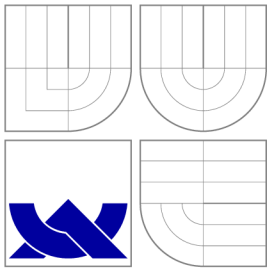
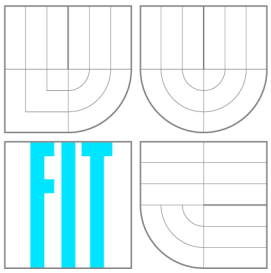


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KLIENT-SERVER APLIKACE PRO MODELOVÁNÍ A SIMULACI NA BÁZI DEVS

CLIENT-SERVER APPLICATION FOR DEVS-BASED MODELLING AND SIMULATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN BRÁZDIL

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.

BRNO 2013

Abstrakt

Tato bakalářská práce se zabývá analýzou, návrhem a implementací aplikace pro modelování a simulaci na bázi DEVS. Jedná se o klient - server aplikaci, přičemž klient je implementován jako samonosná aplikace v jazyku Java. Server se zakládá na jazyku Smalltalk (přesněji Squeak) a vychází z existující aplikace SmallDEVS. V práci je čtenář obeznámen se základy systémů s diskrétními událostmi (DEVS) a Petriho sítí pro modelování a simulaci. Práce se rovněž věnuje návrhu uživatelsky přívětivého grafického rozhraní pro klientskou část, respektive editor.

Abstract

This bachelor's thesis describes the analysis, design and implementation of an application for DEVS - based modelling and simulation. It is a client - server application, where the client is implemented as a seperate desktop Java application. Server is based on the Smalltalk language (more precisely, Squeak) and builds on existing application SmallDEVS. The reader is acquainted with the basics of discrete event systems specification (DEVS) and Petri nets used for modeling and simulation. The thesis is also dedicated to the design of user - friendly graphical interface on the client side, respectively editor.

Klíčová slova

klient, server, DEVS, Petriho síť, Java, Smalltalk, editor, modelování, simulace, SmallDEVS

Keywords

client, server, DEVS, Petri net, Java, Smalltalk, editor, modelling, simulation, SmallDEVS

Citace

Martin Brázdil: Klient-server aplikace pro modelování a simulaci na bázi DEVS, bakalářská práce, Brno, FIT VUT v Brně, 2013

Klient-server aplikace pro modelování a simulaci na bázi DEVS

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Vladimíra Janouška, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Brázdil
13. května 2013

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu bakalářské práce, doc. Ing. Vladimíru Janouškovi, Ph.D., za trpělivost a systematické vedení při vývoji obou aplikací a tvorbě technické zprávy. Také bych rád poděkoval své přítelkyni za projevenou podporu a trpělivost v celém průběhu práce, za pomoc při tvorbě technické zprávy a časté testování klientské části.

Obsah

Kapitola 1

Úvod

Podstatnou část oblasti aplikace informačních technologií do reálného světa tvoří modelování a simulace. V případě, kdy je třeba získat data (informace) o jistém systému za použití systému odlišného, hovoříme o *modelu*. Systémem může být libovolná entita jako křižovatka, rozvoj populace lidstva nebo i chemická reakce. Modelováním se tedy rozumí proces tvorby modelu (systému) při zvolení určité míry abstrakce na základě reálných požadavků či předlohy. Simulací takto vytvořeného modelu pak lze obdržet specifické informace o jeho chování v čase a tím například předvídat budoucí vývoj reálného systému. Z výše uvedeného plyne, že můžeme pracovat i s nedostupnými reálnými systémy a tím získávat potřebné informace pro jeho vytvoření či odladění.

Pro tvorbu rozličných druhů modelů existuje celá řada specializovaných aplikací a nástrojů a doufám, že se díky mé bakalářské práci rozroste o další. Obecně lze říci, že nástroje pro modelování a simulaci disponují grafickým rozhraním, které uživateli rozšiřuje možnosti samotné tvorby modelu a vyhodnocení simulace. Problémem je ovšem přizpůsobit toto rozhraní obvykle rozsáhlému a komplikovanému modelování. Z uvedené problematiky, mimo jiné, vyplývá i cíl mé práce. Chci vytvořit přehlednou aplikaci pro modelování a simulaci tak, aby měl uživatel snadný přístup ke všem důležitým operacím a informacím, přičemž vytvořené modely by byly přístupné více uživatelům současně.

Cíl mé bakalářské práce také plyne z jeho názvu, kde figuruje sousloví „klient - server“, což značí potřebu vytvořit dvě samostatné aplikace, které mezi sebou komunikují na základě předem daných pravidel (komunikační protokol). Toto řešení umožňuje odstínění klienta od uchovávání vytvořených modelů a provádění jejich simulací. Servery také obvykle poskytují možnost obsluhy více klientů a tím mezi nimi sdílet totožná data i operace s daty, což je základ týmové spolupráce více uživatelů. V následujících kapitolách popisuji celý postupný průběh vývoje obou částí, klienta (editoru) i serveru.

Struktura této technické zprávy reflektuje všechny základní etapy vývoje aplikací, nebo obecně softwaru. Vždy se nejprve začíná analýzou a ujasněním požadavků ze strany zákazníka, o čemž pojednává kapitola 2. Po ní následuje fáze návrhu aplikace, ve které musí vývojář vytvořit koncept jádra samotné aplikace a případně i uživatelského (grafického) rozhraní pro interakci s uživatelem. Návrhem, a všem problémů s ním spojeným, je zasvěcena kapitola 3. Jakmile má vývojář ujasněny požadavky a návrh aplikace, může přistoupit ksamotné realizaci (implementaci), kterou se zabývá kapitola 4. Realizace aplikace bývá obvykle propojena s poslední etapou, a sice testováním (kapitola 5). V této fázi vyvstává řada nových informací, lépe řečeno problémů, k provedené implementaci, ba dokonce i návrhu. Tyto informace slouží vývojáři jako podklad pro provedení oprav v doposud vytvořeném díle.

Kapitola 2

Analýza požadavků

Jedním z počátečních bodů vývoje aplikací je ujasnit si a analyzovat zadání od zákazníka, respektive bakalářské práce. Proto v této kapitole budou diskutovány jeho stěžejní body.

V prvních dvou částech je rozepsána problematika modelování a diskrétní simulace systémů na bázi DEVS (kap. 2.1), nebo-li Discrete Event System Specification, a systémů vytvořených za použití Petriho sítí (kap. 2.2).

Třetí, předposlední, část popisuje možnosti komunikace (kap. 2.3) a spojení mezi serverem a jeho případnými klienty. Především se věnuje konkurentnímu spojovanému TCP/IP serveru, který je základem první části mé práce.

Poslední část (kap. 2.4) je zaměřena na existující řešení modelování a simulace systémů na bázi DEVS, konkrétně aplikaci SmallDEVS, a také systémů na bázi Petriho sítí, aplikaci Renew.

2.1 Systém s diskrétními událostmi (DEVS)

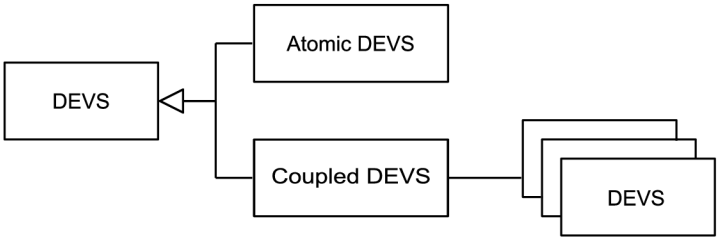
Systém s diskrétními událostmi (Discrete Event System Specification zkráceně DEVS) byl představen Dr. Bernardem P. Zeiglerem počátkem 70. let minulého století za účelem tvorby hierarchických modulárních diskrétních systémů. V diskrétním systému se jeho stav mění *skokově*, to znamená, že na spojité časové ose existuje konečný počet bodů, ve kterých dochází ke změnám stavu. Tyto změny jsou okamžité, tj. mají nulové trvání, a nazývají se událostmi.

V následující kapitole 2.1.1 je popsán základní formalismus DEVS v rozšíření o porty, který je simulován pomocí *abstraktního simulátoru* (viz. kapitola 2.1.2).

2.1.1 Hierarchický systém (model)

V souvislosti s hierarchií modelů v systému existují dva druhy DEVS modelů – *atomické* a *složené*. První jmenované jsou chápány jako základní, dále nedělitelné jednotky, které jsou vnořeny a případně vzájemně propojeny ve spojovaných modelech. Spojované modely se mohou samy o sobě chovat jako atomické, jestliže dojde k jejich zanoření (viz. obrázek 2.1). Stav výsledného systému je tedy určen stavy všech jeho dílčích systémů, které na sebe působí svými externími událostmi.

Základní definici DEVS modelů rozšíříme o použití *portů*, které se reálně využívají v naprosté většině implementací formalismu DEVS především pro svoje zjednodušení specifikace modelů. Porty jsou definovány jako dvojice jeho unikátního názvu a hodnoty. Rozlišují se dva druhy – vstupní a výstupní.



Obrázek 2.1: Znázornění hierarchického modelu.

Simulace uvedeného systému je částečně popsána u obou druhů modelů a rozvedena v kapitole 2.1.2.

Atomický DEVS

Klasický základní (atomický) DEVS model s porty je definován jako sedmice

$$DEVS = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta), \text{ kde}$$

$X = \{(p, v)|p \in InPorts, v \in X_p\}$ je *množina vstupních portů* a jejich hodnot,
$Y = \{(p, v)|p \in OutPorts, v \in Y_p\}$ je *množina výstupních portů* a jejich hodnot,
S je *množina sekvenčních stavů*,
$\delta_{ext}: Q \times X \to S$ je *externí přechodová funkce* určující reakci na externí událost, kde $Q = \{(s, e)|s \in S, 0 \leq e \leq ta(s)\}$ je *množina úplných stavů* a e značí čas uplynutý od poslední události,
$\delta_{int}: S \to S$ je *interní přechodová funkce* definující chování vnitřního konečného stavového automatu,
$\lambda: S \to Y$ je *výstupní funkce*,
$ta: S \to R^+_{0,\infty}$ je *funkce posuvu času*.

Systém se v daném čase nachází v určitém stavu s. Jestliže se nevyskytne žádná externí událost, pak systém zůstává v totožném stavu s po dobu $ta(s)$. Funkce posuvu času $ta(s)$ může obecně nabývat hodnot 0, systém nereaguje na příjem externích událostí – stav s je *přechodný stav*, nebo hodnoty ∞, což způsobí, že system setrvává ve stavu s, než jej uvolní externí událost – stav s je *pasivní*. Při uplynutí zbývajícího času $e = ta(s)$ se vygeneruje výstup na $\lambda(s)$ a systém změní stav na $\delta_{int}(s)$.

V případě, kdy se externí událost $x \in X$ vyskytne ještě před vypršením zbývajícího času e, pak systém změní stav na $\delta_{ext}(s, e, x)$. Výstup $\lambda(x)$ se ovšem vloží na výstup až v čase $e = ta(s)$. V situaci, kdy vyprší zbývající čas (má se provést δ_{int}) a současně dojde k vyvolání externí události (má se provést δ_{ext}), je provedena druhá jmenovaná.

Jestliže se nevyskytne externí událost před dalším posuvem času, interní přechodová funkce určuje nový stav systému. Zato externí přechodová funkce definuje nový stav systému, ve chvíli, kdy se externí událost vyskytne. [8]

Složený DEVS

Klasický složený DEVS model s porty je definován jako osmice

$$N = (X, Y, D, \{M_d|d \in D\}, EIC, EOC, IC, Select), \text{ kde}$$

$X = \{(p, v)|p \in InPorts, v \in X_p\}$ je *množina vstupních portů* a jejich hodnot,

$Y = \{(p, v) | p \in OutPorts, v \in Y_p\}$ je *množina výstupních portů* a jejich hodnot,
D je *množina unikátních názvů* dílčích modelů,
$\{M_d | d \in D\}$ je *množina dílčích modelů* se vstupy $X_d = \{(p, v) | p \in InPorts_d, v \in X_{p,d}\}$ a výstupy $Y_d = \{(p, v) | p \in OutPorts_d, v \in Y_{p,d}\}$,
$EIC \subseteq \{((N, ip_N), (d, ip_d)) | ip_N \in InPorts, d \in D, ip_d \in InPorts_d\}$ je *množina externích vstupních propojení* propojující vstupní porty systému se vstupními porty dílčích modelů,
$EOC \subseteq \{((d, op_d)), (N, op_N) | op_N \in OutPorts, d \in D, op_d \in OutPorts_d\}$ je *množina externích výstupních propojení* propojující výstupní porty dílčích modelů s výstupními porty systému,
$IC \subseteq \{((a, op_a)), (b, ip_b) | a, b \in D, op_a \in OutPorts_a, op_b \in OutPorts_b\}$ je *množina interních propojení* propojující výstupní porty se vstupními porty dílčích modelů,
$Select\colon 2^D - \{\} \to D$ je *preferenční funkce*

s těmito omezeními

1. $\forall((d, op_d), (e, ip_d)) \in IC \Rightarrow d \neq e$ omezující propojení mezi výstupním a vstupním portem totožného dílčího modelu,
2. $\forall((N, ip_N), (d, ip_d)) \in EIC\colon X_{ip_N,N} \subseteq X_{ip_d,d}$,
3. $\forall((d, op_d), (N, op_N)) \in EOC\colon Y_{op_d,d} \subseteq Y_{op_N,N}$,
4. $\forall((a, op_a), (b, ip_b)) \in IC\colon Y_{op_a,a} \subseteq X_{ip_b,b}$.

Jakmile systém N obdrží vstupní událost na některém ze vstupních portů, je předána na vstupy (vstupní porty) dílčích modelů skrze externích vstupních propojení EIC. Po zpracování událostí se na výstupech (výstupních portech) dílčích modelů objeví výstupní události. Ty mohou být předány do vstupních portů dalších dílčích modelů za použití interních propojení IC nebo do výstupních portů systému N prostřednictvím externích výstupních spojení EOC.

Vzhledem k možnostem propojení dílčích modelů může nastat situace, kdy ve stejném modelovém čase je dostupný přechod do nového stavu i ve více modelech současně. V tomto případě se použije preferenční funkce *Select*, která tedy slouží k sekvenčnímu výběru provedení přechodů modelů, dle jejich priority.

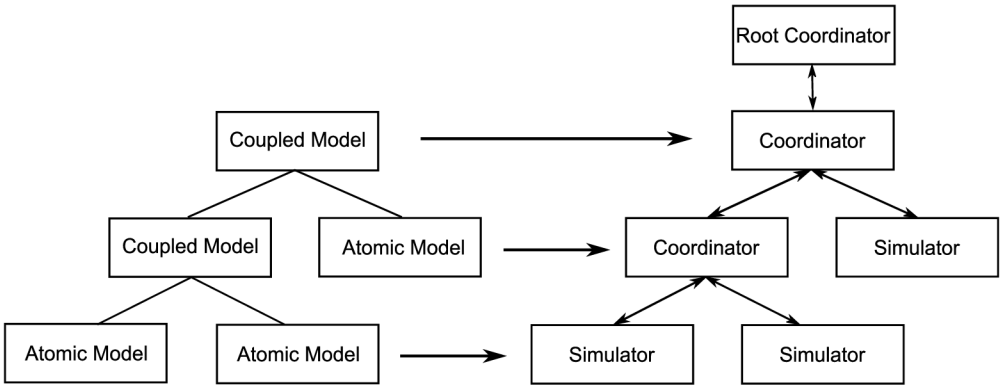
2.1.2 Simulace

Simulace je, stejně jako modely, uspořádána do *hierarchického stromu* závislostí, kde každý model je mapován na vlastní *abstraktní simulátor*. Každý z těchto simulátorů náleží nadřazenému, vyjma kořene stromu zvaného *kořenový koordinátor*. Jak naznačuje obrázek 2.2, složené DEVS modely jsou mapovány na *koordinátory*, které mohou obsahovat další abstraktní simulátory, zatímco atomické DEVS modely na *simulátory*.

Abstraktní simulátory využívají pro zajištění diskrétní simulace především dvou časových proměnných – čas provedení poslední události t_{last} a následující plánované události t_{next}. Poslední jmenovaný je použit ke korektní synchronizaci událostí v rámci jeho a jemu nadřízenému abstraktnímu simulátoru.

Jednotlivé simulátory mezi sebou komunikují pomocí následujících *zpráv*:

(i, t) *inicializační zpráva* je zaslána z nadřazeného simulátoru ke všem jeho podřízeným na začátku simulace.



Obrázek 2.2: Znázornění mapování hierarchického DEVS modelu na abstraktní hierarchický simulátor [8].

$(*, t)$ *vnitřní stavová přechodová zpráva* je zaslána z nadřazeného simulátoru k jeho podřízeným jako požadavek pro provedení interního přechodu a tedy vystavení jeho výstupu.

(y, t) *výstupní zpráva* je zaslána z podřízeného simulátoru k jeho nadřazenému, který obstarává její distribuci, jako informace o dostupnosti výstupu.

(x, t) *vstupní zpráva* je zaslána z nadřazeného simulátoru k jeho podřízeným pro provedení externí přechodové funkce.

$(done, t_{next})$ *potvrzující zpráva* zasílaná z podřízeného simulátoru k nadřazenému jako informace o času provedení příchozí zprávy.

Kořenový koordinátor je zodpovědný za korektní inicializaci dílčích abstraktních simulací a provádění jednotlivých kroků simulace. Ve smyčce zasílá vnitřní stavové přechodové zprávy $(*, t)$ a následně čeká na potvrzující zprávu $(done, t_{next})$.

V následujících kapitolách jsou popsány simulace jednotlivých druhů komponent, a sice atomického a složeného modelu.

Atomický DEVS

Časové proměnné simulátoru lze zapracovat buď do souvislosti s *funkcí posuvu času* $ta(s)$ tak, že

$$t_{next} = t_{last} + ta(s), \tag{2.1}$$

nebo, pokud je dostupný simulační čas t, do zjištění uplynulého času od poslední události e a zbývajícího času do následující plánované události σ:

$$e = t - t_{last} \tag{2.2}$$

$$\sigma = t_{next} - t = ta(s) - e. \tag{2.3}$$

Jakmile dorazí inicializační zpráva (i, t), simulátor reaguje inicializací časů t_{last}, dle rovnice 2.2, a t_{next}, dle rovnice 2.1. Při obdržení a provádění interní stavové přechodové funkce $(*, t)$ se vygeneruje výstup a dojde k zaslání výstupní zprávy (y, t). Následně se vykoná interní přechodová funkce $\delta_{int}(s)$ modelu a do času poslední provedené události t_{last} je přiřazena aktuální hodnota simulačního času t a čas následující plánované události t_{next} se

dopočítá dle rovnice 2.1. Na poslední možnou zprávu od nadřazeného simulátoru (koordinátoru), vstupní zprávu (x, t), simulátor zareaguje provedením externí přechodové funkce $\delta_{ext}(s, e, x)$, kde uplynulý čas e je dopočítán dle rovnice 2.2, a nastavení obou simulačních časů stejným způsobem jako při obdržení výstupní zprávy (y, t).

Složený DEVS

Jak již bylo řečeno, složený DEVS model využívá pro simulaci koordinátor, který zajišťuje synchronizaci simulací a distribuci příchozích zpráv do svých dílčích komponent. Mimo výše uvedené časové proměnné *poslední události*, která je definována u koordinátoru jako

$$t_{last} = max(t_{last,d} | d \in D), \tag{2.4}$$

a *následující plánované události*, definované jako

$$t_{next} = min(t_{next,d} | d \in D), \tag{2.5}$$

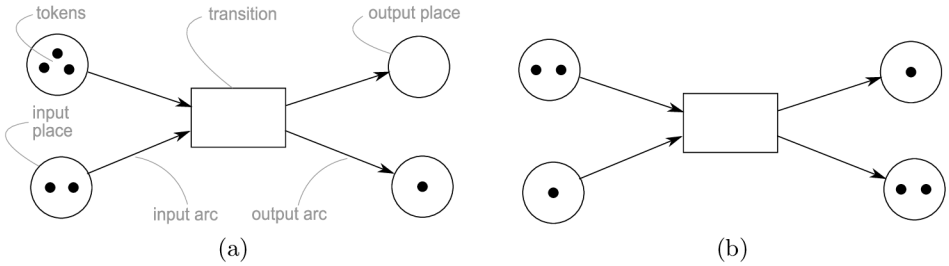
koordinátor používá seznam událostí. Tento seznam uchovává dvojici – dílčí model d a čas jeho následující plánované události $t_{next,d}$, kde položky jsou řazeny dle uvedeného času, případně dle jejich priority.

Po přijetí inicializační zprávy (i, t) ji koordinátor distribuuje všem svým dílčím modelům, obnoví seznam událostí a inicializuje své časová proměnné dle 2.4 a 2.5. Jakmile je obdržena vnitřní stavová přechodová zpráva $(*, t)$, tak koordinátor zvolí odpovídající bezprostřední dílčí model (z první položky seznamu událostí) a přepošle mu tuto zprávu. Reakcí na obdrženou výstupní zprávu (y, t) nad výstupním portem op_d od dílčího modelu d je rozeslání vstupní zprávy (x, t) dalším dílčím modelům, které jsou propojeny s modelem d, skrze interní propojení IC. A dále pokud je výstupní port op_d propojen s výstupním portem op_N koresponduj ícího spojovaného modelu N, dle externích výstupních propojení EOC, je výstupní zpráva (y, t) přeposlána nadřazenému koordinátoru. Po příjmu vstupní zprávy (x, t) nad vstupním portem ip_N složeného modelu N se tento požadavek distribuuje všem dílčím modelům s ním spojených pomocí externích vstupních propojení EIC. Podřízená abstraktní simulace může zaslat potvrzující zprávu $(done, t_{next})$, na kterou koordinátor obnoví seznam událostí, upraví časové proměnné dle rovnic 2.4 a 2.5 a přepošle ji svému nadřazenému koordinátoru.

2.2 Petriho sítě

Petriho sítě jsou pro svoji grafickou názornost a jednoduchost oblíbeným prostředkem modelování *diskrétních*, případně paralelních, *systémů*, které poskytují kvalitní formální analyzovatelnost a srozumitelné modelování synchronizace a komunikace procesů. Tento formální prostředek je dílem C. A. Petriho z 60. let 20. století. [2]

Systém Petriho sítí je definován *konečnou množinou míst* (places) propojených pomocí *hran* (arcs) s *přechody* (transitions) a opačně. Místa mohou obsahovat stavové informace v podobě *značek* (tokens). Přechod se sestává ze vstupních hran (napojeny na vstupní místa) a výstupních hran (napojeny na výstupní místa). Přechod je *proveditelný* právě tehdy, když všechny jeho vstupní hrany obsahují značky. Po provedení přechodu se ze všech vstupních míst odebere značka a přiřadí všem výstupním místům daného přechodu (viz. obrázek 2.3). Stav tohoto systému je definován rozdělením značek do míst.



Obrázek 2.3: Provedení přechodu P/T Petriho sítě (vč. popisků). (a) – stav před provedením, (b) – stav po provedení přechodu.

Tento popis se vztahuje k základním Petriho sítí, ale vzhledem k potřebě zvýšit modelovací schopnosti, bylo doposud vyvinuto několik dalších variant těchto sítí. Tyto varianty jsou řazeny do dvou skupin:

1. *Nízkoúrovňové Petriho sítě* (Low Level Petri Nets zkráceně LLPN), k nimž se řadí např. C - E (Condition - Event Petri Nets) Petriho sítě nebo jejich zobecnění v podobě P/T (Place/Transtion Petri Nets) Petriho sítě, což jsou ony základní sítě popsány výše.

2. *Vysokoúrovňové Petriho sítě* (High Level Petri Nets zkráceně HLPN), mezi které patří predikátové (Predicate - Transition Petri Nets), barvené (Coloured Petri Nets) nebo objektově orientované sítě (Object Oriented Petri Nets nebo-li OOPN). Právě tato skupina je rozvedena v kapitole 2.2.1.

2.2.1 Vysokoúrovňové Petriho sítě

P/T Petriho sítě tak, jak byly výše definovány, jsou vhodnější spíše pro tvorbu systému s vyšší mírou abstrakce, protože i jednoduché operace (např. aritmetické operace) se musí poměrně složitě modelovat. Pokud tedy potřebujeme podrobněji modelovat reálný systém, je potřeba zvolit právě vysokoúrovňové Petriho sítě (dále budu používat zkratku HLPN z anglického názvu High Level Petri Nets).

Prvním zástupcem jsou *predikátové Petriho sítě* (Predicate - Transition Petri Nets), jejichž značky v místech definují libovolná data. Tato data jsou v přechodech případně testována a vyhodnocována. Pak se Petriho síť stává dostatečně obecným formalismem pro popis podrobných systémů, ovšem za ztráty určité jednoduchosti formální analýzy z P/T sítí. Tento neduh odstraňují *barvené Petriho sítě* (popsané v následující podkapitole 2.2.2), respektive usnadňují proces interpretace analýzy pomocí invariantů, jinak jsou sémanticky podobné predikátovým.

Dosud uváděné sítě byly nehierarchickými formalismy Petriho sítí, proto byly vhodnější spíše pro méně rozsáhle systémy. S použitím hierarchického strukturování lze docílit znovupoužitelnosti a tím rozšířit vyjadřovací sílu Petriho sítí. Dalším evolučním krokem při přizpůsobování a rozšiřování formalismů Petriho sítě je zavedení *objektové orientace* (Object Oriented Petri Nets, nebo-li OOPN), což umožnilo vyrovnat se vyšším programovacím jazykům. Tyto sítě jsou ovšem nad rámec mé práce, proto nejsou dále uvažovány.

2.2.2 Barvené Petriho sítě

Barvené Petriho sítě (dále CPN z anglického názvu Coloured Petri Net) rozšiřují základní definici Petriho sítí o mi (systém) *barev* Σ, obdobu datových typů klasických programovacích jazyků, která se navazuje na jednotlivé značky. Manipulací s barvami značek při průchodu přechody, za použití inskripčních jazyků, lze docílit omezení proveditelnosti a případné změny značek v přechodu. Inskripční jazyk je odvozený od funkcionálního jazyku SML (Standard ML).

Nehierarchická CPN se, dle K. Jensena [4], formálně definují jako n-tice

$$CPN = (\Sigma, P, T, A, N, C, G, E, IN), \text{ kde} \tag{2.6}$$

Σ je konečná množina konečných, neprázdných typů zvaná *množina barev*,
P je konečná *množina míst*,
T je konečná *množina přechodů*, přičemž může být $P \cup T = \emptyset$,
A je konečná *množina hran*, kde $P \cap T = P \cap A = T \cap A = \emptyset$,
N je *uzlová funkce* definovaná jako $N\colon A \to (P \times T) \cup (T \times P)$,
C je *funkce barev* definovaná jako $C\colon P \to \Sigma$,
G je *strážní funkce* definovaná jako $G\colon T \to$ *výraz*, kde

$$\forall t \in T\colon \left[Type\big(G(t)\big) = Bool \wedge Type\Big(Var\big(G(t)\big)\Big) \subseteq \Sigma\right],$$

E je *funkce hranových výrazů* definovaná jako $E\colon A \to$ *výraz*, kde

$$\forall a \in A\colon \left[Type\big(E(a)\big) = C\big(p(a)\big)_{MS} \wedge Type\Big(Var\big(E(a)\big)\Big) \subseteq \Sigma\right], \; p(a) \text{ je místo v } N(a),$$

IN je *inicializační funkce* definovaná jako $IN\colon P \to$ *výraz*, kde

$$\forall p \in P\colon \left[Type\big(IN(p)\big) = C(p)_{MS} \wedge Type\big(IN(p)\big) = \emptyset\right].$$

Typ (barva) proměnné v je značen jako $Type(v)$, $Bool$ je pravdivostní typ nabývající hodnot (`true` a `false`), $Var(výraz)$ je množina proměnných daného *výrazu* a symbol S_{MS} je *multimnožina* nad množinou S. Multimnožina m nad množinou S je funkce $m\colon S \to \mathbb{N}$. Obvykle je definována formální sumou

$$\sum_{s \in S} m(s)'s, \text{ kde } m(s) \in \mathbb{N} \text{ je počet výskytů prvku v multimnožině } m. \tag{2.7}$$

Nad multimnožinou jsou definovány operace sčítání, skalárního násobení, porovnávání, odčítání a kardinality. [4]

Uzlová funkce N přiřazuje hranám $a \in A$ zdrojové a cílové uzly, kde uzly musí být různých druhů (místa a přechody). Na rozdíl od klasických P/T Petriho sítí lze mít mezi totožnými uzly více shodně směřujících hran. Funkce barev C definuje místu $p \in P$ množinu přípustných barev $C(p)$, přičemž každá značka musí být typu (barvy) z této množiny. Strážní podmínky přechodu $t \in T$, definované strážní funkcí G, jsou predikáty. Funkce hranových výrazů E umožňuje navázat multimnožinu obecných výrazů hraně $a \in A$, ve kterých mohou být použity konstanty, proměnné nebo λ-výrazy, které jsou vyhodnoceny na přirozené číslo určující počet značek (případně barvy značek). Pro počáteční inicializaci míst slouží inicializační funkce IN.

Modelování a simulace barvených Petriho sítí

Obecné modelování systému pomocí Petriho sítí je založeno na navázaní částí systému na prvky Petriho sítě (místa, přechody a značky). Je vhodné si uvědomit, že jediné dy-

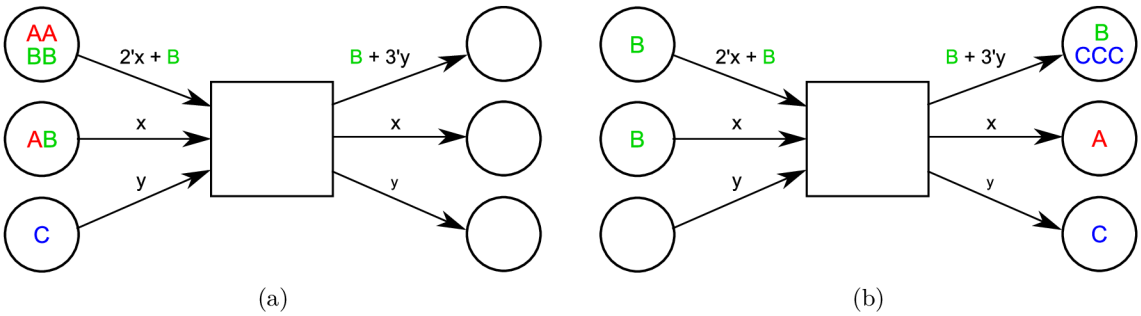
namické součásti Petriho sítí jsou značky, které vznikají a zanikají za použití přechodů. Přechody však vyznačují jediný aktivní, myšleno měnící stav systému, prvek.

Na obrázku 3.6 je znázorněn jednoduchý systém modelovaný za použití CPN. Tato ukázka odpovídá použití tří datových typů (barev) značek, jmenovitě `A`, `B` a `C`. Důležitou částí je také použití proměnných a multimnožin v rámci hran, na které se *navazují* (přiřazují) značky. Jedinou možností navázání je v daném případě takové, kdy proměnná x nese značku s barvou `A` a proměnná y zase značku barvy `C`. Výraz $2'x+$`B` je vyhodnocen jako multimnožina obsahující dvě značky s barvou navázanou na proměnnou x, tedy `A`, a jednu značku s barvou `B`. Obecný tvar hranového výrazu v CPN se zakládá na definici multimnožiny 2.7 a má tvar:

$$n'_1 c_1 + n'_2 c_2 + \ldots + n'_m c_m, \text{ kde}$$

$n_1, n_2, \ldots, n_m$ jsou proměnné, konstanty, nebo funkce (λ-výrazy), které jsou vyhodnoceny jako *počet značek* (jedná se o přirozená čísla $\mathbb{N}$),

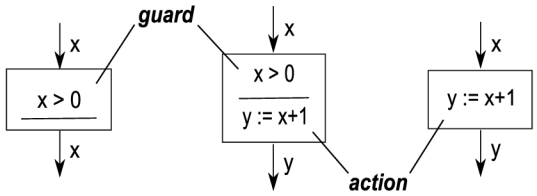
$c_1, c_2, \ldots, c_m$ jsou proměnné, konstanty, nebo funkce (λ-výrazy), které jsou vyhodnoceny jako *barvy značek* (jedná se o barvy z konečné množiny barev Σ).



Obrázek 2.4: Provedení přechodu v CPN, kde x je navázán na značku s typem `A`, y je navázáno na značku s typem `C` a zbylé výrazy na hranách jsou multimnožinami. (a) – stav před provedením, (b) – stav po provedení přechodu.

Simulace CPN se zakládá na opakování jednotlivých kroků v rámci dané simulace. *Krok simulace* nejprve otestuje proveditelnost jednotlivých přechodů, z nichž jeden je zvolen a proveden. *Proveditelnost* kroku určuje jednak možnost odebrat ze všech vstupních míst značky definované vstupní hranou na daný přechod a jednak splnění strážní podmínky. Po provedení přechodu jsou na všech výstupních místech značky dle výrazů na výstupních hranách.

CPN mohou být, mimo jiné, rozšířeny o *testovací hrany* obvykle značící se jako obousměrné šipky. Na rozdíl od vstupních/výstupních hran se jejich sémantika liší v tom, že *neodebírají*, respektive ponechávají, značky v místech. Tato vlastnost umožňuje sdílet persistentní data (značky) mezi více přechody současně. Dalším často používaným rozšířením CPN je rozšíření definice přechodu o akci. *Akce* přechodu umožňuje, stejně jako strážní podmínka, použití obecného výrazu, obvykle se zde vyskytují přiřazovací příkazy. Slouží tedy především ke zpřehlednění modelu oddělením akcí pro zjištění proveditelnosti přechodu a samotného provádění přechodu. Graficky je stráž od akce oddělena horizontální úsečkou, což znázorňuje obrázek 2.5.



Obrázek 2.5: Anotace strážní podmínky a akce přechodu [3].

2.3 Způsoby komunikace

Ve své podstatě existují dva druhy transportních služeb, *spojované* a *nespojované*. Mezi spojované se řadí protokol *TCP* (Transmission Control Protocol) a do skupiny nespojovaných protokol *UDP* (User Datagram Protocol), který je ovšem nespolehlivý a tedy nevyhovuje zadání. Protokol TCP oproti tomu implementuje spolehlivý přenos a bude popsán v následující kapitole 2.3.1.

2.3.1 Konkuretní server nad TCP/IP

Jak již bylo řečeno, protokol TCP je spolehlivý spojovaný transportní protokol, který pracuje s virtuálními okruhy a předpokládá, že spojení je před samotným přenosem dat již navázáno. Používá se společně s protokolem *IP* (Internet Protokol), kde IP obstarává samotný přenos dat a TCP zajišťuje ono spolehlivé připojení (zasílání dat), neboli zajistí jejich rozdělení do menších jednotek (*paketů*) a doručení ve správném pořadí a bez jejich ztráty.

Server může být buď iterativní, který zpracovává požadavky postupně, nebo konkurentní, který naopak požadavky zpracovává souběžně. *Iterativní server* není v praxi příliš využíván a to především z důvodu problémů s ním spojených. A sice jestliže server zpracovává požadavek od klienta, pak ostatní pokusy o připojení jsou řazeny do fronty a některé z nich jsou případně i odmítnuty [7].

Konkurentní server tvoří hlavní proces (viz. obrázek 3.2), který naslouchá na určitém portu a přijímá požadavky na spojení a navazuje je. Pro zpracování daného připojení (klienta) se vytvoří nový proces, který je navázán na vlastní *schránku*[1], a běží nezávisle na ostatních připojeních.

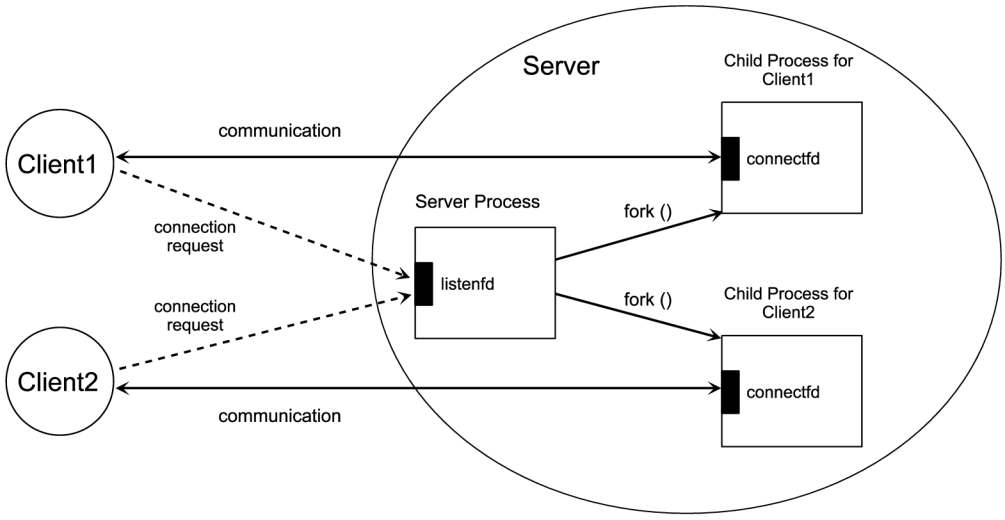
Tento druh serveru tedy teoreticky umožňuje obsluhovat neomezený počet klientů, ovšem limitujícím faktorem je zde hardwarové vybavení koncového serveru a to jak z pohledu náročnosti výpočtů, které jsou klientům dostupné, tak z pohledu paměťové náročnosti pro udržení tolika spojení.

2.4 Existující řešení

Pro modelování a simulaci existuje několik[2] nástrojů disponujících různými technologiemi. V případě modelování na bázi DEVS jsem se zaměřil na *SmallDEVS* (viz. 2.4.1) vzhledem ke skutečnosti, že se jedná o výchozí a základní stavební kámen mé práce. Další inspiraci jsem čerpal z nástroje pro modelování Petriho sítí zvaného *Renew*, popsán v kapitole 2.4.2.

[1] Schránka, implementována rozhraním BSD sockets, je deskriptor umožňující virtuální spojení mezi komunikačními uzly. Definuje ji dvojice IP adresa a číslo portu.

[2] Další DEVS nástroje je možné nalézt na http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm a nástroje pro editaci a simulaci Petriho sítí na http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html.



Obrázek 2.6: Připojování klientů ke konkurentnímu serveru.

2.4.1 Aplikace SmallDEVS

Jak už název SmallDEVS napovídá, jde o aplikaci vytvořenou v implementačním prostředím *Smalltalk* umožňující interaktivní inkrementální editaci a simulaci modelů na bázi DEVS. Aplikace byla v průběhu let podrobena několika úpravám a rozšířením o další funkcionality (např. interaktivní editace modelů DEVS, OOPN interpret aj.) [3]. Vývoj a správu zajišťuje Ústav inteligentních systémů na FIT VUT v Brně.

SmallDEVS je založen na *prototypových* objektech využívajících základní vlastnosti Smalltalku – interaktivního a experimentálního programování (více v 4.1.1), což také umožňuje inkrementální editaci modelů za běhu (při spuštěné simulaci). Grafické uživatelské rozhraní uvedeného editačního a simulačního nástroje také disponuje stromovým prohlížečem repozitáře, jež umožňuje rychlý a přehledný přístup k modelům a dalším komponentám. Dalšími prvky jsou upravené inspektory objektů Smalltalku, které umožňují přímou editaci atributů prototypového objektu, a samozřejmě editory spojovaných a atomických DEVS modelů. S aplikací lze ovšem pracovat i bez uvedených prvků za použití kvalitně navrženého aplikačního rozhraní jádra, čehož využívá i má implementace serveru.

2.4.2 Aplikace Renew

Jedná se o multi - formalisní editor a simulátor především pro *Reference nets*, což jsou objektově orientované vysokoúrovňové Petriho sítě umožňující předávat vnořené sítě pomocí značek (tokens), a ostatních formalismů založených na Petriho sítích [1]. Aplikace Renew (The Reference Net Workshop) je vyvíjena v programovacím jazyce Java a spravován pod záštitou Ústavu informatiky na Fakultě informatiky, matematiky a přírodních věd Univerzity v Hamburku v Německu.

Mezi stěžejní vlastnosti lze zařadit především pokročilé možnosti simulace (mj. vzdálená simulace a použití ladícího režimu pomocí vložených zarážek) a víceúčelové použití aplikace, jež umožňuje použití různých formalismů Petriho sítí. Další, neméně důležitou, vlastností z pohledu uživatele je rozhraní aplikace, které disponuje příjemně jednoduchým vzhledem a ovládáním, což byl i prvek, který jsem se snažil vštípit svému editoru.

Kapitola 3

Návrh aplikace

Pokud si vývojář ujasnil zadání své práce, může přistoupit k dalšímu bodu vývoje aplikace. Tímto bodem je samotný návrh architektury, který se opírá o základní body analýzy požadavků a samotného zadání.

Tato kapitola se dělí do dvou etap, kde první etapa je věnována návrhu architektury serveru 3.1 a druhá zase návrhu architektury klientské aplikace 3.2. Uvedené rozdělení plyne jednak ze zadání, jednak z logického postupu při vypracování mé práce.

3.1 Návrh architektury serveru

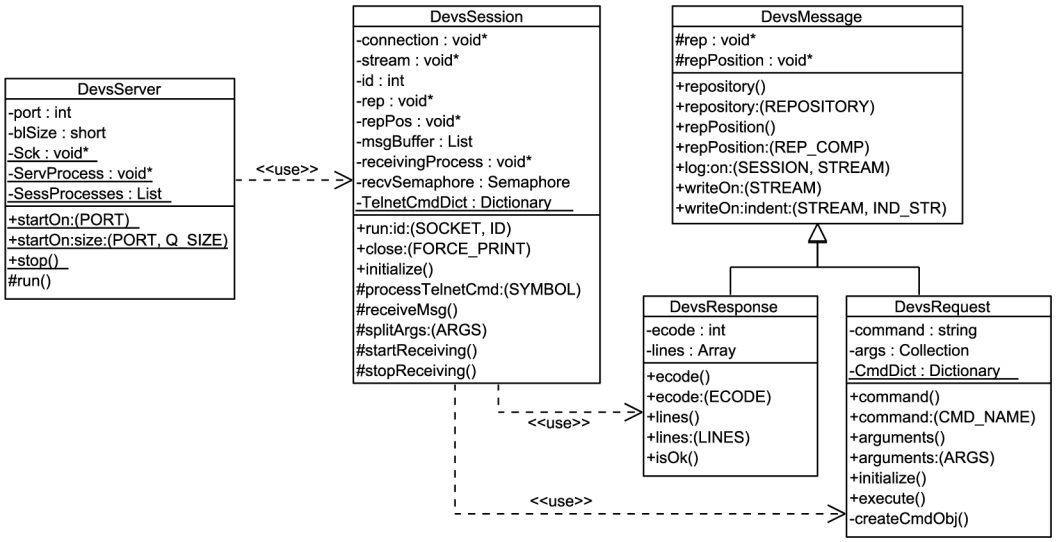
Celá architektura serveru vychází z myšlenky konkurentího serveru (kap. 2.3.1), tedy potřeby *hlavního procesu* s množstvím dalších dětských *procesů pro obsluhu* klientů. Je koncipován jako *telnet* server se zprávami zasílanými v podobě prostého textu. Toto řešení umožňuje automatizované testování skrze příkazový řádek, ale o testování více v kapitole 5.

Při návrhu se ovšem musí vycházet z existující aplikace SmallDEVS, přesněji jeho jádra implementace. Zmíněná relace vzniká v důsledku požadavku zadání na serverovou část mé práce. Od toho se také odvíjí postup při návrhu severu. Nejprve je nutné navrhnout samotné připojení a komunikaci, o čemž pojednává kapitola 3.1.1. A až po ověření validity řešení je možné přistoupit k další části, návrhu zpráv, kterým bude server „rozumět“ a na které bude reagovat. Zprávy jednoduše obalí aktuální jádro aplikace SmallDEVS tak, že budou poskytovat jeho operace ovšem bez použití grafického rozhraní. K návrhu zpráv je vytvořena kapitola 3.1.2.

3.1.1 Server

Jádro návrhu serveru tvoří tři třídy, kde první `DevsServer` spouští server a přijímá žádosti o navázání spojení. Server lze spustit s dvěma parametry – číslo portu (povinný parametr), na kterém server naslouchá, a velikost fronty neobsloužených žádostí o navázání spojení. Druhá třída, `DevsSession` obsluhuje každého klienta zvlášť, respektive ve zvláštním procesu. Poslední třída, s názvem `DevsMessage`, definující obecnou zprávu slouží jako základ pro zprávu zasílanou klientovi `DevsResponse` a zprávu získanou od klienta `DevsRequest`. Třídu zprávy lze tedy označit jako abstraktní nesoucí společný základ pro její dílčí třídy. Princip přijímání a odesílání zpráv je blíže rozepsán v podkapitole 3.1.1.

Server primárně běží opět ve třech hlavních procesech, případně skupinách procesů, kde každý má svoji funkci v rámci komunikace s klientem. První proces je hlavním procesem, který přijímá požadavky na spojení a vytváří procesy tzv. *sezení* (session). Současně



Obrázek 3.1: Hierarchie tříd jádra serveru.

lze ovšem přijmout pouze jeden požadavek na spojení, kdy ostatní musí být odloženy až do vytvoření sezení pro obsluhu právě přijatého požadavku. Jak již bylo zmíněno, druhou hlavní skupinou jsou procesy obsluhy klientů, zvané sezení. V procesech této skupiny dochází ke zpracování požadavků klientů, nebo-li příjmu a reakci na žádosti. Příjem požadavků a jejich prvotní fázi ověření obstarává proces vzniklý rozdvojením procesu sezení klienta, zvaného přijímací proces. Po obdržení požadavku od klienta je vyzvednut z přijímacího procesu a podroben druhé fázi ověření. Následně je požadavek zpracován a výsledek zaslán zpět klientovi. Celý tento postup také znázorňuje sekvenční digram na obrázku 3.2.

Ověření přijímaných zpráv probíhá ve třech fázích:

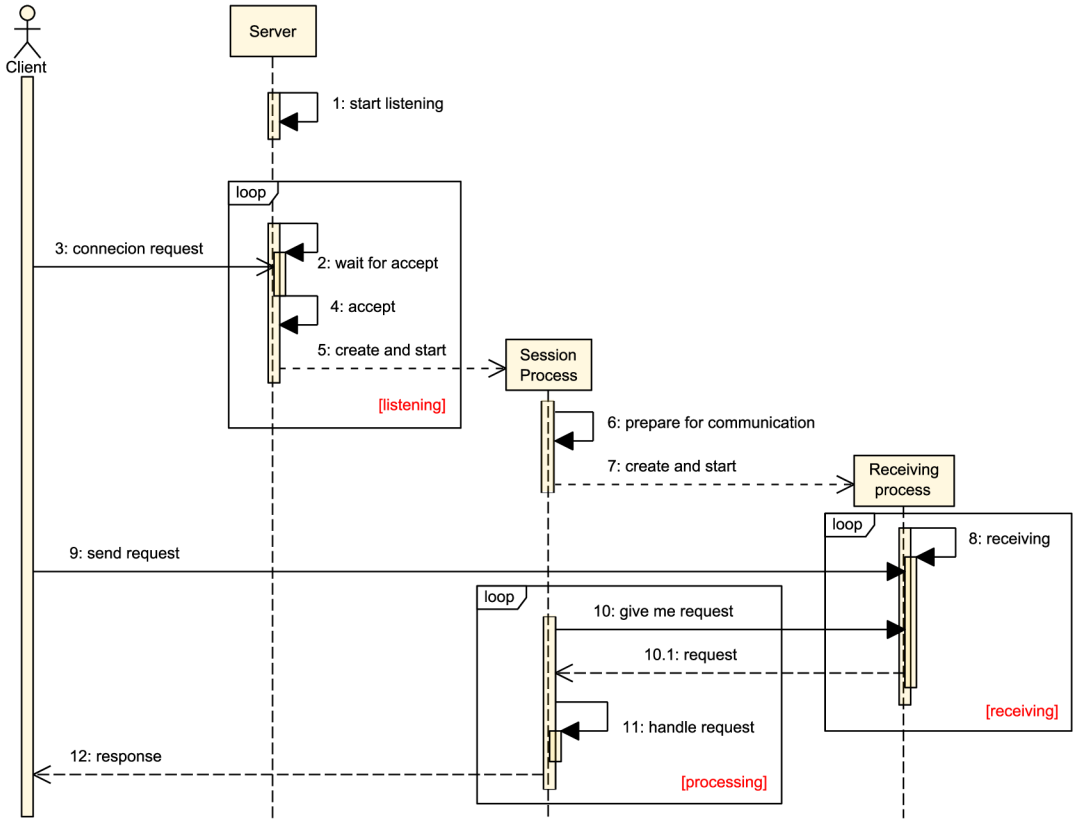
1. fáze se vyznačuje ověřením telnet, respektive protokolu telnet, příkazů.
2. fáze provádí základní syntaktické a sémantické ověření a rozpoznání typu požadavku.
3. fáze se zaměřuje na podrobné rozpoznávání jednotlivých parametrů a jejich hodnot v rámci typu požadavku (zajišťuje třída implementující požadavek, více v kap. 3.1.2).

Přijímání a odesílání zpráv

Aby se předešlo aktivnímu čekání procesu obstarávajícího obsluhu klienta (*session process*) na zprávu, využije se synchronizačního primitiva, *semaforu*. Semafory, jednoduše řečeno, zajišťují synchronní blokující komunikaci mezi procesy. Jejich princip spočívá v použití *čítače* a volání dvou funkcí, `wait()` a `signal()`. První jmenovaná funkce dekrementuje čítač, ale v případě, kdy čítač je nulový, tak se proces zastaví a vyčkává na navýšení čítače. Druhá funkce `signal()` čítač inkrementuje a tím uvolní případné čekající procesy[1].

Přijaté zprávy se tedy vkládají do fronty nevyřízených požadavků a současně se nad semaforem volá funkce `signal()`. V případě, že proces obsluhy klienta potřebuje získat další požadavek pro zpracování, zavolá semaforovou funkci `wait()`. Pokud není blokován, vyzvedne první požadavek z fronty a zpracuje ho. Zprávy ovšem nesmí přeskakovat, neboť by

[1] Více informací o synchronní komunikaci mezi procesy lze nalézt v dokumentu http://greenteapress.com/semaphores/downey08semaphores.pdf.



Obrázek 3.2: Připojování klientů ke konkurentnímu serveru.

se porušila konzistence celé komunikace a jediným řešením vzniklé situace by bylo znovupřipojení ze strany klienta.

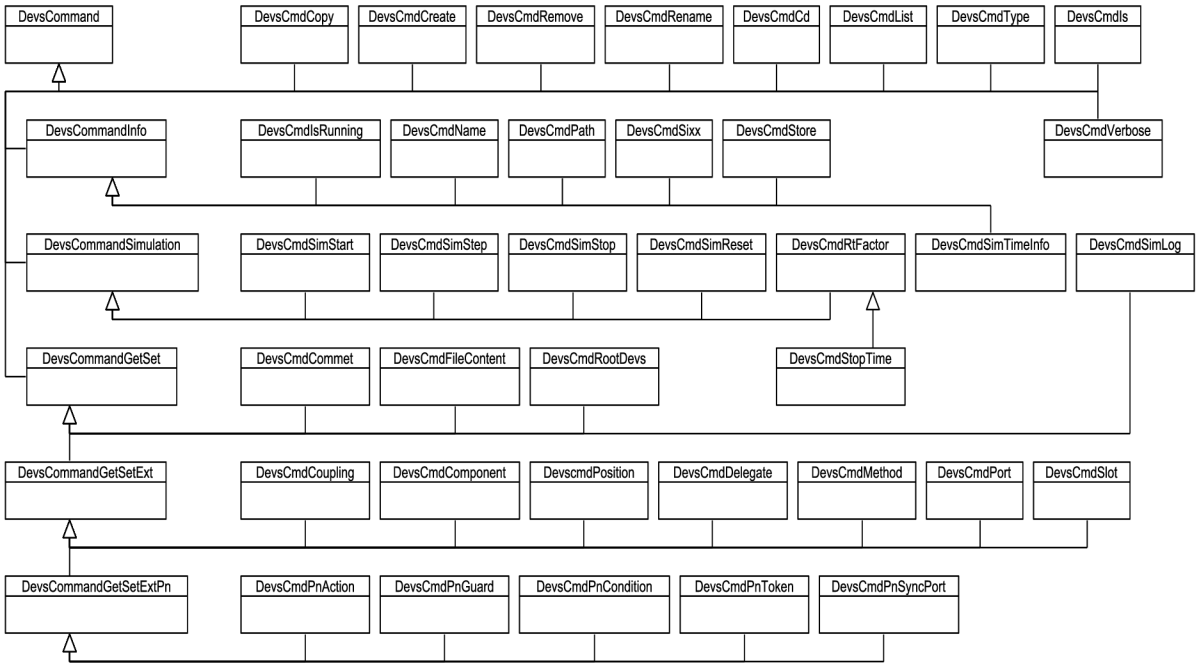
Výsledek zpracování je, stejně jako příchozí požadavek, zaslán jako prostý text skrze telnet protokol. Reakce na každý požadavek zasílá klientovi zpět patřičnou odpověď, i když se jedná o hlášení vzniklé chyby.

3.1.2 Požadavky (zprávy od klienta)

V případě, kdy klient požaduje získání dat ze serveru nebo provedení akce, která způsobí změnu dat, musí použít *požadavků*. Požadavky jsou tedy zprávy informující server, případně nesoucí dodatečná data, o nutnosti provést reakci nad svými daty. Všechny zprávy v návrhu architektury serveru zastávají třídy, respektive objekty. Vzhledem k danému návrhu lze pro zprávy definovat společné chování a to pak mezi nimi šířit a měnit pomocí základních vlastností objektové orientace – dědičnosti a polymorfismu.

Vzniklo několik skupin žádostí s dostatečně podobnou funkčností, což je znázorněno na obrázku 3.3. V následujícím odstavci popíši základní rozhraní dědičnosti zpráv, protože význam ostatních tříd zpráv udává vztah mezi jejich názvem a popisem požadavků v příloze E. Názvy tříd konečných zpráv vždy začínají souslovím `DevsCmd` a následuje odpovídající název požadavku s použitím tzv. „CamelCase“ zápisu názvů[2].

[2]CamelCase definuje způsob zápisu víceslovných frází bez použití mezer, kdy jednotlivá slova začínají velkým písmenem.



Obrázek 3.3: Hierarchie tříd příkazů pro server.

Hlavní abstraktní třídou je v tomto případě `DevsCommand`, která disponuje základním rozhraním pro volání a zahájení požadované operace nebo pro validitu parametrů. Tato třída slouží jako předek dalším abstraktním třídám:

- `DevsCommandInfo` – definuje základ pro dotazovací žádosti, nebo-li žádosti, které umožňují pouze získání dat ze serveru.
- `DevsCommandSimulation` – slouží jako podklad pro ovládání a přizpůsobení simulace.
- `DevsCommandGetSet` – poskytuje základ pro žádosti s jednoduchými dotazovacími a přiřazovacími operacemi (např. získání/nastavení komentáře pro model).

Od poslední třídy, z výše uvedeného výčtu, dědí a rozšiřuje její vlastnosti další základní abstraktní třída `DevsCommandGetSetExt`. Rozšířenými vlastnostmi je myšlena podpora komplikovanějších zápisů dat při dotazovacích i přiřazovacích operací žádostí. Mezi ně se řadí například poziční požadavek nebo požadavek pro operaci nad porty. Z hlediska návrhu poslední podstatnou abstraktní třídou je `DevsCommandGetSetExtPn`, která rozšiřuje vlastnosti třídy `DevsCommandGetSetExt` o dodatečné operace nad prvky vysokoúrovňové Petriho sítě.

Server na přijaté požadavky reaguje zasláním odpovědí, vzápětí po provedení operace plynoucí z obdržené žádosti. Přijmutím odpovědi je klient obeznámen s výsledkem svého požadavku. Více informací o odpovědích popisuje kapitola věnující se komunikačnímu protokolu 3.3.

3.2 Návrh architektury klienta

Druhá, klientská, část mé práce specifikuje koncovou aplikaci, proto je tato kapitola věnována právě jí, přesněji jejímu návrhu. Zaměřuji se především na tři podstatné části a sice editaci

modelů (kap. 3.2.1), komunikaci se serverem (kap. 3.2.2) a uživatelské rozhraní (popsán v kap. 3.2.3).

V návrhu jsem se snažil uplatnit kladné vlastnosti aplikací SmallDEVS a Renew (popsány v kapitole 2.4). Jelikož implementace DEVS formalismu na straně serveru, respektive aplikaci SmallDEVS, se zakládá na prototypových objektech, je důležité pochopit jejich princip.

Prototypový objekt se skládá ze:

- *slotů* – stavové proměnné modelu nesoucí hodnoty dle aktuálního stavu modelu,
- *delegátů* – speciální druh slotů v objektově orientovaném prostředí založeném na prototypových objektech, které slouží k uchování referencí na delegované objekty,
- *metod* – manipulují se stavem modelu (odpovídá definici atomického DEVS modelu z kapitoly 2.1.1).

Komunikace probíhá pomocí reakcí na zprávy, kdy se provede požadovaná metoda nebo se vrátí obsah daného slotu. S výše uvedeným výčtem prvků lze manipulovat (editovat). Z hlediska dědičnosti jsou využity delegace, kdy je neznámá zpráva hledána v delegátech a po nalezení provedena v kontextu původní zprávy.

Sdílené chování prototypových objektů lze implementovat pomocí tzv. „traits“ (v češtině není adekvátního překladu). Trait definuje sdílené chování pomocí objektu, který se vytvoří odděleně a pak jej lze pomocí delegátů v atomických DEVS modelech používat. Případná editace tedy způsobí změnu chování všech atomických DEVS modelů, které jej používají. Definicí se podobá prototypovým atomickým DEVS komponentám s tím, že nepodporuje porty. [3]

3.2.1 Editace a modelování

Editaci, respektive modelování, prvků repozitáře uživateli zpřístupňuje sada *editorů*. V závislosti na různých typech prvků repozitáře je nutné mít různé editory pro obsah těchto prvků. Jelikož lze editovat tři typy prvků, vzniká požadavek na tři druhy editorů:

1. editor pro *grafové struktury* (složené DEVS modely a Petriho sítě),
2. editor pro *atomické DEVS modely*,
3. editor pro *textové prvky* repozitáře.

Editor grafových struktur obecně umožňuje sloučit dva druhy modelů jednak složeného DEVS, jednak Petriho sítí (konkrétně vysokoúrovňových Petriho sítí). Rozlišení, o který typ modelu se jedná, se určuje až dle typu prvku repozitáře. Více informací o návrhu editoru pro grafy je popsáno v kapitole 3.2.3.

Návrh editoru atomických DEVS modelů je inspirován již funkčním řešením v aplikaci SmallDEVS. Editor se dělí horizontálně do tří částí (panelů) z nichž první a poslední jsou prakticky totožné, protože se věnují editaci vstupních, respektive výstupních, portů. Prostřední část se soustředí na jádro prototypového atomického DEVS modelu a sice na editaci slotů, delegátů a metod (viz. nadřazená kapitola 3.2). Vzhledem ke vhodně zvolenému návrhu lze tento editor použít také pro modelování sdílených atomických modelů zvaných traits, ovšem při zneviditelnění postranních panelů pro porty.

Funkce editoru textových prvků repozitáře je zřejmá, edituje prostý text. Proto ji, myslím, není třeba více popisovat.

Přestože jsou uvedené druhy editorů od sebe dosti odlišné, mají společnou vlastnost – *historii úkonů* (možnost vrátit provedené změny zpět). Uvedené vlastnosti lze využít také pro získání provedených změn pro uložení na server. Proto je třeba navrhnout převaděč z formátu používaného *správcem historie* (History Manager, Undo Manager) do programově použitelných struktur.

Převaděč pracuje ve dvou hlavních fázích:

1. fáze provede samotný převod z formátu struktury používaného správcem historie do vlastní struktury.

2. fáze se dělí do více částí proveditelných samostatně při zachování pořadí. Principem této fáze je úprava struktury překonvertované historie tak, aby se usnadnilo a urychlilo její použití – úpravy s daným cílem lze obecně nazvat heuristiky:

   (a) první heuristika se aplikuje na *odstraněné součásti*, pokud byly vytvořeny až po načtení modelu, nebo-li byly nově vytvořeny. V případě splnění výše uvedené závislosti jsou všechny ostatní změny součástí v historii odstraněny. V opačném případě jsou odstraněny všechny změny vyjma změny odstranění součástí.

   (b) druhá heuristika se vztahuje na nově *vytvořené součásti*. Slouží tedy k odstranění následujících (na časové ose) úprav, vztažené k dané součásti, kromě nejnovější úpravy.

   (c) třetí heuristika spojí všechny provedené *editační změny*, vztažené k dané součásti, do jedné.

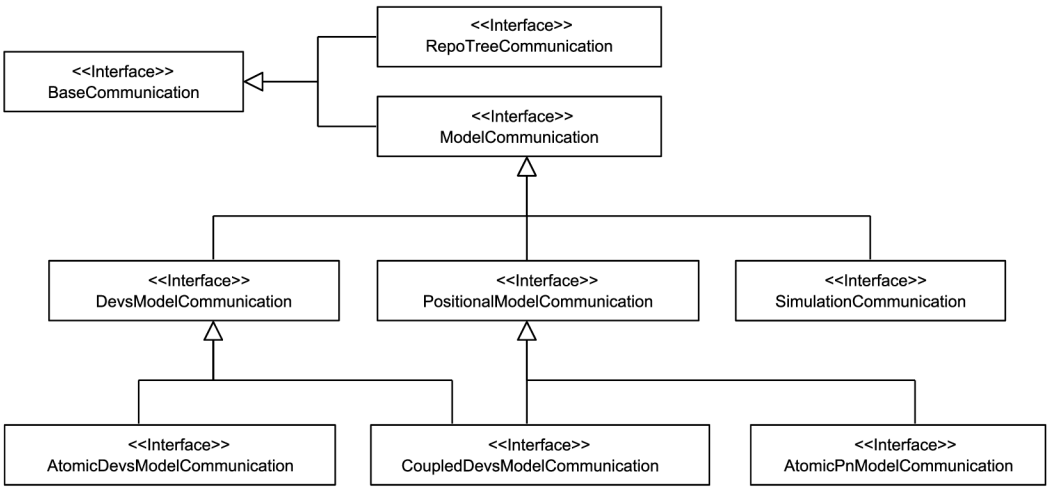
   (d) poslední heuristika je volitelná, protože ji lze uplatnit jen v editorech grafových struktur. Jejím principem je spojení všech provedených *pozičních změn*, vztažené k dané součásti, do jedné.

3.2.2 Komunikační rozhraní

Zajištění komunikace mezi klientem a serverem je jedním z klíčových prvků této práce, proto je třeba ji navrhnout opravdu pečlivě. Jelikož sever používá *komunikační protokol telnet*, tak vzniká totožný nárok i na klienta.

Komunikační rozhraní je odděleno od jádra klienta tak, aby mohlo být v případě potřeby přepracováno bez většího zásahu do dalších částí aplikace, nebo-li s ohledem na pozdější znovupoužitelnost.

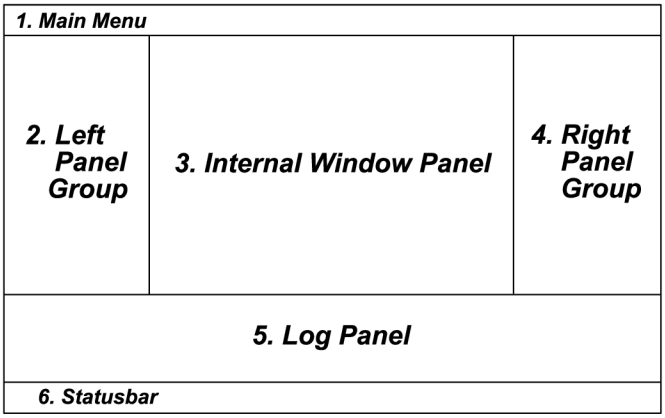
Následující obrázek 3.4 znázorňuje hierarchii tříd, respektive rozhraních, použitou jako základ pro výměnu informací mezi klientem a serverem. Rozhraní `BaseCommunication` slouží jako předek pro rozhraní modelů a repozitáře. Druhý jmenovaný zajišťuje základní editační operace s prvky repozitáře (např. přidávání a odstraňování těchto prvků v rámci hierarchie adresářů). Situace modelů je ukázkovým případem použití dědičnosti v rámci dekompozice objektového návrhu. První úroveň dědičnosti zajišťuje logické oddělení modelů v rámci jejich použití, kdy `DevsModelCommunication` zajišťuje operace s porty DEVS modelů, dále pak `PositionalModelCommunication` operace s hranami (spoji) a rozmístěními komponent a nakonec rozhraní `SimulationCommunication` obstarávající akce se simulací, jako je její ovládání a přizpůsobování. Druhá, koncová, úroveň dědičnosti již specifikuje operace pro jednotlivé druhy modelů. Za povšimnutí stojí násobná dědičnost u rozhraní pro spojované DEVS modely (`CoupledDevsModelCommunication`), což zabezpečuje kombinaci tohoto DEVS modelu s pozicemi.



Obrázek 3.4: Navrhovaná hierarchie tříd pro zajištění nezávislosti klienta na použitém komunikačním rozhraní.

3.2.3 Uživatelské rozhraní

Uživatelské rozhraní musí být navrženo s ohledem na koncového uživatele a to tak, aby jej práce s aplikací neodrazovala od dalšího použití. Musí být tedy snadné, jednoznačné a přehledné z pohledu grafického návrhu a ergonomické z pohledu ovládání. Z důvodu rozsáhlosti textu potřebného pro jeho vysvětlení je popis ovládání aplikace přesunut do přílohy B. Tato kapitola se především věnuje grafickému návrhu rozhraní (GUI) a ovládání aplikace bude zmíněno jen okrajově.



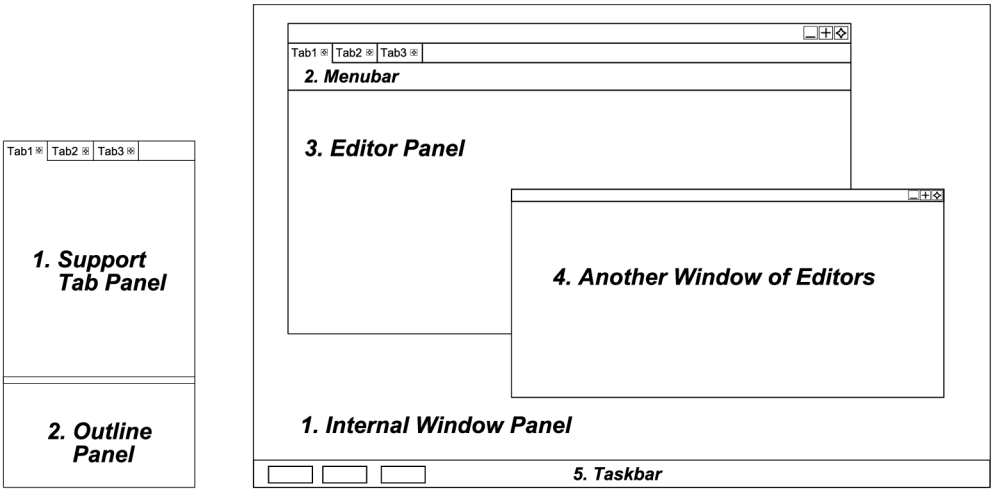
Obrázek 3.5: Základní návrh grafického rozhraní aplikace.

Obrázek 3.5 znázorňuje původní návrh na rozložení panelů, respektive částí, aplikace, který je navržen s ohledem na dostupnost všech důležitých částí se snahou o zachování přehlednosti. První panel, zvaný *hlavní nabídka* (Main Menu), disponuje logicky řazenými nabídkami pro ovládání aplikace např. pro operace se serverem nebo operace ukládání, tvorby modelů. Druhý (Left Panel Group) a čtvrtý (Right Panel Group) panel obsahuje podpůrné panely záložky pro navigaci v repozitáři prvků a pro usnadnění modelování. Druhů panel, jak již bylo zmíněno, se skládá ze záložky pro zobrazení obsahu repozitáře a palety prvků pro modelování složených DEVS modelů nebo Petriho sítí. Čtvrtý panel zobrazuje tabulku

nastavení dle aktivního editoru, nebo prvku repozitáře. Třetí panel (Internal Windows Panel) lze nazvat také jako *plocha pro vnitřní okna editorů*, který se skládá z libovolného počtu jednotlivých editorů. Předposlední, pátý, panel vznikl postupným vývojem návrhu tak, aby uživateli zpřístupnil aktivní odezvu činnost aplikace. Uvedený účel odráží i jeho název a sice *protokolovací panel* (Log Panel). Poslední panel, *stavový řádek* (Statusbar), obecně uživateli zpřístupňuje aktuální stav aplikace. V aplikaci je použit pro zobrazení připojení k serveru. Důležité, některé výše uvedené, panely jsou blíže rozepsány v následujících podkapitolách.

Editace (modelování)

Prvním návrhem editování prvků repozitáře bylo zobrazení jednotlivých editorů v rámci záložek. Ale z důvodu potřeby zobrazit více editorů souběžně je tento návrh rozšířen o vložení panelu záložek editorů do samostatných oken (obrázek 3.6b). Záložky lze tahem myši mezi okny přesunovat.



(a) (b)

Obrázek 3.6: Podrobnější znázornění důležitých částí grafické návrhu aplikace z obrázku 3.5. (a) – pomocný panel, (b) – hlavní pracovní panel.

Panel samostatných oken je rozšířen o *seznam všech oken*, zobrazen ve stejnojmenném panelu (nebo-li Taskbar). Myšlenka seznamu oken pochází z grafického uživatelského rozhraní operačních systémů, kde je využita pro rychlou navigaci mezi okny aplikací. V mém návrhu je jeho funkce obdobná, jen orientovaná na vnitřní okna s editory. Každé okno se tedy skládá z libovolného počtu záložek s editory, kde každý editor navíc obsahuje *panel nástrojů* (Menubar). Nástroji jsou myšleny ovládací prvky převzaté z hlavní nabídky.

Paleta prvků pro modelování

Paleta prvků uživateli poskytuje dostupné prvky pro vložení do modelu. Jejich výčet se liší dle aktivního editoru modelu (např. pro Petriho sítě je dostupné místo, přechod, hrana a testovací hrana). Tahem (při stisku levého tlačítka myši) lze prvky vkládat do modelu, přičemž hrany je nutné nejprve zvolit (označit) a poté tahem myši intuitivně vkládat mezi prvky modelu.

Panel ve spodní části zobrazuje náhled grafové struktury modelu, který umožňuje snadnou a rychlou navigaci. Celý panel palety (založen na návrhu z obrázku 3.6a) je dostupný

pouze pro spojovaný DEVS model a model Petriho sítě, v ostatních případech postrádá význam.

Zobrazení obsahu repozitáře

Mezi jeden z nejdůležitějších součástí uživatelského rozhraní jednoznačně patří přístup k prvkům (modelům) uložených v repozitáři na straně serveru. Prvky se od sebe vzájemně vizuálně odlišují pomocí charakteristických ikon, dle typu prvku, a pomocí názvů, které jsou unikátní v rámci přímých potomků podstromu. Vysvětlivky ikon typů prvků repozitáře obsahuje podkapitola přílohy B.

Zpřístupnění základní editace prvků repozitáře, jako jejich přidávání, přejmenování apod., je z pohledu ovládání poměrně problematická volba. Protože uživatel na první pohled nerozpozná, zda panel s prvky je aktivním panelem, nejvhodnější volbou je ovládání pomocí „vyskakujících menu“ nabídek (popup menus). Tyto nabídky se vždy vztahují k určitému prvku repozitáře a uživateli tedy zpřístupňují jasně dané možnosti. Jsou zobrazeny na popud pravého tlačítka myši. Uvedené řešení je ovšem diskutabilní a to především z pohledu ergonomie, vzniklo ovšem jako kompromis mezi přehledností a zmíněnou ergonomií.

Panel repozitáře, s náhledem 3.6a, navíc obsahuje komentář k aktuálně zvolenému modelu, pokud je ovšem dostupný. Komentář lze samozřejmě editovat, případně odstranit.

Tento způsob zobrazení dat byl navržen jednak s ohledem na intuitivnost prohlížení podobného typu dat v různých souvislostech, jednak pro zachování relace zobrazení s již používanou aplikací SmallDEVS.

Zobrazování vlastností

Podstatnou součástí editace se postupným vývojem návrhu aplikace stal panel (založen na návrhu 3.6a) pro zobrazování a úpravu vlastností aktuálně editovaného modelu. Panel se vyznačuje reaktivním přístupem k vlastnostem modelu, jako je vizualizace jeho součástí – barva pozadí, barva popředí, umístění, název, . . . Dále disponuje zobrazením důležitých dat pro koncového uživatel – např. informace o simulaci, název modelu, apod.

Protokolování událostí

Nápaditým rozšířením uživatelského rozhraní je protokolování událostí, kterými dává aplikace uživateli zpětnou odezvu o aktuálním dění. Protokol, mimo jiné uložitelný do uživatelem zadaného souboru, by měl sloužit jako případný podklad pro řešení chyb aplikace z pohledu vývojáře, respektive administrátora. Proto je vhodné zobrazované zprávy koncipovat věcně, včetně jejich skupiny a přiložit čas vzniku události.

Případné skupiny událostí (řazeny sestupně dle důležitosti):

1. *Fatální chyba* (fatal error) – oznamuje chybu základních operací (např. náhlé odpojení od serveru) znemožňující korektní práci s aplikací.

2. *Chyba* (error) – značí obecnou chybu, po které se sice nelze obnovit, ale lze dále pokračovat v běžné používání aplikace.

3. *Varování* (warn) – oznamuje chybu, ze které se lze obnovit.

4. *Informace* (info) – skupina sloužící pro oznámení informačních zpráv (např. úspěšné uložení modelu).

5. *Sledování* (trace) – značí informace o komunikaci, respektive zasílání/příjem zpráv z/do serveru.

Protokolování událostí, lze teoreticky použít také k výpisu dat o bežící simulaci. Čehož lze využít i v budoucích verzích aplikace v závislosti na podpoře této vlastnosti na serveru. Nynější návrh serveru je, dle zadání pouhým prototypem, proto nebylo rozšíření postupného výpisu informací o běžící simulaci uvažováno.

3.3 Komunikační protokol

Základní myšlenka komunikačního protokolu pochází z možnosti testovat funkčnost serveru za použití prostého příkazového řádku s nainstalovaným telnet klientem. *Telnet* principiálně umožňuje zasílání prostého textu, z čehož se odvíjí také návrh protokolu. Možnosti komunikačního protokolu by měly odrážet možnosti samotného serveru, respektive již vytvořené aplikace SmallDEVS pro editaci a simulaci na bázi DEVS. Současně s uvedenou funkčností musí komunikace probíhat v uživatelsky příjemném rozhraní.

S ohledem na výše zmíněné požadavky byl vytvořen komunikační protokol pro editaci a simulaci na bázi DEVS. Jedná se o prosté textové příkazy, inspirované příkazovým řádkem (terminálem) operačních systémů Linux. Za inspirace notace příkazů v terminálu[3] je možné definovat obecný příkaz jako:

`request_type{ --option[=value]}[CRLF][CRLF].`

Příkazy mohou, v některých případech musí, obsahovat parametry rozšiřující jejich vyjadřovací schopnosti. Parametry jednotlivých příkazů jsou navrženy s ohledem na zachování společné syntaxe a tím i snadnější zapamatovatelnost pro koncového uživatele. Pro ukončení, respektive zaslání, příkazu serveru je nutné přímo za poslední znak zadat dvojici oddělovačů nových řádku pro operační systémy Windows – složen ze znaků `CR` a `LF`. Tyto konce řádku jsou primárním nastavením konzolových telnet klientů, dokonce i na operačních systémech Linux. Limitujícím faktorem uvedeného řešení ukončení příkazů je zápis víceřádkové hodnoty parametru. Řešení, uplatněné i v rámci klientské části je poměrně snadné a sice náhrada výskytů konců řádku z OS Windows na konce řádků známé z OS Linux (tj. použití pouze znaku `LF`). Pro budoucí vývoj by ovšem bylo lepší navrhnout efektivnější způsob syntaxe příkazů, například použitím datového zápisu definovaného jazykem JSON[4].

Odpovědi jsou zasílány také jako prostý text ve tvaru[3]:

`error_code[CRLF][response][CRLF],`

kde na prvním řádku se nachází chybový kód. Další řádky jsou již vyhrazeny dané odpovědi, pokud je definována. Dále je vložen prázdný řádek pro ukončení odpovědi. Tyto řádky jsou od sebe opět odděleny dvojicí znaků `CR` a `LF`.

Příkazy jsou logicky rozděleny do několika skupin:

1. skupina se vyznačuje *základními operacemi* s repozitářem a jeho obsahem (součástmi), jako je například výpis obsahu repozitáře, přidání/odebrání součástí apod.

[3] Použitá notace – `{text}` pro značení možného několikanásobného výskytu, `[text]` značící možný výskyt a `[CRLF]` pro nový řádek z OS Windows.

[4] JSON, nebo-li JavaScript Object Notation, je použit především pro přenos dat a navržen s ohledem na uživatelsky rozpoznatelný a současně strojově zpracovatelný jazyk. Pro více informací doporučuji navštívit webovou stránku http://www.json.org.

2. skupina pro *editační operace s pozičními modely* (složené DEVS a Petriho sítě).
3. skupina pro *editační operace s DEVS modely.*
4. skupina pro *editační operace s modely Petriho sítí.*
5. skupina obsahuje *simulační operace* nad modely včetně operací pro nastavení simulací.

Všechny možné příkazy, myšleno příkazy, kterým server rozumí, jsou z důvodu obsáhlosti přehledně vypsány i s popisem v příloze E včetně výpisu chybových kódů.

Kapitola 4

Implementace aplikace

Předposlední část vývoje aplikací tvoří část praktická, respektive implementační. Proto se v této kapitole věnuji samotné realizaci návrhu na serverové části (kap. 4.2) a navazující části klientské (kap. 4.3). Nejprve je ale vhodné si představit využité nástroje a technologie potřebné pro uskutečnění všech návrhů z kapitoly 3.

4.1 Použité technologie

Pro implementaci aplikace je nezbytným prvkem využití vhodných nástrojů a technologií, které by měly usnadnit práci samotnému vývojáři.

Vzhledem ke skutečnosti, že zdrojová aplikace SmallDEVS je vyvíjena v prostředí programovacího jazyku *Smalltalk* (přesněji ve *Squeaku*) bylo třeba serverovou část postavit ve stejném prostředí. Za to v druhé navazující části byl výběr vhodné technologie volnější, proto jsem použil programovací jazyk *Java* a to z důvodu snadné rozšiřitelnosti o další knihovny a také především díky zkušenostem nabytých z ostatních kurzů.

4.1.1 Smalltalk (Squeak)

Smalltalk je pravým, čistě objektově orientovaným a současně dynamicky typovaným programovacím jazykem vyvíjeným od počátku 80. let minulého století. Považuje se za vzor pro *experimentální programování*[1], což je způsob vývoje založeného na rychlé tvorbě nejasného nebo měnícího se zadání. Tento druh programování umožňuje inkrementálně upravovat, okamžitě testovat, ladit a zkoumat realizaci daného problému.

Experimentální programování je, jak již bylo naznačeno, založeno na *inkrementální kompilaci*, kdy aktuálně upravený a uložený zdrojový kód je ihned zkompilován a připraven k použití. Vývojář tedy nemusí čekat sekundy, v některých případech i minuty, na překlad zdrojových kódů. Obecně zde ovšem vzniká problém s dynamickým typováním proměnných, kdy potenciální typové nesoulady jsou zjištěny až za běhu programu.

Smalltalk se dělí do několika odnoží, z nichž používám Squeak, který sjednocuje programovací jazyk a virtuální stroj, nad kterým běží. Umožňuje tedy jednotný a přehledný vývoj aplikací se všemi vlastnostmi Smalltalku.

[1] Přesný překlad anglického výrazu Exploratory programming je obtížné přesně určit, ale často se používá právě experimentální programování.

4.1.2 Java

Historie jazyka spadá do počátku 90. let minulého století, kdy se projekt původně nazýval Oak a byl zaměřen na výrobky spotřební elektroniky. Postupem času se projekt zaměřil více na internet, byl přejmenován na současný název Java („horká káva“) a začal být součástí internetových prohlížečů [5].

Java (v aktuální verzi 7) je objektově orientovaný multiplatformní programovací jazyk běžící nad virtuálním strojem zvaným *JVM* (Java Virtual Machine), který provádí instrukce z přeloženého zdrojového kódu do *mezikódu* (bytecode). Tímto je teoreticky zajištěno, že jednou napsaný kód lze spustit na jakémkoli koncovém zařízení s JVM. V praxi však tuto přenositelnost doslovně aplikovat nelze z důvodu různých technických příčin. Proto se Java v dnešní době rozděluje do několika platforem, mj. J2EE (Java Enterprise Edition) pro webové služby, J2ME (Java Micro Edition) pro mobilní zařízení a konečně J2SE (Java Standart Edition) pro spouštění aplikací na počítači. Poslední zmíněnou platformu používám jako výchozí při tvorbě klientské části mé práce.

Do základní filozofie tohoto programovacího jazyku náleží především robustnost, stabilita, bezpečnost a velký rozsah knihoven funkcí. V práci využívám následující externí knihovny:

- *LF2Prod Common Components* – rozšíření knihovny Swing o tabulku nastavení s úpravami od Bartosze Firyna,
- *Apache Commons Net* – implementace telnet klienta pomocí schránek (sockets),
- *Apache Log4j 2* – podpora pro protokolování událostí průběhu algoritmů a komunikace,
- *TinyLaF* – přizpůsobitelné téma vzhledu (Look And Feel) aplikace,
- *MigLayout* – další rozšíření knihovny Swing o snadnější rozmístění prvků v dialozích,
- *RED* – podpora editování prostého textu,
- *XStream* – pokročilá serializace datových struktur do formátu XML,
- *JGraphX* – tvorba a editace grafových struktur (k této knihovně je vytvořena vlastní podkapitola 4.1.2),
- *Swing* – tato knihovna sice není externí, ale je vhodné si ji představit (podkapitola 4.1.2).

Knihovna Swing

Knihovnu Swing lze popsat jako na platformě nezávislou integrovanou (od verze Javy 1.2) sadu uživatelských prvků pro ovládání aplikace skrze grafické rozhraní. Je postaven nad starší knihovnou AWT (Abstract Window Toolkit), ale na rozdíl od ní disponuje tzv. *ligh-weight* komponentami, což znamená, že vykreslení probíhá přímo v Javě a operační systém obdrží již hotový grafický prvek pro zobrazení [5]. Tato vlastnost poskytuje onu platformní nezávislost a mnohem lepší využití hardwarových zdrojů.

Knihovna JGraphX (JGraph)

Tato externí knihovna slouží k tvorbě a editaci grafů za použití výše zmíněné knihovny Swing pro vizualizaci grafových prvků. JGraph balík se rozděluje do dvou větví, pro Javascript - mxGraph a pro Javu - JGraphX. Je postavený na designovém vzoru *MVC* (Model-view-controller), který rozděluje datový model, uživatelské rozhraní a řídící logiku do tří nezávislých celků. V případě knihovny JGraphX je význam následující:

- *Model* popisuje základní grafové, výběrové, editační rozhraní.
- *Zobrazení* popisuje vnitřní zobrazování grafových prvků (včetně geometrie) a jejich vzájemné mapování na modelovou část.
- *Ovládání* popisuje vykreslující a editační proces těchto prvků.

Hlavní výhodu představuje široká škála možností přizpůsobení vzhledu a ovládání, dále pak stále aktivní vývoj a podpora. Za zmínku také stojí zobrazení náhledu grafu, což usnadňuje orientaci v rozsáhlých strukturách.

4.2 Realizace serverové části

Realizace serveru využívá *TCP schránek* (sockets) jednak pro příjem klientů, jednak pro obsluhu klientů v *sezeních* (sessions). Ovládání (spuštění a vypnutí) serveru je popsáno v příloze B. Implementaci požadavků na provedení různých akcí nad jádrem aplikace SmallDEVS se věnuje kapitola 4.2.2.

Postup navazování spojení popisuje kapitola 4.2.1. Při ukončování běhu serveru (metoda `stop`), respektive jeho procesu, se využívá sady statických (třídních) proměnných pro zajištění korektního uzavření všech prostředků. Veškeré procesy sezení (procesy obsluhy klientů) se vkládají do seznamu. Dále je zde uložena schránka pro naslouchání a samotný proces pro příjem klientů.

Jelikož aplikační rozhraní aplikace SmallDEVS dosud nedisponuje implementací modelování a simulace, takže ji bylo třeba dokončit, o čemž pojednává kapitola 4.2.3. Dále pak bylo potřeba rozšířit základní sadu komponent repozitáře o komponentu dokumentu `LogDocument`.

Třída `LogDocument` se zakládá na `Document`, avšak podporuje zápis dat za použití znakového proudu (string stream). Při požadavku na spuštění simulace se její výpis ukládá do instance třídy `LogDocument`, která nese název jako daná simulace s připojenou koncovkou `.log` v adresáři `/LOG`. Takto vytvořená komponenta umožnila kontinuálně ukládat výpis informací simulace běhu.

4.2.1 Navazování spojení

Server čeká na žádosti o připojení v nekonečné smyčce ve vlastním procesu. Pokud se žádost dostaví, je vytvořen nový proces sezení s novým identifikátorem (nová instance pro obsluhu klientů `DevsSession`). Identifikátor slouží pro pojmenování sezení při výpisu protokolujících událostí, jako jsou připojení nebo odpojení klienta. Sezení musí být dle návrhu 3.1.1 spuštěno v samostatném procesu, k čemuž používá obligátního volání metody `fork`.

Implementace třídy sezení klientů `DevsSession`, jak ji definuje návrh, je jedním ze stěžejních bodů. Pro spuštění příjmu a reakce na požadavky klienta v nekonečné smyčce musí

být vyhodnocena metoda `run:id:`. Vstupně výstupní komunikace mezi oběma body využívá třídu `SocketStream` z jádra Squeaku, která usnadňuje čtení a zápis prostého textu veschránce (socket). Rozpoznávaní požadavků i s jejich parametry a jejich verifikace popisuje kapitola 4.2.2.

Pro příjem požadavků jako takových je nutné spustit samostatný proces nad metodou `startReceiving`, neboť také běží v nekonečné smyčce. Přijaté žádosti, ve formě prostého textu, vkládá do fronty a navyšuje semafor, implementující synchronní komunikaci mezi procesem příjmu požadavků a jejich zpracováním. V této třídě je také implementována podpora pro příkazy protokolu telnet, ale ke zpracování je zpřístupněn pouze příkaz `AYC` (Are You There) [6], protože ostatní příkazy jsem v klientovi nepotřeboval. Příkaz `AYC` slouží k indikaci, že server stále běží. Pokud ano, je zaslána odpověď `#TELNET AYT OK[CRLF]`, ale obecně se může jednat o jakoukoliv odpověď. Jestliže server neběží dojde k vypršení časovače na straně klienta, což indikuje odpojení serveru. Příkazy protokolu telnet se ovšem do čekající fronty požadavků nevkládají, nýbrž se okamžitě zpracují.

Smyčka obsluhy klienta je ukončena, jakmile se zruší schránka nebo je obdržena závažná výjimka patřící schránce, která není výjimkou `ConnectionTimedOut` (indikuje pouze potenciální odpojení klienta, když po určitou dobu nezaslal požadavek).

4.2.2 Rozpoznávání požadavků a jejich realizace

Po přijetí požadavku nastává fáze verifikace (rozpoznání a ověření jeho validity), která se dělí do dalších tří fází navrhnutých v kapitole 3.1.1:

1. fáze *ověří příkaz telnet protokolu* jednoduše pomocí průchodu skrze slovník (datová struktura na způsob hašovací tabulky).

2. fáze provede *základní syntaktické ověření* struktury požadavku (dle kap. 3.3). Ověření obstarává metoda `splitArgs:` z třídy `DevsSession`, která rozdělí parametry a případně i jejich hodnoty do lineárního seznamu.

3. fázi již zajišťují *samotné implementace žádostí*. Názvy parametrů se vyhledávají v podmíněném příkazu `case` a poté se ověří jejich kombinace. Případné hodnoty parametrů jsou syntakticky kontrolovány za použití regulárních výrazů. Jejich sémantické chyby se ovšem projeví až při reakci na požadavek.

Všechny informace o obdrženém požadavku, mezi které se řadí název a parametry včetně hodnot, jsou vloženy do instance třídy specializované na zapouzdření žádostí `DevsRequest`. Při vyhodnocení metody `execute` této třídy se název žádosti vyhledá ve slovníku známých žádostí a dle výsledku je vytvořen objekt zastupující chtěný požadavek. Pro spuštění vykonávání žádosti (zastupované objektem) je třeba vyhodnotit její metodu `execute` (blíže v následující podkapitole).

Realizace požadavků

Implementace jednotlivých požadavků se zakládá na návrhu z kapitoly 3.1.2 a návrhu komunikačního protokolu z kapitoly 3.3. Při vyhodnocení metody `execute` z třídy `DevsCommand` dochází k výše naznačené verifikaci parametrů a jejich vyhodnocení dle požadavku pomocí metody `opCode:`. Díky vlastnostem objektově orientovaného programování, dědičnosti a polymorfismu, je možno metodu `opCode:` přizpůsobit každé implementaci požadavku.

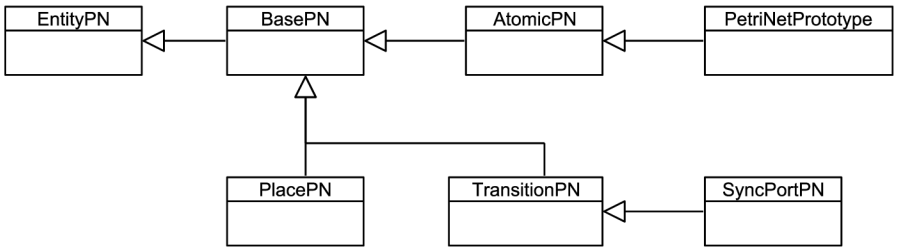
Ve většině žádostí je nutné zadat cestu k požadované komponentě (prvku repozitáře). Implementace umožňuje použít jak absolutní cestu, tak i relativní inspirovanou z OS Linux, kde znak `.` značí aktuální adresář a `..` značí předchozí adresář. Jednotlivé úseky cesty jsou od sebe odděleny lomítkem `/`, kde kořen (root) hierarchie adresářů je znám právě jako lomítko. Po nalezení daného objektu komponenty, dle cesty, se provádí kontrola jeho typu, nebo-li příslušnosti ke třídě. Využívá se dvou přístupů:

- Využití vyhodnocení metody `isKindOf:` základního objektu (třída `Object`), která určí, zda objekt je instancí požadované třídy, přičemž uvažuje i třídy v hierarchii dědění.
- Využití vyhodnocení statické metody `canUnderstand:` třídy (`class`), která určí, zda objekt definuje („rozumí“) požadovanou zprávu.

Po ověření nastává fáze provedení žádosti, ve které se využívá implementačního jádra aplikace SmallDEVS (kap. 2.4.1).

4.2.3 Implementační jádro vysokoúrovňové Petriho sítě

V návrhu se překládalo funkční rozhraní aplikace SmallDEVS[2] pro modelování vysokoúrovňových Petriho sítí, ovšem nebylo implementováno. Proto ji bylo nutné dokončit, alespoň jako prototyp s podporou modelování bez možnosti simulace a navázání na složené DEVS modely.



Obrázek 4.1: Hierarchie tříd pro podporu modelování Petriho sítí.

Implementace je postavena nad navrženým rozhraním ve SmallDEVS za inspirace současného řešení tvorby modelů na bázi DEVS. Návrh hierarchie dědičnosti mezi třídami pro podporu modelování pomocí Petriho sítí znázorňuje obrázek 4.1. Základem je abstraktní třída `EntityPN` podporující pouze definování názvu komponenty. Na ni navazuje třída `BasePN`, která definuje podklad pro všechny ostatní třídy (mj. umožňuje duplikovat objekt a specifikovat nadřazenou komponentu).

První prvek tvořící Petriho síť je místo, které zastupuje třída `PlacePN`. Přístup ke značkám (tokens) umožňuje sada metod `content`, pro získání seznamu značek, a `content:`, pro nastavení značek místa. Druhý prvek představuje přechod (třída `TransitionPN`), jež specifikuje operace nad stáží a akcí. Implementace také zpřístupňuje tři množiny podmínek, rozšiřující stráž i akci, které jsou definovány jako:

- *Vstupní podmínka* (precondition) – podmínka (hodnota) na hraně *vstupující* do přechodu.

[2]Rozhraní aplikace SmallDEVS je popsáno na webové stránce http://perchta.fit.vutbr.cz:8000/projekty/25.

- *Výstupní podmínka* (postcondition) – podmínka (hodnota) na hraně *vystupující* z přechodu.
- *Testovací podmínka* (condition) – podmínka (hodnota) na testovací hraně.

Podmínky stejného druhu jsou uloženy ve slovníku (datová struktura na způsob hašovací tabulky), kde klíč značí název místa a hodnota je podmínka (hodnota) na hraně. Podmínka na hraně může být i prázdná. *Synchronní port*[3] se používá až u objektově orientovaných Petriho sítí, ale z hlediska validní implementace navrženého rozhraní jsou uvedeny i zde.

Abstraktní třída `AtomicPN` implementuje základ pro samotné modelování vysokoúrovňových Petriho sítí, jelikož obsahuje seznam použitých prvků a jejich propojení skrze hrany. Podpora pro hrany ovšem není implementována stejně jako je tomu u složených DEVS modelů. Využívá se množin podmínek přechodů a sice tak, že se hrany z nich rekonstruují. Tato varianta řešení mi v dané implementaci přechodů přišla jednoznačnější, protože tak není nutné mít duplicitně uloženy obdobná data.

Třída použitelná jako konečná reprezentace vysokoúrovňové Petriho sítě nese název `PetriNetPrototype`. Dědí operace z `AtomicPN` a přidává operace pro pozice prvků a pro komentáře. Pozice jsou naopak uloženy obdobně jako u stávající implementace složených DEVS modelů v aplikaci SmallDEVS. Ve slovníku pozic obecně obsahuje dva typy záznamů, pro hranu a pro prvek. Pozici prvku specifikuje název prvku, jako klíč, a bod (tvar `x@y`), jako hodnota. Záznam pozice hrany Petriho sítě se skládá z odlišného hašovacího záznamu, kde klíč tvoří název počátečního prvku a hodnota je název koncového prvku. Hodnotu záznamu pozice hrany představuje pole zlomových bodů (waypoints) hrany.

4.3 Realizace klientské části

Implementace klientské části (editoru) se zakládá na provedeném návrhu z kapitoly 3. Realizace je rozdělena do tří částí, kdy nejprve bylo implementováno spojení a komunikaci serveru, o čemž pojednává podkapitola 4.3.1. Následně se přistoupilo k řešení grafické podoby a nakonec k implementaci jádra aplikace a navázání jej na provedená řešení. Na poslední jmenovanou část se zaměřují další podkapitoly, především pak podkapitola 4.3.4 o samotné implementaci editování modelů.

Uživatelské rozhraní je implementováno tak, jak bylo navrženo v kapitole 3.2.3 s jedním podstatným rozšířením. Jelikož uživatel potřebuje být informován o změnách stavu serveru a jeho dat, je třeba provést obnovu, při které se aktualizuje hierarchie repozitáře a informace zobrazené v panelu s nastavením. Tato obnova může obecně nastat ze dvou příčin:

1. *vynuceně* – ručně, nebo automaticky po určitém časovém intervalu,
2. *operačně* – po libovolné editační operaci s prvky ve stromové struktuře, nebo při manipulaci s modely (především jejich ukládání).

S popisem uživatelského rozhraní se prolínají i další kapitoly, například editace modelů (kap. 4.3.4). Výslednou podobu grafického uživatelského rozhraní lze nalézt v příloze C této technické zprávy. Ovládání je taktéž k nalezení v příloze B.

Uchování, a případné načítaní z (ukládání do) XML souboru `./settings/properties.xml`, konfigurace klienta obstarává třída s názvem `ClientProperties`, která také implementuje

[3] Synchronní porty objektu Petriho sítě jsou speciální metody (přechody) pro atomickou synchronní komunikaci. Synchronizace probíhá zasíláním zpráv ze stráží přechodů. [3]

možnost dalších tříd registrovat si *naslouchání změn* (event listeners) jednotlivých proměnných konfigurace. V případě, kdy konfigurační soubor chybí, je uživatel obeznámen s nastalou situací prostřednictvím komponenty protokolování a použije se výchozí nastavení. Proměnné pro konfiguraci upravují grafické zobrazení grafů a jejich prvků, definují časy důležité pro komunikaci se serverem (mj. interval pravidelné obnovy stavu klienta dle serveru) a specifikují samotný seznam, uživatelem uložených, serverů.

4.3.1 Navazování spojení a komunikace

Jelikož server běží nad protokolem *telnet*, je třeba definovat také telnet klienta. Pro tyto účely byla zvolena všeobecně používaná externí knihovna `Apache Commons Net`, která ve svém základu telnet klienta implementuje. Spravované spojení se serverem by mělo být vždy koncipováno jako jediná instance (singleton) a je tomu tak i v případě mé realizace v podobě třídy `TelnetClient`.

Třída `TelnetClient` rozšiřuje implementaci v externí knihovně o zasílání dotazovacích zpráv „Are You There“ na server pro ověření jeho stálé dostupnosti. Dotazy jsou zasílány v samostatném vlákně po intervalech daných nastavením klienta s názvem `PROP_WAIT_AYT` a v případě, kdy je server, nedostupný dojde k oznámení této skutečnost uživateli a následně ke korektnímu odpojení. Také je zde implementována podpora pro registrování naslouchacích tříd (event listeners) pro události připojení a odpojení.

Zapouzdření požadavku a odpovědi do stejného objektu třídy `Message` umožňuje kontrolu příjmu odpovědi od serveru na danou žádost. Uvedené řešení umožňuje čekat na doručení po určitou dobu (timeout definovaný nastavením s názvem `PROP_TIMEOUT_RECEIVE_MSG`) a po jejím vypršení informovat klienta o chybě a případné pozdější odpovědi sice korektně přijmout, ale ignorovat. Samotné odesílání a příjem zpráv z implementovaného rozhraní z návrhu 3.2.2 v balíku `client.communication.implementation` obstarává třída s názvem `MessageHandler` (respektive její instance) pomocí dvojice vláken. *První skupina vláken* se vytváří pokaždé, kdy klient žádá *odeslání požadavku* na server. Požadavek je označen jako nevyřízený a vloží se na konec fronty čekajících (proměnná `waitingMsgs`), tím se spustí i zmíněný časovač. *Druhé vlákno*, pro *příjem*, běží neustále po dobu připojení k serveru a po přijetí korektní zprávy ji vyzvedne z čekajících požadavků na příjem (první záznam ve frontě) a tím povolí jeho další zpracování. Vzhledem k řešení zasílání a příjmu zpráv prostřednictvím separátních vláken, není jádro klienta zbytečně omezováno.

Pro ověření přijaté odpovědi a případný převod do interní reprezentace klienta se využívá vestavěné podpory Javy pro vyhodnocování řetězců pomocí regulárních výrazů. Takto ověřená a překonvertovaná odpověď je klientovi poskytnuta pro její další zpracování.

4.3.2 Realizace protokolování událostí

Protokolování událostí tak, jak jej definuje návrh z kapitoly 3.2.3, zajišťuje upravená externí knihovna `Log4j 2`. Jako stěžejní třídu lze označit jedináčka (singleton) s názvem `LoggerManager`, který zajišťuje správu všech typů definovaných protokolů a vytváří profil, vzhled, výstupu vložených informací (logu). Stejný vzhled však musí být zapsán i do konfiguračního souboru k protokolování `log4j2.xml` a to z důvodu zachování konzistence danou externí knihovnou.

Libovolná třída, respektive její instance, má díky třídě `LoggerManager`, respektive její instanci, přístup ke zde definované sadě protokolů. Na uvedené protokoly lze zasílat zprávy pomocí rozhraní definovaného v `Log4j 2`, například pro zaslání zprávy o chybě do protokolu je nutné použít metodu `error()` apod.

Výstup, zasílaný protokolujícími objekty, se ukládá do souboru ve složce `./tmp/@log`. Název souboru je dán názvem záložky (třída `LoggerTab`) v protokolujícím panelu, ve které se zobrazuje pouze posledních `N` znaků, kde `N` je hodnota v nastavení klienta `LOG_MAX_CHAR_COUNT`. Obecně lze mít dva typy protokolovaní – globální a lokální. Lokální je určen pro rozšíření v podobě kontinuálního zobrazení záznamu o simulaci. Pro tyto potřeby se musí výpis opatřit značkou (marker) definovanou v `LoggerManager.SIMULATION_MARKER`. Globální protokol zajišťuje, že výpis jeho dílčích protokolů (např. `CONNECTION_LOGGER` a další definované ve třídě `LoggerManager`) se vypisuje na jednom místě.

4.3.3 Přístup k prvkům repozitáře

Hierarchie repozitáře, umístěného na serveru, je v klientské části představována a implementována komponentou `RepositoryTree` s pomocnými třídami `RepositoryItem` a jejich zapouzdřením v podobě třídy `RepositoryItems`.

Třída `RepositoryItem` obsahuje množinu vlastních událostí, které vznikají za účelem oznámení změny (například přejmenování prvku) pro registrované posluchače (event listeners). Posluchači tak obdrží informaci, na niž mohou, dle své implementace reagovat. Prky repozitáře mohou mít více typů současně, zářným příkladem jsou simulace, které překrývají určitý typ modelu. Proto třída `RepositoryItem` obsahuje i seznam typů (prvky výčtu `RepoItemTypes`), kdy primární je vždy prvním v seznamu, v předchozím příkladu by to byl typ simulace. Jejich reprezentace je dle návrhu 3.2.3 rozšířena o jednoznačné ikony a vyskakující menu (v případě uložení ve stromové struktuře).

Jakmile je přijat výčet všech cest, včetně typů, prvků repozitáře, tak se pro vytvoření interní reprezentace stromové struktury nejprve překonvertují jednotlivé prvky repozitáře (stromu) do objektů typu `RepositoryItem` a následně se v instanci třídy `RepositoryItems` (koncipován jako jedináček – singleton) převede do požadované stromové hierarchie. Instance třídy `RepositoryItems` také zajišťuje, aby se nevytvářely duplicitní prvky repozitáře, což udržuje určitou konzistenci v jejich použití.

V případě obnovy stromové hierarchie prvků se zachovává, pokud je to možné, aktuální stav modelů stromu (rozbalení adresářů a označení položek). K tomu jsou využity statické metody pro uložení a načtení stavu modelu stromu umístěné ve třídě `TreeUtils`.

4.3.4 Editace modelů

Pro ukládání upraveného obsahu jednotlivých editorů, dle návrhu 3.2.1, je potřeba implementovat jejich *manažery historie* a *konvertor* této historie do dále programově zpracovatelných struktur. Každý manažer historie musí implementovat rozhraní (interface) třídy zvané `UndoManager`, která definuje navigační metody (`undo()` a `redo()`) a metody pro získání historie provedených úkonů. Uvedeným řešením je vytvořen jednotný přístup editorů a konvertoru (kapitola 4.3.5) k historii.

Jednotlivé editory modelů, sítí (včetně jejich manažerů historie) a prostého textu jsou blíže popsány v následujících podkapitolách.

Nastavení prvků repozitáře a grafu

Je realizován za použití externí knihovny `LF2Prod Common Components` (kap. 4.1.2), přesněji komponenty `PropertySheetPanel`, kterou lze vidět na pravé straně obrázku C.1. Tato

komponenta tvoří a zobrazuje jednotlivé položky nastavení využitím tříd programovacího jazyku Java zvaných *Java Beans*[4] a vlastních anotací nad třídami těchto objektů.

Každá třída, která může podporovat uživatelské nastavení, musí mít vytvořenu odpovídající JavaBean třídu. Každý atribut JavaBean třídy, pokud jej chceme uvést v tabulce nastavení, zase musí obsahovat patřičnou anotaci definovanou komponentou `PropertySheetPanel`. Na následující ukázce zdrojového kódu je patrná struktura tvorby atributů pro nastavení.

```
@PropertyInfo(name = "Verbose", description = "Deep verbose mode", category = MODEL_INFO)
protected boolean verbose = false;
```

Používané Java Beans třídy jsou umístěny v balíku s cestou `client.editor.components.properties` a jejich názvy vždy končí slovem `Bean`. Projevení a uložení provedených změn nad položkou nastavení je okamžitě uloženo skrze registrované posluchače, potomci abstraktní třídy `BeanChangeListener`, změn vlastností objektu (Property change event listeners). Výčet podporovaných Java Bean tříd a jejich význam uvádí tabulka 4.1.

Název třídy	**Popis**
`RepoItemBean`	Základní nastavení vlastností prvku repozitáře (rozšířen o podporu nastavení simulace).
`GraphBean`	Obecné nastavení vlastností samotného grafu.
`GraphItemBean`	Základní grafické nastavení položky grafu.
`GraphEdgeBean`	Nastavení hrany grafu.
`GraphVertexBean`	Geometrické nastavení uzlu grafu.
`DevsBean`	Nastavení složeného DEVS modelu (uzel grafu).
`PortBean`	Obecné nastavení portu DEVS modelu (uzel grafu).
`PortInBean`	Rozšířené nastavení portu DEVS modelu (vložení hodnoty).
`PlaceBean`	Nastavení místa Petriho sítě včetně značek (uzel grafu).
`TransitionBean`	Nastavení přechodu Petriho sítě včetně stráže a akce (uzel grafu).

Tabulka 4.1: Výčet Java Bean tříd pro nastavení komponent.

Editor grafových struktur

Třída editoru pro modelování grafů `EditorGraph` se zakládá na externí knihovně pro podporu tvorby grafových struktur zvanou `JGraphX` (kapitola 4.1.2). Tato třída je společná pro oba druhy grafů, které podporují, pro složené DEVS modely a modely Petriho sítí. Rozlišení typu editovaného grafu se provádí až v úrovni samotného modelování, kde dochází k verifikaci vkládaných položek, mimo jiné na základě přiřazeného modelu (prvku repozitáře).

Použitou knihovnu ovšem bylo třeba přizpůsobit podobě a směru aplikace, protože je vytvořena až příliš obecně. Mezi provedené úpravy se řadí úpravy grafických stylů, dialogová editační okna pro úpravu dat náležících prvku grafu, vkládání zlomových bodů hran (waypoints) nebo validace vložených prvků grafu a jejich propojení.

[4] Java Bean je znovupoužitelná programová komponenta Javy, která zapouzdřuje atributy objektů (přístup k nim je dostupný skrze gettery a settery). V případě změny atributu se tato informace vždy zasílá všem registrovaným posluchačům (event listeners). Více informací lze nalézt na http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html.

Každý typ vkládané komponenty do grafu, ať už uzel nebo hranu, je ovšem třeba upravit. Proto v balíku `client.editor.graph.components` vznikla celá hierarchie tříd pro reprezentaci vlastních komponent. Složené DEVS modely se mohou skládat pouze ze čtyř prvků – dílčího modelu (třída `DevsGraphComp`), vstupního portu (třída `PortInGraphComp`), výstupního portu (třída `PortOutGraphComp`) a hrany (třída `EdgeNormalGraphComp`). Petriho sítě tvoří také čtyři části – místo (třída `PlaceGraphComp`), přechod (třída `TransitionGraphComp`), hrana (třída `EdgeNormalGraphComp`) a testovací hrana (třída `EdgeTestGraphComp`). Takto vytvořené komponenty definují vlastní grafický styl, geometrii a hodnotu (data reprezentovaného prvku) a lze je tak vkládat (vykreslovat) v grafu.

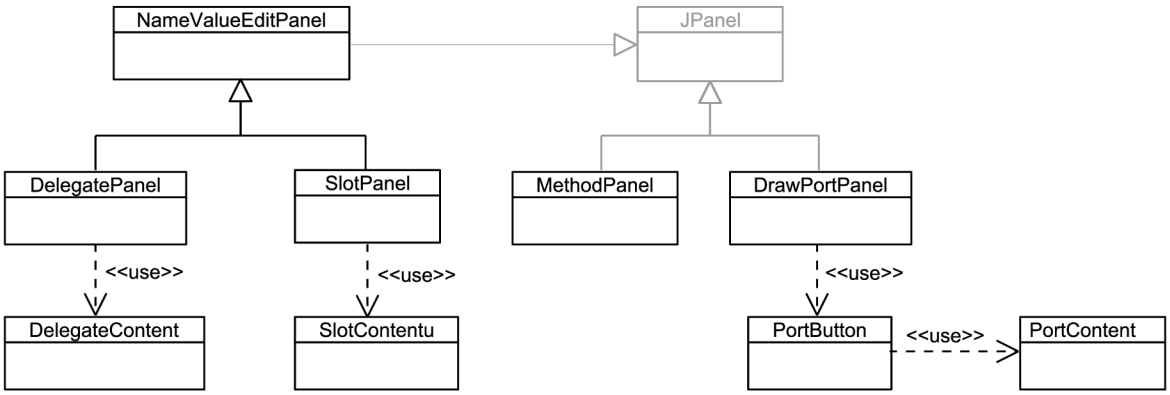
Pro vkládání jednotlivých položek grafů je použita metoda „drag and drop“ (táhni a pusť) v kombinaci se základními editačními operacemi (kopírování prvků apod.). Další operace se sítí uživatel intuitivně nalezne a pochopí, ale za zmínku stojí možnost zobrazit si mřížku, přichytávání prvků k mřížce nebo přibližování a oddalování samotného grafu.

Pro navigaci skrze historii úkonů provedené nad grafem se využívá třídy `UndoManagerGraph`, která rozšiřuje vlastnosti vestavěného správce historie v knihovně `JGraphX`. Vlastnostmi je myšleno především získávání jen části položek v seznamu reprezentující historie.

Náhled na výslednou podobu obou grafických editorů je umístěn v příloze C, konkrétně se jedná o obrázky C.1 (složený DEVS model) a C.3 (model Petriho sítě).

Editor atomických DEVS modelů

Každá složka prototypového atomického DEVS modelu (třída `EditorAtomDevs`) je v editoru reprezentována, pokud je to žádoucí, vlastní strukturou zaobalenou do své vizuální komponenty, což znázorňuje obrázek 4.2. Třída komponenty `NameValueEditPanel` vytváří jednoduchý panel s dvojicí textových komponent a případně jejich editačních tlačítek. Panel pro metodu `MethodPanel` se neváže na žádnou reprezentaci vlastní struktury (třídy), neboť je zastoupen pouhým řetězcem se zdrojovým kódem metody.



Obrázek 4.2: Hierarchie tříd znázorňující závislost vizuálních komponent na reprezentaci struktur prototypového atomického DEVS modelu.

Metody lze případně dělit do tří skupin: pro samotný formalismus DEVS, simulace (inicializace, spuštění nebo zastavení simulace) a ostatní. Toto dělení však dosud implementované jádro aplikace SmallDEVS nepodporuje (jeho grafické rozhraní však ano), proto po uložení a načtení atomického DEVS modelu jsou všechny metody umístěny do skupiny „ostatní“.

Pro posílení kvalitní a pohodlné editace, a také zajištění konzistence použitých metod pro ukládání editovaných prvků zpět na server, je vytvořen vlastní správce historie. Jeho

implementaci představuje třída `UndoManagerAtomicDevs`, která se váže na všechny tři druhy panelů a komponentu portu (`SlotPanel`, `MethodPanel`, `DelegatePanel` a `PortButton`). Pro uložení jednotlivých bodů úprav (historie) se využívá tříd z balíku pro konvertování historie pro uložení na server (`client.editor.convertors`). Podporuje klasické operace jako přidávání, odebírání a změnu obsahu.

Výslednou grafickou podobu editoru pro atomické DEVS komponenty je možné si prohlédnou na obrázku C.2.

Editor prostého textu

Základem editoru prostého textu (třída `EditorText`) lze považovat převzatou editační komponentu z externí knihovny `RED`. Tím se zpřístupnily základní možnosti, jako je kopírování, vkládání textu a historie úprav. Jelikož pro ukládání textového prvku repozitáře není potřeba použít manažer historie pro konvertování změn, je možné použít vestavěný ve zmíněné knihovně. Editor z knihovny `RED` pracuje obdobně jako jemu podobné editory, takže jeho bližší popis je nadbytečný.

4.3.5 Načítání a ukládání modelů

Postup pro načítání modelů ze serveru do reprezentace zobrazitelné klientem (editorem) se definuje pomocí typu prvku repozitáře, v případě prvku simulace se bere v potaz přiřazený model. Zvolené postupy se od sebe logicky liší volbou zasílaných zpráv (požadavků) serveru a převodem jejich odpovědí do interní podoby daného editoru. Při získávání dat ze serveru a jejich konverzi ovšem mohou nastat chyby, o kterých je uživatel informován a proces načítání je přerušen.

V případě načítání modelu, vyznačující se grafovou reprezentací, se nejprve načtou a vykreslí dílčí komponenty a následně jejich spoje (hrany). Tím se zabrání chybám při propojování jednotlivých komponent. Pokud je ovšem modelem atomické DEVS komponenta, případně prototyp nebo trait (kapitola 3.2), pak je pořadí načítání jeho částí prakticky libovolné. V implementaci je zvolen následující postup – prvně porty (vstupní i výstupní), následně metody, sloty a delegáti.

Opakem načítání, ukládání, modelů se blíže věnuje následující podkapitola. Je vhodné ještě poznamenat, že oba směry přenosu modelů, nebo textových dokumentů, mezi serverem a klientem obstarávají třídy `ClientMainHandler` a `ClientGraphHandler`.

Ukládání modelů

Implementace přenosu provedených změn v jednotlivých modelech se opírá o koncept z kapitoly 3.2.1, z něhož vyplývá potřeba vytvořit převaděč úkonů z formátu manažera historie do programově použitelných struktur.

Převaděč je realizován ve třídě `HistoryConvertor`, který slouží pro konvertování změn z historie úprav grafů do jednotné formy změn dědící vlastnosti z abstraktní třídy `CommonChange`. Vždy tak lze mít pouze čtyři typy změn: *vytvářecí* (AddChange), *odstraňovací* (`RemoveChange`), *měnící hodnotu* (ValueChange) a *poziční změnu* (PositionChange). Změny vždy obsahují atributy zdroje změny, původní a nové hodnoty. Nad zkonvertovanou historií je nutno provést sadu navržených heuristik tak, aby bylo umožněno efektivnějšího přístupu k datům.

Při ukládání může dojít k jakékoliv chybě, proto je nutné zabezpečit konzistenci dat mezi klientem a serverem, toho je docíleno dvouúrovňovým ukládáním:

1. úroveň je *testovací*, což značí otestování uložení modelu na server v podobě dočasného modelu. Nejprve se tedy provede konverze historie změn a na základě získaných dat dochází k ukládání do dočasného modelu. Pokud nastane jakákoliv chyba (kromě chyby spojení), je dočasný model odstraněn a ukládání se přeruší. Původní model je v této fázi nezměněn.

2. úroveň využívá již připravených operací z první úrovně a předpokládá, že uložení proběhne z pohledu validity změn v pořádku. Provádí *konečné uložení* do původního modelu a poté odstranění model dočasný. Tím je proces ukládání v pořádku ukončen.

Pořadí reflektování změn je dáno požadavkem zamezení chyb v posloupnosti zasílání požadavků na serveru, tj. aby nedošlo k úpravě hodnoty nové dílčí komponenty dokud není vytvořena. Nejprve se tedy provedou odstraňovací operace, za nimi následující operace vytvářecí a nakonec editační operace (měnící například názvy) dílčích komponent modelu.

4.3.6 Simulace

Samotný problém simulace se díky řešení klient - server aplikace přesunuje na stranu serveru. Klient tedy uživateli pouze zprostředkovává ovládání, nastavení a informace o simulaci. Nastavením je myšleno zadaní, či případná změna atributů simulace, mezi které se řadí konečný čas (trvání) simulace a její real - time faktor (rychlost simulace vzhledem k reálnému času), kořenový model pro simulaci a umístění záznamu o běhu simulace do protokolujícího prvku repozitáře.

Zobrazení výstupu (*protokolu*) běhu simulace si uživatel může zpřístupnit pomocí otevření patřičného prvku repozitáře v editoru prostého textu. V průběhu testování však bylo zjištěno, že uvedené řešení není příliš vhodné, neboť záznamy běhu simulací mohou dosahovat opravu velkých rozměrů (není výjimkou i 1 GiB). Přenos takto rozsáhlých dat skrze běžné připojení pak trvá zbytečně dlouho, nemluvě o zobrazení textu o velikosti 1 GiB v editoru. V dalších verzích je možností řešení jednak zobrazit jen uživatelem definovaný úsek dat, nebo kontinuální zobrazování protokolu simulace při jejím běhu (kapitola 6.1).

Kapitola 5

Testování

Poslední fází vývoje aplikací, nebo softwaru obecně, musí být ověření implementovaného řešení. Testování mého řešení klient - server aplikace pro modelování a simulaci na bázi DEVS jsem rozdělil na dvě části za využití různých metod s postupů pro získání zpětné vazby o funkčnosti aplikace. První částí je testování serverové části, kterému se věnuje podkapitola 5.1. Ve druhé podkapitole 5.2 (testování klientské části) se, kromě samotného ověřování implementace klienta, navíc diskutují dosažené výsledky v porovnání s výchozí aplikací SmallDEVS.

5.1 Testování serverové části

Vzhledem k implementaci serveru pomocí telnet protokolu je možno *automatizovaně* ověřovat funkčnost sadou testovacích vstupů. Funkčností rozumíme korektní provedení operací odpovídající žádostem, které zasílá klient.

Byl vytvořen jednoduchý shellový skript (série příkazů terminálu v OS Unix), který postupně čte vstupní soubory a zasílá jejich obsah na server. Následně se porovnává předpokládaný výstup s aktuálně obdrženým. Vstupní soubory (v adresáři `/in`) jsou definovány názvem ve tvaru `*.in`. Obsah vstupních souborů tvoří několik požadavků, jejichž popis se nachází v příloze E, které jsou ukončeny direktivou na samostatném řádku `#end`. Další dostupná direktiva, značená jako `#clean`, indikuje odstranění dosud obdrženého výstupu odpovědí ze serveru. Řádkové komentáře, jež jsou při vyhodnocování přeskakovány, obecně začínají znakem `#`.

Ukázka vstupního souboru testování se nachází v příloze D a představuje test vytvoření jednoduché vysokoúrovňové Petriho sítě s dvěma místy a jedním přechodem, které jsou propojeny testovací nebo normální hranou. Do přechodu se navíc vkládá stráž i akce. Poslední část (za direktivou `#clean`) naznačuje, jak ze serveru získat informace pro rekonstrukci sítě.

5.2 Testování klientské části

Testování klientské části bylo o poznání obtížnější, neboť využití automatizovaných testů nad grafickým rozhraním aplikace je značně náročné. Proto ověřování funkčnosti jádra klienta probíhalo *manuálně* a prováděl jsem ji v prvotních fázích sám, jako vývojář. Pro provedení testů navrženého a realizovaného grafického rozhraní bylo požádáno několik spolužáků a člověk, do informačních technologií nezasvěcený. Testování probíhalo ve dvou operačních systémech: Linux (Mint verze 14) a Widnows (verze 7 a 8). Po představení předpokládané

funkčnosti všem účastníkům jsem jejich počínání nejprve sledoval a reagoval na jejich výtky. Ve chvíli, kdy již byli zasvěceni do aplikace a testovali funkčnost osamoceně, se mi sešlo několik reakcí, ze kterých jsem, mimo jiné, použil zajímavý návrh na zobrazování aktuální činnosti aplikace. Z čehož vyplynulo rozšíření v podobě protokolování událostí.

Použitelnost vytvořené klientské aplikace se ověřovala na sadě modelů na bázi DEVS, které jsou již v aplikaci SmallDEVS definovány. Mezi DEVS modely, na kterých byla ověřována funkčnost modelace a simulace, se řadí: Cart-And-Pole System (systém „vozíku s tyčí“), Generator And Processor („Generátor a zpracovatel“), Generator And 3 Processors („Generátor se třemi zpracovateli“), Generator And 3 Processors With Shared Memory („Generátor se třemi zpracovateli se sdílenou pamětí“) a Dynamic Structure Example (model pro výpočet faktoriálu).

Vymodelovaný spojovaný DEVS model s názvem Cart-And-Pole System se nachází na obrázku C.1 včetně celé aplikace. Jelikož jsem se snažil zachovat strukturu editace podobnou jako v aplikaci SmallDEVS, lze modelovat stejný systém v obou aplikacích při zachování stejné podoby. V případě jádra modelu Cart-And-Pole System, které je atomickým DEVS modelem nesoucí název `cart and pole`, byla snaha o zachování konzistence editace daného typu modelů v rámci obou aplikací. Náhled na vytvořený model `cart and pole` je umístěn v obrázku C.2.

Ukázka jednoduchého modelu vysokoúrovňové Petriho sítě vyobrazuje náhled C.3. Jak již bylo řečeno v implementační části 4.2.3, SmallDEVS ve svém základu neimplementoval podporu pro modelování vysokoúrovňových Petriho sítích, proto není možné srovnat dosažené výsledky mé aplikace a existujícího řešení.

Při srovnání vytvořené aplikace pro modelování a simulaci na bázi DEVS a existujícího řešení (SmallDEVS) nad testovanými modely vyvstává řada výhod i nevýhod provedeného řešení. Nejprve tedy podstatné výhody mého řešení:

- Přehlednější a intuitivnější možnosti editace modelování.
- Zobrazení většiny potřebných informací o modelu na jednom místě.
- Přívětivější grafické rozhraní.
- Oddělení klienta od výpočetně náročného běhu simulace.

Na druhou stranu jako nevýhody lze označit následující body:

- Vytrácí se možnost přímé úpravy instancí prototypů skrze Inspektor ve Squeaku.
- Rychlost klienta je obecně limitována propustností sítě.
- V celku vysoká paměťová náročnost[1] klienta (editoru) pohybující se od hodnoty 70 MiB (po spuštění) přes 300 MiB (připojení k serveru a načtení repozitáře) a výše v závislosti na otevřených jednotlivých editorů. Všechny uvedené hodnoty odpovídají testování na OS Linux. Řešením v dalších verzích by bylo provést profilování aplikace a jejím základe provedení paměťové optimalizace[2].

[1] Paměťová náročnost je hodnota spotřebované (adresované) paměťových prostředků počítače (operační paměť RAM) při běhu dané aplikace.

[2] Java stroj využívá vlastního nástroje pro správu paměti zvaného *Garbage Collector*, proto je problematické ji spravovat ručně a tak dochází k navýšení spotřeby paměťových prostředků.

Kapitola 6

Závěr

V rámci bakalářské práce byla nejprve navržena a po té implementována klient - server aplikace pro modelování a simulaci na bázi DEVS (systémy s diskrétními událostmi). Po analýze požadavků, kde byly představeny možnosti reprezentace modelů pomocí DEVS formalismu nebo Petriho sítě, a prostudování existujících řešení na poli nástrojů pro modelování a simulace byly navrhnuty a implementovány dvě samostatné aplikace.

Klient (editor) reprezentuje rozhraní pro uživatele poskytující přístup k modelům a simulacím umístěných na vzdáleném serveru. Implementován je jako samonosná multiplatformní aplikace bežící na operačních systémech Windows i Linux, což ještě potenciálně rozšiřuje uživatelskou základnu. Server zpřístupňuje modely a jejich simulace více uživatelům současně. Implementace proběhla v multiplatformním prostředí Squeak (Smalltalk) a je postaven na již existujícím řešení aplikace SmallDEVS.

Obě aplikace byly náležitě otestovány prostřednictvím automatizovaných i manuálních testů reálných koncových uživatelů. Na základě jejich připomínek se řešení přizpůsobilo tak, aby vyhovovalo a bylo validní. Ve výsledku této bakalářské práce je vytvořena funkční klient - server aplikace jako další alternativa pro tvorbu modelů na bázi DEVS a jejich simulaci.

6.1 Možnosti dalšího vývoje

Možnosti další vývoje jsou z pohledu dalšího přizpůsobení se požadavkům uživatelů poměrně široké (především grafického rozhraní klienta). Zaměřil bych se však na dvě rozsáhlejší změny, se kterými aktuální návrh počítá.

První předpokládaná úprava spočívá v navázání klientské části (editoru) na server aktuálně vyvíjeným v diplomové práci kolegy Bc. Michala Šimary. Tento zásah do samotné implementace jádra editoru by měl být minimální, díky oddělení komunikačního rozhraní se serverem. Bude ale třeba nové komunikační rozhraní realizovat a zasadit jej do stávajícího řešení klientské části aplikace.

Druhým potenciálním rozšířením je průběžné zobrazování protokolu událostí generované běžícími simulacemi na straně serveru, které by usnadnilo přístup k rozsáhlým textovým záznamům. Řešením by bylo ustanovení nového spojení určeného pouze pro zasílání simulačního protokolu, kde by každý záznam obsahoval unikátní a jednoznačný identifikátor simulace. Dle příchozího záznamu s identifikátorem by se rozdělil výpis informací do samostatných záložek v již implementovaném zobrazování protokolu o činnosti editoru. Alternativou k uvedenému řešení by byl výběr uživatelem určeného úseku záznamu.

Literatura

[1] Cabac, L.: *Modeling Petri Net-Based Multi - Agent Applications*. Dizertační práce, Universität Hamburg, Hamburg, Germany, 2010.

[2] Janoušek, V.: *Modelování objektů Petriho sítěmi*. Dizertační práce, Brno, CZ, 1998.

[3] Janoušek, V.: *Simulace a návrh vyvíjejících se systémů*. Brno, CZ: Fakulta informačních techlogií VUT, 2011, ISBN 978-80-214-4414-0.

[4] Kurt, J.: Coloured Petri Nets: A high level language for system design and analysis. In *Proceedings on Advances in Petri nets 1990*, APN 90, New York, NY, USA: Springer-Verlag New York, Inc., 1991, ISBN 0-387-53863-1, s. 342–416.

[5] Loy M., Eckstein R.: *Java Swing*. Developing GUIs in Java, O'Reilly Media, Incorporated, druhé vydání, 2002, ISBN 978-05-960-0408-8.

[6] Postel J., Reynolds J. K.: RFC 854: Telnet Protocol Specification. [online], Květen 1983, aktualizováno v RFC 5198.
Dostupné z: http://tools.ietf.org/html/rfc854

[7] Stevens W. R., Fenner B., Rudoff A. M.: *Unix network programming: The Sockets Networking API*. Addison-Wesley, třetí vydání, 2004, ISBN 01-314-1155-1.

[8] Zeigler B. P., Praehofer H., Kim T.: *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. San Diego, USA: Academic Press, druhé vydání, 2000, ISBN 978-01-277-8455-7.

Příloha A

Obsah přiloženého CD

`./apidoc/` (adresář) – vygenerovaná dokumentace aplikačního rozhraní klienta (editoru) ze zdrojového kódu a jeho komentářů (`javadoc`)

`./apidoc/index.html` (soubor) – otevření vygenerované dokumentace aplikačního rozhraní klienta (editoru) ve webovém prohlížeči

`./doc/` (adresář) – elektronická verze této technické zprávy a její zdrojové soubory v LaTeXu

`./run/client/` (adresář) – spustitelná aplikace klienta a jeho náležitosti

`./run/client/README.txt` (soubor) – návod na instalaci a spuštění klienta (editoru)

`./run/server/` (adresář) – prostředky potřebné pro instalaci serveru v prostředí Squeak

`./run/server/README.txt` (soubor) – návod na instalaci a spuštění serveru v prostředí Squeak

`./src/client/` (adresář) – zdrojové soubory klientské části aplikace

`./src/server/src/` (adresář) – zdrojové soubory serverové části aplikace

`./src/server/tests/` (adresář) – testovací soubory serverové části aplikace

`./README.txt` (soubor) – manuál k aplikaci

Příloha B

Manuál

Ovládání klienta

V této části manuálu jsou uvedeny základy ovládání klienta (editoru), podrobnější informace lze nalézt v nápovědě programu (položka menu s názvem Help)

Pro připojení klienta je nutné v hlavni nabídce zvolit `Server->Connect To` a následně vyplnit název hostujícího počítače na sítí (hostname nebo IP adresa) a port. Tyto údaje si může uživatel uložit pro další použití. Znovu načíst aktualizovaná data (obnovit klienta) lze buď manuálně stisknutím tlačítka v levém spodním rohu okna klienta, nebo nastavením intervalu obnovy (hlavní nabídka `Edit->Preferences`).

Vytvořit nový prvek repozitáře lze jednak z hlavní nabídky `File->New`, nebo po zobrazení vyskakující nabídky nad prvkem v repozitáři. Tato nabídka také disponuje základními editačními operacemi (kopírování apod.), otevřením editoru nad daným prvkem a případně i ovládání simulací. Alternativou k otevření editoru z vyskakující nabídky je dvojklik nebo klávesa `ENTER` nad označeným prvkem v repozitáři.

Ukládání provedených změn obsahu prvku v editoru lze provést jedna z lišty editoru, nebo hlavního menu (`File->Save`), nebo klávesou zkratkou `Ctrl+S` nad panel (oknem) otevřeného dílčího editoru. Uživateli je také umožněn tisk grafové struktury pomocí tiskárny nebo do souboru (`*.ps` případně `*.pdf`).

Modelování grafových struktur probíhá pomocí metody „táhni a pusť“ (drag and drop) z palety dostupných prvků. Lze také použít metodu kopírování (copy/cut and paste). Samozřejmostí editorů je klasická navigace v historii úkonů (Undo – `Ctrl+Z`, Redo – `Ctrl+Y`).

Prvky repozitáře


Složený DEVS model


Atomický DEVS model


Model Petriho sítě

Běžící simulace


Zastavená simulace


Trait nebo prototyp


Složka (adresář)


Petriho síť definovaná v jazyce PNTalk


Obyčejný textový prvek (obsahuje prostý text)


Textový prvek, která obsahuje zdrojový kód LaTeXu

Textový prvek obsahující výpis dat ze simulace

Ovládání serveru

Server lze spustit dvěma způsoby (vždy je ale třeba zadat číslo portu, v uvedených příkladech je to 33333):

1. Použít klasické instanciace objektů pomocí metody `new`, ovšem nevýhodou je nutnost použít časovač (limit trvání běhu serveru).

```
| server |
server := DevsServer new.
server startOn: 33333.
(Delay forSeconds: 5) wait.
server stop.
```

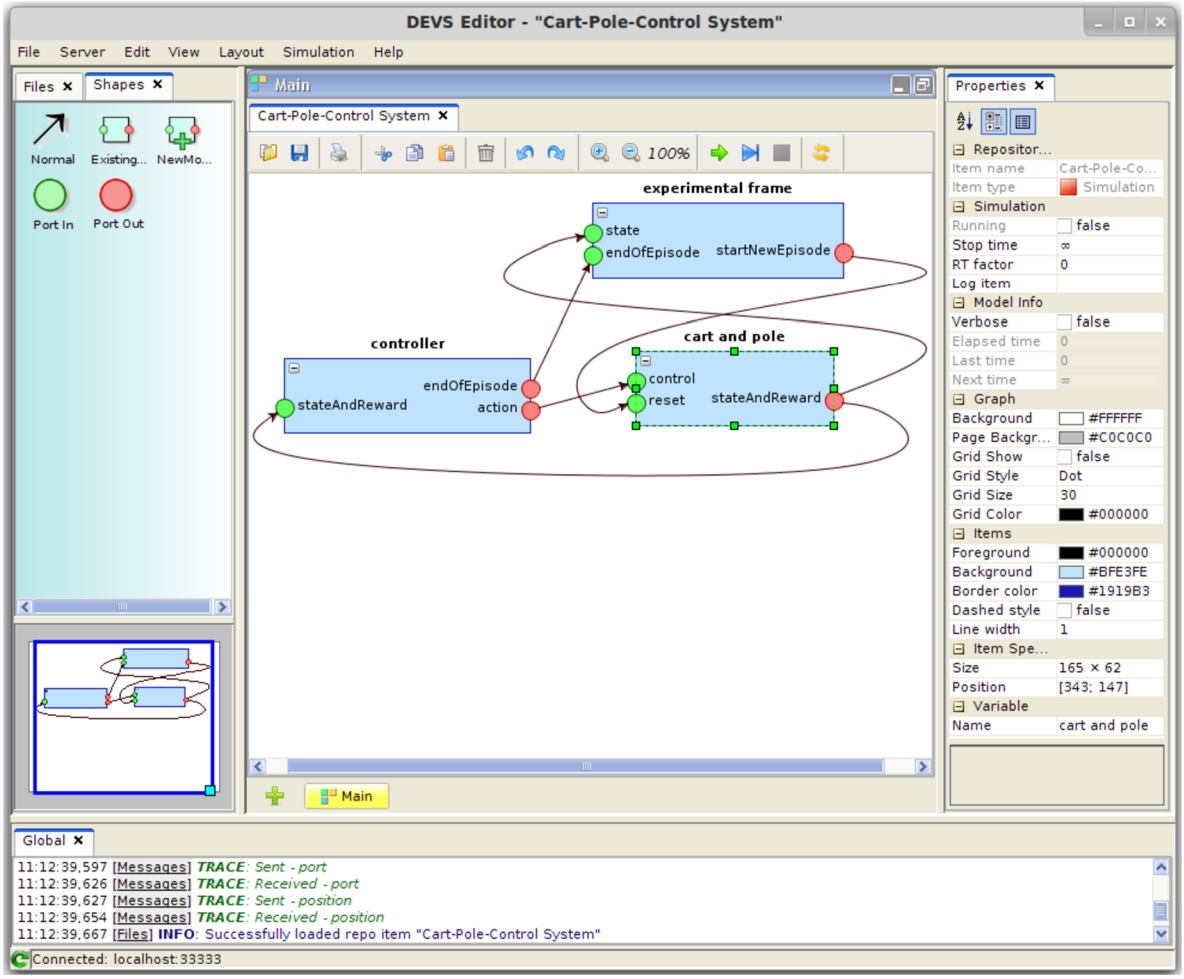
2. Použít statické (třídní) metody `startOn:` (s frontou čekajících o velikosti 4) a `stop`. První (nebo druhá) volání metody spustí server a po vyhodnocení metody `stop` je server zastaven.

```
DevsServer startOn: 33333.
" or next line can be used to specify backlog size "
DevsServer startOn: 33333 size: 50.
DevsServer stop.
```

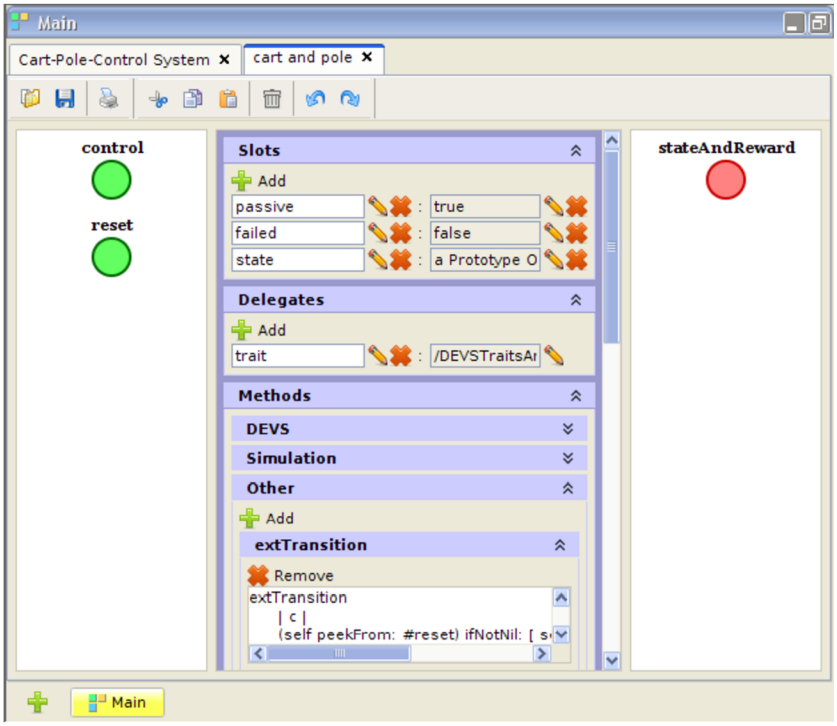
Pro protokolování událostí na serveru (příjem požadavků, obdržené výjimky při zpracování požadavků apod.) je využita `Transcript` komponenta Squeaku.

Příloha C

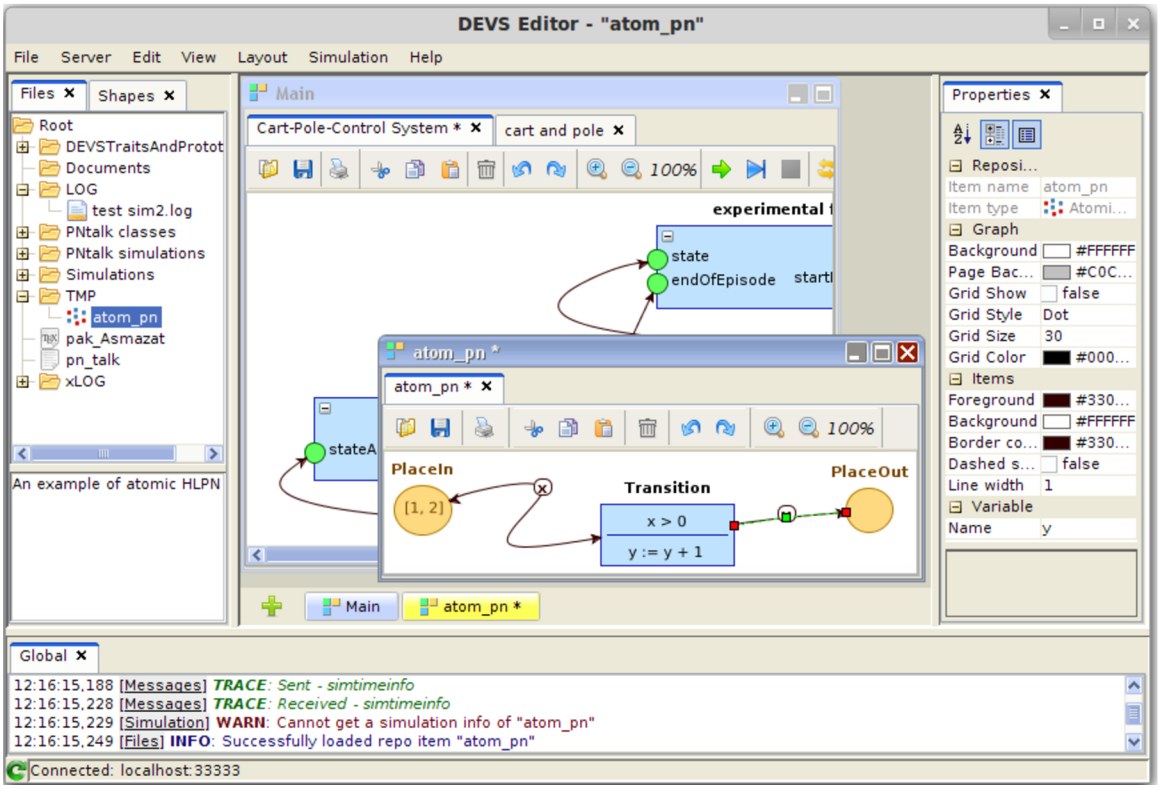
Grafického rozhraní klientské aplikace



Obrázek C.1: Ukázka grafického rozhraní s načteným složeným DEVS modelem systému „Vozíku s tyčí“ (Cart And Pole system). Také je zde patrný panel s nastavením označeného dílčího modelu `cart and pole`.



Obrázek C.2: Ukázka editoru prototypového atomického DEVS modelu.



Obrázek C.3: Grafické rozhraní s více - okenním prostředím s ukázkou editoru modelu Petriho sítě a panelu s hierarchickým zobrazením obsahu repozitáře serveru.

Příloha D

Ukázka testovacího souboru serveru

```
# recreate TMP folder and create Atomic Petri net component
remove --target="/TMP/"
#end
create --target="/" --type="folder" --name="TMP"
#end
create --target="/TMP/" --type="atompn" --name="atom_pn"
#end
# modeling of Atomic Petri net component
component --target="/TMP/atom_pn" --set=(!'"PlaceIn" -> "place"'!, !'"Transition" -> "
    transition"'!, !'"PlaceOut" -> "place"'!) --add
#end
coupling --target="/TMP/atom_pn" --set=(!'"PlaceIn", "test" -> "Transition"'!, !'"
    Transition" -> "PlaceOut"'!) --add
#end
pnguard --target="/TMP/atom_pn" --name="Transition" --set="x > 0"
#end
pnaction --target="/TMP/atom_pn" --name="Transition" --set="y := y + 1"
#end
pntoken --target="/TMP/atom_pn" --name="PlaceIn" --set=(!'1'!, !'2'!) --add
#end
pncondition --target="/TMP/atom_pn" --name="Transition" --pos="" --set=(!'"PlaceIn" ->
    "x"'!)
#end
pncondition --target="/TMP/atom_pn" --name="Transition" --pos="pre" --set=()
#end
pncondition --target="/TMP/atom_pn" --name="Transition" --pos="post" --set=(!'"PlaceOut
    " -> "y"'!)
#end
position --target="/TMP/atom_pn" --set=(!'"PlaceIn" -> 106@147'!, !'"Transition" -> 261
    @161'!, !'("PlaceIn" -> "Transition") -> 106@147 176@137 206@224 261@161'!, !'("
    Transition" -> "PlaceOut") -> 261@161 426@155'!, !'"PlaceOut" -> 426@155'!)
#end
# verifying Atomic Petri net component
#clean
component --target="/TMP/atom_pn"
#end
pnguard --target="/TMP/atom_pn" --name="Transition"
#end
pnaction --target="/TMP/atom_pn" --name="Transition"
#end
pntoken --target="/TMP/atom_pn" --name="PlaceOut"
#end
pntoken --target="/TMP/atom_pn" --name="PlaceIn"
#end
pncondition --target="/TMP/atom_pn" --name="Transition" --pos=""
#end
pncondition --target="/TMP/atom_pn" --name="Transition" --pos="pre"
#end
pncondition --target="/TMP/atom_pn" --name="Transition" --pos="post"
#end
position --target="/TMP/atom_pn"
#end
```

Příloha E

Komunikační protokol

Odpovědi serveru (zprávy pro klienta)

Popis chybových kódů:

`0` – Požadavek byl zpracován bez chyby (vše v pořádku).

`1` – Nastala obecná chyba na serveru.

`2` – Syntaktická chyba požadavku.

`3` – Sémantická chyba požadavku.

`4` – Syntaktická nebo sémantická chyba vkládaného Smalltalk zdrojového kódu (chyba při překladu). Například při vkládání metod.

Žádosti klientů (zprávy pro sever)

Případná hodnota parametru typu seznamu je vždy ve tvaru: `(items)` kde `items` je obsah seznamu.

Základní operace s repozitářem a jeho obsahem

`cd` – Přejde na zadanou komponentu, která může obsahovat dílčí komponenty.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.

`type` – Vypíše typy zadané komponenty oddělené čárkou. Dostupné typy jsou: document (prostý textový dokument), latex (LaTeXdokument), log (Log dokument), pnclass (Petriho síť definovaná pomocí PNTalk), prot_trait (trait nebo prototyp), simulation_run (běžící simulace), simulation (zastavená simulace), coupldevs (složený DEVS model), atomdevs (atomický DEVS model), atompn (model Petriho sítě), folder (složka), atom (atomická komponenta = není složka). V případě, že komponeta je simulace, pak je typ rozšířen o typ jejího root DEVS modelu.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.

`is` – Zjistí, zda komponenta je požadovaného typu (dle `type`).

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
`type=`	Ano	Typ komponenty v uvozovkách, dle `type`.

`list` – Vypíše seznam dílčích komponent.

Parametr	Povinný	Popis
bez dalších parametrů		Zobrazí seznam přímých dílčích komponent od aktuální pozice v repozitáři (lze změnit pomocí `cd`).
`all`	Ne	Zobrazí kompletní seznam dílčích komponent (i zanořených) od aktuální pozice.
`types`	Ne	K výpisu přidá typy (dle `type`) komponenty ve tvaru `[TYPE1,TYPE2] PATH`, kde `,TYPE2` nemusí být definován.

`path` – Získá absolutní cestu k požadované komponentě.

Parametr	Povinný	Popis
bez dalších parametrů		Zobrazí absolutní cestu komponenty na aktuální pozici v repozitáři (lze změnit pomocí `cd`).
`target=`	Ne	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.

`name` – Získá název požadované komponenty. Parametry a jejich význam jsou v relaci s funkcí tohoto požadavku jako u `path`.

`store` – Získá řetězec, ze kterého lze požadovanou komponentu zrekonstruovat. Parametry a jejich význam jsou v relaci s funkcí tohoto požadavku jako u `path`.

`sixx` – Obdobně jako požadavek `store`, jen získaný text je ve formátu SIXX (Smalltalk Instance eXchange in XML).

`filecontent` – Získá (nastaví) obsah textového dokumentu.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
`set=`	Ne	Nový obsah dokument v uvozovkách.
`delete`	Ne	Odstraní všechen text z dokumentu.

`create` – Vytvoří novou komponentu na zadaném umístění.

Parametr	Povinný	Popis
`target=`	Ano	Cesta k nadřazené komponentě v uvozovkách, kde Root je `"/"`.
`type=`	Ano	Typ nové komponenty v uvozovkách, dle `type`.
`name=`	Ano	Název nové komponenty v uvozovkách.

`remove` – Odstraní komponentu, přičemž Root komponentu odstranit nelze.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.

`rename` – Přejmenuje komponentu, přičemž Root komponentu přejmenovat nelze.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
`name=`	Ano	Nový název komponenty v uvozovkách.

`copy` – Vytvoří hloubkovou kopii požadované komponenty a uloží ji na zadanou cestu, případně odstraní původní komponentu.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke zdrojové komponentě v uvozovkách, kde Root je `"/"`.
`to=`	Ano	Cesta k nové cílové komponentě (možno zadat její nový název) v uvozovkách, kde Root je `"/"`.
`remove`	Ne	Parametr udává, zda odstranit zdrojovou komponentu.

Editační operace s pozičními modely

`component` – Umožňuje získání seznamu, přidání a odebrání dílčích komponent v síti/modelu.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí seznam dílčích komponent ve tvaru: `"COMP_NAME" -> "COMP_PATH"` odřádkovaný pro složený DEVS model, `"COMP_NAME" -> "PN_COMP_TYPE"` odřádkovaný pro Petriho síť, kde `PN_COMP_TYPE` nabývá hodnot `place`, nebo `transition`.
`set=`	Ne	Nastaví dílčí komponenty zadané seznamem ve tvaru: `!'"COMP_NAME" -> "COMP_PATH"'!` oddělený čárkami pro složený DEVS model, `!'"COMP_NAME" -> "PN_COMP_TYPE"'!` oddělený čárkami pro Petriho síť.
`add`	Ne	V kombinaci s parametrem `set=` přidá dílčí komponenty.
`remove=`	Ne	Odstraní dílčí komponenty zadané seznamem ve tvaru: `!'"COMP_NAME"'!` oddělený čárkami.
`removeAll`	Ne	Odstraní všechny dílčí komponenty.
`old=`	Ne	V kombinaci s parametrem `new=` přejmenuje dílčí komponentu. Hodnotou je řetězec s *původním* názvem.
`new=`	Ne	V kombinaci s parametrem `old=` přejmenuje dílčí komponentu. Hodnotou je řetězec s *novým* názvem.

`coupling` – Umožňuje různé editační operace se spoji (hranami) modelů.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí odřádkovaný seznam spojů ve tvaru: `"C_B_NAME", "P_OUT_NAME" -> "C_E_NAME", "P_IN_NAME"` pro spoje složeného DEVS modelu, `"C_B_NAME", "test" -> "C_E_NAME"` pro hrany Petriho sítí, přičemž `, "test"` indikuje testovací hranu a není tedy povinný.
`at=`	Ne	Získá spoje vystupující z dílčí komponenty zadané názvem v uvozovkách. Výsledek je stejného tvaru jako u získání všech spojů.
`set=`	Ne	Nastaví spoje zadané seznamem ve tvaru: `!'line'!` kde `line` jsou tvary stejné jako u získání spojů.
`add`	Ne	V kombinaci s parametrem `set=` přidá spoje.
`remove=`	Ne	Odstraní spoje zadané seznamem ve tvaru obdobném jako u parametru `set=`.
`removeAll`	Ne	Odstraní všechny spoje z modelu.

`position` – Umožňuje různé editační operace se pozicemi dílčích komponent v modelu/síti.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí odřádkovaný seznam všech pozic ve tvaru: `("C_B_NAME", "P_OUT_NAME" -> "C_E_NAME", "P_IN_NAME") -> x_b@y_b x@y ... x_e@y_e` pro spoje složeného DEVS modelu, `("C_B_NAME" -> "C_E_NAME") -> x_b@y_b x@y ... x_e@y_e` pro hrany Petriho sítě, `"COMP_NAME" -> x@y` pro dílčí komponenty.
`arc=`	Ne	V použití se žádostí o získání pozic nebo parametrem `at=` udává, zda vypsat pouze pozice pro spoje (hrany), nebo dílčí komponenty. V případě neuvedení tohoto parametru se získají všechny pozice. Jedná se o `true`/`false` hodnotu v uvozovkách.
`at=`	Ne	Získá spoje vystupující z dílčí komponenty zadané názvem v uvozovkách. Výsledek je stejného tvaru jako u získání všech pozic.
`set=`	Ne	Nastaví pozice zadané seznamem ve tvaru: `!'line'!` kde `line` jsou tvary stejné jako u získání pozic.
`add`	Ne	V kombinaci s parametrem `set=` přidá pozice.
`remove=`	Ne	Odstraní pozice zadané seznamem e tvaru: `!'"C_B_NAME", "P_OUT_NAME" -> "C_E_NAME", "P_IN_NAME"'!` pro spoje složeného DEVS modelu, `!'"C_B_NAME" -> "C_E_NAME"'!` pro hrany Petriho sítě, `!'"COMP_NAME"'!` pro dílčí komponenty.
`removeAll`	Ne	Odstraní všechny pozice z modelu/sítě.

`comment` – Umožňuje různé editační operace s komentáři obou DEVS modelů, simulací a Petriho sítěmi.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí komentář požadované komponenty.
`set=`	Ne	Nastaví komentář zadaný v uvozovkách.
`remove`	Ne	Odstraní (vymaže) komentář komponenty.

`verbose` – Nastavení výpisu informací do logu pro danou komponentu (oba DEVS modely, simulace a Petriho sítě).

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí `true`/`false` hodnotu indukující, zda komponenta je nastavena do verbose módu.
`set=`	Ne	Nastaví verbose mód komponenty. Hodnotou je `true`/`false` v uvozovkách.
`deep`	Ne	V kombinaci s parametrem `set=` distribuuje nastavení do všech svých dílčích komponent.

Editační operace s DEVS modely

`slot` – Umožňuje různé editační operace se sloty atomického DEVS modelu.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí odřádkovaný seznam názvů slotů.
`at=`	Ne	Získá hodnotu slotu zadaného názvu v uvozovkách. Výsledek je ve tvaru: `!'VALUE'!, !'CODE'!` odřádkováno.
`valueAll`	Ne	Získá seznam slotů včetně jejich hodnot ve tvaru: `"SLOT_NAME" -> !'VALUE'!, !'CODE'!` odřádkováno.
`set=`	Ne	Nastaví sloty zadané seznamem ve tvaru: `!'"SLOT_NAME" -> "CODE"'!` oddělený čárkami. Pokud je `CODE` prázdný řetězec, pak se dosadí **nil**.
`add`	Ne	V kombinaci s parametrem `set=` přidá sloty.
`remove=`	Ne	Odstraní sloty zadané seznamem ve tvaru: `!'SLOT_NAME'!` oddělený čárkami.
`removeAll`	Ne	Odstraní všechny sloty z modelu.

`method` – Umožňuje různé editační operace s metodami atomického DEVS modelu.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí odřádkovaný seznam názvů metod.
`at=`	Ne	Získá zdrojový kód zadaného názvu metody (v uvozovkách).
`set=`	Ne	Nastaví (odstraní všechny metody a přidá) metodu zadané ve tvaru: `(!'METHOD_NAME[CRLF]CODE'!)` => formát je dán definicí zprávy ve Smalltalku.
`add`	Ne	V kombinaci s parametrem `set=` přidá metodu.
`remove=`	Ne	Odstraní metody zadané seznamem ve tvaru: `!'METHOD_NAME'!` oddělený čárkami.
`removeAll`	Ne	Odstraní všechny sloty z modelu.

`delegate` – Umožňuje různé editační operace s delegáty atomického DEVS modelu.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí odřádkovaný seznam názvů delegátů.
`at=`	Ne	Získá absolutní cestu zadaného názvu delegáta (v uvozovkách).
`valueAll`	Ne	Získá seznam delegátů včetně jejich absolutních cest ve tvaru: `"DELEG_NAME" -> "PATH"` odřádkováno.
`set=`	Ne	Nastaví delegáty zadané seznamem ve tvaru: `!'"DELEG_NAME" -> "PATH"'!` oddělený čárkami.
`add`	Ne	V kombinaci s parametrem `set=` přidá delegáty.
`remove=`	Ne	Odstraní delegáty zadané seznamem ve tvaru: `!'DELEG_NAME'!` oddělený čárkami.
`removeAll`	Ne	Odstraní všechny delegáty z modelu.

`port` – Umožňuje různé editační operace s porty DEVS modelu.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
`type=`	Ano	Specifikuje typ portu v uvozovkách. Přípustné hodnoty jsou – `in` pro vstupní porty a `out` pro výstupní porty.

bez dalších parametrů		Zobrazí odřádkovaný seznam názvů portů.
`at=`	Ne	Získá hodnotu slotu zadaného názvu v uvozovkách. Výsledek je ve tvaru: `!'VALUE'!, !'CODE'!` odřádkováno.
`valueAll`	Ne	Získá seznam portů včetně jejich hodnot ve tvaru: `"PORT_NAME" -> !'VALUE'!, !'CODE'!` odřádkováno.
`put=`	Ne	Přiřadí hodnoty slotů zadané seznamem ve tvaru: `!'"PORT_NAME" -> "CODE"'!` oddělený čárkami. Pokud je `CODE` prázdný řetězec, pak se dosadí **nil**.
`set=`	Ne	Nastaví porty zadané seznamem ve tvaru: `!'PORT_NAME'!` oddělený čárkami.
`add`	Ne	V kombinaci s parametrem `set=` přidá porty.
`remove=`	Ne	Odstraní porty zadané seznamem ve tvaru: `!'PORT_NAME'!` oddělený čárkami.
`removeAll`	Ne	Odstraní všechny porty z modelu.
`old=`	Ne	V kombinaci s parametrem `new=` přejmenuje port. Hodnotou je řetězec s *původním* názvem.
`new=`	Ne	V kombinaci s parametrem `old=` přejmenuje port. Hodnotou je řetězec s *novým* názvem.

Editační operace s modely Petriho sítí

`pntoken` – Umožňuje různé editační operace se značkami (tokens) v místě v dané komponentě Petriho sítě.

Parametr	**Povinný**	**Popis**
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
`name=`	Ano	Název místa komponenty v uvozovkách.
bez dalších parametrů		Zobrazí seznam značek v místě ve tvaru: `!'token'!` odřádkováno, kde `token` je obecné reálné číslo nebo řetězec.
`set=`	Ne	Nastaví značky v místě zadané seznamem ve tvaru: `!'token'!` oddělený čárkami.
`add`	Ne	V kombinaci s parametrem `set=` přidá značky do místa.
`remove=`	Ne	Odstraní značky z místa zadané seznamem ve stejném tvaru jako u `set=`.
`removeAll`	Ne	Odstraní všechny značky z místa.
`test=`	Ne	Otestuje, zda zvolené značky jsou v místě dostupné. Vrací `true`/`false` hodnotu. Vstupní hodnotou je seznam značek ve stejném tvaru jako u `set=`.

`pncondition` – Umožňuje různé editační operace s podmínkami (hodnotami na hranách) patřící k přechodu v dané komponentě Petriho sítě.

Parametr	**Povinný**	**Popis**
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
`name=`	Ano	Název přechodu komponenty v uvozovkách.
`pos=`	Ne	Určí pozici podmínky vzhledem k přechodu. Hodnota je: `"pre"` – vstupní podmínka pro vstupní hranu z místa, `"post"` – výstupní podmínka pro výstupní hranu z přechodu. Pokud není tento parametr použit nebo je prázdný `""`, pak je požadavek určen pro podmínku na testovací hraně.
bez dalších parametrů		Zobrazí seznam podmínek pro přechod ve tvaru: `"PLACE_NAME" -> "COND"` odřádkovaný, kde `COND` může být prázdný.

`at=`	Ne	Získá podmínku přechodu vázaného na zadaný název místa v uvozovkách. Výsledkem je hodnota podmínky **bez uvozovek**.
`set=`	Ne	Nastaví podmínky pro přechod zadané seznamem ve tvaru: `!'"PLACE_NAME" -> "COND"'!` oddělený čárkami, kde `COND` může být prázdný.
`add`	Ne	V kombinaci s parametrem `set=` přidá podmínky pro přechod.
`remove=`	Ne	Odstraní podmínky pro přechod vázané na místa zadané seznamem ve tvaru: `!'PLACE_NAME'!` oddělený čárkami.
`removeAll`	Ne	Odstraní všechny podmínky pro přechod.

`pnguard` – Umožňuje různé editační operace se stáží přechodu v dané komponentě Petriho sítě.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
`name=`	Ano	Název přechodu komponenty v uvozovkách.
bez dalších parametrů		Zobrazí hodnotu stráže přechodu **bez uvozovek**.
`set=`	Ne	Nastaví stráž přechodu zadanou jako řetězec v uvozovkách (může být i prázdný).
`removeAll`	Ne	Odstraní obsah stráže přechodu.

`pnaction` – Umožňuje různé editační operace s akcí přechodu v dané komponentě Petriho sítě. Parametry a jejich význam jsou v relaci s funkcí tohoto požadavku jako u `pnguard`.

Simulační operace

`simstart` – Spustí simulaci zadané simulační komponenty.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
`async`	Ne	Indikuje provedení operaci v asynchronním režimu (v separátním vlákně), což je doporučovaná volba.

`simstep` – Provede jeden simulační krok zadané simulační komponenty. Parametry a jejich význam jsou v relaci s funkcí tohoto požadavku jako u `simstart`.

`simstop` – (Po)zastavení simulace zadané simulační komponenty. Parametry a jejich význam jsou v relaci s funkcí tohoto požadavku jako u `simstart`.

`simreset` – (Po)zastavení simulace zadané simulační komponenty a její inicializace (simulační čas na 0 a stopTime na nekonečno). Parametry a jejich význam jsou v relaci s funkcí tohoto požadavku jako u `simstart`.

`simtimeinfo` – Získá informace o simulačních časech (čas další naplánované události, čas předchozí naplánované události a celkový simulační čas) dané simulační komponenty.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí informace o simulačních časech ve tvaru: `nextTime -> "float"[CRLF]lastTime -> "float"[CRLF]time -> "float"` kde `float` značí reálné číslo s přesností na 3 des. místa, nebo `Infinity`.

`simlog` – Získá (nastaví) logovací komponentu dané simulační komponenty.

Parametr	Povinný	Popis
`target=`	Ano	Cesta k sim. komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí cestu k logovací komponentě.
`set=`	Ano	Vloží logovací komponentu zadanou cestou v uvozovkách, Root je `"/"`.

`isrunning` – Zjistí, zda daná simulace, dle simulační komponenty, je spuštěna. Vrací `true`/`false` hodnotu.

`stoptime` – Získá (nastaví) délku simulace (stop time) dané simulační komponenty.

Parametr	Povinný	Popis
`target=`	Ano	Cesta ke komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Vrátí hodnotu „stop time“, která je obecné `float` reálné číslo i `Infinity`.
`async`	Ne	Indikuje provedení operaci v asynchronním režimu (v separátním vlákně), což je doporučovaná volba.
`set=`	Ano	Nastaví délku simulace (stop time), kde hodnota je obecné `float` reálné číslo i `Infinity` **bez uvozovek**.

`rtfactor` – Získá (nastaví) real time faktor (0 - rychlá simulace = žádný RT) dané simulační komponenty. Parametry a jejich význam jsou v relaci s funkcí tohoto požadavku jako u `stoptime`.

`rootdevs` – Získá (nastaví) kořenovou komponentu (DEVS model nebo model Petriho síť) dané simulační komponenty.

Parametr	Povinný	Popis
`target=`	Ano	Cesta k sim. komponentě v uvozovkách, kde Root je `"/"`.
bez dalších parametrů		Zobrazí cestu ke komponentě modelu.
`set=`	Ano	Vloží (**původní ale odstraní**) komponentu modelu zadanou cestou v uvozovkách, kde Root je `"/"`.