ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Podpora simulace a vizualizace v nástroji DynaCASE
Student:	Jan Blizničenko
Vedoucí:	Ing. Robert Pergl, Ph.D.
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2015/16

Pokyny pro vypracování

DynaCASE je experimentální nástroj pro konceptuální modelování, který byl vyvinut s použitím čistě objektových technologií na platformě Pharo v rámci předmětů BI-SP1 a BI-SP2. Cílem této práce je pokračovat v rozvoji nástroje návrhem a implementací podpory pro simulace a vizualizace konceptuálních diagramů. Návrh proveďte s ohledem na potřebnou obecnost z hlediska současných i možných budoucích konceptuálních metamodelů. Po analýze, návrhu a implementaci proveďte testování formou jednotkových testů a typových příkladů.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof.Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 24. ledna 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Podpora simulace a vizualizace v nástroji DynaCASE

Jan Blizničenko

Vedoucí práce: Ing. Robert Pergl, Ph.D.

12. května 2015

Poděkování

Děkuji všem, kteří mi umožnili tuto práci vytvořit, jakožto i těm, kteří mi pomohli se samotnou prací.

V první řadě děkuji svému vedoucímu práce, Ing. Robertu Perglovi, Ph.D., za cenné podněty během tvorby bakalářské práce a za umožnění vzniku projektu DynaCASE. Simulace by nebylo možné implementovat bez samotné aplikace DynaCASE, tudíž dále děkuji také mým kolegům v týmu stojícím za vývojem tohoto nástoje, zejména Peterovi Uhnákovi. V neposlední řadě patří můj vděk mnoha členům komunity kolem platformy Pharo, jejichž užitečné nástroje i cenné rady mi pomohly s implementací.

Závěrem děkuji svojí rodině a svým přátelům za podporu, a to nejen při tvorbě této práce, ale i během celého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. § 46 odst. 6 tohoto zákona 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 a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 12. května 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2015 Jan Blizničenko. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Blizničenko, Jan. *Podpora simulace a vizualizace v nástroji DynaCASE*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Aplikace DynaCASE je nástroj pro konceptuální modelování pomocí diagramů vyvinutý na platformě Pharo. Cílem této práce je jeho rozšíření o simulace a jejich vizualizaci. Práce obsahuje analýzu diagramů a nástrojů pro jejich simulace a dále návrh takového řešení pro DynaCASE, které umožňuje volitelné využití pro rozličné metamodely. Nakonec se práce zabývá jeho implementací a rozšířením tohoto nástroje o simulovatelné Petriho sítě.

Klíčová slova Diagramy, Modelování, CASE nástroj, Simulace, Petriho sítě.

Abstract

DynaCASE application developed on Pharo platform is a tool for creating diagrams used in conceptual modeling. The aim of the thesis is to add simulations and visualisation to its supported functions. Work focuses on designing solution for DynaCASE to allow optional usage for various metamodels, based on diagrams and simulation tools analysis. Final part consists of implementation and adding simulable Petri nets.

Keywords Diagrams, Modeling, CASE tool, Simulation, Petri nets.

Obsah

Seznam obrázků

Úvod

Představení problematiky

Diagramy

V rozličných oborech lidské činnosti se setkáváme s potřebou modelovat skutečný svět. Existuje velké možství modelovacích notací založených na diagramech. Pro tyto notace existuje spousta modelovacích nástrojů. Některé existující nástroje podporují mnoho potřebných a užitečných funkcí, avšak pouze pro konkrétní dané metamodely, tedy popisy druhů diagramů [1]. Ostatní nástroje jsou naopak obecné a snadno rozšiřitelné o nové metamodely, ale neobsahují dostatek funkcí pro práci s nimi. Z tohoto důvodu vznikla aplikace DynaCASE.

DynaCASE

DynaCASE je experimentální nástroj pro konceptuální modelování vyvinutý na platformě Pharo [2]. Platforma Pharo je kombinací čistě objektového programovacího jazyka založeném na jazyku Smalltalk, virtuálního stroje a vývojového prostředí[3]. DynaCASE se snaží spojit výhody současných modelovacích nástrojů snadnou rozšiřitelností o nové druhy a varianty diagramů, ale i pokročilými funkcemi s možností jejich úprav podle potřeby.

Projekt DynaCASE vznikl v roce 2014 na katedře softwarového inženýrství Fakulty informačních technologií ČVUT jako studentský projekt týmu 4 studentů, jehož jsem byl součástí. DynaCASE je nástupcem nástroje OpenCABE, který byl také vyvíjen v rámci této katedry, avšak vyznačoval se velkou složitostí a nízkou flexibilitou. Důvodem bylo především použití frameworku Eclipse, který byl pro tyto potřeby příliš málo flexibilní, a nedostatečně dynamických vlastností programovacího jazyka Java.

DynaCASE v současné době podporuje tvorbu diagramů OntoUML a stavových automatů.

Pro zobrazování je použit vizualizační balíček Roassal [4], který slouží k vykreslování dvourozměrné grafiky založené na elementech spojených hranami. Roassal je nezávislou součástí Moose [5], což je nástroj pro analýzu softwaru a dat, jehož důležitou součástí je právě vizualizace. Roassal i Moose jsou vyvíjeny ve Pharu.

Simulace

Část modelovacích notací využívajících diagramy slouží k modelování procesů. Obsahují tedy kromě statické struktury také informace o možném ději. Pro snadnější představu modelovaného procesu je vhodné mít možnost tento děj simulovat, tedy znázornit jeho průběh.

Další variantou simulace je výpočet statistických údajů při realizaci modelovaného procesu, kdy je vhodné získat hodnoty co nejrychleji. V takovém případě nejen že průběh není dobré znázorňovat, ale může pak být vhodnější použít přímé matematické metody nad informacemi získanými z diagramu. K této variantě simulací se tato práce přímo nevztahuje.

Cíl práce

Cílem práce je rozšíření nástroje DynaCASE o podporu simulací a vizualizace těchto simulací. Návrh musí být proveden s ohledem na potřebnou obecnost pro současné i možné budoucí modelovací notace. Součástí práce bude analýza, návrh, implementace a vypracování jednotkových testů a typových příkladů. Výsledné rozšíření má umožnit použití simulací u libovolného metamodelu, avšak nesmí být vyžadováno. Pro ověření bude součástí cílů také rozšířit DynaCASE o Petriho sítě a umožnit jejich simulaci.

Metodika

Samotnému návrhu a implementaci předchází rešerše simulovatelných modelovacích notací společně s některými nástroji pro jejich simulaci.

Práce pokračuje popisem struktury DynaCASE, dále návrhem struktury částí pro simulace a výčtem možností pro integraci do DynaCASE. Po určení hlavní struktury následuje detailní návrh a implementace samotných tříd umožňujících simulaci. Poté se práce věnuje integraci s grafickým rozhraním DynaCASE.

Další částí je návrh a implementace Petriho sítí do DynaCASE, pro které bude také implementováno umožnění simulace.

V průběhu práce bylo nutné testovat správnost implementace, a proto se také věnuji popisu testování.

Poslední část je věnována možnostem budoucího vývoje a použití této práce.

Kapitola 1

Analýza modelovacích notací a programů

Tato kapitola se věnuje rešerši známých simulovatelných notací a také programů, které tyto notace mohou simulovat. Cílem této části není detailní popis všech součástí, pravidel a principů, ale utvoření obecného přehledu.

Nejrozsáhlejší část je věnována Petriho sítím. Pro Petriho sítě existuje mnoho variant a rozšíření [6], které mohou mít na simulaci značný vliv, a proto z pouhých rozšíření Petriho sítí lze získat rozsáhlou představu o využití simulací. Dalším důvodem je skutečnost, že návrh a implementace Petriho sítí pro nástroj DynaCASE je součástí této práce, a proto je vhodné počítat s možným rozšířením.

1.1 Petriho sítě

Petriho sítě jsou univerzálním matematickým a vizuálním konceptem pro modelování a analýzu procesů. Nejčastěji jsou využívány pro modelování počítačových systémů, zejména průběhu paralelních procesů a toků dat v síti [7][8].

1.1.1 Základ Petriho sítí

Základem Petriho sítí je orientovaný, bipartitní, ohodnocený graf. Vrchol je buď místem (angl. *place*), které reprezentuje podmínky, a nebo přechodem (angl. *transition*), který reprezentuje událost [9]. Místa jsou graficky reprezentována elipsou, přechody obdélníkem. Každé místo je označeno celým nezáporným číslem k, přičemž říkáme, že dané místo obsahuje k tokenů (někdy také *značek* nebo *teček*), značených jako k černých teček uvnitř elipsy. Stav Petriho sítě je dán počty tokenů v jednotlivých místech. Máme-li tedy 5 míst, stav je tvořen pěticí číselných hodnot. Tento popis vychází z [7][8][10][11][12], kde lze nalézt podrobnější definice a příklady použití.

Hrana (angl. *arc*) vede buď z vrcholu do místa nebo z místa do vrcholu. Je ohodnocena vahou (angl. *weight*) a značena šipkou. Pokud má navíc váhu vyšší než 1, zpravidla se u šipky uvádí číselná hodnota váhy [9]. Hrana je jednoznačně identifikována uspořádanou dvojicí vrcholů, které spojuje [7][8]. Nelze tedy mít více hran z jednoho místa do jednoho přechodu nebo naopak, ale je možné mít jednu hranu z místa do přechodu a druhou z přechodu do místa.

U každého přechodu můžeme hovořit o jeho vstupních a výstupních místech, kde vstupní jsou ta, z nichž vede hrana do daného přechodu (vstupní hrana), výstupní naopak z daného přechodu do místa (výstupní hrana).

Každý přechod je a nebo není proveditelný (angl. *enabled*) v závislosti na vstupních hranách a místech. Místo pojmu *proveditelný* se v některé literatuře vyskytuje pojem *uschopněný*, na význam to však vliv nemá. Proveditelný je za podmínky, že neexistuje žádné vstupní místo, které má méně tokenů, než je váha příslušné vstupní hrany.

Pokud je poveditelný, může být proveden (provést – angl. *fire*) tak, že je ze všech vstupních míst odstraněno tolik tokenů, jakou má příslušná vstupní hrana váhu. Zároveň je do výstupních míst je přidáno tolik tokenů, jakou má příslušná výstupní hrana váhu.

Petriho sítě nejsou vždy deterministické. Pokud máme dva přechody se stejným vstupním místem, které obsahuje jediný token, potom provedení jednoho přechodu způsobí, že druhý přestane být proveditelný, přičemž to, který z nich to bude, nemusí být předem určeno.

Pro potřeby simulace je tedy potřeba postupně provádět přechody, čímž se mění stavy vstupních a výstupních míst. Vzhledem k nedeterminističnosti nelze vždy stanovit pevné pravidlo o pořadí provádění. Provádění může být buď náhodné nebo určené externím dějem nebo rozhodnutím. Příkladem náhodného pořadí může být simulace toku paketů v síti, kde je známa pravěpodobnost, s jakou dojde ke ztrátě paketu. Kdy k jedné konkrétní ztrátě paketu dojde je však děj považovatelný za náhodný, proto předem nevíme, zda provést přechod znamenající průchod paketu do cílového zařízení a nebo přechod znamenající ztrátu paketu po cestě.

1.1.2 Rozšíření a varianty

Aby Petriho sítě umožnily modelovat větší množství procesů, bylo vytvořeno několik rozšíření a úprav Petriho sítí. Každé z těchto rozšíření může znamenat i jiný průběh simulace, a proto je nutné se s nimi obeznámit. Následuje jejich přehled.

1.1.2.1 Nové hrany

Pro Petriho sítě byly vytvořeny nové hrany [13][14][15][16], které slouží buď k rozšíření modelovacích možností nebo ke zpřehlednění grafu.

První hranou je hrana ihnibiční (angl. *inhibitor*), která je variantou vstupní hrany. Přechod je proveditelný právě tehdy, pokud jsou splněny podmínky standardních vstupních hran a navíc pro všechny vstupní inhibiční hrany platí, že příslušná vstupní místa neobsahují žádné tokeny. Při provedení přechodu se z příslušných vstupních míst neodeberou žádné tokeny. Inhibiční hrany tedy nemohou mít váhu.

Další vstupní hranou je hrana nulovací (angl. *reset*). Tato hrana nemá žádný vliv na proveditelnost přechodu, takže pokud má například přechod pouze nulovací vstupní hrany, tak je vždy proveditelný. Při provedení přechodu se však z příslušného vstupního místa odeberou všechny tokeny. Ani nulovací hrany tedy nemají váhu.

Testovací (angl. *test* nebo *testing*) hrana slouží pouze ke zjednodušení návrhu a nepřidává žádnou novou funkčnost. Na proveditelnost přechodu mají normální a testovací hrany stejný vliv, avšak pokud tento přechod provedeme, testovací hrany nijak nezmění množství tokenů ve svých příslušných místech. Kdybychom mezi místem a přechodem vytvořili vstupní i výstupní hranu, čímž bychom vytvořili smyčku složenou z místa, přechodu a dvou hran, získáme funkci totožnou s funkcí testovací hrany. Všechny tokeny v tomto místě totiž projdou přes přechod zpátky do tohoto místa, takže se počet tokenů ve výsledku nijak nezmění.

Je patrné, že pro tato rozšíření nestačí jedna jednoduchá podmínka v přechodu použitá postupně na všechny vstupní místa a hrany, ale je potřeba rozhodnutí na základě typů hran.

1.1.2.2 Kapacita míst

V základních Petriho sítích, ani s výše zmíněnými hranami, nemají výstupní hrany ani místa žádný vliv na proveditelnost. Toto se mění při zavedení kapacity míst [7][17]. Ke každému místu může být přiřazena kapacita – tedy maximální počet tokenů v místě – a pokud by provedení přechodu znamenalo zvýšení počtu tokenů některého výstupního místa nad kapacitu, pak tento přechod není proveditelný.

1.1.2.3 Barevné petriho sítě

Barevné (angl. *colored*) Petriho sítě dávají tokenům nový druh informace, a to barvu [7][8][11]. Jednotlivé tokeny tedy mohou reprezentovat různé objekty. Stav každého místa již není reprezentován jedinou hodnotou – počtem tokenů, ale hodnotami pro každou barvu zvlášť. Podobná změna se týká hran. Zatímco nyní měly hrany váhu, tedy určení počtu odstraněných nebo vytvořených tokenů, nyní může mít každá hrana předpis pro každou barvu [18]. Přechody pak navíc mohou obsahovat komplexnější podmínky [6].

Místo dotazování na obecnou váhu hran a obecný počet tokenů v místě se obdobnou metodou ptáme na větší množství těchto hodnot. Tyto hod-

noty však musí být definovány včetně množiny druhů barev, což je rozdílné pro každý konkrétní diagram.

1.1.2.4 Petriho sítě s prioritami

Petriho sítě s prioritami (angl. *Petri nets with priorities*), nazývané též prioritní nebo prioritizované (angl. *priority* nebo *prioritized*), přidávají priority jednotlivým přechodům [17]. Proveden je pak vždy proveditelný přechod s nejvyšší prioritou. Pokud existuje více proveditelných přechodů se stejnou prioritou, není určeno, který z nich se má provést.

1.1.2.5 Časované Petriho sítě

Dynamičtější a šířeji použitelnější variantou priorit je přidání času. Výsledným Petriho sítím se říká časované (angl. *timed*), občas také časové (angl. *time*) [11][7][12][19]. Časované Petriho sítě však nejsou jedním konkrétním modelem nebo rozšířením, jelikož, ač jsou samy variantou Petriho sítí, mají samy několik variant, které se mohou prolínat, jelikož běh času i časové podmínky mohou být spojeny se všemi možnými prvky Petriho sítě a mohou být určeny i globálně pro celou síť [17][8]. Pojem času obecně bývá reprezentován nezáporným celým číslem. Je však možné použít jakýkoliv model času, například minuty se sekundami.

Pokud dojde ke splnění časových podmínek společně se standardními podmínkami Petriho sítí, pak to může znamenat buď možnost nebo nutnost provést přechod. Čas se posune při provedení přechodu, ale je možné uvažovat i o variantě s neustálým plynutím času podle skutečnosti, případně v určitém měřítku ke skutečnosti. Nemáme tu tedy pouze informace v jednotlivých místech, které nejsou nijak přímo závislé, pokud nemají společný přechod, ale děje se zde i globální posun času, který může mít vliv na všechna existující místa.

Pro potřeby simulací to znamená, že při provedení jednoho jediného přechodu se mohou změnit hodnoty i v prvcích, které by jinak s tímto přechodem nebyly nijak spojeny. Může také být potřeba získat informace ze samotného diagramu. Rovněž počet tokenů již nemusí být možné vyjádřit pouhou jedinou hodnotou, ale buď uspořádanou n-ticí časových hodnot a nebo dokonce samostatnými objekty uchovávajícími informace.

1.1.2.6 Stochastické Petriho sítě

Ačkoliv ani časované Petriho sítě nejsou deterministické, tak lze onu nedeterminističnost ještě zvýšit použitím speciálního typu časovaných Petriho sítí, a to stochastických (angl. *stochastic*). Ty nahrazují pevné hodnoty časů náhodným generováním těchto hodnot v průběhu simulace, explicitně tedy pracují s pravděpodobností [7][8]. Tyto náhodné hodnoty lze generovat z různých rozdělení podle potřeby konkrétního diagramu.

1.1.2.7 Hierarchické Petriho sítě

Pro zpřehlednění a vyšší znovupoužitelnost byly zavedeny hierarchické (angl. *hierarchical*) Petriho sítě. Každý přechod tady můžeme nahradit jinou Petriho sítí, tedy podsítí [11][17][20][21]. Tyto podsítě mají navíc definováno napojení na vnější síť.

Ačkoliv to ve většině literatury není uváděno, tak podle [17] lze použít hierarchii i u míst.

Zpřehlednění spočívá v možnosti zakreslit celou část sítě jako jediný přechod, kdežto znovupoužitelnost je umožněna díky tomu, že stejnou podsíť můžeme použít na několika různých místech. Při simulaci v takovém případě vznikne několik výskytů těchto podsítí, každá s vlastním stavem.

1.1.2.8 Petriho sítě vysoké úrovně

Pojem Petriho sítě vysoké úrovně (angl. *high-level Petri nets*), někdy také *vysokoúrovňové Petriho sítě*, je používán pro takové Petriho sítě, které obsahují kombinaci několika výše uvedených rozšíření. Nejčastěji jde o použití hierarchie, času a barev [7][11][21]. Mohou mít však některé prvky zcela navíc, například globální proměnné používané a měněné během provádění přechodů.

1.1.2.9 Objektové Petriho sítě

Objektové (angl. *object*), někdy také nazývané *objektově orientované* (angl. *object oriented*), Petriho sítě mění tokeny, jakožto pasivní informaci vycházející z číselné hodnoty, na samostatné objekty [6][22][23]. Každý token tedy může mít svůj vlastní stav, může putovat sítí a při každém provedení přechodu, kdy je s ním manipulováno, může tento svůj stav změnit. Pro síť, ve které se nachází, se chová podobně, jako barevné Petriho sítě, tudíž na základě jeho stavu můžeme rozhodovat o proveditelnosti a výsledku provedení přechodu. Samotný token však obsahuje vlastní Petriho síť nebo dokonce vlastní proměnné a pokročilou logiku [6], podle čehož určuje svůj vnější stav.

1.1.3 Aplikace pro simulaci Petriho sítí

Pro Petriho sítě vzniklo mnoho desítek nástrojů, z nichž velké množství umí provádět simulace. Seznam mnoha z nich lze nalézt na webových stránkách University of Hamburg [24]. Následuje stručný popis vybraných nástrojů, společně s vyjmenováním jejich funkcí.

1.1.3.1 CPN Tools

CPN Tools [25] podporuje hierarchické, barevné, časované a stochastické Petriho sítě [20]. Obsahuje inhibiční a nulovací hrany. Vyznačuje proveditelnost přechodu barevným zvýrazněním, tokeny zobrazuje pomocí číslených hodnot s jejich případnými dalšimi údaji, jako je časování.

Umožňuje provést konkrétní přechod přímo z GUI, spustit opakování, kdy se z proveditelných přechodů vybere vždy náhodně jeden [26], případně zobrazit přímo výsledek několika takových provedení přechodů. V obou těchto případech si lze vybrat, kolik opakování se má provést a s jakou prodlevou mezi jednotlivými provedeními.

Jako jediný ze zkoumaných nástrojů umožňuje provádět simulaci průběžně, tedy nezakazuje úpravu diagramu během běhu simulace. To je užitečné vzhledem k možnosti ručních úprav stavu. Nevýhodou je, že při změně struktury simulovaného diagramu, tedy přidání nebo odebrání prvků, dojde zpravidla k chybě programu, následkem čehož přestane reagovat.

V dokumentaci CPN Tools [18] je uveden příklad, kdy je definována množina barev *dice*, tedy kostka, s hodnotami 1 až 6. Provedením přechodu dojde k vygenerování 5 kostek do výstupního místa. Odtud vedou dva další přechody, z nichž jeden se provede pouze pokud v tomto místě je 5 totožných barev, tedy 5 kostek se stejnou hodnotou, druhý pak v opačném případě a umožní hod opakovat.

1.1.3.2 ExSpecT

ExSpecT umožňuje simulaci hierarchických, barevných, časovaných a stochastických Petriho sítí a obsahuje testovací hrany [27]. Ve stavech lze ukládat i komplexnější infomace, jako je například soubor textových dat nebo jakýkoliv jiný objekt programovacího jazyka. Je možné v něm definovat prvky, které musí být převeditelné na existující prvky Petriho sítí, čímž zjednodušují návrh opakujících se částí [28].

Funkčnost nástoje nebylo možné prakticky ověřit, jelikož při pokusu o jeho spuštění došlo vždy k chybě a následnému ukončení programu.

1.1.3.3 PIPE 2

Platform-Independent Petri Net Editor 2 (PIPE 2) [29] umožňuje simulaci časovaných Petriho sítí a obsahuje kapacity míst a inhibiční hrany.

Jedním způsobem simulace je jednorázový matematickým výpočet, který vygeneruje výslednou statistiku. Dalším je animace, kde uživatel kliknutím na proveditelný přechod způsobí jeho provedení nebo může spustit opakované provádění náhodných stavů.

1.1.3.4 WoPeD

WoPeD [30] simuluje hierarchické Petriho sítě. Pro ně má velmi dobrou podporu, jako je možnost vnoření nebo přeskočení podsítě během simulace. Je zaměřený převážně na modelování firemních procesů. Umožňuje generování statistik při okamžitých simulacích, získání údajů vhodných pro firemní procesy a také opakované provádění náhodných stavů i konkrétních stavu pomocí kliknutí myši.

1.1.3.5 TAPAAL

Nástroj TAPAAL [31] je inspirovaný programem PIPE 2. Má podobné uživatelské rozhraní i některé funkce. Je zaměřený na časované Petriho sítě. Oproti PIPE 2 však obsahuje některé další užitečné prvky. Hlavním z nich je sdílení míst a přechodů mezi jednotlivými sítěmi, takže i během simulace mají totožné počty tokenů. Díky tomu obsahuje možnost hierarchie, byť jinou, než je definována v hierarchických Petriho sítích. Dále obsahuje transportní hrany [32], které vedou od místa, přes přechod, do dalšího místa a vyznačují se tím, že při použití časování nezmění čas náležící tokenu, který přenesou ze vstupního do výstupního místa.

1.1.3.6 TINA

Posledním zkoumaným nástrojem je TINA – *TIme petri Net Analyzer* [33]. Na rozdíl od předchozích nástrojů zaujme jednoduchostí grafického rozhraní, jelikož je složen z jednotlivých komunikujících nástrojů a aplikace tvořící uživatelské rozhraní neumožňuje pracovat se všemi podporovanými prvky, jako je například inhibiční hrana. Obsahuje rozšíření v podobě časovaných Petriho sítí a priorit. Umožňuje však tvorbu statistik ze simulací založených na okamžitých výpočtech a i provádění konkrétních přechodů kliknutím myši.

1.2 Stavové automaty

1.2.1 Principy stavových automatů

Stavové automaty jsou matematickým modelem pro výpočetní procesy, prakticky používané například pro analýzu vstupu nebo pro generování výstupu pro ovládání strojů, elektroniky nebo i objektů programovacího jazyka, jejichž chování lze rozdělit do jednotlivých stavů [34][35].

Skládají se z množiny stavů, vstupních symbolů, přechodových funkcí, které určují, do jakého stavu přejít z aktuálního stavu a vstupního symbolu, dále z označení počátečního stavu a z množiny koncových stavů.

Pokud je počet stavů konečný, mluvíme o konečném stavovém automatu. Vzhledem k tomu, že nekonečný počet stavů nelze diagramy vyobrazit ani je vizuálně simulovat, budu dále psát pouze o konečných automatech.

Z pohledu diagramu se jedná o stavy a přechody. Některé stavy mají speciální význam, konkrétně počáteční a koncový stav. Přechody propojují tyto stavy. Počáteční stav může být znázorněn pouhou šipkou vedoucí z prázdného prostoru do tohoto stavu. Lze se také často setkat se zakreslením speciálního pseudostavu – malého černého kolečka, ze kterého vede přechod do stavu považovaného za počáteční. V takovém případě se však tento pseudostav, tedy toto kolečko, běžně nazývá *počátečním stav*, přestože ve skutečnosti stavem není. Při simulaci automat začne s aktivním počátečním stavem (skutečným stavem, nikoliv pseudostavem) a postupně přechází mezi stavy na základě vstupu. Do

jakého stavu přejít jednoznačně určuje to, jaká je přechodová funkce z aktuálního stavu pro daný vstup.

Automatem pro analýzu vstupu je tento vstup přijat ve chvíli, pokud po jeho celém zpracování je aktivní koncový stav. Pokud automat slouží pro řízení nebo ovládání, potřebuje kromě vstupu i výstup. Výstup je určen posledním vykonaným přechodem v případě automatu Mealyho typu nebo aktivním stavem v případě automatu Moorova typu [34].

Znamená to tedy, že během simulace přichází v každém kroku vstupy, na základě kterých se deaktivuje jeden stav a aktivuje další nebo opět ten samý. Zároveň při každém přechodu může vygenerovat výstup. Vstup, a tedy i výstup, se může lišit při každém jednotlivém běhu.

Automat může být buď úplný nebo neúplný. Neúplný nemá pro všechny možné vstupy a stavy definovány přechody. Pokud nastane situace, že přijde vstup, pro který přechod není definován, přestane být aktivní jakýkoliv stav [36] a praktický důsledek záleží na významu automatu.

1.2.2 Nedeterministický konečný automat

V předchozí části byl popsán automat s jednoznačným určením přechodu na základě stavu a vstupu, takový automat se nazývá deterministický. Další variantou je nedeterministický. Takový může mít pro jednu kombinaci vstupu a stavu definováno více přechodů, tudíž je možné změnit aktivní stav na jeden z více stavů a nelze o výběru stavu předem rozhodnout. Při samotném použití tedy automat zvolí ty přechody, které vedou k nejlepšímu pokračování automatu, tedy například k přijmutí celého vstupu. Dále může nedeterministický automat obsahovat epsilon přechody, tedy přechody proveditelné i bez vstupu a existují i varianty s více počátečními stavy. Epsilon přechody i další počáteční stavy se dají úpravami odstranit bez změny významu automatu.

Prakticky toho lze dosáhnout následujícími způsoby. Jednou možností je, že o výběru rozhodujeme na základě předchozí analýzy všech možných cest grafem, ze kterých se vybere ta nejlepší. Druhá možnost spočívá v postupném vytváření kopií grafu pro každou možnou cestu, kde každá kopie bude mít vlastní aktivní stav, případně v možnosti mít více aktivních stavů [36]. To může být vhodné pro simulaci nebo analýzu vstupu. Nejrychlejší, ale nejméně užitečnou, možností je rozhodování na základě náhody.

1.2.3 Hierarchický stavový automat

Hierachický stavový automat je rozšíření, při kterém každý stav může obsahovat vlastní automat. Pokud se tedy aktivuje stav obsahující vlastní automat, aktivuje se i stav tohoto vnitřního automatu. Tak jako normální stavové automaty i tyto vnitřní automaty musí mít počáteční stav [35][37][38].

1.2.4 UML Stavový automat

UML Stavový automat je kombinací deterministických automatů Mealyho i Moorova typu s podporou hierarchie [39][40]. Je tedy možné definovat libovolnou akci na základě aktivace stavu nebo provedení přechodu. Definuje takzvané události, jako je například přijetí signálu, uplynutí času nebo změna stavu libovolného booleovského výrazu. Umožňuje definovat proměnné nad celým stavovým automatem, které se mohou změnit na základě aktivace stavu nebo provedení přechodu. Podle kombinace typu události a proměnných se může provést příslušný přechod. Přechod může být rozdělen na části, kdy se v první části definuje událost a v druhé se větví podle podrobnějších podmínek.

Hierarchii automatů vylepšuje schopností každého stavu obsahovat více automatů běžících současně. Tyto automaty se neovlivňují přímo, ale mohou mít společné proměnné.

1.2.5 Aplikace pro simulaci stavových automatů

1.2.5.1 Automata Editor

Automata Editor [41] podporuje deterministické i nedeterministické konečné automaty. Umožňuje provádět simulace pomocí krokování, a to po jednotlivých krocích nebo automatickým během s možností nastavit rychlost a je možné se v simulaci i vrátit do předchozího stavu. Nedeterministické chování řeší tím, že umožní více stavům najednou, aby byly aktivní. Jakmile pro daný vstup není definován přechod, stav přestane být aktivní a pokud existují další aktivní stavy, simulace pokračuje. Animace je provedena tak, že aktivní stavy jsou braveně zvýrazněny a při provádění přechodu mezi stavy plynule putuje malý kroužek podél hrany.

1.2.5.2 Visual Automata Simulator

Visual Automata Simulator [42] umožňuje vytvořit deterministický nebo nedeterministický konečný automat. Po spuštění simulace se zobrazí výsledek, tudíž zda automat přijal nebo nepřijal vstup. Neobsahuje animaci pomocí krokování mezi stavy.

1.2.5.3 Enterprise Architect

Enterprise Architect [43] je komerční nástroj pro modelování UML a BPMN diagramů. Součástí je i simulátor UML Stavových automatů [44].

Umožňuje automatické i manuální krokování průběhu automatu včetně plné podpory hierarchie, kdy lze vnitřní automaty provádět nebo přeskakovat. Lze definovat vlastní proměnné, podmínky pro přechody a akce při provedení přechodů nebo aktivaci stavů. V průběhu zvýrazňuje aktuální a možné následující stavy.

Dále je možné použít vlastní grafické uživatelské rozhraní, tedy například okno se zobrazením výstupů a s tlačítky schopnými změnit proměnnou diagramu [44].

1.3 UML diagramy

UML – Universal Modeling Language, je souborem diagramů určených pro modelování informačních systémů a dalších programů [45][46].

1.3.1 Diagram aktivit

UML diagram aktivit se skládá z uzlů spojených hranami a slouží k zachycení sekvence aktivit vedoucích od počátku k cíli [47]. Uzly se dělí na akční, řídící nebo objektové. Akční uzly se dělí na akce a aktivity, kde akce je nedělitelná jednotka vyjadřující určitou část činnosti, kdežto aktivita se může skládat z několika akcí. Hrany se dělí na řídící, které určují přechody mezi akčními uzly a využívají k tomu řídící uzly, a na objektové, které znázorňují předání objektu mezi dvěma akčními uzly.

Principielně jde o definici speciálních významů nad Petriho sítěmi, což umožňuje jasná pravidla pro simulaci vycházející právě z Petriho sítí [48].

1.3.2 Sekvenční diagram

Sekvenční diagram popisuje časově uspořádanou posloupnost zasílání zpráv mezi objekty v rámci jedné činnosti. Skládá se z objektů zobrazených jako vertikální přímky vedle sebe, mezi nimiž v určité výšce probíhají zprávy znázorněné šipkami. Diagram je poté vyhodnocován po jednotlivých zprávách odshora dolů.

1.3.3 Diagram interakcí

Diagram interakcí je kombinací diagramu aktivit a sekvenčního diagramu. Hlavní struktura je založena na diagramu aktivit, kde se jednotlivé aktivity mohou skládat z vlastních sekvenčních diagramů.

1.3.4 Nástroj Enterprise Architect

Modelovací nástroj Enterprise Architect [43] byl již zmíněn v souvislosti s UML stavovými automaty. Umožňuje simulovat také diagram aktivit a sekvenční diagram [49]. Tak jako v případě UML stavových automatů je možné simulaci ručně krokovat nebo spustit automatické krokování, tedy běh simulace. Barevně zvýrazňuje vždy aktuálně aktivní část diagramu nebo ztmavuje ostatní části.

V současné verzi neumožňuje simulaci diagramu interakcí [43].

1.4 BPMN

1.4.1 Popis

Business Process Modeling Notation, zkráceně BPMN, je notací principielně podobnou UML diagramu aktivit [50][51], přičemž je více zaměřena na modelování firemních procesů [52][53][54]. BPMN je výrazně komplexnější, obsahuje více typů uzlů i hran s odlišnou sémantikou. Zatímco diagram aktivit má určený jeden počáteční uzel, BPMN umožňuje běh několika simultánních procesů ve zvláštních částech diagramu, takzvaných kontextech (angl. *swim lanes – plavecké dráhy*), každý s vlastní počáteční událostí. Tyto simultánní procesy se mohou vzájemně ovlivňovat pomocí spojení uzlů hranami napříč oběma částmi diagramu. Například může jeden spustit druhý nebo jeden čekat na aktivaci určitého stavu v druhém. BPMN tedy lépe umožňuje najednou modelovat více procesů, které mají vlastní životní cyklus, ale vzájemně se ovlivňují.

1.4.2 Nástroje

1.4.2.1 Enterprise Architect

Enterprise Architect [43] dokáže simulovat kromě UML diagramů i BPMN [49][55]. Probíhá podobně, jako v případě UML diagramu aktivit. Je to možné však jen v některých variantách tohoto programu [49]. Z tohoto důvodu jsem nemohl prakticky ověřit jeho praktickou funkčnost.

1.4.2.2 BPMN Simulator

BPMN Simulator [56] neumožňuje diagramy modelovat, ale přijímá definice BPMN diagramů ve formátu BPMN 2.0 XML. Tyto importované diagramy poté dokáže simulovat. Na rozdíl od Enterprise Architectu je zde možné spustit několik průběhů simulací najednou a to dynamicky během již běžící simulace. Během simulace putuje grafem symbol hvězdy. Pokud se nad některým prvkem grafu nachází hvězda, pak je tato část barevně zvýrazněna. Hvězda buď leží v určitém uzlu a nebo se plynule posouvá po hraně. Příkladem využití simulace více průběhů může být zpracování poštovních zásilek, kde pošta také zpracovává mnoho zásilek současně. V případě, že se někde nachází nutnost rozhodnout, kterou cestou v grafu se dále vydat, je možné zaškrnout, která z cest se má používat. I to je možné udělat během již běžící simulace.

Některé prvky však nepodporuje úplně. Například neumožňuje simulovat situaci, kdy je definováno, že pro pokračování na další aktivitu musí být aktivita z jiného kontextu právě aktivní.

1.4.2.3 Simul8

Simul8 je nástroj umožňující simulaci BPMN, včetně podrobného nastavení chování při rozhodnutí, kterou cestou grafem pokračovat. Také je v něm možné definovat vlastní grafickou podobu jednotlivých prvků [57]. Nemohl jsem však ověřit jeho praktickou funkčnost, protože se jedná o komerční software.

1.4.2.4 Visual Paradigm

V programu Visual Paradigm [58] je před spuštěním simulace potřeba nastavit jeden nebo více scénářu, tedy cest grafem BPMN. Ke každému scénáři je možné určit procentuální pravděpodobnost, že nastane. Poté lze scénáře provést pomocí simulace [59]. Není však možné spustit paralelní průběh více událostí během jediné simulace.

1.5 BORM

Business Object Relation Modeling, zkráceně BORM, je, oproti BPMN, jednoduchá notace založená na dvou diagramech. Object relations diagram a behavior analysis diagram. První z nich lze simulovat, jelikož slouží k modelování děje z pohledů jednotlivých oddělených účastníků, kteří si mezi sebou v průběhu jejich činnosti předávají zprávy [54][60]. Účastník může čekat na zprávu od druhého před možností pokračovat ve svých akcích. Účastník, jakožto objekt, obsahuje orientovaný bipartitní graf složený z aktivit a stavů, které jsou propojeny přechody. Přechodu mezi jednotlivými účastníky se říká komunikace. Jeden stav je definován jako počáteční a další mohou být definovány jako konečné. Na základě provádění aktivit se mění stavy.

1.5.1 Nástroje

1.5.1.1 BORM Portal

Webový nástoj BORM Portal [61] vyvinutý v Seaside, tedy webové platformě také určené pro prostředí Pharo, obsahuje možnost simulace po krocích s možností opakování. Aktivní prvky zvýrazňuje změnou barvy. Při potřebě rozhodnutí zobrazí dialogové okno s dotazem, kudy pokračovat. Je možné vybrat více možností, přestože mohou mít výlučný význam. Simulace poté probíhá oběma směry zároveň.

1.5.1.2 Craft.CASE

Nástoj Craft.CASE [62] umožňuje simulaci pomocí jednotvých kroků nebo jejich opakování, kdy je jako jeden krok změna aktivního prvku ve všech účastnících, je-li to možné. Během krokování nedochází k přímým krokům mezi aktivitami, ale je zde jakýsi mezikrok, kdy je barevně zvýrazněný přechod

společně s předchozí aktivitou. V případě nutnosti rozhodnutí je zobrazeno dialogové okno s dotazem, která z podmínek platí, a tudíž kudy pokračovat. Je možné si toto nastavení zapamatovat a spustit již nastavenou simulaci. U podmínek lze vybrat více možností, přestože mohou mít výlučný význam. Aktivní stavy jsou zvýrazněny změnou barvy, právě aktivovaný stav potom ještě rychlým bliknutím okraje prvku. Craft.CASE existuje ve formě desktopové nebo webové aplikace [62], přičemž tento text popisuje chování aplikace webové.

1.6 DEMO

Design & Engineering Methology for Organisations, zkráceně DEMO, je notace složená z 5 druhů vzájemně propojených diagramů, tvořících dohromady popis procesů jedné organizace [63]. Každý z těchto diagramů poskytuje jiný druh náhledu, od celkového přehledu možných interakcí podniku s třetími stranami, přes přehled vnitřních procesů, zodpovědnost jednotlivých oddělení nebo zaměstnanců, až po detailní popis každého procesu. Některé z těchto diagramů se principielně podobají diagramům BORM nebo BPMN, takže vyžadují i podobné principy simulace [54].

Jediný nalezený nástroj s funkcí podobnou simulacím je webová aplikace demoworld.nl [64]. Ta umožňuje napřed popsat průběh interakce mezi jednotlivými stranami, a poté takto definovaný průběh plynule přehrát. Jedná se tedy spíše o vizualizaci předem nastavené posloupnosti akcí.

1.7 Shrnutí

Existují tedy dvě možná chápání simulací. Prvním je simulace pomocí matematických principů, kdy dojde k analýze modelu a následnému výpočtu statistických údajů. Řešení toho způsobu simulací však není cílem této práce. Pro tuto práci je důležitější simulace pomocí postupného krokování v diagramu. Během krokování zpravidla dochází k vizualizaci těchto stavů nebo změn.

Simulovatelné diagramy se skládají z prvků považovatelných za pasivní a aktivní. Pasivní prvky zpravidla nesou informace o aktuálním stavu, aktivní slouží pro provádění změn. Aktivním prvkem může být hrana spojující pasivní prvky a nebo uzel, jehož sousední hrany představují nebo obsahují podmínky. Kromě hran mohou být podmínky umístěné také přímo v aktivních prvcích.

V průběhu simulace dochází ke změnám aktivních uzlů diagramu (např. u konečných automatů), což bývá vizuálně znázorněno jako zabarvení nebo orámování daného uzlu nebo znevýraznění ostatních, změnám informací v uzlu (počet tokenů Petriho sítě), znázorňovaných jednorázovou změnou údajů, nebo ke změnám globálních informací v samotném diagramu (možnost u UML stavových automatů), což zpravidla znázorněno není. Může se jednat také o putování objektu skrz graf (u objektových Petriho sítí), znázorňovaném jedno-

rázovou změnou umístění objektu nebo plynulým posunem mezi jednotlivými uzly.

Průchody totožným grafem se mohou lišit podle několika faktorů. Podle jednorázového vstupu, který je zpracováván (například u konečných automatů), podle průběžných vstupů (možnost u UML stavových automatů) nebo případně náhodným výběrem (u stochastickch Petriho sítí).

Jednotlivý krok během průchodu může být závislý na aktuálním stavu, na stavu nebo informacích v jiných částech diagramu, na současné podobě vstupu, typech a pravidel prvků, na prioritách prvků, a to prioritách statických (prioritizované nebo časované Petriho sítě) nebo dynamických (stochastrické Petriho sítě), dále na vědomém výběru na základě větší vhodnosti dané možnosti (nedeterministický konečný automat), či může jít o přímý náhodný výběr.

Při simulaci je vhodné mít možnost provedení jednoho kroku, vrácení posledních kroků, opakované provádění kroků, tedy běh, a to s omezeným i neomezeným počtem opakování. Pak je také potřeba schopnost vrátit stav simulace na začátek i schopnost ponechání stavu po simulaci. Užitečná a běžná je také možnost provedení kroku jednoho konkrétního prvku. Při opakování je pak dobré umožnit nastavení prodlevy mezi jednotlivými kroky. U hierarchických diagramů lze vnitřní diagram přeskočit a zůstat v aktuálním diagramu a nebo se přesunout do vnořeného diagramu a ten dále krokovat.

Kapitola 2

Návrh struktury

V této kapitole je uveden popis současné struktury a základních principů DynaCASE, který je následován návrhem základní struktury simulací a propojení této struktury s DynaCASE.

2.1 Struktura DynaCASE

2.1.1 MVC

DynaCASE je založena na architektuře MVC – model-view-controller. Tato architektura dělí aplikaci na tři části. Model reprezentuje data a view, tedy zobrazení, reprezentuje uživatelské rozhraní. Tyto dvě části jsou nezávislé a díky tomu lze pracovat s modelem i bez této zobrazovací části. Controller je část zodpovědná za propojení modelu a view, tedy určí, co z modelu se má zobrazit pomocí view.

Modelem je hierarchie tříd rozdělená na diagramy a elementy, přičemž diagram obsahuje množinu elementů. Speciálním případem elementu je hrana. Modelem byl dosud myšlen celý systém tříd. Jako model je však zpravidla označována i konkrétní třída nebo její instance, například třída reprezentující přechod konečného automatu.

Vrstva controlleru kopíruje hierarchii modelu, tudíž pro každý diagram existuje controller diagramu a pro každý element existuje controller elementu.

Aby model nemusel předem vědět o existenci controlleru, jsou zde použity události reprezentované třídami. Controllery nebo jakékoliv jiné objekty se mohou přihlásit k odběru událostí daného typu, například změně, smazání apod. Ve chvíli, kdy je pak tento objekt modelu změněn, oznámí všem odběratelům, že tato událost nastala. Na základě toho se provedou bloky [65] kódu určené těmito odběrateli. Příkladem takového bloku kódu je aktualizace zobrazení tohoto elementu, tedy třeba změna popisku, změna velikosti podle délky názvu apod.

2.1.2 GUI

GUI, tedy grafické uživatelské rozhraní, je založené na balíčku pro tvorbu uživatelského rozhraní Spec, který je součástí Phara. Následuje popis tříd relevatních pro tuto práci, vyobrazených také na diagramu 2.1.

Nejdůležitější částí je *DCEditor*. *DCEditor* je rámec obsahující *DCCanvasModel*, který již obsahuje přímo vykreslovací plochu vizualizačního balíčku Roassal. *DCEditor* dále obsahuje paletu nástrojů pro tvorbu elementů *DCPalette* a hlavní menu *DCEditorMenu*, které obsahuje například tlačítka pro export, přiblížení a podobně. Druhou důležitou částí je *DCDynamicForm*, což je rámec schopný průběžných změn svého obsahu, tedy přidávání a odstraňování součástí. Používá se zde pro zobrazení textových polí, jako je například název vybraného prvku. Při změně informací se příslušná hodnota upraví i v modelu.

Při otevření diagramu je pro něj vytvořena instance třídy *DCEditor*. Ta vytvoří příslušný controller diagramu, tedy instanci podtřídy *DCDiagramController*, a z ní se načtou informace pro paletu. Následně je vytvořen *DCCanvasModel* pro daný controller diagramu, ze kterého získá a zpracuje potřebné údaje. Součástí tohoto procesu je i vytvoření kontextového menu pro všechny prvky. Kontextové menu je nabídka zobrazená při kliknutí pravým tlačítkem na prvek ve vykreslovací ploše. Kontextové menu takto vytvoří také při vzniku nového prvku.

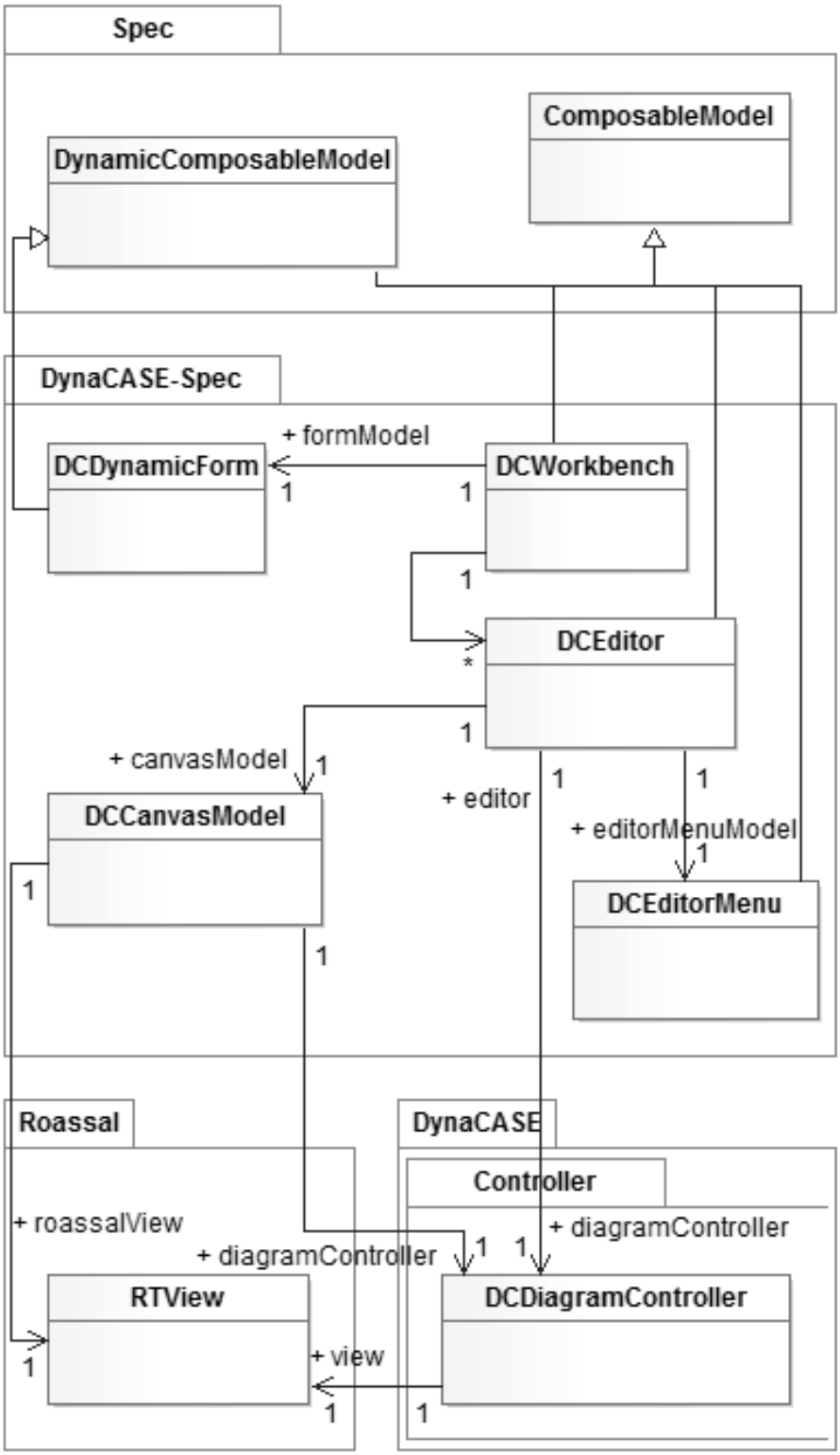
2.2 Možnosti struktury vrstev

Samotné třídy a metody pro simulace je potřeba umístit na určité pozice v MVC architektuře nebo kolem ní. Ať už by bylo zvoleno jakékoliv řešení, tak je většinu vizualizace možno řešit pomocí vlastností architektury MVC. Pokud změníme informaci v modelu, pak se tato změna automaticky, pomocí událostí, promítne i do zobrazení. Pokud tedy například během simulace odebereme v modelu místa v Petriho síti jeden token, pak je tato změna okamžitě zobrazena. Nejsou tedy vyřešeny pouze vizualizace toho, co není součástí modelu, tedy například plynulých animací, aktivních stavů apod.

Následuje výčet možností struktury samotných simulací vůči struktuře DynaCASE.

2.2.1 Přímá integrace

První a nejpřímější možností je vložit třídy a metody pro simulace do již existující vrstev, konkrétně do controlleru a GUI. Nevýhoda tohoto řešení je ta, že by se simulace staly neoddělitelnou součástí DynaCASE a ta už by se bez simulací neobešla. Simulace však ve skutečnosti nebudou pro velkou část diagramů vůbec použity. Proto je vhodné, aby vše kolem samotných simulací bylo oddělené od současné struktury a bylo možné pracovat s DynaCASE i bez balíčků se simulacemi, a to bez nutnosti měnit později kód.



Obrázek 2.1: GUI DynaCASE – UML diagram tříd

2.2.2 Ovládání controlleru

Další možností je postavení simulací v nové vrstvě nad controllery. Tedy tak jako model neví o existenci controllerů, tak by controllery nevěděly o existenci simulačních tříd – simulátorů. Tyto třídy ze simulační vrstvy by také kopírovaly strukturu controllerů, vzájemně si předávaly informace během simulace a případné změny propagovaly do controllerů, které by pak měnily obsah modelu a dokázaly by i zasahovat do vizualizační části, tedy například pro přidání zvýraznění, plynulých animací apod. Nevýhoda by spočívala v neschopnosti použití simulací bez existence vrstvy controlleru, tedy pouze nad samostatným modelem.

2.2.3 Ovládání modelu

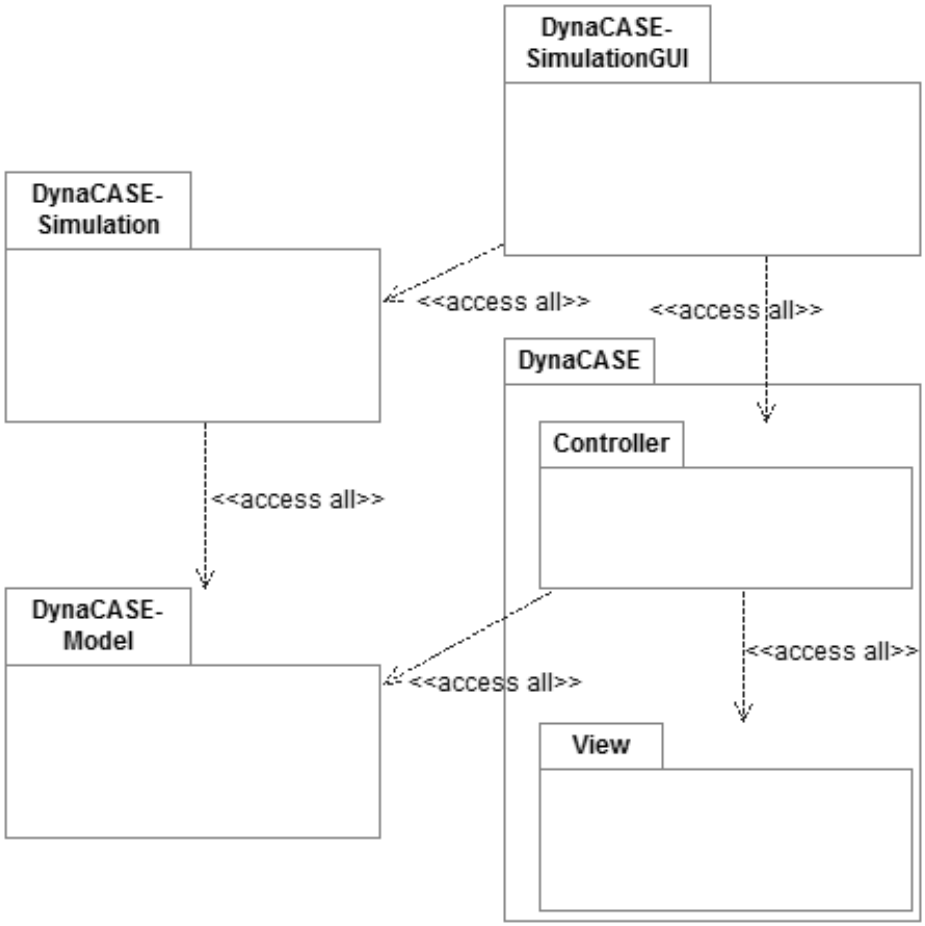
Poslední možností je postavení hlavní simulační vrstvy nad modelem. Mezi hlavní výhody patří schopnost provádět simulace i nad samotným modelem a snazší změny informací v modelu. Nelze však přidávat dodatečné animace a hlavně ani tlačítka pro spuštění nebo ovládání simulace. Je to z důvodu, že ve vrstvě nad modelem není možné, resp. vhodné, jakkoliv sahat na controllery, zobrazení nebo uživatelské rozhraní. Proto je potřeba přidat ještě další vrstvu pro GUI, která bude postavena nad touto simulační vrstvou nad modelem a zároveň nad vrstvou controllerů. Tím vznikne obdoba vztahu model-controller i v částech pro simulaci.

2.2.4 Výsledné řešení

Nevýhoda první možnosti, tedy přímé integrace, je příliš závažná. Cílem totiž bylo vytvořit volitelné rozšíření, nikoliv DynaCASE přímo rozšířit bez možnosti návratu do stavu bez něj.

Rozhodnutí tedy probíhalo mezi druhou a třetí možností. Díky tomu, že není potřeba vizualizovat příliš mnoho informací a dějů oproti těm, které obsahuje samotný model, tak se nevýhoda postavení nad modelem nejeví příliš závažná. Další strukturalizace, tedy rozdělení simulací do dvou vrstev, pak může pomoci během pozdějších úprav. Na druhou stranu to znamená více kódu kvůli propojení těchto vrstev.

Ani jedna z těchto dvou možností se mi nejeví jako výrazně lepší nebo horší a po konzultaci s Damienem Cassou [66], členem výzkumné skupiny vyvíjející platformu Pharo, jsem zvolil řešení pomocí ovládání modelu a zvláštní vrstvy pro interakci s uživatelským rozhraním. Vrstvu a zároveň balíček simulující model jsem nazval *Simulation*, tedy simulace, vrstvu a balíček propojující Simulation a GUI jsem nazval *SimulationGUI*, tedy grafické uživatelské rozhraní simulace. Výsledná struktura balíčků a vrstev je znázorněna na diagramu 2.2.



Obrázek 2.2: Řešení struktury balíčků a vrstev – UML diagram tříd

2.3 Možnosti integrace do DynaCASE

Struktura řešení propojení modelu s vrstvou nad modelem je jednodušší. Stačí vytvořit strukturu tříd pro simulaci, dále zpravidla označované jako simulátory, a určit jim, jak a na co se mají namapovat. Pro propojení vrstvy nad controllery a simulátory, dále označovanou jako simulační GUI, je však několik možností. Problémem je, že základní aplikace nesmí vědět, že simulace existuje, a přesto musí obsahovat alespoň tlačítko pro spuštění simulace.

2.3.1 Pluginy

Standardním řešením bývají pluginy, tedy systém, který registruje všechny nástavby nad aplikací. Po přidání funkčního rozšíření, zvané plugin, do aplikace jsou pluginy spuštěny nebo v průběhu dotazovány a ty poté provedou

potřebné úpravy nebo spustí vlastní přidané činnosti. Výhoda tohoto systému je jeho univerzálnost a možnost dát i uživateli kontrolu nad tím, jaké součásti mají a nemají být použity.

2.3.2 Pragma a rozšíření tříd

Další možností je využití kombinace dvou prvků platformy Pharo.

Prvním z nich jsou direktivy pro virutální stroj zvané pragma [67][65]. Tyto direktivy slouží k označení metod a nechovají se tedy jako standardní kód. Například všechny metody obsahující řádek kódu

```
<menuExtension>
```

jsou z instancí dané třídy dohledatelné pomocí kódu

```
Pragma allNamed: #menuExtension in: self class.
```

Druhým potřebným prvkem je možnost rozšíření tříd z jiných balíčků. Máme-li například třídu *DCDiagramController* v balíčku DynaCASE, tak můžeme v této třídě vytvořit metodu a jako její protokol (kategorii metody) uvést název našeho balíčku, SimulationGUI, ve formátu *SimulationGUI. Pak je toto rozšíření součástí balíčku SimulationGUI, ale při běhu kódu se chová jako plnohodnotná metoda třídy *DCDiagramController*. Při pozdějším načítání tříd do systému se tato metoda ve třídě *DCDiagramController* automaticky vytvoří.

Pokud tedy takto rozšíříme třídu controlleru a zároveň metodu označíme jako pragma, pak je možné přímo z kódu controlleru zjistit všechny takto přidané metody a provést jejich kód, tedy kód z rozšíření, aniž by se v původním nerozšířeném balíčku objevila jediná přímá zmínka o jakékoliv konkrétní třídě nebo čemkoliv jiném, co v hlavní aplikaci není. Lze tedy například spustit metodu obsahující přidání potřebných tlačítek do uživatelského rozhraní sloužících pro spuštění simulace.

Tato varianta už je nyní použita v DynaCASE pro získání výčtu příkladů konkrétních diagramů a plní zamýšlenou funkci. Jediná možná nevýhoda je ta, že kvůli každému podobnému rozšíření jediné třídy musí vzniknout nová metoda, takže by se mohlo stát, že tam takových metod bude příliš a že mohou dvě rozšíření použít stejný název metody, což by způsobilo, že jedna přepíše druhou.

2.3.3 Třídy pro rozšíření

Alternativou je vznik třídy reprezentující rozšíření, od které by všechna rozšíření dědila a pokud bychom takovou třídu nazvali například *DCExtension*, bylo by možné všechny od ní dědící třídy získat pomocí příkazu

```
DCExtension allSubclasses.
```

Poté by již stačilo spustit předem dané metody načítající obsah rozšíření. Tato varianta je ale svými nevýhodami podobná předchozí variantě s rozšířením

tříd, jelikož i tady by každé rozšíření nezávisle udělalo svoji třídu a mohlo by dojít ke konfliktu názvů. To se ale samozřejmě může stát i u jakékoliv jiné třídy.

2.3.4 Inicializace tříd

Posledním možným řešením je využití inicializace, tedy načítání, tříd. Díky tomu, že je Pharo čistě objektový jazyk, tak i třídy jsou objekty. A tak jako se instance po vytvoření inicializují, tedy je automaticky zavolána metoda initialize, tak i po vytvoření třídy je její metoda s názvem initialize zavolána. Díky tomu je možné spustit kód ihned poté, co je třída načtena do systému. V této metodě pak stačí zavolat metodu už existující známé třídy z původní aplikace, opět třeba takovou, která slouží k přidání tlačítek do seznamu tlačítek pro danou třídu, odkud si její instance tento seznam převezme.

Využití inicializace tříd však není příliš obvyklé. Je to převážně z důvodu, že v době volání inicializace třídy ještě neexistují žádné instance tříd, které by byly tímto rozšiřovány o kód z inicializované třídy. Proto lze volat jen metody tříd a ne instancí. Další nevýhoda spočívá v problému, že inicializační metoda tříd není volána vždy, protože volání této metody záleží na konkrétním způsobu nahrávání kódu do prostředí Pharo. Například po vytvoření třídy programátorem není inicializační metoda zavolána. Poslední nevýhodou je to, že bude potřeba vytvořit metodu initialize ve všech podtřídách, jelikož na straně třídy není volána metoda initialize nadtřídy automaticky.

2.3.5 Vybrané řešení

V tuto chvíli není jisté, jaká další rozšíření budou ještě následovat, kolik jich bude a jakého budou rozsahu a zaměření, proto je nejspíše předčasné navrhovat komplexní systém pro pluginy.

Možnost s použitím pragma výrazu a rozšířením tříd o nové metody by, i přes svoji nevýhodu, byla vyhovující a funkční. To samé platí o speciální třídě pro rozšíření. Minimálně do doby, než by přislo výrazně větší množství rozšíření.

Použití inicializace třídy umožňuje vyšší čistotu kódu a využití principů objektového jazyka, přestože, podle mého názoru, není implementace inicializací tříd zcela dokonale vyřešená. Proto je experimentem, který nemusí dlouhodobě fungovat. Rozhodl jsem se však tento experiment absolvovat s tím, že pokud by se tato metoda ukázala jako dlouhodobě problémová, tak bude nahrazena jiným řešením.

Kapitola 3

Simulace modelu

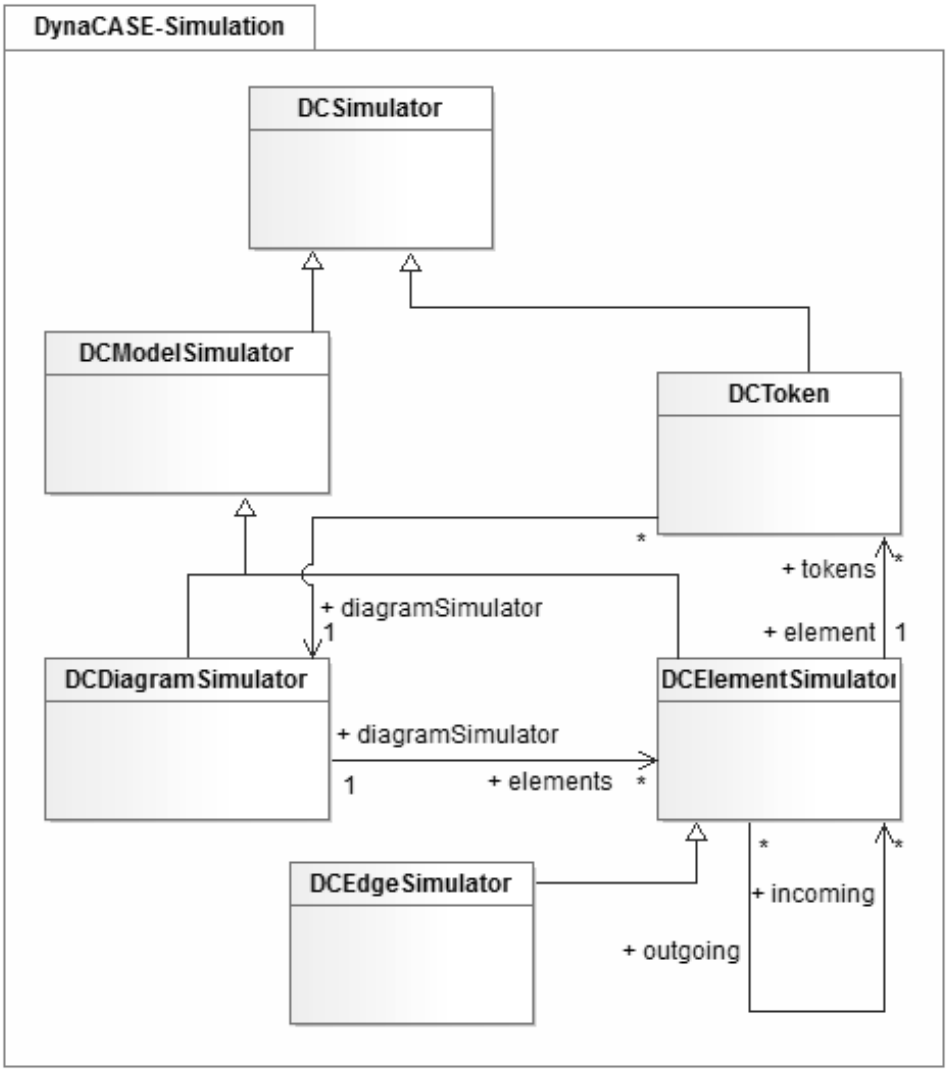
V této kapitole je popsán návrh a implementace tříd pro simulaci modelu diagramů.

3.1 Struktura tříd

Aby mohly simulátory (třídy pro simulaci) dostatečně kopírovat strukturu vrstvy modelu, musí i samotná struktura tříd kopírovat strukturu tříd modelu. Nejvyšší třídou tedy bude obecný *DCSimulator*, který se dělí na simuátor diagramu *DCDiagramSimulator* a simulátor elementů *DCElementSimulator*, který má ještě speciální případ simulátoru hrany *DCEdgeSimulator*. Jak se ukázalo například u objektových Petriho sítí nebo nedeterministických stavových automatů, tak může být vhodné mít také samostatný inteligentní objekt procházející diagramem – token, který bude obsažen v konkrétních elementech. Třída pro token, *DCToken*, má s elementem společné to, že je obsažen v diagramu. Diagram a element však mají důležitější společné rysy. Diagram i element mají přímou vazbu na prvek z modelu a navíc v sobě obsahují množinu prvků: diagram obsahuje elementy a element obsahuje tokeny. Proto je v hierarchii mezi *DCSimulator* a *DCDiagramSimulator* s *DCElementSimulator* třída *DCModelSimulator*, tedy takový simulátor, který je napojený na model a navíc umožňuje procházení svých prvků. Toto je vyobrazeno na diagramu 3.1.

Z těchto tříd pak budou dědit třídy pro simulaci konkrétních druhů diagramů, tedy například třída pro simulaci diagramu Petriho sítí *DCPetriNetSimulator*, bude dědit od třídy *DCDiagramSimulator*.

Vzhledem k tomu, že je často potřeba, aby byly předávány informace mezi sousedními elementy a od elementů k diagramu, rozhodl jsem se o silné propojení těchto tříd. Tudíž element i token zná svůj diagram, token zná svůj element a elementy znají své sousedy. Přináší to nevýhodu v podobě komplikovanější tvorby instancí, ale díky tomu, že přidávání a odebírání nových simulátorů není možné a ani smysluplné, tak jde pouze o jednorázovou kom-



Obrázek 3.1: Třídy pro simulaci – UML diagram tříd

plikaci při prvním načítání instancí. Je tu však zásadnější výhoda v podobě možného přímého přístupu k potřebné informaci, místo prohledávání simulátoru diagramu za účelem nalezení sousedního elementu.

3.1.1 Tvorba instancí

Struktura simulace je vytvořena po předání diagramu nové instanci třídy *DCDiagramSimulator*. Tento simulátor obsahuje informace o přiřazení elementů modelu se simulátory modelu, proto pro každý model vytvoří simulátor, který se na model napojí. Následně každému simulátoru přikáže, aby si zjistil a načetl své sousední elementy. Ty je získají pomocí dotazování simulátoru diagramu na simulátor elementu pro jejich sousední modely. Nakonec si diagram a všechny elementy načtou z modelu případné další informace o sobě, např. tokeny.

Vzhledem k tomu, že simulace mění model, tak je potřeba, aby šel vrátit stav před simulací. O to se starají metody *loadState* a *saveState*, kde *saveState* uloží aktuální model do zvláštní proměnné a loadState vezme tento stav z proměnné a nahradí jím model. Toto se děje na úrovni jednotlivých elementů, přičemž simulátor diagramu v metodách *loadState* a *saveState* jen deleguje tyto zprávy všem svým elementům. Při zavolání metody *loadState* se také zastaví běh, tedy se zavolá metoda *stop*. Možnosti vylepšení tohoto řešení uvádím v podsekci 7.1.1 kapitoly možností budoucího vývoje.

Simulátory nezjišťují, zda nebyl z modelu odebrán nebo přidán prvek, tudíž simulace probíhá jako by tam stále byl. O znovunačtení těchto prvků se stará třída pro uživatelské rozhraní simulací.

3.2 Krokování

3.2.1 Kroky a jejich opakování

Jak se ukázalo, tak kromě jediného simulátoru BPMN, kde probíhalo několik zcela nezávislých simulací najednou, šlo vždy o sled diskrétních kroků. Základem simulační akce je proto jeden krok – *step*. Tento krok by mělo být možné provést nad celým diagramem, který sám rozhodne, jak tento krok simulovat, ale i nad jednotlivým elementem zvolitelným z uživatelského rozhraní nebo dokonce nad tokenem. Princip kroků je tedy obsažen už v *DCSimulator* a je možné ho kdykoliv spustit.

Kromě samotatných kroků je také potřeba kroky automaticky opakovat. Mezi kroky zpravidla musí být časový rozestup. Vznikly proto také metody *repeat*, *repeatTimes:* a *repeatWithDelay:*, které v samostatném vlákně, resp. procesu uvnitř virtuálního stroje, opakují kroky a prokládají je pauzami. V samostatném vlákně z toho důvodu, že při provádění tohoto kódu je jinak zablokováno samotné prostředí, a to nejen obnovování vykreslovací plochy, ale i možnost opakování přerušít.

V průběhu implementace jsem pracoval také s metodou *repeatWithoutDelay*, tedy opakování bez prodlevy mezi kroky, ale při vykreslování jsem se setkal s problémy plynoucími z nedostatečné synchornizace vlákna s vláknem obsluhujícím grafické prostředí, což vedlo k chybám při vykreslování.

Opakování je možné přerušit zavoláním metody *stop*, která změní hodnotu proměnné, jež určuje, zda by měl běh pokračovat. Před každým dalším krokem v rámci běhu je stav této proměnné ověřován. Čas samozřejmě může být řešen i externím časovačem, který vždy provede krok sám.

Pro diagramy s obsahem času, jako jsou například časované Petriho sítě a nebo většina business notací, jeden krok nemusí nutně znamenat změnu aktivního stavu, přesun tokenů a podobně, ale může znamenat pouhé posunutí času. Akce prováděné při zavolání metody *step* jsou totiž určeny nastavitelnými akcemi, které se v daném pořadí provedou. Toho je dosaženo pomocí kolekce *actions*, do které lze přidávat bloky kódu, které jsou provedeny na prvku předaném jako vstup tohoto bloku.

3.2.2 Výběr krokovaných prvků

V závislosti na druhu diagramu se výrazně liší, které prvky jsou simulovány a případně i v jakém pořadí. Bylo tedy vhodné navrhnout takový systém, který by umožnil krokování diagramu, elementů i tokenů. Musí být možné krokovat všechny elementy, část z nich nebo jen jeden, přičemž ten jeden moct vybrat náhodně nebo podle priorit, mít možnost vybrat jeden ze všech s nejvyšší prioritou apod.

Proto jsem v rámci simulačních tříd vytvořil systém nastavení, pomocí kterého je získána kolekce pro provádění akcí a tedy i krok. Jde o rozdělení natavení do kategorií, které jsou aplikovány pro každý krok v pořadí

- *from* – výběr zdrojové kolekce prvků, například všech tokenů, tokenů právě krokovatelných, elementů, sebe sama...,
- *priority* – filtrování podle priority prvku, například výběr pouze prvků s nejvyšší prioritou,
- *order* – určení pořadí prvků, například náhodné, podle priorit, podle priorit obráceně, nespecifikované pořadí apod.,
- *select* – výběr určitého množství prvků, například jednoho prvního, určitého počtu nebo všech.

Každá kategorie využívá vlastní proměnnou obsahující blok, který získá prvky z předchozího bloku a vrátí upravené prvky podle sebe. Předchozí blok je určen pomocí metod, takže metoda pro získání bloku pro *order* vrátí blok *priority*. To je vhodné proto, že jsou kroky a akce definovány na úrovni *DCSimulator*. Tomu se pak stačí tako zeptat na vlastní zdroj prvků pro akce, kde diagram nebo element vrátí select, kdežto token vrátí rovnou sebe. Kromě

tokenu tedy platí, že krok (step) spustí akce, které jsou použity na kolekci získanou z bloku select, který pro sebe získá kolekci z bloku order atd.

Takto je umožněno nastavit libovolnou kombinaci prvků pro krokování. Také je možné samostatně přidat nový způsob řazení, aniž by to jakkoliv ovlivnilo další kategorie.

Výběr jednotlivých nastavení se děje pomocí metod, jejichž název vždy začíná názvem kategorie. Pokud by se časem ukázalo, že vznikne příliš mnoho takových metod, bylo by možné zkusit pro každou kategorii vytvořit objekt, získat jedinou metodou tento objekt a dále pracovat s ním.

Při inicializaci všech simulátorů je zavolána také metoda *setDefaultStepping* (přeložitelná jako „nastavit výchozí krokování“), ve které může každý simulátor, a hlavně jeho podtřída pro konkrétní diagram, definovat výchozi nastavení pro krokování pomoci nastavení jednotlivých kategorií.

Kapitola 4

Integrace s grafickým rozhraním

Tato kapitola je věnována grafickému rozhraní pro simulace. To se skládá ze dvou částí. Jednou je rozšíření samotné DynaCASE o možnost simulací, druhou je grafické rozhraní pro ovládání simulace. Obě části jsem vyřešil pomocí jediné třídy, *DCSimulationGUI*. Jak jsem již předeslal v části o možnostech integrace do DynaCASE, tak je integrace založena na inicializaci třídy *DCSimulationGUI*, což se děje automaticky při načtení této třídy do systému. Kompletní propojení tříd je vyobrazeno v diagramu 4.1. Každý nový diagram se simulací vytvoří vlastní podtřídu, například *DCPetriNetSimulationGUI*, kde budou nastavena potřebná specifika.

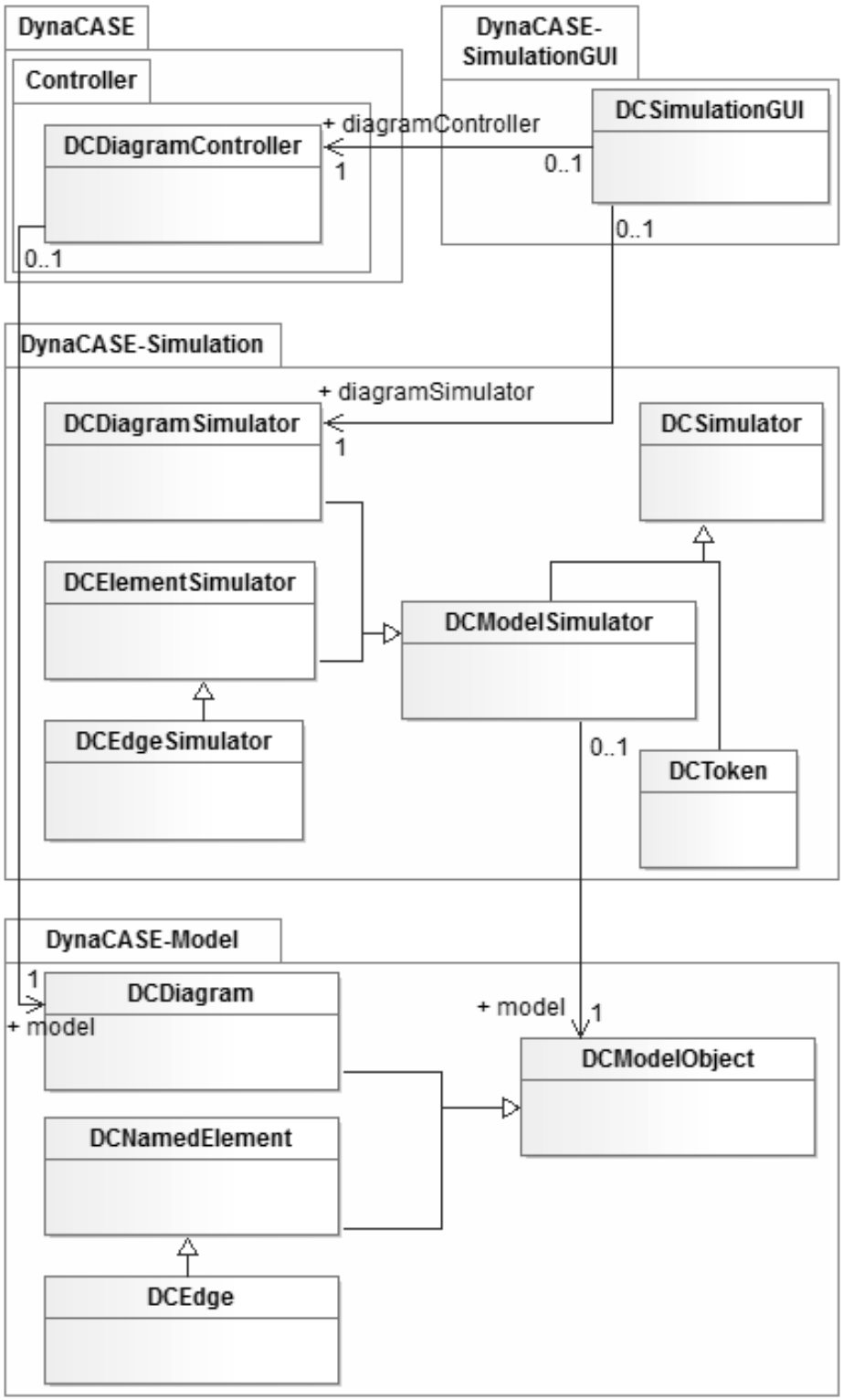
4.1 Rozšíření grafického rozhraní DynaCASE

Je potřeba přidat tlačítko pro spuštění simulací do hlavního menu a rozhodl jsem se také vytvořit možnost přidání tlačítka do kontextového menu jednotlivých prvků.

4.1.1 Rozšíření hlavního menu

Hlavní menu je obsaženo v *DCEditorMenu*, což je rámec Spec, do kterého lze vkládat další objekty, v tomto případě menu s tlačítky. V původní podobě DynaCASE byla všechna tlačítka načtena při první inicializaci *DCEditorMenu* z jeho metody a již nebylo možné tato tlačítka jakkoliv měnit, jelikož tento rámec Spec, třída *ComposableModel*, dodatečné změny neumožňuje.

Jednou možností by bylo vytvořit třídní metodu *DCEditorMenu* pro přidání tlačítka do jeho instancí a během inicializace *DCSimulationGUI* tuto metodu zavolat a tím tlačítko přidat. To by však znamenalo mít toto tlačítko ve všech existujících diagramech, včetně těch, u který nemá simulace smysl. Zvolenou alternativou se stala změna *DCEditorMenu* z *ComposableModel* na *DynamicComposableModel*, který umožňuje změnit obsah i když už je otevřen. Přidal jsem tedy metody pro přidání tlačítek na úrovni instance,



Obrázek 4.1: Propojení vrstev pro simulaci – UML diagram tříd

které jsou volány z controlleru diagramu, jelikož *DCEditorMenu* zná svůj controller diagramu. Evidenci dodatečných tlačítek jsem vložil do třídy *DCDiagramController*, které jsem přidal metodu pro přidání tlačítek do své evidence (proměnné s kolekcí bloků pro přidání tlačítka) a právě tuto metodu volá příslušná podtřída *DCSimulationGUI* při své inicializaci. Tudíž *DCPetriNetSimulationGUI* bude volat metodu pro přidání tlačítka třídy *DCPetriNetController*. *DCEditorMenu* si pak při načtení nového controlleru diagramu od něj vyžádá přidání dodatečných tlačítek. *DCOntoUMLController* nepřidá žádné, *DCPetriNetController* přidá tlačítko pro spuštění *DCPetriNetSimulationGUI*, kterému předá sebe, tedy controller diagramu.

4.1.2 Rozšíření kontextového menu

Tak jako by se měla vyskytovat tlačítka hlavního menu související se simulacemi jen u simulovatelných diagramů, tak tlačítka kontextových menu související se simulacemi by se měla vyskytovat jen u simulovatelných elementů. Takové tlačítko by například u simulovatelných Petriho sítí nemělo smysl u míst ani hran. Jen u přechodů.

U rozšíření kontextového menu jsem postupoval obdobně jako u hlavního menu. Podtřídy *DCSimulationGUI* obsahují výčet controllerů simulovatelných elementů a během inicializace *DCSimulationGUI* je těmto třídám těchto controllerů přidáno tlačítko stejným systémem, jako controllerům diagramů tlačítko pro hlavní menu. Z controllerů elementů si tato dodatečná tlačítka vyžádá *DCCanvasModel*.

4.2 Grafické rozhraní pro ovládání simulace

Po stisku tlačítka z hlavního menu je zavolána metoda *controller:* třídy *DCSimulationGUI*, které je předán controller k simulaci. Zde jsou prohledány existující instance *DCSimulationGUI* a pokud je nalezena některá ovládající simulaci právě pro tento controller, tak je vrácena ta. Pokud žádná instance není, pak je vytvořena nová instance pro tento controller. Všechny instance sebe sama jsou získány pomocí kódu

```
self allInstances.
```

Při vytvoření nové instance je vytvořen simulátor diagramu, otevřeno okno s ovládacími prvky pro tento simulátor a toto okno při zavření vykoná blok obsahující příkaz pro ukončení simulace. Díky tomu simulace skončí při zavření okna s ovládáním simulace. Součástí ukončení simulace je i načtení modelu do stavu před simulací, tj. zavolání metody *loadState* na příslušný simulátor diagramu.

Tato třída se také přihlásí k odběru událostí přidání a odebrání elementu z diagramu a pokud se to stane, pak dojde k načtení původního stavu elementů

modelu před simulací a k následnému vytvoření nového simulátoru, který již bude obsahovat nový nebo nebude obsahovat odstraněný element.

Okno s uživatelským rozhraním pro simulace je složené z tlačítek získatelných z metody *buttonItems*, kterou lze v podtřídách překrýt a tím definovat vlastní tlačítka pro potřeby konkrétních typů diagramu. Mezi univerzální tlačítka patří načtení původního stavu modelu a uložení stavu modelu.

Krok pomocí tlačítka z kontextového menu je realizován tak, že je předán controller příslušného elementu třídě *DCSimulationGUI*. Ta i v tomto případě vytvoří nebo získá vlastní instanci pro controller diagramu tohoto controlleru elementu. Vyhledání nebo vytvoření instance je realizováno stejnou metodou, jako v případě stisknutí tlačítka v hlavním menu spouštícího simulaci. Této instanci předá controller elementu, u kterého má být tento krok proveden. Ten již od simulátoru diagramu získá simulátor tohoto elementu a provede jeho krok.

Kapitola 5

Návrh a implementace Petriho sítí

Důvodem implementace Petriho sítí bylo převážně testování simulací, proto jsem se rozhodl implementovat pouze základní Petriho sítě, rozšířené o inhibiční a nulovací hrany. Tato kapitola tedy obsahuje popis návrhu a implementace simulovatelných Petriho sítí do DynaCASE.

5.1 Model a zobrazení Petriho sítí

5.1.1 Tvorba modelu a controllerů

Model Petriho sítí se skládá z diagramu, hran a uzlů. Uzly se dělí na místa a přechody. Místa obsahují informace o počtu tokenů. Z hran máme normální vstupní hrany, inhibiční hrany, nulovací hrany a výstupní hrany. Normální vstupní a výstupní hrany obsahují informaci o váze, proto jsem je sdružil pod společnou nadtřídu *DCPetriNetWeightedArc*, znamenající hranu s vahou.

Pro řešení vykreslení míst jsem zvažoval tři možnosti. První možností bylo přidání vlastního tvaru obsahující ikonku tokenů, to by však nebylo vhodné, jelikož je nutné tuto ikonku schovat, pokud zde žádný token není. Druhou možností bylo použití složeného tvaru (angl. *composite shape*) Roassalu, tj. vykreslovacího balíčku použitém v DynaCASE. I přes veškerou snahu, a několik oprav od autorů i ode mně v průběhu tvorby této práce [68][69], se bohužel ukázalo, že složené tvary zatím nejsou plně funkční pro využití v DynaCASE. Volba tedy padla na rozdělení do tří samostatných tvarů. První tvar je ohraničující kolečko, druhý token a třetí popisek s počtem tokenů, přičemž tyto tvary jsou propojeny a při pohybu prvního se pohnou i ostatní.

5.1.2 Integrace Petriho sítí do DynaCASE

DynaCASE obsahovala fixně dané seznamy druhů diagramů, které lze vytvořit z hlavního menu. To však nebylo akceptovatelné, jelikož by musela existovat centrální „autorita“, která by každý existující druh diagramu do tohoto menu přidala. Navrhl jsem tedy systém pro automatické získávání možných diagramů.

Lze předpokládat, že všechny podtřídy třídy *DCDiagramController* jsou diagramy spustitelné v DynaCASE, proto třída *DCWorkshop*, která je zodpovědná za výše zmíněné menu, získá všechny tyto podtřídy voláním

```
DCDiagramController allSubclasses.
```

Tyto podtřídy nově obsahují informace o názvu tohoto diagramu, metodu vytvářející nový diagram a ikonu pro daný druh diagramu, což jsou informace, které *DCWorkshop* pro správné načtení seznamu diagramů potřebuje.

5.2 Rozšíření Petriho sítí o simulace

Použití simulací pro Petriho sítě se skládalo z vytvoření struktury simulátorů Petriho sítí, tedy podtříd struktury obecných simulátorů, a z vytvoření podtřídy *DCSimulationGUI*.

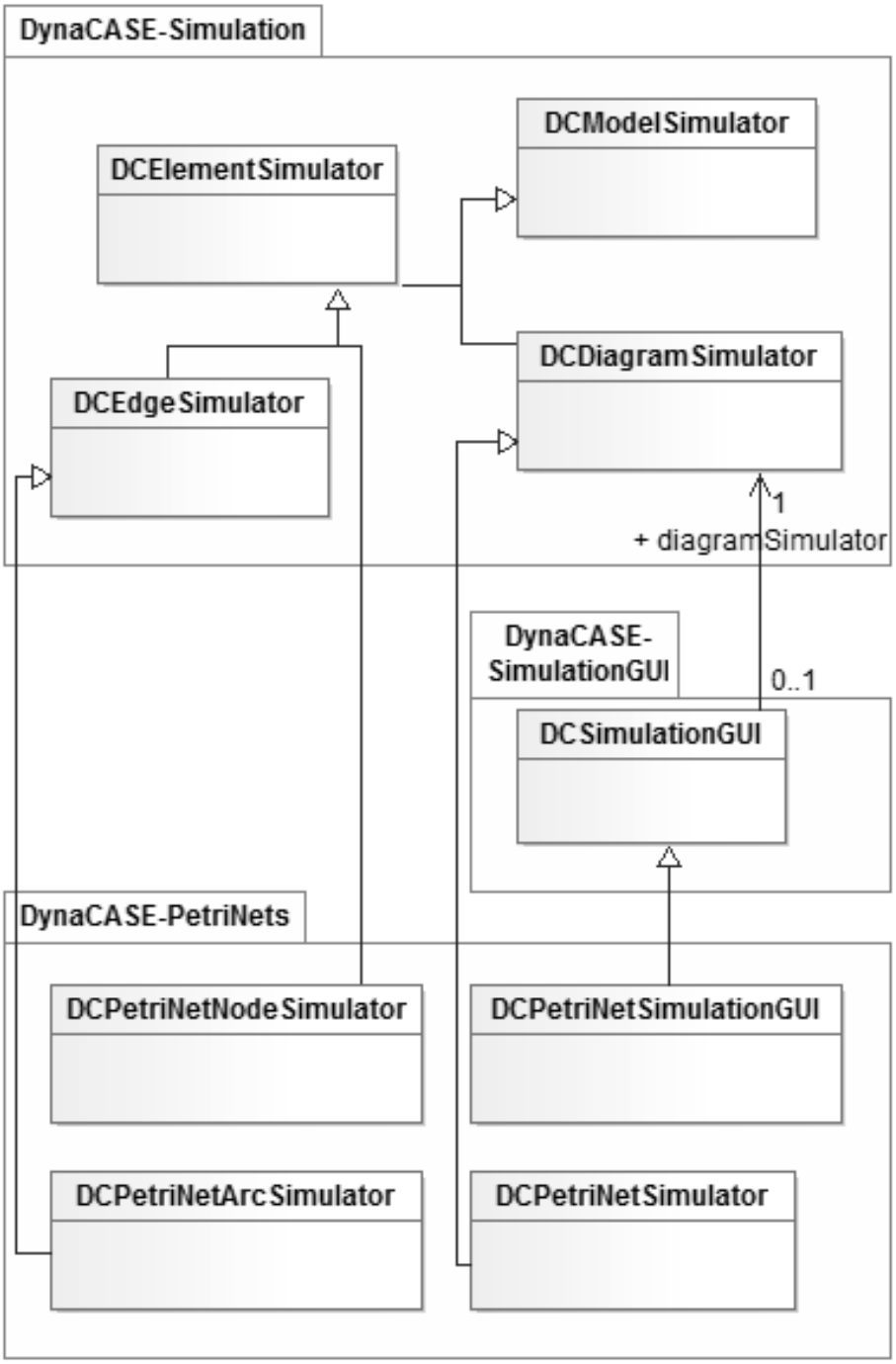
Bylo potřeba definovat metodu zjišťující proveditelnost přechodu a metodu provádějící tento přechod. Proveditelnost záleží pouze na vstupních hranách a místech. Proto se simulátor přechodu zeptá vstupních hran, zda jsou každá proveditelná, a pokud ano, tak je proveditelný celý přechod. Každá z hran získá tuto informaci na základě vlastní třídy a případně počtu tokenů ve vstupním místě.

Provedení přechodu je řešeno tak, že jsou nejdříve vykonány akce vstupních hran, které ovládají svá vstupní místa. Poté následují akce výstupních hran, které ve výstupních místech vytvoří tokeny podle své váhy. Samotný přechod tedy na žádný token nesahá, jen ovládá své přilehlé hrany a ty se samy rozhodují o akci podle vlastní třídy. Přechod bude muset přímo manipulovat s tokeny pouze u barevných Petriho sítí, protože sám přechod bude moct obsahovat vlastní prováděcí logiku.

U simulátoru diagramu Petriho sítí je využito předefinování metody *DCSimulationGUI*, *setDefaultStepping*, pro nastavení výchozího krokování. Tím je krok jednoho náhodného proveditelného přechodu, tedy v terminologii simulátorů *krokovatelného elementu – steppable element*.

Ačkoliv simulátory pracují s pojmem tokenů, tak se u základních Petriho sítí nejedná o objekty považovatelné za jakkoliv inteligentní, ale jedná se o pouhé hodnoty – čísla. Proto se tu v metodách nepracuje s metodami simulátorů jako je *tokens*, ale metodami *tokensAmount*, vyjadřujícími používání prostých celých čísel.

Nakonec byla vytvořena podtřída třídy *DCSimulationGUI*, která definuje tlačítka pro krok, krok všech přechodů, běh a ukončení běhu a dále všechny metody, které podle *DCSimulationGUI* definovat musí. Diagram dědičnosti tříd mezi simulacemi a Petriho sítěmí se nachází v diagramu 5.1.



Obrázek 5.1: Dědičnost tříd mezi simulacemi a Petriho sítěmi – UML diagram tříd

Kapitola 6

Testování

Součástí implementace je ověřování její správnosti – testování. Testování této práce se dělí na dva druhy. Jedním jsou automatické jednotkové testy a druhým typové příklady spouštěné a ověřované vývojářem. Tato kapitola se oběma těmto částem věnuje.

6.1 Jednotkové testy

Jednotkové testy jsou provedeny pomocí systému pro testování obsaženém v prostředí Pharo. Testovány jsou tímto způsobem třídy pro simulaci a třídy Petriho sítí. Tyto testy jsou spouštěny pomocí nástroje *Test Runner*.

6.1.1 Testy simulátorů

Všechny simulátory jsou abstraktní třídy, proto bylo nutné vytvořit systém pro testování těchto tříd i bez existujících tříd pro konkrétní diagramy. Z tohoto důvodu jsem vytvořil třídy určené pouze k testování, které obsahují metody pro změnu stavu, který napodobují. Názvy techto podtříd začínají slovem *Dummy*.

Jedním druhem *Dummy* tříd jsou podtřídy simulátorů, které jsou nutné proto, že simulátory jsou třídy abstraktní, takže by nebylo možné vytvořit jejich instanci, která by byla testována. Každá z těchto tříd slouží pro testování právě těch tříd, ze kterých dědí.

Druhým druhem jsou *Dummy* třídy zcela nezávislé na skutečných třídách. Slouží pro nezávislé testování vyžadující i jiné třídy, než je třída právě testovaná, díky čemuž jsou tyto testy skutečně jednotkové a změna jedné třídy nijak neovlivní výsledek testů třídy jiné.

Výhodou tohoto systému je schopnost podrobně testovat všechny možnosti, ale nevýhodou je nemožnost z testovacích tříd dědit, kvůli jejich silnému provázání s *Dummy* třídami.

Testovány jsou všechny simulátory, avšak nebylo možné jednotkovými testy testovat třídu *DCSimulationGUI*, kvůli jejímu silnému provázání s uživatelským rozhraním a vizuálními prvky. Proto je pro ni používán pouze manuální test založený na typovém příkladu Petriho sítí.

6.1.2 Petriho sítě

Testovací třídy modelu a controllerů dědí z testovacích tříd z balíčku DynaCASE. K tomu navíc testují všechny jejich přidané nebo změněné funkce a vlastnosti.

Simulátory elementů a diagramů Petriho sítí spoléhají na testy z balíčku pro simulace popsané v předchozí části této práce – 6.1.1. K nim přidávají testování svého vlastního nastavení a dodatečných metod.

Tak jako v případě třídy *DCSimulationGUI* nebyly vytvořeny jednotkové testy ani pro její podtřídu *DCPetriNetSimulationGUI*.

6.2 Typové příklady

Testování grafického uživatelského rozhraní pro Petriho sítě i simulace je založené na ručním otevření příkladu, spuštění simulací a následném pozorování.

Při otevření hlavního menu prostředí Pharo se zde nachází příklady jednotlivých diagramů. Jedním z nich jsou Petriho sítě. Po otevření tohoto příkladu je nutné spustit simulaci kliknutím na tlačítko *Simulation* nebo na položku *step* kontextového menu přechodu. Tím je otevřeno uživatelské rozhraní pro simulace.

Tlačítko *step* by při každém kliknutí mělo provést jeden náhodný přechod, tlačítko *run* by mělo spustit nepřetržité opakování těchto kroků, tlačítko *stop* by tento běh mělo ukončit, tlačítko *loadState* má nahrát stav před první změnou po otevření uživatelského rozhraní pro simulaci a *saveState* by mělo tento uložený stav přepsat aktuálním, takže příští kliknutí na tlačítko *loadState* nahraje tento nově uložený stav.

Při kliknutí na tlačítko *loadState* by se také mělo ukončit opakování – běh. Po zavření okna, přidání prvku do diagramu nebo odebrání prvku z diagramu by mělo dojít ke stejnému efektu, jako při použití tlačítka *loadState*. Přidání nebo odebrání prvku by také mělo aktualizovat strukturu simulátorů, které musí s tímto novým stavem diagramu počítat při příštím provádění kroků.

Kapitola 7

Možnosti budoucího vývoje

Vývoj DynaCASE může v souvislosti s touto prací pokračovat mnoha směry. Lze dále vylepšovat podporu simulací, použít simulace na další diagramy a nebo přidat další varianty a rozšíření Petriho sítí.

7.1 Vývoj simulací

7.1.1 Vracení změn

Nejvýznamnějším možným vylepšením simulací by bylo umožnění provádět kroky nejen dopředu, ale i dozadu, tedy opakované vrácení poslední změny. Ačkoliv by bylo snazší toto implementovat pomocí ukládání jednotlivých stavů na úrovni simulace, vhodnější by bylo navrhnout a implementovat možnost návratu zpět pro celou aplikaci DynaCASE a jen toho při simulaci využít. Pro tento účel se zpravidla používá jeden z následujících dvou návrhových vzorů. Jedním z nich je *memento pattern* a druhým *command pattern*. Oba umožňují provádět kroky zpět do historie a naopak opakovat změny dopředu z historie [70].

7.1.2 GUI

Druhou možností je vylepšení chování uživatelského rozhraní. Tedy například

- integrace do stávajícího okna, místo používání vlastního,
- kromě tlačítek umožnit vložení i textového pole do tohoto okna,
- zvýrazňování krokovatelných prvků

a další drobnější změny.

Na konec je třeba uvést, že je možné, se ukáže řešení části pro uživatelské rozhraní pomocí jediné třídy jako nedostatečně členité, a proto bude potřeba tuto vrstvu více rozdělit.

7.1.3 Frekvence opakování kroků při běhu

Dalším vylepšením by bylo přesnější časování kroků při běhu. Nyní jsou použity časové odstupy od ukončení kroku do dalšího kroku, tudíž celý jeden cyklus obsahuje tento čas prodlevy plus čas provádění kroku plus ověřování, zda se má v běhu pokračovat. Lepší variantou by bylo zavolání kroku po přesném čase, tedy měření času nezávisle na provádení kroků.

7.1.4 Simulace při použití hierarchie

Mnoho notací má své hierarchické varianty, při kterých dochází buď k souběžné simulaci více propojených diagramů zároveň nebo k přepínání mezi jednotlivými diagramy. Podpora pro hierarchie se zatím nevyskytuje ani v základní aplikaci DynaCASE, ale ve chvíli, kdy bude do DynaCASE navržena a implementována, bude vhodné zajistit podporu hierarchie i u simulací.

7.1.5 Nahrazení inicializace tříd

Jak jsem naznačil v podsekcích této práce 2.3.4 a 2.3.5, tak je současně implementované řešení integrace simulací do DynaCASE pomocí inicializace tříd experimentem, který vyžaduje dlouhodobější trvání. Pokud by se ukázalo, že toto řešení nevyhovuje, bude nutné způsob integrace nahradit jiným. Další možnosti jsou uvedeny v sekci 2.3.

7.2 Další použití simulací

Simulace bude možné použít i pro další druhy diagramů. Z již existujících jsou to stavové automaty, pokud pro ně bude definován účel, tedy jako analyzátor vstupu a nebo generátor výstupů, tj. ovladač dalšího objektu. Souběžně s touto prací vzniká implementace diagramů BORM, pro které se také dají simulace použít.

7.3 Rozvoj Petriho sítí

Pro Petriho sítě existuje velké množství rozšíření a variant, z nichž je mnoho velmi užitečných, a proto by bylo dobré je implementovat, tudíž i tímto směrem by se další práce mohla vydat.

Závěr

Cílem této práce bylo rozšíření nástroje DynaCASE o podporu simulací a vizualizace těchto simulací. Po zjištění potřebných funkcí a získání přehledu o tom, jak tyto funkce realizují již existující nástroje, jsem navrhl a implementoval funkční systém pro simulace pomocí krokování. Simulovat lze takto samotný model bez grafického rozhraní, ale je možné i využít tohoto rozhraní k pohodlnějšímu ovládání a k vizualizaci simulace. Kroky simulace lze provádět ručně, a to na celý diagram i na konkrétní jeho elementy, případně i automatickým opakováním.

DynaCASE jsem rozšířil nejen o simulace a jejich vizualizaci, ale i o Petriho sítě, ve kterých jsem simulaci použil, tudíž jsou výsledkem simulovatelné Petriho sítě.

V rámci svojí práce jsem také v mnoha ohledech vylepšil samotný nástroj DynaCASE a v neposlední řadě i použitý balíček pro vizualizaci Roassal, u kterého jsem zjistil i opravil některé jeho nedostatky.

Pro ověření správnosti implementace jsem vytvořil jednotkové testy a metodiku manuálního testování na příkladu.

Tak jako na samotném nástroji DynaCASE, lze i na výsledku mojí práce najít mnoho cest pro její vylepšení nebo využití, takže kromě splnění cílů jsem i umožnil a popsal další možnou práci.

Použité zdroje

[1] Mylopoulos, J.: Metamodeling [online prezentace]. 2004, [cit. 2015-05-07]. Dostupné z: http://www.cs.toronto.edu/~jm/2507S/Notes04/Meta.pdf

[2] Nico [N. Petton], Esteban [E. Lorenzano] a Damien [D. Cassou]: Pharo - Documentation. Pharo [online]. [cit. 2015-05-07]. Dostupné z: http://pharo.org/documentation

[3] Nico [N. Petton], Esteban [E. Lorenzano] a Damien [D. Cassou]: About Pharo. Pharo [online]. [cit. 2015-05-07]. Dostupné z: http://pharo.org/about

[4] Object Profile: Roassal [online]. [cit. 2015-05-07]. Dostupné z: http://objectprofile.com/Roassal.html

[5] kolektiv autorů: Moose [online]. [cit. 2015-05-07]. Dostupné z: http://www.moosetechnology.org

[6] Kochaníčková, M.: Petriho sítě. 2008, [cit. 2015-05-07]. Dostupné z: http://phoenix.inf.upol.cz/esf/ucebni/petriho_site.pdf.renamed

[7] Murata, T.: Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, ročník 77, č. 4, Duben 1989: s. 541–580, [cit. 2015-05-07]. Dostupné z: http://www2.ing.unipi.it/~a009435/issw/extra/murata.pdf

[8] Hrabák, M.: *Možnosti formalizace modelování procesů*. Diplomová práce, České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra ekonomiky, manažerství a humanitních věd, 2013.

[9] [Chen, M.]: Petri Nets [online]. [cit. 2015-05-07]. Dostupné z: http://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq.html

[10] Silva, J. R. a P. M. G. del Foyo: Timed Petri Nets. In *Petri Nets - Manufacturing and Computer Science*, InTech, 2012, ISBN 978-953-51-0700-2, s. 359–378, doi:10.5772/50117. Dostupné z: http://www.intechopen.com/books/export/citation/BibTex/petri-nets-manufacturing-and-computer-science/timed-petri-nets

[11] Aalst, W. M. P. van der a K. van Hee: *Workflow management: models, methods, and systems*. 2000, [cit. 2015-05-07].

[12] Cerone, A. a A. Maggiolo-Schettini: Time-based expressivity of time Petri nets for system specification. *Theoretical Computer Science*, ročník 216, č. 1, 1999: s. 1–53. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0304397598000085

[13] Laboratoire d'Informatique de paris 6: Using ITS modeler to design and analyze systems using discrete Time Petri nets. [online]. [přístup 2015-05-07]. Dostupné z: http://move.lip6.fr/software/DDD/coloane.php

[14] Hinz, S., K. Schmidt a Ch. Stahl: Transforming BPEL to Petri nets. In *Business Process Management*, 2005, s. 220–235. Dostupné z: http://www.researchgate.net/profile/Christian_Stahl/publication/221586005_Transforming_BPEL_to_Petri_Nets/links/0c960519145f366622000000.pdf

[15] Michael [Michael Westergaard]: CPN Tools 4: Place-Transition Petri Nets [online]. 2013, [cit. 2015-05-07]. Dostupné z: https://westergaard.eu/2013/08/cpn-tools-4-place-transition-petri-nets/

[16] Aalst, W. M. P. van der et al.: Soundness of workflow nets with reset arcs. In *Transactions on Petri Nets and Other Models of Concurrency III*, 2009, s. 50–70. Dostupné z: http://profs.info.uaic.ro/~otto/Referate/resetwf.pdf

[17] Dorda, M., Ing., Ph.D.: Úvod do Petriho sítí [online prezentace]. [cit. 2015-05-07]. Dostupné z: http://homel.vsb.cz/~dor028/Nekonvencni_metody_1.pdf

[18] Aalst, W. van der: Analysis of Process Models: Introduction, state space analysis and simulation in CPN Tools [online prezentace]. [cit. 2015-05-07]. Dostupné z: http://cpntools.org/_media/book/analysis.pdf

[19] Haddad, S.: Time and Timed Petri Nets [online prezentace]. 2011, [cit. 2015-05-07]. Dostupné z: http://www.lsv.ens-cachan.fr/~haddad/disc11-part1.pdf

[20] Aalst, W. van der: Hierarchical Petri nets [online prezentace]. [cit. 2015-05-07]. Dostupné z: http://cpntools.org/_media/book/hcpn.pdf

[21] Janoušek, V., Ing. a Doc. RNDr. M. Češka, CSc.: *Modelování objektů Petriho sítěmi*. Dizertační práce, Vysoké učení technické v Brně, Fakulta elektrotechniky a informatiky, 1998. Dostupné z: http://www.fit.vutbr.cz/~janousek/publications/phdthesis.pdf

[22] Valk, R.: Object petri nets. In *Lectures on Concurrency and Petri Nets*, 2004, s. 819–848. Dostupné z: http://disciplinas.stoa.usp.br/pluginfile.php/92436/mod_resource/content/1/Valk-PNObjects.pdf

[23] Lakos, C.: *From coloured Petri nets to object Petri nets*. Hamburg, 1995. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.45.1523&rep=rep1&type=pdf

[24] [kolektiv autorů z University of Hamburg]: Petri Nets Tools Database Quick Overview [online]. Dostupné z: https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html

[25] kolektiv autorů: CPN Tools [software]. [přístup 2015-05-07]. Verze 4.0.1. Dostupné z: http://cpntools.org/download

[26] AIS Group, Eindhoven University of Technology: CPN Tools Simulator (Simulator/CPN) [zdrojový kód softwaru]. [přístup 2015-05-07]. Dostupné z: https://svn.win.tue.nl/repos/cpntools/Simulator/trunk/simulator/sim/sim.sml

[27] [Eindhoven University of Technology a Deloitte & Touche Bakkenist]: ExSpect User Manual [manuál k softwaru]. 1999, [cit. 2015-05-07]. Dostupné z: http://www.exspect.com/ex641usermanual.pdf

[28] Basten, T., R. Bol, M. Voorhoeve: Simulating and Analyzing Railway Interlockings in ExSpect. [cit. 2015-05-07]. Dostupné z: http://www.es.ele.tue.nl/~tbasten/papers/simisl.pdf

[29] kolektiv autorů: PIPE: Platform Independent Petri Net Editor [software]. [přístup 2015-05-07]. Verze 4.3.0. Dostupné z: https://github.com/sarahtattersall/PIPE

[30] kolektiv autorů: WoPeD: Workflow Petri Net Designer [software]. [přístup 2015-05-07]. Verze 3.2.0.20140709. Dostupné z: http://woped.dhbw-karlsruhe.de/woped/?page_id=22

[31] kolektiv autorů: TAPAAL [software]. [přístup 2015-05-07]. Verze 3.1.3. Dostupné z: http://www.tapaal.net/download/

[32] kyrke [K. Y. Jørgensen] et al.: TAPAAL Wiki: Modelling Features [online]. [cit. 2015-05-07]. Dostupné z: http://wiki.tapaal.net/documentation/usermanual/modelling/features#transport_arcs

[33] Berthomieu, B. et al.: Tina Toolbox [TIme petri Net Analyzer] [software]. [přístup 2015-05-07]. Verze 3.3.0. Dostupné z: http://www.tapaal.net/download/

[34] Kolouch, J., Doc., Ing., CSc.: Stavové automaty [online]. [cit. 2015-05-07]. Dostupné z: http://www.urel.feec.vutbr.cz/~kolouch/pld/1_prednasky/kapitola06.html

[35] Šilhavík, P., Bc.: *Stavové automaty v C++ pro robotické aplikace*. Diplomová práce, České vysoké učení technické v Praze, Fakulta elektrotechnická, katedra řídící techniky, Praha, 2013. Dostupné z: https://rtime.felk.cvut.cz/~sojka/students/Dp_2013_silhavik.pdf

[36] Udacity: Simulating Nondeterminism [online videoklip]. [cit. 2015-05-07]. Dostupné z: https://www.udacity.com/course/viewer#!/c-cs262/l-48746090/e-48624730/m-48692646

[37] EventHelix.com: Hierarchical State Machine [online]. [cit. 2015-05-07]. Dostupné z: http://www.eventhelix.com/realtimemantra/hierarchicalstatemachine.htm

[38] Anand, M. a Samek, M.: Hierarchical State Machines - a Fundamentally Important Way of Design [online prezentace]. [cit. 2015-05-07]. Dostupné z: http://www.cis.upenn.edu/~lee/06cse480/lec-HSM.pdf

[39] Fakhroutdinov, K.: State Machine Diagrams [online]. [cit. 2015-05-07]. Dostupné z: http://www.uml-diagrams.org/state-machine-diagrams.html

[40] Sparx Systems: UML 2 State Machine Diagram. UML 2 Tutorial [online]. [cit. 2015-05-07]. Dostupné z: http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_statediagram.html

[41] Kříž, M. a L. Staněk: Automata editor version 2 [software]. 2013-04-26, [přístup 2015-05-07]. version 2. Dostupné z: http://sourceforge.net/projects/automataeditor/

[42] Bovel, J.: Visual Automata Simulator [software]. 11/24/2006, [přístup 2015-05-07]. Dostupné z: http://download.cnet.com/Visual-Automata-Simulator/3000-2053_4-54760.html

[43] Sparx Systems: Enterprise Architect [software]. [přístup 2015-05-07]. Verze 11.1.1110. Dostupné z: http://www.sparxsystems.com.au/products/ea/trial.html

[44] sparxsystems [Sparx Systems]: State Machine Simulation in Enterprise Architect [online]. 24 April 2013 [cit. 2015-05-07]. Dostupné z: http://community.sparxsystems.com/white-papers/692-state-machine-simulation-in-enterprise-architect

[45] Čápka, D.: 1. díl - Úvod do UML. UML. ITnetwork.cz [online]. [cit. 2015-05-07]. Dostupné z: http://www.itnetwork.cz/uml-uvod-historie-vyznam-a-diagramy

[46] Ambler S. W.: Introduction to the Diagrams of UML 2.X. Agile modeling [online]. [cit. 2015-05-07]. Dostupné z: http://www.agilemodeling.com/essays/umlDiagrams.htm

[47] Szturcová, D.: Objektově orientované technologie, Business proces, Diagram aktivit [online prezentace]. [cit. 2015-05-07]. Dostupné z: http://gis.vsb.cz/wikivyuka/images/7/73/OotxAD.pdf

[48] Sparx Systems: Acitivty Diagram [online]. [cit. 2015-05-07]. Dostupné z: http://www.sparxsystems.eu/resources/project-development-with-uml-and-ea/activity-diagram/

[49] Sparx Systems: BPMN Simulation [online]. [cit. 2015-05-07]. Dostupné z: http://www.sparxsystems.com/enterprise_architect_user_guide/9.2/model_simulation/bpmn_simulation.html

[50] GEAMBAŞU, C. V.: BPMN vs. UML Activity Diagram for Business Process Modeling. *Accounting and Management Information Systems*, ročník 11, č. 4, 2012: s. 637–651. Dostupné z: http://www.cig.ase.ro/articles/11_4_7.pdf

[51] Sparx Systems: Comparing UML Activities to BPMN Processes [online]. [cit. 2015-05-07]. Dostupné z: http://www.sparxsystems.com/enterprise_architect_user_guide/9.2/model_simulation/bpmn_simulation_comparison.html

[52] Oiry, H.: Frequently Asked Questions (under development). Business Process Modeling Notation (BPMN) Information [online]. [cit. 2015-05-07]. Dostupné z: http://www.omg.org/bpmn/Documents/FAQ.htm

[53] White, S. A.: Process Modeling Notations and Workflow Patterns. [cit. 2015-05-07]. Dostupné z: http://www.omg.org/bpmn/Documents/Notations_and_Workflow_Patterns.pdf

[54] Vejražková, Z., Bc.: *Business Process Modeling and Simulation: DEMO, BORM and BPMN*. Diplomová práce, Czech Technical University in Prague, Faculty of Information Technology, Department of Software Engineering, 2013. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/vejrazuz_2013dipl.pdf

[55] Oiry, H.: IMINFO - BPMN Business Process Modeling and Simulation with Enterpriste Architect. In Youtube [online videoklip]. 11. 10. 2013 [cit. 2015-05-07]. Dostupné z: https://www.youtube.com/watch?v=oI6xDtNXKuo

[56] Schweitzer, S.: BPMN Simulator [software]. [přístup 2015-05-07]. Dostupné z: https://code.google.com/p/bpmn-simulator/

[57] SIMUL8 Corporation: SIMUL8 - Manufacturing Simulation Demonstration. In Youtube [online videoklip]. 9. 5. 2012 [cit. 2015-05-07]. Dostupné z: https://www.youtube.com/watch?v=_JZl1Lnpuz8

[58] Visual Paradigm: Visual Paradigm [software]. [přístup 2015-05-07]. Verze 12.1. Dostupné z: http://www.visual-paradigm.com/download/

[59] Visual Paradigm: Turn BPMN Business Process Diagram into Movie. In Youtube [online videoklip]. 4. 11. 2013 [cit. 2015-05-07]. Dostupné z: https://www.youtube.com/watch?v=oF4VkkvIi7k

[60] Pergl, R., Ing., Ph.D.: Metoda BORM® a nástroj OpenCASE [online prezentace]. 2012, [cit. 2015-05-07]. Dostupné z: http://ccm.fit.cvut.cz/wp-content/uploads/2013/02/OpenCASE_FIT.pdf

[61] Balda, M.: BORM Portal [software]. [přístup 2015-05-07].

[62] CRAFT.CASE LIMITED: Craft.CASE [online software]. [přístup 2015-05-07]. Dostupné z: https://online.craftcase.com

[63] Enterpriste Enginnering Institute: Welcome to the website of the Enterprise Enginnering Institute [online]. [cit. 2015-05-07]. Dostupné z: http://www.demo.nl/

[64] ForMetis: Online DEMO modeling tool for process design and simulation [online software]. [přístup 2015-05-07]. Dostupné z: http://demoworld.nl/

[65] Ducasse, S. et al.: Syntax in a nutshell [online]. In *Pharo by Example*, 2009, ISBN 978-3-9523341-4-0, [cit. 2015-05-07]. Dostupné z: http://pharo.gforge.inria.fr/PBE1/PBE1ch4.html

[66] Cassou, D.: [osobní rozhovor o rozšíření programů s MVC architekturou]. Praha 2015-03-19.

[67] Ducasse, S. et al.: Reflection [online]. In *Pharo by Example*, 2009, ISBN 978-3-9523341-4-0, [cit. 2015-05-07]. Dostupné z: http://pharo.gforge.inria.fr/PBE1/PBE1ch15.html

[68] abergel [A. Bergel]: Roassal TRCompositeShape not sending callbacks [online hromadná e-mailová diskuse]. Duben 2015. Dostupné z: http://forum.world.st/Roassal-TRCompositeShape-not-sending-callbacks-td4814221.html

[69] abergel [A. Bergel] a P. Uhnák: Roassal TRCompositeShape regression [online hromadná e-mailová diskuse]. Březen–Duben 2015. Dostupné z: http://forum.world.st/Roassal-TRCompositeShape-regression-td4816726.html

[70] Birch, J.: Reversibility Patterns: Memento and Command [online blog]. July 23, 2009 [cit. 2015-05-07]. Dostupné z: http://www.colourcoding.net/blog/archive/2009/07/23/reversibility-patterns-memento-and-command.aspx

Příloha **A**

Seznam použitých zkratek

MVC	Model-view-controller – návrhový vzor oddělující data a zobrazení
GUI	Graphical user interface – Grafické uživatelské rozhraní
XML	Extensible Markup Language – rozšiřitelný značkovací jazyk
UML	Unified Modeling Language
BPMN	Business Process Modeling Notation
BORM	Business Object Relation Modeling
DEMO	Design and Engineering Methodology for Organizations
ČVUT	České vysoké učení technické v Praze

Příloha **B**

Obsah přiloženého CD

Součástí přiloženého CD je

- zdrojová forma práce ve formátu LaTeX s dalšími zdrojovými soubory,
- práce ve formátu PDF,
- obraz spustitelný v prostředí Pharo 4 obsahující implementaci a
- kompletní prostředí Pharo 4 se spustitelnými soubory.

```
readme.txt ..... stručný popis obsahu CD
impl ..... adresář se spustitelnou formou implementace
    doc ..... podklady pro dokumentaci v anglickém jazyce na webu
    exe ..... adresář se spustitelným prostředím Pharo 4 s implementací
        linux ..... adresář s prostředím Pharo 4 pro Linux s implementací
        osx ..... adresář s prostředím Pharo 4 pro OS X s implementací
        win .... adresář s prostředím Pharo 4 pro Windows s implementací
    img ..... adresář s obrazem pro Pharo 4 s implementací
src ..... zdroje textu práce
    abstract.pdf ..... zadání ve formátu PDF
    bp_bliznicenko.bib ... použité informační zdroje ve formátu BibTeX
    bp_bliznicenko.tex ..... text práce ve formátu LaTeX
text ..... text práce
    bp_bliznicenko.pdf ..... text práce ve formátu PDF
```