními kontejnery. Kontejnery se běžně zakládají na nějakém *image*, což je připravený kontejner, který lze rozšířit o další funkce. Typickým použitím základní *image* je samostatná instalace některé z distribucí operačního systému Linux.

Instalaci jednoho kontejneru je možné také rozšířit či upravit pomocí konfiguračního souboru nazvaného **Dockerfile**. Ten vychází ze základní *image* a následně se nastaví příkazy, které jsou spuštěny vždy při sestavení kontejneru. Tento typ konfigurace jsem v aplikaci použil pro kontejner *php*. Jedná se o základní kontejner aplikace, který je založen na oficiálním kontejneru pro PHP [34]. Při sestavení kontejneru jsou doinstalovány potřebné nástroje. Jedná se například o NodeJS pro klientskou aplikaci, Composer pro správu PHP knihoven nebo také *nano* pro případnou úpravu souborů na serveru.

Kontejner s PHP je tedy sestaven, nyní potřebuji nastavit kontejner tak, aby byl přístupný z hostitelského stroje. Toto nastavení probíhá pomocí pomocného nástroje pro Docker, který se nazývá Docker Compose [35]. Konfigurace se nachází v souboru *docker-compose.yml*. Prvním nastavením je otevření portu 80. To zajistí takové chování systému, že pokaždé, když přijde na hostitelský stroj požadavek na portu 80, tento požadavek bude vyřízen kontejnerem a tudíž zpracován pomocí PHP. Dále je nastaveno několik *volumes*, které propojí souborový systém kontejneru a hostujícího stroje. Díky tomu jsou umožněny úpravy některých konfiguračních souborů přímo na hostitelském stroji a zároveň nejsou smazány ani po smazání kontejneru.

Hlavním použitím nástroje Docker Compose je však správa více kontejnerů v rámci jednoho souboru. V aplikaci nyní chybí databáze i mercure server. Pro databázi byl vybrán PostgreSQL. Pro něj, rovněž jako pro PHP, existuje připravený oficiální *image*, který je připraven pro použití při orchestraci pomocí Docker Compose. Dle doporučeného nastavení jsem tedy přidal nový kontejner s názvem *postgres*. Zajímavostí jsou zde proměnné prostředí jako je například název databáze nebo přihlašovací údaje výchozího uživatele. Poslední kontejner nese název *mercure*. Založen je na oficiálním *image* pro Mercure Hub [36]. Ten by bylo možné zahrnout do kontejneru *php*, nicméně je vzhledem k poměrně složitému nastavení vhodé využít právě připravený Docker *image*.

5.4 Průběh nasazení

Docker umožňuje část se samotným nastavením velmi zjednodušit. Dále popíši jednotlivé kroky postupu nasazení. Nejprve je nutné nahrát všechny soubory aplikace na cílový stroj. To lze provést dvěma způsoby. Přímočarým překopírování z přiloženého nosiče nebo stažením projektu pomocí nástroje Git ze systému GitLab, který je dostupný pro použití v rámci projektů na FIT ČVUT. Následně musí proběhnout instalace nástrojů Docker a Docker Compose. Poté stačí aplikaci sestavit příkazem `docker-compose up -d`. Aplikace nainstaluje všechny své komponenty a spustí se. Tento postup je pro administrátory po krocích sepsán v souboru *Readme.md*, který je součástí projektu.

5.5 Wolfram Mathematica

Stále zbývá jedna součást aplikace, která není v části nasazení zmíněna, ale je jeho nedílnou součástí a tím je Wolfram Mathematica. Důvod, proč není její instalace součástí sestavení aplikace pomocí

Dockeru je především ten, že instalační soubor Wolfram Mathematica má velikost několik GB. To by způsobilo problémy při stahování aplikace pomocí nástroje Git. Obecně by tyto soubory neměly být součástí repozitáře, jelikož nejsou přímo součástí projektu. Proto je její instalaci nutné provést manuálně.

Nejprve musí administrátor dostat instalační soubor na server. K tomu je připraven adresář *Mathematic/installer*. Na serveru je propojen s adresářem */var/www/installer* uvnitř kontejneru. Následně je nutné se přesunout do kontejneru pomocí příkazu `docker exec -it php bash`. Uvnitř kontejneru spustí administrátor instalaci. Při instalaci doporučuji potvrdit výchozí lokaci pro instalaci, aplikace s touto lokací dále počítá ve svém výchozím nastavení.

Jednou z nevýhod využití právě programu Wolfram Mathematica je její aktivace. Nelze ji totiž používat bez získání licenčního klíče a jeho aktivování pro danou instanci. Aktivaci není samozřejmě možné provádět bez zásahu administrátora, který musí před použitím zadat aktivační klíč a heslo. Jedná se tedy o další nutnou manuální akci před plnohodnotným spuštěním aplikace. Aktivace probíhá stejně jako instalace uvnitř kontejneru. Po přístupu do kontejneru *php* administrátor spustí příkaz `WolframKernel`, nebo ho spustí pomocí absolutní cesty zvolené při instalaci (ve výchozím stavu `\usr\local\Wolfram\<verze>\Executables\WolframKernel`). Následně se spustí průvodce aktivací – zde je hlavní kód *MathID*, který označuje kód instalace a používá se pro získání hesla k aktivaci. Administrátor nakonec zadá aktivační kód a vygenerované heslo (oboje lze získat na stránce https://user.wolfram.com/).

5.6 Cloud FIT

Jelikož se jedná o webovou aplikaci, pro jejíž plnou funkci není ani přes přítomnost nástroje Docker spuštění aplikace triviální, je pro hodnocení i prezentaci žádoucí nasadit ji na veřejně dostupný server. Fakulta informačních technologií mi pro tyto účely poskytla server v rámci služby Cloud FIT. Cloud FIT je služba spravovaná ICT oddělením FIT ČVUT. Jedná se o platformu přístupnou všem pracovníkům a studentům fakulty pro vytvoření a správu uživatelských serverů. [30] Nasazení probíhalo na přiděleném serveru pomocí fakultní VPN. Server má operační systém Linux s distribucí Ubuntu. Kroky vedoucí k nasazená byly standardní tak, jak je popsáno v předcházejících podkapitolách. V době dokončení práce je aplikace přístupná přes URL https://vratidan.stud.fit.cvut.cz/.

Kapitola 6
Závěr

Cílem práce bylo navrhnout a implementovat webovou aplikaci pro práci se spolehlivostními modely. Vzhledem k velikosti problematiky bylo zadání omezeno na dva typy spolehlivostních modelů: blokové a Markovské.

V analytické části jsem se zabýval zpočátku rozborem obdobných existujících řešení. Pro každé z nich jsem popsal všechny klíčové části, které se týkají výsledné aplikace. Následně jsem zdůraznil výhody a stěžejní funkce, které musí být přítomny v každé podobné aplikaci. Nevýhody, které se převážně týkají uživatelského rozhraní, podrobně rozebírám se zaměřením na to, aby k takovým problémům nedošlo při návrhu aplikace. Na základě těchto poznatků a zadání jsem definoval funkční a nefunkční požadavky spolu s případy užití. Nakonec jsem analyzoval vhodné nástroje pro implementaci. Výstup vedl k výběru programovacích jazyků a frameworků pro serverovou a klientskou část aplikace.

V návrhu jsem se věnoval několika stěžejním bodům webové aplikace, které mají přímý vliv na splnění definovaných cílů serverové i klientské části. Server zajišťuje především definici entit a vztahu mezi nimi. Tato data nadále poskytuje ve formě rozhraní pro komunikaci s klienty. To je navrženo dle standardní REST architektury. Klient pomocí zmíněného rozhraní komunikuje se serverem a s využitím externích knihoven umožňuje interaktivní tvorbu spolehlivostního modelu a dokumentace. Právě v těchto bodech je kladen důraz na paralelní práci více uživatelů.

Součástí kapitoly *Testování a nasazení* je popis veškeré konfigurace a postup instalace celého projektu. Nasazení je provedeno pomocí nástroje Docker. Kapitola je dále věnována testování, to je prováděno ve dvou úrovních, nejprve jsou popsány jednotkové testy pro otestování jednotlivých metody na serveru, dále testy rozhraní pro komunikaci mezi serverem a klientem.

Práce obsahuje několik součástí, s jejichž implementací jsem do té doby neměl žádné zkušenosti. Jednou z nich je synchronizace uživatelů v reálném čase. Po analýze možností a několika implementacích jsem nejvíce času věnoval důkladnému nastavení Mercure serveru. Správné nastavení komunikace, především ze strany klienta mi zabralo mnoho času, v některých případech jsem narazil i na omezení této komunikace ze strany prohlížeče. Synchronizaci jsem úspěšně vyřešil správným nastavením Mercure serveru a konfigurací připojení ze strany klienta. K tomu mi pomohla oficiální dokumentace Mercure i dokumentace nástrojů pro JavaScript.

Dokumentace projektu se skládá z více částí, tou hlavní je textový editor. Při jejím zpracování jsem se chtěl připodobnit známým textovým editorů ve webovém prostředí (např. Google Docs). Mnoho času jsem strávil výběrem vhodné knihovny, která by mi něco takového umožnila. Nakonec jsem zvolil knihovnu QuillJs, kterou jsem pomocí složitých rozšíření zvládl dostat do stavu, který

umožňuje v textu používat základní formáty, na které jsou uživatelé zvyklí z ostatních aplikací. Mezi ně patří například tučný text, kurzíva, seznamy nebo vkládání obrázků. Není to ale jediný problém, na který jsem při práci s dokumentací narazil. Jedním z bodů, který je v dokumentaci samozřejmý je export do externího souboru. Při výběru formátu jsem se soustředil primárně na *pdf* a *docx* pro Microsoft Word. V průbehu implementace jsem objevil problémy s podporou některých textových formátů, jako je odsazení, obrázky a další. Proto jsem omezil export pouze na dokument pro Microsoft Word, který zvládá všechny textové formáty používané v textovém editoru dokumentace.

Zdaleka s největšími potíže způsobila práce s programem Wolfram Mathematica a WolframScript pro příkazovou řádku. Notebook, který je vrácen jako výstup spolehlivostních modelů jsem musel nejdříve sestavit pomocí šablony a následně generovat pomocí příkazové řádky. Wolfram Mathematica však musí být pro toto použití správně aktivována, k tomu je vyžadován licenční klíč. Bohužel z důvodu změny přístupu fakulty nebylo možné program jednoduše aktivovat z příkazové řádky. V aplikaci nasazené na fakultní službě CloudFIT, jsem využil časově omezené aktivační klíče. Správný postup je popsán aktivace je popsán v této práci i uživatelské příručce.

Příloha A

Uživatelská příručka

A.1 Přihlášení a registrace

Prvním krokem pro používání aplikace je vytvoření nového uživatelského účtu. To je možné na adrese *vratidan.stud.fit.cvut.cz/register* (aktuální doména, pro odlišné instalace se bude lišit), která obsahuje registrační formulář určený pro nové uživatele.

Uživatel nejdříve zadá email s heslem a stiskne tlačítko *Register*. Pokud jsou zadané údaje platné a zatím neexistuje uživatel se stejným emailem, je účet vytvořen a uživatel přesměrován do aplikace. Pokud při registraci nastane problém, zobrazí se chybová hláška a uživatel je požádán, aby registraci opakoval.

V případě, že má již uživatel vytvořený účet, probíhá přihlášení pomocí formuláře na stránce *vratidan.stud.fit.cvut.cz/login*. Uživatel zadá svoje přihlašovací údaje, stiskne *Login* a pokud jsou údaje správné, je přesměrován do aplikace.

A.2 Projekt

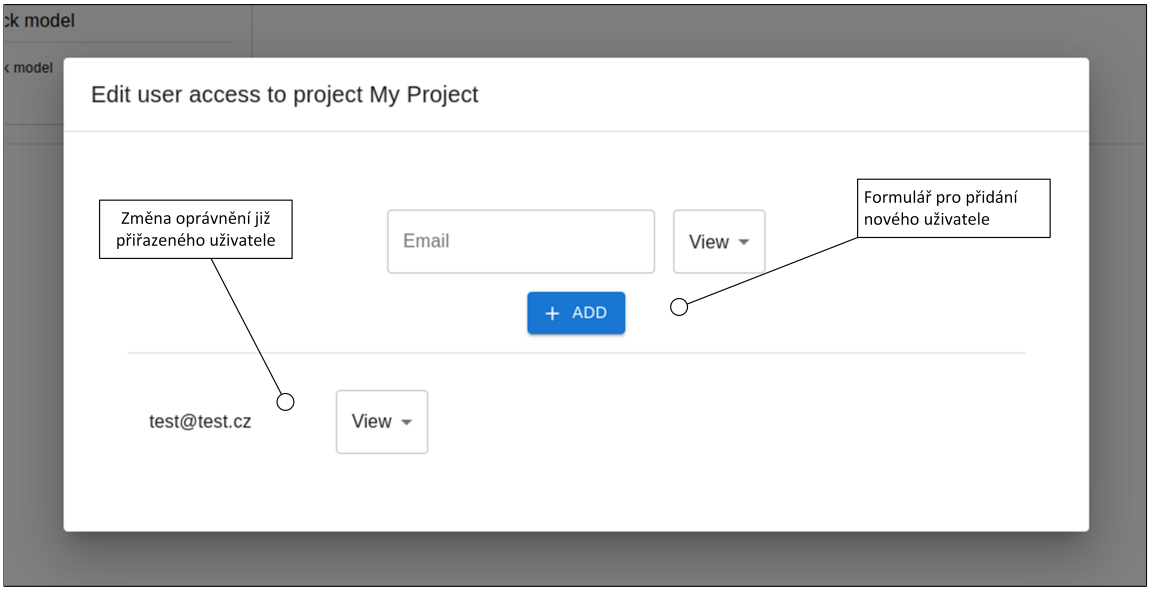
Po přihlášení je uživatel přesměrován na stránku se seznamem projektů, z počátku je tento seznam prázdný. Projekt se zde může objevit dvěma způsoby.

- Uživatel byl přidán k projektu jiným uživatelem.
- Uživatel vytvořil vlastní projekt.

Projekt lze přidat ikonou + vedle titulku. Po stisknutí se zobrazí formulář pro vytvoření projektu. Po jeho odeslání je uživatel přesměrován na detail projektu.

Detail projektu obsahuje seznam modelů, dokumentací a dále také správu uživatel. Zde je možné přiřadit nové uživatele k projektu (horní část dialogu), nastavit jim oprávnění nebo jim přístup zrušit (spodní část dialogu).

Hlavní část detailu projektu je ovšem věnována seznamu jednotlivých modelů a dokumentací. Nový model či dokumentaci lze vytvořit podobně jako u projektu ikonou + vedle příslušného titulku (*Models* a *Documentations*). Na to navazuje formulář pro jejich vytvoření. V případě výběru modelu je formulář tvořen textovými poli pro název a popis společně s *radio* tlačítky pro určení typu modelu. Naopak formulář pro dokumentaci je tvořen pouze dvěma textovými poli - pro název a verzi.



Obrázek A.1 Formulář pro správu uživatelů přiřazených k projektu - V horní části je formulář pro přidání nového uživatele, ve spodní části je již přidaný uživatel s emailem *test@test.cz*, kterému můžeme změnit oprávnění nebo zrušit přístup

A.3 Model

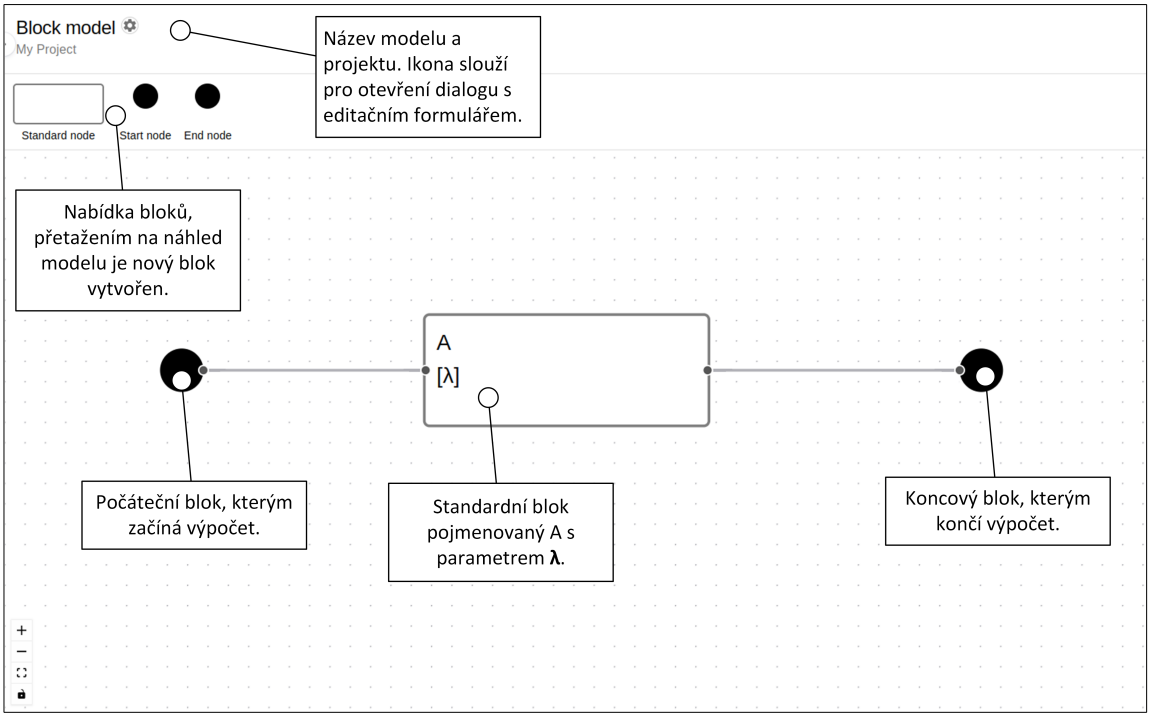
Práce s modelem se liší podle jeho typu. Některé aspekty jsou však stejné pro oba typy. Model se vytváří v hlavní části okna. Nad ní je lišta s bloky, které lze v modelu použít. Každý blok se vytvoří přetažením z této lišty právě do hlavní částí modelu na místo, na které bude umístěn. Všechny modely mají také boční lištu, která slouží pro dodatečná nastavení nebo ovládání modelu či jeho částí. Jedním ze společným prvků je export do notebooku pro program Wolfram Mathematica, který se nachází pod záložkou *EXPORT*.

A.3.1 Blokový model

Blokový model má tři typy prvků, které může uživatel využít. Počáteční a koncový blok je možné vložit vždy pouze jeden od každého typu, slouží pro identifikaci začátku a konce modelu. To je následně využíváno při počítání nebo validaci navrženého modelu. Třetí typ bloku již ovlivňuje výsledek, jedná se o standardní blok blokového modelu, kterému může uživatel pomocí boční lišty nastavit parametr pravděpodobnosti bezporuchového stavu.

Bloky jsou propojovány pomocí bodů na levé či pravé straně bloku. Pokaždé platí pravidlo, že levý bod je vstupní a pravý bod je výstupní (toto platí i u Markovského modelu), každý z nich může být použit pro libovolný počet hran. Dva bloky poté propojíme přetažením kurzoru z výstupního do vstupního bodu. Standardní bloky mají jeden vstupní a jeden výstupní bod, oproti tomu počáteční blok má pouze výstupní bod a naopak koncový blok má pouze vstupní bod.

Standardním blokům lze kromě parametrů nastavit také název nebo barvu bloku. Název je využíván ve výpočtech a barva je pouze orientační a nenese žádný implicitní význam. Hrany nepředstavují žádné číselné parametry, avšak je lze pro zvýšení přehlednosti také pojmenovat.



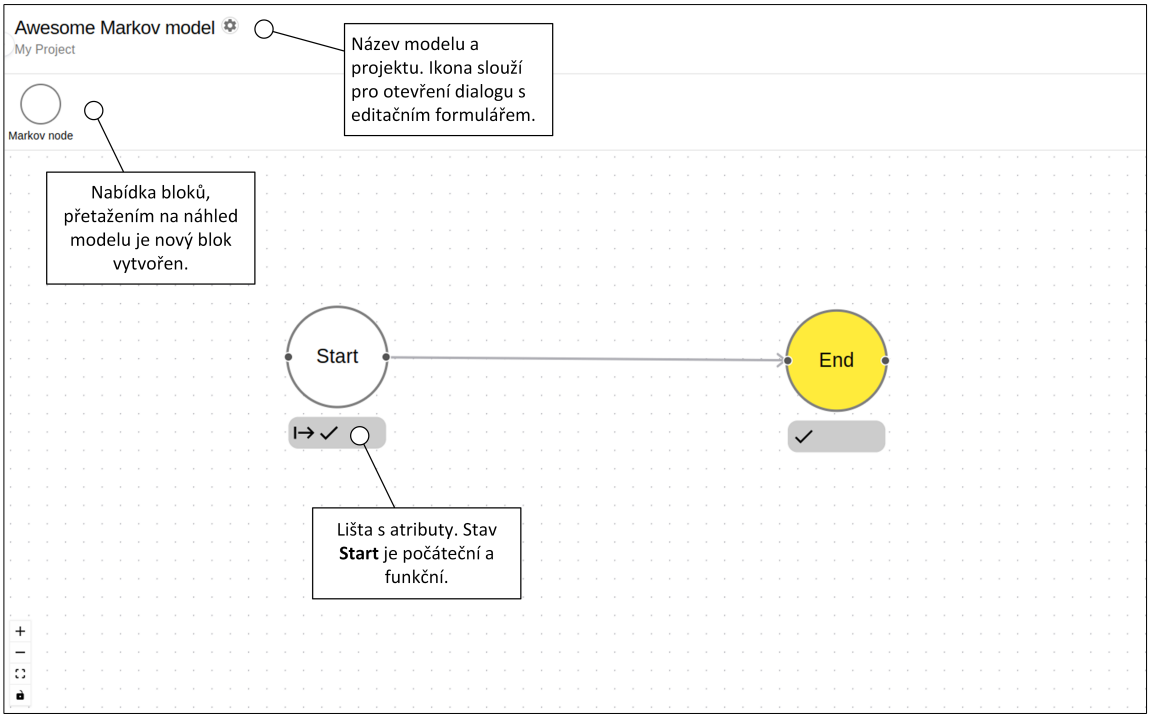
Obrázek A.2 Blokový model

A.3.2 Markovský model

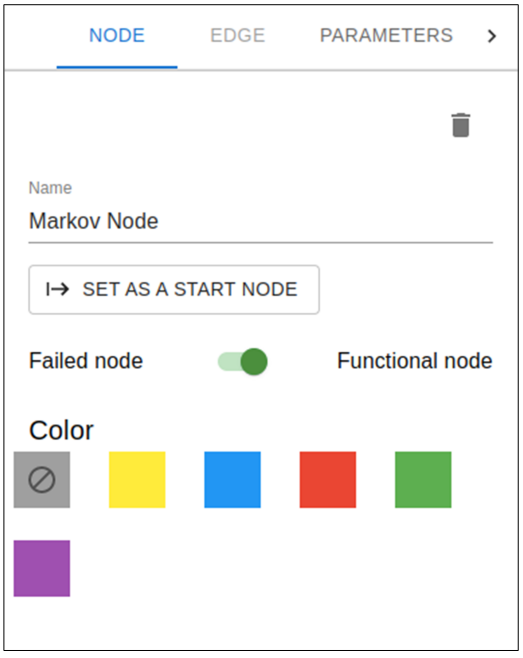
Markovský model se skládá z bloků pouze jednoho typu. Znázorněn je kruhovým stavem s názvem uprostřed a dvěma body pro propojení s dalšími stavy. Opět zde platí, že spojovací bod vlevo je vstupní a vpravo výstupní, hranu nelze vytvořit spojením dvou vstupních nebo dvou výstupních bodů.

Každému stavu může uživatel zadat název a barvu, oboje slouží pro lepší identifikaci a rozdělení stavu do skupin, nenesou žádný význam pro výpočet. Počet barev je omezen na šest základních včetně výchozí bílé. Význam však již ovlivňují nastavení, zda se jedná o počáteční stav a dále nastavení, zda je v daném stavu systém funkční či poruchový. Všechny tyto informace se nastavují v boční liště (záložka *NODE*). Informace o počátečním stavu či funkci nebo poruše stavu jsou znázorněny i v modelu a to pomocí lišty atributů pod samotným stavem. Při výpočtu má dále speciální význam absorpční stav, pro ten není připravena žádná funkce v nastavení. Jako absorpční je automaticky označen každý stav, který nemá žádné výstupní hrany.

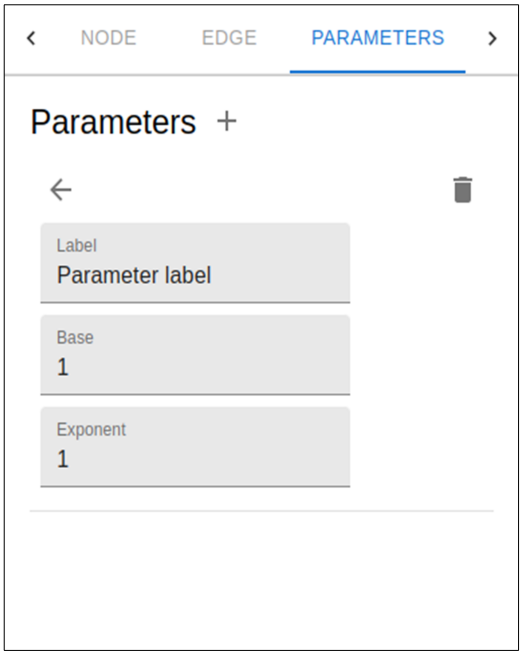
V tomto typu modelu se přiřazují parametry naopak hranám, jedná se o pravděpodobnost přechodu ze zdrojového do cílového stavu dané hrany. K tomu lze ke každé hraně přidat její název sloužící znovu primárně pro orientaci v modelu. Hrany jsou vždy orientované a jejich směr nese význam pro výpočet. Z jednoho stavu do druhého může existovat pouze jedna hrana daného směru. Pokud jsou mezi dvěma stavy dvě hrany musí mít tedy nutně opačnou orientaci.



Obrázek A.3 Markovský model



Obrázek A.4 Boční lišta pro úpravu stavu



Obrázek A.5 Boční lišta pro definici parametrů

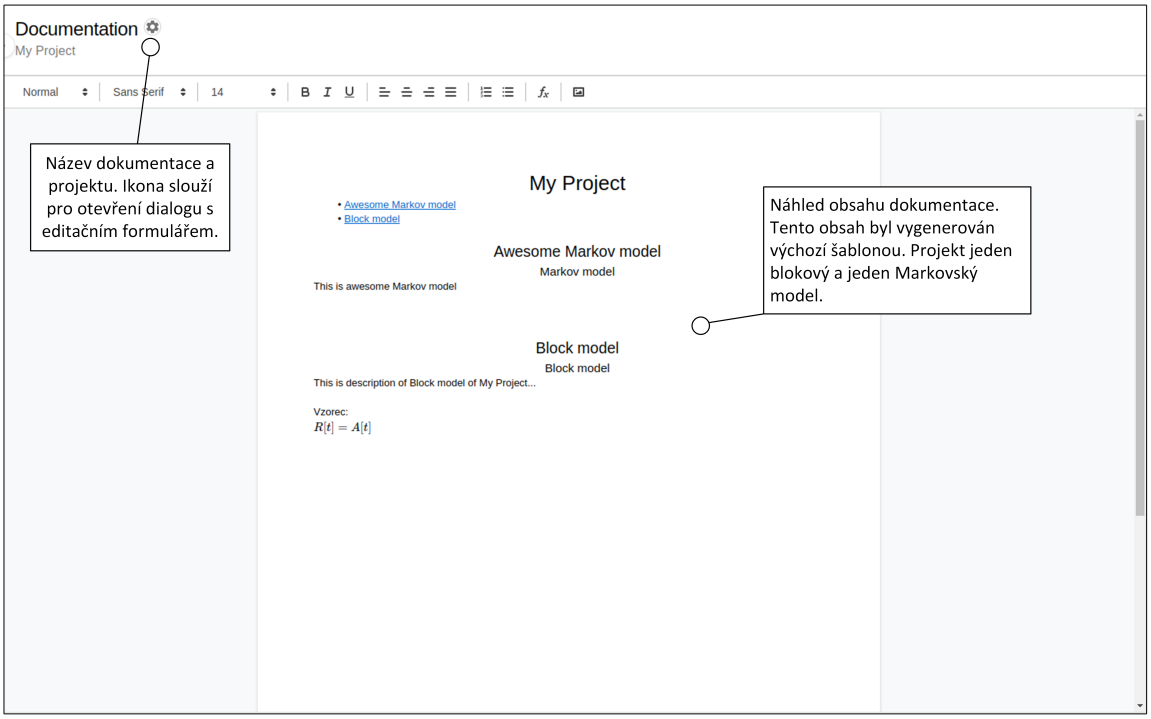
A.4 Dokumentace

Dokumentace je druhým typem položky projektu. Po jejím vytvoření pomocí formuláře (obdobně jako pro model) je uživatel přesměrován na stránku, kde nalezne hlavní část okna s textem dokumentace a lištou pro formátování textu, společně s boční lištou podobnou té pro model. Text dokumentace může být vytvořen dvěma způsoby:

- Ruční sepsání v textovém editoru.
- Vygenerování podle aktuálních modelů projektu.

Pokud uživatel sepisuje dokumentaci manuálně, je pro to připravena lišta s formáty textu, kde najde základní formáty jako je úroveň nadpisu, tučný text, kurzíva, vložení obrázků a podobně.

Pro vygenerování dokumentace je připravena záložka *GENERATE*, která obsahuje tlačítko otevírající dialog s nastavením. V něm najde dostupné šablony a případně jejich dodatečná nastavení. Poté je potvrzením dokumentace vygenerována. **Tímto krokem je dosavadní text v dokumentaci smazán!** Následně je možné text manuálně doplnit či pozměnit. Pro získání textu mimo použití webové aplikace slouží záložka *EXPORT*, kde lze dokumentaci převést do *.docx* dokumentu pro Microsoft Word.



Obrázek A.6 Textový editor dokumentace

Bibliografie

1. HLAVIČKA, Jan. *Diagnostika a spolehlivost*. Vyd. 2. Praha: Vydavatelství ČVUT, 1998. ISBN 80-010-1846-6.
2. HLAVIČKA, Jan. *Číslicové systémy odolné proti poruchám*. 1. vydání. Praha: ČVUT, 1992. ISBN 80-010-0852-5.
3. *Effective Failure Rate of "n"Active Redundant Units, with "m"Required for Success (without Repair)* [online]. New York: Reliability Analytics Corporation, 2010 [cit. 2022-01-10]. Dostupné z: `https://reliabilityanalyticstoolkit.appspot.com/active_redundancy_without_repair`.
4. *Effective Failure Rate of Two Active Redundant Units, with Different Failure Rates, without Repair* [online]. New York: Reliability Analytics Corporation, 2010 [cit. 2022-01-10]. Dostupné z: `https://reliabilityanalyticstoolkit.appspot.com/active_with_different_failure_rates_without_repair`.
5. *MIL-HDBK-217F(N2) Parts Count Prediction Calculator* [online]. New York: Reliability Analytics Corporation, 2010 [cit. 2022-01-10]. Dostupné z: `https://reliabilityanalyticstoolkit.appspot.com/mil_hdbk_217F_parts_count`.
6. *PHP: What is PHP?* [Online]. 2022 [cit. 2022-03-17]. Dostupné z: `https://www.php.net/manual/en/intro-whatis.php`.
7. *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`.
8. *Laravel* [online]. 2022 [cit. 2022-03-18]. Dostupné z: `https://laravel.com/docs/9.x`.
9. *Symfony, High Performance PHP Framework for Web Development* [online]. 2022 [cit. 2022-03-18]. Dostupné z: `https://symfony.com/`.
10. *Nette – Pohodlný a bezpečný vývoj webových aplikací v PHP* [online]. 2022 [cit. 2022-03-18]. Dostupné z: `https://nette.org/cs/`.
11. *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/`.
12. *The Laravel PHP Framework – Web App Construction for Everyone* [online]. 2022 [cit. 2022-04-03]. Dostupné z: `https://kinsta.com/knowledgebase/what-is-laravel/`.

13. *The Most Popular Databases – 2006/2021 - Update May 2021 - Statistics and Data* [online]. 2021 [cit. 2022-04-03]. Dostupné z: https://statisticsanddata.org/data/the-most-popular-databases-2006-2021/.
14. *PostgreSQL: About* [online]. 2022 [cit. 2022-04-03]. Dostupné z: https://www.postgresql.org/about/.
15. *What is MySQL? Everything You Need to Know — Talend* [online]. 2022 [cit. 2022-04-03]. Dostupné z: https://www.talend.com/resources/what-is-mysql/.
16. *React – A JavaScript library for building user interfaces* [online]. 2022 [cit. 2022-03-18]. Dostupné z: https://reactjs.org/.
17. *Vue.js - The Progressive JavaScript Framework — Vue.js* [online]. 2022 [cit. 2022-03-18]. Dostupné z: https://vuejs.org/.
18. *Introduction — Vue.js* [online]. 2022 [cit. 2022-04-03]. Dostupné z: https://vuejs.org/guide/introduction.html.
19. *10 Things You Need to Know About Vue.js Frontend Framework* [online]. 2022 [cit. 2022-04-03]. Dostupné z: https://kinsta.com/blog/vue-js/.
20. *Dokumenty Google* [online]. [B.r.] [cit. 2022-03-18]. Dostupné z: https://docs.google.com/.
21. *Whimsical - Where Great Ideas Take Shape* [online]. 2022 [cit. 2022-03-18]. Dostupné z: https://whimsical.com/.
22. *Web Periodic Background Synchronization API - Web APIs — MDN* [online]. 2022 [cit. 2022-03-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Web_Periodic_Background_Synchronization_API.
23. *Writing WebSocket servers - Web APIs — MDN* [online]. 2022 [cit. 2022-03-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.
24. *Mercure.rocks: Real-time APIs Made Easy* [online]. [B.r.] [cit. 2022-03-30]. Dostupné z: https://mercure.rocks/.
25. *JSON Web Tokens - jwt.io* [online]. 2022 [cit. 2022-03-18]. Dostupné z: https://jwt.io/.
26. *Docker overview — Docker Documentation* [online]. 2021 [cit. 2022-03-18]. Dostupné z: https://docs.docker.com/get-started/overview/.
27. *Full Lifecycle Modeling for Business, Software and Systems — Sparx Systems* [online]. 2022 [cit. 2022-04-11]. Dostupné z: https://sparxsystems.com/products/ea/index.html.
28. *React Flow - Custom Node Example* [online]. 2022 [cit. 2022-01-09]. Dostupné z: https://reactflow.dev/examples/custom-node/.
29. *Overview — React Flow* [online]. 2022 [cit. 2022-03-21]. Dostupné z: https://reactflow.dev/docs/examples/overview/.
30. *CloudFIT* [online]. 2022 [cit. 2022-04-11]. Dostupné z: https://help.fit.cvut.cz/cloud-fit/index.html.
31. *What is Unit Testing? Definition from WhatIs.com* [online]. 2019 [cit. 2022-04-11]. Dostupné z: https://www.techtarget.com/searchsoftwarequality/definition/unit-testing.
32. *PHPUnit – The PHP Testing Framework* [online]. 2020 [cit. 2022-04-08]. Dostupné z: https://phpunit.de/.
33. *Code Coverage Analysis — PHPUnit 9.5 Manual* [online]. 2021 [cit. 2022-04-08]. Dostupné z: https://phpunit.readthedocs.io/en/9.5/code-coverage-analysis.html.

34. *Php - Official Image* [online]. 2022 [cit. 2022-04-11]. Dostupné z: `https://hub.docker.com/_/php`.

35. *Overview of Docker Compose* [online]. 2021 [cit. 2022-04-11]. Dostupné z: `https://docs.docker.com/compose/`.

36. *Dunglas/mercure - Docker Image* [online]. 2022 [cit. 2022-04-11]. Dostupné z: `https://hub.docker.com/r/dunglas/mercure`.