VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

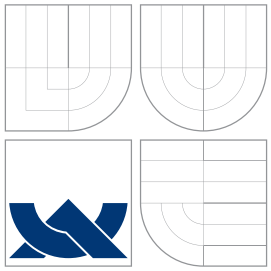
SIMULAČNÍ ARCHITEKTURA ZALOŽENÁ NA SLUŽBÁCH

DISERTAČNÍ PRÁCE
PHD THESIS

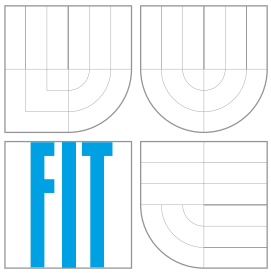
AUTOR PRÁCE
AUTHOR

Ing. PETR POLÁŠEK

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SIMULAČNÍ ARCHITEKTURA ZALOŽENÁ NA SLUŽBÁCH

SERVICE-ORIENTED SIMULATION ARCHITECTURE

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE
AUTHOR

Ing. PETR POLÁŠEK

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. RNDr. MILAN ČEŠKA, CSc.

BRNO 2014

Abstrakt

Tato práce se zabývá problematikou návrhu, modelování a simulace heterogenních systémů, s důrazem na systémy založené na diskrétních událostech. Mezi hlavní přínosy této práce patří návrh simulační architektury založené na službách a vytvoření chybějícího formátu pro popis diskrétních modelů. Možnosti navržené architektury jsou představeny na použití jejích nejdůležitějších komponent v případových studiích. V neposlední řadě práce mapuje oblast systémů založených na diskrétních událostech, se zaměřením na jejich návrh, modelování a simulaci včetně existujících podpůrných prostředků a nástrojů a jejich vzájemnou integraci.

Abstract

This thesis focuses on design, modeling and simulation of heterogeneous systems with emphasis on discrete-event systems. It proposes service-oriented simulation architecture where modeling and simulation is treated as a service and establishes a DEVS Meta Language that is intended for implementation of simulation models based on the DEVS formalism. Special M&S techniques are described and integration of existing simulation tools is discussed as well.

Klíčová slova

Modelování, simulace, multiparadigmatické modelování, reflektivní simulace, transformace modelů, simulační architektura orientovaná na služby, simulátor jako služba, webové služby

Keywords

Modeling, simulation, multiparadigm modeling, reflective simulation, model transformation, simulation architecture, service-oriented simulation architecture, simulator as a service, web services

Citace

Petr Polášek: Simulační architektura založená na
službách, disertační práce, Brno, FIT VUT v Brně, 2014

Simulační architektura založená na službách

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením svého školitele Prof. RNDr. Milana Češky, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Petr Polášek
31. srpna 2014

Poděkování

Na tomto místě bych rád poděkoval všem, kteří přispěli ke vzniku tohoto textu. Byl to především můj školitel Prof. RNDr. Milan Češka, CSc. a Doc. Ing. Vladimír Janoušek, Ph.D. V neposlední řadě patří poděkování také mé rodině a mým blízkým za jejich trpělivost a podporu.

Obsah

Úvod **3**

1 Vymezení oblasti zájmu **5**

2 Návrh a vývoj systémů založený na modelování a simulaci **7**
2.1 Modelování a simulace 7
2.2 Inteligentní a vyvíjející se systémy 8
2.3 Systémy s diskrétními událostmi 9
2.3.1 DEVS formalismus 10
2.3.2 Petriho sítě 12
2.4 Návrh a vývoj inteligentních systémů 15
2.4.1 Experimentální programování a modelování 16
2.4.2 Multiparadigmatický přístup 18
2.4.3 Interaktivní vývoj 19
2.4.4 Model kontinuity 19
2.4.5 Reflektivní simulace 20

3 Současný stav **22**
3.1 Simulační nástroje 23
3.1.1 Renew 23
3.1.2 PNTalk 23
3.1.3 SmallDEVS 24
3.1.4 DEVSJava 24
3.1.5 PythonDEVS 25
3.1.6 DEVS-C++ 25
3.2 Nevýhody simulačních nástrojů 26

4 Motivace a cíle práce **27**

5 Architektury založené na službách **29**
5.1 Webové služby 30
5.2 Existující architektury pro modelování a simulaci 31
5.2.1 Microsoft Robotics Developer Studio 31
5.2.2 DEVS bus a HLA 32
5.2.3 Multi-Simulation Interface 32
5.2.4 Simulator Integration Platform 32
5.3 Hodnocení simulačních architektur 33

6 Simulační architektura založená na službách 34
6.1 Možnosti a výhody architektury 43
6.2 Popis modelů 44
6.2.1 DEVS Meta Language 44
6.2.2 DEVSML Framework 49
6.2.3 Petri Net Meta Language 51
6.2.4 Shadow Net System formát 52
6.3 Transformace modelů 52
6.3.1 DEVS a konečné automaty 52
6.3.2 DEVS a Petriho sítě 58
6.4 Integrace nástrojů a aplikací 62
6.5 Simulátor jako služba v cloudovém prostředí 63

7 Případová studie - systém pro řízení dopravy 65
7.1 Reflektivní simulace 65
7.2 Model systému 66
7.2.1 Model dopravy 67
7.2.2 Model systému pro řízení dopravy 71
7.2.3 Model řízení simulace 71
7.3 Simulační nástroj Renew jako služba 74
7.4 Použití knihovny modelů 77

8 Případová studie - genetické algoritmy 79
8.1 Detaily modelu 80

Závěr 82

Reference 84

Publikace autora 89

A Příklad rozhraní služeb 91

B Definiční dokument simulační služby 92

C Jazyk DEVSML 96
C.1 Příklady jednoduchých modelů 96
C.2 Model procesoru 98
C.3 Popis struktury DEVSML 102

D PNML reprezentace jednoduché Petriho sítě 106

Úvod

Modelování a simulace již od svého vzniku hraje roli důležitého prostředku při návrhu systémů. V současné době je k dispozici velké množství různých aplikací a nástrojů, které nám usnadňují tvorbu modelů komplexních systémů, umožňují jejich simulaci a my tak můžeme zkoumat jejich chování ve virtuálním prostředí. Tyto simulační nástroje se neustále vyvíjejí a stejně tak i formální metody a formalismy potřebné pro návrh a analýzu chování modelů. I přes neustálý vývoj však hlavním problémem zůstává nekompatibilita mezi jednotlivými nástroji, což znamená, že je prakticky nemožné používat více nástrojů současně. Modely jsou poplatné danému nástroji, jsou nepřenositelné a není možné budovat knihovny znovupoužitelných modelů. Z tohoto důvodu jsme nuceni používat jeden nástroj od návrhu systému až po jeho nasazení v reálném prostředí. A to i přesto, že by experimentování s modelem systému mohlo být pohodlnější v jiném nástroji, než ve kterém probíhal jeho návrh. Modelovací formalismy sice poskytují dobrý matematický model, ale narážíme zde na nedostatek univerzálních prostředků, které by umožňovaly přenositelnost modelů mezi simulačními systémy.

Další nevýhodou současných simulačních nástrojů je, že to jsou ve většině případů klasické programy běžící na lokální infrastruktuře. Jsou tak obtížně škálovatelné, náročné na údržbu a nevhodné pro týmovou práci.

V práci bude kladen důraz na systémy založené na diskrétních událostech. Tomu budou odpovídat i diskutované modelovací formalismy. Hlavním cílem práce bude zjednodušit použití a umožnit integraci rozdílných existujících simulačních nástrojů a vybraných modelovacích formalismů a stanovit univerzální prostředky pro výměnu modelů.

Jako řešení současných problémů si v této práci představíme navrženou simulační architekturu založenou na službách, definujeme její základní komponenty a představíme koncept simulátoru jako služby (SaaS). V závěru práce si pak ukážeme možnosti navržené architektury na dvou případových studiích a ověříme tak její použitelnost.

Aktuálnost probírané problematiky je možné ověřit v [26], [27], [28] a [29].

Obsah jednotlivých kapitol První kapitola se snaží vymezit oblast zájmu celé práce, do níž patří *inteligentní* a *vyvíjející se systémy*, s důrazem na *systémy založené na diskrétních událostech.*

Druhá kapitola se věnuje návrhu a vývoji systémů založeném na modelování a simulaci. Zde jsou definovány základní pojmy používané v rámci celé práce a popsány dva základní modelovací formalismy: *Discrete Event Systems Specification (DEVS)* (viz kapitola 2.3.1) a *Petriho sítě* (viz kapitola

2.3.2), které tak tvoří jednotnou modelovací platformu používanou pro popis modelů systémů založených na diskrétních událostech. Popsány jsou také výhody, které jejich použití přináší.

Ve třetí kapitole je diskutován současný stav v oblasti modelování a simulace se zaměřením na použití rozličných simulačních nástrojů. Jsou zde také nastíněny současné problémy, se kterými se při návrhu a vývoji systémů vývojáři v současné době potýkají.

Čtvrtá kapitola objasňuje motivaci a definuje cíle práce v návaznosti na popis současného stavu popsaného v předchozí kapitole.

V následující, páté kapitole, se seznámíme s principy architektur založených na službách, popíšeme si jejich základní komponenty, vlastnosti a způsob komunikace. Dále představíme existující architektury používané pro modelování a simulaci a shrneme jejich vlastnosti, výhody a nevýhody.

V šesté kapitole představíme navrhované řešení, tedy otevřenou a modulární simulační architekturu založenou na službách, určenou pro vzdálené modelování a simulaci. Seznámíme se s detailním popisem jednotlivých komponent celé architektury, jejich rozhraním, vlastnostmi a komunikací. Stručně budou probrány i možnosti integrace existujících nástrojů. V rámci celé kapitoly je také představen vyvinutý metajazyk *DEVSML (DEVS Meta Language*, určený pro popis modelů založených na diskrétních událostech. Dále je zde diskutován multiparadigmatický přístup k modelování a simulaci se zaměřením na transformaci modelů, včetně možností, které celá architektura v této oblasti poskytuje. Jako konkrétní příspěvek k tomuto tématu jsou představeny modifikované konečné automaty, které poskytují další možnost popisu modelů založených na diskrétních událostech.

Sedmá kapitola se věnuje případové studii, v níž je představen návrh, tvorba a simulace modelu systému pro řízení dopravy v jednoduché křižovatce. V celé studii je hlavním pojmem simulátor jako služba, který nám názorně představuje možnosti použití celé architektury pro spuštění i několikaúrovňové reflektivní simulace anticipujícího systému.

Osmá kapitola popisuje druhou případovou studii, zaměřenou na popis genetického algoritmu pomocí modelu založeného na diskrétních událostech. Tato případová studie ukazuje model vytvořený pomocí vzdálené simulační služby a používá další důležitou komponentu celé architektury - tzv. prezentační službu.

1 Vymezení oblasti zájmu

Celá práce se věnuje problematice návrhu, modelování a simulaci systémů s důrazem na *inteligentní* a *vyvíjející se systémy*. Současné produkty používané v běžném životě, ať už se jedná o automobil, pračku či výtah, mají právě charakter inteligentních systémů. Takové systémy bývají většinou složité a více či méně komunikují s lidmi, ať už to jsou uživatelé nebo operátoři. Často také bývají do určité míry nebo zcela autonomní, popřípadě jsou i schopné se adaptovat v novém prostředí a reagovat na okolní podněty. Tyto jejich vlastnosti vedou ke složité struktuře a chování, což klade speciální požadavky na jejich návrh a vývoj. Proto je vhodné pro reprezentaci takového systému použít jeho model. Modelování umožňuje navrhovat a vyvíjet i takové systémy, u nichž není možné pracovat s jejich fyzickou reprezentaci, například kvůli bezpečnosti. Simulováním modelu složitého systému pak můžeme lépe poznat jeho vlastnosti a to nás vede k jeho lepšímu pochopení. Kromě toho nám simulace také dává možnost umístit model do speciálního prostředí a sledovat jeho chování v něm. Modelování a simulace tak zcela jistě hrají důležitou roli při návrhu a vývoji heterogenních systémů.

Tradiční softwarové inženýrství se věnuje vývoji systémů, u nichž se po jejich nasazení neuvažuje další vývoj nebo adaptace. Tyto systémy mají pevně definovanou strukturu, svůj počáteční stav a pravidla běhu. Všemu, co se děje po jejich nasazení v reálném prostředí se již nevěnuje tak velká pozornost. Samozřejmě, že takový přístup má své důvody a zcela jistě je vhodný pro většinu případů. Na druhé straně však existují situace, kdy se tento klasický přístup bude jevit jako nevýhodný. Jako příklad může sloužit oblast kritických systémů, kde není možné systém jednoduše zastavit a výměna nebo oprava jeho částí tak musí probíhat za chodu. Mezi takové kritické systémy patří bezpečnostní systémy, či systémy řízení důležitých technologických zařízení jako jsou elektrárny nebo strategické komplexy zajištění obrany státu. Návrh a vývoj takových systémů, které se v průběhu svého životního cyklu vyvíjí a jejichž změny můžeme provádět bez nutnosti jejich zastavení, budou předmětem této práce.

Dále se jeví jako velmi vhodné a názorné chápat tyto systémy a jejich abstraktní reprezentace jako *systémy založené na diskrétních událostech* (viz kapitola 2.3). Tomuto zaměření bude odpovídat i zvolená modelovací platforma, tedy vybrané modelovací formalismy používané pro popis takových systémů a také zvolené simulační nástroje. Budou jimi *Petriho sítě*, formalismus *DEVS* (viz kapitola 2.3) a nástroje určené pro práci s těmito modely (viz kapitola 3.1). V žádném smyslu se nejedná o omezení množiny uvažovaných systémů, poněvadž i systémy, které by se mohly zdát, že nepatří do této skupiny, mohou být vhodným způsobem převedeny - tak například systémy

popsané diferenciálními rovnicemi můžeme převést na systémy s diskrétním časem, a ty mohou být chápány jako speciální případ systémů s diskrétními událostmi. A právě systémy s diskrétními událostmi stojí v popředí zájmu této práce.

2 Návrh a vývoj systémů založený na modelování a simulaci

2.1 Modelování a simulace

Modelování je činnost, která nám slouží k získávání informací o jednom systému pomocí jeho modelu. Systém chápeme jako soubor elementárních částí, označovaných jako prvky systému, které mezi sebou mají určité vazby. Formálněji jej můžeme definovat jako dvojici $S = (U, R)$, kde:

- U je univerzum, představující konečnou množinu prvků systému: $U = \{u_1, u_2, ..., u_N\}$
- $u = (X, Y)$ je prvek systému, kde X je množina vstupních proměnných a Y je množina výstupních proměnných
- $R_{ij} \subseteq Y_i \times X_j$ je propojení prvku u_i s prvkem u_j
- Charakteristika systému R je množina propojení všech prvků $R = \bigcup_{i,j=1}^{N} R_{ij}$

Modelem obvykle rozumíme nějakou entitu M, která pro naše účely zastupuje jinou entitu P (označovanou jako předloha), přičemž požadujeme, aby soudy vyvozené z entity M platily s jistou přijatelnou přesností i o entitě P. Při modelování se pak snažíme využít podobnosti mezi systémy. Proto model chápeme také jako systém, který sice bývá účelově jednodušší než modelovaný systém, avšak pro naše potřeby jej reprezentuje stále s dostatečnou přesností. Můžeme tedy říci, že mezi modelem a modelovaným systémem existuje, resp. je to v našem zájmu, aby existoval, homomorfní vztah (viz obrázek 16).

Máme-li k dispozici jednodušší reprezentaci systému, je pro nás pak výhodné získávat znalosti o tomto systému právě pomocí experimentování s jeho modelem. V tomto okamžiku takový model, s nímž můžeme experimentovat, označujeme jako simulační model a tuto činnost nazýváme simulací. Velmi často však modelování a simulaci používáme nejen pro poznání existujícího systému, ale i pro návrh a vývoj systému budoucího. V tomto případě vytváříme model systému, který ještě neexistuje. Takový model označujeme jako abstraktní a jeho úkolem je popsat pouze důležité části systému a abstrahovat nás tak od méně důležitých vlastností vzhledem k jeho účelu. Mezi abstraktním modelem a simulačním modelem platí vztah izomorfní [25] (viz obrázek 16).

Toto rozlišování mezi abstraktním modelem, simulačním modelem a vlastním systémem se v praxi projevuje nepříznivě tím, že jsme nuceni vytvářet

3 typy systémů, musíme nalézt a pochopit vztahy mezi nimi a umět je mezi sebou jednoduše převést. Mohlo by tedy být pro nás velmi výhodné, kdybychom mohli jeden model používat jak při návrhu, tak i při vývoji a experimentování. Pro jisté typy systémů by dokonce mohl být model i vyvíjeným systémem.

2.2 Inteligentní a vyvíjející se systémy

Abychom o něco lépe porozuměli oblasti zájmu této práce, bude výhodné na tomto místě podrobněji, i když neformálně, definovat *inteligentní* a *vyvíjející se systémy. Inteligentním systémem* pro účely této práce budeme rozumět systém, umístěný v prostředí buď virtuálním či fyzickém, zpravidla obsahující elektroniku, senzory a akční členy s více či méně inteligentním řízením. Takové systémy obvykle komunikují s okolím, tedy se svými uživateli nebo operátory a mají schopnost adaptace, přičemž je u nich často vyžadováno autonomní chování. Inteligentní systém tak často reaguje na podněty ze svého okolí a tomu se přizpůsobuje.

Inteligenci systému budeme hodnotit podle jeho chování. Dle Marvina Minskyho, otce a zakladatele oboru umělé inteligence, totiž *inteligentní systém vykazuje takové chování, které - kdyby jej vykazoval člověk - bychom považovali za projev jeho inteligence.* Tato definice tak zahrnuje mezi inteligentní systémy i systémy poměrně jednoduché, jako je systém řízení osvětlení, ale i systémy podstatně složitější, mezi něž patří robotické a multiagentní systémy.

Po inteligentním systému často vyžadujeme, aby uměl reagovat na podněty a uměl se přizpůsobit změnám v jeho okolí. Systém, disponující touto vlastností, můžeme nazvat jako *adaptivní systém.* Pro úspěšnou adaptaci se může systém pokusit odhadnout svůj budoucí vývoj. Toho využívají především dynamické systémy, které mají znalost nejen o modelu okolí, ale i o svém vlastním modelu. Této znalosti následně využívají právě pro předvídání svého vývoje v budoucnosti. Takové systémy označujeme jako *reflektivní* či *anticipující systémy.* Mezi další výrazné vlastnosti inteligentních systémů patří schopnost *učit se*, což může znamenat třeba rozšiřování znalostní databáze nebo modifikaci rozhodovacích algoritmů či podmínek. *Multiagentní systémy* patří mezi výrazné představitele takových učících se systémů, kde oblast jejich aplikace je velmi široká s uplatněním od senzorových sítí až po systémy získávání znalostí z internetového prostředí. Tyto systémy jsou pak díky učení schopny jistého vývoje. Avšak schopnost učit se není jedinou možností, jak se systém může vyvíjet, mezi další patří třeba změna jeho struktury, ať už v době jeho vývoje či při jeho nasazení. Takové systémy označujeme jako *vyvíjející se* a představit si je můžeme na příkladu robotického hráče fotbalu. Pro něj je charakteristické, že se nejen učí z nových herních situací,

ale umožňuje i změnu svého řídicího systému na přání trenéra, který mění herní strategii svého týmu.

U velkého množství současných systémů skutečně najdeme právě vlastnosti, které je řadí do kategorie inteligentních či vyvíjejících se systémů. Mezi ně patří například: robotický vysavač se svou schopností reagovat na okolní podněty, inteligentní budova, jako adaptivní systém, výrobní systém schopný optimalizovat výrobu, síť senzorů se schopností učit se nebo systém pro řízení dopravy, který představuje anticipující systém.

Systémy s diskrétními událostmi, které budou představeny v následující kapitole 2.3, jsou vhodnou abstrakcí pro modelování nejen počítačových systémů, ale obecně všech dynamických systémů, u nichž má smysl zkoumat a analyzovat jejich chování. Zdá se tedy příhodné použít tohoto způsobu i pro modelování inteligentních systémů, které jsou zcela jistě systémy dynamickými. Proto budou systémy s diskrétními událostmi tvořit hlavní abstraktní reprezentaci používanou v této práci.

2.3 Systémy s diskrétními událostmi

Jak již bylo řečeno, systémy založené na diskrétních událostech se zdají být vhodnou abstrakcí pro modelování dynamických systémů v rámci jejich návrhu a vývoje. Dostatečným důvodem pro tento způsob modelování je to, že jako systémy s diskrétními událostmi můžeme modelovat všechny typy dynamických systémů, u nichž nás zajímá jejich chování, které následně zkoumáme. Dále také je můžeme použít i pro modelování všech způsobů aplikací vyvíjených systémů.

Neformálně lze systém s diskrétními událostmi popsat jako systém se vstupy a výstupy. Na některé vstupy může systém reagovat výstupy, které mohou být pozorovatelné jako události a na některé viditelně reagovat nemusí, přičemž jen přechází mezi svými vnitřními stavy. Výstup může takový systém generovat bez vnější příčiny. Uvnitř systém přechází mezi vnitřními stavy. Změna stavu může proběhnout buď samovolně, pokud uplynul nějaký čas, nebo v reakci na vnější událost, označovanou jako událost na vstupu. Výstup tak vždy závisí na aktuálním stavu a je viditelný jako událost při samovolném přechodu systému do stavu následujícího. Tato neformální definice byla převzata z [8].

V následujících kapitolách si představíme a krátce popíšeme dva formalismy, jako vhodné prostředky pro popis systémů založených na diskrétních událostech.

2.3.1 DEVS formalismus

DEVS (Discrete EVent Systems) formalismus vyvinul prof. B. Zeigler v 80. letech jako prostředek pro univerzální a rigorózní popis diskrétních modelů [38]. Tento formalismus umožňuje popsat systém na dvou úrovních pomocí tzv. *atomických* a *spojovaných komponent*. *Atomický DEVS*, popisuje chování systému a definuje jeho reakce na externí podněty a způsob generování výstupů modelu. *Spojovaný DEVS*, pak popisuje strukturu systému jako síť propojených komponent.

Atomické komponenty Pomocí atomických DEVS komponent jsou popsány elementární části systému. Jejich chování je popsáno jako sekvence deterministických přechodů mezi stavy modelu. Popisují, jakým způsobem bude model reagovat na vstupní podněty a jak budou generovány výstupy modelu. Atomická komponenta je definována jako n-tice:

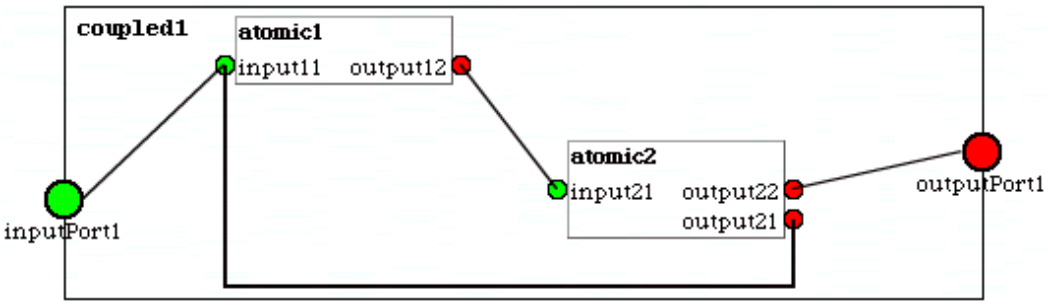
$$atomicDevs =< S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda >$$

- X je množina vstupních událostí
- Y je množina výstupních událostí
- S je množina všech stavů, v nichž se může komponenta nacházet
- $\delta_{int} : S \to S$ je interní přechodová funkce
- $\delta_{ext} : Q \times X \to S$ je externí přechodová funkce, kde $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$
- $ta : S \to \mathbb{R}^{+}_{0,\infty}$ je funkce posunu času
- $\lambda : S \to Y$ je výstupní funkce

Spojované komponenty Spojované DEVS komponenty popisují systém jako síť propojených atomických nebo také spojovaných komponent. Propojení jednotlivých komponent definuje, jak se komponenty vzájemně ovlivňují. Tato úroveň popisu zavádí hierarchickou strukturu modelu. Spojovaná komponenta je definována jako n-tice:

$$coupledDEVS =< X, Y, D, \{M_i \mid i \in D\}, EIC, EOC, IC, select >$$

- X je množina vstupních událostí



Obrázek 1: Jednoduchý DEVS model

- Y je množina výstupních událostí
- D je množina jmen vnitřních komponent
- $\{M_i\}$, kde $i \in D$ je množina vnitřních komponent, ze kterých se spojovaná komponenta skládá. Vnitřní komponentou může být buď atomická nebo spojovaná komponenta.
- $EIC \subseteq X \times \bigcup_{i \in D} X_i$ je množina vstupních propojení (external input couplings), tedy propojení vstupních portů spojované a vnitřní komponenty
- $EOC \subseteq \bigcup_{i \in D} Y_i \times Y$ je množina výstupních propojení (external output couplings), tedy propojení výstupních portů vnitřní a spojované komponenty
- $IC \subseteq \bigcup_{i \in D} Y_i \times \bigcup_{i \in D} X_i$ je množina vnitřních propojení (internal couplings), tedy propojení vstupních a výstupních portů vnitřních komponent
- $select : 2^D \rightarrow D$ je funkce výběru v případě výskytu souběžných událostí

Struktura jednoduchého DEVS modelu, který se skládá z jedné spojované komponenty *coupled1* a dvou propojených atomických komponent uvnitř, je zobrazena pro ilustraci na obrázku 1. Spojovaná komponenta má jeden vstupní a výstupní port, označené jako *inputPort1* a *outputPort1*. Na oba porty jsou napojeny obě atomické komponenty *atomic1* a *atomic2*, které jsou navzájem propojeny přes své vstupní a výstupní porty - *output12-input21* a *input11-output21*.

DEVS jako základní platforma pro modelování Od svého počátku se DEVS stal hojně užívaným formalismem na poli modelování a simulace. Bylo také vyvinuto několik jeho rozšíření a modifikací. Díky tomu se rozšířily i třídy modelů, které je možné pomocí tohoto formalismu popsat. Tato rozšíření zahrnují: *Fuzzy DEVS*, *Real Time DEVS*, *Parallel DEVS*, *Dynamic Structure DEVS*, *Celullar DEVS*, apod. Samotný DEVS je tedy možné považovat za základní formalismus pro návrh a vývoj rozličných typů dynamických systémů, což podporují koncepty a standardy *DEVS bus* nebo *HLA* (viz kapitola 5.2.2), které jej využívají jako základní modelovací platformu.

V [38] je prokázáno, že DEVS může být modifikován nebo přímo použit pro modelování jiných typů systémů než jsou systémy založené na diskrétních událostech. To mohou být buď *systémy s diskrétním časem (Discrete Time Systems)*, což je speciální případ systémů s diskrétními událostmi, nebo *systémy popsané pomocí diferenciálních rovnic (Differential Systems)* - rozdělení a označení je převzato z [38]. Takové systémy lze simulovat buď naprosto věrně, nebo je možné jejich chování simulovat s požadovanou přesností.

V jiných publikacích [9], [10] jsou prezentovány způsoby převodu různých formalismů, jako jsou časované automaty nebo stavové diagramy, do DEVS. Prostřednictvím numerické integrace je také možné do DEVS převést i diferenciální rovnice. Díky tomu lze DEVS chápat jako společný základ pro multiparadigmatické modelování a simulaci (více viz kapitoly 2.4.2 a 6.3).

Samotný DEVS slouží k popisu simulačních modelů na nízké úrovni, kde je možné velmi přesně definovat chování systému. Pro samotné účely ověřování různých vlastností modelu, při verifikaci nebo validaci se však používá jiných prostředků, jako je Petriho síť nebo časované automaty. Ty slouží pro abstraktní popis na vyšší úrovni a jsou tak vhodné pro specifikaci. Je-li však třeba při modelování použít multiparadigmatický přístup 2.4.2, pak DEVS je schopen plnit roli základní modelovací platformy a zajistit tak jednotný přístup ke všem částem modelu i jeho simulaci.

2.3.2 Petriho sítě

Petriho sítě byly poprvé uvedeny v r. 1962, kdy je C. A. Petri zavedl ve své disertační práci. Tvoří třídu diskrétních matematických modelů umožňujících popis řídících toků, prostředků a závislostí uvnitř modelovaných systémů. Vznikly jako zobecnění konečných automatů, které rozšiřují o tzv. přechody a značení. Petriho sítě patří mezi populární, názorné a velmi oblíbené matematické formalismy pro modelování diskrétních a paralelních systémů. Jejich hlavními výhodami jsou konceptuální jednoduchost, srozumitelný grafický zápis, možnosti simulace a formální analýzy. Model je popsán *místy* se stavovou informací v podobě *značek*, dále *přechody* vyjadřujícími změny stavu

a *hranami*, které slouží k propojení míst a přechodů. Petriho síť můžeme považovat za matematicky definovaný stroj, který je možné analyzovat a verifikovat formálními metodami.

Formálně Petriho síť definujeme jako n-tici:

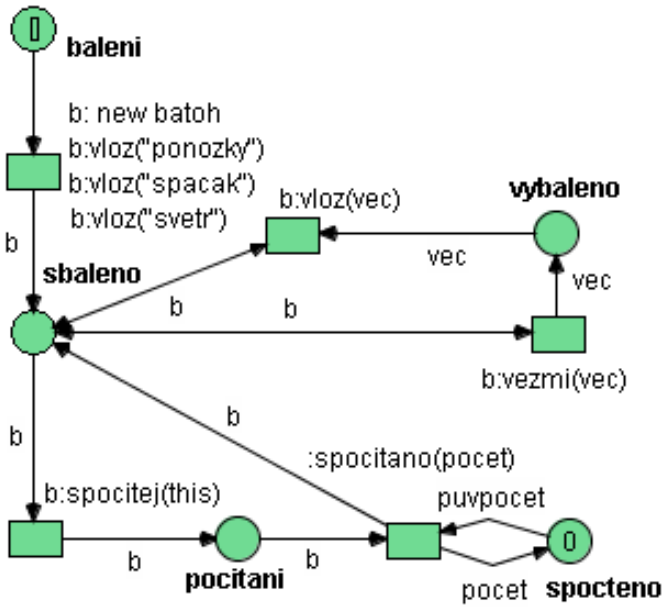
$N = (P, T, F, W, C, M_0)$, kde

- P je konečná množina míst (places)
- T je konečná množina přechodů (transitions), kde $P \cap T = \emptyset$
- F je binární toková relace (flow relation): $F \subseteq (P \times T) \cup (T \times P)$
- $W : F \to \mathbb{N} \setminus \{0\}$ je váha (weight) přechodů
- $C : P \to \mathbb{N} \cup \{\omega\}$ je zobrazení určující kapacitu (capacity) míst. ω je *supremum* množiny $\mathbb{N}$ [62] a slouží k označení neomezené kapacity místa
- $M_0 : P \to \mathbb{N} \cup \{\omega\}$ je počáteční značení, kde $\forall p \in P : M(p) \leq C(p)$

Princip dynamiky Petriho sítě spočívá v provádění přechodů, proto si uveďme, co rozumíme proveditelností přechodu a jak se po jeho provedení změní značení sítě:

- Mějme vstupní množinu $\bullet t$ přechodu t: $\bullet t = \{p | pFt\}$, kde $t \in T$ a $p \in P$
- Obdobně mějme výstupní množinu $t\bullet$ přechodu t: $t\bullet = \{p | tFp\}$, kde $t \in T$ a $p \in P$
- Přechod $t \in T$ budeme považovat za proveditelný, jestliže: $\forall p \in \bullet t : M(p) \geq W(p,t)$ a $\forall p \in t\bullet : M(p) \leq C(p) - W(p,t)$
- Přičemž jeho provedením získáme následné značení $M'(p) = M(p) - W(p,t) + W(t,p)$ a symbolicky toto provedení značíme jako $M[t > M'$

Existuje velké množství různých variant Petriho sítí, kde snahou je zvýšit modelovací schopnosti, úrovně popisu a přiblížení se tak co nejblíže modelovanému systému. Jako příklad můžeme uvést *C/E sítě* [35], *P/T sítě*, či *vysokoúrovňové sítě*, jako jsou např. *barvené sítě* [36].

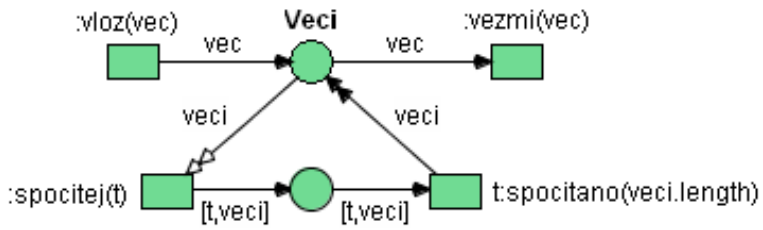


Obrázek 2: Petriho síť s referencemi reprezentující turistu

Petriho sítě s referencemi Pro účely této práce se podrobněji zmíníme o rozšířených Petriho sítích, nazvaných *Petriho sítě s referencemi*, označované anglicky jako *reference nets* [37]. Jsou to klasické Petriho sítě s rozšířeným inskripčním jazykem a možností dynamické instanciace jednotlivých sítí, přičemž každá síť je reprezentována nezávislým objektem a odkaz (reference) na takovou síť může být předáván ve formě značky. Ke komunikaci mezi různými instancemi je možné použít tzv. *synchronních komunikačních kanálů*. Inskripční jazyk je ve skutečnosti jednoduchý programovací jazyk, který umožňuje provádět rozličné akce při provádění přechodů. Jednou z akcí může být právě vytvoření nové instance reprezentující nějakou známou Petriho síť. Pro potřeby simulace je možné do těchto sítí zavést i časování, které určuje, kdy se příslušná značka může z místa odstranit. Dále je také možné použít speciálních typů hran, které umožňují přesunutí více značek najednou (flexible arc), popřípadě umístí všechny značky do proměnné typu pole (clear arc).

Bez jakékoli další formální definice uvedeme příklad Petriho sítě s referencemi a na tomto příkladu ukážeme zmíněná rozšíření. To nám pomůže lépe pochopit složitější Petriho sítě použité v případové studii v kapitole 7.

Na obrázcích 2 a 3 jsou dvě jednoduché Petriho síťě reprezentující turistu a jeho batoh. Turista chystající se na výlet má svůj batoh, který na počátku naplní potřebnými věcmi ($b : vloz(\text{"}svetr\text{"})$). V průběhu výletu pak



Obrázek 3: Petriho síť s referencemi reprezentující turistův batoh

potřebné věci může vybalit ($b : vezmi(vec)$), použít a dát je zpět do batohu ($b : vloz(vec)$). Kdykoliv se také může rozhodnout, že obsah batohu překontroluje, což udělá jednoduchým způsobem tak, že všechny věci přepočítá ($b : spocitej(this)$). U sítě reprezentující turistu vidíme, že hned na začátku je vytvořena nová instance (objekt) reprezentující batoh. V inskripčním jazyce je to vyjádřeno jako $b : new\ batoh$. Odkaz na tuto síť je předáván v proměnné označené jako b, přičemž interakce s příslušným objektem probíhá za pomocí synchronních kanálů $b : vezmi(vec)$ nebo $b : vloz(vec)$. V případě počítání věcí v batohu je interakce obousměrná. Turista nejprve zahájí počítání ($b : spocitej(this)$) a nazpět obdrží počet věcí ($t : spocitano(pocet)$). Při počítání jsou všechny věci v batohu přemístěny z místa $Veci$ a uloženy v proměnné *veci* typu pole. K tomu slouží speciální hrana s dvěmi prázdnými šipkami (clear arc). Po získání počtu věcí ($veci.length$) jsou věci vyčteny z pole a navráceny zpět pomocí hrany, vyznačené na obrázku dvěmi plnými černými šipkami (flexible arc).

I v tomto jednoduchém příkladu vidíme, že inskripční jazyk je velmi podobný programovacímu jazyku a význam konstrukcí je intuitivně a snadno pochopitelný. Speciální vlastnosti sítí s referencemi, jako jsou odkazy, komunikační kanály a zvláštní typy hran společně s inskripčním jazykem, nám tak umožňují zachovat grafovou strukturu relativně jednoduchou a snadno uchopitelnou i pro relativně složité modely.

2.4 Návrh a vývoj inteligentních systémů

Jak již bylo řečeno, u dynamických systémů se pro jejich návrh a vývoj používá modelování a simulace. Hlavním rysem modelování je nahrazení skutečných systémů odpovídajícím (abstraktním) modelem a následná práce či experimentování s tímto modelem místo reálného systému.

Schopnosti *vyvíjet se*, *učit se*, *adaptovat se* či *anticipovat* považujeme za stěžejní vlastnosti inteligentních systémů. Tyto jejich vlastnosti kladou spe-

cifické požadavky na jejich návrh i vývoj a v neposlední řadě i na použité nástroje. Tvorba modelu při vývoji a experimentování s ním, nám umožní lépe pochopit chování a vlastnosti celého systému. V protikladu ke klasickému softwarovému inženýrství jsme zde nuceni zamyslet se nad vývojem systému i po jeho nasazení. To vše vede ke smazání tradičního rozdílu mezi vývojem systému a jeho nasazením v reálném světě. V případě inteligentních systémů je tak vhodné pracovat se systémem po celou dobu jeho životního cyklu stejně, což vede k potřebě nerozlišovat mezi systémem a jeho modelem.

Přístup, kdy se model využívá jako nástroj pro reprezentaci systému po celou dobu návrhu a vývoje je označován anglickým termínem *Model based design*. Jeho hlavním účelem je odstranění složitostí, které vyplývají z konstrukce prototypů při vývoji systémů a možnost ověření jeho důležitých vlastností pomocí formálních metod. Jednou z oblastí, kde je tento postup velmi vhodné použít je právě návrh a vývoj inteligentních systémů.

Takové systémy nemusí být na začátku vývojového cyklu možné fyzicky sestrojit. Může se jednat o koncepčně složité systémy, které jsou náročné na realizaci a na začátku vývoje nemusí být jasná jejich specifikace. Ta buď chybí, nebo je neúplná a vyvíjí se společně s nimi. Bývají to také velmi často systémy, které se velmi mění v průběhu vývoje. Proto je důležité mít možnost experimentovat s nimi a mít možnost jejich úpravy nejen mezi jednotlivými fázemi vývoje, ale také během nich a během vlastní simulace. Zvláště analýza a inspekce stavu systému je důležitá, a to i po následném nasazení v reálném prostředí. Tento přístup bývá nazýván jako *experimentální programování a modelování.*

Velmi výhodné je popsat model na různých úrovních abstrakce a různými formalismy, což může být právě u návrhu inteligentních systémů výhodou, neboť to mohou být systémy heterogenní, u kterých potřebujeme sledovat celou škálu vlastností. Takový postup, kdy je při modelování použito více formalismů, je nazýván jako *multiparadigmatické modelování.*

Další podkapitoly se těmito přístupy detailněji zabývají.

2.4.1 Experimentální programování a modelování

Experimentální programování je jedním z přístupů, který se používá v okamžiku psaní programu, kdy na počátku neexistuje jasná specifikace řešení problému. Tato situace může nastat například, pokud si nejsme jisti tím, jak bude aplikace nebo program ve skutečnosti vypadat nebo nejsme schopni určit nejvhodnější postup pro řešení problému.

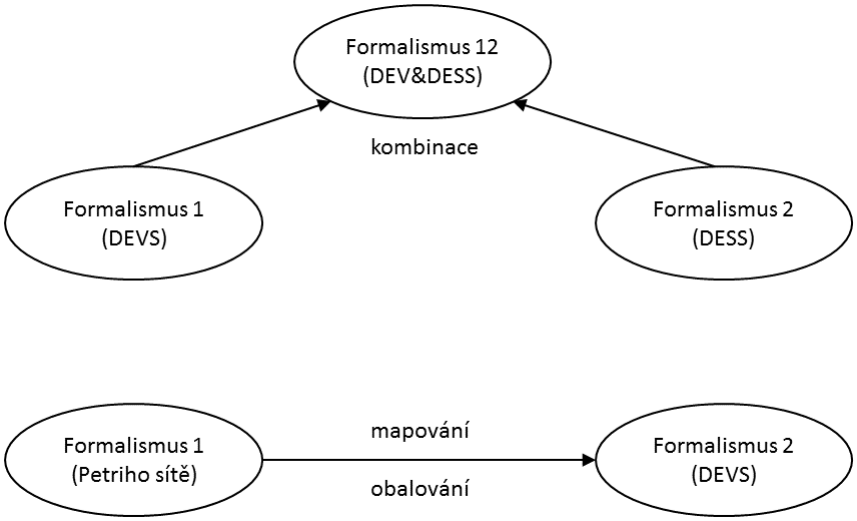
U klasického přístupu v programování jsou charakteristické tyto kroky: program upravujeme (modifikujeme nebo tvoříme) a abychom si ověřili správnost provedených změn, je nutné jej zkompilovat, spustit a otestovat. Pro

další změny je nutné běh programu zastavit, modifikovat jej a znovu provést kompilaci a testování, kterým ověříme jeho chování po provedených změnách. Takováto sekvence kroků je jistě velmi omezující.

Naproti tomu, při experimentálním programování začínáme s jistou částí programu nebo jeho prototypem, který může být velmi odlišný, popřípadě může zpočátku plnit úplně jinou funkci. Ten spustíme a jeho úpravu popřípadě celou tvorbu provádíme za jeho běhu, bez jeho zastavení a bez nutné rekompilace. Změny můžeme ověřit ihned po jejich provedení, bez nutnosti restartovat neboť stále běží. Dokonce nad ním mohou v okamžiku modifikace být prováděny testy ověřující jeho správnost. Výhodou tohoto přístupu je mimo jiné i to, že program může být nasazen již během svého vývoje a bez svého restartování může být upravován. Takový přístup má i řadu dalších výhod, ovšem nutnou podmínkou je existence programovacího jazyka, prostředí a nástrojů, které takovýto přístup umožňují. Používaný jazyk musí být dostatečně dynamický a rozšiřitelný a musí jej dostatečně podporovat vývojová prostředí. Jedním z takovýchto prostředí je Squeak [49] s programovacím jazykem Smalltalk [1], popřípadě Self [2]. Nutno podotknout, že v praxi se většinou používá vhodné kombinace klasického a experimentálního programovacího stylu.

Experimentální modelování je pak totožný přístup, kdy program je nahrazen modelem nebo model skutečně představuje. Tento přístup umožňuje tvorbu a úpravu modelu během jeho simulace, což je vlastnost, kterou simulační nástroje ve velké většině nedisponují. Při vývoji inteligentních systémů s nimi experimentovat potřebujeme. Příkladem může být evoluční algoritmus, jehož funkčnost, správnost a jiné vlastnosti zkoumáme při jeho tvorbě. S algoritmem experimentujeme v tom smyslu, že měníme jeho různé parametry a zjišťujeme, jak jeho chování změna ovlivnila. Těmito parametry mohou být pravděpodobnost mutace, křížení, počet jedinců v jedné populaci, ale i reprezentace jednotlivých chromozómů. V tomto případě velmi oceníme, pokud nám prostředí umožňuje změnu parametrů přímo za běhu samotného algoritmu. Příklad genetického algoritmu, který byl tímto způsobem vytvořen a odladěn v prostředí SmallDEVS [12], [13], [14] (viz kapitola 3.1.3) je uveden v kapitole 8. Toto prostředí poskytuje prostředky pro experimentální modelování a dovoluje změnu modelu za jeho běhu.

Na závěr je třeba poznamenat, že možnosti, které nám experimentální programování nabízí, mohou být ještě dále rozšířeny, když model sloužící pro návrh systému budeme považovat za systém samotný. Tento model pak může být v praxi nasazen a interpretován. Takový postup se nazývá jako *model kontinuity*.



Obrázek 4: Multiparadigmatické modelování - princip mapování a obalování

2.4.2 Multiparadigmatický přístup

Multiparadigmatický přístup se vyznačuje použitím několika různých formalismů při tvorbě modelů. Tyto modely pak označujeme jako multimodely. I když zcela jistě použití jediného formalismu, znamená řadu výhod, mezi něž patří jednotný přístup k modelům, jejich návrhu, simulacím a omezení množiny použitých nástrojů, v praxi vždy nemusíme být schopni nalézt jednu formu abstrakce pro daný systém a multiparadigmatické modelování pro nás pak znamená řešení. Kupříkladu část modelu jaderné elektrárny je možné popsat pomocí diferenciálních rovnic, avšak pro její kontrolní prvky, ovládající její činnost, je vhodné použít systémy založené na diskrétních událostech.

Abychom mohli využít výhody jediného formalismu, pak je potřeba různé formalismy použité při tvorbě modelů a pro jejich simulaci nějak sjednotit. V praxi existují dva hlavní přístupy. Jedním z nich je jejich *kombinace*, druhý typ reprezentuje *mapování* (mapping) a *obalování* (wrapping). Tyto přístupy jsou znázorněny na obrázku 4. V případě kombinace je nutný vznik nového formalismu, který kombinuje jiné, z nichž vznikl. Takovým případem je DEV&DESS [38]. V případě mapování nebo obalování žádný nový formalismus nevzniká, ale na jeden formalismus mapujeme nebo jím obalujeme druhý.

V obou případech můžeme vidět hlavní výhodu multiparadigmatického přístupu, čímž je možnost využití více formalismů pro popis modelu, bez komplikace vlastní simulace, kde je již použit formalismus jen jeden. V pří-

padě mnoha reálných systémů nám tak tento přístup může značně ulehčit práci.

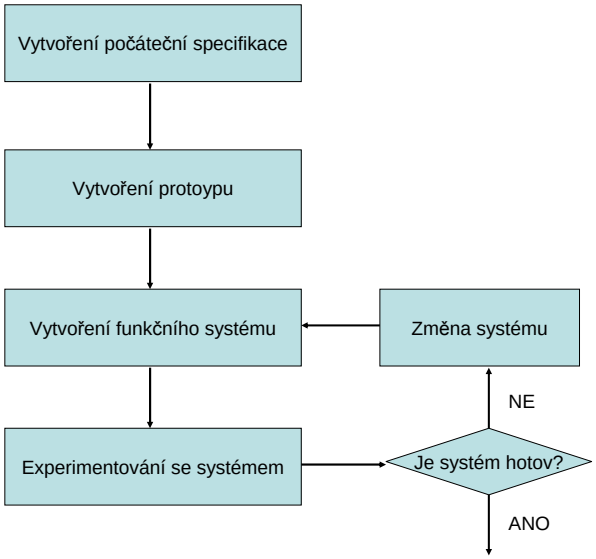
2.4.3 Interaktivní vývoj

Inteligentní, adaptivní a vyvíjející se systémy zahrnují systémy, které nemají jasnou specifikaci. Je to dáno tím, že tyto systémy často bývají složité a je tak velmi těžké vytvořit jednoznačnou specifikaci, která by je snadno popsala. Takové systémy se vyvíjejí nejlépe za použití techniky rychlého prototypování, tedy inkrementálně v krátkých fázích, které se mohou opakovat. Způsob, jakým jsou změny v systému při rychlém prototypování prováděny, není nikde určen. Závisí totiž na programovacím jazyce, popřípadě na použitých vývojových nástrojích. Nejprve vytvoříme počáteční specifikaci, která si neklade za cíl popsat každou vlastnost systému. Podle ní je vytvořen prototyp systému. Zde je možné dodefinovat další vlastnosti systému a požadavky na něj. Jakmile máme hotovou funkční verzi systému, vhodnou ke spuštění a experimentování, klademe si otázku, jestli odpovídá počáteční specifikaci a našim požadavkům, tedy těm, které jsme si stanovili na počátku a také těm, které vyvstaly během vývoje. Pokud dospějeme k názoru, že ano, pak je vývoj hotov, pokud však nikoli, pozměníme vyvíjený systém a znovu vyhodnocujeme, jestli je hotov. Názorněji je tento způsob vývoje zobrazen na obrázku 5.

Důležité je, aby jednotlivé vývojové cykly byly co nejkratší. Měly by tak reprezentovat jen takové změny systému, které mohou být provedeny v krátkém časovém horizontu. Důraz klademe na experimentování se systémem a nepovažujeme za žádoucí, abychom museli podstatným způsobem změnit způsob, jakým vyhodnocujeme, jestli je systém hotov. Technika označovaná jako *experimentální programování a modelování* se zdá být velmi vhodná k použití pro interaktivní vývoj. Umožňuje totiž provádět změny za běhu systému a okamžitě sledovat jejich efekt. Odstraňuje se tím nutnost systém rekompilovat a restartovat po provedených změnách. Samozřejmě toto je možné jen pokud máme k dispozici vhodné jazyky a nástroje.

2.4.4 Model kontinuity

Modelování a simulace z pohledu softwarového inženýrství pracuje se systémem na vyšší úrovni abstrakce, kterou splňuje model systému. Model je zde intuitivní a snadno pochopitelnou reprezentací nějakého systému. Tvorba takového modelu probíhá za pomocí nástrojů, které nám umožňují vytvořit platformně nezávislý model. Ten pak je nutné transformovat na reprezentaci, která je již určena pro konkrétní platformu a teprve takový model pak mů-



Obrázek 5: Inkrementální vývoj systémů

žeme snadno podrobit další analýze a experimentovat s ním. Po dokončení vývoje pak je nutné z modelu vygenerovat spustitelný kód, čímž může být třeba program, který vyvíjený systém řídí.

Princip postupu *modelu kontinuity* spočívá ve výrazném zjednodušení výše uvedeného postupu. Hlavní charakteristikou tohoto postupu je snaha o použití jedné specifikace modelu pokud možno ve všech stádiích vývoje systému tak, aby se model stal modelovaným systémem. Hlavní výhodou je odstranění nutnosti transformovat model na jinou reprezentaci, která může být nutná pro jeho spuštění, verifikaci či modifikaci. Není také potřeba generovat výsledný kód na konci vývoje a model tak může být použit jako vyvíjený systém. Protože takovým modelem často bývají programy řídicích systémů v reálném světě [33], [34] a jsou přítomné po celou dobu existence systému, používáme zde pojmu *živý model*.

2.4.5 Reflektivní simulace

Simulace systémů, které obsahují simulující prvky se nazývají *vnořené simulace*. Rozlišujeme pak vnější a vnitřní/vnořený model, přičemž simulace vnitřního modelu probíhá v rámci simulace vnějšího modelu. Speciálním případem vnořené simulace je tzv. *reflektivní simulace*, kdy vnořený model je

schopen simulovat svůj mateřský systém. Tento přístup je možné použít například pro optimalizaci chodu systému. Můžeme si zajisté představit systém, jehož jedna část je určena ke krátkodobým předpovědím. S použitím vnořené simulace tak bude model přesnější a bude lépe reprezentovat systém. Z tohoto důvodu by měl také dávat lepší výsledky.

Důležité je si uvědomit, že vnořené simulace nemusí probíhat ve stejném simulačním čase jako simulace vnějšího systému a také si ukážeme, že nemusí probíhat ani na stejném stroji, ale mohou být řízeny vzdáleně. V rámci této práce budeme s výhodou využívat reflektivní simulaci pro předpovědi budoucího vývoje systému. V rámci případové studie v kapitole 7 si ukážeme systém, který bude schopen vzdáleně spouštět sadu simulací vlastního modelu. Jejich výsledky pak použije pro odhad svého budoucího vývoje. Na základě tohoto odhadu pak zvolí variantu své vlastní rekonfigurace.

Reflektivní simulace má speciální nároky na simulační prostředí. Vyžaduje důsledné oddělení vnitřního a vnějšího modelu tak, aby se nemohly navzájem ovlivňovat, ale na druhé straně je třeba zajistit i jejich správnou komunikaci. Dále také nutné umožnit správnou inicializaci vnořené simulace. V následujících kapitolách uvidíme, jak navržená simulační architektura řeší přirozeně tento problém komunikace a správnou inicializaci v případové studii.

3 Současný stav

Společně s rostoucí oblibou modelování a simulace, jako oboru, který má mimořádný význam pro tvorbu a vývoj rozličných typů systémů, vzrůstá také potřeba jej přizpůsobovat uživatelům a nabídnout jim pohodlnější a propracovanější přístupy a metodiky. Uživatelé či vývojáři mají k dispozici široké spektrum nástrojů určených pro návrh systému a jeho následnou simulaci. Každý z nástrojů má určité vlastnosti, výhody a nevýhody a je tak většinou určen pro konkrétní aplikaci. Chceme-li výhody různých nástrojů zkombinovat a využít, například při návrhu složitějších heterogenních systémů, stojíme takřka před neřešitelným úkolem, jelikož tyto nástroje podporují různé modelovací a programovací jazyky, různé formy zápisu modelu, nemáme zajištěnu přenositelnost modelů ani simulací, simulační výstup je nekompatibilní, apod. Rozdílné nástroje spolu navzájem nekomunikují nebo je komunikace omezena jen na malou množinu aplikací, které vyžadují totožný programový model a jazyk a vyžadují častou synchronizaci (viz např. *Remote Method Invocation* v kapitole 7.3).

Rozhodneme-li se i přes tyto všechny nedostatky používat nějakou množinu nástrojů, zjišťujeme, že je nutné tyto nástroje nainstalovat na lokální infrastrukturu, udržovat je a aktualizovat. Musíme se tedy seznámit s programovou dokumentací, nastudovat specifické ovládání programů a to i přesto, že tyto nástroje plní často velmi podobnou funkci. Pak teprve stojíme před problémy spojenými s nekompatibilitou nástrojů a jejich produktů (části modelů, simulace, výstupy). Kompatibilitu jsme pak nuceni zajistit naším vlastním řešením, což může znamenat manuální úpravu produktu v jednoduchém případě (např. odstranění uvozovek, které neumí interpret textového výstupu načíst) nebo v případě složitějším třeba i implementaci externí aplikace (např. automatický převodník modelů z jednoho formátu na jiný). Stejné problémy však neřešíme jen my, ale i ostatní týmy na světě a možná i oni mají své vlastní řešení připravené pro jejich množinu aplikací, které bychom mohli využít.

Způsob, jakým se simulační nástroje běžně používají je navíc značně limitující. Velmi často totiž tyto nástroje běží na lokální infrastruktuře (často to bývá náš vlastní počítač). Důsledkem je nejen jejich náročná údržba, ale ani jejich funkcionalita není dostupná ostatním členům týmu. Také škálovatelnost je tímto velmi omezená. Nástroje nemají buď žádné, nebo mají rozdílné aplikační a programové rozhraní, univerzální formáty pro výměnu dat neexistují a komunikační kanály je složité nastavit - prakticky vše potřebné je nutné nahradit vlastním řešením. Stejně tak neexistuje žádné centrální úložiště, kde bychom mohli modelovací a simulační artefakty, čímž rozumíme hlavně, modely a části modelů, sdílet a nabídnout ostatním. I kdyby však

takové úložiště existovalo, tak nemáme (nebo do nedávné doby nebyly - více viz kapitola 6.2) k dispozici dostatečně obecné standardy pro jejich popis.

Nástroje používané v oblasti modelování a simulace však mají, i přes svou rozmanitost, jistou výhodu oproti obecným programům, které jsou velmi různorodé. Touto výhodou je společná doména a účel. Díky tomu by mělo být možné vytvořit jednotnou a rozšiřitelnou modelovací platformu, která sjednotí komunikaci, umožní integraci a nastaví její pravidla.

V následující kapitole si uvedeme konkrétní kritiku současného stavu na množině existujících a používaných simulačních nástrojů. Nejprve nástroje stručně popíšeme a poté budeme diskutovat jejich výhody a nevýhody.

3.1 Simulační nástroje

Jelikož se v této práci zaměřujeme na diskrétní systémy (viz kapitola 2.3), odpovídá tomu i spektrum diskutovaných modelovacích a simulačních nástrojů. Jedná se o nástroje podporující návrh a simulaci modelů založených na DEVSu či Petriho sítích. Množina popsaných nástrojů není ani zdaleka úplná, jedná se spíše o výběr oblíbených a referenčních nástrojů a také o nástroje, se kterými má autor práce nějakou osobní zkušenost. Z tohoto důvodu se na nich rozhodl demonstrovat některé aspekty této práce.

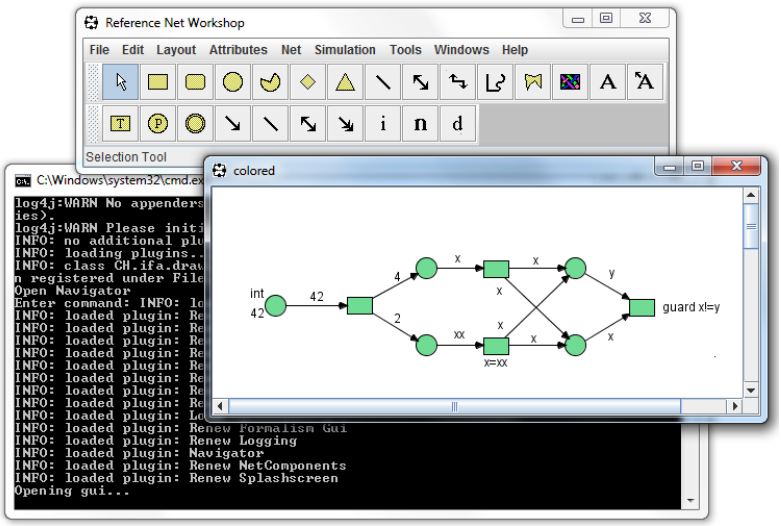
3.1.1 Renew

Simulační nástroj *Renew* [4], [5] umožňuje simulaci modelů popsaných rozšířenými Petriho sítěmi, konkrétně *Petriho sítěmi s referencemi* [3] - viz kapitola 2.3.2. Renew je editor a simulátor, implementovaný v jazyce Java, který umožňuje návrh a vývoj modelu systému založený na simulaci. Jeho příjemné grafické uživatelské prostředí umožňuje dobrou vizualizaci modelu a společně s ovládacími prvky také dobrou kontrolu nad spuštěnou simulací. Petriho sítě s referencemi nám pak umožňují vytvářet modely s vnořenými sítěmi, které jsou při simulaci dynamicky instanciovány. Prostředí nástroje Renew je pro ilustraci ukázáno na obrázku 6.

Nástroj Renew nám pro účely této práce poskytl vhodný prostředek pro dosažení vnořené a reflektivní simulace v rámci jedné z případových studií (viz kapitola 7).

3.1.2 PNTalk

PNtalk [47], [48] je modelovací a simulační prostředí určené pro tvorbu modelů založených na objektově orientovaných Petriho sítích (OOPN), které kombinují vlastnosti Petriho sítí s objektově-orientovaným návrhem systémů.



Obrázek 6: Prostředí nástroje Renew

Celý nástroj je možné považovat ze experimentální nástroj určený především pro výzkum a výuku, který podporuje návrhové metody zaměřené na vývoj adaptivních systémů. Nástroj existuje ve dvou verzích: *PNTalk* a *PNtalk 2.0*, přičemž verze 2.0 je nová implementace a rozšíření původního PNTalku.

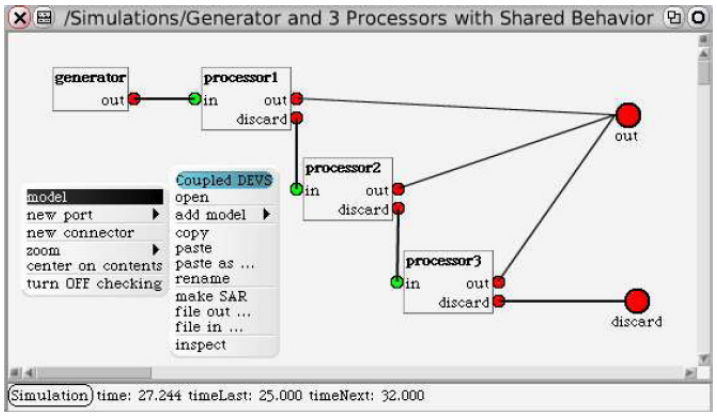
3.1.3 SmallDEVS

SmallDEVS představuje jednu z implementací DEVS formalismu [12], [13], která je vyvíjena na Fakultě informačních technologií v Brně. Je to experimentální simulační nástroj, vytvořený v prostředí Squeak a napsaný v jazyce Smalltalk, jehož předností velmi obratně využívá. Celý nástroj umožňuje tvorbu a simulaci modelů popsaných pomocí DEVS formalismu. Obsahuje příjemné grafické uživatelské rozhraní, které může uživatel využít pro experimentální programování a modelování. Prostředky pro experimentování s modelem jsou velmi dobře implementovány a vyvíjený model je tak možné upravovat i v průběhu své vlastní simulace. Na obrázku 7 je snímek z prostředí SmallDEVS, na němž je předvedena část jeho uživatelského rozhraní.

Prostředí SmallDEVS bylo pro účely této práce zvoleno za prostředí pro implementaci jedné z případových studií (viz kapitola 8).

3.1.4 DEVSJava

Modelovací a simulační nástroj DEVSJava [11] je považován za referenční implementaci DEVS formalismu. Umožňuje návrh a simulaci DEVS modelů. Je



Obrázek 7: Prostředí SmallDEVS

vyvíjen na Arizona State University v centru ACIMS (Arizona Center for Integrative Simulation). Mezi klíčové vlastnosti poslední verze tohoto nástroje (3.0), implementovaného v jazyce Java, patří možnost modifikovat strukturu modelu v průběhu jeho simulace, což autoři nástroje označují jako *variable-structure modeling*. K dispozici je také celá řada rozšíření, které umožňují modelování systémů popsaných diferenciálními rovnicemi nebo použití celulárních automatů. Nástroj také podporuje *model kontinuity*, což znamená, že tentýž model může být bez úprav simulován, verifikován a také spuštěn v cílovém prostředí.

3.1.5 PythonDEVS

Simulační nástroj PythonDEVS [16] byl původně vyvíjen jako nástroj pro výuku a také jako prostředek k výzkumu jednotlivých variant DEVS formalismu. Umožňuje tvorbu a následnou simulaci modelů. Postupně byl rozšířen a podporuje tak varianty zahrnující *real-time DEVS* nebo *paralelní DEVS*. Pro specifikaci chování modelu je nutné použít jazyka Python, se kterým je tak vytvořený model velmi úzce propojen a pro jeho simulaci je také nutný interpret tohoto jazyka.

3.1.6 DEVS-C++

DEVS-C++ [15] je modulární hierarchické simulační prostředí, které umožňuje simulaci modelů založených na paralelní variantě DEVS formalismu. Tato implementace simulačního nástroje využívá možností objektově orientovaného jazyka C++ pro specifikaci modelu a jeho chování, přičemž je možné použít tzv. kontejnerových tříd a dědičnosti mezi nimi. Nástroj obsahuje sadu

předdefinovaných modelů, které lze dále rozšiřovat, a tak je možné jednoduše vytvořit modely komplexních systémů.

3.2 Nevýhody simulačních nástrojů

Popsané simulační nástroje mají rozdílné vlastnosti - některé podporují vizuální modelování (např. SmallDEVS, PNTalk), jiné umožňují pouze jejich textovou definici ve stylu podobném klasickému programování (DEVS-C++). Všechny nástroje podporují tvorbu simulačních modelů. Tyto modely jsou však úzce spojeny s nástrojem, ve kterém byly vytvořené. Toto je patrné především u DEVS nástrojů, kde je chování modelu často specifikováno prostředky příslušného programovacího jazyka - například u nástroje DEVSJava to je jazyk Java, u PythonDEVSu je to Python, u DEVS-C++ je to jazyk C++. Uživatel tak sice rád přistoupí na pohodlnou definici chování modelu pomocí jemu blízkého programovacího jazyka, ovšem za cenu velkého rizika tvorby nevalidních DEVS modelů, které nebudou odpovídat definici formalismu. Kvůli tomu pak nemusí být možné model verifikovat nebo analyzovat standardními prostředky. Z tohoto důvodu také není možné vytvořit znovupoužitelné modely ani knihovny takových modelů, protože simulační model vytvořený např. v prostředí DEVSJava není možné použít v prostředí PythonDEVS.

Některé nástroje, jako je např. Renew řeší nedostatky v oblasti přenositelnosti modelů pomocí podpory existujícího dostatečně obecného výrazového prostředku pro popis modelů, jakým je např. jazyk PNML pro Petriho sítě (více viz kapitola 6.2).

Většina uvedených nástrojů také není vzdáleně přístupná. Pro jejich použití, je nutné je nainstalovat na lokální stroj, přičemž instalační procedura obsahuje manuální kroky a vyžaduje několik prerekvizit, mezi něž patří virtuální stroj pro Java aplikace, interpret jazyka Python, sada potřebných knihoven, apod. Většina nástrojů také přímo nepodporuje vzdálenou ani distribuovanou simulaci a neumožňuje ani komunikaci mezi svými instalacemi. Některé nástroje tyto nedostatky řeší použitím technologií *R*MI (Remote Method Invocation) (viz kapitola 7.3) či *CORBA (Common Object Request Broke Architecture)* [59], ty však mají svá omezení. Tím je nutnost časté synchronizace nebo potřeba prostředníka pro komunikaci, jako je *Object Request Broker*. Tyto technologie také nejsou přímo zaměřeny na implementaci webových služeb (viz kapitola 5.1).

4 Motivace a cíle práce

Současné problémy a nedostatky v oblasti modelování a simulace jsme si shrnuli v předchozí kapitole. S ohledem na jejich řešení či zmírnění tato práce navrhuje otevřenou a modulární architekturu založenou na *službách*, určenou pro integraci existujících simulačních nástrojů (více o *službách* a *webových službách* je uvedeno v kapitole 5). Společná doména a účel simulačních nástrojů by měl usnadnit vytvoření jednotné modelovací platformy, přičemž hlavní snahou je umožnit:

- Použití rozličných modelovacích a simulačních nástrojů a aplikací a zveřejnění jejich funkcionality
- Sjednocení komunikace mezi simulačními nástroji
- Použití různých modelovacích formalismů
- Sdílení modelů
- Snadnou a vzdálenou manipulaci s modely a simulacemi
- Použití speciálních M&S technik, mezi něž patří: interaktivní, multiparadigmatické a experimentální programování a modelování.
- Snadné rozšíření a škálování architektury pro různé aplikace

Stěžejním cílem bude také návrh nového přístupu k modelování a simulaci, kdy nejdůležitějším prvkem nebude nástroj, ale (vzdálená) služba, s přesně definovaným rozhraním, pravidly komunikace a funkcionalitou. Budeme se také snažit navrhnout a definovat důležité služby pro modelování a simulaci a představit výhody tohoto přístupu, kterými jsou: otevřenost, modulárnost, rozšiřitelnost, snadné (vzdálené) použití a jednoduchá údržba.

S ohledem na zaměření této práce na systémy založené na diskrétních událostech se budeme snažit, aby hlavními prostředky pro modelování byly formalismy DEVS a Petriho sítě. Oba budou sloužit jako základní platforma pro různé modelovací techniky. Návrhem modifikovaných konečných automatů pro popis modelů zavedeme alternativní možnost popisu DEVS modelů. Prozkoumáním možností převodu modelů, bychom měli docílit znovupoužitelných modelů, které bude možné migrovat mezi simulačními prostředími. Pro tyto účely bude nutné nejen sjednotit komunikaci mezi různými nástroji, ale také vyvinout či určit jednotný výrazový prostředek pro popis modelů.

Celá práce si, vzhledem k rozsáhlosti probírané tématiky, neklade za cíl věnovat se všem probíraným oblastem do detailu, ale pouze v určitém omezeném rozsahu se bude snažit ukázat, že zvolený přístup je správný a vhodně

zvolenou případovou studií demonstrovat jeho možnosti. Snahou také bude nastínit směr dalšího výzkumu, vývoje a rozšíření simulační architektury.

Vytvoření simulační architektury založené na službách je plně v souladu se snahou prof. B.Zeiglera, tvůrce DEVS formalismu [6], o to aby model byl nezávislý na simulátoru, což přináší řadu výhod:

- model vyjádřený v jednom formalismu může být spuštěn různými simulátory
- simulační algoritmy mohou být formálně definovány a jejich správnost formálně dokázána
- zdroje potřebné k simulaci modelu mohou být měřítkem složitosti modelu

Skutečnost, že DEVS formalismus a Petriho sítě v simulační architektuře budou hrát roli jednotného výrazového prostředku, znamená, že získáme možnost nízkoúrovňového popisu chování modelů a také jejich lehce uchopitelnou reprezentaci. Díky tomu budeme v souladu se současným směrem v oblasti modelování a simulace. Tím je použití modelu jako vyvíjeného systému a jeho interaktivní vývoj založený na experimentálním programování.

5 Architektury založené na službách

Architektury orientované na služby (anglicky Service-oriented architectures) představují jednu z technik softwarového inženýrství, která využívá rozdělení složitého systému na komponenty. Komponentami jsou v tomto případě nezávislé a navzájem spolupracující služby. Služba je v rámci systému něco, co poskytuje přidanou hodnotu, může to být třeba nějaký dílčí úkol nebo složitý výpočet. Každá ze služeb má dobře definované rozhraní a poskytuje předem definované operace. Velmi důležitým rysem služeb je také to, že jejich využití je nezávislé na použité platformě a na implementačním jazyku. Služby tak tvoří část systému, která nemá buď žádné, nebo jen slabé vazby na své okolí. Komunikace mezi službami pak probíhá podle předem daného standardu a umožňuje ji přesně definované rozhraní služby, kterému rozumí i druhá strana (viz kapitola Webové služby).

Základní vlastnosti služeb jako základních komponent architektur založených na službách můžeme shrnout v následujícím seznamu:

1. *Jasný kontrakt* - Každá služba má přesně definovaný kontrakt, který definuje její technické rozhraní a formát dat, který služba přijímá či odesílá. Tímto způsobem svému okolí oznamuje, co poskytuje za služby.

2. *Nezávislost* - Použití služby je nezávislé na použité platformě nebo implementačním jazyce - viz *Princip abstrakce*. Nezávislost služby je nutnou podmínkou pro její znovupoužitelnost.

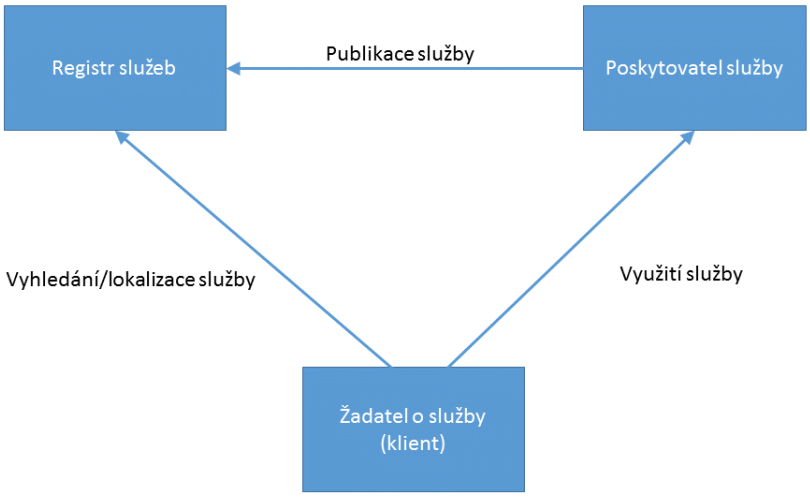
3. *Znovupoužitelnost* - Službu jako jednoduchou komponentu musí být možné použít i v rámci odlišného systému či aplikace.

4. *Bezstavovost* - Služba je bezstavová komponenta, která je nezávislá na kontextu a stavu jiných služeb či komponent. Tato vlastnost je důležitá pro dosažení znovupoužitelnosti.

5. *Jednoznačná identifikace* - Každá služba musí být jednoznačně identifikovatelná v rámci celé architektury.

6. *Princip abstrakce* - Implementační detaily musí být skryty, což umožňuje volné vazby na své okolí. Princip abstrakce zvyšuje granularitu systému a usnadňuje jeho administraci a úpravu.

Na obrázku 8 vidíme princip komunikace v architektuře orientované na služby. Na jedné straně je *poskytovatel služby (service provider)* a na straně druhé *klient/konzument (service consumer)*. Poskytovatelem služby je entita, která nabízí svému okolí nějaké předem definované služby. Klientem je pak



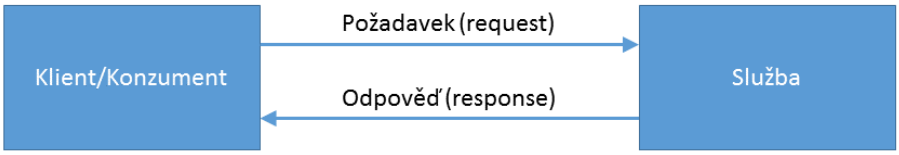
Obrázek 8: Architektura založená na službách

entita, která tyto služby využívá. Je-li potřeba, pak i poskytovatel se může stát konzumentem a využívat tak jinou vzdálenou službu. Důležité je, že každá služba má své přesně definované rozhraní. Implementace služby se může změnit, ale rozhraní zůstává a klient tak nepozná žádnou změnu. Pro klienta je důležité vědět, jaké služby daný poskytovatel nabízí a jak s ním může komunikovat, aby mohl služby využít, proto potřebuje mít k dispozici kompletní popis poskytovatele.

Komunikace probíhá na principu vzdáleného volání procedur (RPC) a je typu *klient-server*. Základem je dvojice zpráv: *požadavek (request)* generovaný klientem a *odpověď (response)* odesílaná poskytovatelem služby.

5.1 Webové služby

Jedním z prostředků realizace architektur orientovaných na služby jsou *webové služby*. Webová služba v tomto případě tvoří základní komponentu celé architektury a poskytuje své služby po síti či přes internet. Komunikace využívá nejčastěji protokol HTTP (Hypertext Transfer Protocol), pomocí kterého se přenášejí dvojice zpráv požadavek/odpověď (request/response). Pro popis zpráv a jejich kódování se pak nejčastěji používá *SOAP (Sime Object Access Protocol)* [52], který je založen na jazyce *XML (eXtensible Markup Language)*. V XML jazyce je popsané i rozhraní služeb, konkrétně se jedná o speciální jazyk *WSDL (Web Service Description Language)* [53], který definuje veřejné operace poskytované službou a datové struktury pro výměnu dat.



Obrázek 9: Komunikace v architektuře založené na webových službách

Princip komunikace v architektuře používající webové služby je znázorněn na obrázku 9. Poskytovatel služeb publikuje služby ve veřejném registru, kde je uložena definice služby a její umístění či adresa - nejčastěji URL (Uniform Resource Locator). Žadatel pak může službu podle různých kritérií v registru služeb vyhledat a využít. Pokud v architektuře není registr služeb k dispozici, pak je nutné, aby žadatel znal adresu služby předem.

Kromě popsaného protokolu SOAP, existuje ještě jeho alternativa, nazývaná jako *REST (REpresentational State Transfer)*. Ta jazyk XML nepoužívá, ale je založená na URL. Je sice jednodušší na použití, ale není přímo určena pro distribuovaná prostředí a vyžaduje HTTP protokol jako transportační vrstvu. Nemá také tak velikou podporu mezi nástroji pro tvorbu webových služeb.

5.2 Existující architektury pro modelování a simulaci

Mezi existující architektury pro vzdálenou a distribuovanou simulaci patří *Microsoft Robotic Studio*, *Simulator Integration Platform (SINPL)* [40], [30], *DEVSBus* a *High Level Architecture (HLA)* [31] a *Multi-Simulation Interface (MSI)* [41].

5.2.1 Microsoft Robotics Developer Studio

Microsoft Robotics Developer Studio [39] reprezentuje architekturu pro kontrolu a simulaci robotických aplikací. Je tvořené simulačním prostředím, vyvinutým pro operační systém Windows. Určené je, jak pro zájmové skupiny, tak i pro akademické pracovníky a komerční vývojáře, kterým umožňuje vytvořit robotické aplikace pro různé hardwarové platformy. Celé prostředí je tvořeno modelovacími a simulačními nástroji a umožňuje jednoduchý přístup k modelu robota a jeho senzorům či čidlům. Interakce s roboty může probíhat přes webový prohlížeč, nebo je možné použít aplikaci pro operační systém Microsoft Windows.Tímto způsobem je možné se jednoduše dotázat na stav modelovaného robota. Vývojáři mohou používat vizuální programovací jazyk, kdy umístěním a propojením jednoduchých bloků, mohou vytvořit celé

robotické aplikace. Toto prostředí také umožňuje simulaci realistických 3D modelů.

5.2.2 DEVS bus a HLA

Mnoho přístupů a metodologií své použití zakládá na DEVS formalismu, patří mezi ně například *DEVS bus*, koncept představený v roce 1995, který využívá DEVS modely jako obálku pro široké spektrum jiných modelů, které tak mohou spolupracovat v síťové simulaci. Takovou simulaci umožňuje *HLA (High Level Architecture)*. Je to standard, vyhlášený Ministerstvem obrany Spojených Států, umožňující interoperabilitu modelů v distribuované simulaci.

HLA umožňuje integraci autonomních simulátorů v jeden distribuovaný simulační systém. Simulace jsou definovány v rámci federací, kde jedna federace je simulační systém, který se skládá ze dvou a více simulátorů, komunikujících přes Run-Time Infrastructure (RTI). HLA je architekturou, která může být nasazena v odlišných prostředích, jako jsou vysoce náročné komerční simulace nebo vojenské simulátory.

V sítích, orientovaných na diskrétní události, mezi něž se řadí HLA, jsou komponentami DEVS systémy, které spolu komunikují přes své vstupní a výstupní rozhraní. Tyto komponenty nemají jinou další možnost, jak získat nebo ovlivnit stav jiné komponenty nebo její časování, než pomocí tohoto rozhraní. Veškerá komunikace probíhá pouze výměnou zpráv. Konkrétněji to znamená, že události generované jednou komponentou, jsou přes její výstupní porty poslány na vstupní porty jiných komponent a tím vznikne externí událost, která způsobí změnu stavu v ovlivněných komponentách.

5.2.3 Multi-Simulation Interface

Protože architektura HLA je pro použití složitá, vznikla potřeba vytvořit její jednodušší variantu. Z toho důvodu vznikl open-source engine Multi-Simulation Interface (MSI), který slouží ke stejným účelům jako HLA, ale je mnohem jednodušší a snazší na používání. MSI jednoduše spojuje různé simulace pomocí časové a datové synchronizace. Výhodami oproti HLA jsou jednoduchost a rychlost. Více o MSI je možné najít v [41].

5.2.4 Simulator Integration Platform

Simulator Integration Platform (SINPL) byla představená v roce 2004 a jejími autory jsou Alberto Coen-Porsini, Ignazio Gallo a Antonella Zanzi z University degli Studi dell'Insubria v Itálii. SINPL je open-source platforma

umožňující integraci heterogenních simulací v distribuovaném prostředí a vývoj webových simulátorů a simulačních prostředí.

Hlavním prvkem v celé architektuře, který se stará o běh simulace je *Distributed Simulation Controller*. DSC řídí běh celé simulace a řídí komunikaci mezi jednotlivými simulátory. Pro tyto účely DSC využívá High Level Petri Nets (HLPN). Dalšími důležitými prvky jsou nástroje, které umožňují návrh celých simulací, mezi ně patří např. Simulation Editor. Simulační architektura může být zobrazena ve 2D nebo 3D animované reprezentaci, která ukazuje datové toky a důležité události během simulace.

Celá architektura zpočátku umožňovala pouze komunikaci založenou na CORBA, ale její nevýhody, jako byl např. problém s přístupem ke vzdáleným simulačním objektům mimo místní síť (LAN), donutily její autory umožnit i komunikaci přes HTTP protokol.

Podrobněji je celá architektura popsána v [40] a rozšíření pro integraci webových simulátorů je věnován článek [30]. I přes tyto zdroje, však není možné k celé architektuře najít dostatek informací, které by obsahovaly podrobnou specifikaci a blíže popsaly její část, která používá webových služeb.

5.3 Hodnocení simulačních architektur

Výše popsané architektury jsou buď nedostatečně obecné (Robotic Studio) nebo koncepčně zastaralé a neumožňující dekompozici na služby (HLA). V případě MSI se pak jedná spíše o systém pro propojení různých simulací, který slouží stejnému účelu jako HLA. Obecně se tak dá říci, že koncept nezávislých služeb v dnešních simulačních architekturách chybí nebo je využíván jen částečně.

6 Simulační architektura založená na službách

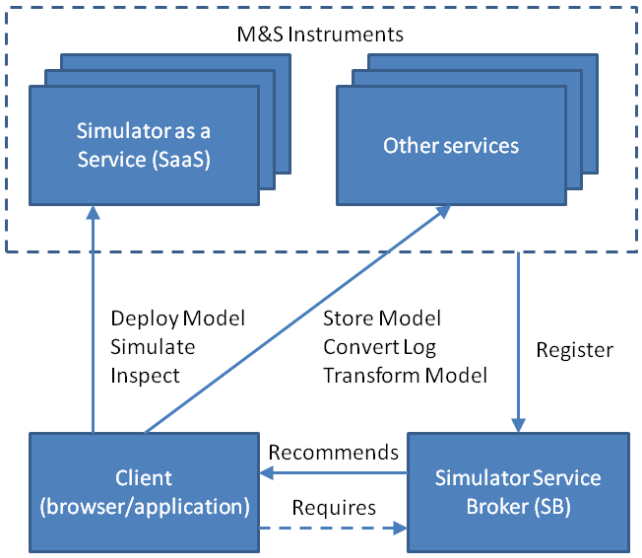
Navrhovaná simulační architektura je založená na webových službách, které zde tvoří diskrétní aplikace poskytující své služby v počítačové síti nebo přes internet. Pro definici rozhraní služeb je použito jazyka WSDL. Komunikace s ostatními prvky v síti, kterými mohou být klient či jiné služby, probíhá pomocí zpráv s využitím protokolu SOAP. Celá komunikace má tedy jednotný formát a tím je jazyk XML, který patří mezi velmi rozšířené a podporované výrazové prostředky.

Výhody webových služeb nám umožňují sestavit otevřenou a vysoce modulární architekturu, která tvoří simulační systém, kde jednotlivé komponenty poskytující služby mají dobře definované rozhraní (API) a jsou tak snadno použitelné. Na jedné straně tak máme množinu (propojených) služeb - např. simulátory s rozhraním, které definují poskytované veřejné operace a datové struktury pro výměnu dat, a na druhé straně je klient či konzument, který službu využívá pro splnění svých cílů, např. získání znalostí o systému prostřednictvím simulace jeho modelu.

Celá architektura je navržena tak aby poskytovala důležité modelovací a simulační služby, standardizovala jejich komunikaci, výměnu dat a zjednodušila integraci jednotlivých komponent. Architektura je zobrazena na obrázku 10, kde můžeme vidět všechny základní entity, kterými jsou:

1. Registr služeb (Service Broker) - adresář služeb
2. Simulační služby (Simulation as a Service - SaaS)
3. Ostatní služby
   (a) Transformační služba - zajišťuje převod modelů
   (b) Prezentační služba - úprava a převod simulačního výstupu
   (c) Knihovana modelů (repozitář) - úložiště pro simulační artefakty
4. Klient/konzument

Simulační služby představují zcela nový koncept, který nabízí stejnou funkcionalitu jako současné simulační nástroje, ale řeší jejich nevýhody. Simulátor jako služba (anglicky Simulator as a Service - SaaS) je nezávislá webová služba, která zcela ukrývá svoji implementaci a nabízí jednotný přístup ke svým funkcím. Simulační služby umožňují klientovi nahrát model a spustit jeho simulaci. U běžících simulací je možné zkoumat jejich stav a výsledky. Každá simulační služba poskytuje určitou množinu operací, které



Obrázek 10: Simulační architektura založená na službách

klient může využívat. Množiny operací různých simulačních služeb nemusí být shodné ani úplné a mohou se překrývat. Ve skutečnosti si klient mezi různými službami vybírá tu, která splňuje jeho požadavky a nic mu nebrání využít i více simulačních služeb a porovnat výsledky. Stručný přehled kategorií operací je ukázán v tabulce 1, přičemž konkrétní příklad jejich použití je ukázán níže.

Každá ze simulačních služeb má možnost se registrovat v *Registru služeb (Service Broker)*, který slouží jako UDDI registr (Universal Description, Discovery and Integration) a může být využíván pro doporučení služby na základě požadavků klienta. Takovým požadavkem může být například modelovací formalismus, množina požadovaných operací, apod. Pro správné doporučení může registr služeb implementovat sadu pravidel, podle kterých pak provede doporučení. Registr poskytuje přístup k WSDL dokumentům jednotlivých služeb, kde jsou popsány veřejné operace, formát výměny zpráv a datové typy. Obsahuje také umístění služeb a na požádání je poskytuje klientovi, např. jako URL. Dále jsou v registru služeb obsaženy i jiné než simulační služby a klient je tak může snadno nalézt a začít používat. Příklad definičního dokumentu jednoduché simulační služby je uveden v příloze B.

V některých simulačních prostředích, nemusí registr služeb existovat. Klient pak musí mít znalost o všech dostupných službách a může se rozhodnout sám, kterou z nich použije. V tomto případě je však nutná změna na straně klienta v případě, že se změní rozhraní nebo umístění dostupných služeb.

Tabulka 1: Operace simulačních služeb

Kategorie	Popis/Účel
Administrace a konfigurace	Řízení simulace Konfigurace simulační služby
Inspekce a monitorování	Inspekce modelu Inspekce simulace Získání simulačního výstupu/logu Monitorování simulační služby Zprávy o událostech
Manipulace s modelem/simulací	Návrh modelu Editace modelu Manipulace se simulací
Přístup k úložišti	Knihovna modelů Simulační repozitář

Klient je konzumentem služeb a využívá je pro splnění svých cílů či stanovených úkolů. Pro doporučení vhodné služby a pro získání jejího umístění používá Registr služeb. Dále pak již komunikuje přímo s danou službou. Komunikace probíhá pomocí protokolu SOAP. Příklad takové komunikace je ukázán níže, kde je ukázána dvojice zpráv (request/response) pro jednoduché požadavky na spuštění a zastavení simulace. V prvním případě klient požaduje spuštění simulačního modelu a simulační služba odpoví zprávou s jednoznačným identifikátorem běžící simulace. Vlastní model může být ve zprávě poslán buď přímo jako součást XML zprávy nebo přes tzv. *přílohy (SOAP attachments)*, popřípadě je možné použít i tzv. *Message Transmission Optimization Mechanism (MTOM)* [54]. Další možností je uvedení odkazu na umístění modelu, který může být již nahrán v simulátoru nebo uložen ve vzdálené knihovně modelů (viz dále). Simulační služba si pak model sama opatří. Zde jen poznamenejme, že požadavek na spuštění simulace může obsahovat množství dodatečných parametrů, nutných pro spuštění simulace. Komunikace při požadavku na zastavení běžící simulace je zřejmá z příkladu níže a nevyžaduje další komentář.

Příklad komunikace mezi klientem a službou

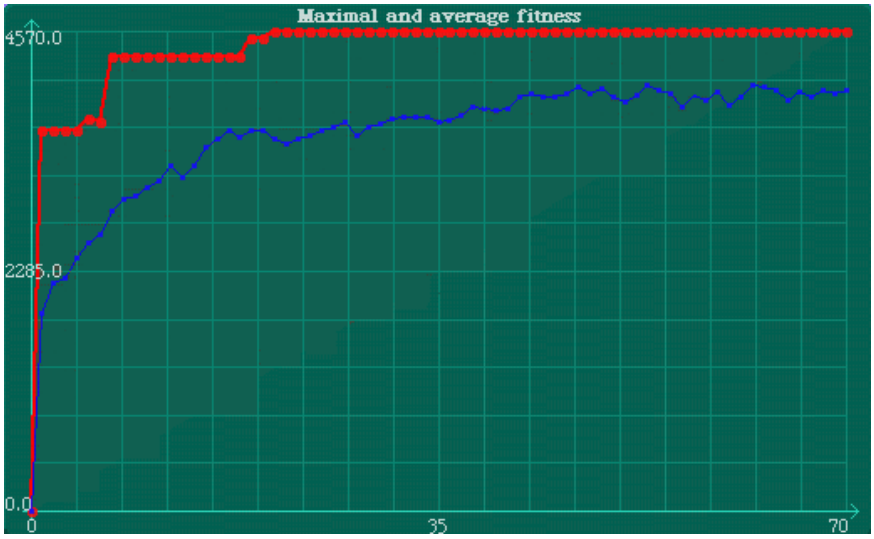
```
Požadavek na spuštění simulace:

<m:startSimulation xmlns:m="http://www.soa/soap">
  <aModel xsi:type="xsd:model">
```

```
    <![CDATA[Base64 encoded model goes here]]>
  </aModel>
</m:startSimulation>

Odpověď na požadavek spuštění simulace:

<m:startSimulation xmlns:m="http://www.soa/soap">
  <aString xsi:type="xsd:string">ID.SIM00023</aString>
</m:startSimulation>

Požadavek na zastavení simulace:

<m:stopSimulation xmlns:m="http://www.soa/soap">
  <aString xsi:type="xsd:string">ID.SIM.00023</aString>
</m:stopSimulation>

Odpověď po úspěšném zastavení simulace:

<m:stopSimulationResponse xmlns:m="http://www.soa/soap>
  <result xsi:type="xsd:boolean">true</result>
</m:stopSimulationResponse>
```

Ostatní služby poskytují jiné než simulační, avšak neméně důležité funkce. Jednou z nich je *prezentační služba*, která může být použita například k převodu nesrozumitelného a těžko čitelného simulačního výstupu (většinou v textové podobě), na více srozumitelný grafický výstup. Jako příklad je níže uveden simulační výstup vytvořený modelem genetického algoritmu (podrobněji popsaného v kapitole 8), který byl vytvořen a simulován v prostředí *SmallDEVS* [12]. Výstup obsahuje maximální a průměrnou hodnotu tzv. *fitness funkce*, která vyjadřuje kvalitu řešení reprezentovaného tímto jedincem pro každou generaci vytvořenou během simulace. Na obrázku 11 pak můžeme vidět grafickou podobu textového simulačního výstupu tak, jak ji vytvořila jednoduchá prezentační služba běžící v prostředí Squeak [49] s rozšířením SoapOpera [50]. Toto rozšíření přidává do systému podporu SOAP protokolu a umožňuje implementaci webových služeb ve Smalltalku [1]. Tato jednoduchá prezentační služba výrazně zvyšuje čitelnost simulačního výstupu a zvyšuje uživatelský komfort při inspekci běžících simulací.

Část textového simulačního výstupu modelu genetického algoritmu

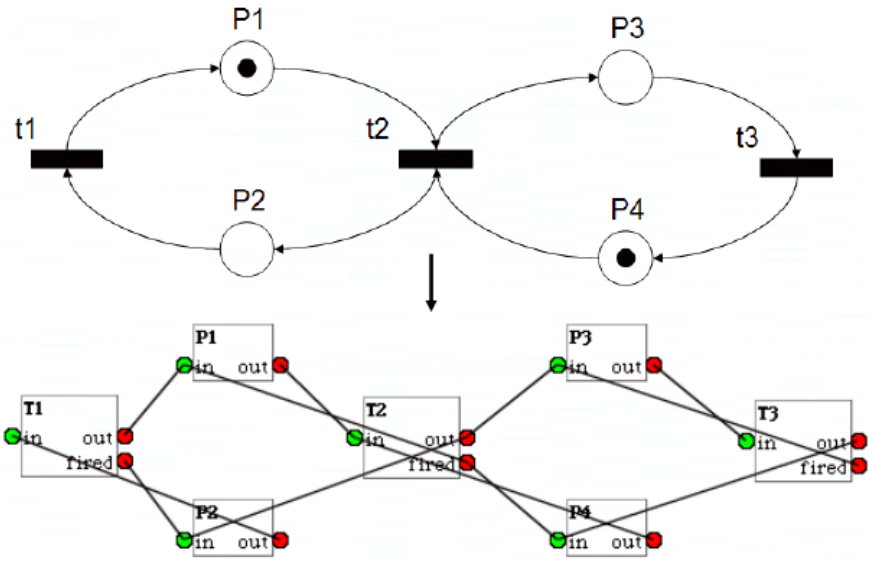
```
<trace>
```



Obrázek 11: Grafická reprezentace vytvořená z textového simulačního výstupu doručeného prezentační službě

```
  <title>Maximal and average fitness</title>
  <generation id="0">
    <value name="max_fitness">0</value>
    <value name="average_fitness">0</value>
  </generation>
  <generation id="1">
    <value name="max_fitness">3640</value>
    <value name="average_fitness">1886</value>
  </generation>
  <generation id="2">
    <value name="max_fitness">3640</value>
    <value name="average_fitness">2183</value>
  </generation>
  ...
</trace>
```

Jednou z dalších důležitých služeb, která v celé architektuře umožňuje multiparadigmatické modelování je *transformační služba*. Tato služba umožňuje automatický nebo poloautomatický převod modelů z jednoho formalismu do druhého. Díky tomu pak vývojáři mohou použít rozdílné formalismy při návrhu modelu a jeho následné simulaci. Příklad takové transformace je ukázán na obrázku 12, kde jednoduchý model vyjádřený Petriho sítí je převeden na model v klasickém DEVS formalismu [38]. V tomto případě trans-

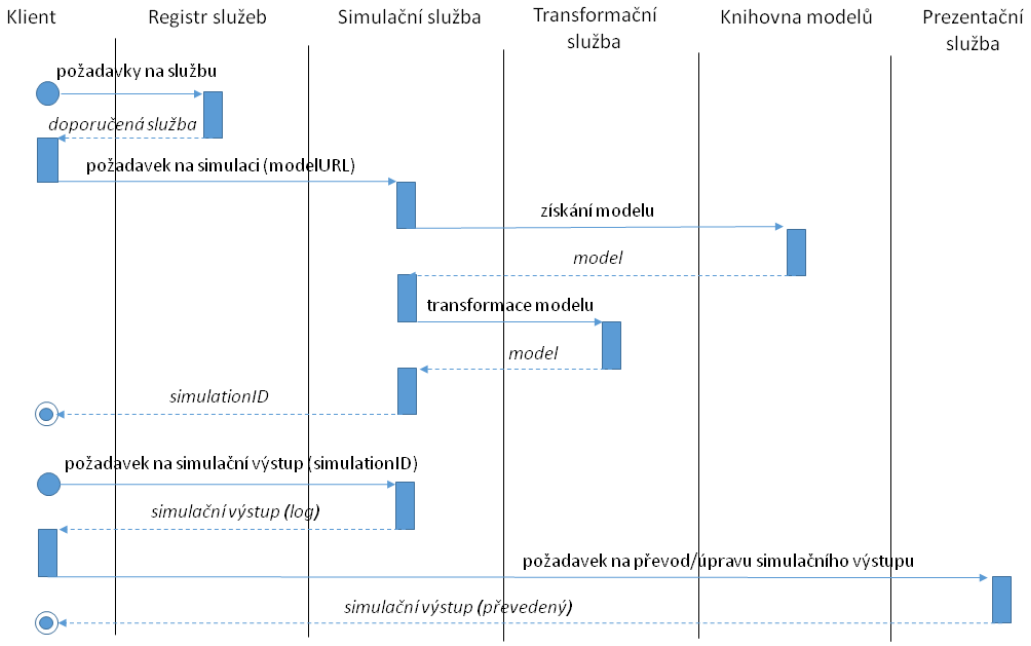


Obrázek 12: Transformace modelu popsaného Petriho sítí na model v klasickém DEVS formalismu

formační služba může použít algoritmus převodu, který je popsán v kapitole 6.3.2. Obrázek je z prostředí SmallDEVS, kde již takový převedený model může být spuštěn a simulován. Ve stručnosti jen uveďme, že každé místo v Petriho síti bylo převedeno na atomickou komponentu. Propojení vstupních a výstupních portů pak reprezentují hrany v Petriho síti. Exekuční sémantika je definována chováním atomických komponent.

Další ze služeb, nazvaná jako *knihovna modelů*, poskytuje základní a důležitou funkcionalitu. Je to místo, které slouží jako úložiště pro modely, dokumenty, knihovny a jiné soubory důležité pro simulaci. Tyto prostředky mohou být organizovány do projektů a sdíleny s jinými klienty. Celá služba tak zlepšuje spolupráci při větších projektech a umožňuje znovupoužitelnost vytvořených modelů. Protože se předpokládá, že knihovna modelů bude používána více uživateli, je vhodné, aby implementovala přístupová práva pro omezení přístupu k jednotlivým zdrojům. Jen autentikovaní uživatelé, kteří byli autorizováni, pak mohou získat přístup k chráněným projektům.

V případě autentizace uživatelů je nutné se vyrovnat s bezstavovostí webových služeb. Tato skutečnost vyžaduje, aby buď každý požadavek byl opatřen autentizačními údaji (např. jméno a heslo) nebo je možné použít speciálního dočasného klíče, nazvaného jako *security token*, který identifikuje ověřenou relaci. Tento klíč musí uživatel opět přiložit ke každému požadavku poslanému webové službě. Více o problematice bezpečnosti v architekturách orientovaných na služby je možné nalézt v [63].

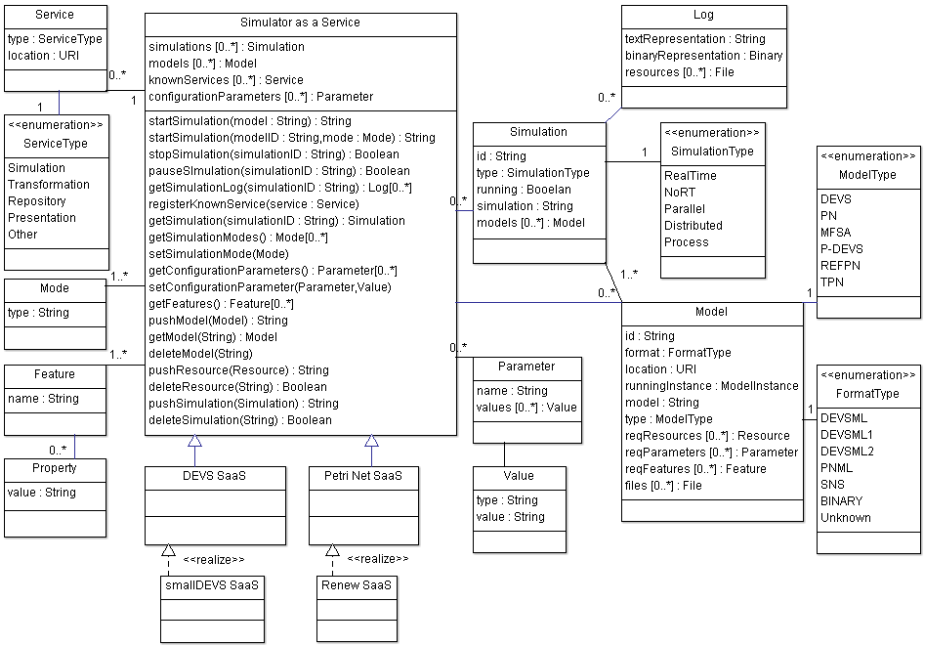


Obrázek 13: Příklad využití služeb pro simulaci

Příklad použití výše popsaných služeb pro spuštění simulace a získání simulačního výstupu je ukázán na obrázku 13. Před začátkem simulace využije klient registru služeb pro doporučení vhodné služby. Registr služeb je pro klienta důležitý i v případě, že klient již má předem vybranou simulační službu, protože obsahuje její definiční dokumenty a umístění. Před spuštěním simulace je třeba dodat simulační službě příslušný model. V tomto případě klient doručí pouze adresu modelu a simulační služba si tento model již sama nahraje z příslušné knihovny modelů. V případě, že se jedná o model, který není kompatibilní s danou simulační službou (např. model je popsaný v jiném formalismu), pak simulační služba kontaktuje transformační službu a ta zajistí jeho převod. Jakmile je model převeden, simulační služba může začít provádět simulaci a v odpovědi pošle klientovi přidělený identifikátor běžící simulace.

Po úspěšném startu simulace pak může klient poslat požadavek na získání simulačního výstupu, pro jehož převedení lze využít dostupné prezentační služby. Na závěr ještě poznamenejme, že registr služeb je využíván nejen klientem, ale i simulační služba jej může kontaktovat např. pro doporučení vhodné transformační služby.

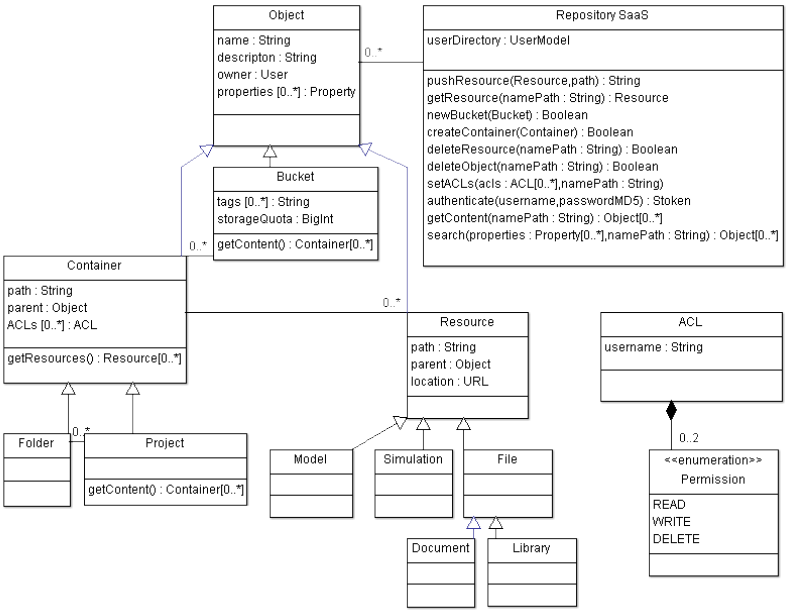
Na obrázku 14 je ukázán příklad rozhraní simulační služby včetně popisu třídní reprezentace významných objektů. Tento diagram tříd může sloužit jako základ pro konkrétní implementaci a zobrazuje hlavní operace, které



Obrázek 14: Rozhraní simulační služby

může vzdálený klient využít. Mezi ně patří operace určené k nahrání modelu, spuštění simulace, získání simulačního výstupu, apod. Konkrétní realizace simulační služby, *SmallDEVS SaaS* a *Renew SaaS*, obsahují navíc operace určené pro manipulaci s modelem pro konkrétní formalismy a jsou zobrazeny v příloze A. Každá simulační služba má možnost specifikovat své konfigurační parametry, které slouží pro její nastavení. Vlastnosti služby jsou vyjádřeny třídou *Feature* a popisují možnosti, které služba nabízí. Jedná se například o množinu podporovaných formátů modelů, typy simulačního výstupu, dále speciální funkce jako možnost změny modelu za běhu, typy podporovaných simulací, možnost získání stavu modelu nebo i simulace, apod. Simulační služba obsahuje také rozhraní pro registraci jiných známých služeb, které je možné použít. Toho je využito v případové studii uvedené v kapitole 7, kde simulační služba využívá jiné (registrované) simulační služby, na kterých spouští sadu reflektivních simulací.

Některé pokročilé operace sloužící např. k získání běžící simulace (*getSimulation*) nebo modelu a jejímu přenosu na jinou simulační službu (*pushSimulation*) tato práce dále nepopisuje a nebyly v rámci případových studií ani ověřeny. Jejich uvedení v diagramech tříd slouží k naznačení možností dalšího výzkumu a rozšíření. Jedná se totiž o netriviální operace, které vyžadují další podpůrné prostředky pro popis stavu běžících simulací či modelů a také pro jejich přenos. Toto však přesahuje rámec této práce, která řeší spíše pro-



Obrázek 15: Rozhraní knihovny modelů

blémy kompatibility simulačních modelů z hlediska popisu jejich struktury či chování a nikoli jejich běžících instancí. Totéž platí i pro uvedené formáty modelů a také výčet typů simulací a modelů, které jsou zde uvedeny jen pro ilustraci. Ověření jejich použití mezi službami v rámci celé architektury by vyžadovalo integraci příslušného simulátoru a zavedení popisu pro nové typy modelů.

Příklad rozhraní knihovny modelů, resp. obecnějšího repozitáře umožňující jednoduchoou autentikaci uživatelů je na obrázku 15. Struktura tohoto repozitáře je rozdělena do *oddílů* (nazvaných jako Buckets). Každému oddílu je možné přiřadit tzv. *storage kvótu*, která určuje maximální prostor, který může tento oddíl alokovat v repozitáři. Po jeho vyčerpání, bude každý další požadavek na vytvoření objektů v tomto oddílu odmítnut. V každém oddílu je možné vytvářet hierarchickou strukturu s adresáři a přidělenými právy pro jednotlivé uživatele (tzv. Access Control Lists). Autentikovaní uživatelé mohou být autorizováni na základě přiděleného *security tokenu*, přiděleného na omezenou dobu, který musí být součástí požadavku na přístup k neveřejným objektům v repozitáři. Ke každému objektu uloženému v repozitáři je možné přiřadit jméno, popis, vlastníka a specifikovat jeho atributy, což jsou textové řetězce určené pro vyhledávání (*search*). Jako uživatelský model (User Model) je možné použít např. adresářový server podporující LDAP (Lightweight Directory Access Protocol) [67].

6.1 Možnosti a výhody architektury

Architektura se snaží zejména zjednodušit návrh a tvorbu modelů, přičemž podporuje jejich migraci, simulaci a inspekci. Snaží se sjednotit přístup k důležitým simulačním službám, které tvoří nezávislé stavební bloky celé architektury a navrhuje jednotný přístup k modelům a jejich simulacím, které mezi sebou mohou v rámci celé architektury komunikovat. Dále nabízí možnost vzdálené simulace a zaměřuje se na znovupoužitelnost modelů použitím univerzálních jazyků pro jejich popis. Architektura, díky své orientaci na služby, umožňuje integraci rozličných modelovacích a simulačních nástrojů a nabízí jejich funkcionalitu vzdáleným klientům. Uživatel či vývojář tak může, po integraci příslušných nástrojů, využít jejich možností a vyvíjet modely v souladu s experimentálním programováním a multiparadigmatickým přístupem (viz kapitola 2).

I když se jedná o architekturu, která je dostatečně obecná a v principu neslouží pouze pro konkrétní typy systémů či modelů, my se pro účely této práce omezíme na modely systémů založených na diskrétních událostech. Hlavními formalismy, které služby v rámci této architektury používají, jsou tedy: DEVS a Petriho sítě. Pro zajištění migrace modelů mezi simulačními službami mohou být použity tyto univerzální jazyky pro popis modelů: DEVSML (viz kapitola 6.2.1) a PNML (viz kapitola 6.2.3, popřípadě to mohou být i jiné proprietární jazyky (viz kapitola 6.2.4), které mají dostatečnou sílu, aby popsaly model.

Distribuovaná simulace Distribuovanou simulací rozumíme několik simulací spojených do většího celku, které spolu mohou komunikovat. Navržená simulační architektura není přímo pro takovou distribuovanou simulaci navržena a navržené rozhraní simulačních služeb neobsahuje přímou podporu ani pro simulaci distribuovaných systémů, kdy různé části modelu běží v různých simulačních uzlech, které si navzájem synchronizují simulační čas. Koncept reflektivní simulace představený v případové studii (viz kapitola 7) však ukazuje, že je možné spustit množinu nezávislých simulací, které se mohou navzájem ovlivňovat pomocí svých simulačních výstupů.

Proto, aby architektura skutečně distribuovanou simulaci umožňovala, by bylo nutné doplnit rozhraní simulačních služeb o příslušné metody určené pro komunikaci mezi simulacemi, popřípadě navrhnout službu zajišťující řízení celé simulace. Téma distribuované simulace však přesahuje rámec této práce a mohlo by být součástí navazujícího výzkumu.

6.2 Popis modelů

Model, jako abstraktní reprezentace systému je jednou z hlavních entit předávaných mezi simulačními službami. Aby byla zajištěna kompatibilita modelů napříč těmito službami, je nutné zvolit dostatečně univerzální prostředek pro jejich popis. Simulační architektura představená v předchozích kapitolách umožňuje migraci modelů a používá jako modelovací platformu Petriho sítě a DEVS formalismus, omezíme se tedy na univerzální popis takových modelů. V případě Petriho sítí takový prostředek již existuje, avšak pro DEVS bylo nutné jej vytvořit. Vznikl tak *DEVS Meta Language* [68], který se stal předlohou (viz [45]) k *DEVS Modeling Language* [46], což je v současné době používaný standard.

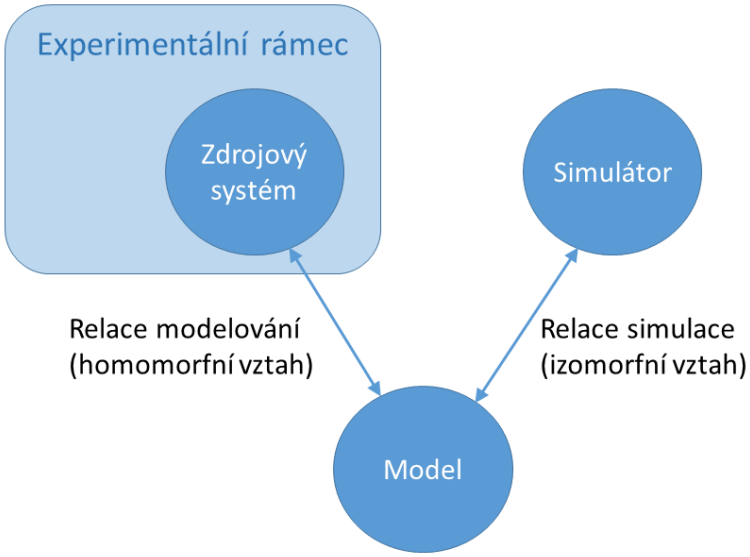
6.2.1 DEVS Meta Language

Formalismus DEVS tvoří formální prostředek pro popis modelů založených na diskrétních událostech. Teoreticky by měly být všechny takové modely nezávislé na simulátoru a také na *experimentálním rámci* (angl. experimental frame - viz obrázek 16), který specifikuje podmínky, za kterých systém pozorujeme a experimentujeme s ním. Ve skutečnosti však do nedávné doby neexistoval žádný standard, který by sjednocoval jejich popis a umožňoval tak jejich nezávislost. Existuje mnoho implementací DEVS formalismu (viz kapitola 3.1) a modely, v nich vytvořené jsou poplatné jejich implementačnímu jazyku a navzájem nejsou mezi různými prostředími kompatibilní. Jinými slovy, znovupoužitelnost modelu je vázána pouze na prostředí, ve kterém byl vytvořen. Z těchto důvodů také neexistují žádné knihovny znovupoužitelných modelů.

V této kapitole si popíšeme navržený metajazyk, který výše uvedené nevýhody odstraňuje. Jedná se o metajazyk pro popis simulačních modelů, nazvaný jako *DEVS Meta Language (DEVSML)* [68]. Vychází z jazyků XML a JavaML [21] a tvoří tak univerzální prostředek pro popis simulačních modelů, s cílem umožnit jejich použití v jiných simulačních prostředích, bez nutnosti výraznějších změn či složitých manuálních úprav. Model implementovaný v tomto jazyce by tak měl být nezávislý na konkrétní implementaci simulačního nástroje a jednoduše, například za pomocí XSL transformace (viz dále), převeden a spuštěn v libovolném simulačním nástroji.

V navrhované architektuře hraje DEVSML roli jednoho z hlavních výrazových prostředků pro výměnu informací mezi simulačními službami a vzdáleným klientem.

Pro zjednodušení tvorby modelů popsaných v tomto jazyce byl vytvořen modelovací nástroj, který umožňuje grafický návrh modelu a vytvoření jeho



Obrázek 16: Simulační modelování

DEVSML reprezentace. Tento nástroj bude představen v kapitole 6.2.2.

DEVS Standardization Group Standardizací DEVS formalismu se věnuje skupina DEVS Standardization Group [58]. Jedná se o skupinu vědců, kteří se zabývají následujícími problémy:

- Standardizace popisu DEVS modelů
- Standardizace simulačních nástrojů - vytvoření specifikace minimálního simulačního nástroje tak, aby byl DEVS kompatibilní
- Interoperabilita existujících nástrojů

Jazyk DEVSML patří do oblasti *Standardizace popisu DEVS modelů*. Nejméně dva projekty existují ve světě s podobným cílem. První projekt [18], [19] představuje modelovací prostředí, které popisuje strukturu modelů pomocí XML, avšak popis chování atomických komponent již je možné specifikovat pouze pomocí pseudokódu. Další řešení je představeno v [20]. Pro popis modelů je zde použito výhradně jazyka XML, ale popis přechodových funkcí je definován pravidly s konečnou množinou stavů, takže tímto způsobem mohou být definovány pouze konečné automaty. Tím je omezena popisná síla modelů.

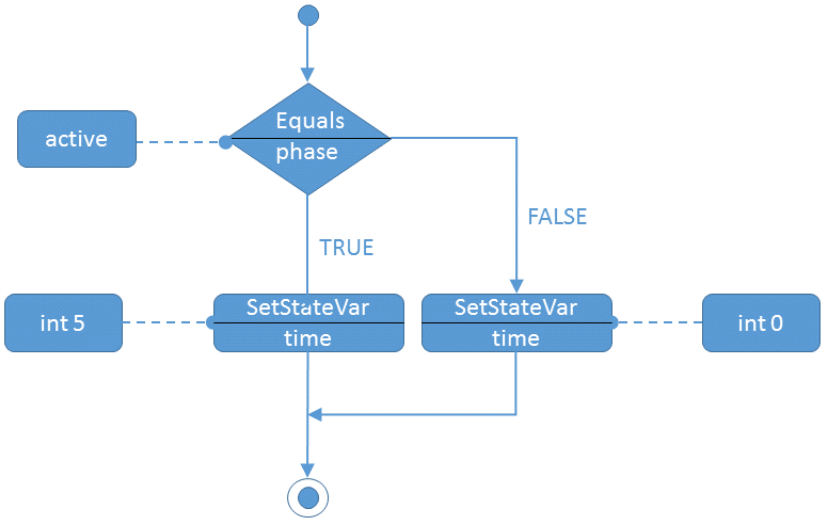
Historie DEVSML Hlavní motivací pro vznik jazyka DEVSML bylo zpřístupnění modelů z prostředí SmallDEVS pro jiná simulační prostředí. Cílem bylo využít výhod nadstandardních funkcí jiných simulačních prostředí a nástrojů. Koncept jazyka DEVSML byl publikován v [68], kde je možné nalézt další informace. Po jeho uveřejnění, na tento výzkum navázala vědecká skupina vedená J. L. R. Martínem a S. Mittalem na Complutense University of Madrid a University of Arizona, která vytvořila podobný jazyk, avšak s názvem DEVS Modeling Language [45], [46]. Tento jazyk měl sloužit podobnému účelu. Jeho první varianta (označená jako DEVSML 1.0) však obsahovala značení pro importování tříd používaných v jazyce Java a byla tak hlavně určená pro jimi používaný simulační nástroj DEVSJava. Ve své druhé verzi doznal tento jazyk značného vylepšení a má v současné tobě největší tendence stát se používaným standardem.

Struktura modelu Díky použití jazyka XML je hierarchická struktura modelu vyjádřena zcela přirozeně. Atomické a spojované komponenty jsou popsány odděleně. Popis spojované komponenty obsahuje definici vstupních a výstupních portů, seznam vnitřních komponent včetně odkazů na dokumenty s jejich definicemi a dále definuje propojení mezi komponentami. Podobně jsou popsány i atomické komponenty, u kterých je potřeba definovat navíc i jejich chování. Každá komponenta je definována v samostatném XML dokumentu, který může být uložen lokálně nebo kdekoli na síti či internetu.

Pro zjednodušení popisu je možné použít koncept *nadřazeného modelu (super model)*, který zavádí princip dědění, tak jak jej známe z jiných programovacích jazyků. Z nadřazeného modelu je možné dědit definice portů, stavových proměnných a také funkcí.

Příklad modelu tvořeného jednou spojovanou a jednou atomickou komponentou je zobrazen v příloze C.1. Zaměříme-li se na strukturu modelu, tak vidíme, že spojovaná komponenta *sampleCoupled* má jeden vstupní a jeden výstupní port a je tvořena jedinou atomickou komponentou *sampleAtomic*, která má také po jednom vstupním a výstupním portu, přičemž oba jsou propojené s porty spojované komponenty. V příloze je uvedena definice struktury obou komponent, jejichž popis portů je velmi podobný. Kromě struktury atomické komponenty je v příloze naznačen i popis jejího chování.

Popis chování modelu Popis struktury atomických a spojovaných komponent je celkem jednoduchý. To však neplatí pro popis chování atomických komponent. Pro zopakování jen uveďme, že toto chování je dané interní a externí přechodovou funkcí, výstupní funkcí a funkcí posunu času (viz kapitola 2.3.1). Na příkladu v příloze C.1 je vidět, že všechny tyto funkce jsou popsány



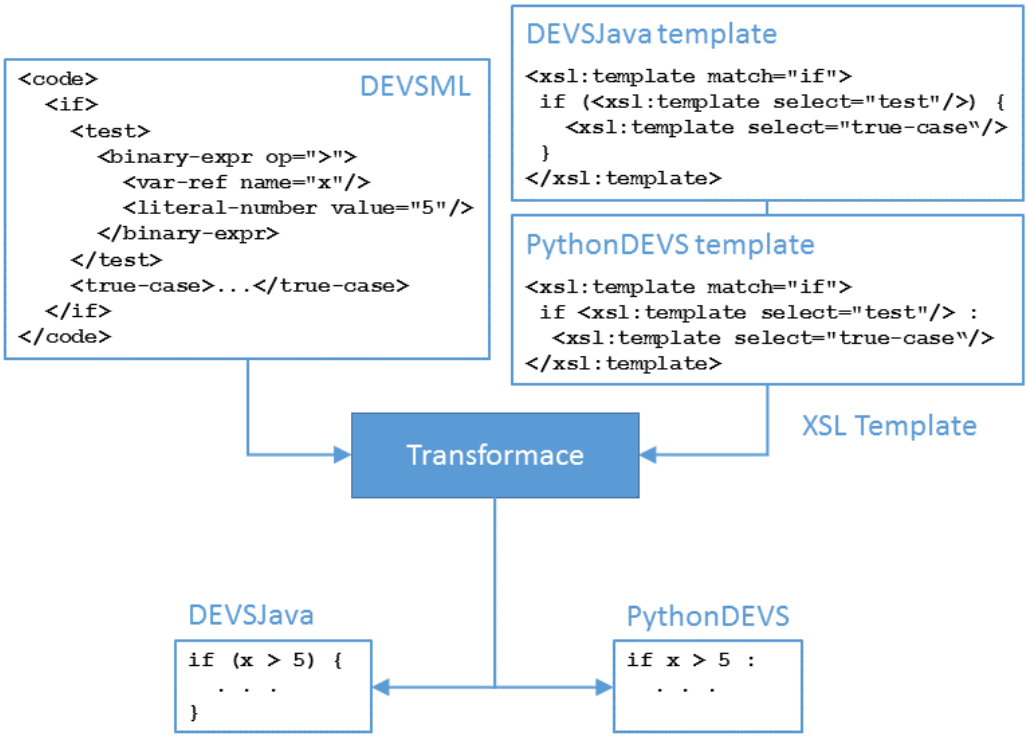
Obrázek 17: Vizuální jazyk pro popis DEVSML funkcí

v částech XML dokumentu označených značkami: `<output-function>`, `<internal-transition-function>`, `<external-transition-function>` a `<ta>`. Značka `<state-variables>` pak obsahuje všechny stavové proměnné, na které je možné se z funkcí odkazovat a jejichž hodnoty určují stav modelu.

Pro definici všech čtyř funkcí atomických komponent čerpá DEVSML inspiraci z JavaML [21], který přináší alternativu zápisu Java kódu v jazyce XML. Snadno je tak možné převést kód do jiného programovacího jazyka a díky tomu je možné jej optimalizovat a verifikovat. JavaML tak poskytuje výrazový prostředek pro popis chování atomické komponenty a vyjádření jejích základních funkcí.

K dispozici jsou základní elementy pro výrazy, mezi něž patří číselné či řetězcové hodnoty nebo unární a binární operátory (větší než, rovno, apod.). Základní syntaktické elementy také umožňují přečíst a nastavit hodnotu stavové nebo lokální proměnné. Dále je možné použít také konstrukce pro vyjádření podmíněného větvení (tzv. if-then-else), cyklu s podmínkou na začátku, resp. na konci (while, resp. until) a cyklu s definovaným počtem průchodů (for). Definici funkce posunu času je také možné zjednodušit výčtem stavových proměnných a k nim přidělených hodnot času.

Mezi předdefinované proměnné a funkce, které je možné využít při definici chování, patří zejména proměnná `elapsed`, resp. `sigma` obsahující čas uplynulý od posledního přechodu, resp. čas do změny stavu pomocí provedení interního přechodu a dále funkce `continue`, určená pro snížení hodnoty `sigma` o uplynulý čas `elapsed`.



Obrázek 18: XSL Transformace

Způsoby popisu chování I když je implementace DEVS modelů přímo v DEVSML jazyce možná, přímá tvorba textového (XML) popisu však není uživatelsky přívětivá. Proto je příhodnější mít k dispozici jiný prostředek pro popis chování. Tím může být například jiný jednoduchý jazyk nebo pseudokód, představený v [68], který je podobný programovacímu jazyku Lisp [64]. Abychom byli konzistentní s grafickým návrhem struktury modelu, byl vytvořen i vizuální jazyk pro definici funkcí - jednoduchý příklad je uveden obrázku 17. Tento přístup se v praxi příliš neosvědčil a jeho podpora byla vždy jen experimentální. Další metodou, která však nebyla ověřena, neboť vyžaduje prostředky pro tvorbu takového popisu, je použití modifikovaných konečných automatů, které jsou popsané v kapitole 6.3.1.

Použití jiných jazyků než DEVSML má smysl pro zlepšení uživatelského komfortu, protože není třeba se učit nový jazyk pro popis chování. Popřípadě nám mohou poskytnout lepší vhled do chování modelu.

Nicméně, v případě použití jiného jazyka, je nutné nejen umět jej převést do DEVSML kódu, ale také mít možnost opačného převodu. Teprve pak je možné DEVSML kód použít jako jediný standardní popis chování, který je

potřebný.

Převod modelu do cílového simulačního prostředí DEVSML popisuje, jak strukturu jednotlivých komponent modelu, tak i sémantiku funkcí určujících chování atomických komponent. Díky tomu může být DEVSML dokument použit přímo pro transformaci modelů do vybraného simulačního nástroje. Tento převod je založen na XSL transformaci, pro kterou je nutné vytvořit nejprve XSL předpis (XSL template). Princip tohoto převodu je zobrazen na obrázku 18. Pro každý simulační nástroj nebo platformu je tedy nutné vytvořit pouze novou XSL šablonu, která je použita pro transformaci, a modely mohou zůstat nezměněné. Takových šablon může být i více a mohou být součástí obecnějšího převodního programu, který je používá a sám doplňuje další informace.

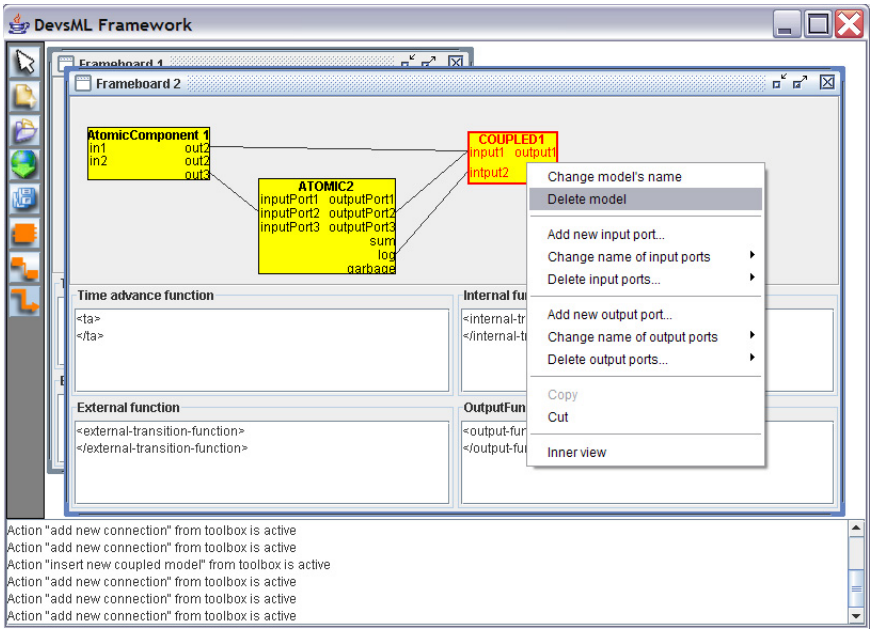
V příloze C.2 je ukázán model jednoduchého procesoru popsaný různými způsoby. Nejprve v jazyce DEVSML, poté ve Smalltalku a na závěr v jazyce Java. Model ve Smalltalku, resp. v Javě je určený pro simulační prostředí SmallDEVS, resp. DEVSJava. Tento model je jednodušší varianta modelu procesoru ukázaného v kapitole 6.3.1, určená pro zpracování jednoho typu úloh.

6.2.2 DEVSML Framework

DEVSML Framework je prostředí určené pro návrh a tvorbu DEVS modelů, přičemž pro jejich popis používá právě jazyka DEVSML a využívá tak jeho předností a propaguje jej. Model je možné vytvořit jako zcela nový nebo jako kompozici konstrukčních bloků, což jsou již vytvořené modely, které je možné do prostředí nahrát a vzájemně propojit. Takto mohou být použity i vzdálené modely a je tedy možné využívat a vytvářet celé knihovny modelů, které mohou být uloženy kdekoliv na internetu. Vytvořený model je pak možné v prostředí i simulovat, přičemž v současné době je podporován pouze klasický DEVS (viz kapitola 2.3.1).

Součástí nástroje je také aplikační a programové rozhraní určené pro tvorbu modelů. Jeho použitím tak může jakýkoliv existující nástroj popsat své modely pomocí DEVSML a vytvořit tak přenositelné modely. Toto rozhraní je zatím k dispozici jen pro jazyk Java.

Grafické uživatelské prostředí zobrazuje přehledně model, který může být snadno upravován. Samotný model je pak možné uložit v podobě DEVSML dokumentů, případně jej dále transformovat do jiného formátu pro jiná běžně používaná simulační prostředí. V současné době je částečně implementovaná podpora pro prostředí DEVSJava a SmallDEVS. Kromě definice základních funkcí atomického modelu, je možné v nástroji definovat i vlastní funkce,



Obrázek 19: DEVSML Framework

a ty pak využít při implementaci chování modelu. Tyto uživatelské funkce jsou definovány jako vlastnosti atomických komponent. Při návrhu modelu je také možné použít dědičnosti v jazyce DEVSML pro definici jednotlivých komponent.

Na obrázku 19 je ukázáno, jak celé prostředí vypadá a uveden příklad tvorby jednoho z modelů.

Celý nástroj je implementován v jazyce Java za použití komponent z knihoven Abstract Window Toolkit (AWT) [65] a Swing [66]. Aplikace je rozdělena na tři části. První částí je *editor* s grafickým uživatelským rozhraním, které umožňuje návrh modelu a definici chování jeho atomických komponent. Druhá část je používaná pro *transformaci* modelu na ekvivalentní reprezentaci použitelnou v jiném simulačním nástroji. Třetí část nástroje je *simulátor*, který je možné použít pro simulaci vytvořených modelů.

Transformační část aplikace není jen implementace XSL transformací. Umožňuje také nahrání modelu ze vzdálené služby, jako je knihovna modelů a umí zpracovávat výsledky transformace tak, aby mohly být přímo použitelné ve vybraném cílovém nástroji.

6.2.3 Petri Net Meta Language

Perti Net Meta Language (PNML) [51] je metajazyk pro popis modelů založených na Petriho sítích. Hlavním účelem PNML je umožnit interoperabilitu mezi různými modelovacími nástroji pro modely založené na Petriho sítích. Díky tomuto standardu je možné tyto modely přenášet mezi různými nástroji.

Pro usnadnění použití tohoto jazyka byl vytvořen tzv. PNML framework, který poskytuje aplikační a programové rozhraní určené pro vytváření PNML modelu. Teoreticky tak stačí napsat rozšíření pro libovolný existující modelovací nástroj, které bude mít přístup k vytvořenému modelu a pouhým voláním metod z tohoto rozhraní vytvoří odpovídající PNML model který bude odpovídat standardu a bude tak snadno přenositelný.

Samotný PNML standard se skládá ze dvou částí:

Sémantická část Tato část definuje sémantický model Petriho sítí. Matematické definice jsou jeho součástí. Podporovány jsou následující typy Petriho sítí:

- Vysokoúrovňové Petriho sítě
- Symetrické sítě
- P/T Petriho sítě

Syntaktická část Definuje abstraktní a konkrétní syntaxi pro různé typy Petriho sítí pokrytých v první části. Abstraktní syntaxe je specifikovaná pomocí UML [22], [23] a konkrétně je pak síť popsána pomocí RELAX NG [24].

Jednou z hlavních vlastností PNML je jeho otevřenost: rozlišuje mezi obecnými vlastnosti všech typů Petriho sítí a vlastnostmi specifickými pro určité typy sítí. Specifické vlastnosti je třeba definovat zvlášť v tzv. definici pro každý typ Petriho sítě. (Petri Net Type Definition - PNTD), přičemž každý typ musí mít svůj vlastní jmenný prostor určený jednoznačným URI (namespace URI).

Některé ze specifických vlastností, které jsou používané více než jedním typem Petriho sítí, mohou být definovány v tzv. ustavovacím dokumentu (conventions document). Sem patří například značení (počet značek) v P/T Petriho sítích. V případě, že daný typ používá ustavovací dokument, pak jeho definice musí obsahovat odkazy na specifické vlastnosti v něm uvedené.

Příklad jednoduché Petriho sítě popsané pomocí PNML je možné nalézt v příloze D.

6.2.4 Shadow Net System formát

Jako jeden ze zástupců proprietárních formátů pro popis modelů si uvedeme *Shadow Net System formát (SNS)*. Je to formát vyvinutý autory nástroje Renew, určený pro přenos modelů mezi různými instalacemi tohoto nástroje pro účely simulace. Formát je binární a není tak čitelný. Umožňuje uložit sdružené modely vytvořené v tomto nástroji do jednoho binárního souboru, který pak je možné přenést do jiného systému a načíst jej pro účely simulace. Modely uložené v tomto formátu však neobsahují popis své grafické reprezentace. Formát popisuje pouze sémantické informace, které jsou důležité pro simulaci včetně modelovacího formalismu a konfigurací simulačního výstupu. Přicházíme tak o možnost vizuální simulace a model již nelze nadále upravovat v grafickém editoru cílového systému. Proti PNML však má tento formát jednu výhodu a tou je malá velikost modelů.

6.3 Transformace modelů

Vzhledem k tomu, že část této práce je možné chápat jako příspěvek k multiparadigmatickému modelování, které navrhovaná architektura umožňuje a podporuje, je třeba na tomto místě pojednat o prozkoumaných, upravených a vytvořených možnostech převodů modelů na formalismy, které tvoří základní modelovací platformu této práce, tedy na DEVS a Petriho sítě.

Všechny níže uvedené postupy mohou být chápány jako návod pro implementaci algoritmu převodu pro *transformační službu*, která tvoří jednu ze stěžejních komponent celé architektury (viz kapitola 6).

6.3.1 DEVS a konečné automaty

Jak jsme si již v kapitole 2.3.1 uvedli, popis atomických komponent zahrnuje i jejich chování. Toto chování je vyjádřeno pomocí čtyř základních funkcí: interní (δ_{int}) a externí (δ_{ext}) přechodové funkce, funkce časového posunu (t_a) a výstupní funkce (λ). Tyto čtyři odlišné funkce nám určují dohromady chování modelu a často není snadné z nich určit závislosti mezi nimi navzájem a pochopit tak chování modelu. Zdá se tedy, že by mohlo být vhodné použít jiné a názornější formy zápisu chování. Vhodným prostředkem se zdají být konečné automaty, jako matematický model, který může srozumitelně popsat stavy systému a jeho chování. Pro naše účely konečné automaty modifikujeme a rozšíříme o hranové výrazy a přidáním času vytvoříme formální model pro zápis chování atomických komponent DEVS systémů.

Nejprve definujme základní pojmy:

- $V_p = \{p_1, p_2, p_3, .., p_m\}$, resp. $V_g = \{g_1, g_2, g_3, .., g_n\}$ je množina *fázových proměnných (phase variables)*, resp. *strážních proměnných (guard variables)*, kde každá proměnná může nabývat hodnoty z množin $V_{p1}, V_{p2}, V_{p3}, .., V_{pm}$, resp. $V_{g1}, V_{g2}, V_{g3}, .., V_{gn}$. Množiny V_{pm} a V_{gn} mohou být definovány buď výčtem možných hodnot (např. $\{true, false\}$, $\{passive, active\}$) nebo mohou nabývat hodnot z množin reálných či přirozených čísel (např. $V_{pm} \simeq \mathbb{R}$ nebo $V_{gn} \simeq \mathbb{N}$).

- M-ární relace $V_P \subseteq V_{p1} \times V_{p2} \times V_{p3} \times .. \times V_{pm}$ pak tvoří prostor částečných stavů DEVS modelu, nazývaných jako *fáze (phases)*.

- N-ární relace $V_G \subseteq V_{g1} \times V_{g2} \times V_{g3} \times .. \times V_{gn}$ nám určuje dohromady s fázemi úplný stavový prostor DEVS modelu, přičemž strážní proměnné vyjadřují podmínky, které musí být splněny, aby bylo možné provést příslušný přechod v konečném automatu.

- $S = V_P \times V_G$ je $m \cdot n$-ární relace určující úplný stavový prostor DEVS modelu.

- $X = \{(p, v) | p \in InputPorts, v \in X_p\}$ je množina vstupních portů a jejich hodnot, kde $InputPorts$ značí množinu jmen vstupních portů a X_p množinu přípustných hodnot pro daný port $p \in InputPorts$.

- $Y = \{(p, v) | p \in OutputPorts, v \in Y_p\}$ je množina výstupních portů a jejich hodnot, kde $OutputPorts$ značí množinu jmen výstupních portů a Y_p množinu přípustných hodnot pro daný port $p \in OutputPorts$.

Modifikované konečné automaty pro popis chování DEVS modelů si v tuto chvíli můžeme definovat následujícím způsobem:
$FSA = (P, T_{ext}, T_{int}, W_{ext}, W_{int})$, kde

- $P = \{phase_1, phase_2, .., phase_n\}$ je množina fází, kde $phase_n \in V_P$.

- $T_{ext} \subseteq P \times P$ je binární relace, nazvěme ji *externí přechod*.

- $T_{int} \subseteq P \times P$ je binární relace, nazvěme ji *interní přechod*.

- $W_{ext} : T_{ext} \to G \times 2^X \times E \times A$, kde G je soubor podmínek (guards) pro provedení přechodu, určený parciálním zobrazením: $G_z : V_g \to V_G$, přičemž přechod $t \in T_{ext}$ je pak možné provést jen tehdy, když jsou splněny všechny tyto podmínky. Jako $true \in G$ budeme označovat speciální podmínku, která je za všech okolností splněna. Pro jednotlivé podmínky dále umožníme použití binárních relačních operátorů, kterými jsou $==$, $<$, $\geq$, apod. 2^X značí přicházející vstup (tzn. páry

vstupních portů a jejich hodnot). $E \subseteq \mathbb{R}^+$ označuje uplynulý čas (elapsed time) od provedení posledního přechodu, přičemž jeho konkrétní hodnotu budeme označovat jako $e \in E$. A je pak soubor akcí (actions), určený parciálním zobrazením: $A_z : V_g \to V_G$, které se mají provést po vykonání přechodu - jedná se tedy o přiřazení nových hodnot proměnným. Jako $noAction \in A$ budeme označovat akci, která neprovede žádné přiřazení. Přiřazení budeme označovat symbolem $=$.

- $W_{int} : T_{int} \to G \times T \times 2^Y \times A$, kde G je soubor podmínek (guards), definovaný výše, přičemž přechod $t \in T_{int}$ je pak možné provést jen tehdy, když jsou splněny všechny tyto podmínky. $T \subseteq \mathbb{R}^+$ označuje časovou podmínku pro provedení přechodu, přičemž přechod $t \in T_{int}$ je možné provést jen tehdy, pokud model setrvá v aktuálním stavu po dobu $time \in T$. 2^Y značí výstup, který je generován před provedením přechodu a znakem $\oslash \in 2^Y$ budeme označovat situaci, kdy nebude generován žádný výstup. Množina A značí výše definovaný soubor akcí (actions).

Zde si uveďme příklad jednoduchých DEVS modelů, jejichž chování vyjádříme pomocí výše definovaných modifikovaných konečných automatů.

Generátor s přepínačem Tento jednoduchý generátor pracuje ve dvou módech: *passive* and *active*. Když je v módu *passive*, nevykonává žádnou činnost, pouze čeká na aktivaci přes vstupní port. V módu *active* generuje v pravidelných časových intervalech, daných periodou *period* výstup 1. Generátor v aktivní fázi (tedy v módu *active*) může být vypnut posláním 0 na vstupní port. Model je inspirován modelem uvedeném v [38], kde však chybí přepínač. Model generátoru může být popsán v klasickém DEVS formalismu následujícím způsobem:

- Nejprve si označme $e \in \mathbb{R}^+$ jako čas uplynulý od provedení posledního přechodu a $period \in \mathbb{R}^+$ jako časovou konstantu definující interval mezi jednotlivými výstupy generátoru.
- $S = \{(s, \sigma) | s \in \{active, passive\}, \sigma \in \mathbb{R}^+\}$
- $X = \{(in, 1), (in, 0)\}$
- $Y = \{(out, 1)\}$
- $\delta_{int}(active, \sigma) = (active, period)$
  $\delta_{int}(passive, \sigma) = notDefined$

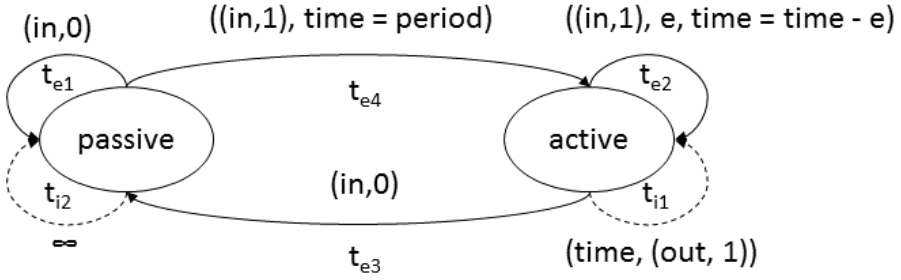
- $t_a(active, \sigma) = \sigma$
  $t_a(passive, \sigma) = \infty$

- $\delta_{ext}(((passive, \sigma), e), (in, 0)) = (passive, \sigma)$ - zde nezáleží na konkrétní hodnotě času e
  $\delta_{ext}(((passive, \sigma), e), (in, 1)) = (active, period)$
  $\delta_{ext}(((active, \sigma), e), (in, 0)) = (passive, \sigma)$ - zde nezáleží na konkrétní hodnotě času e
  $\delta_{ext}(((active, \sigma), e), (in, 1)) = (active, \sigma - e)$

- $\lambda(active, \sigma) = (out, 1)$
  $\lambda(passive, \sigma) = notDefined$

Grafická reprezentace chování pomocí modifikovaných konečných automatů je ukázána na obrázku 20. Poznamenejme jen, že přechod t_{i2} může být vynechán. Na obrázku byl dále vynechán uplynulý čas e v přechodech t_{e1}, t_{e3}, t_{e4} jelikož nemá žádný vliv na provedení přechodu. Formálně tento konkrétní model můžeme popsat takto:

- $V_p = \{phase\}$, kde $V_{phase} = \{active, passive\}$
- $V_g = \{time\}$, kde $V_{time} = \mathbb{R}^+$
- $V_P = \{active, passive\}$
- $V_G = \mathbb{R}^+$
- $S = V_P \times V_G = \{active, passive\} \times \mathbb{R}^+$
- $P = V_P = \{active, passive\}$
- $T_{int} = \{t_{i1}, t_{i2}\}$, kde $t_{i1} = (active, active)$ a $t_{i2} = (passive, passive)$
- $T_{ext} = \{t_{e1}, t_{e2}, t_{e3}, t_{e4}\}$, kde $t_{e1} = (passive, passive)$, $t_{e2} = (active, active)$, $t_{e3} = (passive, active)$ a $t_{e4} = (active, passive)$
- $W_{int}(t_{i1}) = (true, time, (out, 1), (time = period))$
  $W_{int}(t_{i2}) = (true, \infty, \oslash, noAction)$ - tento přechod může být vynechán, poněvadž nemůže být nikdy proveden
- $W_{ext}(t_{e1}) = (true, (in, 0), e, noAction)$
  $W_{ext}(t_{e2}) = (true, (in, 1), e, (time = time - e))$
  $W_{ext}(t_{e3}) = (true, (in, 0), e, noAction)$
  $W_{ext}(t_{e4}) = (true, (in, 1), e, (time = period))$



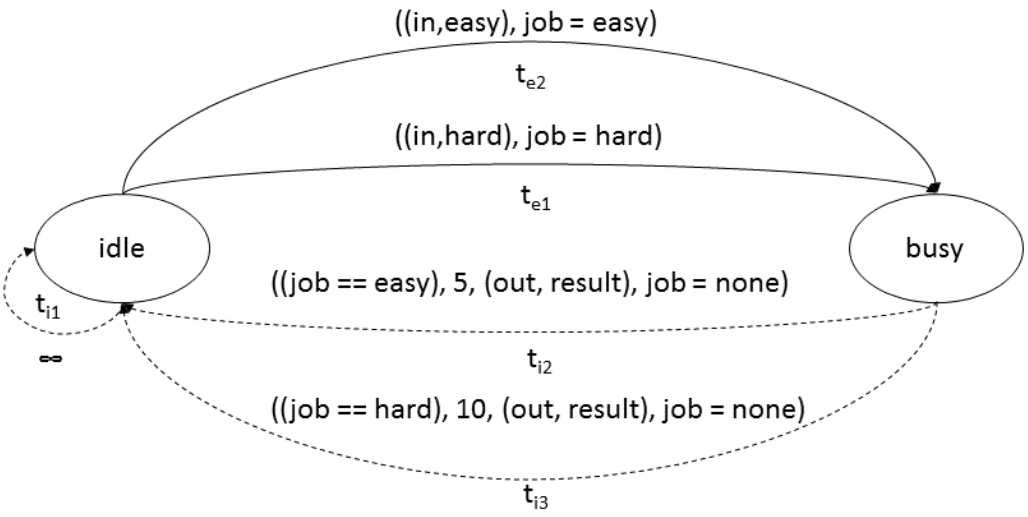
Obrázek 20: Modifikovaný konečný automat, který popisuje názorně chování generátoru s přepínačem

Procesor bez vyrovnávací paměti Tento příklad představuje model procesoru bez vyrovnávací paměti. Je to jednotka, která může přijmout přes své vstupní porty dva typy úkolů: *easy* (jednoduchý výpočet) nebo *hard* (náročný výpočet). Když se procesor nachází ve stavu *idle* (nečinný), pak nevykonává žádnou práci a jen čeká na vstup. Po přijetí úkolu, procesor se přepne do stavu *busy* a začne provádět výpočet. Délka výpočtu záleží na typu úkolu (náročný výpočet (10), jednoduchý výpočet (5)), který určuje dobu, po kterou procesor setrvá ve stavu *busy*. Poté procesor generuje výsledek na výstupní port a přepne se do stavu *idle*.

Grafická reprezentace chování procesoru je na obrázku 21. Poznamenejme jen, že červené části modelu jsou nepovinné a nemusí být uvedeny, jelikož nastavení proměnné *job* na hodnotu *none* neovlivní chování modelu. Tato část modelu je zde uvedena jen abychom ukázali interní stav se všemi čtyřmi prvky přechodové relace. Pro jednoduchost byl na obrázku opět vynechán uplynulý čas e v přechodech t_{e1} and t_{e2}, jelikož je jeho hodnota nepodstatná. Z celého popisu je zřejmé, že procesor ve stavu *busy* nereaguje na nově příchozí úkoly (výpočty) a zahazuje je - tento jednoduchý model totiž nemá k dispozici žádnou vyrovnávací paměť pro ukládání těchto požadavků.

Přeskočme definici chování tohoto jednoduchého procesoru pomocí DEVS formalismu a uveďme si formální zápis jeho chování modifikovaným konečným automatem:

- $V_p = \{phase\}$, kde $V_{phase} = \{idle, busy\}$
- $V_g = \{job\}$, kde $V_{job} = \{easy, hard, none\}$
- $V_P = \{idle, busy\}$
- $V_G = \{easy, hard, none\}$



Obrázek 21: Modifikovaný konečný automat, který popisuje názorně chování procesoru

- $S = V_P \times V_G = \{active, passive\} \times \{easy, hard\}$
- $P = V_P = \{idle, busy\}$
- $T_{int} = \{t_{i1}, t_{i2}, t_{i3}\}$, kde $t_{i1} = (idle, idle)$, $t_{i2} = (busy, idle)$ a $t_{i3} = (busy, idle)$
- $T_{ext} = \{t_{e1}, t_{e2}\}$, kde $t_{e1} = (idle, busy)$ a $t_{e2} = (idle, busy)$
- $W_{int}(t_{i1}) = (true, \infty, \oslash, noAction)$
  $W_{int}(t_{i2}) = ((job == easy), 5, (out, result), (job = none))$
  $W_{int}(t_{i3}) = ((job == hard), 10, (out, result), (job = none))$
- $W_{ext}(t_{e1}) = (true, (in, hard), e, (job = hard))$
  $W_{ext}(t_{e2}) = (true, (in, easy), e, (job = easy))$

V [38] je definován *DEVS Definition Language* který používá tzv. *přechodové diagramy (transition diagrams)*. Ty však neumožňují definovat funkci časového posunu t_a.

Modifikované konečné automaty slouží k intuitivnímu popisu chování modelu a zobrazení množiny dostupných stavů. Jednoduché modely nepotřebují žádný dodatečný výklad. Na druhé straně grafický popis modifikovaných konečných automatů se může jevit jako komplikovaný pro složitější modely,

jelikož každý přechod může mít definovány všechny čtyři prvky přechodové relace (jako např. přechod t_{e1} na obrázku 20) a soubory podmínek G a akcí A mohou být také dlouhé výrazy. I přesto se však zdají být tyto automaty vhodným prostředkem pro popis modelů, zvláště v počátečních fázích tvorby modelu.

6.3.2 DEVS a Petriho sítě

V této kapitole je uvedena jedna z možností, jak převádět DEVS model na ekvivalentní reprezentaci v Petriho síti.

Jak Petriho sítě, tak i DEVS tvoří třídu modelovacích technik, které se dají použít pro tvorbu modelů, založených na diskrétních událostech. Je tedy na místě zabývat se možnostmi převodu mezi nimi navzájem. Článek [6] navrhuje postup, jehož aplikací je možné převést Petriho síť na ekvivalentní DEVS model. Tento popis umožňuje simulovat chování Petriho sítě v nástroji, který je určen pro simulaci DEVS modelů, bez nutnosti jeho úpravy. Tato transformace nám umožňuje simulaci heterogenních modelů skládající se z Petriho sítí a DEVS komponent v jednom prostředí. Cílem této kapitoly není prezentovaný převod vydávat za vlastní, ale pouze jej uvést, okomentovat a zhodnotit.

Třemi základními částmi Petriho sítí jsou *místa* (T), *přechody* (P) a *tokováa relace* (F). Postup prezentovaný v článku [6] nám ukazuje, že je možné místa a přechody reprezentovat za pomocí atomických modelů a tokovou relaci pomocí propojení jejich vstupů a výstupů.

Místa Příklad atomického modelu představujícího místo v Petriho síti je uveden na obrázku 22. Tento model, označený jako *Place*, je definován následujícím způsobem:

$Place =< S, t_a, \delta_{int}, X, \delta_{ext}, Y, \lambda >$, kde
$X = \{IN\}$ je množina vstupů modelu, kde $IN \in \mathbb{N}^+$
$Y = \{OUT\}$ je množina výstupů modelu, kde $OUT \in (id, tokens)$
$S = \{tokens\} \times \{id\} \times \{active, passive\}$, kde $tokens \in \mathbb{N}^+{}_0$ je počet značek v místě a $id \in \mathbb{N}^+$ je jednoznačný identifikátor místa.
$t_a(tokens, id, active) = 0$
$t_a(tokens, id, passive) = \infty$
$\delta_{int}(tokens, id, active) = (tokens, id, passive)$
$\delta_{ext}((tokens, id, active \vee passive), e, (id = 0, t)) = (tokens{+}t, id, active)$
$\delta_{ext}((tokens, id, active \vee passive), e, (id <> 0, t)) = (tokens{-}t, id, active)$
$\lambda(tokens, id, active) = (id, tokens)$

Obrázek 22: Model místa

Pomocí vstupního portu je modelu předávána informace o tom, kolik značek je nutné odečíst popřípadě přičíst v případě provedení přechodu. Tato informace je kódována do zprávy, z níž je možné vyčíst tyto informace: identifikátor místa, pro něž je zpráva určena, počet značek a operaci přičtení nebo odečtení značek. V případě zaslání zprávy s hodnotou $id = 0$ jsou značky přičítány, jinak, pokud je zaslané id rovno id místa, tak jsou značky odečítány. Ihned po provedení operace je zaslána přes výstupní port informace o novém stavu místa (viz dále).

Výstupní port je používán pro zaslání informace o aktuálním počtu značek v místě přechodům, které mohou na základě této informace určit, zda mohou být provedeny či nikoli. Zpráva posílaná tímto portem musí obsahovat tuto informaci: identifikátor místa, z něhož je zpráva odesílána a počet značek v tomto místě.

Zprávy mohou být mapovány na přirozená čísla, což je důvod, proč jsou v definici uvedena jako kladná přirozená čísla. Příkladem může být třeba toto mapování:

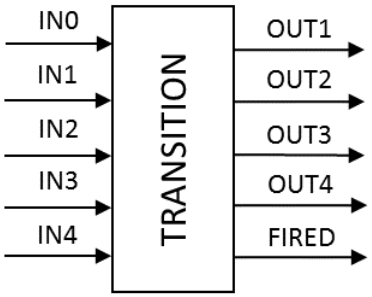
Zpráva	Význam
01..04	přičtení jedné až čtyř značek
11..14	odečtení jedné až čtyř značek od místa s $id = 1$
21..24	odečtení jedné až čtyř značek od místa s $id = 2$
101..104	odečtení jedné až čtyř značek od místa s $id = 1..9$

Přechody Příklad atomického modelu reprezentujícího přechod je uveden na obrázku 23. Tento model, označený jako *Transition*, je definován následujícím způsobem:

$Transition =< S, t_a, \delta_{int}, X, \delta_{ext}, Y, \lambda >$, kde

$X = \{IN_0, IN_1, IN_2, IN_3, IN_4\}$ je množina vstupů modelu, kde $IN_x \in \mathbb{N}^+$

$Y = \{OUT_1 = 1, OUT_2 = 2, OUT_3 = 3, OUT_4 = 4, FIRED \in \mathbb{N}^+\}$ je množina výstupů modelu



Obrázek 23: Model přechodu

$S = \{inputs\} \times \{enabled\}$, kde $inputs \in \mathbb{N}^+{}_0$ je počet připojených míst a $enabled \in /true, false/$ nese informaci o tom, jestli je daný přechod proveditelný nebo není.
$\delta_{int}(0, true \vee false) = (0, true \vee false)$
$t_a(inputs, true) = random(0, 60)$
$t_a(inputs, false) = \infty$
$\lambda(tokens, id, active) = (id, tokens)$

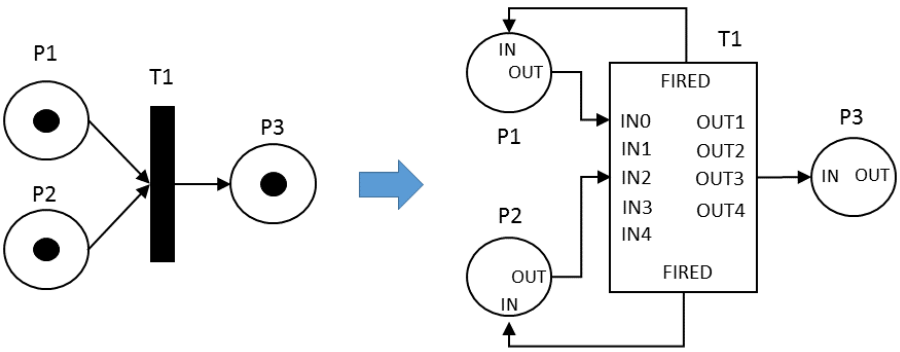
Dále je pro lepší čitelnost definována funkce δ_{ext} pomocí pseudokódu:

δ_{ext} :
```
- ulož do databáze informace o novém stavu místa
- mají-li všechna místa dostatečný počet značek, tak nastav
přechod do stavu, kdy je možné jej provést: enabled = true
```

Mějme spojení $F_x^1 = (P^1, t)$, kde index x označuje váhu W určující, kolik značek se má z místa při provedení přechodu odebrat. Vstupní porty IN_x modelu přechodu t slouží k příjmu zprávy o aktuálním počtu značek v připojených místech $p_n^1 \in P^1$. Pomocí této zprávy může přechod určit, jestli je proveditelný či nikoli. Proveditelný je v případě, že ve všech místech p_n^1 je počet značek stejný nebo větší než je index x.

Mějme teď spojení $F_x^2 = (t, P^2)$, kde index x opět označuje váhu W hrany. Výstupní porty OUT_x pak slouží v případě, že je přechod proveditelný k zaslání informace připojeným místům $p_n^2 \in P^2$, že je potřeba přičíst x značek.

Výstupní port, označený jako $FIRED$, slouží po provedení přechodu k zaslání informace místům p_n^1, že si mají odečíst značky. Počet značek je kódován v zaslané zprávě, stejně tak, jako index místa, pro něž je zpráva určena.



Obrázek 24: Příklad Petriho sítě a odpovídajícího DEVS modelu

Model přechodu má ještě k dispozici databázi obsahující informaci o aktuálním stavu napojených míst p_n^1. Jsou v ní uloženy přechody napojené na porty IN_x a jejich aktuální počet značek. Tato databáze je aktualizována a kontrolována vždy v případě provádění externího přechodu, kdy je do ní zapsán nový stav a v případě, že je pro všechny místa $aktPocetZnacek \geq x$, tak je stavová proměnná *enabled* nastavena na hodnou *true*, což určuje, že je přechod proveditelný. V článku [6] je uvedeno, že informace uložené v databázi nedefinují stav modelu přechodu. Bylo by však výhodnější, kdyby součástí stavu modelu byly.

Příklad jednoduché Petriho sítě a odpovídajícího DEVS modelu je uveden na obrázku 24. U modelu jsou uvedeny pro větší srozumitelnost dva porty $FIRED$, přičemž ve skutečnosti má model tento port jen jeden. Další příklad je uveden na obrázku 12

Hodnocení uvedeného převodu Popsaný postup má nevýhodu v tom, že je platný pouze pro tzv. *klasický DEVS (classic DEVS)*, tedy základní variantu DEVS formalismu. V případě, že bychom chtěli síť simulovat v paralelním DEVS simulátoru, určeného pro variantu označovanou jako *Parallel DEVS*, tak by mohla nastat konfliktní situace, kdy by se dva přechody, napojené na jedno místo, provedly současně a zaslaly by současně (v případě stejného náhodně vygenerovaného času funkcí t_a) zprávu na porty OUT_x a $FIRED$. To by nebylo korektní chování, protože po provedení prvního přechodu by v místě nemuselo být dostatek značek pro provedení přechodu druhého. V případě klasického DEVSu tato situace nenastane. Současně vyvolané výstupní funkce λ u dvou komponent by totiž byly simulátorem zpracovávány sekvenčně. Po zpracování první by se na výstupu místa, z něhož by se značky odebíraly, objevila informace o novém stavu a ta by dorazila na

vstup druhého přechodu, kde by nastal konflikt mezi interní a externí přechodovou funkcí. Přednost by pak dostala interní funkce a po změně stavu by k vyvolání externí přechodové funkce již nedošlo.

Převod Petriho sítě na ekvivalentní reprezentaci DEVS modelu nám umožňuje zacházet s Petriho sítí stejně jako s DEVS modelem a umožňuje nám jej simulovat v nástroji podporujícím DEVS formalismus (např. DEVSJava, PythonDEVS, DEVSC++ nebo také SmallDEVS). Velmi vhodné by bylo celý převod realizovat automaticky a DEVS model reprezentovat v jazyce DEVSML 6.2.1, který umožňuje migraci DEVS modelů mezi simulačními prostředími. Úprava uvedeného postupu tak, aby byl model platný i pro Paralelní DEVS simulátor a implementace automatického převodníku (například v prostředí SmallDEVS) by mohla být součásti výzkumu navazujícího na tuto práci. Celý převodník pak může být součástí simulační architektury jako veřejně dostupná transformační služba (viz kapitola 6). Uvedený transformační postup by se pak dal snadno verifikovat za pomocí simulace. Výsledky simulace by měly ukázat, že chování modelu je totožné s chováním Petriho sítě, kterou reprezentuje.

6.4 Integrace nástrojů a aplikací

Na tomto místě se stručně zmíníme o integraci nástrojů a aplikací do simulační architektury. Představíme si jeden z nástrojů, který byl použit pro implementaci webových služeb a klientských aplikací a popíšeme postup, jak vytvořit z existující aplikace webovou službu tak, aby mohla být k dispozici v navržené simulační architektuře.

Pro integraci a instalaci existujících simulačních aplikací a nástrojů je vhodné použít *Apache Axis2* [55], jako základní platformu pro webové služby. Je to přepracovaný a vylepšený následovník široce používaného *Apache Axis* [56] - SOAP platformy s implementacemi v jazycích Java a C. Axis2 slouží nejen jako samostatný webový server, na který se jednotlivé webové služby mohou instalovat bez nutnosti jeho restartování, ale umožňuje také přidat rozhraní webové služby k existující aplikaci a vygenerovat zdrojové kódy (tzv. stubs) připravené pro implementaci klientské aplikace, která přistupuje ke vzdáleným službám. Pro tento účel podporuje standardy WSDL 2.0 a SOAP 1.2.

Při vytváření webové služby je možné použít dva rozdílné přístupy, jeden z nich je tzv. přístup *shora-dolů (top-down approach)* nebo *odspodu-nahoru (bottom-up approach)*. V případě prvního přístupu se začíná nejprve vytvářet kontrakt služby, a pak teprve její implementace. V případě druhého přístupu se pokračuje naopak, nejprve se implementuje funkcionalita služby a z ní se vytváří její kontrakt. Abychom z existujícího nástroje vytvořili aplikaci

schopnou pracovat jako webová služba, je nutné vytvořit program, který obsahuje operační logiku celé služby a z něj dále komponentu webové služby.

Dále, aby bylo možné volat metody vzdálené webové služby, musíme mít k dispozici klientskou aplikaci, která rozumí rozhraní služby a ví jakým způsobem používat její veřejné metody. Platforma Axis2 obsahuje podpůrné prostředky pro vytvoření základní implementace takové aplikace. Pro tento účel potřebuje pouze WSDL popis služby, který který definuje veřejné operace poskytované službou a datové struktury pro výměnu dat. Tento popis je možné získat na požádání z webového serveru, na němž je služba umístěna. Z něj pak, pomocí nástroje Axis2, je možné vygenerovat základní třídy nové klientské aplikace.

6.5 Simulátor jako služba v cloudovém prostředí

Protože celá simulační architektura prezentovaná v této práci je založená na vzdáleném přístupu ke službám a směřuje k použití jednoduchých klientských aplikací pro rozličné modelovací a simulační úkoly, popřípadě umožňuje ovládání složitých simulací přes internetový prohlížeč, jeví se nám v tomto ohledu velmi podobná tzv. cloudovému prostředí.

Samotný termín *cloud* se používá v souvislosti s ukládáním dat nebo vzdáleným výpočetním výkonem či vzdálenou infrastrukturou. Cloudové prostředí bychom si mohli charakterizovat jako vzdálenou platformu, která nám poskytuje nějakou službu či aplikaci, která neběží lokálně na našem počítači, ale je celá (či z velké části) umístěna někde na internetu nebo ve vzdálených datových centrech. Mezi hlavní výhody tohoto přístupu patří:

1. Škálovatelnost - rozšiřitelný datový prostor, téměř neomezený výpočetní výkon poskytovaný na požádání.

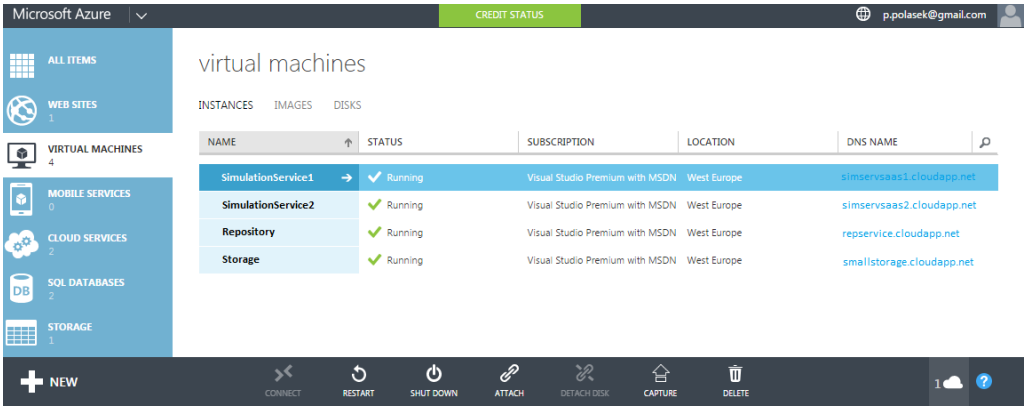
2. Neomezený/vzdálený přístup - Jednoduchý přístup ke službám odkudkoliv bez závislosti na použité platformě

3. Jednoduché klientské aplikace s bohatými možnostmi a funkcemi

4. Nulová či snadná údržba - snadná instalace, konfigurace a ovládání

Všimněme si, že všechny výše vyjmenované výhody cloudového prostředí či aplikací patří i mezi námi stanovené požadavky na simulační architekturu. Takové prostředí je proto vhodnou platformou pro umístění základních komponent navržené simulační architektury, tedy webových služeb.

Termínem *cloudová služba* se označuje taková služba, program či aplikace, která nepracuje lokálně na počítači, ale která je umístěná v cloudovém pro-



Obrázek 25: Webová aplikace pro konfiguraci Windows Azure

středí a máme k ní vzdálený přístup. Taková cloudová služba je z hlediska svých vlastností a výhod téměř totožná s *webovou službou*.

Virtuální stroje běžící v cloudovém prostředí, které je možné na požádání sestavit a spustit s požadovanou konfigurací, nám tu přinášejí obrovské výhody. Zvláště v případě, že se potýkáme s náročnou integrací, instalací a konfigurací webových služeb. Proto je vhodné mít připravené obrazy (tzv. images) předem nakonfigurovaných virtuálních strojů, které je možné na požádání spustit a zapojit do celé architektury. Snadná tvorba, spuštění a údržba virtuálních strojů společně se škálovatelností v cloudovém prostředí nám jej činí vhodným distribuovaným simulačním prostředím, založeným na službách.

Windows Azure Mezi poskytovatele cloudové infrastruktury patří Windows Azure [61]. V rámci případové studie v kapitole 7 bylo prostředí Windows Azure úspěšně použito k instalaci a správě simulačních služeb a ke spuštění reflektivní simulace. Na této platformě byly vytvořeny obrazy funkčních virtuálních strojů s předinstalovanou simulační službou. Díky jednoduché obsluze pak byly jednotlivé stroje spouštěny na požádání a následně použity ke spuštění vzdálené simulace. Po získání simulačního logu je možné, pro úsporu nákladů, tyto stroje jednoduše opět zastavit a v případě spotřeby opět spustit. Díky tomu je možné vykrýt nárazovou zátěž či zvýšené požadavky na výkon.

Stálé cloudové úložiště v prostředí Windows Azure může používat také služba *knihovna modelů*, jako místo pro ukládání modelů a jiných simulačních artefaktů. Díky tomu není potřeba se starat o zálohování a máme tak vždy rychlý přístup k požadovanému obsahu. Webová aplikace pro ovládání cloudového prostředí Windows Azure je ukázána na obrázku 25.

7 Případová studie - systém pro řízení dopravy

V této případové studii byl navržen systém pro řízení dopravy na jednoduché křižovatce, zobrazené na obrázku 28. Tento systém je schopen reagovat na dopravní situaci a podle potřeby modifikovat své vlastní nastavení, tzn. časové intervaly rozsvěcování jednotlivých semaforů. Pro rozhodnutí vhodného nastavení časových intervalů využívá sadu reflektivních simulací pro odhad své budoucnosti a podle výsledků těchto simulací vybere nejvhodnější nastavení a to aplikuje sám na sebe. Reflektivní simulace jsou spouštěny buď lokálně, nebo vzdáleně za použití simulačních služeb.

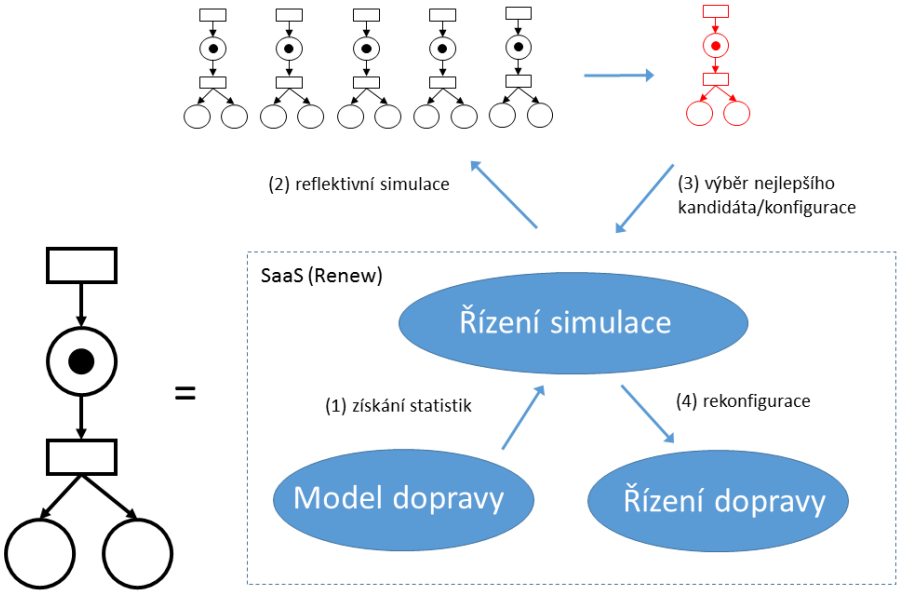
Celý systém je modelován rozšířenými Petriho sítěmi s referencemi. Křižovatka a model řízení dopravy byl inspirován příkladem prezentovaným v [7], přičemž model byl dále rozšířen za účelem umožnění reflektivní simulace a následné rekonfigurace. Ostatní modely - model dopravy (na obrázku 29) a model řízení simulace (na obrázku 32) byly vytvořeny autorem této práce.

Klíčovou aplikací pro tuto případovou studii je simulační nástroj *Renew* (viz kapitola 3.1.1). Těsné provázání jazyku Java a Petriho sítí s referencemi nám usnadní implementaci simulační služby a umožní přístup ke vzdáleným službám přímo z Petriho sítí. Celý nástroj Renew byl použit jako jádro simulační služby a pokud jej přidáme do simulační architektury, pak může sloužit jako veřejný simulátor poskytující své služby vzdáleně přes síť.

Účelem této případové studie je představit funkční implementaci simulační služby a knihovny modelů, jako základních komponent navrhované simulační architektury. Na simulaci netriviálního modelu s použitím reflektivní simulace se budeme snažit ukázat možnosti těchto služeb a ověřit je v praxi.

7.1 Reflektivní simulace

Hlavní koncept reflektivní simulace je znázorněn na obrázcích 26 a 27. Model pro řízení simulace, který reflektivní simulaci spouští, je zobrazen na obrázku 26. Na obrázku 27 pak můžeme vidět model dopravy s modelem pro její řízení a způsob, jakým se navzájem ovlivňují. Řídicí systém reaguje na změny v dopravě (např. vysoký počet vozidel v jednom jízdním směru), což může vést ke změně jeho konfigurace (modifikace časového plánu signálních světel) a tato změna zase zpětně ovlivňuje dopravní situaci (např. zvýšení či snížení průjezdnosti) přičemž i tato změna může vést k další rekonfiguraci řídicího systému, atd. Důležité je si uvědomit, že tyto zpětné vazby obou systémů mohou způsobit nechtěnou oscilaci - snadno totiž může dojít k situaci, při níž systém pro řízení dopravy upřednostní jeden směr s cílem zvýšení průjezdnosti, ale ve směru jiném dojde k nárůstu čekajících vozidel. V následujícím kroku pak systém upřednostní původní směr, ale vozidla budou zase čekat v



Obrázek 26: Princip reflektivní simulace

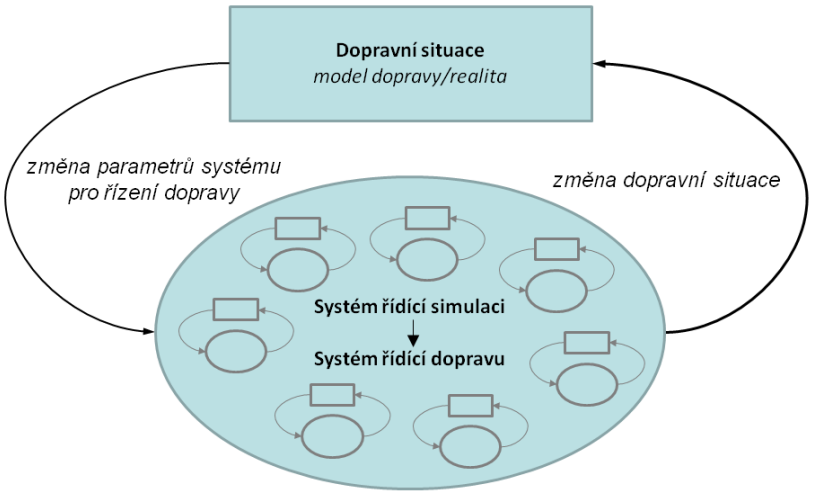
jiném směru. V tomto případě je skutečně výhodné, pokud řídící systém je schopen krátkodobé predikce důsledků svého chování při změně konfigurace. Tím si může ověřit, zda-li změnou svých parametrů opravdu docílil zlepšení dopravní situace v křižovatce. Pokud ne, pak se může stejným způsobem rozhodovat o jiné rekonfiguraci. Pro tyto účely stačí, když systém umí spustit simulaci, popř. simulace svého vlastního modelu a rozhodovat se na základě jejich výsledků.

Reflektivní simulací (někdy také v textu označenou obecnějším termínem vnořená simulace) tak v této studii rozumíme simulaci zaměřenou na budoucnost modelu, jejíž výsledky sám model používá pro rozhodování o změně své vlastní konfigurace. Takovou reflektivní simulaci je možné porovnat s vnořenou simulací prezentovanou v [43] a [44].

7.2 Model systému

Jak je možné vidět na obrázku 28, doprava je rozdělena na tři fáze. V první a třetí fázi jsou zahrnuty jen navzájem neblokující směry - VA, VD, VB, VE. To znamená, že vozidla v těchto fázích mohou projíždět křižovatkou, aniž by si navzájem dávaly přednost. V druhé fázi ve směru VC a VF však vozidla odbočující doleva musí dávat přednost protijedoucím vozidlům.

Přechody pro chodce jsou vyznačeny jako PA, PB, PC, PD. I když model řízení dopravy obsahuje světla řídící tyto přechody a umožňuje také



Obrázek 27: Koncept reflektivní simulace prezentovaný ve vztahu k modelům v případové studii

změnu jejich světelných intervalů, v modelu dopravy byly pro zjednodušení vynechány. Zde tyto přechody připomínáme jen, abychom poskytli kompletní popis řešeného problému.

Model celého systému zahrnuje tři hlavní modely popsané Petriho sítěmi: *model dopravy*, model reprezentující *systém pro řízení dopravy* and *model pro řízení simulace*.

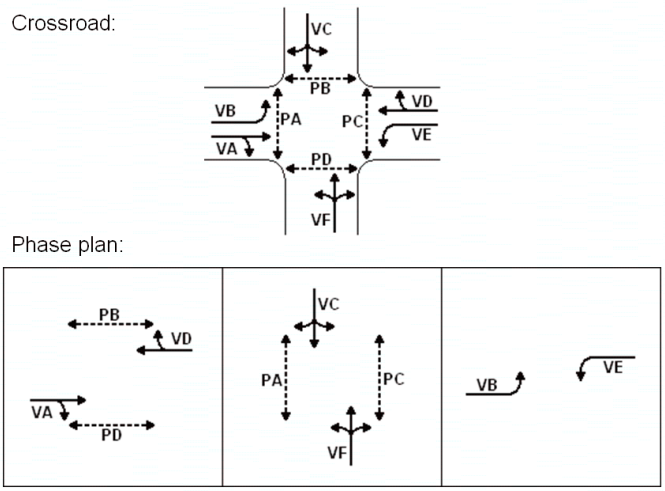
7.2.1 Model dopravy

Model dopravy je zobrazen na obrázku 29. Skládá se ze dvou hlavních částí:

- Vlastní model dopravy - zelená vrchní část sítě
- Model pro získávání statistik o provozu - červená spodní část sítě

Hlavní část sítě představuje jednoduchý *model dopravy*, kde značky reprezentují vozidla. Každé vozidlo buď čeká na průjezd křižovatkou, v místech označených jako *Arrival*, nebo právě projíždí křižovatkou - v tomto případě je značka vozidla v místech označených jako *Transit*. Po průjezdu křižovatkou je značka vozidla umístěna do místa *Exit*. Model obsahuje všechny tři fáze, na které je průjezd křižovatkou rozdělen (viz kapitola 7.2), přičemž průjezd křižovatkou v každém jednotlivém jízdním pruhu je řízen za pomocí synchronizačního místa se značkou [] (tzv. black token).

Jak lze vyčíst z modelu, směry VA a VD mají každý dva jízdní pruhy, směry VB a VF mají každý tři jízdní pruhy a zbylé směry VC a VF mají



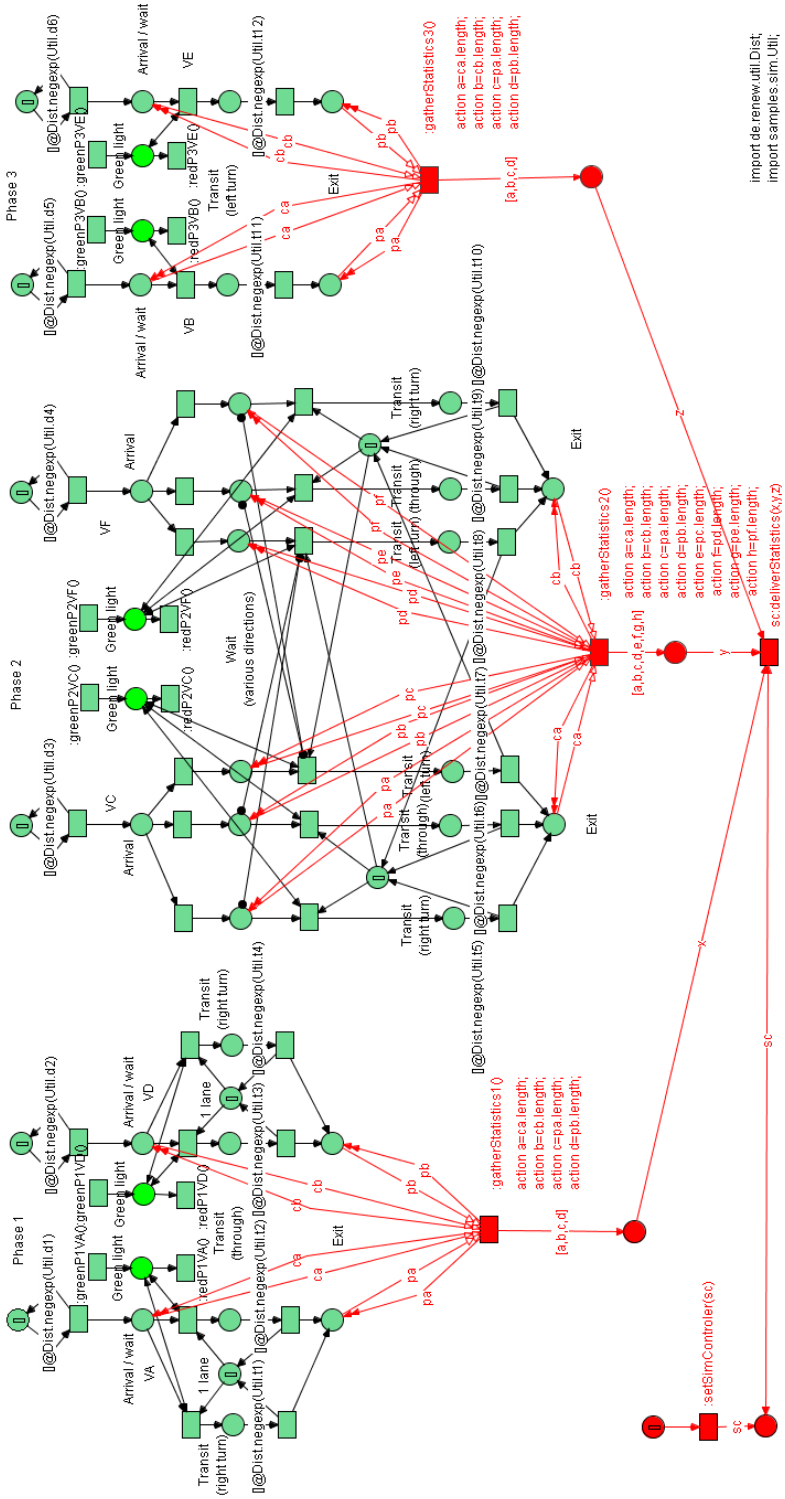
Obrázek 28: Schéma křižovatky a její fázový plán průjezdu (obrázek byl převzat z [7])

každý po jednom jízdním pruhu. Vozidla jsou generována podle exponenciálního rozložení se střední hodnotou, určenou konfiguračním parametrem modelu *Util.dx*, pomocí speciální knihovní funkce *Dist.negexp()* nástroje Renew.

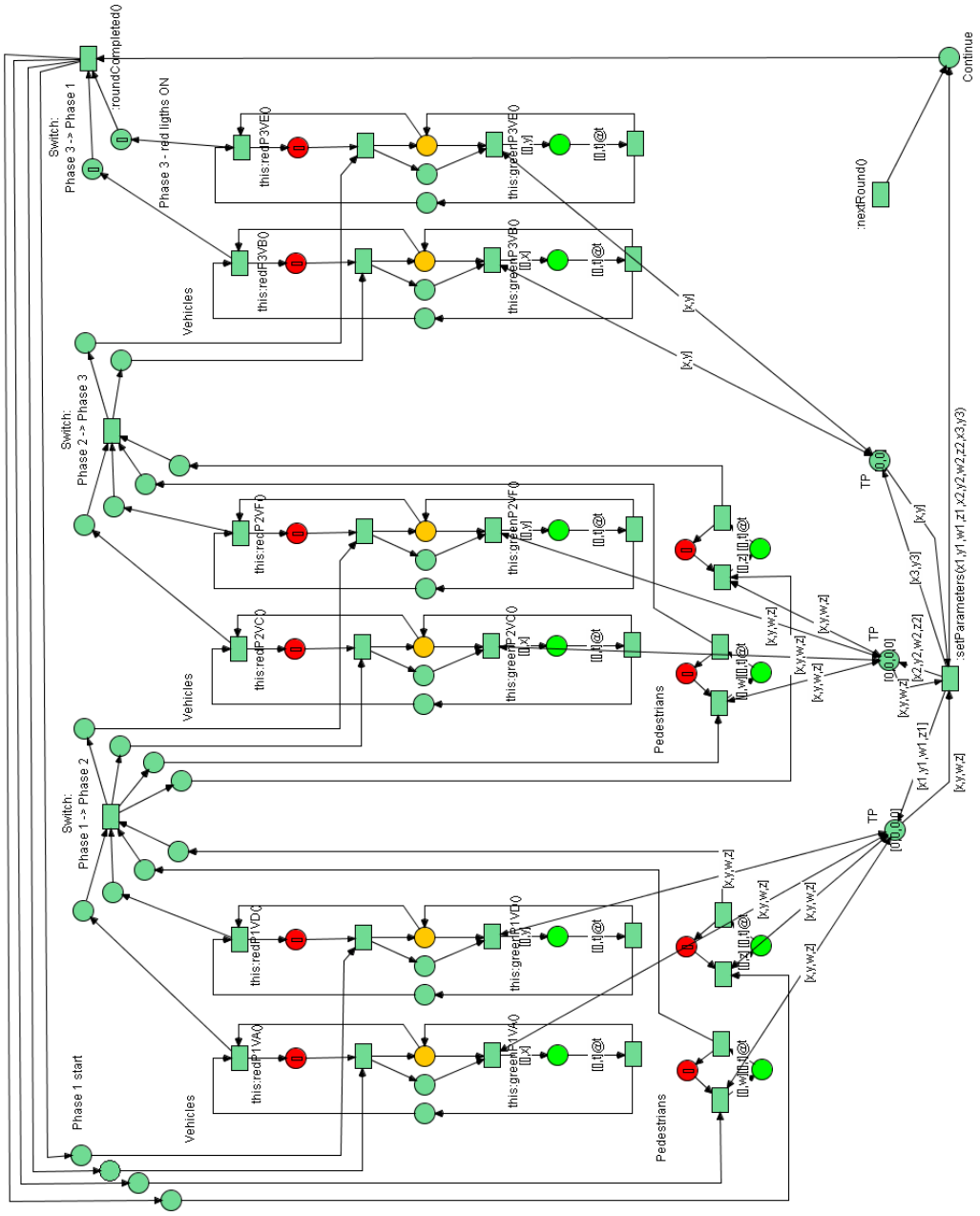
Každý jízdní pruh a každý směr má přiřazený čas, který značí, dobu průjezdu křižovatkou pro jedno vozidlo. Konkrétní časové hodnoty jsou stanoveny konfiguračními parametry modelu *Util.tx*. Uplynutí časového intervalu značí, že vozidlo opustilo křižovatku a pouze tehdy je značka vozidla přemístěna do místa *Exit*.

Část modelu vyznačená červeně slouží ke sběru statistik o provozu. Ty zahrnují počet vozidel čekajících na průjezd křižovatkou v každém jízdním pruhu a počet vozidel, které úspěšně projely křižovatkou v každé fázi. Pro získání této informace jsou použity speciální typy hran, tzv. flexible arcs a clear arcs (viz kapitola 2.3.2). Tyto hrany slouží k přesunutí většího množství značek pomocí jednoho přechodu a vytváří pole se všemi značkami, k němuž je navázána proměnná. *Synchronní komunikační kanály* (tzv. uplinky a downlinky) jsou použity k odstartování procesu sběru statistik. Uplink *gatherStatistics* a downlink *deliverStatistics* slouží právě pro tento účel. Počty vozidel, reprezentované proměnnými x, y a z, jsou dodávány modelu pro řízení simulace, přičemž pro tento účel se používá reference *sc*, která je uložena v modelu pomocí uplinku *setSimControler*.

Abychom umožnili vnořené simulace celého modelu s různými parametry, obsahuje model dopravy část, která umožňuje jeho inicializaci. Jedná se



Obrázek 29: Model dopravy



Obrázek 30: Hlavní část modelu pro řízení dopravy

například o nastavení čekajících vozidel před odstartováním simulace. Tato část modelu však na obrázku 29 není znázorněna.

7.2.2 Model systému pro řízení dopravy

Petriho síť reprezentující model systému pro řízení dopravy je na obrázku 30 a 31. Celý model se skládá ze dvou oddělených částí:

- Model řídicího systému - hlavní část modelu, která slouží pro řízení dopravy
- Synchronizační část - zajišťuje synchronizaci s modelem dopravy

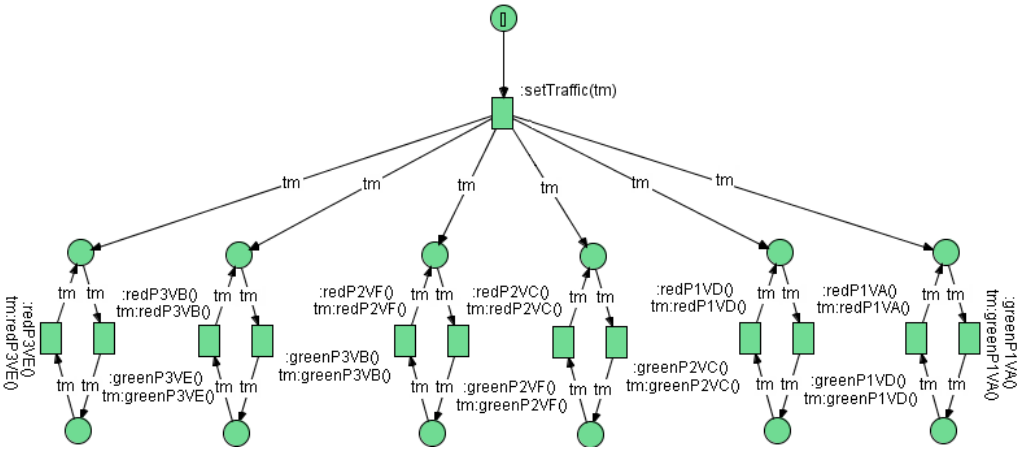
Síť řídicího systému je hlavní část celého systému pro řízení dopravy. Modeluje řízení dopravy ve všech třech fázích pomocí signálních světel pro chodce a pro vozidla. V každé fázi mohou křižovatkou projíždět pouze vozidla z odpovídající fáze v modelu dopravy. Přepínání mezi fázemi je možné provést jen tehdy, pokud je zakázán průjezd všemi jízdními pruhy z předchozí fáze (tzn. svítí červená světla všech semaforů). Chování celého řídicího systému je podmíněno tzv. časovými plány, které obsahují časové intervaly, po které svítí zelená světla povolující průjezd či průchod pro jednotlivé směry v dané fázi. Tyto časové intervaly pro všechny fáze jsou uloženy v odpovídajících místech označených jako *TP* a jsou hlavními parametry celého řídicího systému.

Synchronní kanál *setParameters* umožňuje změnu těchto parametrů. Změna spočívá v uložení nových časových hodnot do *TP* míst. Aby nastavení hlavních parametrů systému proběhlo ve správný okamžik, obsahuje model dva uplinky *nextRound* a *roundCompleted*, které zajišťují, že k modifikaci může dojít pouze před tím, než započne první fáze.

Účelem synchronizační části modelu je synchronizace stavu semaforu s modelem dopravy. Jinými slovy, synchronizační síť sděluje aktivní fázi modelu dopravy. Downlinky jako *tm:greenP1VA()* nebo *tm:redP1VA()* jsou používány k notifikaci modelu dopravy o změnách světelných signálů a uplinky jako *:greenP1VA()* nebo *:redP1VA()* jsou používány jako komunikační prostředek pro stejný účel mezi hlavní částí řídicího modelu a jeho synchronizační částí.

7.2.3 Model řízení simulace

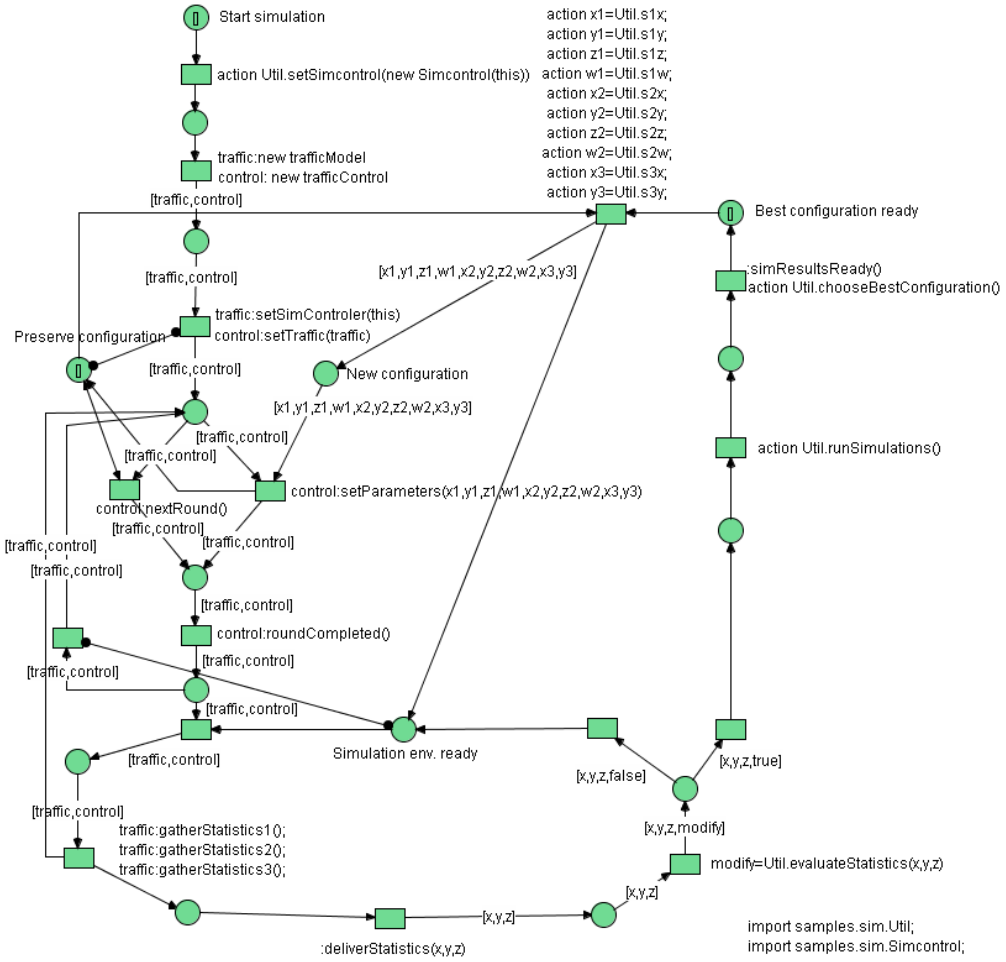
Petriho síť na obrázku 32 slouží k řízení simulace celého modelu, včetně vnořených simulací. Dále slouží k nastavení a rekonfiguraci modelu řídicího systému, přičemž se rozhoduje podle statistik z modelu dopravy a výsledků vnořených simulací. Celý model řízení simulace provádí tyto hlavní činnosti:



Obrázek 31: Model řízení dopravy - část modelu zajišťující synchronizaci s modelem dopravy

1. Inicializuje model dopravy a model řídicího systému a má v sobě uloženy jejich reference.
2. Začíná simulaci
3. Sbírá a analyzuje data z modelu dopravy
4. Komunikuje se vzdálenými simulačními službami
5. Spouští sadu vnořených simulací svého vlastního modelu s odlišnou konfigurací vnitřního modelu pro řízení dopravy
6. Sbírá a interpretuje výsledky vnořených simulací
7. Rekonfiguruje model systému řídícího dopravu podle výsledků vnořených simulací.

Protože Petriho sítě s referencemi v prostředí Renew mohou bez potíží spolupracovat a komunikovat s programy v jazyce Java, je část modelu pro řízení simulace implementována právě v tomto jazyce. Tato implementace se používá přímo v přechodových akcích, v modelu označenými prefixem *action*, kde v inskripčním jazyce jsou uvedena volání třídních metod. V pravém dolním rohu na obrázku 32 vidíme dvě importované třídy. Třída *sample.sim.Util* je určena pro analýzu dopravních dat a slouží ke generování a spuštění vzdálených nebo i lokálních vnořených simulací. Tato třída využívá množinu knihoven, které zajišťují komunikaci se simulačními službami. Třída *samples.sim.Simcontrol* pak je tzv. obalovací třída, která slouží pro přístup



Obrázek 32: Model pro řízení simulace

k síti pro řízení simulací - umožňuje volání synchronních kanálů v této síti a provedení příslušných přechodů přímo z kódu v jazyce Java.

Model řízení simulace na počátku vytvoří instance sítí modelu dopravy a modelu systému řídícího dopravu. Jejich reference jsou předávány napříč sítí v proměnných pojmenovaných jako *traffic* a *control*. Tyto reference slouží pro komunikaci s oběma sítěmi přes synchronní kanály. Reference na vlastní síť pro řízení simulace je předána modelu pro řízení dopravy a umožňuje tak dodání dat o dopravní situaci (dopravních statistik) na požádání.

Poté, co je model systému pro řízení dopravy inicializován a nakonfigurován, začíná okamžitě řídit dopravu. Jeho opětovná rekonfigurace je možná pouze před započetím první fáze. K rekonfiguraci samozřejmě může dojít pouze tehdy, když je nová sada parametrů k dispozici, jinak model pokračuje

s původní konfigurací a nezměněný pokračuje v řízení dopravy.

Model řídící simulaci periodicky sbírá statistiky (*traffic: gatherStatistics()*) z modelu dopravy, provádí jejich vyhodnocení a rozhoduje o tom, jestli je nutné změnit časové plány semaforů v modelu řídícím dopravu - k tomu slouží inskripce *modify=Util.evaluateStatistics()*. Cílem je vždy zlepšení dopravní situace a zvýšení množství vozidel projíždějících křižovatkou. Pokud rozhodovací algoritmus dojde k závěru, že rekonfigurace je nutná, pak vygeneruje množinu modelů-kandidátů s různými konfiguracemi a pro každý z nich spustí vnořenou simulaci (*action Util.runSimulations()*). Cílem je zjistit, zda-li daná konfigurace přinese nějaké zlepšení. Každou simulaci kontroluje jedno vlákno a to komunikuje se simulační službou. Pro vnořenou simulaci je možné použít jednu nebo více služeb běžících lokálně nebo vzdáleně. Komunikace pak probíhá standardním způsobem pomocí protokolu SOAP za použití veřejného API tak, jak bylo nastíněno v kapitole 6.

Vnořené simulace jsou po nějakém čase zastaveny. Po získání simulačního výstupu (logu) a jeho interpretaci je rozhodnuto, který kandidát nejvíce vylepšil dopravní situaci v krátkém čase v simulovaném prostředí. Konfigurace tohoto kandidáta je uložena v místě označeném jako *New configuration* a v příhodném okamžiku pak použita pro rekonfiguraci modelu řídícího dopravu.

Vnořená simulace potřebuje nejen simulační model, který obsahuje všechny tři výše popsané Petriho sítě (model řízení simulace, model systému řídícího dopravu a model dopravy), ale i veškeré potřebné knihovny, konfigurační a inicializační parametry. Toto vše je posíláno simulační službě, která vnořenou simulaci provádí. Protože v současné době nástroj Renew nepodporuje spouštění simulace přímo z modelu popsaném pomocí PNML, je model popsán ve formátu *shadow net* (viz kapitola 6.2) a je posílán jako jeden soubor. Takový model obsahuje jen sémantické informace a nikoli grafickou reprezentaci. Tuto nevýhodu by bylo možné odstranit rozšířením nástroje Renew o podporu spouštění simulací přímo z PNML modelů.

I když je možné nastavit úroveň vnořených simulací, kdy model ve vnořené simulaci může také spustit vnořenou simulaci sebe sama, přičemž i tento model může spustit další vnořenou simulaci, testy ukazují, že tento přístup nepřináší žádné vylepšení a spíše způsobuje nepřiměřené nároky na výpočetní výkon, který s vyšší úrovní zanoření narůstá.

7.3 Simulační nástroj Renew jako služba

Aby mohl model spustit reflektivní simulace vzdáleně, jak bylo popsáno v předchozích kapitolách, je nutné mít k dispozici simulační službu. Pro tyto účely jsme zvolili simulační nástroj Renew (viz kapitola 3.1.1). Bylo však nutné z něj nejprve vytvořit webovou službu, aby mohl simulační služby po-

skytovat i vzdáleným klientům. Přestože se může zdát, že by mohlo stačit spouštět simulace jen lokálně, z obecného hlediska to není pravda. I když simulace relativně jednoduchého modelu dopravního řídicího systému, představeného v případové studii, není nijak náročná, pokud bychom chtěli experimentovat s vnořenými simulacemi spouštějícími další simulace a to třeba v několika úrovních, pak se již můžeme dostat do situace, kdy budeme potřebovat dodatečný výpočetní výkon. Totéž může nastat v případě několikanásobné vnořené simulace složitějšího simulačního modelu. Proto má smysl používat simulátor jako vzdálenou službu a lokální simulaci chápat jen jako speciální případ.

Při tvorbě simulační služby bylo využito přístupu *bottom-up*, popsaného v kapitole 6.4. Nejprve tak byl vytvořen program s operační logikou celé služby a pak teprve byla vytvořena komponenta webové služby. Celá implementace je vytvořena v jazyce Java a má metody s jednoduchými kontrakty, které umožňují spustit nebo zastavit simulaci, získat simulační výstup, atd. Tato implementace již přímo volá funkcionalitu simulačního nástroje (ten tak tvoří její jádro). Je tedy pouze potřeba, aby aplikace Renew byla na stejném systému, kde je umístěna i webová služba. Všechny třídy programu s operační logikou jsou součástí tzv. deployable service artefakt, což je (v případě použití platformy Axis2) jeden soubor v podobě archivu. Ten je možné nahrát na webový server, kde je okamžitě dynamicky nainstalován a zpřístupněn bez nutnosti restartovat celý server. Pro zautomatizování celého procesu, tedy instalace nástroje Renew a instalace služby, je také možné použít sadu skriptů, které vše provedou. Aplikace Renew je spouštěna jako externí program, což provádí operační logika služby, která musí dodat potřebný model a počáteční konfiguraci pro spuštění simulace. V případě, že jsou potřeba nějaké dodatečné knihovny, i ty musí služba dodat tak, aby mohl být simulační model spuštěn. Tyto knihovny mohou být buď dodány klientem zároveň s modelem, nebo mohou být získány z knihovny modelů, popřípadě mohou být i předinstalované s aplikací Renew.

Aby bylo možné volat metody vzdálené webové služby, byly za pomocí prostředku Axis2 vygenerovány základní třídy klientské aplikace, která pak může být volána přímo z Petriho sítí s referencemi. Tímto způsobem třída *Util* z modelu řídícího simulaci přistupuje k webové službě, kde spouští novou simulaci.

Níže vidíme část WSDL popisu simulační služby, která definuje operaci getSimulationLog.

Část WSDL souboru obsahující definici operace getLog

```
<xs:element name="getSimulationLog">
  <xs:complexType>
```

```
    <xs:sequence>
      <xs:element minOccurs="0" name="p_simulationId"
       nillable="true" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="getLogsResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" name="return"
       nillable="true" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Níže je uveden skutečný SOAP požadavek na získání simulačního výstupu (logu) z běžící simulace a příklad takového výstupu získaného ze simulační služby, v jejímž jádře pracuje nástroj Renew.

Příklad požadavku na získání simulačního výstupu

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope
  xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
  <soapenv:Body>
    <ns1:getLogs xmlns:ns1="http://renew.saas">
      <ns1:p_simulationId>SIM.25</ns1:p_simulationId>
    </ns1:getLogs>
  </soapenv:Body>
</soapenv:Envelope>
```

Příklad logu vytvořeného simulační službou Renew

```
[16:28:50] [SIMCONTROL] [INFO] Statistics to be evaluated
[16:28:50] [SIMCONTROL] [INFO] Phase1 (VA, VD):
                    [WAIT: 42, 39],
                    [THROUGHPUT: 10, 12]
[16:28:51] [SIMCONTROL] [INFO] Phase2 (VC, VF):
                    [WAIT: 12, 13, 11, 21],
                    [THROUGHPUT: 17, 15, 15, 11]
[16:28:51] [SIMCONTROL] [INFO] Phase3 (VB, VE):
                    [WAIT: 0, 0],
```

```
                    [THROUGHPUT: 39, 32]
[16:28:51] [SIMCONTROL] [INFO] Creating Renew Client
[16:28:51] [RENEW CLIENT] [INFO] Setup Renew client:
           http://10.51.121.56:8080/axis2/services/RenewService
[16:28:51] [SIMCONTROL] [INFO] Encoding model to Base64
[16:28:51] [SIMCONTROL] [INFO] Request 1 for remote simulation
[16:28:51] [SIMCONTROL] [INFO] Simulation ID assigned: SIM.25
[16:28:51] [SIMCONTROL] [INFO] Request 2 for remote simulation
[16:28:51] [SIMCONTROL] [INFO] Simulation ID assigned: SIM.26
...
[16:28:51] [SIMCONTROL] [INFO] Waiting for simulation results
[16:29:01] [SIMCONTROL] [INFO] Get results from simulations:
                   [SIM.25, SIM.26, SIM.27, SIM.28, SIM.29,
                    SIM.30, SIM.31, SIM.32, SIM.33, SIM.34]
[14-04-07 16:29:01] [SIMCONTROL] [INFO] Got simulation results
```

Každá jednotlivá instance aplikace Renew může být integrována do simulační architektury jako samostatná webová služba. Instalací do Axis2 serveru a registrací v adresáři služeb se stane viditelnou pro vzdálené klienty, kteří tak mohou standardním způsobem použít její veřejné metody pro simulační účely.

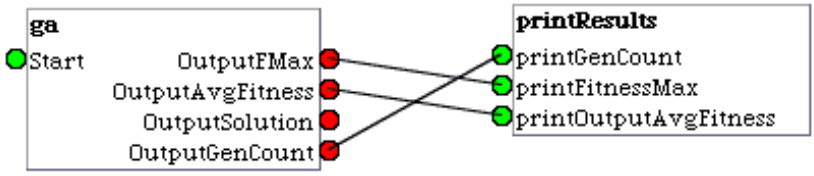
Protože v případové studii je simulační nástroj Renew rozšířen tak, aby podporoval vzdálenou simulaci, mohla by se zdát oprávněná námitka, proč pro tento účel byly použity webové služby, když nástroj Renew obsahuje podporu pro RMI (Remote Method Invocation) [60].

Pojďme si tedy porovnat oba komunikační standardy. SOAP používá jazyka XML a je nezávislý na platformě či implementačním jazyku dané služby. Naproti tomu je RMI mnohem méně obecný a je určen pouze pro programy napsané v jazyce Java. I když RMI dosahuje menší latence při komunikaci, představuje SOAP mnohem robustnější technologii, která umožňuje vytvořit architekturu, jejíž komponenty tvoří nezávislé či navzájem spolupracující služby, k nimž mají přístup rozličné typy klientských aplikací. RMI je více zaměřen na propojení malých aplikací, které musí používat shodný programový model, což vyžaduje častou synchronizaci. SOAP také díky své rozšířené podpoře představuje výhodnější volbu.

7.4 Použití knihovny modelů

Aby nemusel systém, spouštějící reflektivní simulace posílat webové službě vždy svůj vlastní model, byla v této případové studii použita *knihovna modelů*. Místo vlastního modelu je tak ve zprávě pro spuštění simulace zasílán

pouze odkaz na model umístěný ve vzdálené knihovně. Společně s modelem jsou ve vzdáleném úložišti přístupné i potřebné knihovny pro nahrání modelu - ty jsou nutné jen pokud je cílový systém nemá k dispozici. Pro zrychlení spouštění vzdálené simulace byla také implementována vyrovnávací paměť pro načtené modely na straně simulační služby. Tím došlo k výraznému snížení zatížení knihovny modelů.



Obrázek 33: GaDEVS - spojovaná komponenta

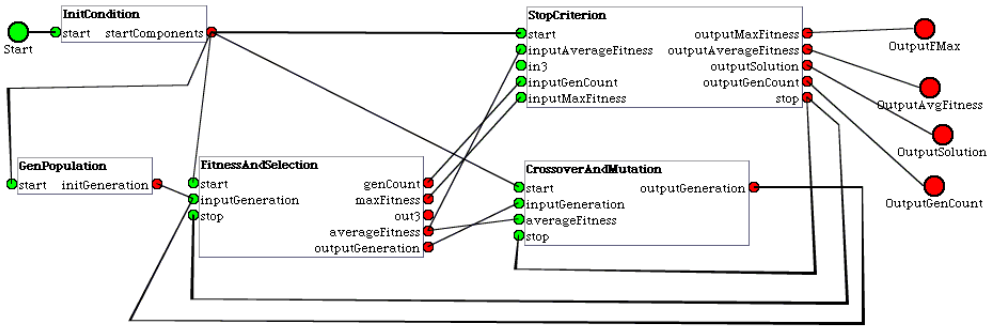
8 Případová studie - genetické algoritmy

V této případové studii si představíme DEVS reprezentaci genetického algoritmu, navrženého tak, aby mohl řešit potenciálně libovolnou úlohu. Tato reprezentace byla označena zkratkou GaDEVS (Genetický algoritmus vyjádřený pomocí DEVS). Model vznikl za účelem předvedení experimentálního přístupu při návrhu systémů a také pro ověření *prezentační služby*. Celá reprezentace byla převzata z [32], dále upravena a implementována v prostředí SmallDEVS.

Genetické algoritmy jsou velmi často implementovány tak, aby co nejlépe řešily konkrétní problém. Jejich vývoj znamená experimentování s napsaným algoritmem a s nastavením jeho parametrů, kterými mohou být např. počet a kódování chromozómů, počet generací, pravděpodobnost křížení a mutace, apod. Abychom snadněji viděli spojitost mezi modelováním, simulací a genetickými algoritmy, můžeme považovat genetický algoritmus za model řešeného problému. Pro experimentování s modelem a jeho vývoj budeme používat simulace, která nám pomůže odkrýt jeho důležité atributy a vlastnosti, např. jak nastavení jeho parametrů ovlivní rychlost konvergence k hledanému řešení, překonávání lokálních maxim, atd.

Protože nám DEVS formalismus poskytuje možnost jednoduše vytvořit hierarchický systém a popsat jeho chování, je ideálním prostředkem pro alternativní reprezentaci modelu genetických algoritmů. Ve spojitosti s grafickým prostředím SmallDEVSu, které nám umožňuje využít experimentálního programování, je tvorba takového algoritmu snadnější než klasický přístup. Změna algoritmu totiž v tomto případě nevede k nutnosti jeho restartu, ale může být prováděna za jeho běhu. Navíc je možné kdykoli (i při spuštěné simulaci) vyměnit hlavní objekt reprezentující řešený problém a začít tak řešit zcela jinou úlohu.

Konkrétní implementace modelu je podrobněji popsána v následující ka-



Obrázek 34: GaDEVS - vnitřní atomické komponenty

pitole a jeho grafická reprezentace je uvedena na obrázcích 33 a 34.

8.1 Detaily modelu

Model genetického algoritmu je implementován jako spojovaná komponenta *ga* (viz obrázek 33), která se skládá z několika vnitřních atomických komponent: *InitCondition*, *GenPopulation*, *FitnessAndSelection*, *CrossoverAndMutation* a *StopCriterion* (viz obrázek 34). Komponenta *printResults*, napojená přímo na *ga*, je zodpovědná za ukládání a zobrazení simulačního výstupu generovaného při exekuci modelu genetického algoritmu. Příklad takového výstupu je uveden v kapitole 6 na obrázku 11. Zde je důležité, že tato komponenta může s výhodou použít prezentační službu, představenou také v kapitole 6, které předá textový simulační výstup a obdrží jeho grafickou reprezentaci, kterou pak může poskytnout uživateli. Celá komunikace mezi modelem, který představuje klienta, a službou tak probíhá standardně v XML pomocí SOAP protokolu podle definice služby.

Reprezentace generace Nejdůležitějším objektem, který je předáván mezi komponentami je objekt reprezentující jednu *generaci*, která obsahuje všechny jedince a také hlavní parametry genetického algoritmu. Tento objekt má metody určené pro inicializaci nové generace, křížení a mutaci chromozómů, ohodnocení chromozómů podle fitness funkce a selekci. Všechny metody jsou používané jednotlivými komponentami v celém modelu.

Popis jednotlivých komponent v modelu Komponenta *InitCondition* slouží k inicializaci a odstartování celého genetického algoritmu. Jednoduše jej spouští tím, že pošle hodnotu 1 na výstupní port označený jako *startComponents*.

Vytvoření počáteční generace probíhá v komponentě *GenPopulation*, která je použita pouze při startu genetického algoritmu. Nová generace je po vytvoření poslána na výstupní port označený jako *initGeneration*, odkud ji převezme komponenta *FitnessAndSelection*. Inicializace generace zahrnuje vytvoření všech chromozómů, které obsahují zpočátku náhodné geny. Dále se také v rámci inicializace nastaví hlavní parametry důležité pro chod genetického algoritmu, mezi které patří pravděpodobnost křížení, pravděpodobnost mutace, zapnutí/vypnutí elitismu, apod.

Komponenta *FitnessAndSelection* ohodnocuje všechny chromozómy v generaci a některé z nich vybírá a označuje pro křížení. Celá generace s označenými chromozómy je pak poslána přes výstupní port označený jako *outputGeneration* do komponenty *CrossoverAndMutation*, která provádí křížení a mutaci. Nová generace je následně poslána zpět do komponenty *FitnessAndSelection*.

StopCriterion vyhodnocuje důležité informace o genetickém algoritmu a aktuální generaci a umožňuje jeho zastavení. To je provedeno posláním hodnoty 1 na výstupní port *stop*. Důležité parametry jako maximální či průměrná dosažená hodnota fitness funkce, počet generací, aj. jsou přístupny na výstupních portech komponenty *ga* a tvoří tak simulační výstup.

Funkčnost výše popsaného modelu genetického algoritmu byla ověřena při řešení tzv. *Knapsack problému* a při triviální úloze hledání maximálního počtu jedniček v jednom bytu.

Závěr

V této práci byly popsány požadavky na návrh a vývoj systémů v oblasti modelování a simulace. Byly představeny M&S techniky, které jsou vhodné pro interaktivní vývoj systémů, přičemž hlavní důraz byl kladen na systémy založené na diskrétních událostech. Tomu také odpovídal výběr modelovacích formalismů určených pro popis modelů v rámci této práce, kterými byly DEVS a Petriho sítě. Byly představeny používané simulační nástroje a diskutovány jejich výhody a nevýhody. Jako hlavní nevýhody a překážky při tvorbě modelů, jejich vývoji a simulaci byly identifikovány: nekompatibilita současných nástrojů, přílišná provázanost modelu a simulačního nástroje a chybějící či nedostatečně používané standardy pro univerzální popis modelů. Bylo zjištěno, že tyto nedostatky vedou k nemožnosti vytvářet znovupoužitelné modely, které bychom mohli využít v rozdílných simulačních prostředích a je také prakticky nemožné použít při vývoji více simulačních nástrojů, aniž by bylo třeba věnovat velké množství času na zajištění kompatibility mezi používanými nástroji a modely.

Dosažené výsledky Jako řešení současných nedostatků byla navržena otevřená a modulární simulační architektura založená na službách. Byly představeny její základní komponenty a jako hlavní koncept byl popsán simulátor jako služba (SaaS). Navržená architektura byla porovnána s existujícími simulačními architekturami, které byly shledány jako nedostatečně obecné nebo koncepčně zastaralé. Dále byl také představen vyvinutý univerzální jazyk *DEVS Meta Language*, určený pro popis modelů založených na DEVS formalismu včetně simulačního prostředí, které jej podporuje. Jako alternativu k základním modelovacím formalismům používaným v rámci této práce byly představeny a definovány modifikované konečné automaty, které tak představují další prostředek pro názorný popis chování systémů založených na diskrétních událostech.

Ověření výsledků práce Použití celé architektury bylo demonstrováno na dvou případových studiích. V jedné studii byla představena implementace simulační služby, kde existující simulační nástroj Renew byl upraven pro použití v rámci celé architektury a mohl tak pracovat jako vzdálená simulační služba. Díky tomu bylo možné vytvořit model dopravního řídicího systému pro jednoduchou křižovatku, který pomocí reflektivní simulace (vzdálené a třeba i několikaúrovňové) je schopen reagovat na změnu v dopravní situaci a provést svou rekonfiguraci. V druhé případové simulaci byl představen model obecného genetického algoritmu určeného pro řešení teoreticky libovolného

problému. Celý model byl vyvíjen a simulován interaktivním způsobem za pomocí nástroje SmallDEVS. V této studii byly použity důležité komponenty celé architektury, kterými jsou simulační a prezentační služba.

Související publikace Dílčí výsledky celé práce byly zveřejněny v řadě publikací uvedených v kapitole Publikace autora. Jedná se především o koncept celé architektury představený v [70] a [71] a také o navržený metajazyk pro popis modelů publikovaný v [68] a [69]. Zvláště [68] se stala inspirací pro jiné výzkumné týmy, o čemž svědčí i řada dosažených citací. Případová studie, uvedená v kapitole 7 byla publikována v [72].

Navazující výzkum Celý koncept simulační architektury založený na službách by mohl být zcela jistě předmětem dalšího výzkumu. Ten by se mohl věnovat jejímu širšímu ověření a popřípadě i dalším rozšířením. Jednoduchá rozšiřitelnost rozhraní jednotlivých služeb v rámci celé architektury je v tomto případě výhodou. Zcela jistě by další pozornost zasluhovalo umístění jednotlivých služeb do cloudového prostředí za účelem snížení nákladů na údržbu a dosažení lepší škálovatelnosti. Samostatným výzkumným celkem by mohlo být multiparadigmatické modelovaní za použití simulačních služeb či použití architektury pro verifikaci modelů a vývoj samostatných verifikačních služeb. Distribuovaná simulace a podpora vývoje distribuovaných modelů, popřípadě modelů založených nejen na diskrétních událostech by mohlo být dalším užitečným rozšířením celé architektury.

Reference

[1] Sandberg, D., W.: *Smalltalk and exploratory programming.* ACM Sigplan Notices, volume 23, issue 10, p. 85-92, 1988

[2] Ungar, D., Smith, R.: SELF: The Power of Simplicity. In: *Proceedings of OOPSLA '87 Conference*, pp. 227–241, 1989

[3] Cabac, L., Duvigneau, M., Moldt, D., Rolke H.: The Power of Simplicity. Modeling dynamic architectures using nets-within-nets. In: *Proceedings, volume 3536 of Lecture Notes in Computer Science*, pp. 148–167. 26th International Conference, Applications and Theory of Petri Nets 2005, Miami, USA, 2005

[4] Kummer, O., Wienberg, F., Duvigneau, M., Kohler, M., Moldt, D., Rolke, H.: Renew - the Reference Net Workshop. In: *Tool Demonstrations*, pp. 99–102. 24th International Conference on Application and Theory of Petri Nets 2003. Department of Technology Management, Technische Universiteit Eindhoven, Beta Research School for Operations Management and Logistics, 2003

[5] Renew - The Reference Net Workshop, `http://www.renew.de`

[6] Jacques, C., J., D., Wainer, G., A.: Using the CD++ DEVS Toolkit to Develop Petri Nets. In: *Proceedings of the 2002 Summer Computer Simulation Conference*, San Diego, CA, USA, 2002

[7] Turek, R.: *Modelování vybraných dopravních problémů s využitím Petriho sítí* (in Czech) [Modeling of Chosen Traffic Problems with Petri Nets]. In: Posterus.sk, Portal pre odborne publikovanie [Portal for Scientific Publications], vol. 3, nr. 12, 2010

[8] Janoušek, V.: *Simulace a návrh vyvíjejících se systémů.* Fakulta informačních technologií, Vysoké učení technické v Brně, Brno, 2011

[9] Lara, J., Vangheluwe, H.: AToM3: A tool for multi-formalism and metamodelling. In: *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, London, UK, April 2002

[10] Giambiasi, N., Paillet, J., L., Chane, F.: From Timed Automata to DEVS Models. In: *Proceedings of the 2003 Winter Simulation Conference*, 2003

[11] Sarjoughian, H., S., Zeigler, B., P.: DEVSJAVA: Basis for a DEVS-based collaborative M&S environments. In: *Proceedings of the 1998 International Conference on Web Based Modelling and Simulation*, San Diego, Canada, ACM, 1998

[12] SmallDEVS, `www.fit.vutbr.cz/ janousek/smalldevs`

[13] Janousek, V., Kironsky, E.: SmallDEVS, an Interactive Modeling and Simulation Tool for Smalltalk. In: *Proceedings of MOSIS'06*, Ostrava, Czech Republic, MARQ, pp. 91–98, 2006

[14] Janoušek, V., Kironský, E: Exploratory Modeling with SmallDEVS. In: *Proceedings of ESM'2006*, Toulouse, France, 2006

[15] Zeigler, B., P., Moon, Y., Kim, D., Kim, J., G.: DEVS-C++: A High Performance Modeling and Simulation Environment. In: *HICSS '96 Proceedings of the 29th Hawaii International Conference on System Sciences Volume 1: Software Technology and Architecture*, pp. 350–359, IEEE Computer Society, 1996.

[16] Bolduc, J., Vangheluwe, H.: *A Modeling and Simulation Package for Classic Hierarchical DEVS*. Technical Report. Modelling, Simulation & Design Lab, McGill University `http://msdl.cs.mcgill.ca/projects/projects/DEVS`

[17] Fishwick, P.: XML Based Modeling and Simulation: Using XML For Simulation Modeling. In: *Proceedings of the 2002 Winter Simulation Conference*: Exploring New Frontiers 2002, San Diego, California, pp. 616–622, 2002.

[18] Wang, Y. H., Wang, L., H.: *A modeling and simulation example using DEVSW*. Simulation Symposium, 1998, pp. 210–217, IEEE, 1998.

[19] Wang, Y., H., Lu, Y., C.: An XML-based DEVS Modeling Tool to Enhance Simulation Interoperability. In: *Proceedings of 14th European Simulation Symposium and Exhibition*, Dresden, Germany, 2002

[20] Schafer, A. *Visualisierung und XML-Darstellung von DEVS-Modellen*. M.S. Thesis. Fakultät für Informatik, Universität der Bundeswehr München, 2003

[21] Badros, G.: JavaML: A Markup Language for Java Source Code. In: *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, Netherlands, pp. 159–177, 2000

[22] France, R., Evans, A., Lano, K. and Rumpe, B.: The UML as a Formal Modeling Notation. In: *Proceedings of OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, Munich, University of Technology, pp. 75–81, 1997.

[23] Unified Modeling Language `http://www.uml.org`

[24] RelaxNG `http://relaxng.org`

[25] Rábová Z. a kolektiv: *Modelování a simulace*. VUT Brno, 2002, ISBN 80-214-0480-9

[26] Wang, W., Wang, W., Li, Q. and Yang, F.: Ontological, Epistemological, and Teleological Perspectives on Service-Oriented Simulation Frameworks. In: *Ontology, Epistemology, and Teleology for Modeling and Simulation Intelligent Systems Reference Library Volume 44*, pp. 335–358, 2013

[27] Shi, Y., Lu, M., Xiao, M. a Zhang, D.: Research on Service-Oriented and Component-Based Simulation Platform. In: *Emerging Research in Web Information Systems and Mining Communications in Computer and Information Science Volume 238*, pp. 19–27, 2011.

[28] Wang, W.G., Wang, W.P., Zhu, Y.F., Li, Q.: *Service-Oriented Simulation Framework: An Overview and Unifying Methodology*. Simulation 87(3), pp. 221—253, 2011

[29] Thomshon D. : *Application of Service-Oriented Architecture to Distributed Simulation*. Technical paper, The MITRE Corporation, 2008

[30] Coen-Porsini, A., Gallo, I., Zanzi, A.: Integration of web based simulators in the SINPL platform. In: *Proceedings of ESM'2006*, p.259–263, Toulouse, France, 2006

[31] Kuhl, F., Weatherly, R., Dahmann, J.: *Creating computer simulation systems - An introduction to the High Level Architecture* Prentice Hall PTR, 2000

[32] First Year Report *V-Lab - Integrating the Stochastic Learning Automaton Paradigm and Distributed Neuro-Fuzzy Techniques into Advanced Engineering Environments* NASA Ames NRA Grant Number NAG 2-147

[33] Hu, X., Zeigler, B., P.: Model Continuity to Support Software Development for Distributed Robotic Systems: A Team Formation Example. In: *Journal of Intelligent and Robotic Systems*, vol. 39, issue 1, pp. 71–87, 2004

[34] Hu, X., Zeigler, B., P., Couretas, J.: *DEVS-on–a-Chip: Implementing DEVS in Real-Time Java on a Tiny Internet Interface for Scalable Factory Automation.* IEEE International Conference on Systems, Man, And Cybernetics, October, 2001

[35] Petri, C., A.: *Komunikation mit Automaten* (in German). PhD thesis, Institut fur Instrumentelle Mathematik, University Bonn, Germany, 1962.

[36] Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 1: Basic Concepts. In: *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.

[37] Cabac, L., Duvigneau, M., Moldt, D., Rolke H.: Modeling dynamic architectures using nets-within-nets. In: *Proceedings, volume 3536 of Lecture Notes in Computer Science*, pp. 148–167. 26th International Conference, Applications and Theory of Petri Nets 2005, Miami, USA, 2005

[38] Zeigler, B., P., Kim, T., G., Praehofer, H.: *Theory of Modeling and Simulation*, Second Edition. Academic Press, 2000

[39] Johns, K., Taylor, T.: *Professional Microsoft Robotics Developer Studio*, Paperback, Wiley, 2008

[40] Coen-Porsini, A., Gallo, I., Zanzi, A.: Designing and enacting simulations using distributed components. In: *Proceedings of Computer and Information Sciences - ISCIS2004*, 19th International Symposium, Springer, 2004

[41] Multi-Simulation interface: `http://msi.sourceforge.net/`

[42] Dahmann, J., S., Fujimoto, R., M., Weatherly, R., M.: The Department of Defense High Level Architecture. In: *Proceedings of the 1997 Winter Simulation Conference*, 1997

[43] Sklenar, J.: Introduction to OOPN in Simula `http://staff.um.edu.mt/jskl1/talk.html`

[44] Kindler, E.: Reflective Simulation - Simulation of Systems That Simulate. In: *Proceedings of ESM - European Simulation and Modeling Conference*, Porto, Portugal, 2005

[45] Mittal, S., Risco-Martín, J., L., Zeigler, B., P.: DEVSML: automating DEVS execution over SOA towards transparent simulators In: *Proceedings of the 2007 spring simulation multiconference*, March 25-29, Norfolk, Virginia, 2007

[46] Mittal, S., Douglass, S.: DEVSML 2.0: The language and the stack. In: *Proceedings of the Spring Simulation 2012 Multiconference*, Orlando, FL, 2012

[47] PNTalk Project, `http://perchta.fit.vutbr.cz:8000/projekty/12`

[48] PNTalk Project, `http://perchta.fit.vutbr.cz:8000/pntalk2k`

[49] Squeak: `http://www.squeak.org/`

[50] SoapOpera - multi-transport, multi-encoding SOAP with ORB `http://www.mars.dti.ne.jp/ umejava/smalltalk/soapOpera`

[51] PNML Reference site `http://www.pnml.org/`

[52] SOAP: Messaging Framework (Second Edition) `http://www.w3.org/TR/soap12-part1`

[53] WSDL (Web Services Description Language) `http://www.w3.org/TR/wsdl`

[54] W3C - SOAP Message Transmission Optimization Mechanism `http://www.w3.org/TR/2005/REC-soap12-mtom-20050125`

[55] Apache Axis2 - Apache Axis2/Java - Next Generation Web Services `http://axis.apache.org/axis2/java/core`

[56] Web Services - Axis `http://axis.apache.org/axis`

[57] Meta-modelování, ATOM3: `http://msdl.cs.mcgill.ca/`

[58] DEVS Standardization Group: `http://www.sce.carleton.ca/faculty/wainer/standard/`

[59] CORBA (Common Object Request Broker Architecture): `http://www.corba.org`

[60] RMI (Remote Method Invocation): http://en.wikipedia.org/wiki/Java_remote_method_invocation

[61] Windows Azure cloud platform http://www.windowsazure.com

[62] Češka, M., Vojnar, T.: *Petriho sítě*, http://www.fit.vutbr.cz/study/courses/PES/public/Prednasky/PES-05-PT_site.pdf

[63] Ferenc, J.: *Bezpečnost ve službově orientované architektuře*. Brno, 2008. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Dostupné také z: http://is.muni.cz/th/98993/fi_m/dp.pdf

[64] McCarthy, J., Brayton, R., Edwards, D., Fox, P., Hodes, L., Luckham, D., Maling, K., Park, D. et al.: *LISP I Programmers Manual*. Artificial Intelligence Group, M.I.T. Computation Center and Research Laboratory. Boston, 1960

[65] Abstract Window Toolkit http://en.wikipedia.org/wiki/Abstract_Window_Toolkit

[66] Java Swing - http://en.wikipedia.org/wiki/Swing_(Java)

[67] Tuttle, S., Ehlenberger, A., Gorthi, R., et al: *Understanding LDAP - Design and implementation*. IBM, RedBooks, 2014

Publikace autora

[68] Janoušek, V., Polášek, P., Slavíček, P.: Towards DEVS Meta Language. In: *Proceedings of 4th International Industrial Simulation Conference*, University of Palermo, Palermo, Italy, 2006, ISBN 90-77381-26-0

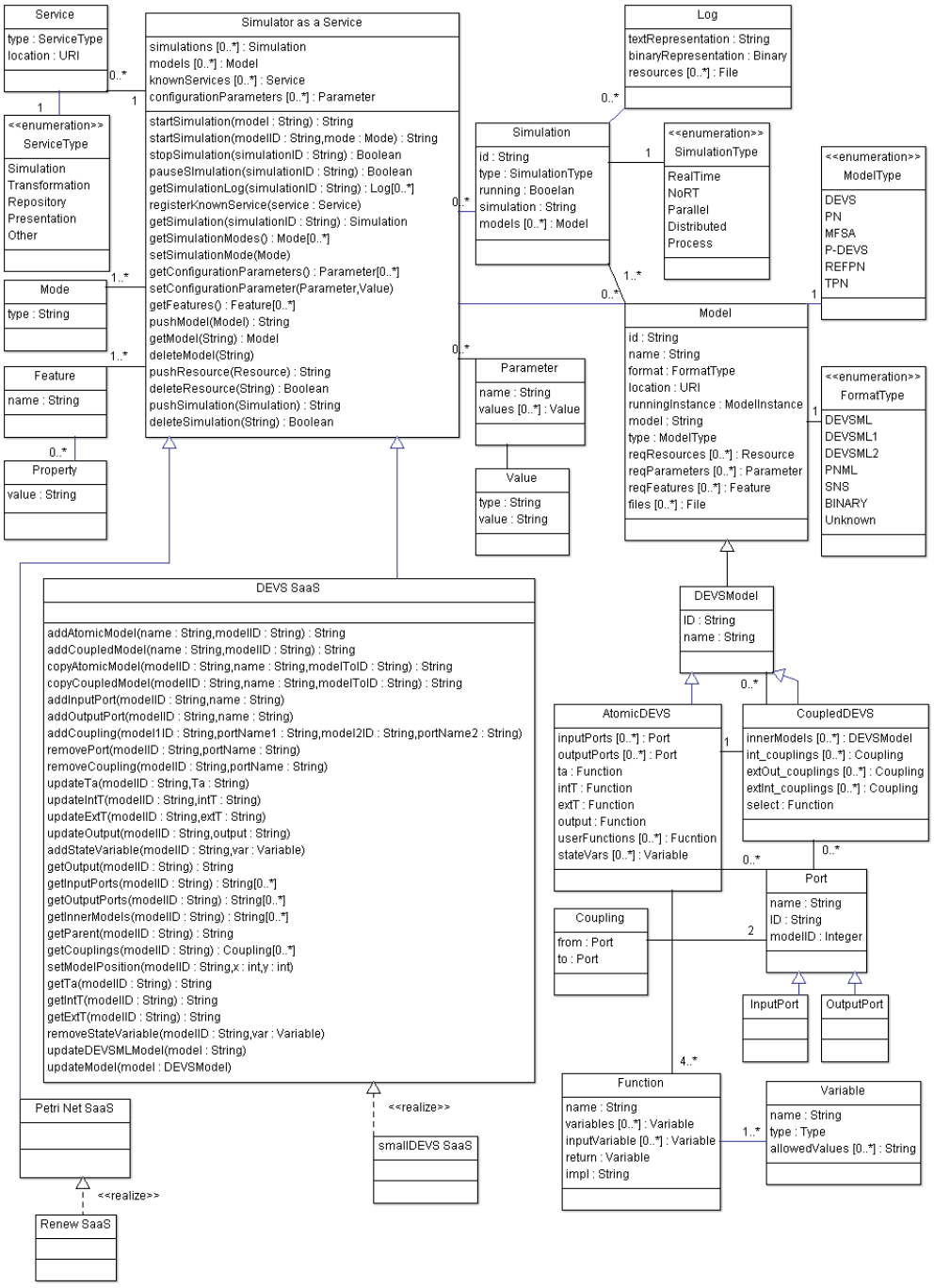
[69] Janoušek, V., Polášek, P., Slavíček, P.: Metajazyk pro popis DEVS formalismu. In: *Proceedings of NETSS*, Ostrava, 2006

[70] Janousek, V., Kironsky, E., Polasek, P.: An Architecture for Simulation-Based Evolutionary Design of Systems, In: *Proceedings of the 16th International Conference on System Science*, pp. 396–405. Wroclaw, Poland, 2007, 978-83-7493-339-1

[71] Janousek, V., Polasek, P.: Modeling and Simulation Management in Distibuted Environment Using Web Services, In: *Proceedings of WOSC 2008 - 14TH International Congress of Cybernetics and Systems*, Wroclaw, 2008

[72] Polasek, P., Janousek, V., Ceska, M.: Petri Net Simulation as a Service, In: *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'14) in Tunis*, Tunisia, pp. 353–362. CEUR-WS.org, Aachen, 2014, ISSN 1613-0073

A Příklad rozhraní služeb



Obrázek 35: Příklad rozhraní simulační služby s diagramem tříd pro implementaci

B Definiční dokument simulační služby

Zde je uveden příklad WSDL dokumentu pro jednoduchou simulační službu s operacemi pro spuštění simulace (*startSimulation*), zastavení simulace (*stopSimulation*) a získání simulačního výstupu (*getLogs*):

```
<wsdl:documentation>SaaS</wsdl:documentation>
<wsdl:types>
  <xs:schema attributeFormDefault="qualified"
             elementFormDefault="qualified"
             targetNamespace="http://SaaS.saas">
    <xs:element name="startSimulation">
      <xs:complexType>
        <xs:element name="model" type="xs:string"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="startSimulationResponse">
      <xs:complexType>
        <xs:element name="return" type="xs:string"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="stopSimulation">
      <xs:complexType>
        <xs:element name="simulationId" type="xs:string"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="stopSimulationResponse">
      <xs:complexType>
        <xs:element name="return" type="xs:boolean"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="getLogs">
      <xs:complexType>
        <xs:element name="simulationId" type="xs:string"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="getLogsResponse">
      <xs:complexType>
        <xs:element name="return" type="xs:string"/>
      </xs:complexType>
    </xs:element>
```

```
  </xs:schema>
</wsdl:types>
<wsdl:message name="stopSimulationRequest">
  <wsdl:part name="parameters"
   element="ns:stopSimulation"/>
</wsdl:message>
<wsdl:message name="stopSimulationResponse">
  <wsdl:part name="parameters"
   element="ns:stopSimulationResponse"/>
</wsdl:message>
<wsdl:message name="getLogsRequest">
  <wsdl:part name="parameters" element="ns:getLogs"/>
</wsdl:message>
<wsdl:message name="getLogsResponse">
  <wsdl:part name="parameters"
   element="ns:getLogsResponse"/>
</wsdl:message>
<wsdl:message name="startSimulationRequest">
  <wsdl:part name="parameters"
   element="ns:startSimulation"/>
</wsdl:message>
<wsdl:message name="startSimulationResponse">
  <wsdl:part name="parameters"
   element="ns:startSimulationResponse"/>
</wsdl:message>
<wsdl:portType name="SaaSPortType">
  <wsdl:operation name="stopSimulation">
    <wsdl:input message="ns:stopSimulationRequest"
     wsaw:Action="urn:stopSimulation"/>
    <wsdl:output message="ns:stopSimulationResponse"
     wsaw:Action="urn:stopSimulationResponse"/>
  </wsdl:operation>
  <wsdl:operation name="getLogs">
    <wsdl:input message="ns:getLogsRequest"
     wsaw:Action="urn:getLogs"/>
    <wsdl:output message="ns:getLogsResponse"
     wsaw:Action="urn:getLogsResponse"/>
  </wsdl:operation>
  <wsdl:operation name="startSimulation">
    <wsdl:input message="ns:startSimulationRequest"
     wsaw:Action="urn:startSimulation"/>
```

```
    <wsdl:output message="ns:startSimulationResponse"
     wsaw:Action="urn:startSimulationResponse"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="SaaSSoapBinding"
 type="ns:SaaSPortType">
  <soap:binding
   transport="http://schemas.soap.org/soap/http"
   style="document"/>
  <wsdl:operation name="stopSimulation">
    <soap:operation soapAction="urn:stopSimulation"
     style="document"/>
    <wsdl:input><soap:body use="literal"/></wsdl:input>
    <wsdl:output><soap:body use="literal"/></wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="getLogs">
    <soap:operation soapAction="urn:getLogs"
     style="document"/>
    <wsdl:input><soap:body use="literal"/></wsdl:input>
    <wsdl:output><soap:body use="literal"/></wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="startSimulation">
    <soap:operation soapAction="urn:startSimulation"
     style="document"/>
    <wsdl:input><soap:body use="literal"/></wsdl:input>
    <wsdl:output><soap:body use="literal"/></wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:binding name="SaaSHttpBinding"
 type="ns:SaaSPortType">
  <http:binding verb="POST"/>
  <wsdl:operation name="stopSimulation">
    <http:operation location="stopSimulation"/>
    <wsdl:input>
      <mime:content type="application/xml"
       part="parameters"/>
    </wsdl:input>
    <wsdl:output>
      <mime:content type="application/xml"
       part="parameters"/>
    </wsdl:output>
```

```
    </wsdl:operation>
    <wsdl:operation name="getLogs">
      <http:operation location="getLogs"/>
      <wsdl:input>
        <mime:content type="application/xml"
         part="parameters"/>
      </wsdl:input>
      <wsdl:output>
        <mime:content type="application/xml"
         part="parameters"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="startSimulation">
      <http:operation location="startSimulation"/>
      <wsdl:input>
        <mime:content type="application/xml"
         part="parameters"/>
      </wsdl:input>
      <wsdl:output>
        <mime:content type="application/xml"
         part="parameters"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="SaaS">
    <wsdl:port name="SaaSSoapEndpoint"
     binding="ns:SaaSSoapBinding">
      <Soap:address
        location="http://srv:8080/SaaS.SaaSSoapEndpoint/"/>
    </wsdl:port>
    <wsdl:port name="SaaSHttpEndpoint"
     binding="ns:SaaSHttpBinding">
      <http:address
        location="http://srv:8080/SaaS.SaaSHttpEndpoint/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

C Jazyk DEVSML

C.1 Příklady jednoduchých modelů

Příklad jednoduchého modelu

```
<model type="DEVSML">
  <name>Sample model</name>
  <description>This is a sample model</description>
  <authors>
    <author>
      <name>Petr Polášek</name>
      <email>polasek@fit.vutbr.cz</email>
    </author>
  </authors>
  <root-model>
   <source>http://devsml/repository/sampleCoupled.xml</source>
  </root-model>
</model>
```

Příklad spojované komponenty

```
<coupled name="sampleCoupled" modelX="16" modelY="22">
  <ports>
    <input>
      <port name="in1"/>
    </input>
    <output>
      <port name="out1"/>
    </output>
  </ports>
  <D> <component name="sampleAtomic"
          source="http://devsml/repository/sampleAtomic.xml"
          modelX="29" modelY="32"/>
  </D>
  <influences>
    <influence source="self" source-port="in1"
          target="sampleAtomic" target-port="inAtomic1"/>
    <influence source="sampleAtomic" source-port="outAtomic1"
          target="self" target-port="out1"/>
  </influences>
</coupled>
```

Příklad atomické komponenty

```
<atomic name="sampleAtomic" modelX="26" modelY="28">
  <state-variables>
   <state-variable name="a" type="integer" initial-value="2"/>
  </state-variables>
  <ports>
    <input>
      <port name="inAtomic1"/>
    </input>
    <output>
      <port name="outAtomic1"/>
    </output>
  </ports>
  <ta>...</ta>
  <internal-transition-function>...
  </internal-transition-function>
  <external-transition-function>...
  </external-transition-function>
  <output-function>...</output-function>
  <functions>
    <function name="init" type="void">...</function>
    <function name="reset" type="void">...</function>
    <function name="userFunction" type="integer">
      <parameters>
        <parameter name="parameter1" type="integer"/>
      </parameters>
      <block>
        <if>
          <test>
            <binary-expr op=">">
            <var-ref name="parameter1"/>
            <literal-number value="10"/>
            </binary-expr>
          </test>
          <true-case>...</true-case>
        </if>
      </block>
    </function>
  </functions>
</atomic>
```

C.2 Model procesoru

Jednoduchý procesor

V následujícím textu je ukázán model jednoduchého procesoru popsaný různými způsoby. Nejprve v jazyce DEVSML, poté ve Smalltalku a na závěr v jazce Java. Model ve Smalltalku, resp. v Javě je určený pro simulační prostředí SmallDEVS, resp. DEVSJava. Tento model je jednodušší varianta modelu procesoru ukázaného v kapitole 6.3.1, určená pro zpracování jednoho typu úloh.

Model procesoru (DEVSML)

```
<atomic name="Processor" modelX="34" modelY="56">
  <state-variables>
    <state-variable name="job" type="string"
     initial-value="none"/>
    <state-variable name="phase" type="string"
     initial-value="passive"/>
    <state-variable name="procTime" type="double"
     initial-value="5"/>
  </state-variables>
  <ports>
    <input>
      <port name="in"/>
    </input>
    <output>
      <port name="out"/>
    </output>
  </ports>
  <ta>
    <time-def>
      <statevar-ref name="phase"/>
      <literal-string>busy</literal-string>
      <statevar-ref name="procTime"/>
    </time-def>
  </ta>
  <internal-transition-function>
    <set-state-var name="job">
      <literal-string name="none"/>
    </set-state-var>
    <set-state-var name="phase">
```

```
      <literal-string name="passive"/>
    </set-state-var>
  </internal-transition-function>
  <external-transition-function>
    <function-call name="continue">
      <arguments>
        <var-ref name="elapsed"/>
      </arguments>
    </send>
    <if>
      <test>
        <binary-expr op="=">
          <statevar-ref name="phase"/>
          <literal-string>passive</literal-string>
        </binary-expr>
      </test>
      <true-case>
        <set-state-var name="job">
          <input name="in">
        </set-state-var>
        <set-state-var name="phase">
          <literal-string>busy</literal-string>
        </set-state-var>
      </true-case>
    </if>
  </external-transition-function>
  <output-function>
    <output name="out">
      <statevar-ref name="job"/>
    </output>
  </output-function>
  <functions></functions>
</atomic>
```

Model procesoru (SmallDEVS)

```
(AtomicDEVSPrototype new
 instVarNamed: 'name' put: #Processor;
 instVarNamed: 'timeLast' put: 0;
 instVarNamed: 'timeNext' put: (Float infinity);
 instVarNamed: 'verbose' put: true;
 instVarNamed: 'elapsed' put: 0;
 instVarNamed: 'savedTimeAdvance' put: nil;
 removeAllPorts;
 addInputPortOfType: InputPort named: #in;
 addOutputPortOfType: OutputPort named: #out;
addSlot: #job withValue: 'none';
addSlot: #name withValue: 'Processor';
addSlot: #phase withValue: 'passive';
addSlot: #procTime withValue: 5;
addSlot: #sigma withValue: (Float infinity);
addMethod: 'extTransition
  self sigma: (self sigma - self elapsed).
  self phase = ''passive'' ifTrue: [
    self job: (self peekFrom: #in).
    self phase: ''busy''.
    self sigma: (self procTime)].';
addMethod: 'intTransition
  self job: ''none''.
  self phase: ''passive''.
  self sigma: (Float infinity).';
addMethod: 'outputFnc
  self poke: (self job) to: #out.';
addMethod: 'prepareToStart';
addMethod: 'prepareToStop';
addMethod: 'timeAdvance
  ^self sigma.';
 yourself)
```

Model procesoru (DEVSJava)

```
public class Processor extends Atomic {
  protected String job;
  protected double procTime;
```

```
  public Proc(String name){
    super(name);
    addInport("in");
    addOutport("out");
    phases.add("busy");
    procTime = 5;
    sigma = INFINITY;
  }

  public void initialize(){
    phase = "passive";
    job = new Entity("none");
    super.initialize();
  }

  public void deltext(double e, Message x) {
    continue(e);
    if (phaseIs("passive")) {
      for (int i = 0; i < x.getLength(); i++) {
        if (messageOnPort(x, "in", i)) {
          Entity val = x.getValOnPort("in", i);
          job = (String) val;
          holdIn("busy", procTime);
        }
      }
    }
  }

  public void deltint( ) {
    passivate();
    job = new Entity("none");
  }

  public Message out( ) {
    Message m = new Message();
    if (phaseIs("busy")) {
      m.add(makeContent("out", job));
    }
    return m;
  }
}
```

C.3 Popis struktury DEVSML

Definice modelu

```
<!ELEMENT model (name, description, authors*, root-model)>
<!ATTLIST model type (DEVSML|DEVSML1|DEVSML2) #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT authors (author+)>
<!ELEMENT author (name, email?, institute?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT institute (#PCDATA)>
<!ELEMENT root-model (source)>
<!ELEMENT source (#PCDATA)>
```

Definice spojované komponenty

```
<!ELEMENT coupled (description?, ports, D, influences)>
<!ATTLIST coupled name CDATA #REQUIRED>
<!ATTLIST coupled modelX CDATA #REQUIRED>
<!ATTLIST coupled modelY CDATA #REQUIRED>
<!ELEMENT description (#PCDATA)>
<!ELEMENT ports (input*, output*)>
<!ELEMENT input (port*)>
<!ELEMENT output(port*)>
<!ELEMENT D (component*)>
<!ELEMENT influences (influence*)>
<!ELEMENT port EMPTY>
<!ATTLIST port name CDATA #REQUIRED>
<!ELEMENT component EMPTY>
<!ATTLIST component name CDATA #REQUIRED>
<!ATTLIST component source CDATA #REQUIRED>
<!ATTLIST component modelX CDATA #REQUIRED>
<!ATTLIST component modelY CDATA #REQUIRED>
<!ELEMENT influence EMPTY>
<!ATTLIST influence source CDATA #REQUIRED>
<!ATTLIST influence source-port CDATA #REQUIRED>
<!ATTLIST influence target CDATA #REQUIRED>
<!ATTLIST influence target-port CDATA #REQUIRED>
```

Definice atomické komponenty

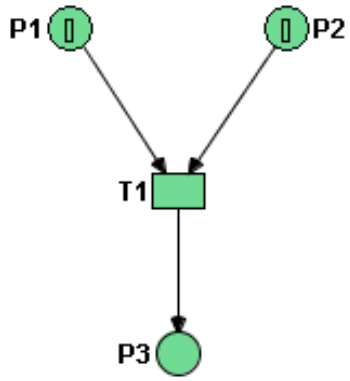
```
<!ELEMENT atomic (description?, state-variables, ports, ta,
internal-transition-function, external-transition-function,
output-function, functions?)>
<!ATTLIST coupled name CDATA #REQUIRED>
<!ATTLIST coupled modelX CDATA #REQUIRED>
<!ATTLIST coupled modelY CDATA #REQUIRED>
<!ATTLIST coupled super-model CDATA>
<!ELEMENT state-variables (state-variable*)>
<!ELEMENT ports (input*, output*)>
<!ELEMENT input (port*)>
<!ELEMENT output(port*)>
<!ELEMENT time-def (statevar-ref*,
(literal-string|literal-number|literal-boolean),
(statevar-ref|literal-number)>
<!ELEMENT ta (time-def*|(%stmt-elements;)*)>
<!ELEMENT internal-transition-function (%stmt-elements;)*>
<!ELEMENT external-transition-function (%stmt-elements;)*>
<!ELEMENT output-function (%stmt-elements;)*>
<!ELEMENT functions (function*)>
<!ELEMENT state-variable EMPTY>
<!ATTLIST state-variable name CDATA #REQUIRED>
<!ATTLIST state-variable type (integer|string|double|boolean)
"integer">
<!ATTLIST state-variable initial-value CDATA>
<!ATTLIST state-variable visibility (private|protected)
"private">
<!ELEMENT port EMPTY>
<!ATTLIST port name CDATA #REQUIRED>
<!ELEMENT function (parameters?,block)>
<!ATTLIST function name CDATA #REQUIRED>
<!ATTLIST function type (integer|string|real|boolean)
#REQUIRED>
<!ATTLIST function visibility (private|protected) "private">
<!ELEMENT parameters (parameter*)>
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter name CDATA #REQUIRED>
<!ATTLIST parameter type (integer|string|real|boolean)
#REQUIRED>
```

Definice funkcí

```
<!ELEMENT block ((%stmt-elements;)*)>
<!ENTITY % expr-elements "var-ref|conditional-expr|
binary-expr|unary-expr|literal-number|literal-string|
literal-boolean|literal-null|input|output|statevar-ref">
<!ENTITY % stmt-elements "block|local-variable|if|loop|
do-loop|return|function-call|continue|break|%expr-elements;">
<!ENTITY % kind-attribute "kind (integer|double) #IMPLIED">
<!ELEMENT local-variable (type,(%expr-elements;)?)>
<!ATTLIST local-variable name CDATA #REQUIRED>
<!ELEMENT var-ref EMPTY>
<!ATTLIST var-ref name CDATA #REQUIRED>
<!ELEMENT statevar-ref EMPTY>
<!ATTLIST statevar-ref name CDATA #REQUIRED>
<!ELEMENT var-set EMPTY>
<!ATTLIST var-set name CDATA #REQUIRED>
<!ELEMENT statevar-set EMPTY>
<!ATTLIST statevar-set name CDATA #REQUIRED>
<!ELEMENT assignment-expr (lvalue,(%expr-elements;))>
<!ELEMENT lvalue (var-set|statevar-ser)>
<!ELEMENT binary-expr ((%expr-elements;),(%expr-elements;))>
<!ATTLIST binary-expr op CDATA #REQUIRED>
<!ELEMENT unary-expr (%expr-elements;)>
<!ATTLIST unary-expr op CDATA #REQUIRED post (true|false)
#IMPLIED>
<!ELEMENT literal-boolean EMPTY>
<!ATTLIST literal-boolean value (true|false) #REQUIRED>
<!ELEMENT literal-null EMPTY>
<!ELEMENT literal-number EMPTY>
<!ATTLIST literal-number value CDATA #REQUIRED
%kind-attribute;>
<!ELEMENT literal-string (#PCDATA)>
<!ATTLIST literal-string length CDATA #REQUIRED>
<!ELEMENT conditional-expr ((%expr-elements;),
(%expr-elements;),(%expr-elements;))>
<!ELEMENT if (test,true-case,false-case?)>
<!ELEMENT test (%expr-elements;)>
<!ELEMENT true-case (%stmt-elements;)*>
<!ELEMENT false-case (%stmt-elements;)*>
<!ELEMENT loop (init*,test?,update*,(%stmt-elements;)?)>
```

```
<!ATTLIST loop kind (for|while) #IMPLIED>
<!ELEMENT init (local-variable|%expr-elements;)*>
<!ELEMENT update (%expr-elements;)>
<!ELEMENT do-loop ((%stmt-elements;)?,test?)>
<!ELEMENT continue EMPTY>
<!ELEMENT break EMPTY>
<!ELEMENT return (%expr-elements;)?>
<!ELEMENT function-call (arguments)?>
<!ATTLIST function-call name CDATA #REQUIRED>
<!ELEMENT arguments (argument)?>
<!ELEMENT argument (%expr-elements;)?>
<!ELEMENT input EMPTY>
<!ATTLIST input name CDATA #REQUIRED>
<!ELEMENT output (var-ref)>
<!ATTLIST output name CDATA #REQUIRED>
<!ELEMENT set-state-var (%expr-elements;)>
<!ATTLIST set-state-var name CDATA #REQUIRED>
```

D PNML reprezentace jednoduché Petriho sítě



Obrázek 36: Jednoduchá Petriho síť

Jednoduchá Petriho síť popsaná v jayzce PNML

```
<pnml xmlns="RefNet">
   <net id="netId1408634100372" type="RefNet">
      <place id="3">
         <initialMarking>
            <graphics>
               <offset x="0" y="0"/>
            </graphics>
            <text>[]</text>
         </initialMarking>
         <name>
            <graphics>
               <offset x="-20" y="0"/>
            </graphics>
            <text>P1</text>
         </name>
         <graphics>
            <position x="90" y="75"/>
            <dimension x="20" y="20"/>
            <fill color="rgb(112,219,147)"/>
            <line color="rgb(0,0,0)"/>
         </graphics>
      </place>
```

```
<place id="4">
   <initialMarking>
      <graphics>
         <offset x="0" y="0"/>
      </graphics>
      <text>[]</text>
   </initialMarking>
   <name>
      <graphics>
         <offset x="20" y="0"/>
      </graphics>
      <text>P2</text>
   </name>
   <graphics>
      <position x="190" y="75"/>
      <dimension x="20" y="20"/>
      <fill color="rgb(112,219,147)"/>
      <line color="rgb(0,0,0)"/>
   </graphics>
</place>
<arc id="6" source="3" target="7">
   <type>
      <text>ordinary</text>
   </type>
   <graphics>
      <line color="rgb(0,0,0)" style="solid"/>
   </graphics>
</arc>
<transition id="7">
   <name>
      <graphics>
         <offset x="-20" y="0"/>
      </graphics>
      <text>T1</text>
   </name>
   <graphics>
      <position x="140" y="150"/>
      <dimension x="24" y="16"/>
      <fill color="rgb(112,219,147)"/>
      <line color="rgb(0,0,0)"/>
   </graphics>
```

```
      </transition>
      <arc id="8" source="4" target="7">
         <type>
            <text>ordinary</text>
         </type>
         <graphics>
            <line color="rgb(0,0,0)" style="solid"/>
         </graphics>
      </arc>
      <arc id="9" source="7" target="10">
         <type>
            <text>ordinary</text>
         </type>
         <graphics>
            <line color="rgb(0,0,0)" style="solid"/>
         </graphics>
      </arc>
      <place id="10">
         <name>
            <graphics>
               <offset x="-20" y="0"/>
            </graphics>
            <text>P3</text>
         </name>
         <graphics>
            <position x="140" y="225"/>
            <dimension x="20" y="20"/>
            <fill color="rgb(112,219,147)"/>
            <line color="rgb(0,0,0)"/>
         </graphics>
      </place>
      <name>
         <text>simpleNet</text>
      </name>
   </net>
</pnml>
```