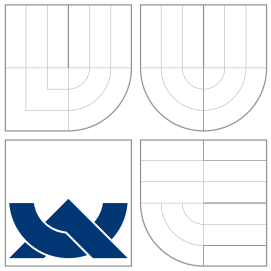
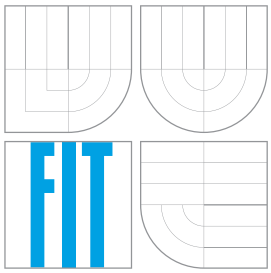


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SIMULAČNÍ ARCHITEKTURA ZALOŽENÁ NA SLUŽBÁCH

SERVICE-ORIENTED SIMULATON ARCHITECTURE

TEZE DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE
AUTHOR

Ing. PETR POLÁŠEK

VEDOUCÍ PRÁCE
SUPERVISOR

Prof. RNDr. MILAN ČEŠKA, CSc.

BRNO 2014

Obsah

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 narážime 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 budeme klást 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 [13], [14], [15] a [16].

1 Systémy s diskrétními událostmi

Systémy založené na diskrétních událostech se zdají být vhodnou abstrakcí pro modelování dynamických a i inteligentních 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 [40].

V následujících kapitolách si představíme a krátce popíšeme dva forma-

lismy, jako vhodné prostředky pro popis systémů založených na diskrétních událostech.

1.1 DEVS formalismus

DEVS (Discrete EVent Systems) formalismus tvoří prostředek pro univerzální a rigorózní popis diskrétních modelů [28]. 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

- 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
- $select : 2^D \to D$ je funkce výběru v případě výskytu souběžných událostí

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.

V [28] je prokázáno, že DEVS může být modifikován nebo přímo použit i 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 [28]. Takové systémy lze simulovat buď naprosto věrně, nebo je možné jejich chování simulovat s požadovanou přesností.

1.2 Petriho sítě

Petriho sítě 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 $\mathbb{N}$ [41] 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 zvýšení modelovací schopnosti, úrovně popisu a přiblížení se tak co nejblíže modelovanému systému. Jako příklad můžeme uvést *C/E sítě* [25], *P/T sítě*, či *vysokoúrovňové sítě*, jako jsou např. *barvené sítě* [26].

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* [27]. 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.

2 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.

Mezi základní vlastnosti služeb patří: *jasný kontrakt*, *nezávislost*, *znovupoužitelnost* a *jednoznačná identifikace*. Dále také musí každá ze služeb splňovat tzv. *princip abstrakce*, kdterý určuje aby implementační detaily byly skryty, což umožňuje volné vazby na své okolí. Princip abstrakce zvyšuje granularitu systému a usnadňuje jeho administraci a úpravu.

V architektuře orientované na služby existují dvě základní entity, kterými jsou *poskytovatel služby (service provider)* a *klient/konzument (service consumer)*. Poskytovatelem služby je entita, která nabízí svému okolí nějaké předem definované služby. Klientem je pak entita, která tyto služby využívá.

Komunikace mezi nimi pak probíhá podle předem daného standardu, je založena na principu vzdáleného volání procedur a je typu *klient-server*. Základem je dvojice zpráv: *požadavek (request)* generovaný klientem a *odpověď (response)* odesílaná poskytovatelem služby.

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 síťi 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 používá protokolu *SOAP (Sime Object Access Protocol)* [34], který je založen na jazyce *XML (eXtensible Markup Language)*. V tomto jazyce je popsané i rozhraní služeb, konkrétně se jedná o speciální jazyk *WSDL (Web Service Description Language)* [35], který definuje veřejné operace poskytované službou a datové struktury pro výměnu dat.

Existující architektury pro modelování a simulaci Jako existující architektury pro vzdálenou a distribuovanou simulaci si můžeme uvést *Microsoft Robotic Studio* [29], *Simulator Integration Platform (SINPL)* [20], *High Level Architecture (HLA)* [21], [30] a *Multi-Simulation Interface (MSI)* [23].

Tyto architektury byly shledány buď jako 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. Integraci různých simulačních nástrojů tyto architektury spíše nepodporují. Obecně se tak dá říci, že koncept nezávislých služeb v dnešních simulačních architekturách chybí nebo je využíván jen částečně.

3 Simulační architektura založená na službách

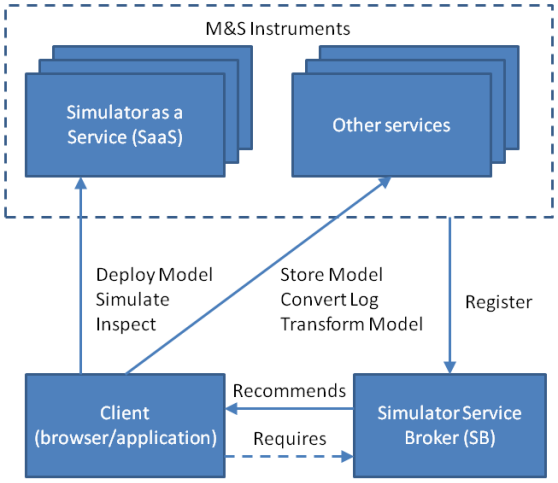
Navrhovaná simulační architektura je založená na webových službách, kde webovou službou rozumíme diskrétní aplikaci, která poskytuje služby v počítačové síti nebo přes internet.

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, 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.

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 1, 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 1: 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.

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

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 mode-

lovací 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.

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 jednoduchý požadavek na spuštění simulace. Vlastní model může být ve zprávě poslán buď přímo jako součást XML zprávy nebo přes tzv. *přílohy (SOAP attachments)*, popřípadě je možné použít *Message Transmission Optimization Mechanism (MTOM)* [37]. Další možností je zaslání 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.

Příklad komunikace mezi klientem a službou

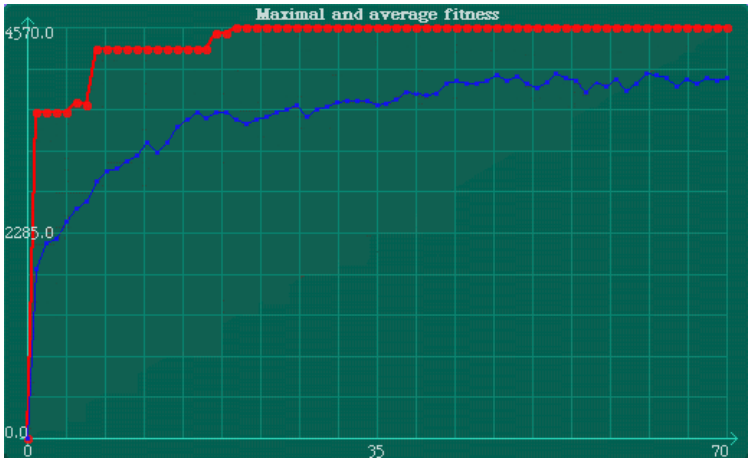
```
Požadavek na spuštění simulace:

<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>
```

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 5), který byl vytvořen a simulován v prostředí



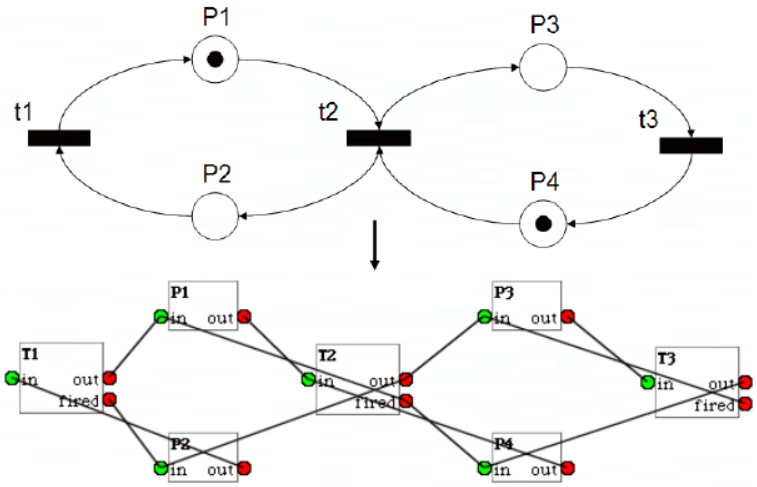
Obrázek 2: Grafická reprezentace vytvořená z textového simulačního výstupu doručeného prezentační službě

SmallDEVS [36]. Výstup obsahuje maximální a průměrnou hodnotu tzv. *fitness funkce*, která vyjadřuje kvalitu řešení reprezentovaného tímto jedincem pro každou generaci vytvořenou během simulace. Na obrázku 2 pak můžeme vidět grafickou podobu textového simulačního výstupu, tak jak ji vytvořila jednoduchá prezentační služba běžící v prostředí Squeak [38] s rozšířením SoapOpera [39]. 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>
  <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>
  ...
</trace>
```

Jednou z dalších důležitých služeb, která 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

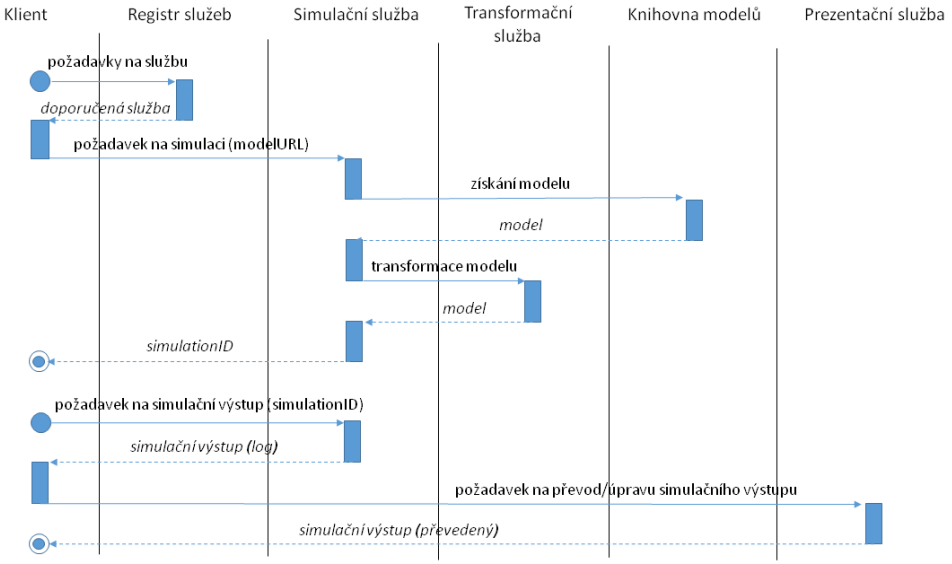


Obrázek 3: Transformace modelu popsaného Petriho sítí na model v klasickém DEVS formalismu

jeho následné simulaci. Příklad takové transformace je ukázán na obrázku 3, kde jednoduchý model vyjádřený Petriho sítí je převeden na model v klasickém DEVS formalismu [28]. V tomto případě transformační služba použila algoritmus převodu, který je popsán v kapitole 3.2.2. Obrázek je z prostředí SmallDEVS, kde pak takový převedený model může být spuštěn a simulován.

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.

Příklad použití výše popsaných služeb pro spuštění simulace a získání simulačního výstupu je ukázán na obrázku 4. 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. Po převedení modelu již simulační služba může začít provádět simulaci a v odpovědi pošle klientovi přidělený identifikátor běžící



Obrázek 4: Příklad využití služeb pro simulaci

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.

Distribuovaná simulace Distribuovanou simulaci 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. 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.

3.1 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 pouze těchto modelů. V případě Petriho sítí takový prostředek již existuje, avšak pro DEVS bylo nutné jej vytvořit. Vznikl tak *DEVS Meta Language* [42], který se stal předlohou (viz [19]) k *DEVS Modeling Language* [18], což je v současné době používaný standard.

3.1.1 DEVS Meta Language

Jazyk DEVSML (DEVS Meta Language) [42] je určený pro popis simulačních DEVS modelů a jeho hlavním cílem je jejich univerzální popis, založený na standardech XML a JavaML. Díky tomu jsou takové modely jednoduše znovupoužitelné v různých simulačních prostředích, bez výraznějších změn či složitých manuálních úprav. Takový model je nezávislý na konkrétní implementaci simulačního nástroje a může být jednoduše, například za pomocí XSL transformace, převeden a simulován v libovolném simulačním nástroji. V navrhované architektuře hraje DEVSML roli jednoho z hlavních výrazových prostředku pro výměnu informací mezi simulačními službami a vzdáleným klientem.

Struktura modelu Díky použití jazyka XML je hierarchická struktura modelu vyjádřena zcela přirozeně. Atomické a spojované komponenty jsou popsány odděleně. Popis spojované DEVS komponentu 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 uveden níže.

Příklad spojované komponenty

```
<coupled name="sampleCoupled" modelX="16" modelY="22">
  <ports>
    <input>
      <port name="in1"/>
    </input>
    <output>
      <port name="out1"/>
```

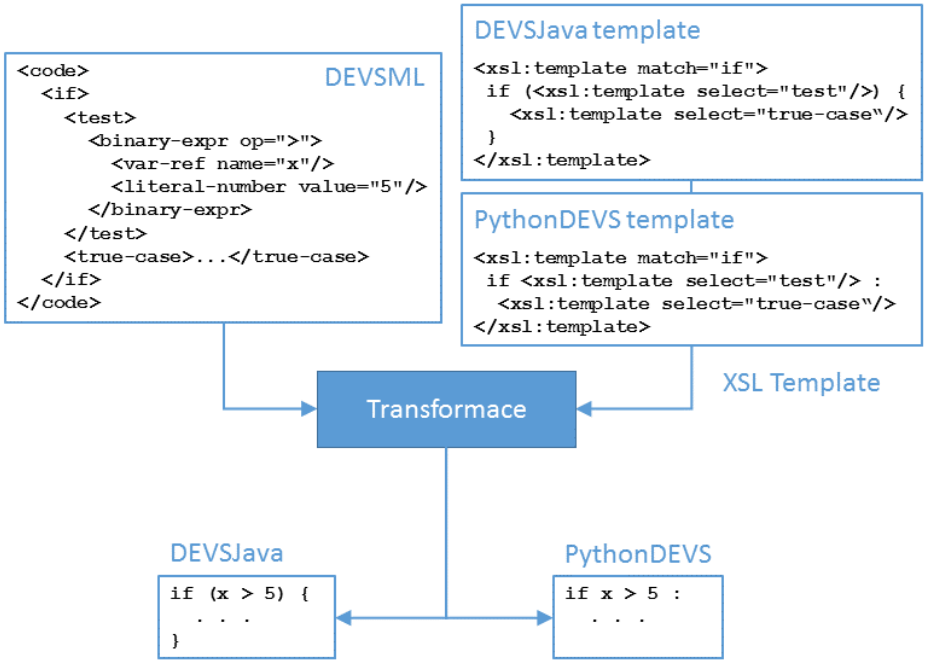
```
    </output>
  </ports>
  <D> <component name="sampleAtomic"
              source="http://devsml/repository/sampleAtomic.xml"
              modelX="29" modelY="32"/>
  </D>
  <influences>
    <influence source="self" source-port="in1"
              target="sampleAtomic" target-port="inAtomic1"/>
    <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>
  </functions>
</atomic>
```

Popis chování modelu Popis struktury atomických a spojovaných komponent je celkem jednoduchý. To však neplatí pro popis chování atomických komponent určené čtyřmi základními funkcemi, které je potřeba specifikovat. Tyto funkce mohou používat a měnit hodnoty tzv. stavových proměnných. Hodnoty všech stavových proměnných určují stav modelu.

Pro definici všech čtyř funkcí atomických komponent čerpá DEVSML inspiraci od JavaML [10], který přináší alternativu zápisu Java kódu v jazyce



Obrázek 5: XSLT Transformace

XML. Snadno je tak možné převést kód do jiného programovacího jazyka a díky tomu je možné tento kód optimalizovat a verifikovat. JavaML představuje výrazový prostředek pro popis chování atomické komponenty a jejích základních funkcí.

Základní syntaktické elementy umožňují přečíst a nastavit hodnotu stavové proměnné, přečíst a nastavit hodnotu lokální proměnné, dále umožňují zápis podmíněného větvení a různé druhy cyklů - s podmínkou na konci, resp. na začátku a s definovaným počtem průchodů. Pro výrazy pak je možné použít číselné a řetězcové hodnoty, unární a binární operátory.

Pro definici chování atomických komponent je možné také použít jiný jednoduchý jazyk nebo pseudo kód. Může jím být např. jazyk podobný Lispu, kterým je možné specifikovat funkce modelu. Další možností je použití vizuálních jazyků, jako je např. diagram aktivit v UML [11], [12].

Převod modelu do cílového simulačního prostředí Protože DEVSML popisuje, jak strukturu jednotlivých komponent modelu, tak i jejich chování, můžeme jej použít 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) definující samotný převod. Princip tohoto převodu je zobrazen na obrázku 5. Pro každý simulační nástroj nebo platformu je tedy nutné vytvořit pouze nový XSL předpis, který

je použit pro transformaci, a modely mohou zůstat nezměněné.

Pro srovnání s DEVSML si můžeme uvést podobné projekty, které se také snaží o standardizaci popisu DEVS modelů [24]. Jedním je DEVSW [7], [8], který sice popisuje strukturu modelů pomocí XML, ale popis chování atomických komponent je možné specifikovat pouze pomocí pseudokódu. Elegantní řešení také bylo představeno v [9], které sice používá jen 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. Dalším je DEVS Modeling Language [18], který navazuje na DEVSML [19], existuje ve dvou verzích a má v současné době největší tendence stát se používaným standardem.

3.1.2 PNML

Perti Net Meta Language (PNML) [17] je metajazyk pro popis modelů založených na Petriho sítích. Jeho hlavním účelem je umožnit interoperabilitu mezi různými modelovacími nástroji pro modely založené na Petriho sítích. Díky tomuto standardu je možné tyto modely přenášet mezi různými nástroji.

PNML standard se skládá ze dvou částí. *Sémantická část* definuje sémantický model Petriho sítí. Matematické definice jsou jeho součástí. *Syntaktická část* definuje abstraktní a konkrétní syntaxi pro různé typy Petriho sítí. Abstraktní syntaxe je specifikovaná pomocí UML a konkrétně je pak síť popsána pomocí RELAX NG [31].

3.2 Transformace modelů

Pokud chceme používat různé modelovací formalismy (tzv. multiparadigmatický přístup), je potřeba mít k dispozici možnosti převodu mezi nimi. Představíme si zde možnosti převodu mezi DEVS, Petriho sítěmi a konečnými automaty. Všechny níže uvedené postupy mohou být chápány jako návod pro implementaci algoritmu převodu pro *transformační službu*, která tvoří jednu ze stěžejních komponent celé simulační architektury. S touto službou je pak možné realizovat převody modelů zcela automaticky, přičemž transformační postup se dá snadno verifikovat za pomocí simulace. Výsledky simulace mohou totiž ukázat, zda chování původního a převedeného modelu je totožné.

3.2.1 DEVS a konečné automaty

Chování atomických komponent v DEVS formalismu je vyjádřeno pomocí čtyř základních funkcí. Tyto odlišné funkce nám určují dohromady chování modelu a často není snadné z určit závislosti mezi nimi navzájem a pochopit tak chování modelu. Vhodnějším 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 $p \in OutputPorts$.

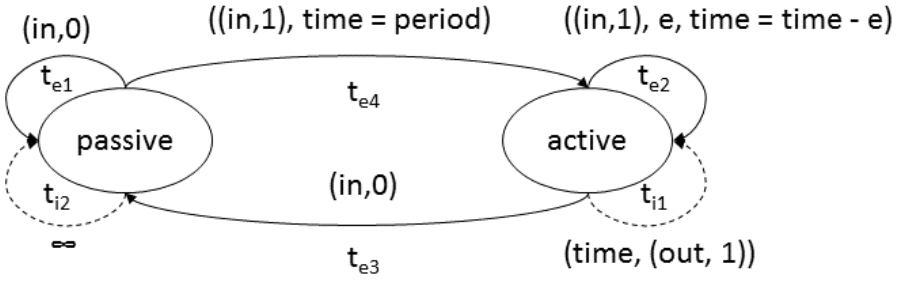
Modifikované konečné automaty pro popis chování DEVS modelů si v tuto chvíli můžeme definovat následujícím způsobem:
$FSA = (P, T_{ext}, T_{int}, W_{ext}, W_{int})$, kde

- $P = \{phase_1, phase_2, .., phase_n\}$ 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,



Obrázek 6: Modifikovaný konečný automat, který popisuje názorně chování generátoru s přepínačem

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: $A_z : V_g \rightarrow V_G$, 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 $=$.

- $W_{int} : T_{int} \rightarrow G \times T \times 2^Y \times A$, 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).

Pro ilustraci si uvedeme případ jednoduchého DEVS modelu, jehož chování vyjádříme pomocí modifikovaného konečného automatu a ukžeme si jeho grafickou reprezentaci:

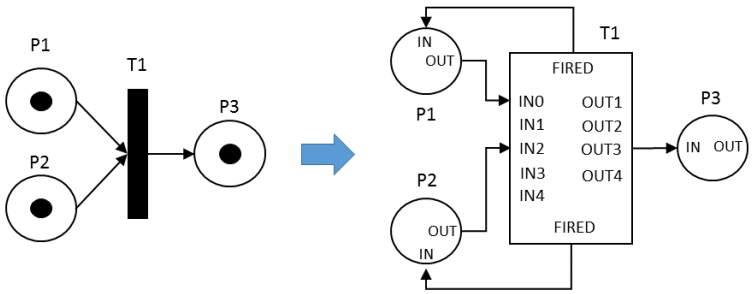
Generátor s přepínačem Jednoduchý generátor pracuje ve dvou módech: *passive* and *active*. Když je v módu *passive*, nevykonává žádnou činnost, pouze čeká na aktivaci přes vstupní port. V módu *active* generuje v pravidelných časových intervalech, daných periodou *period* výstup 1. Generátor v aktivní fázi (tedy v módu *active*) může být vypnut posláním 0 na vstupní port. Model je inspirován modelem uvedeném v [28], kde však chybí přepínač.

Grafická reprezentace chování pomocí modifikovaných konečných automatů je ukázána na obrázku 6. 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 si pak 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
- $W_{ext}(t_{e1}) = (true, (in, 0), e, noAction)$
  $W_{ext}(t_{e2}) = (true, (in, 1), e, (time = time - e))$
  $W_{ext}(t_{e3}) = (true, (in, 0), e, noAction)$
  $W_{ext}(t_{e4}) = (true, (in, 1), e, (time = period))$

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 6) 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.

V [28] je definován *DEVS Definition Language* který používá tzv. *přechodové diagramy (transition diagrams)*. Ty však neumožňují definovat funkci časového posunu t_a.



Obrázek 7: Příklad Petriho sítě a odpovídajícího DEVS modelu

3.2.2 DEVS a Petriho sítě

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 [5] 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.

Hlavní princip převodu spočívá v nalezení vhodné reprezentace pro místa, přechody a tokovou relaci v Petriho síti. Místa a přechody mohou být vyjádřeny atomickými modely a toková relace je realizována propojením jejich vstupů a výstupů. Příklad je na obrázku 7.

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. Což 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, protože současně vyvolané výstupní funkce u dvou komponent by byly simulátorem zpracovávány sekvenčně a 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.

3.3 Integrace nástrojů a aplikací

Pro integraci existujících simulačních aplikací a nástrojů do simulační architektruy je potřeba vytvořit rozhraní webové služby a implementovat ji. Teprve tehdy mohou klientské aplikace používat funkcionalitu těchto nástrojů,

které mohou být snadno zahrnuty do registru služeb. Jako základní platformu pro webové služby a jejich implementaci, včetně klientských aplikací, je vhodné použít nástroj *Apache Axis2* [32]. Ten 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.

3.4 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, je velmi podobná tzv. cloudovému prostředí.

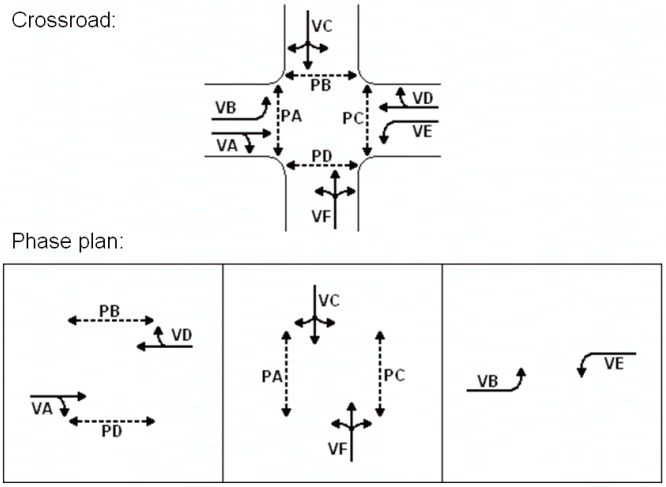
Samotný termín *cloud* se používá v souvislosti s ukládáním dat nebo vzdáleným výpočetním výkonem či vzdálenou infrastrukturou. Cloudové prostředí bychom si mohli charakterizovat jako vzdálenou platformu, která nám poskytuje nějakou službu či aplikaci, která neběží lokálně na našem počítači, ale je celá (či z velké části) umístěna někde na internetu nebo ve vzdálených datových centrech. Mezi hlavní výhody tohoto přístupu patří: *škálovatelnost (rozšiřitelnost)*, *neomezený vzdálený přístup* ke službám bez závislosti na použité platformě, *jednoduché klientské aplikace* s bohatými možnostmi a funkcemi a *snadná údržba*, instalace, konfigurace a ovládání. Všechny výhody cloudového prostředí a aplikací patří i mezi námi stanovené požadavky na simulační architekturu. Takové prostředí je vhodnou platformou pro umístění základních komponent celé architektury, tedy webových služeb.

4 Případová studie - systém pro řízení dopravy

V této případové studii byl navržen systém pro řízení dopravy na jednoduché křižovatce, zobrazené na obrázku 8. Tento systém je schopen reagovat na dopravní situaci a podle potřeby modifikovat své vlastní nastavení (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 [6], přičemž model byl dále rozšířen za účelem umožnění reflektivní simulace a následné rekonfigurace. Ostatní modely - model dopravy a model řízení simulace (na obrázku 9) byly vytvořeny autorem této práce.

Klíčovou aplikací pro tuto případovou studii je simulační nástroj *Renew* [4], [3]. Těsné provázání jazyku Java a Petriho sítí s referencemi nám



Obrázek 8: Schéma křižovatky a její fázový plán průjezdu (obrázek byl převzat z [6])

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.

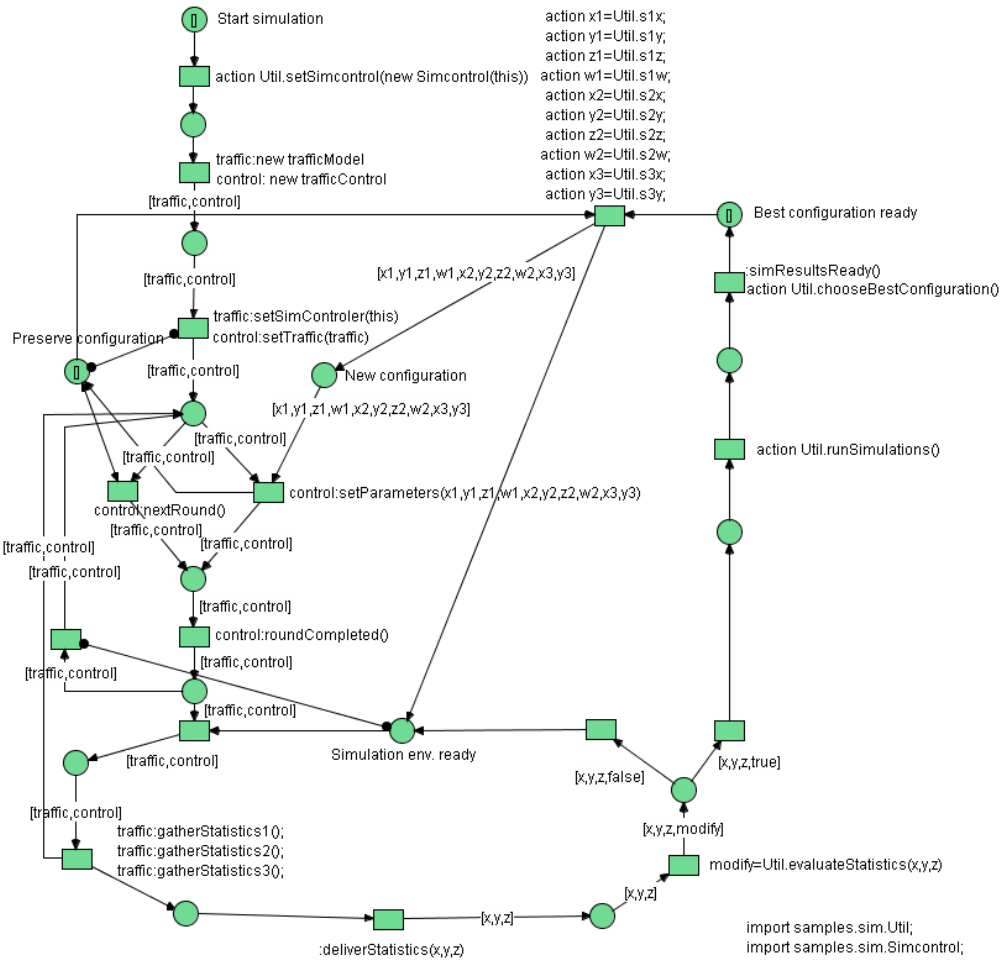
Účelem této případové studie je představit funkční implementaci simulační služby a knihovny modelů, jako základních komponent navrhované simulační architekury. Na simulaci netriviálního modelu s použitím reflektivní simulace se budeme snažit ukázat možnosti těchto služeb a ověřit je v praxi.

4.1 Detaily modelu

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

Model dopravy Jak je možné vidět na obrázku 8, doprava je rozdělena na tři fáze. V první a třetí fázi jsou zahrnuty jen neblokující směry - VA, VD, VB, VE, kdy vozidla 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.

V Petriho síti modelu dopravy reprezentují značky vozidla. Každé vozidlo buď čeká na průjezd křižovatkou, nebo právě projíždí křižovatkou. Vozidla jsou generována podle exponenciálního rozložení se střední hodnotou, určenou konfiguračním parametrem modelu. Každý jízdní pruh a každý směr má přiřazený čas, který značí, dobu průjezdu křižovatkou pro jedno vozidlo. Část



Obrázek 9: Model řízení simulace, včetně vnořených simulací

Petriho sítě slouží ke sběru statistik o provozu. To zahrnuje 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.

Model systému pro řízení dopravy Petriho síť reprezentující model systému pro řízení dopravy se skládá ze dvou oddělených částí - *synchronizační* a *řídicí*. Účelem *synchonizační části* modelu je synchronizace stavu semaforů s modelem dopravy. Jinými slovy, synchronizační síť sděluje aktivní fázi modelu dopravy. *Řídicí část* je hlavní část modelu. Modeluje řízení dopravy pro všechny tři fáze pomocí signálních světel pro chodce a pro vozidla. V každé fázi mohou křižovatkou projíždět pouze vozidla z odpovídající

fáze v modelu dopravy. 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.

Model řízení simulace Petriho síť na obrázku 9 slouží k řízení simulace celého modelu, včetně reflektivních simulací. Dále slouží k nastavení a rekonfiguraci modelu řídicího systému podle statistik z modelu dopravy a výsledků reflektivních simulací. Celý model řízení simulace provádí tyto hlavní činosti:

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 modelu pro řízení dopravy, popřípadě i modelu dopravy
6. Sírá a interpretuje výsledky vnořených simulací
7. Rekonfiguruje modelu systému řídícího dopravu podle výsledku 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 Javě a tato implementace je použita přímo v přechodových akcích. Dvě importované třídy (*Util* a *Simcontrol*) jsou určeny pro analýzu dopravních dat a ke spuštění vzdálených reflektivních simulací.

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. To je nutné např. pro získání dopravních statistik.

Poté, co je model systému pro řízení dopravy inicializován a nakonfigurován, začíná okamžitě řídit dopravu.

Model řídící simulaci periodicky sbírá statistiky, provádí jejich vyhodnocení a rozhoduje o tom, jestli je nutné změnit časové plány semaforů v modelu řídícím dopravu. 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 svých vlastních modelů-kandidátů s různými konfiguracemi a pro každý z nich spustí vnořenou simulaci. Cílem těchto simulací je zjistit, zda-li daná konfigurace

přinese nějaké zlepšení v krátké budoucnosti. Pro tyto simulace je možné použít jednu nebo více služeb běžících lokálně nebo vzdáleně. Ze simulačních výsledků je vybrán nejvhodnější kandidát a jeho konfigurace je použita pro rekonfiguraci celého systému.

Model systému umožňuje nastavit i úroveň vnořených simulací a model ve vnořené simulaci tak může snadno spustit vnořenou simulaci sebe sama, přičemž i tento model může spustit další vnořenou simulaci.

4.2 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 [4], [3], ze kterého jsme vytvořili webovou službu umožňující vzdálenou simulaci. Lokální simulaci chápeme jen jako speciální případ.

Operační logika služby spouští aplikaci Renew jako externí program, kterému dodá 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.

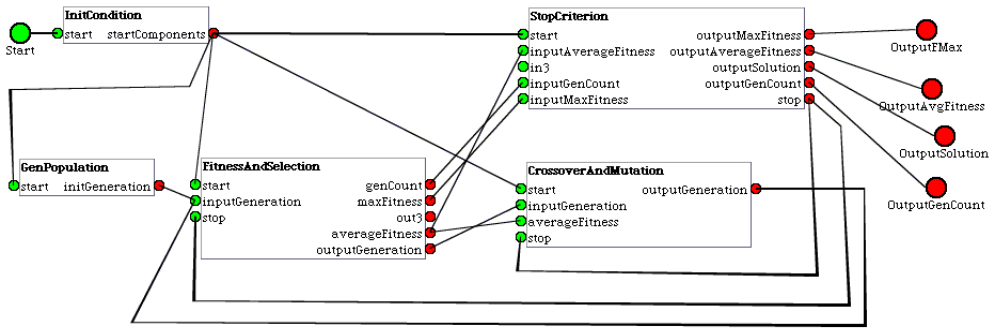
Aby bylo možné volat metody vzdálené webové služby, byly za pomocí prostředku Axis2 (viz kapitola 3.3) vygenerovány základní třídy klientské aplikace, která může být volána přímo z Petriho sítí s referencemi. Tímto způsobem třída Util z modelu řídícího simulaci přistupuje k webové službě, kde spouští novou simulaci.

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.

5 Případová studie - genetické algoritmy

V rámci této případové studie si představíme DEVS reprezentaci genetického algoritmu, navrženého pro řešení potenciálně libovolné úlohy. Tato reprezentace byla převzata z [22], dále upravena a implementována v prostředí SmallDEVS.

Vývoj genetických algoritmů zahrnuje experimentování s napsaným algoritmem a s jeho parametry. V prostředí SmallDEVS je takový vývoj snadnější než v jiném nástroji - 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.



Obrázek 10: GaDEVS - atomické komponenty

Tato případová studie používá interaktivní modelování a ověřuje možnosti simulační služby, prezentační služby a knihovny modelů. Jádro simulační služby zde tvoří nástroj SmallDEVS.

5.1 Detaily modelu

Model genetického algoritmu je implementován jako spojovaná komponenta, která se skládá z několika atomických komponent uvedených na obrázku 10. Celý model je napojen na další komponentu, která je zodpovědná za ukládání, zobrazení a zpracování simulačního výstupu generovaného při exekuci modelu. Tato komponenta používá prezentační službu, která převádí textový simulační výstup na jeho grafickou reprezentaci. Různé varianty modelu genetického algoritmu byly v průběhu jeho vývoje ukládány ve vzdálené knihovně modelů.

Nejdůležitějším objektem, který je předáván mezi komponentami je objekt reprezentující *generaci*, který obsahuje všechny jedince a hlavní parametry genetického algoritmu.

Komponenta *InitCondition* slouží k inicializaci a odstartování celého genetického algoritmu. Spouští ostatní komponenty tím, že pošle hodnotu 1 na svůj výstupní port označený jako *startComponents*. Komponenta *GenPopulation* vytváří počáteční generaci a je použita pouze při startu genetického algoritmu. *FitnessAndSelection* slouží k ohodnocení všech chromozómů v generaci. Některé z nich vybere pro křížení a označí je. Křížení a mutaci pak vykonává *CrossoverAndMutation*. Komponenta *StopCriterion* vyhodnocuje důležité informace o genetickém algoritmu a aktuální generaci a umožňuje jeho zastavení.

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

Závěr

V této práci byly popsány požadavky na návrh a vývoj systémů založený na modelování a simulaci. 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 odpovídá výběr modelovacích formalismů určených pro popis modelů v rámci této práce, kterými jsou 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. Tyto nedostatky vedou k nemožnosti vytvářet znovupoužitelné modely, které by bylo možné použít v rozdílných simulačních prostředích a je také prakticky nemožné použít při vývoji více simulačních nástrojů aniž by bylo třeba věnovat velké množství času zajištění kompatibility mezi používanými simulačními nástroji a modely.

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 nový univerzální jazyk DEVS Meta Language, určený pro popis modelů včetně simulačního prostředí, které jej podporuje. Tento jazyk tvoří hlavní výrazový prostředek pro výměnu modelů mezi simulačními službami a stal se inspirací pro podobný jazyk, nazvaný DEVS Meta Language, který má tendence stát se unznávaným standardem.

Použití celé architektury pro vzdálený návrh a simulaci modelů založených na diskrétních událostech bylo demonstrováno na dvou případových studiích. V jedné studii byla představena implementace simulační služby, kde byl existující simulační nástroj Renew upraven pro použití v rámci simulační architektury, aby mohl 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 ověřeny další důležité komponenty celé architektury, kterými jsou transformační a prezentační služba.

Celý koncept simulační architektury založený na službách by mohl být předmětem dalšího výzkumu a zasluhoval by širší ověření a popřípadě i další rozšíření. Jednoduchá rozšiřitelnost rozhraní jednotlivých služeb v rámci celé architektury je v tomto případě výhodou. Zcela jistě by další pozornost za-

sluhovalo 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 s podporou webových služeb a podpora vývoje distribuovaných modelů, popřípadě modelů založených nejen na diskrétních událostech by mohlo být dalším užitečným rozšířením celé architektury.

Reference

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

[2] Kindler, E. - Reflective Simulation - Simulation of Systems That Simulate. In: *Proc. of ESM - European Simulation and Modeling*, Porto, Portugal (2005)

[3] 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)

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

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

[6] 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

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

[8] 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

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

[10] 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

[11] 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).

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

[13] 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*, 2013, pp. 335–358 (2013)

[14] 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.

[15] 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

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

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

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

[19] 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, 2007, Norfolk, Virginia

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

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

[22] 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

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

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

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

[26] 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.

[27] 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

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

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

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

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

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

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

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

[35] Web Services Description Language `http://www.w3.org/TR/wsdl`

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

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

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

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

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

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

Publikace autora

[42] 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

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

[44] 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

[45] 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

[46] 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