VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií

DISERTAČNÍ PRÁCE

k získání akademického titulu Doktor (Ph.D.)

ve studijním programu
INFORMAČNÍ TECHNOLOGIE

Ing. Pavel Slavíček

Distribuované simulační prostředí

Školitel: **doc. Ing. Zdeňka Rábová, CSc.**
Datum státní doktorské zkoušky: 8.6.2005
Datum odevzdání práce: 15.6.2006

Práce je k dispozici: Vědecké oddělení FIT, Božetěchova 2, 612 66 Brno

Abstrakt

Zájem o oblast modelování a simulace neustále vzrůstá. Modelování a simulace se v dnešní době využívají v mnoha oblastech lidské činnosti. Jsou nezastupitelné při návrhu a analýze komplexních systémů. K provedení simulace komplexního modelu v přijatelném čase je třeba vysoký výpočetní výkon. Jednou z cest k získání potřebného výkonu je využití distribuovaného zpracování, které spočívá v rozložení řešené úlohy na více výpočetních uzlů a její paralelní zpracování. Tato cesta se v dnešní době stává velice populární.

Předložená disertační práce se zabývá návrhem architektury distribuovaného simulačního systému. Navržená architektura je založena na agentním systému a DEVS formalismu. Agentní systém zajišťuje distribuované zpracování, DEVS formalismus je využit jako základní modelovací a simulační nástroj. Agentní systémy jsou moderní technologií a v posledních letech také velmi diskutovaným tématem, zájem o tuto oblast stále narůstá. Systémy založené na této technologii umožňují vytvářet dynamické distribuované systémy, které jsou schopny reagovat na změny prostředí, ve kterém agentní systém pracuje. DEVS formalismus nabízí známý a přehledný nástroj pro modelování a simulace diskrétních systémů. Existuje řada rozšíření původního formalismu, která zjednodušují jeho využití při výstavbě modelů. Na základě navržené architektury byl implementován prototyp simulačního prostředí. Jeho funkčnost byla ověřena na simulaci komplexního modelu, jako ukázkový model byl zvolen model z oblasti dynamiky tekutin.

Vývoj nové architektury pro simulaci DEVS modelů s sebou nese problém portability modelů. Dnešním trendem v simulačních systémech je seskupovat modely a simulační prostředí do těsně provázaných celků. Simulační modely jsou obvykle implementovány prostředky jazyka, ve kterém je implementován simulátor. Tento přístup znemožňuje sdílet tyto modely mezi různými simulačními systémy. V závěru práce je představen metajazyk pro specifikaci simulačních DEVS modelů. Modely specifikované pomocí tohoto jazyka lze poté potenciálně transformovat do kteréhokoliv simulačního prostředí pro DEVS formalismus, což je hlavním přínosem navrženého metajazyka. Vývoj modelu může probíhat v sekvenčním simulačním prostředí, ve kterém se obvykle jednodušeji provádí testování modelu, a výsledný model poté může být simulován ve specializovaném simulačním prostředí, například distribuované či real–time simulační prostředí.

Klíčová slova

Distribuovaná simulace, DEVS formalismus, agentní systém, agent, metamodel

Prohlášení

Prohlašuji, že tuto disertační práci jsem vypracoval samostatně, a že v seznamu literatury jsem uvedl veškeré prameny, ze kterých jsem čerpal.

V Brně 15.6.2006

.....

Ing. Pavel Slavíček

Poděkování

Na tomto místě bych rád poděkoval všem, kteří jakýmkoliv způsobem přispěli k vytvoření této disertační práce.

Moje poděkování patří doc. Ing. Zdeňce Rábové, Csc. za odborné vedení a pomoc při volbě tématu práce, Ing. Vladimíru Janouškovi, Ph.D. za cenné rady a pomoc v průběhu celé práce, celé mé rodině za trpělivost a podporu.

Věda nikdy nevyřeší jeden problém,
aniž by vyprodukovala deset nových.

George Bernard Shaw

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Modelování a simulace systémů dnes tvoří širokou oblast informačních technologií, která se věnuje zkoumání reálného světa prostřednictvím jeho abstrakcí a experimentů nad těmito abstrakcemi. Zájem o tuto oblast neustále vzrůstá. Modelování a simulace se dnes využívají v mnoha oblastech lidské činnosti. Jsou nezastupitelné při návrhu a analýze komplexních systémů. K provedení simulace komplexního modelu v přijatelném čase je třeba vysoký výpočetní výkon. Jednou z cest pro získání potřebného výkonu je využití distribuovaného zpracování, které spočívá v rozložení řešené úlohy na více výpočetních uzlů a její paralelní zpracování. Jednotlivé výpočetní uzly systému jsou propojeny pomocí počítačové sítě a komunikace v tomto systému probíhá pomocí zasílání zpráv. Tato cesta k získání vysokého výpočetního výkonu se v dnešní době stává velice populární, jednou z příčin této popularity je dostupnost rychlého internetu. Tato celosvětová síť prošla během posledních pár let bouřlivým vývojem, expandovala do domácností a stala se z ní běžná součást každodenního života. Z původního systému pro výměnu dokumentů a elektronické pošty se internet transformoval v síť poskytující mnoho služeb. Internet také nabízí možnost jednoduchého propojení mnoha výpočetních uzlů distribuovaného systému. Druhou příčinou této popularity je vysoký výkon běžných pracovních stanic připojených k internetu vzhledem k jejich pořizovací ceně. Díky distribuovanému systému lze získat vysoký výpočetní výkon pro simulace komplexních modelů s minimální finanční investicí.

Jedna z hlavních nevýhod distribuovaného systému je jeho obtížná implementace. Návrh a implementace distribuovaného systému je velmi složitý proces. Předložená disertační práce se zabývá návrhem architektury distribuovaného simulačního prostředí, která zjednoduší jeho vývoj a umožní využití umělé inteligence. Navržená architektura je založena na agentním systému a DEVS formalismu. Agentní systém v této architektuře zajišťuje distribuované zpracování, DEVS formalismus je zde využit jako základní modelovací a simulační nástroj. Využití agentního systému pro realizaci distribuovaného zpracování je hlavním zjednodušením návrhu distribuovaného simulačního systému. Agentní systémy jsou v posledních letech velmi diskutovaným tématem a zájem o ně stále a poměrně výrazně narůstá. O tomto faktu svědčí počty publikovaných prací a množství odborných konferencí zaměřených na tuto problematiku. DEVS formalismus nabízí známý a přehledný nástroj pro modelování a simulace diskrétních systémů. Existuje řada rozšíření původního formalismu, která zjednodušují jeho využití při výstavbě netriviálních modelů.

Vývoj nové architektury pro simulaci DEVS modelů s sebou nese problém portability modelů. Dnešním trendem v simulačních systémech je seskupovat modely a simulační prostředí do těsně provázaných celků. Simulační DEVS modely jsou obvykle implementovány prostředky jazyka, ve kterém je implementován simulátor. Simulační modely implementované tímto způsobem jsou úzce spjaty s těmito simulátory a nelze je využít v jiné implementaci DEVS formalismu. V závěru disertační práce je představena koncepce metajazyka pro specifikace simulačních DEVS modelů. Modely specifikované pomocí tohoto metajazyka nejsou úzce spjaty s konkrétním simulačním prostředím a lze je

potenciálně transformovat do kteréhokoliv simulačního systému pro DEVS formalismus.

Přínos disertační práce lze rozdělit na dvě části. První částí přínosu je návrh architekturty, která spojuje technologii mobilních agentů a DEVS formalismu. Druhá část je návrh metajazyka pro specifikace simulačních DEVS modelů, které nebudou závislé na konkrétním simulačním prostředí. Motivací pro návrh architektury a metajazyka detailněji diskutuje následující sekce kapitoly.

1.1 Motivace

Na Fakultě informačních technologií Vysokého učení technického v Brně (a před jejím vznikem na Ústavu informatiky a výpočetní techniky na Fakultě elektrotechniky a informatiky) je výzkumu metod pro vývoj počítačových systémů věnována dlouhodobá pozornost. Na této fakultě jsou rozšiřovány základní metody modelování a simulace diskrétních i spojitých systémů za účelem modelování a simulace komplexních, heterogenních, paralelních i objektově orientovaných systémů. Z pohledu mé disertační práce jsou zajímavé například práce [24, 14, 46]. Modelováním a simulacemi komplexních systémů se na Fakultě informačních technologií věnuje výzkumná skupina Modelování a simulace systémů, v jejímž rámci probíhá výzkum simulačních metod a prostředků. Do této skupiny obsahově spadá tato práce.

Distribuovaná simulace je dnes hojně využívaný nástroj při návrhu a analýze komplexních systémů pomocí modelování a simulace. Umožňuje získat velký výpočetní výkon s minimálními finančními náklady. Hlavní motivací návrhu nové architektury distribuovaného simulačního prostředí bylo zjednodušení jeho návrhu a implementace. Architektura, která by zjednodušila vývoj takového systému, by urychlila rozvoj v této oblasti a zpřístupnila ji pro širší využití.

Agentní systémy jsou moderní technologií, která v současné době zaznamenává bouřlivý vývoj. Systémy založené na této technologii umožňují vytvářet dynamické distribuované systémy, které jsou schopny reagovat na změny prostředí. Například jsou schopny reagovat na změnu uzlů v distribuovaném systému. Tato vlastnost bude například neocenitelná v případě, kdy bude internet využit jako propojovací síť distribuovaného simulačního systému a prohlížeč www stránek jako platforma pro jeho uzel. V případě takového konceptu odpadá složitá instalace distribuvaného systému. Systém využívá pouze programové vybavení, které je v dnešní době běžnou součástí pracovních stanic připojených k internetu. Díky tomuto přístupu lze získat velmi jednoduše rozšiřitelný systém.

Nejčastěji uváděnou společnou vlastností agentů je jejich autonomie. Agent je tedy aktivní prvek systému vytvořený k předem zamýšlenému účelu. Zde se otevírají možnosti pro využití umělé inteligence. Agent vybavený umělou inteligencí by byl schopen optimalizovat průběh distribuované simulace, optimalizovat dekompozici modelu apod. Zde vidím velké možnosti dalšího výzkumu.

Jedním z hlavních požadavků distribuovaného zpracování pro model je jeho, již zmiňovaná, snadná dekompozice. Model je dekomponován na části, které jsou poté simulovány na jednotlivých uzlech distribuovaného systému. Velkou výhodou pro využití v distribuovaném prostředí je komunikace jednotlivých částí modelu v průběhu simulace pomocí zasílání zpráv. Obě tyto vlastnosti má DEVS formalismus. Z tohoto důvodu byl tento formalismus zvolen jako základní modelovací a simulační nástroj v navržené architektuře. Jedná se o obecně známý prostředek pro modelování a simulace diskrétních systémů. Existuje řada rozšíření původního formalismu, která zjednodušují jeho využití při výstavbě modelů.

1.2 Cíle práce

V úvodu byl rámcově popsán obsah disertační práce, zde budou konkretizovány její cíle. Ve své diplomové práci [58] jsem se zabýval návrhem a implementací simulačního prostředí pro DEVS formalismus. Jednalo se o simulační prostředí založené na klasickém DEVS formalismu. Již během psaní diplomové práce jsem se zabýval myšlenkou využití distribuovaného zpracování při simulaci. Od této myšlenky vedla cesta ke studiu architektur existujících distribuovaných simulačních systémů a k návrhu vlastní architektury distribuovaného simulačního prostředí. Výsledkem byly publikace, ve kterých byla představena navržená architektura distribuovaného simulačního systému [57, 59].

Dnešním trendem je v simulačních systémech seskupovat modely a simulační prostředí do těsně provázaných celků. Simulační modely jsou obvykle implementovány prostředky jazyka, ve kterém je implementován simulátor. Tento přístup znemožňuje sdílet tyto modely mezi různými simulačními systémy. Všeobecně platí, že při vytváření nového simulačního prostředí jsou všechny základní modely opakovaně konstruovány od základů.

Rozhodl jsem se zaměřit disertační práci na oblast distribuované simulace a přenositelnost simulačních modelů mezi různými simulačními prostředími. Stanovil jsem tři základní cíle disertační práce:

1. Navrhnout architekturu distribuovaného simulačního systému. Systém založený na této architektuře bude schopen reagovat na dynamické změny prostředí, jako je například změna počtu výpočetních uzlů v průběhu simulace.

2. Navrhnout metajazyk pro specifikaci simulačních modelů založených na DEVS formalismu. Implementované modely nebudou závislé na konkrétním simulačním prostředí pro DEVS. Tyto modely bude potenciálně možné transformovat pro jakékoliv simulační prostředí určené pro DEVS formalismus.

3. Implementovat prototyp simulačního prostředí založený na navržené architektuře a provést simulaci netriviálního modelu.

1.3 Struktura práce

Po úvodní kapitole následuje kapitola druhá, která se zabývá současným stavem v oblasti distribuované simulace a implementací DEVS formalismu. Diskutuje některé známé a zajímavé systémy a architektury, jako je například *High Level Architecture* [6]. Při výběru simulačních systémů k popisu v této kapitole jsem se soustředil především na prostředí založená na DEVS formalismu. Z pohledu mé práce byl nejzajímavější systém *James* [7].

Třetí kapitola má za úkol čtenáře seznámit se dvěma stavebními kameny navržené architektury. První sekce formálně zavádí DEVS formalismus. Popisuje simulaci modelů specifikovaných pomocí tohoto formalismu. Druhá sekce má za úkol uvést čtenáře do problematiky agentních systémů. Definice základních pojmů v oblasti agentních systémů není jednoduchá. V současné době neexistuje jednotné vnímání v této problematice. Kapitola uvádí obecné rozdělení agentů a popisuje normu FIPA [33] pro agentní systémy.

Čtvrtá kapitola reprezentuje první část jádra disertační práce. V této kapitole je popsána navržená architektura distribuovaného simulačního prostředí. Architektura je založena na dvou stavebních kamenech, které byly popsány v předešlé kapitole. Kapitola detailněji popisuje propojení technologie agentního systému a DEVS formalismu. V úvodu se zabývá obecnou architekturou distribuovaného systému a diskutuje přednosti a nevýhody tohoto typu simulace. Popisuje některé důležité problémy distribuovaného zpracování a prezentuje očekávání od navržené architektury. Po obecném úvodu

představuje navrženou architekturu společně s popisem jednotlivých částí. Následující sekce se detailněji zabývají částmi navržené architektury a službami. Další sekce se zabývá bezpečností distribuovaného systému a diskutuje možné útoky na systém, včetně možné ochrany proti těmto útokům. Poslední sekce porovnává návrženou architekturou se simulačními nástroji z kapitoly 2.2.

Pátá kapitola velmi krátce seznamuje s prototypem simulačního prostředí, který je založen na navržené architektuře. V úvodu jsou vysvětleny důvody volby implementačního jazyka Java. Celá kapitola je rozdělena na tři sekce, v první sekci je vysvětlen princip implementace modelů a na jednoduchých příkladech je ukázána implementace atomické a spojované komponenty. Druhá sekce se zabývá implementací simulace. Simulace v navržené architektuře je založena na agentním systému, při implementaci prototypu byla využita kostra agentního systému z projektu JADE [36]. Jsou zde vysvětleny základní principy funkce tohoto agentního systému a je zde uveden způsob implementace simulace jako chování agentů. Třetí sekce popisuje použití prototypu a poslední sekce diskutuje stav implementace.

Šestá kapitola popisuje simulaci netriviálního modelu pomocí prototypu simulačního prostředí. Jako netriviální model byl zvolen model z oblasti dynamiky tekutin. Na začátku kapitoly je krátký úvod do této problematiky a metodiky výpočtu tohoto složitého problému. Poté je představen simulovaný model a vysvětlen způsob implementace pomocí DEVS formalismu. V závěru kapitoly jsou prezentovány a diskutovány naměřené hodnoty při simulacích.

Sedmá kapitola reprezentuje druhou část jádra disertační práce. Cílem osmé kapitoly je prezentovat navržený jazyk DEVSML (*DEVS formalism MetaLanguage*) pro specifikace simulačních modelů založených na DEVS formalismu. V první části je vysvětlena motivace vzniku jazyka, ve druhé části popisuje jazyk DEVSML včetně transformace modelů pro různá simulační prostředí. V závěru kapitoly jsou diskutovány výhody využití jazyka DEVSML.

Kapitola 2

Současný stav v dané oblasti

Tato kapitola se zabývá současným stavem v oblasti distribuované simulace a implementací DEVS formalismu. Diskutuje některé známé a zajímavé systémy a architektury. Při výběru simulačních systému k popisu v této kapitole jsem se soustředil především na prostředí založené na DEVS formalismu. V průběhu studia a v průběhu psaní disertační práce se mi nepodařilo nalézt simulační systém, který by měl obdobnou architekturou, jako je navržená architektura.

2.1 Implementace DEVS formalismu

Na internetu lze nalézt mnoho projektů zabývajících se implementací DEVS formalismu. Zde jsou uvedeny pouze nejznámější projekty, které uvádí i seznam výzkumné skupiny pro standartizaci DEVS formalismu [39].

2.1.1 DEVSJava

Spoluautorem této simulační knihovny je i autor DEVS formalismu Bernard Zeigler (*University of Arizona*, U.S.A.)[1] [47]. Knihovna nenabízí pouze implementaci klasického formalismu DEVS, ale také jeho různé obdoby jako je například *RealTime–DEVS* apod. Jde o propracovaný a rozsáhlý projekt. K projektu je přístupná podrobná dokumentace včetně zdrojových kódů. Dokumentace i zdrojové kódy lze získat z domovské stránky projektu [11], je třeba bezplatná registrace. Tato knihovna je v dnešní době považována za referenční implementaci DEVS formalismu.

2.1.2 JDEVS

Autorem knihovny je Jean–Baptiste Filippi (*University of Corsica*, France)[2] [9], hlavním záměrem bylo modelování a simulace ekosystémů a tomuto záměru je uzpůsobena i knihovna. Jde o rozsáhlý projekt podporující i 3D vizualizaci výsledků. Velkým přínosem tohoto projektu je návrh a implementace rozšíření původního DEVS formalismu nazvaného *Neural–DEVS*. Rozšíření je implementace neuronové sítě do atomické komponenty. Takto upravený formalismus lze využít pro modelování systémů, u kterých nejsme schopni přesně popsat jejich chování (například právě zmiňované ekosystémy). Díky implementované neuronové síti můžeme chování atomické komponenty ,,naučit" pouze z empiricky získaných dat. Odpadná nutnost přesně specifikovat její přechodové a transformační

[1] University of Arizona, `http://www.arizona.edu`
[2] University of Corsica, `http://www.univ-corse.fr`

funkce na základě přesných znalostí modelovaného systému. Navržené rozšíření je velmi zajímavé a pravděpodobně i velmi užitečné při modelování i jiných komplexních systémů. Dokumentaci a zdrojové kódy lze získat z domácích stránek autora [8].

2.1.3 DEVS/C++

Je implementací DEVS formalismu pro programovací jazyk C++, autory jsou Hyup J. Cho a Young K. Cho. Knihovna DEVS/C++ je vyvíjena na stejné univerzitě jako *DEVSJava*, implementace je postavena na paralelní verzi DEVS formalismu [48]. Dokumentaci a zdrojové kódy lze získat na stránkách projektu [11].

2.1.4 PythonDEVS

PythonDEVS je implementací DEVS formalismu v jazyce Python[3] především pro potřeby výuky na *McGill University*[4]. Později se tento nástroj začal využívat i pro výzkum, především v oblasti sémantiky DEVS a také jako odrazový můstek pro implementace dalších rozšíření DEVS formalismu. PythonDEVS lze získat včetně zdrojových kódů z domovské stránky projektu[5]. Zde lze také najít vizuální modelovací nástroje a implementaci *Real–Time DEVS* formalismu založené právě na PythonDEVS.

2.2 Simulační nástroje, simulační systémy

Tato sekce uvádí přehled několika zajímavých simulačních systémů a architektur distribuovaných simulačních systémů společně s jejich krátkým popisem. Při výběru simulačních systémů k popisu v této kapitole jsem se soustředil především na prostředí založené na DEVS formalismu.

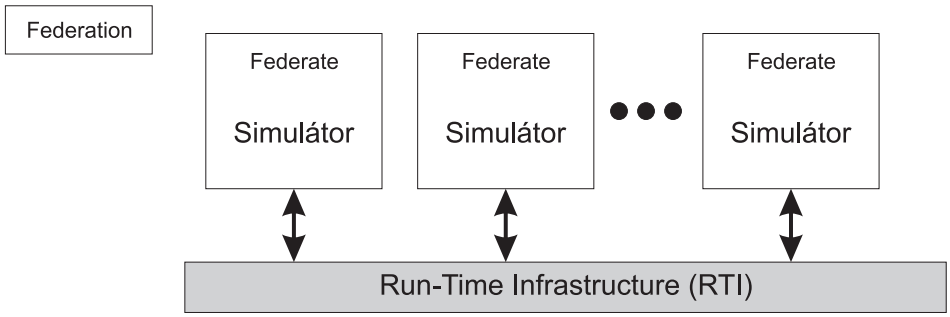
2.2.1 High–Level Architecture

HLA (*High–Level Architecture*) [25, 6] je norma pro propojování simulačních systémů do jednoho funkčního celku, která umožňuje jednotlivým částem systému sdílení informací. Jedná se o velice známou a rozšířenou architekturu, která se využívá v praxi. HLA kombinuje existující simulační systémy místo vytváření obrovských složitých monolitických systémů. Umožňuje využívat existující systémy pro nové aplikace, využívat různé programovací jazyky, různé operační systémy či různé modelovací techniky apod. HLA bylo navrženo na *US Department of Defense*[6]. Je předepsaným standardem pro interoperabilitu vojenských simulačních systémů USA a později se stal standardem i pro NATO. HLA může být využito pro jakýkoliv typ simulačního systému. Zájem o tento standard vzrůstá i v nevojenském prostředí, například v oblasti průmyslu a přepravy. HLA se stalo standardem díky IEEE[7].

HLA kombinuje několik simulačních systémů nazývané *federates* v jeden simulační systém nazvaný *federation* (obrázek 2.1). Abychom toto byli schopni provést, je třeba:

- zdokumentovat sdílené informace v rámci *federation* pomocí dokumentu *Federation Object Model (FOM)*. Dokument popisuje společný jazyk federace. Poskytuje informace o třídách objektů, jejich atributech a interakcích, které budou sdíleny uvnitř federace. Jeden z důvodů flexibility

[3] The Python Programming Language, `http://www.python.org`
[4] McGill University, `http://www.mcgill.ca`
[5] MSDL, `http://moncs.cs.mcgill.ca/MSDL/research/projects/DEVS`
[6] US Department of Defense, `http://www.defenselink.mil`
[7] IEEE, `http://www.ieee.org`



Obrázek 2.1: *High–Level Architecture, federace v HLA*

HLA je možnost definice vlastního FOM pro jednotlivé federace. Pokud je federace založena na obecně známé a využívané oblasti, jsou již FOM modely připraveny a není třeba je vytvářet

- výměna informací mezi jednotlivými simulačními systémy (*federates*) probíhá pomocí *Run–Time Infrastructure (RTI)*, jedná se o standardizovanou specifikaci.

Run–Time Infrastructure (RTI)

Run–Time Infrastructure (RTI) propojuje jednotlivé simulační systémy a umožňuje jednotlivým federacím (*federates*) sdílení informací. Mohou sdílet objekty, jejich atributy a dokonce i interakce mezi objekty. RTI umožňuje synchronizovat čas mezi jednotlivými federacemi. HLA podporuje několik metod řízení času (například reálný čas, čas řízení diskrétními událostmi apod.). RTI může obsahovat funkce pro transformaci sdílených dat. Celý systém RTI je definován jako množina služeb, které může simulační systém využívat, definice celého rozhraní je součástí standardu HLA.

Specifikace HLA

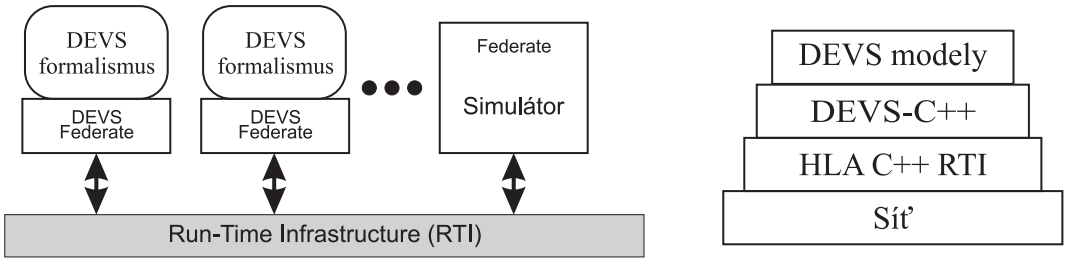
Specifikace HLA se skládá ze tří základních částí. První jsou pravidla pro HLA. Pravidla zaručují správné interakce mezi jednotlivými simulačními systémy v rámci federace. Dále definují jednotlivé odpovědnosti těchto dvou částí. Druhou částí je definice rozhranní. To obsahuje definici rozhranní pro RTI. Dokument také definuje metody, které musí federace a její části implementovat. Dále obsahuje definice pro správu času, správu federace apod. Třetí částí je tzv. *Object Model Template (OMT)*. Předepisuje formáty pro sdílené informace. Definuje tři základní modely. Prvním modelem je *Federation Object Model (FOM)*. Druhým modelem *Simulation Object Model (SOM)*, který je soustředěný na vnitřní specifikaci federace. Třetím modelem je *Management Object Model (MOM)*, který identifikuje objekty a interakce, které jsou využity pro řízení federace. Podrobné informace lze získat v normě IEEE 1516.

2.2.2 DEVS/HLA

DEVS/HLA je distribuované simulační prostředí, které vyhovuje normě HLA a je založeno na DEVS formalismu. Spoluautorem tohoto simulačního nástroje je autor DEVS formalismu Bernard Zeigler a bylo vyvinuto ve stejné laboratoři jako DEVSJava či DEVS/C++. Projekt vznikl jako součást projektu instituce DARPA (*The Defense Advanced Research Projects Agency*[8]). K tomuto projektu nelze

[8]The Defense Advanced Research Projects Agency, `http://www.darpa.mil`

získat zdrojové kódy. Z domovské stránky projektu[9] lze získat dokument s obecným detailním popisem systému. DEVS/HLA lze využít v rozsáhlém simulačním systému založeném na HLA. Prvky implementující architekturu HLA poskytují podporu distribuovaného zpracování, vrstva DEVS je implementací DEVS formalismu. Implementace vychází projektu DEVS/C++. Architekturu systému ukazuje obrázek 2.2.



Obrázek 2.2: *Architektura systému DEVS/HLA*

2.2.3 Swarm

Swarm [35] je balíček programů pro multi–agentní simulaci komplexních systémů. Jeho vývoj začal na *Santa Fe Institute*[10] v roce 1994. Cílem tohoto projektu je vytvořit užitečný nástroj pro výzkum modelování a simulace komplexních biologických systémů. Simulace v tomto simulačním prostředí je založena na vzájemném ovlivňování entit uvnitř systému, entity jsou reprezentovány agenty. Agentní systém je tedy v tomto projektu využit jako modelovací prostředek pro vytváření komplexních modelů nikoliv jako implementace distribuovanosti. Jedná se o velmi zajímavý systém. Inspirativní v tomto projektu je jeho celková koncepce, architektura simulačního jádra a způsob implementace simulačních modelů.

2.2.4 Mason (Multi–Agent Simulator Of Neighborhoods)

Projekt vyvíjený na univerzitě *G. Masona*[11], který hledá další možnosti rozšíření a inovací zahájené projektem Swarm, kterým se také nechal inspirovat. Mason je koncipován jako simulační knihovna pro simulace založené na agentech. Použití této simulační knihovny je velice široké, lze jí využít pro modelování robotických systémů, agentů v počítačových hrách a její použitelnost sahá až po složité reálné sociální a fyzikální modely. Jedná se o velice zajímavý a propracovaný projekt ke kterému je dostatek informací včetně zdrojových kódů [19].

2.2.5 James

James (*Java–Based Agent modeling Environment for Simulation*) je prototyp simulačního nástroje, který je vyvíjen na univerzitě v *Ulmu*[12]. Jedná se o nástroj pro simulace diskrétních modelů založených na agentech. Nejzajímavější na tomto projektu je způsob implementace agentů. Celý simulační systém je založen na paralelní verzi DEVS formalismu [48]. Agenti jsou v systému reprezentováni pomocí atomické komponenty. Aktivity agenta jsou modelovány pomocí vkládání zpráv na výstupní port. Vnímání agenta i jeho reakce na okolní prostředí jsou řízeny pomocí stavu komponenty a její časové

[9] ACIMS, DEVS/HLA, `http://acims.arizona.edu/SOFTWARE/software.shtml`
[10] The Santa Fe Institute, `http://www.santafe.edu`
[11] The George Mason University, `http://www.gmu.edu`
[12] University of Ulm, `http://www.uni-ulm.de`

funkce. Pro tento projekt nelze získat detailnější dokumentaci ani zdrojové kódy projektu. Dle informací od autorů nebude projekt nikdy distribuován se zdrojovými kódy a podrobnou dokumentací. Důvodem je spolupráce společnosti NASA[13] na tomto projektu. Dílčí informace lze získat na domovských stránkách projektu [7] a z publikovaných článků na vědeckých konferencích, například [28].

2.2.6 Farm

Farm [4] je distribuované simulační prostředí pro simulace rozsáhlých agentních systémů implementované v programovacím jazyce Java a vyvíjený na univerzitě v *Massachusetts*[14]. Využívá komponentní architekturu, která jednoduše umožňuje toto prostředí dále rozšiřovat a modifikovat jeho jednotlivé části. Agenti operující v tomto systému jsou skutečné objekty simulace. Typicky se jedná o jednoduché (což ale nemusí být pravidlem) a částečně autonomní entity, které mají v tomto systému definované cíle a své role. Agenti pracují v pseudo reálném čase, kde má každý agent přidělen přesně stanovený čas CPU. V podstatě se jedná o implementaci agentního systému s nástroji pro administraci systému, jeho analýzu a vizualizaci stavu systému. Informace lze získat na domovských stránkách projektu [37] včetně zdrojových kódů.

[13] National Aeronautics and Space Administration, `http://www.nasa.gov`
[14] University of Massachusetts, `http://www.umass.edu`

Kapitola 3

Základní stavební prvky navržené architektury

Úkolem této kapitoly je seznámit čtenáře se základními stavebními prvky navržené architektury. V úvodu kaptioly jsou definovány základní pojmy z oblasti modelování a simulací, tyto pojmy jsou využívány v dalším textu. Dále jsou v této kapitole uvedeny základní vlastnosti a pojmy technologii, které jsou nutné k popisu navržené architektury. Architektura se opírá o dva základní stavební kameny, prvním je DEVS formalismus a druhým je agentní systém. Druhá sekce kapitoly (3.2) formálně zavádí DEVS formalismus. V této sekci je popsán algoritmus simulace a představena paralelní revize původního formalismu využívanou v distribuované simulaci. Třetí sekce kapitoly (3.3) má za úkol uvést čtenáře do problematiky agentních systémů. Definice základních pojmů obvykle bývá jednoduchou záležitostí, ovšem v případě pojmů používaných v oblasti agentních systémů není situace tak jednoduchá. V současné době neexistuje jednotné vnímání v této problematice. V první části tato sekce definuje základní pojmy, další část je věnována základnímu dělení agentů a poslední část popisuje normu pro agentní systémy FIPA. Obsah této části kapitoly byl převzat ze zdrojů [46, 23, 33, 30], kde je možné tuto problematiku dále studovat.

3.1 Základní pojmy

V dalším textu práce budou využívány některé základní pojmy z oblasti teorie modelování a simulací, z tohoto důvodu zde budou ty nejzákladnější zavedeny. Další definice a pojmy z této oblasti lze studovat například z [26, 50], obsah této sekce byl čerpán z [26].

3.1.1 Systém

K definici pojmu systém je nutné nejprve definovat dva pomocné pojmy. Prvním pojmem je *univerzum*, druhým pojmem je *charakteristika univerza*.

Definice 1 *Univerzum je konečná množina prvků* $\{u_1, u_2, \ldots, u_N\}$ *mezi nimiž existují vazby* $R_{i,j}$ *charakterizované nějakými výstupními a vstupními veličinami. Předpokládá se, že každý prvek univerza má alespoň jeden vstup a jeden výstup.*

Definice 2 *Charakteristika univerza je definována jako* $R = \bigcup_{i,j=1}^{n} R_{i,j}$*, množinu všech vazeb mezi prvky univerza.*

Nyní již máme definované vše potřebné k definici pojmu *systém*.

Definice 3 *Systém je dvojice $S = (U, R)$, kde U je univerzum a R je charakteristika univerza.*

Charakteristika univerza reprezentuje relaci mezi prvky univerza. Relací mezi prvky se chápe propojení vstupních a výstupních bran prvků v systému. Pokud v systému existuje prvek $u \in U$, pak prvek má:

- množinu vstupních bran $X_u = \{x_{u1}, x_{u2}, \ldots, x_{um}\}$
- množinu výstupních bran $Y_u = \{y_{u1}, y_{u2}, \ldots, y_{un}\}$

Během simulace, která probíhá ve stanoveném časovém rozmezí $T = (t_{start}, t_{end})$, se stav systému mění. Změna systému je dána změnou charakteristiky systému, nebo změnou informace obsažené v systému (hodnotami na vstupech/výstupech jednotlivých prvků univerza nebo jejich vnitřními stavy). Každý prvek univerza má své chování, které určuje reakci prvku na vstupní podněty.

Chování systému

Na základě chování prvků rozlišujeme prvky: *diskrétní, spojité, deterministické, nedeterministické*. V případě diskrétních prvků je časová množina konečná, nebo také spočetná. U spojitých prvků se jejich stav mění spojitě na stanoveném časovém intervalu. Rozdíl mezi deterministickými a nedeterministickými prvky systému spočívá ve stanovení odezvy. U deterministických prvků je jejich odezva jednoznačně dána vstupními a stavovými vektory. Stochastické prvky mají nedeterministické chování, odezva na vstup je dána pravdivostním rozložením.

Okolí systému

Z hlediska otevřenosti systému vůči okolí, které lze považovat za zdroj ovlivňování, se systémy dělí na systémy: *otevřené, relativně uzavřené, uzavřené*. Pokud dochází ke komunikaci s okolím, komunikace s prvky vně uvažovaného systému, jedná se o systém otevřený. Relativně otevřený systém má vymezené okolí, které je pak možné zahrnout do systému jako další prvek, čím se poté systém stává vůči okolí uzavřený. U uzavřeného systému nedochází ke komunikaci s prvky vně systému, ke komunikaci dochází pouze mezi prvky uvnitř systému.

3.1.2 Model, modelování, simulace

Modelováním se rozumí získávání informací o jednom systému prostřednictvím jiného systému–modelu. Simulace reprezentuje proces experimentování s modelem, analýzou výsledků simulačních experimentů získáváme informace o chování modelu, které můžeme při dostatečné adekvátnosti modelu interpretovat jako chování modelovaného systému. Simulační model je speciální typ modelu, model je proveditelný, tj. lze s ním na počítači provádět simulační experimenty. Výsledky získáváme v ryze numerické podobě. Pro nepatrně pozměněný model musíme celý postup řešení opakovat. Proto tyto modely nabyly na významu až teprve s rozvojem výpočetní techniky.

3.2 DEVS Formalismus

Stav diskrétního modelu je reprezentován stavovými veličinami a během simulace se v konečném časovém intervalu mění jen v konečně mnoha okamžicích. To znamená, že na spojité časové ose existuje pouze konečně mnoho bodů, ve kterých může nastat změna stavu modelu. Tato změna je reprezentována změnou hodnot stavových veličin. V těchto časových okamžicích tedy nastávají tzv. *události* (*events*). Tyto změny jsou elementární a okamžité, tzn. mají nulovou dobou trvání. Stav modelu systému je tedy řízen *diskrétními událostmi*. Mezi jednotlivými událostmi se stav systému nemění. Tyto „skokové" změny stavových veličin představují hlavní odlišnost od systémů spojitých, kde se hodnoty stavových veličin mohou měnit spojitě v průběhu celého časového intervalu.

Formalismus DEVS (*Discrete EVent System Specification*) navrhl Zeigler [50] jako prostředek pro obecný a formální popis pro modelování a simulace diskrétních systémů. Modely jsou popsány v abstraktní rovině, ve které je časová osa spojitá a v průběhu ohraničeného časového intervalu se může vyskytnout pouze konečně mnoho událostí, které mohou způsobit změnu stavu modelu. DEVS formalismus umožňuje popis systému ve dvou úrovních. Na nižší úrovni, nazývané jako *atomická*, popisuje chování systému jako sekvenci deterministrických přechodů mezi sekvenčními stavy modelu, reakce modelu na externí vstupy a jakým způsobem jsou generovány výstupy z modelu. Na vyšší úrovni popisu, nazývaného jako *spojovanou*, popisuje systém jako síť propojených komponent. Komponentou je atomický DEVS model. Propojení mezi komponentami definuje vzájemné ovlivňování komponent, výstupní událost jedné komponenty se může pomocí propojení stát vstupní událostí jiné komponenty. V [50] je dokázáno, že pro každý spojovaný model lze zkonstruovat ekvivalentní atomický model. A tak lze jakýkoliv DEVS model nahradit atomickým modelem. Díky této vlastnosti může spojovaný model obsahovat nejen atomické modely, ale také jiné spojované modely. Tak lze vytvářet hirearchické modely.

3.2.1 Atomická komponenta

Atomický DEVS formalismus je struktura popisující rozdílné aspekty chování systému. Pomocí atomického formalismu modelujeme elementární části systému. Chování systému modeluje jako sekvenci deterministických přechodů mezi sekvenčními stavy, dále definuje reakce na externí podněty a způsob generování výstupů z modelu. Formálně je atomický DEVS definován jako n–tice (3.1).

$$atomicDEVS = \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle \tag{3.1}$$

Časová osa $\mathbf{T}$ je spojitá a není uváděna explicitně:

$$T = \Re$$

Množina vnitřních stavů $\mathbf{S}$ je množina všech povolených stavů modelu. Dynamika atomického modelu je složena z uspořádáné sekvence stavů z množiny S. Typicky bude množina S strukturovaná.

$$S = \times_{i=1}^{n} S_i$$

Čas, po který systém zůstane ve stavu před přechodem do dalšího stavu, je definován pomocí *funkce času přechodu* $\mathbf{ta}$:

$$ta : S \rightarrow \Re_{0,+\infty}^{+}$$

Stejně tak, jako postupuje čas v reálném světě, musí být hodnota funkce ta nezáporné číslo. Případ $ta(s) = 0$ reprezentuje okamžitý přechod do dalšího stavu. Pokud se systém nachází vždy v konečném

stavu $s \in S$, poté $ta(s) = \infty$.
Interní přechodová funkce $\delta_{\mathbf{int}}$:

$$\delta_{int} : S \to S$$

Tato funkce modeluje přechod z jednoho vnitřního stavu do následujícího vnitřního stavu. Funkce δ_{int} popisuje chování konečného stavového automatu, funkce ta přidává vývoj v čase.
Množina výstupů $\mathbf{Y}$ je množina všech povolených výstupů z modelu. Typicky bude množina Y strukturovaná:

$$Y = \times_{i=1}^{l} Y_i$$

Tím je formalizováno l výstupních portů. Každý výstupní port je identifikován jeho unikatním indexem i. Indexy mohou být odvozeny od unikátních jmen jednotlivých portů pro jednodušší práci s modelem.
Výstupní funkce λ mapuje interní stav $s \in S$ na množinu výstupů Y.

$$\lambda : S \to Y \cup \{\emptyset\}$$

Výstupní události modelu jsou generovány pouze v okamžiku interního přechodu. V tomto okamžiku je aktuální stav (před provedením interní přechodové funkce) použit jako vstup pro výstupní funkci. V žádném jiném okamžiku není výstupní událost generována.

K popsaní úplného stavu systému v jakémkoliv časovém okamžiku není sekvenční stav $s \in S$ dostatečný. Je nutné brát v úvahu i uplynulý čas e od provedení posledního přechodu do aktuálního stavu $s \in S$. Z tohoto důvodu definujeme *totální stav* Q:

$$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$$

Uplynulý čas e nabývá hodnot v intervalu 0 (přechod se právě provedl) až $ta(s)$ (čas pro provedení přechodu do nasledujícího stavu). Můžeme také definovat funkci σ pro výpočet času zbývajícího do provedení přechodu:

$$\sigma = ta(s) - e$$

Až do této chvíle byl definován autonomní systém. Systém který nepřijímá žádné externí podněty. *Množina vstupů* $\mathbf{X}$ je množina všech povolených vstupů do modelu. Typicky bude množina X strukturovaná.

$$X = \times_{i=1}^{m} X_i$$

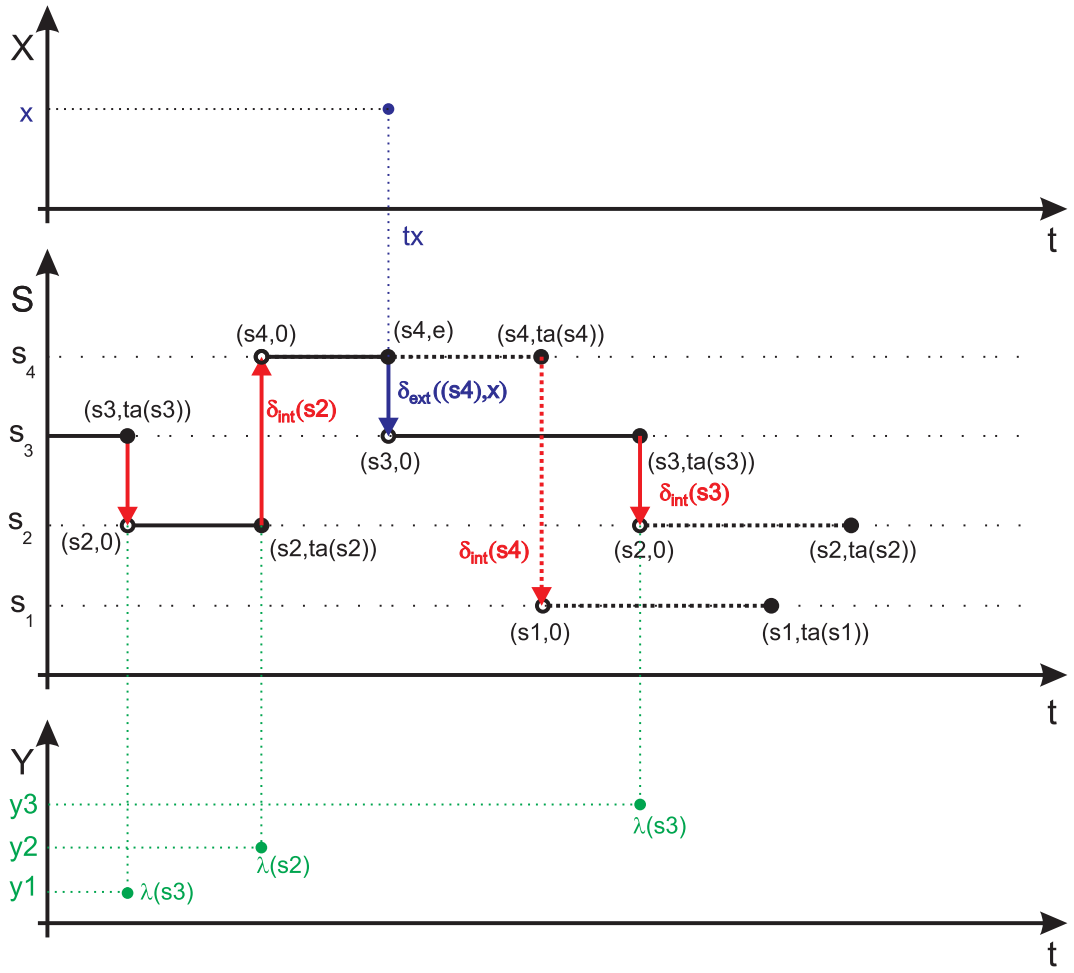
Tím je formalizováno m vstupních portů. Každý vstupní port je identifikován jeho unikátním indexem i. Indexy mohou být odvozeny od unikátních jmen jednotlivých vstupních portů pro jednodušší práci s modelem.
Množina Ω obsahuje všechny přípustné vstupní segmenty ω :

$$\omega : T \to X \cup \{\emptyset\}$$

V uzavřeném časovém intervalu generuje vstupní segment pouze konečně mnoho *externích událostí* různé od události prázdné. Tyto externí události zapříčiní přerušení autonomního chování modelu a reakci na externí událost popsanou pomocí *externí přechodové funkce* $\delta_{\mathbf{ext}}$:

$$\delta_{ext} : Q \times X \to S$$

Reakce systému na externí událost závisí nejen na aktuálním sekvenčním stavu a vstupu, ale také na uplynulém čase. Externí přechodová funkce umožňuje popisovat velkou třídu chování, které jsou typické pro tyto diskrétní modely (např. synchronizace). Pokud není vstupní událost x uvedena ve specifikaci externí přechodové funkce, je tato událost ignorována.



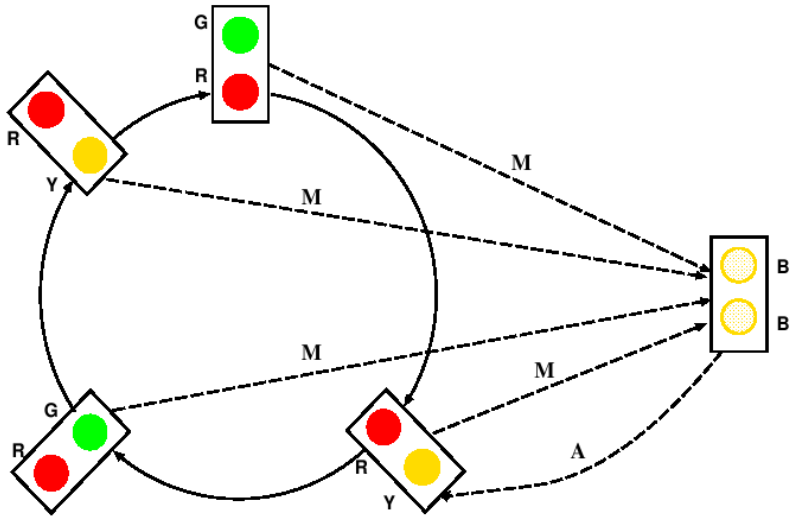
Obrázek 3.1: *Ukázka průběhu stavy atomického modelu*

Na obrázku 3.1 je znázorněn příklad průběhu stavy atomického modelu. Na obrázku provedl model interní přechod do stavu $s2$. Bez žádných vstupních událostí zůstává model ve stavu $s2$ po čas $ta(s2)$. V tomto časovém intervalu nabývá uplynulý čas e, po který se nachází systém ve stavu $s2$, hodnoty od 0 do $ta(s2)$ s totálním stavem $q = (s2, e)$. V okamžiku, kdy uplynulý čas dosáhne hodnoty $ta(s2)$, je generován první výstup $y2 = \lambda(s2)$, poté model skokově přejde do nového stavu $s4 = \delta_{int}(s2)$. V autonomním režimu by model setrval ve stavu $s4$ po čas $ta(s4)$ a poté by přešel do stavu (po vygenerování výstupu) do stavu $s1 = \delta_{int}(s4)$. Tento model ale nepracuje v autonomním režimu a než uplynulý čas e dosáhne hodnoty $ta(s4)$, nastane na vstupu externí událost x. V tomto okamžiku model přejde do nového stavu $s3 = \delta_{ext}((s4, e), x)$, není generován žádný výstup z modelu a model pracuje dále v autonomním režimu tzn. zůstává ve stavu $s3$ maximálně po čas $ta(s3)$.

Příklad atomické DEVS komponenty

Uvažujme model světelné signalizace na křižovatce. Světelná signalizace může pracovat ve dvou režimech. První režim je automatický (na obrázku a v dalším textu označen jako A), přepínání signalizace se provádí dle obecně známých pravidel. Druhý režim je manuální (dále pouze M), na obou stranách blikají žluté barvy. Přepnutí z jednoho do druhého režimu představuje pro model externí signál (událost). Ve formálním zápisu jsou uvedeny tyto značky R (*Red*) červená, G (*Green*) ze-

lená, Y (*Yellow*) žlutá, B (*Blink*) blikání, M (*Manual*) přepnutí do manuálního režimu, A (*Automatic*) přepnutí do automatického režumu. Situaci ukazuje obrázek 3.2. Vztah 3.2 formálně definuje atomický model.



Obrázek 3.2: *Ukázka atomické komponenty*

$$SemaphoreModel = \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle \tag{3.2}$$

Kde

$$T = R$$

$$X = \{M, A\}$$

$$\omega : T \rightarrow X \cup \{\emptyset\}$$

$$S = \{RG, RY, GR, YR, BB\}$$

$$\delta_{int}(RG) = RY;\ \ \delta_{int}(RY) = GR;\ \ \delta_{int}(GR) = YR;\ \ \delta_{int}(YR) = RG$$

$$ta(RG) = 60s;\ \ ta(RY) = 10s;\ \ ta(GR) = 50s;\ \ ta(YR) = 10s\ \ ta(BB) = +\infty$$

$$\delta_{ext}((RG, e), M) = BB;\ \ \delta_{ext}((RY, e), M) = BB;\ \ \delta_{ext}((GR, e), M) = BB$$

$$\delta_{ext}((YR, e), M) = BB;\ \ \delta_{ext}((BB, e), A) = RY$$

$$Y = \{GREEN, YELLOW, BLINK\}$$

$$\lambda_1(RG) = \lambda_1(RY) = RED; \lambda_1(GR) = GREEN;$$

$$\lambda_1(YR) = YELLOW; \lambda_1(BB) = BLINK$$

3.2.2 Spojovaná komponenta

Z pohledu této části formalismu se model skládá z propojené sítě komponent, přičemž komponentou se rozumí model popsaný pomocí atomického formalismu. Definovaná propojení mezi komponentami modelu popisují, jakým způsobem se jednotlivé komponenty vzájemně ovlivňují. Výstup z jedné komponenty může být pomocí propojení přiveden na vstup komponenty jiné. Spojovaná komponenta je formálně definován jako n–tice (3.3).

$$coupledDEVS = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle \tag{3.3}$$

Self označuje samotnou komponentu. $\mathbf{X_{self}}$ je (může být strukturovaná) množina povolených vstupů do komponenty. $\mathbf{Y_{self}}$ je (může být strukturovaná) množina povolených výstupů z komponenty. $\mathbf{D}$ je množina jedinečných referencí (jmen) na komponenty modelu. Reference na vlastní spojovanou komponentu je reprezentována pomocí $self$ a proto se již nenachází v množině D. Množina komponent je

$$\{M_i | i \in D\}$$

Každá komponenta je atomická komponenta.

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i \rangle, \forall i \in D$$

Množina referencí na komponenty ovlivňovaných komponentou $i \in D \cup \{self\}$ je označena jako $\mathbf{I_i}$. Množina všech těchto množin popisuje strukturu propojovací sítě.

$$\{I_i | i \in D \cup \{self\}\}$$

Z důvodů modularity nemůže žádná komponenta ovlivňovat komponentu mimo její rozsah. Přesněji řečeno, komponenta spojovaného modelu může ovlivňovat pouze ostatní komponenty spojované komponenty či spojovanou komponentu samotnou:

$$\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}$$

V případě, kdy komponenta ovlivňuje přímo sama sebe a $ta = 0$, vzniká nežádoucí závislost. Z tohoto důvodu zavádíme omezení, které zabraňuje tomuto stavu a vylučuje, aby komponenta ovlivňovala přímo sama sebe.

$$\forall i \in D \cup \{self\} : i \notin I_i$$

Pro transformaci výstupní události jedné komponenty na odpovídající vstupní událost ovlivňované komponenty je definována *výstupně–vstupní translační funkce* $\mathbf{Z_{i,j}}$:

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\},$$

$$Z_{self,j} : X_{self} \rightarrow X_j, \forall j \in D,$$

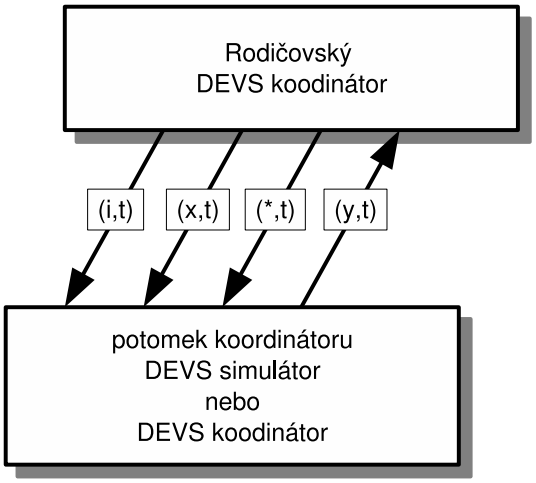
$$Z_{i,self} : Y_i \rightarrow Y_{self}, \forall i \in D,$$

$$Z_{i,j} : Y_i \rightarrow X_j, \forall i, j \in D$$

Společně I_i a $Z_{i,j}$ kompletně specifikují strukturu propojení a chování spojované komponenty.

Jako výsledek propojení několika komponent může nastat stav, ve kterém se objeví přechod do nového stavu u více komponent ve stejný modelový čas. V sekvenčích simulačních systémech jsou tyto kolize řešeny pomocí *výběrové funkce* $select$. Tato funkce určí, která komponenta se bude zpracovávat nejdříve. *Výběrová funkce* **select** je definována:

$$select : 2^D \rightarrow D$$



Obrázek 3.3: *Komunikace DEVS koordinátoru a jeho potomků*

3.2.3 Simulace DEVS modelů

Simulátor diskrétních modelů pracuje na základě seznamu událostí, který je seřazen dle času aktivace událostí. Události jsou prováděny dle seznamu v sekvenčním pořadí. Události provádějí změny stavu modelu a plánují nové události, které vkládáji do seznamu událostí, mohou provádět rušení již naplánovaných událostí a jejich odstranění ze seznamu.

Na základě všeobecných úvah je formulován simulační algoritmus pro DEVS formalismus. Pro atomický DEVS je formulován *simulátor*, pro spojovaný DEVS je formulován *koordinátor*. Hierarchický simulátor pro spojovaný model se skládá z *simulátorů* a *koordinátorů* a používá čtyři typy zpráv (situaci ukazuje obrázek 3.3). Initializační zpráva (i, t) je poslána při inicializaci z rodičovského simulátoru všem jeho potomkům. Plánování události je prováděno pomocí zprávy pro změnu stavu $(*, t)$ a jsou zasílány z koordinátora jeho potomkům, které mají naplánovanou změnu vnitřního stavu na modelový čas t. Zpráva (y, t) je zasílána potomky jejich koordinátorům, informuje je o výstupní události komponenty. Naopak, zprávou (x, t) informuje koordninátor své potomky o vstupní události.

Simulátor atomické komponenty

Simulátor používá dvě proměnné t_l a t_n. Proměnná t_l uchovává hodnotu simulačního času, ve kterém nastala poslední událost, proměnná t_n uchovává hodnotu simulačního času na který je naplánována další událost. Z definice *funkce času přechodu* atomické komponenty vyplývá:

$$t_n = t_l + ta(s)$$

Pokud máme k dispozici aktuální simulační čas, simulátor může vypočítat uplynulý čas od poslední události:

$$e = t - t_l$$

A také čas zbývající do další události:

$$\sigma = t_n - t = ta(s) - e$$

Čas další události t_n je zaslán rodičovskému koordinátorovi pro zajištění správné synchronizace událostí. Jak ukazuje algoritmus 1, je nutné před každým spuštěním simulace provést inicializaci simulátoru. Inicializace se provádí pomocí $message(i, t)$, po přijetí této zprávy inicializuje simulátor

Algoritmus 1: Algoritmus simulátoru pro atomickou komponentu

Data:
$parent$ rodičovský koordinátor, kam jsou zasílány zprávy
t_l čas poslední události
t_n čas další události
$DEVS$ komponenta s totálním stavem (s, e)
y výstupní hodnota

when $i - message(i, t)$ **do**
  $t_l = t - e$;
  $t_n = t_l + ta(s)$;
end

when $* - message(*, t)$ **do**
  if $t \neq t_n$ **then**
    error: bad synchronization;
  else
    $y = \lambda(s)$;
    send $y - message(y, t)$ to $parent$;
    $s = \delta_{int}(s)$;
    $t_l = t$;
    $t_n = t_l + ta(s)$;
  end
end

when $x - message(x, t)$ **do**
  if $not(t_l \leq t \leq t_n)$ **then**
    error: bad synchronization;
  else
    $e = t - t_l$;
    $s = \delta_{ext}((s, e), x)$;
    $t_l = t$;
    $t_n = t_l + ta(s)$;
  end
end

proměnné t_l a t_n. $Message(*, t)$ vyvolá provedení interní události komponenty, komponenta vypočítá výstup y a simulátor pošle $message(y, t)$, která informuje nadřazený simulátor o externí události. $Message(x, t)$ informuje simulátor o přijetí externí události x v simulačním čase t a komponenta provede externí přechodovou funkci.

Koordinátor spojované komponenty

V koordinátoru pro spojovaný model je každá komponenta modelu řízena vlastním simulátorem, který je zodpovědný za korektní simulaci komponenty. Koordinátor přiřazený spojované komponentě je zodpovědný za synchronizaci simulátorů komponent a obsluhu externích událostí.

Korektní synchronizaci komponent zajišťuje koordinátor pomocí seznamu událostí, tento seznam obsahuje dvojice simulátoru a času další události t_n komponenty. Seznam je setříděn dle času t_n. Pokud nastane případ, že má v seznamu více položek stejný čas t_n, je použita funkce $select$ spojované komponenty. Čas t_n první komponenty v seznamu reprezentuje čas další události celé spojované komponenty.

$$t_n = min\{t_{n_d} | d \in D\}$$

Tento čas je poskytnut nadřazenému koordinátorovi jako čas další události celé spojované komponenty. Obdobným způsobem je definován čas poslední události.

$$t_l = max\{t_{l_d} | d \in D\}$$

Jak již bylo uvedeno dříve, koordinátor komunikuje pomocí stejných zpráv jako simulátor (viz. algoritmus 2). Když koordinátor obdrží $message(i, t)$, rozešle ji všem svým podřízeným simulátorům a koordinátorům. Po obdržení odpovědi od všech podřízených prvků nastaví koordinátor proměnnou t_l na maximální hodnotu ze všech zjištěných časů poslední události jeho podřízených komponent a t_n na minimální hodnotu ze všech časů další události jeho podřízených komponent.

Když koordinátor obdrží $message(*, t)$, přesměruje ji první komponentě z $event - list$. Tato komponenta provede svoji přechodovou funkci a může vygenerovat výstupní $message(y, t)$. Po dokončení interní události a všech externích událostí způsobené výstupem komponenty, je vypočítán

Algoritmus 2: Algoritmus koordinátora pro spojovanou komponentu

Data:
$DEVSN = (X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, select)$ přiřazená komponenta
$parent$ rodičovský koordinátor
t_l čas poslední události, t_n čas další události
$event - list$ seznam elementů (d, t_{n_d}) seřazených dle t_{n_d} a $select$

when $i - message(i, t)$ **do**
  begin
    foreach d *in* D **do**
      send $i - message(i, t)$ to d;
    end
    sort $event - list$;
    $t_l = max\{t_{l_d} | d \in D\}$;
    $t_n = min\{t_{n_d} | d \in D\}$;
  end
end

when $* - message(*, t)$ **do**
  if $t \neq t_n$ **then**
    error: bad synchronization;
  else
    $d^* = first(event - list)$;
    send $* - message(*, t)$ to d^*;
    sort $event - list$;
    $t_l = t$;
    $t_n = min\{t_{n_d} | d \in D\}$;
  end
end

when $x - message(x, t)$ **do**
  if $not(t_l \leq t \leq t_n)$ **then**
    error: bad synchronization;
  else
    $receivers = \{r | r \in D, N \in I_r, Z_{N,r}(x) \neq \emptyset\}$;
    foreach r *in* $receivers$ **do**
      send $x - message(x_r, t)$ where $x_r = Z_{N,r}(x)$ to r;
    end
    sort $event - list$;
    $t_l = t$;
    $t_n = min\{t_{n_d} | d \in D\}$;
  end
end

when $y - message(y_{d^*}, t)$ **do**
  if $(d^* \in I_N)$ *and* $(Z_{d^*,N}(y_{d^*}) \neq \emptyset)$ **then**
    send $y - message(y_N, t)$ with $y_N = Z_{d^*,N}(y_{d^*})$ to $parent$;
  else
    remove y_{d^*} from d^*;
    $receivers = \{r | r \in D, N \in I_r, Z_{N,r}(x) \neq \emptyset\}$;
    foreach r *in* $receivers$ **do**
      send $x - message(x_r, t)$ where $x_r = Z_{d^*,r}(y_{d^*})$ to r;
    end
  end
end

čas příští události jako minimální čas příští události všech potomků koordinátora. Když koordinátor obdrží $message(y, t)$, která nese výstupní informaci vybrané komponenty (viz. předchozí odstavec) zjistí, zda má tuto zprávu předat nadřazenému koordinátorovi a vytvoří množinu komponent, kterým má být zpráva doručena.

Kořenový koordinátor

Kořenový koordinátor je odpovědný za start a ukončení simulace. Simulace je implementovaná jako smyčka, ve které tento kořenový simulátor cyklicky zasílá $message(*, t)$ své podřízené komponentě. Podřízenou komponentou je jedna spojovaná komponenta, který zastřešuje simulovaný model. V případě velice jednoduchého modelu může být touto podřízenou komponentou pouze simulátor atomické komponenty[1]. Smyčka se provádí na základě podmínky $konec\,simulace$, tato podmínka nejčastěji obsahuje omezení simulačního času. Před spuštěním vlastní simulace je nutné provést inicializaci komponent modelu, ta je provedena zasláním inicializační zprávy $message(i, t)$ podřízené komponentě. Celý algoritmus (algoritmus 3) je velmi jednoduchý.

3.2.4 P–DEVS paralelní revize formalismu

P–DEVS je revizí původní verze DEVS formalismu, která odstraňuje veškeré sekvenční závislosti a umožňuje tento formalismus využít v paralelním prostředí. Upravuje definici atomické i spojované komponenty. Tato sekce diskutuje rozdíly oproti klasické verzi, navržená architektura je založena na této paralelní verzi formalismu.

[1] V případě, že celý model je reprezentován pouze atomickou komponentou.

Algoritmus 3: Algoritmus kořenového koordinátora

```
Data:
    t aktuální simulační čas;
    child podřízený koordinátor (simulátor);
begin
    t = t_0;
    send message(i, t) to child;
    t = child.t_n;
    while not(konec simulace) do
        send message(*, t) to child;
        t = child.t_n;
    end
end
```

P–DEVS atomická komponenta

Definici atomické komponenty pro paralelní DEVS formalismus uvádí vztah 3.4. Revize atomické komponenty přináší tři hlavní rozdíly. Prvním viditelným rozdílem je nová *funkce souběhu* δ_{con}. Funkce poskytuje kontrolu nad kolizním chováním modelu v případě, že model obdrží externí podnět v čase interního přechodu ($e = 0$ nebo $e = ta(s)$).

$$PAtomic = \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{con}, Y, \lambda \rangle \tag{3.4}$$

Funkce souběhu δ_{con} je definována :

$$\delta_{con} : S \times X^b \rightarrow S$$

X^b je množina multimnožin pro prvky z X. Paralelní zpracování umožňuje generovat několik externích událostí souběžně. A z tohoto důvodu musí být model schopen zpracovat více externích událostí v jeden modelový čas. Proto jsou odlišné i definice *externí přechodové funkce* a *funkce výstupní*:

$$\delta_{ext} : Q \times X^b \rightarrow S, \quad \lambda : S \rightarrow Y^b$$

Význam a definice ostatních částí je stejná jako u atomické komponenty klasického DEVS formalismu (vztah 3.1).

P–DEVS spojovaná komponenta

Z definice spojované komponenty byla vypuštěna výběrová funkce $select$. V paralelním prostředí tato funkce ztratila smysl, protože komponenty mohou být zpracovány paralelně. Formální definici spojovaného modelu uvádí n–tice (3.5).

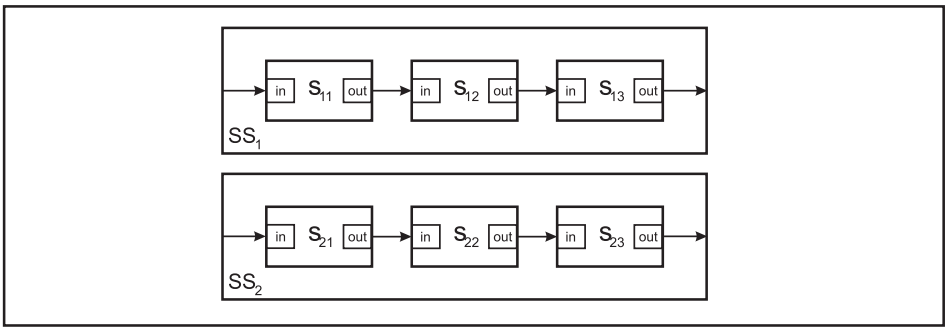
$$PCoupled = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle \tag{3.5}$$

Druhý rozdíl proti klasické definici spojované komponenty (vztah 3.3) spočívá v definici interních komponent modelu. Množina $\{M_i\}$ neobsahuje klasické atomické komponenty, ale atomické komponenty paralelní verze formalismu. I pro paralelní formalismus platí, že pro jakýkoliv spojovaný model lze zkonstruovat ekvivalentní atomický model. Proto i v paralelním prostředí může spojovaná komponenta obsahovat i jiné spojované komponenty. Význam a definice ostatních částí je stejná jako u spojované komponenty klasického DEVS formalismu (vztah 3.3).

3.2.5 Paralelní a distribuovaná simulace DEVS modelů

Tato sekce představuje několik způsobů paralelní/distribuované simulace DEVS modelů. V dalším textu této kapitoly předpokládejme, že model je rozdělen na komponenty a každá komponenta je zpracovávána na jednom procesoru.

Uvažujme model se dvěma komponentami, které nejsou mezi sebou spojeny (obrázek 3.4). U tohoto modelu nezáleží na pořadí zpracování komponent, či zda komponenty budou zpracovány paralelně, výsledek bude vždy stejný. Taková nezávislost komponent modelu ale není pravidlem. Kom-



Obrázek 3.4: *Nezávislé komponenty, které mohou být simulovány paralelně*

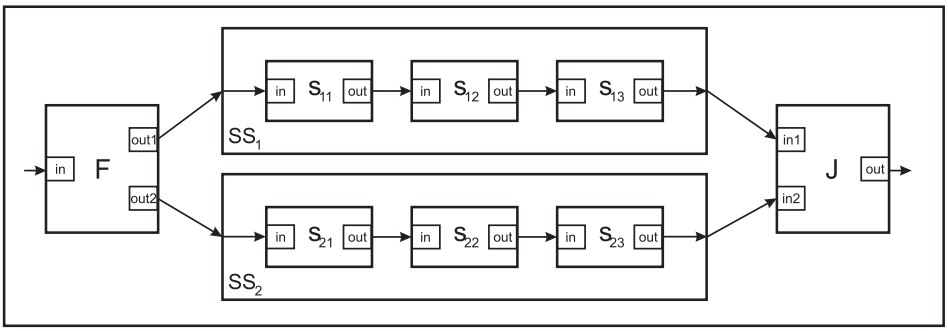
ponenty mohou obvykle pracovat nezávisle pouze po limitovaný čas a poté se mezi nimi projevují závislosti. Závislost se v diskrétních modelech projevuje jako závislost mezi událostmi. Pokud komponenta modelu přímo či nepřímo ovlivňuje události v jiné komponentě modelu, říkáme, že je zde *kauzální závislost* mezi událostmi. Pro korektnost simulace je nutné tuto kauzalitu dodržet, příčiny v modelu musí předcházet efektům. Pro dvě události E_j a E_i platí, že pokud je událost E_j kauzálně závislá na události E_i, musí být událost E_i zpracována před událostí E_j. Událost E_j je kauzálně závislá na události E_i pokud:

- událost E_i plánuje událost E_j
- událost E_i modifikuje stav komponenty, který se využívá i v události E_j
- událost E_i plánuje událost E_k a událost E_j je kauzálně závislá na události E_k

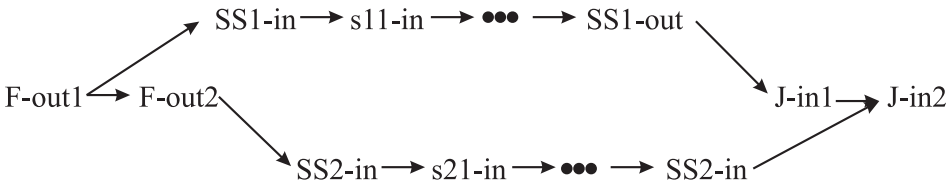
Z toho vyplývá, že relace kauzální závislosti mezi událostmi je tranzitivní. Pokud událost E_k je kauzálně závislá na E_i a E_j je kauzálně závislá na E_k, potom je událost E_j kauzálně závislá na E_i. Dále je třeba, aby relace kauzální závislosti událostí nebyla cyklická a čas přiřazený události E_i nesmí být menší, než čas přiřazený jakékoliv na ni kauzálně závislé události E_j.

Ve spojované komponentě jsou závislosti mezi komponentami vytvářeny výhradně vazbami mezi výstupy a vstupy komponent. Model na obrázku 3.5 obsahuje čtyři komponenty. Komponenty SS_1 a SS_2 mají vstupy připojeny na výstupy komponenty F a jejich výstupy jsou připojeny na vstupy komponenty J. Na obrázku 3.6 je zobrazena relace kauzální závislosti mezi jednotlivými událostmi. Z obrázku je patrné, že mezi komponentami SS_1 a SS_2 nejsou přímé závislosti a jejich události mohou být proto zpracovány paralelně. Je ale nutné události komponent SS_1 a SS_2 synchronizovat na výstupu komponenty F a na vstupu komponenty J.

Bylo již publikováno mnoho metod paralelní/distribuované simulace DEVS modelů. Všechny tyto metody lze rozdělit do dvou základních kategorií. První kategorií jsou tzv. *konzervativní metody*. V těchto metodách jsou striktně dodržovány kauzální závislosti a není dovoleno jejich porušení. Druhou kategorií jsou tzv. *optimistické metody*. V těchto metodách je povoleno porušit kauzální závislost, je ale nutné toto porušení detekovat a opravit.



Obrázek 3.5: *Nezávislé komponenty SS_1 a SS_2, se synchronizačními body F a J*



Obrázek 3.6: *Relace kauzální závislosti událostí z př. na obrázku 3.5*

Konzervativní simulátor

Tuto metodu publikoval K. M. Chandy a J. Mistra [5] již koncem 70. let minulého stolení a od té doby bylo publikováno mnoho variant a vylepšení této metody. Metoda a všechny její varianty jsou založeny na stejné myšlence, striktně se vyvarovat porušení kauzality.

Fundamentálním problémem je v konzervativním simulátoru rozhodnutí, zda simulátor může zpracovat událost, zda je její zpracování bezpečné. To znamená, že bude zaručeno, že se na vstupu již neobjeví událost s časem předcházející čas právě zpracovávané události. Simulátor nezpracovává příchozí události ihned, ukládá tyto události do fronty seřazené dle jejich času. Dále simulátor uchovává čas poslední vstupní události t_{ip} ze všech vstupních portů. Tento čas reprezentuje čas události, která bude zpracována a je minimálním časem ve frontě. Pokud je fronta událostí prázdná, t_{ip} reprezentuje čas poslední zpracované události. Při inicializaci je tento čas nastaven na 0.

Události jsou v simulátoru zpracovávány striktně ve správném časovém pořadí. Pokud je komponenta ovlivňována pouze jedinou komponentou, její vstupní události musí být také ve správném časovém pořadí. A proto platí, že pokud je komponenta ovlivňována pouze jedinou komponentou, je garantováno že, zpracování přijaté události je bezpečné a může být ihned po přijetí zpracována. Problém nastává, pokud je komponenta ovlivňována více komponentami.

Pokud uvažujeme komponenty z obrázku 3.5, komponenta J příjme vstupní událost z komponenty SS_2 v čase t_{in2}, ale poslední vstupní událost od SS_1 byla v čase t_{in1} a platí $t_{in1} < t_{in2}$. Není tedy zaručeno, že komponenta J neobdrží vstupní událost z komponenty SS_1 v čase mezi t_{in1} a t_{in2}. Simulátor komponenty J je z tohoto důvodu blokován, dokud eventuálně neobdrží vstupní událost od komponenty SS_1. Obecně je simulátor blokován, dokud neobdrží vstupní události od všech komponent, které jí ovlivňují. Tento problém se v konzervativní simulaci stává kritický, pokud nejsou události v modelu distribuovány rovnoměrně. Například pro model na obrázku 3.5 bude kritické, pokud budou události z komponenty F zasílány pouze komponentě SS_1, potom bude komponenta J dostávat vstupní události pouze na port $in1$. Druhý vstup $in2$ zůstane prázdný. Simulátor nemůže garantovat bezpečnost provedení událostí z portu $in1$, a tak zůstane simulátor blokovaný a nemůže pokračovat.

Problém blokování simulátoru lze odstranit využitím tzv. *prázdných zpráv* a dodáním nové vlastnosti komponentě označovanou jako *lookahead*. Tato nová vlastnost komponenty reprezentuje schop-

nost komponenty předpovědět, že nebude v budoucnu po jistý časový interval produkovat žádnou výstupní událost. Pomocí prázdné zprávy informuje komponenta ostatní ovlivňované komponenty, že se na jejím výstupu nevyskytne výstupní událost. Komponenty tuto informaci využijí při synchronizaci a rozhodování, zda je již bezpečné zpracovat událost. Algoritmy simulace DEVS modelu pomocí konzervativních simulátorů lze nalézt v [50].

Optimistický simulátor

Narozdíl od konzervativního simulátoru podstupuje optimistický simulátor riziko možnosti porušení kauzální závislosti. Dovoluje zpracovat události, u kterých nelze garantovat bezpečnost jejich zpracování. Díky tomuto chování může od ovlivňujících komponent obdržet zprávu s událostí, která má menší čas, než je lokální čas komponenty. Takové události jsou označovány jako *straggler* události. Jejich výskyt naznačuje, že došlo k porušení kauzality a musí být opraveno. Oprava se provádí obnovením stavu komponenty, ve kterém se nacházela před staggler událostí, a anulováním všech odeslaných výstupních zpráv s časem vyšším, než je čas staggler události. Abychom tento *rollback* komponenty, jak bývá obnovování stavu komponenty nazýváno, mohli provést, je nutné v simulátoru ukládat historii stavů komponenty a historii všech vstupních/výstupních zpráv. Anulování výstupních událostí se provádí pomocí speciálních zpráv tzv. *antizpráv*. Ukládání informací pro provedení rollback operace reprezentuje hlavní režii systému v optimistické simulaci, především jsou kladeny vysoké nároky na paměť. V průběhu simulace může být volná paměť velmi rychle vyčerpána. Pokud lze garantovat, že bude požadován rollback do nějaké minimální hodnoty času, jsou uložené informace s nižší hodnotou času již nepotřebné a lze je proto smazat. Operace smazání nepotřebných uložených informací je nazývána *sbírání fosílií (fossil collection)*. Minimální hodnotu času lze získat jako minimum času poslední události v modelu minimální hodnota času odeslané zprávy. Tento přístup pro paralelní simulace DEVS modelů je znám jako *Time–Warp DEVS simulátor* [50].

Bylo publikováno několik alternativ Time–Warp simulátoru. Tyto alternativy se snaží redukovat režii způsobenou antizprávami. V podstatě se všechny zabývají základní otázkou, zdali mohou být odeslány výstupní zprávy, nebo má být jejich odeslání pozdrženo aby bylo redukováno riziko jejich anulování. Extrémní případ takového přístupu reprezentuje *bezrizikový optimistický paralelní simulátor* [50]. V tomto případě simulátor pozdrží odesílané zprávy mezi procesory, jejichž zpracování by bylo stále potencionálně nebezpečné. V rámci jednoho procesoru je ale simulace prováděna optimisticky tzn., že porušení kauzality se vyskytne pouze lokálně a poté je pouze lokálně prováděn rollback. Tento přístup pro paralelní simulaci DEVS modelu se doporučuje aplikovat pro víceprocesorové výpočetní systémy se sdílenou pamětí.

Paralelní simulátor pro P–DEVS formalismus

Paralelní verze DEVS formalismu byla navržena pro maximální využití paralelismu na úrovni spojované komponenty bez nutnosti využití optimistického způsobu zpracování. Jedná se o konzervativní metodu simulace. Narozdíl od metod paralelní/distribuované simulace uvedených v předchozích sekcích tato metoda nevyžaduje ukládat informace pro obnovení stavu, nedochází k blokování simulátorů, a proto metoda nevykazuje režii s tím spojenou. Tato metoda simulace je využita v navržené architektuře.

Stejně jako u abstraktního simulátoru pro klasický DEVS formalismu, i zde je pro atomickou komponentou simulátor a pro spojovanou komponentu koordinátor. Abstraktní simulátor pro P–DEVS používá čtyři zprávy: $(@, t)$ a $(done, t)$ jsou využívány k synchronizaci, (y, t) a (q, t) pro přenos dat. Simulátor pro atomickou P–DEVS komponentu (algoritmus 4) používá jednu zprávu $(*, t)$ pro synchronizaci tří přechodových funkcí atomického modelu. Pokud na vstupu komponenty nejsou žádná

data a $t_n \neq t$, zpracuje simulátor externí přechodovou funkci komponenty. V případě, že na vstupu nejsou data a $t_n = t$, zpracuje interní přechodovou funkci, pokud na vstupu jsou data a $t_n = t$, zpracuje funkci souběhu.

Abstraktní simulátor pro spojovanou P–DEVS komponentu (algoritmus 5). Hlavním rozdílem oproti klasické verzi pro DEVS, kde se komponenty zpracovávají sekvenčně a pro výběr zpracovávané komponenty je použita funkce $select$, je zde možnost paralelního zpracování komponent.

Algoritmus 4: Algoritmus simulátoru pro atomickou P–DEVS komponentu

Data:
$parent$ rodičovský koordinátor, kam jsou zasílány zprávy
t_l čas poslední události, t_n čas další události
$DEVS$ komponenta s totálním stavem (s, e)
y vystupní hodnota z příslušného modelu

when $message(@, t)$ **do**
  if $t \neq t_n$ **then**
    error: bad synchronization;
  else
    $y = \lambda(s)$;
    send $message(y, t)$;
    send $message(done, t)$;
  end
end

when $message(q, t)$ **do**
  lock bag;
  add q to bag;
  unlock bag;
  send $message(done, t)$;
end

when $message(*, t)$ **do**
  case $(t \geq t_n \; or \; t \leq t_l)$
    error: bad synchronization;
  end
  case $(t_l \leq t \leq t_n) \; and \; (bag \neq \emptyset)$
    $e := t - t_l$;
    $s := \delta_{ext}(s, e, bag)$;
    empty bag;
  end
  case $(t = t_N) \; and \; (bag = \emptyset)$
    $s := \delta_{int}(s)$;
  end
  case $(t = t_N) \; and \; (bag \neq \emptyset)$
    $s := \delta_{con}(s, bag)$;
    empty bag;
  end
  $t_l = t$;
  $t_N = t_l + ta(s)$;
  send $message(done, t_N)$;
end

3.3 Agentní systémy

Agentní systémy jsou interdisciplinárním oborem, který se zabývá studiem architektury a racionality systémů, které se skládají z entit (agentů). Opírají se o výsledky výzkumu v oboru umělé inteligence, počítačových věd, softwarového inženýrství, ale zasahují i do oborů přírodních a společenských. Z umělé inteligence přebírají tyto systémy především metody reprezentace a využívání znalostí, metody formalizace znalostních modelů pomocí výrazových prostředků speciálních logik a algoritmy strojového učení. Z počítačových věd čerpají multiagentní systémy poznatky z oblasti komunikačních prostředků, zvláště na nižších úrovních a metody využívané při distribuovaném programování. Sekce definují základní pojmy, které jsou třeba k zavedení pojmu agentní systém a agent agentního systému.

Pojem agentní systém je uváděn převážně ve spojení s prostředím, ve kterém agenti působí. *Agent* znamená aktivní prvek vybavený určitou dávkou inteligence, která umožňuje agentovi řešit zadaný problém. Popis agentního systému je rozdělen na dvě části. V první části je definován pojem agent a jeho charakteristické vlastnosti agentů. Ve druhé části bude definován pojem prostředí.

Algoritmus 5: Algoritmus koordinátora pro spojovanou P–DEVS komponentu

Data:
$parent$ rodičovský koordinátor
$event - list$ seznam elementů (d, t_{n_d}) seřazených dle t_{n_d} a $select$

when $message(@, t)$ **do**
  if $t \neq t_n$ **then**
    error: bad synchronization;
  else
    $t_l = t$;
    for *all imminent child* i *with minimum* t_N **do**
      send $message(@, t)$ to i;
      cache i in *synchronize* set;
    end
    wait until $(done, t)$'s are received;
    send $message(done, t)$ to $parent$;
  end
end

when $message(y, t)$ **do**
  for *all influencess* j *of child* i **do**
    $q = z_{i,j}(y)$;
    send $message(q, t)$ to child j;
    cache j in *synchronize* set;
  end
  wait until $(done, t)$'s are received;
  if $self \in I_i$ **then**
    $y = z_{i,self}(y)$;
    send $message(y, t)$ to $parent$;
  end
end

when $message(q, t)$ **do**
  lock bag;
  add event q to bag;
  unlock bag;
end

when $message(*, t)$ **do**
  if *not*($t_l \leq t \leq t_N$) **then**
    error: bad synchronization;
  else
    for *all receivers,* $j \in I_{self}$ *and all* $q \in bag$ **do**
      $q = z_{self,j}(q)$;
      send $message(q, t)$ to j;
      cache j in *synchronize* set;
    end
    empty bag;
    wait until $(done, t)$'s are received;
    for *all* i *in the* synchronize *set* **do**
      send $message(*, t)$ to i;
    end
    wait until $(done, t)$'s are received;
    $t_l = t$;
    t_N =min. of component's t_N's;
    clear *synchronize* set;
    send $message(done, t)$ to $parent$;
  end
end

Agent

V současnosti není možné v literatuře najít univerzální a unikátní definici pojmu agent. Obecná definice zní:

Definice 4 *Agent je skutečná či virtuální entita, která se vyskytuje v prostředí, ve kterém vykonává nějaké akce, která může vnímat a popisovat částečně toto prostředí, která může komunikovat s jinými agenty a která vykazuje autonomní chování, které je logickým důsledkem vlastního pozorování, znalostí a interakcí s ostatními agenty.*

Agent je tedy aktivní prvek systému vytvořený k předem zamýšlenému účelu. Ve skutečnosti jde o velice složitý umělý systém. Cílem agenta je transformace systému z aktuálního stavu do požadovaného cílového stavu. Existují tři základní přístupy, které používá člověk při řešení podobných úloh:

Algoritmus 6: Algoritmus kořenového koordinátora

Data:
t aktuální simulační čas;
$child$ podřízený koordinátor (simulátor);
begin
  $t := t_N$ of the $child$;
  while *not(konec simulace)* **do**
    send $message(@, t)$ to $child$;
    wait until $(done, t)$ is received;
    send $message(*, t)$ to $child$;
    wait until $(done, t_N)$ is received;
    $t = t_N$;
  end
end

1. Pokud zná postup ke změně, transformuje systém do požadovaného stavu pomocí tohoto postupu.
2. Konzultuje problém s jiným člověkem (expertem), o kterém věří, že by mu mohl s řešením pomoci. Získá–li potřebné informace, postupuje dle bodu 1.
3. Podrobí systém zkoumání, při kterém hledá vnitřní zákonitosti systému a postupně nabývá potřebné poznatky o systému. Získá–li potřebné informace, postupuje dle bodu 1.

Postupy dle bodů 2 a 3 lze kombinovat.

Aby byl schopen agent řešit problémy obdobným způsobem, měl by mít schopnost formulovat cíle a nalézat postupy akcí vedoucí k dosažení těchto cílů. I když se tato vlastnost jeví jako samozřejmá, není ve skutečnosti nutnou podmínkou aby se chování agenta jevilo jako inteligentní. Často lze totiž za inteligentní chování považovat i takové chování, které je řízeno pouze reflexí na vnější podněty. V každém případě však agent musí být schopen získávat informace o stavu prostředí, ve kterém pracuje a na základě těchto informací jednat.

Obecné dělení agentů Agenty můžeme obecně rozdělit takto:

- Biologický agent (člověk)
- Technický agent (robot)
- Programový agent (softbot)
  - Agenti v počítačových hrách, počítačové viry, agenti pro specifické úlohy, agenti jako entity umělého života

V dalším textu práce budou pod pojmem agent uvažováni pouze programoví agenti bez ohledu na jejich výše uvedené rozlišení. Agenti jsou realizováni formou implementovaných kódů a algoritmů a bývají buď ve formě mobilní, kdy jsou schopni měnit své umístění a pracovat na různých platformách, nebo ve formě statické, kdy je tvoří aplikace schopné poskytovat zdroje, služby ap. na některém pevném místě v síti, nebo v samostatném počítači. Implementačním jazykem těchto agentů je v současné době převážně jazyk *Java*. Architektura programového agenta je dnes také standardizována (standard FIPA) a bude uvedena na konci této kapitoly.

Nejčastěji uváděnou společnou vlastností agentů je autonomie.

Definice 5 *Autonomie je vlastnost agenta, spočívající v samostatnosti rozhodování o svém chování v rámci daného systému, bez implicitní závislosti na jakýchkoliv jiných prvcích tohoto systému.*

Agent inteligentní Samotné slovo *inteligentní* navozuje představu, že agent řeší úlohy inteligentním způsobem. Tím se myslí agentova schopnost plnit cíle, které jsou v jeho zájmu a to pomocí jeho vlastní „inteligence“, ve většině případů logickou dedukcí. Agentovy záměry budou převážně souviset s účelem jeho vzniku. Agent je však nemusí mít vždy pevně zabudované, ale může je také přejímat od jiných agentů a pak se jimi řídit.

Agent reaktivní Tento agent bezprostředně reaguje na jisté změny prostředí (nebo své změny vůči prostředí), aniž by měl vnitřní reprezentaci znalostí o tomto prostředí. Jeho reakce nejsou výsledkem výpočtů či dedukcí na základě znalostí, ale pouze reakcemi na podněty. Reaktivita je vedle autonomity druhou významnou vlastností agentů.

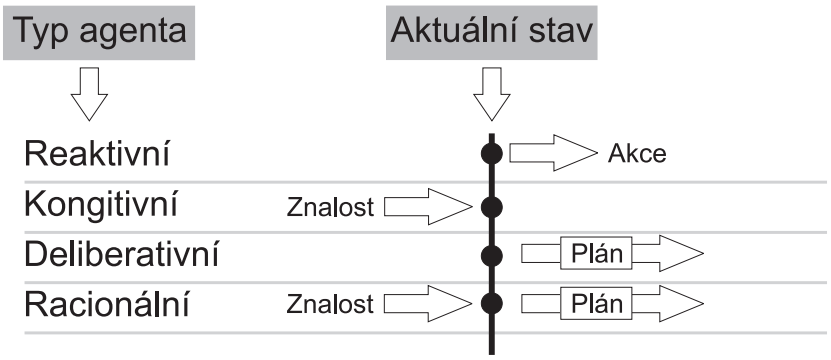
Definice 6 *Reaktivita je vlastnost agenta, spočívající v jeho reakci na změny prostředí tak, aby dosáhl cíle, pro který byl navržen.*

Agent deliberativní Na rozdíl od reaktivního agenta má tento typ agenta schopnost plánovat postup svých akcí vedoucích k dosažení zvolených nebo zadaných záměrů/cílů. To znamená, že agent musí mít schopnost různých výpočtů, což bude později v textu chápáno jako druh vnitřní činnosti agenta. K dosažení svých záměrů pak agent ovlivňuje okolní prostředí tak, aby získal nějakou výhodu. Toto jednání je další z často uváděných vlastností agentů a nazývá se proaktivita.

Definice 7 *Proaktivita je vlastnost agenta, spočívající v jeho ovlivňování okolního prostředí tak, aby jeho stav usnadňoval dosažení agentových záměrů.*

Agent kognitivní Kognitivní agent má schopnost vyvozovat logické závěry ze svých pozorování svého okolního prostředí. Takový agent musí především být schopen se učit a vytvářet si svou vlastní bázi znalostí. Do ní si během svého působení ukládá informace získané interakcí s okolím nebo znalosti získané dedukcí. Kognitivní agent nemusí mít nutně deliberativní schopnosti. Pak provádí pouze vnitřní akce, například analyzuje scénu, provádí překlad, nebo získává znalosti (dolování znalostí z dat).

Agent racionální Racionální agent má všechny výše uvedené vlastnosti a jeho struktura obsahuje jak plánovací jednotku, tak i kognitivní jednotku včetně báze znalostí. Je to agent, který je na základě svých poznatků schopen se učit a pak plánovat svoji činnost tak, aby dosáhl svých cílů racionálním způsobem. Stojí nejvýše na pomyslné hierarchii uvedených agentů (obrázek 3.7).



Obrázek 3.7: *Rozdělení agentů*

Definice 5, 6 a 7 odpovídají definicím uvedeným v [41] či [46].

Prostředí

Prostředí je vše, s čím agent přichází během své činnosti do styku. Je to tedy agentní systém bez jediného svého prvku–agenta. Prostředí z hlediska agenta pak může být:

- *Plně pozorovatelné/částečně pozorovatelné*
  Prostředí je pro agenta plně pozorovatelné, pokud může svými senzory sledovat jeho celý stav.
- *Statické/dynamické*
  Prostředí je pro agenta statické, pokud se může měnit pouze jeho akcemi.
- *Deterministické/nedeterministické*
  Prostředí je deterministické, jestliže je jeho stav po vykonání nějaké akce dán pouze touto akcí a předcházejícím (původním) stavem tohoto prostředí.

- *Diskrétní/spojité*
  Prostředí je diskrétní, pokud má konečně nebo spočetně mnoho stavů. Pro agenta založeného na číslicových počítačích/procesorech je každé prostředí diskrétní. Údaje ze senzorů se totiž ukládají do konečného binárního řetězce a proto existuje pouze konečná množina stavů tohoto prostředí.

3.3.1 Standardizace v oblasti agentních systémů

Vzhledem k rychlému vývoji v oblasti distribuované umělé inteligence a agentních systémů bylo nutné přikročit k zavádění standardů. Architektura starších systémů té doby byla vyvíjena spíše intuitivně, často neumožňovala plánovat a realizovat složitější systémy s velkým počtem agentů. Rovněž vyvstala potřeba sjednocení komunikace agentů. Bylo zřejmé, že znovupoužitelnost a vzájemná propojitelnost jednotlivých systémů by mohla rozšířit možnosti jejich využití.

Prvním pokusem o standardizaci byl jazyk pro komunikaci agentů KQML (*Knowledge Query and Manipulation Language*) [30]. V roce 1996 vznikla mezinárodní nezisková organizace FIPA (*Foundation for Intelligent Physical Agents*) [33], která produkuje dokumenty. Tyto dokumenty mají být standardy v oblasti agentních systémů. Záběr standardů této organizace je široký a vede od specifikace životního cyklu agenta, mobility agentů, komunikačních jazyků, přes ontologie, komunikační protokoly až po bezpečnost v agentních systémech. Výchozím prvkem všech specifikací je FIPA architektura agenta. V současné době již většina nově vyvíjených agentních systémů tyto standardy podporuje. Jedním z důvodů této podpory je i snadná dostupnost několika implementací koster agentních systémů. Tyto kostry agentního systému dávají uživateli k dispozici jádro systému, zahrnující jak komunikační protokoly, tak i podpůrné služby, kostry agentů a nástroje pro vývoj a ladění nových systémů. Uživatel se tak může soustředit především na vlastní funkce systému, což velice urychluje a usnadňuje vývoj agentního systému. Jedním z takto dostupných systémů je JADE [36], vyvíjený na univerzitě v Parmě ve spolupráci s italskými telekomunikacemi.

3.3.2 FIPA

FIPA (*Foundation for Intelligent Physical Agents*) [33] je mezinárodní nezisková organizace, která je zaměřená na podporu průmyslu inteligentních agentů vývojem specifikací podporujících interoperabilitu mezi agenty a aplikacemi na agentech založených. Organizace produkuje dokumenty, které mají být standardy v oblasti agentních systémů. Tyto specifikace vznikají spoluprací mezi členy organizace, jimiž jsou firmy a univerzity, které se zabývají oblastí agentních systémů. Všechny navržené specifikace jsou volně přístupné.

Postupně vznikly specifikace FIPA 97, FIPA 98 a FIPA 2000. V prvních dvou skupinách byly zahrnuty specifikace agentní infrastruktury, aplikací, komunikačního jazyka, služeb a pomocné ontologie. S rozvojem agentních technologií stála FIPA před problémem, zda pravidelně revidovat jednotlivé specifikace a doplňovat nové vlastnosti, nebo zda zvolit abstraktnější přístup. Rozhodla se pro druhou možnost a FIPA 2000 tedy popisuje abstraktní architekturu, která může zahrnovat širokou oblast používaných mechanismů, jako jsou různé způsoby přenosu a reprezentace zpráv, řídící služby, apod. Je tedy umožněn vývoj agentů těsněji spjatých s jejich výpočetním prostředím, kteří spolupracující s agenty v jiných prostředích. FIPA rozděluje specifikace do následujících skupin: *abstraktní architektura*, *aplikace*, *správa agentů*, *přenos zpráv mezi agenty* a *komunikace agentů*. Následující sekce kapitoly popisují tyto skupiny detailněji.

Abstraktní architektura

Architekturu prvků agentního systému, jejich jednotlivé elementy a jejich vazby charakterizuje jedna ze základních FIPA specifikací [31]. Cílem tohoto dokumentu je sjednotit architekturu agentních systémů a umožnit snadnou znovupoužitelnost prvků a také schopnost vzájemné spolupráce systémů užívajících různé technologie.

Největší význam má zavedení sémantiky pro komunikace mezi agenty, kteří mohou využívat různé metody přenosu zpráv (*MT–Messaging Transports*), různé jazyky pro meziagentní komunikaci (*ACL–Agent Communication Language*) a různé jazyky obsahu zprávy (*content languages*). Proto je nutné tyto systémy popsat velmi abstraktně. Návrh architektury je rozdělen do následujících celků:

- Subsystém služeb, registrace a zjišťování služeb dostupných agentům a ostatním službám
- Výměna zpráv, podpora technologií pro zasílání a příjem
- Reprezentace zprávy v ACL jazycích
- Reprezentace informace v jazycích obsahu zpráv
- Podpora katalogových služeb (*directory services*)

Z důvodu abstrakce návrhu je možné při realizaci jednotlivých elementů systému volit vlastní technologie a řešení. Například je možné systém katalogových služeb implementovat pomocí LDAP (*Lightweight Directory Access Protocol*) [27] a sdílet jej takto v celém agentním systému.

Služby Abstraktní architektura podle FIPA 2000 definuje dva typy podpůrných služeb [31] a těmi jsou: *řídící služby* a *služby spojené s přenosem zpráv*. Služby mohou být implementovány jako agenti nebo jako software ke kterému se přistupuje pomocí volání metod (např. *Java Remote Method Invocation* [34]). Agent, nabízející služby, je ve svém chování omezen více než obecný agent. Tento agent nemá volnost rozhodování, která agentům obyčejně přísluší. Například nemůže poskytování služeb libovolně odmítnout.

Řídící služby (Directory Services). Základním úkolem této služby je poskytnout místo, kde si budou agenti registrovat své katalogové záznamy. Ostatní agenti mohou v těchto záznamech vyhledávat agenty, se kterými chtějí spolupracovat.

Záznam každého agenta je tvořen minimálně dvojicí [jméno agenta, umístění agenta]. Dále může záznam obsahovat různé atributy, jako agentem nabízené služby, cena za jeho použití, omezení při používání, apod. Při vyhledávání zadá agent službě klíčové položky, které chce vyhledat. Služba poté prochází všechny záznamy a ty ve kterých se klíčové položky vyskytují předá zpět agentovi. Ten si s jejich pomocí může zvolit agenta, jehož parametry mu pro spolupráci vyhovují nejvíce. Protože záznam obsahuje i jméno a umístění zvoleného agenta, má hledající agent k dispozici všechny údaje potřebné pro případnou vzájemnou komunikaci.

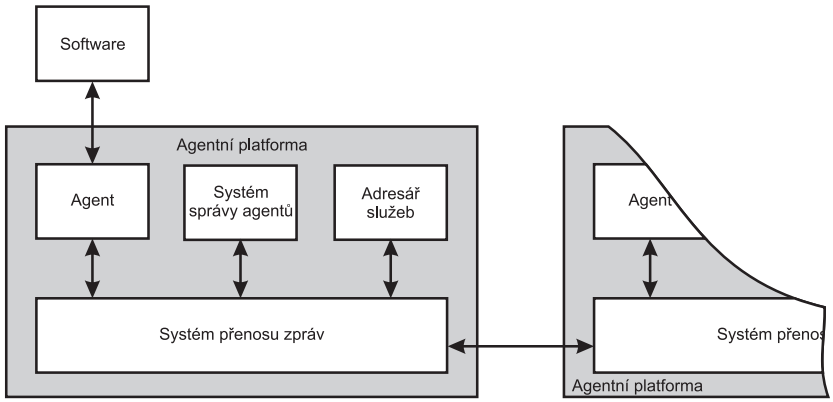
Služby spojené s přenosem zpráv (Message Transport Services). V agentních systémech definovaných podle FIPA agenti komunikují posíláním zpráv. Základním úkolem těchto služeb je poskytnout platformu pro zasílání a příjem zpráv mezi agenty. Tyto služby jsou povinnou částí pro každou konkrétní realizaci abstraktní architektury FIPA. Služby podporují tyto akce: *zahájit přenos*, *přerušit přenos*, *zaslání zprávy* a *doručení zprávy*.

Aplikace

Specifikace pro oblast aplikací popisují ontologii a specifikaci služeb pro konkrétní typy aplikací. Ty zahrnují monitorovací a řídící aplikace, osobní a cestovní asistenty, audiovizuální a mediální aplikace, poskytování a správu síťových služeb a správu kvality služeb. FIPA zde specifikuje také metody integrace stávajících aplikací s agentními systémy. Další informace lze nalézt v [31], [23].

Specifikace správy agentů

Správa agentů zajišťuje normativní prostředí, ve kterém agenti existují a operují. Poskytuje služby pro vytváření, registraci, umístění, komunikaci, migraci a ukončování činnosti agentů. Referenční model správy agentů dle specifikace FIPA je zobrazen na obrázku 3.8. Vlastní implementace není předmětem standardizace FIPA a je záležitostí vývojářů jednotlivých agentních systémů.



Obrázek 3.8: *Referenční model správy agentů dle specifikace FIPA*

Referenční model správy agentů se skládá z následujících logických komponent (detailní popis lze získat v [32]) :

Agent je výpočetní proces, který implementuje autonomii a podporu komunikace a komunikuje pomocí ACL (*Agent Communication Language*). Agent je základním prvkem agentní platformy (AP), která mu poskytuje různé služby, může zahrnovat přístup k externímu software, lidské obsluze nebo komunikačním prostředkům. Agent musí mít alespoň jednoho vlastníka a měl by mít v systému jednoznačnou identifikaci AID (*Agent Identifier*). Pro potřeby komunikace si může zaregistrovat libovolný počet transportních adres.

Adresář služeb (DF–*Directory Facilitator*) je nepovinná komponenta AP. Pokud je přítomna, implementuje službu tzv. žlutých stránek. Agenti mohou pomocí DF zaregistrovat své služby, podporované komunikační protokoly, jazyky apod. Ostatní agenti poté mohou pomocí dotazu zjistit seznam nabízených služeb. V rámci platformy AP může existovat více DF.

Systém správy agentů (AMS–*Agent Management System*) tento povinný prvek poskytuje služby pro kontrolu přístupu a používání platformy (AP). Každá platforma má právě jeden tento prvek. Agent se musí zaregistrovat pomocí AMS, aby obdržel svůj platný identifikátor (AID). AMS si udržuje seznam těchto identifikátorů, který také obsahuje transportní adresy agentů registrovaných na konkrétní agentní platformě. AMS se stará o vytváření a rušení agentů, rozhoduje o jejich dynamické registraci,

dohlíží na jejich přesuny a poskytuje tzv. službu bílých stránek, tj. vyhledávání jmen a adres agentů. Seznam služeb poskytovaných agenty poskytuje adresář služeb (DF).

Systém přenosu zpráv (MTS–*Message Transport System*) někdy též nazývaný *Agent Communication Channel (ACC)*, tvoří standardní metodu komunikace mezi agenty. Zprávy lze doručovat nejen v rámci aktuální AP, ale i agentům v ostatních AP. Kterýkoliv FIPA agent musí mít přístup alespoň k jedné MTS a všechny zprávy posílané přes MTS musí mít jako příjemce konkrétního agenta.

Agentní platforma (AP–*Agent Platform*) poskytuje fyzickou infrastrukturu do které jsou agenti umístěni. Platforma se skládá z počítačů, operačního systému, podpůrného software a prvků FIPA agent managementu. Vlastní vnitřní implementace platforem a agentů není předmětem standardizace v rámci FIPA a je záležitostí vývojářů jednotlivých agentních systémů. FIPA definuje pouze způsob komunikace mezi platformami a agenty z různých platforem, ale ne mezi agenty v rámci jedné platformy.

Komunikace agentů

Jazyky pro komunikaci mezi agenty se opírají o teorii řečového aktu, která vychází z analýzy komunikace v přirozeném jazyce. Model této komunikace podle FIPA je založen na předpokladu, že dva komunikující agenti používají stejnou ontologii, která určuje sémantiku zprávy v rámci dané komunikační domény. Za těchto podmínek pak agenti přisuzují výrazům (termům) ve zprávě stejný význam. Ke komunikaci v dané doméně je možné použít explicitní ontologie, nebo ontologie deklarativní (která pak musí být někde uložena). Za tímto účelem bývá v systému agent pro ontologii (*OA–Ontology Agent*). Speciálním případem je implicitní ontologie, která bývá zakódována přímo v softwaru, který ji využívá. Není tedy nutné ontologii externě ukládat.

FIPA vyvinula specifikace popisující komunikační jazyk na abstraktní úrovni, spolu s knihovnami předdefinovaných komunikačních (řečových) aktů, vyjednávacích a aukčních protokolů a jazyků obsahu zpráv. Tento jazyk, nazývaný FIPA–ACL, vychází do značné míry ze staršího standardu KQML (*Knowledge Query and Manipulation Language*) [30]. Lépe definuje sémantiku, komunikační protokoly a umožňuje složité strukturované vyjednávání agentů. V současné době je považován za standard mezi průmyslovými i vědeckými aplikacemi. FIPA–ACL definuje parametry, které musí a které může zpráva obsahovat.

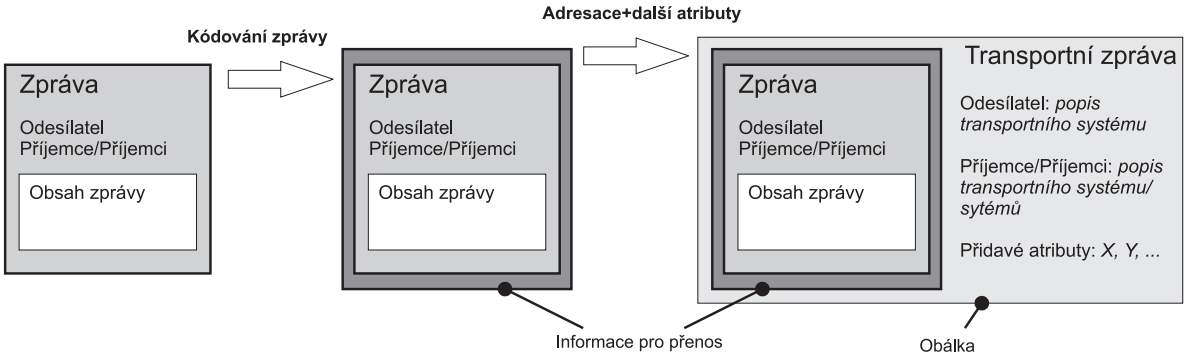
Přenos zpráv

V agentních systémech definovaných podle FIPA spolu agenti komunikují pomocí zpráv. Specifikace se zabývá strukturou zprávy a vlastním přenosem.

Struktura zprávy (*message structure*) zpráva je zapsána ve formátu ACL, např. FIPA–ACL. Vlastní obsah zprávy je vyjádřen pomocí jazyka obsahu zprávy. Formát obsahu zprávy může být popsán pomocí ontologie, vysvětlující jednotlivé části obsahu. Zpráva obsahuje jméno odesílatele a příjemce.

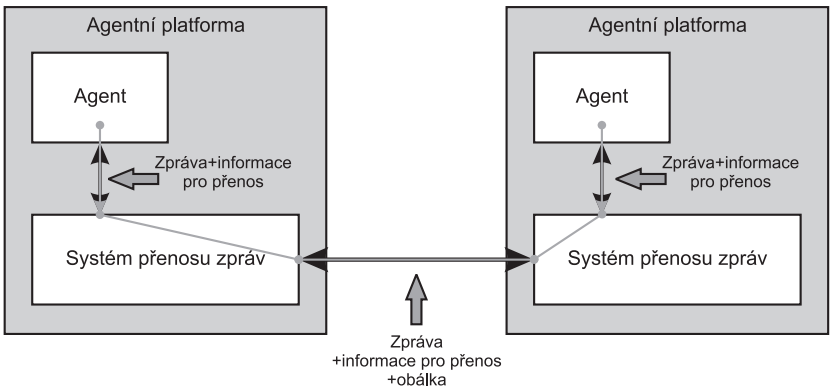
Přenos zprávy (*message transport*) po odeslání je zpráva zakódována do formátu vhodného pro přenos. K zakódované zprávě je připojena obálka (*envelope*) s informacemi potřebnými pro směrování a přenos zprávy. FIPA specifikace popisují několik protokolů pro přenos zpráv (*MTP–Message Transport Protocol*), jako IIOP, HTTP nebo WAP. Reprezentace obálky zprávy odpovídá použitému MTP, např. XML pro protokol HTTP.

Služba přenosu zpráv může v každém prostředí podporovat libovolný počet MTP a zajišťuje též překlad mezi standardními protokoly definovanými dle FIPA, používanými pro komunikaci mezi různými prostředími a protokoly užívanými v rámci jednoho prostředí.



Obrázek 3.9: *Kódování zprávy do transportního formátu*

Proces transformace a přenosu zprávy od jednoho agenta k druhému ilustrují obrázky 3.9 a 3.10 Zpráva od agenta vyjádřená v jazyce ACL je předána MTS. Zde je přidána obálka zprávy s informacemi pro směrování a přenos. Poté MTS odešle zprávu vhodným protokolem pro přenos zpráv cílovému agentovi nebo dalšímu MTS v rámci jiné agentní platformy. Každý MTS může do obálky zprávy přidávat vlastní informace, naopak žádná z informací nesmí být odebrána.



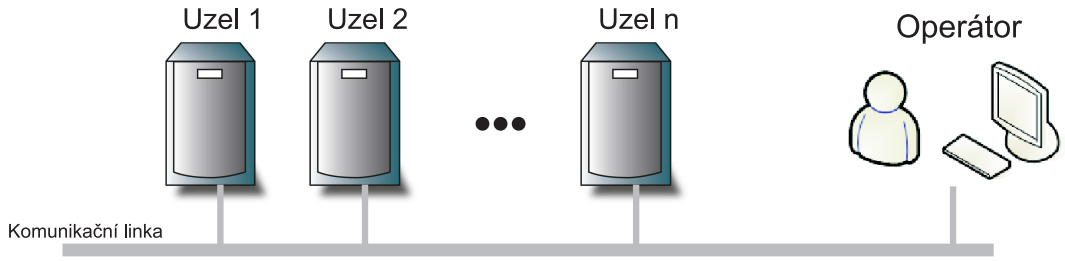
Obrázek 3.10: *Model přenosu zpráv mezi agenty*

Kapitola 4

Návrh architektury

Kapitola seznamuje čtenáře s návrhem architektury distribuovaného simulačního systému. Tato kapitola reprezentuje první část jádra disertační práce. V úvodu se zabývá obecnou architekturou distribuovaného systému a diskutuje přednosti a nevýhody tohoto typu simulace. Popisuje některé důležité problémy distribuovaného zpracování a prezentuje očekávání od navržené architektury. Po obecném úvodu představuje navrženou architekturu společně s krátkým popisem jednotlivých částí. Další sekce se detailněji zabývají částmi navržené architektury a službami. Jedna sekce kapitoly je věnována popisu způsobu vytváření modelu a vysvětlení odlišností od jiných projektů (např. DEVSJava [47], či DEVS/C++ [49]). Na ni navazuje sekce s popisem implementace simulace pomocí agentního systému. Další sekce se zabývá bezpečností distribuovaného systému a diskutuje možné útoky na systém včetně možné ochrany proti těmto útokům. Poslední sekce porovnává návrženou architekturu se simulačními systémy z kapitoly 2.2.

Obecná architektura distribuovaného simulačního systému je zobrazena na obrázku 4.1. Simulovaný model je rozdělen na několik částí a ty jsou rozmístěny na uzly distribuovaného systému. Na uzlech se provádějí simulace jednotlivých částí modelu paralelně. Uzly jsou fyzicky oddělené výpočetní systémy, které jsou propojeny komunikační linkou (nejčastěji počítačovou sítí). V průběhu simulace spolu jednotlivé uzly komunikují pomocí zpráv.

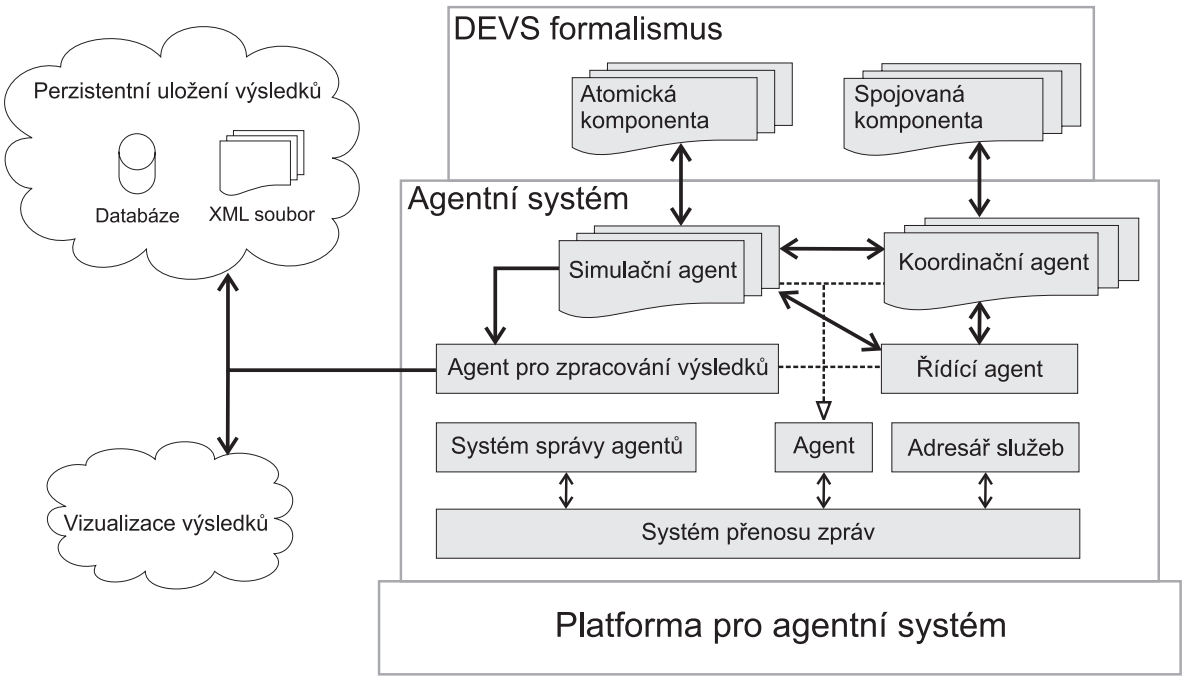


Obrázek 4.1: *Obecná architektura distribuovaného simulačního systému*

Z obecné koncepce distribuované simulace vyplývá jeden důležitý požadavek na simulovaný model. Model je nutné dekomponovat na části, které jsou poté simulovány na jednotlivých uzlech distribuovaného systému. Pokud zvolíme modelovací formalismus, ve kterém mají modely vhodnou strukturu (například hierarchickou či modulární), značně to zjednoduší práci s modelem v distribuovaném prostředí. Na druhé straně pokud ale zvolíme modelovací formalismus, ve kterém modely nemají takovou strukturu, mohlo by být velmi obtížné tyto modely dekomponovat. Při návrhu architektury a výběru modelovacího formalismu je nutné tento požadavek brát v úvahu.

Distribuovaná simulace je v dnešní době velmi zajímavý způsob simulace komplexních modelů. Například umožňuje získat vysoký výpočetní výkon, využít speciální hardwarové vybavení pro simulaci některých částí modelu apod. Některé tyto výhody byly již diskutovány v úvodní kapitole. Na druhé straně má ale i své nevýhody, první je samozřejmě již diskutovaný obtížný vývoj. Další nevýhodou je vysoká režie komunikace mezi uzly distribuovaného systému. I přes tyto nevýhody bude pravděpodobně využívána stále častěji. Existuje několik architektur pro distribuované simulační prostředí (nejznámější byly uvedeny v kapitole 2). Například HLA reprezentuje velmi zajímavou architekturu pro simulační systémy. Její základní myšlenkou je však propojení existujících simulačních systémů v jeden celek. Navržená architektura se přímo soustředí na distribuovanost a maximálně zjednodušuje vývoj distribuovaného simulačního prostředí. Umožňuje jednoduché využití umělé inteligence pro optimalizaci simulace.

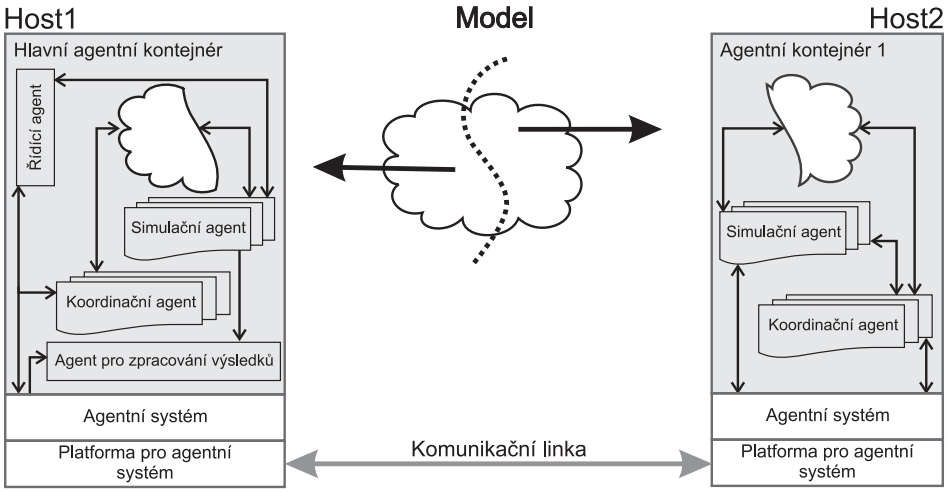
Navržená architektura je uvedena na obrázku 4.2. Jako základní modelovací formalismus byl zvolen P–DEVS formalismus (krátký úvod byl uveden v kapitole 3.2) především z důvodu jeho výhodných vlastností pro využití v distribuovaném prostředí. Modely popsané pomocí DEVS mají hierarchickou strukturu, a tak lze model jednoduše dekomponovat na menší části, které budou simulovány na jednotlivých uzlech distribuovaného systému. Veškerá komunikace jednotlivých částí modelu v průběhu simulace probíhá pouze pomocí zpráv. Pro tento formalismus byl definován i algoritmus simulace pro paralelním prostředí. Navíc se jedná o obecně známý modelovací prostředek.



Obrázek 4.2: *Architektura navrženého distribuovaného simulačního systému*

Distribuované zpracování v navržené architektuře zajišťuje agentní systém. Architektura agentního systému vychází ze specifikace FIPA (kapitola 3.3.2). Platforma pro agentní systém zde reprezentuje výpočetní systém, na kterém je možné agentní systém spustit a provozovat. Konkrétní požadavky na tuto platformu budou vycházet z konkrétní implementace agentního systému v navržené architektuře. Následující kapitola (kapitola 5) popisuje prototyp distribuovaného simulačního prostředí založeného na navržené architektuře. Jako implementační jazyk zde byl použit jazyk *Java*. Na agentní platformu je v tomto případě kladen pouze jediný požadavek, musí obsahovat interpret jazyka *Java*.

V architektuře uvedené na obrázku 4.2 jsou zobrazeny pouze základní typy agentů, kteří jsou nezbytně nutné k simulaci. *Řídící agent* je hlavní agent simulace, který inicializuje a ukončuje simulaci. Jemu jsou podřízení ostatní agenti simulace (*koordinační* a *simulační agent*). Agenti pro simulaci obsahují implementaci abstraktního simulátoru pro paralelní verzi DEVS formalismu (algoritmus 5 a 4). Princip distribuovaného zpracování založený na této architektuře je zobrazen na obrázku 4.3.



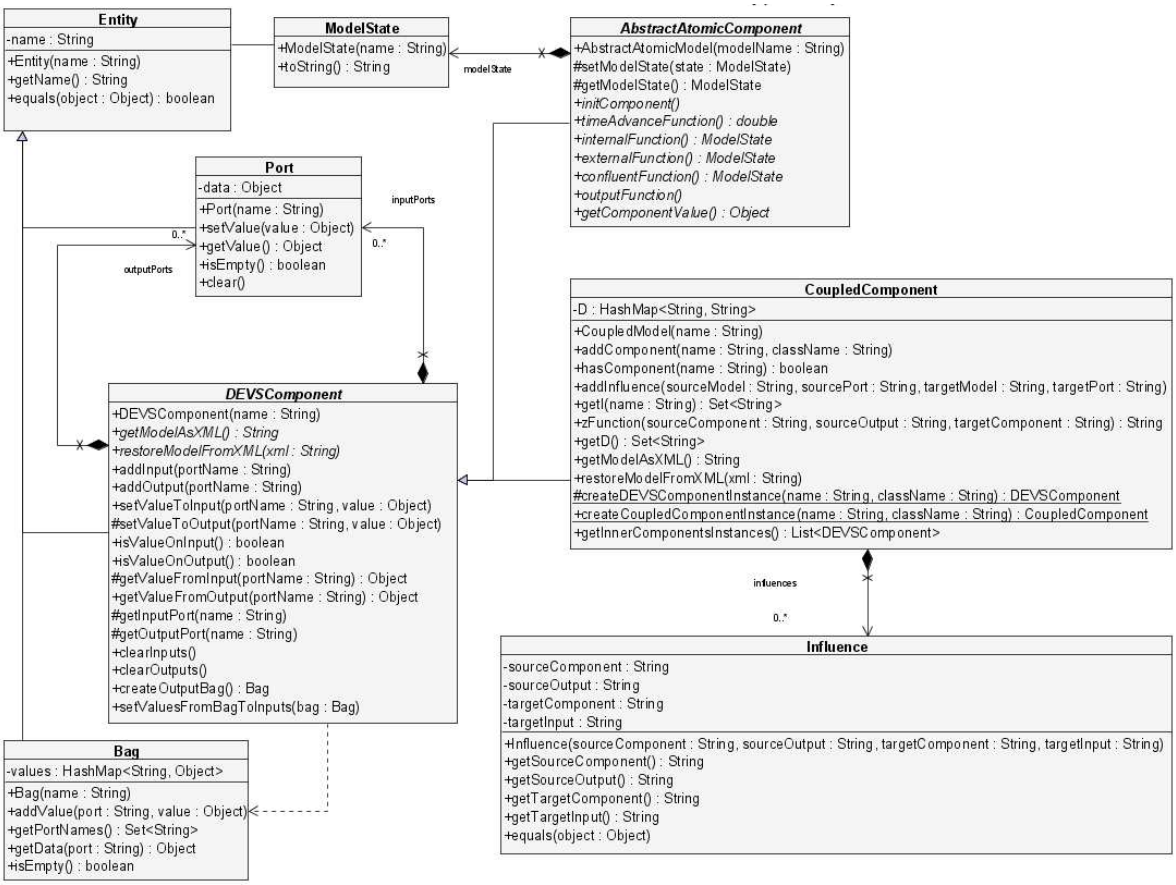
Obrázek 4.3: *Princip rozdělení modelu v navržené architektuře*

Pro každou atomickou komponentu je automaticky vytvořen simulační agent, který komponentu nese. Agentovi je přiřazen jednoznačný identifikátor (AID) na základě názvu atomické komponenty. Pro každou spojovanou komponentu je vytvořen koordinační agent. Stejně jako u simulačního agenta je koordinačnímu agentovi přiřazen jednoznačný identifikátor vycházející z názvu spojované komponenty. Koordinační agent nese spojovanou komponentu, má tak neustále k dispozici seznam jmen vnitřních komponent. Jelikož jsou identifikátory agentů odvozeny od názvu komponenty, má tak koordinační agent k dispozici seznam agentů, kteří simulují vnitřní komponenty spojovaného modelu. Díky tomuto seznamu může s těmito agenty komunikovat. V dalším textu budou agenti označováni jako podřízení agenti a koordinační agent spojované komponenty jako jejich nadřízený agent.

Prostředí, ve kterém agenti v agentním systému pracují, se nazývá agentní kontejner. Na každém uzlu distribuovaného systému je alespoň jeden kontejner. Během spuštění simulace se v hlavním kontejneru vygenerují simulační agenti dle postupu uvedeným v předešlém odstavci. Tím je v podstatě model rozdělen mezi jednotlivé agenty. Poté jsou simulační a koordinační agenti přesunuti na jednotlivé agentní kontejnery (uzly) distribuovaného systému. Simulace probíhá paralelně dle algoritmu pro paralelní DEVS formalismus. V příkladu zobrazeném na obrázku 4.3 systém obsahuje dva uzly a každý uzel jeden agentní kontejner. Model je rozdělen mezi tyto dva uzly. Agenti pak komunikují pomocí systému přenosu zpráv, pro agenty je tato komunikace transparentní. Následující sekce kapitoly podrobněji popisují navrženou architekturu.

4.1 Modelování

Implementace DEVS formalismu bude obdobná jako u ostatních projektů (například DEVSJava [47], či DEVS/C++ [49]). Model bude implementován jako kolekce tříd, které jsou potomky připravených tříd. Celá část systému pro modelování je zobrazena pomocí UML diagramu tříd na obrázku 4.4.



Obrázek 4.4: *Diagram tříd pro část systému implementující modelování*

Hlavní třídou v tomto diagramu je třída *DEVSModel*, která reprezentuje implementaci společných vlastností atomické a spojované komponenty P–DEVS formalismu. Jejím hlavním úkolem je implementovat podporu vstupních a výstupních portů komponenty. V definici atomické a spojované komponenty odpovídají množinám X a Y. Implementace portu je provedena pomocí třídy *Port*, kterou obsahují dva vektory třídy *DEVSModel* reprezentující vstupní a výstupní porty komponenty. Jednotlivé porty jsou identifikovány jménem portu. Zajímavé jsou abstraktní metody *getModelAsXML* a *restoreModelFromXML*. Tyto metody budou implementovány potomky této třídy a budou sloužit k ukládání a obnovení stavu celého modelu (viz. kapitola 4.3.3). Ostatní metody této třídy slouží k operacím s vstupně/výstupními porty a s daty na těchto portech.

Třída *AbstractAtomicModel* je implementací atomické komponenty P–DEVS formalismu. Tato třída obsahuje několik abstraktních metod, které reprezentují přechodové a transformační třídy atomické komponenty. Souvislost metod této třídy a funkcí v definici atomické komponenty (vztah 3.4) je shrnut v tabulce 4.1.

Implementace konkrétní atomické komponenty poté spočívá v implementaci těchto metod. Pro atomické komponenty je nutné implementovat metody *getModelAsXML* a *restoreModelFromXML*.

Třída *CoupledModel* je implementací spojované komponenty P–DEVS formalismu. Implementace této třídy se odlišuje od implementací obdobné třídy v projektech uvedených v kapitole 2.1. Třídy v těchto projektech obsahují přímé reference na objekty reprezentující vnitřní komponenty. Zde tomu tak není. Třída *CoupledModel* neobsahuje referenci na objekt vnitřní komponenty žádnou. Ob-

Funkce v definici atomické komponenty	Metoda třídy *AbstractAtomicModel*
ta	*timeAdvanceFunction*
δ_{int}	*internalFunction*
δ_{ext}	*externalFunction*
δ_{con}	*confluentFunction*
λ	*outputFunction*

Tabulka 4.1: *Vztah metod třídy AbstractAtomicModel s definicí atomické komponenty*

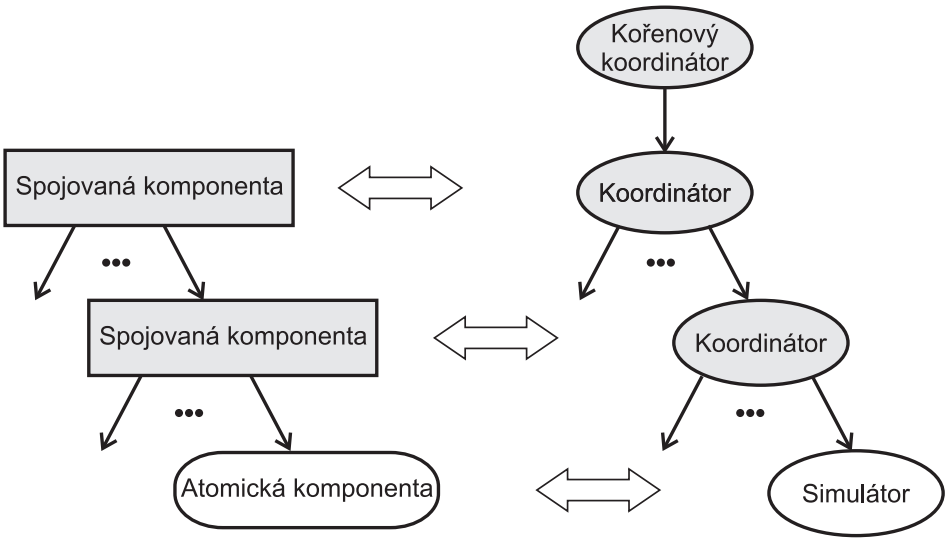
sahuje pouze seznam jmen vnitřních komponent společně s názvem třídy, který tuto komponentu implementuje. Tento přístup umožňuje poté velmi jednoduše model dekomponovat. Koordinační agent s sebou nese pouze potřebná data. Dle jména komponenty a názvu implementující třídy je dynamicky vytvořen objekt, který je přidělen simulačnímu či koordinačnímu agentovi na základě testu, zda je objekt potomek třídy *AbstractAtomicModel* nebo *CoupledModel*. Vytváření instancí je prováděno pomocí metod *createModelInstance* a *getInnerModelsInstances*. Implementace spojované komponenty poté spočívá pouze v definici seznamu vnitřních komponent (pomocí metody *addModel*) a definice propojení vnitřních komponent. Propojení se definuje pomocí metody *addInfluence*. Metoda vytvoří objekt reprezentující propojení a zařadí ho do vnitřního seznamu propojení. Z tohoto seznamu jsou poté vypočítávány množiny D nebo například $I_i, i \in D$.

4.2 Simulace

Z definice P–DEVS formalismu a algoritmů abstraktních simulátorů vyplývá, že simulace modelů je prováděna dvěma typy simulátorů (viz. kapitola 3.2.3). Každá atomická komponenta má přidělen vlastní simulátor a každá spojovaná komponenta vlastní koordinátor. Každý simulátor a koordinátor komponent, umístěných ve spojované komponentě, komunikuje výhradně s koordinátorem spojované komponenty, ve které je přidělená komponenta umístěna. Simulátory a koordinátory vytvářejí stejnou hierarchickou architekturu jako má simulovaný DEVS model. Situaci ukazuje obrázek 4.5. Celou simulaci řídí tzv. kořenový koordinátor, který má za úkol inicializaci a ukončení simulace (viz. kapitola 3.2.3). Simulace v navržené architektuře je prováděna pomocí agentů. A právě v tomto bodě dochází k propojení agentního systému a DEVS formalismu. V architektuře jsou definovány tři typy agentů.

Prvním typem agenta je *řídící agent*. Tento agent je zodpovědný za inicializaci, řízení a ukončení simulace. V chování agenta je implementován algoritmus kořenového koordinátora (algoritmus 6). Inicializace, řízení a ukončení simulace jsou hlavní činnosti tohoto agenta, avšak ne jediné. Agent dává svým podřízeným agentům pokyny pro uložení výsledků simulace, případně pokyn pro uložení stavu modelu, pokud je tato služba implementována. Další funkcí může být vyhodnocování výpadků uzlů systému (viz. kapitola 4.3.3). Při simulaci se v systému nachází pouze jediný agent tohoto typu.

Druhým typem agenta je *koordinační agent*. Koordinační agent je zodpovědný za koordinaci spojované komponenty. Pro každou spojovanou komponentu je agent automaticky vytvořen dle postupu uvedeného v předešlé sekci kapitoly, kterou sebou nese. Z tohoto důvodu spojovaná komponenta neobsahuje přímé reference na objekty vnitřních komponent, ale pouze seznam jmen vnitřních komponent. V chování agenta je implementován algoritmus koordinátora pro paralelní DEVS formalismus (algoritmus 5). Další funkcí agenta je rozesílat obdržené zprávy, například pokyn pro uložení stavu modelu apod. svým podřízeným agentům a čekat na jejich odpovědi. Po obdržení odpovědi od všech podřízených agentů oznámit svému nadřazenému agentovi dokončení operace. Jak již bylo uvedeno



Obrázek 4.5: *Hierarchická struktura simulátorů*

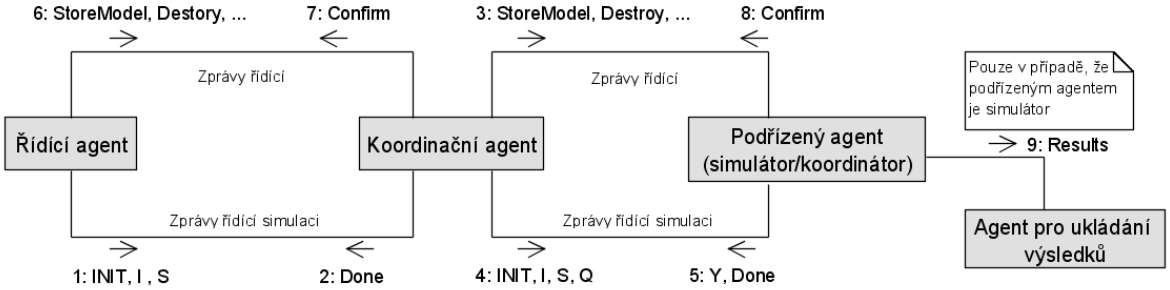
v předešlé kapitole, jsou identifikátory agentů odvozovány z názvů přiřazených komponent. Seznam identifikátorů podřízených agentů koordinační agent získá ze seznamu vnitřních komponent přiřazené spojované komponenty. Odvození identifikátoru agenta z názvu komponenty je závislé na konkrétní implementaci agentního systému.

Třetím typem agenta je *simulační agent*. Simulační agent provádí simulaci atomické komponenty. Pro každou atomickou komponentu je agent automaticky vytvořen dle postupu uvedeného v předešlé sekci kapitoly, kterou sebou nese. V chování agenta je implementován algoritmus simulátoru atomické komponenty pro paralelní DEVS formalismus (algoritmus 4). Hlavní funkcí agenta je výpočet času další události a zpracování vstupně/výstupních událostí komponenty, mezi další funkce může například patřit zpracování pokynu pro uložení stavu modelu.

Komunikace mezi simulačními agenty probíhá pomocí dvou typů zpráv. Prvním typem zpráv jsou tzv. *zprávy řídící*. Tyto zprávy slouží k řízení agentů a operacím, které se přímo netýkají průběhu simulace. Jedná se například o zprávu dávající agentovi pokyn k uložení výsledků simulace či k ukončení své činnosti a provedení sebedestrukce. Druhým typem zpráv jsou tzv. *zprávy řídící simulaci*. Tyto zprávy slouží k řízení průběhu simulace a jejich definice vychází přímo z algoritmů pro abstraktní simulátor pro P–DEVS formalismus.

Řídící agent využívá tři zprávy pro řízení simulace (*INIT*, *I* a *S*) a přijímá pouze zprávu o ukončení operace (*Done*) od koordinačního agenta. Koordinační agent s podřízenými agenty komunikuje pomocí čtyř zpráv pro řízení simulace (*INIT*, *I*, *S*, *Q*) a přijímá zprávu o ukončení operace podřízeného agenta (*Done*) a případnou informační zprávu o výstupní události vnitřní komponenty modelu (*Y*). Podřízeným agentem může být jiný koordinační agent či simulační agent.

Množství řídících zpráv je závislé na počtu implementovaných služeb. Například v případě potřeby uložení aktuálních výsledků simulace vydává řídící agent pokyn koordinačnímu agentovi pomocí zprávy *StoreModel*. Koordinační agent tuto zprávu rozešle podřízeným agentům. Pokud je podřízeným agentem simulační agent, zasílá zprávu s výsledky simulace agentovi pro uložení výsledků simulace. Podřízení agenti informují své nadřízené agenty o ukončení operace pomocí zprávy *Confirm*. Situace je zobrazena na obrázku 4.6. Na obrázku 4.7 je pomocí sekvenčního UML diagramu zobrazen příklad komunikace mezi agenty v průběhu simulace. Řídící agent zašle koordinačnímu agentovi inicializační zprávu *INIT* (zpráva č. 1). Koordinační agent získá z přiřazené spojované komponenty seznam jmen



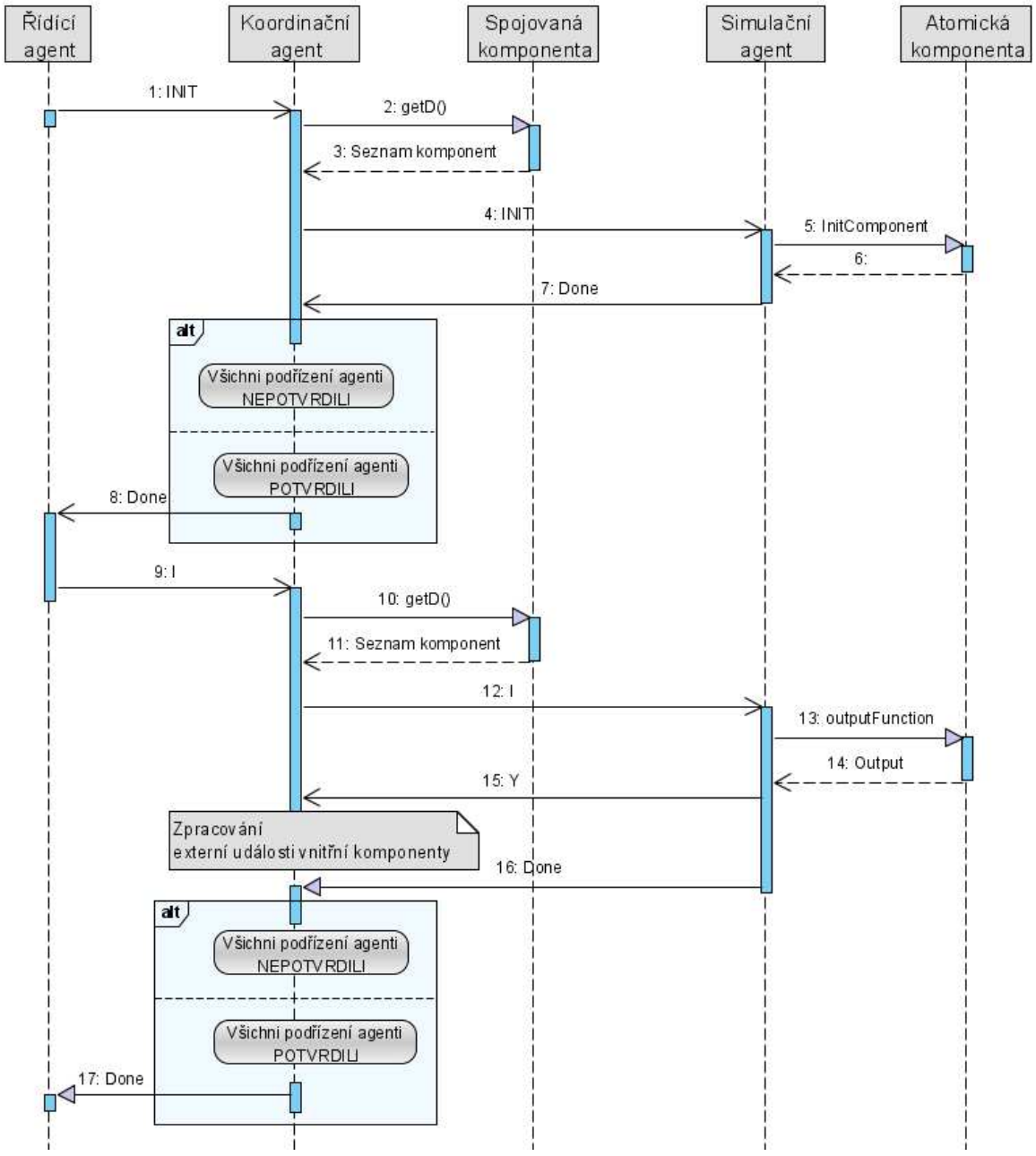
Obrázek 4.6: *Komunikace agentů pomocí zpráv v průběhu simulace*

vnitřních komponent (zprávy č. 2 a 3) a na základě tohoto seznamu vytvoří seznam příjemců inicializační zprávy podřízených agentů. Tuto zprávu jim zašle. V obrázku je uvažován pouze jeden podřízený agent (simulační agent), který tuto zprávu příjme (zpráva č. 4). Simulační agent provede inicializaci přiřazené atomické komponenty (zprávy 5 a 6), poté odešle nadřízenému agentovi informaci o provedení operace (zpráva č. 7). Koordinační agent po příjmu zprávy zkontroluje, zda již obdržel potvrzení od všech svých podřízených komponent. Pokud ano, odešle nadřízené komponentě (v tomto případě řídícímu agentovi) informaci o provedení operace (zpráva číslo 8). Koordinační agent postupuje přesně dle algoritmu pro kořenový koordinátor, odešle koordinačnímu agentovi zprávu *I* (zpráva č. 9). Koordinační agent opět získá seznam jmen vnitřních komponent a vytvoří seznam příjemců zprávy *I* (zprávy 10 a 11) a zprávu jim zašle. Simulační agent po obdržení této zprávy provede výstupní funkci atomické komponenty (zprávy 13 a 14). Pokud komponenta vyprodukovala výstup, oznámí tuto situaci nadřazenému agentovi pomocí zprávy *Y* (zpráva č. 15) a zašle ji informaci o ukončení operace (zpráva č. 16). Koordinační agent po příjmu zprávy zkontroluje, zda již obdržel potvrzení od všech svých podřízených komponent. Pokud ano, odešle nadřízené komponentě (v tomto případě opět řídícímu agentovi) informaci o provedení operace (zpráva číslo 17). Tímto způsobem jsou v chování agentů implementovány algoritmy pro simulaci P–DEVS komponent. Konkrétní způsob implementace chování agentů je závislý na konkrétní implementaci využívaného agentního systému.

4.3 Služby v systému

Implementace podpůrných služeb (např. záznam výsledků simulace či ukládání informací o průběhu simulace) je v tradičních simulačních systémech poměrně triviální záležitost. Všechny části systému mají stejný přístup ke zdrojům výpočetního systému (například k souborovému systému). V distribuovaném prostředí tomu tak ale být nemusí. V prostředí, kde mohou být jednotlivé části modelu rozprostřeny na fyzicky oddělených výpočetních systémech, část systému umístěna na jednom z uzlů nemusí mít přístup kde zdrojům výpočetního systému jiných uzlů. Implementace těchto podpůrných služeb se v distribuovaném prostředí stává složitou.

Při návrhu architektury jsem se maximálně snažil využít prostředky agentního systému. Jednotlivé služby jsou implementovány jako agenti, kteří svoje služby v systému nabízejí prostřednictvím *adresáře služeb* (DF) (viz. kapitola 3.3.2). Zde zaregistrují název nabízené služby společně se svým identifikátorem. Pokud chce komponenta systému využívat některou z podpůrných služeb, v adresáři služeb si příslušnou službu vyhledá společně s identifikátorem agenta, který tuto službu implementuje. S tímto agentem poté komunikuje již přímo pomocí zpráv. Tento přístup řešení je jednoduchý a velice flexibilní.



Obrázek 4.7: *Příklad komunikace agentů v průběhu simulace*

4.3.1 Záznam výsledků simulace

Záznam výsledků simulace je nejdůležitější podpůrnou službou. Existuje mnoho způsobů a formátů pro uložení výsledků simulace (uložení do souboru, do databáze, ve formátu XML,...). Z tohoto důvodu obsahuje návrh architektury abstraktní třídu (*StoringAgent*) pro agenta, který tuto službu bude implementovat. Agent implementuje službu, která je v adresáři služeb označená jako DEVS–STORING.

Tato třída obsahuje tři abstraktní metody. První metodou je *beforeSetup*, tato metoda bude automaticky volána při inicializaci agenta. Je určena k inicializaci zdrojů, které budou třeba k ukládání výsledků (například otevření souboru, připojení k databázi apod.). Druhou metodou je metoda *handleData*, která bude volána pokaždé, kdy agent obdrží data k uložení. Data jsou metodě předána ve formě čtyř parametrů, modelový čas, název modelu, vlastní data pro uložení a identifikátor agenta, který zprávu s daty zaslal. V této metodě bude docházet k vlastnímu ukládání dat. Poslední metodou je *beforeTakeDown*, která slouží ke korektnímu ukončení práce se zdroji a je automaticky volána před odstraněním agenta ze systému.

Chování agenta je poměrně jednoduché. Při inicializaci agent zaregistruje svoji službu v adresáři služeb a poté pouze přijímá zprávy obsahující data k uložení, název modelu, modelový čas. Při každém obdržení takové zprávy poté volá metodu *handleData*.

Pokud chceme mít výsledky simulace vizualizované již v průběhu simulace a zbytečně příliš nezatěžovat systém další komunikací, je možné tuto vizualizaci provádět na základě dat získaných v metodě *handleData*. Tento přístup je vhodný, i pokud je nutné provádět složitou vizualizaci, která spotřebuje mnoho výpočetního výkonu jako je například zobrazování realistické 3D scény. V tomto případě je možné agenta umístit na speciální uzel distribuovaného systému, který nebude využíván k simulaci, ale bude sloužit pouze pro online vizualizaci výsledků (může se jednat i o výpočetní systém se specializovaným hardwarovým vybavením).

4.3.2 Informace o průběhu simulace

Tato služba je využívána hlavně při vývoji systému a implementaci modelu. Touto cestou je možné získat informace o chování agentů v průběhu simulace. Principiálně se tato služba velmi podobá službě pro ukládání výsledků simulace. Uvnitř agenta lze implementovat funkci, která na základě informací získaných z adresáře služeb zašle příslušnému agentovi informace o právě prováděné činnosti. I když v mnoha implementacích agentních systémů jsou k dispozici nástroje pro ladění, je tato služba užitečná při ladění algoritmů simulace. Vývojář získá jednoduché a přesné informace, co se právě v danou chvíli odehrává. Například o tom, že simulační agent právě obdržel zprávu od svého koordinačního agenta s externím vstupem. Vývojář tak nemusí pracovat s univerzálním, ale mnohdy velmi složitým nástrojem pro sledování agentů v systému.

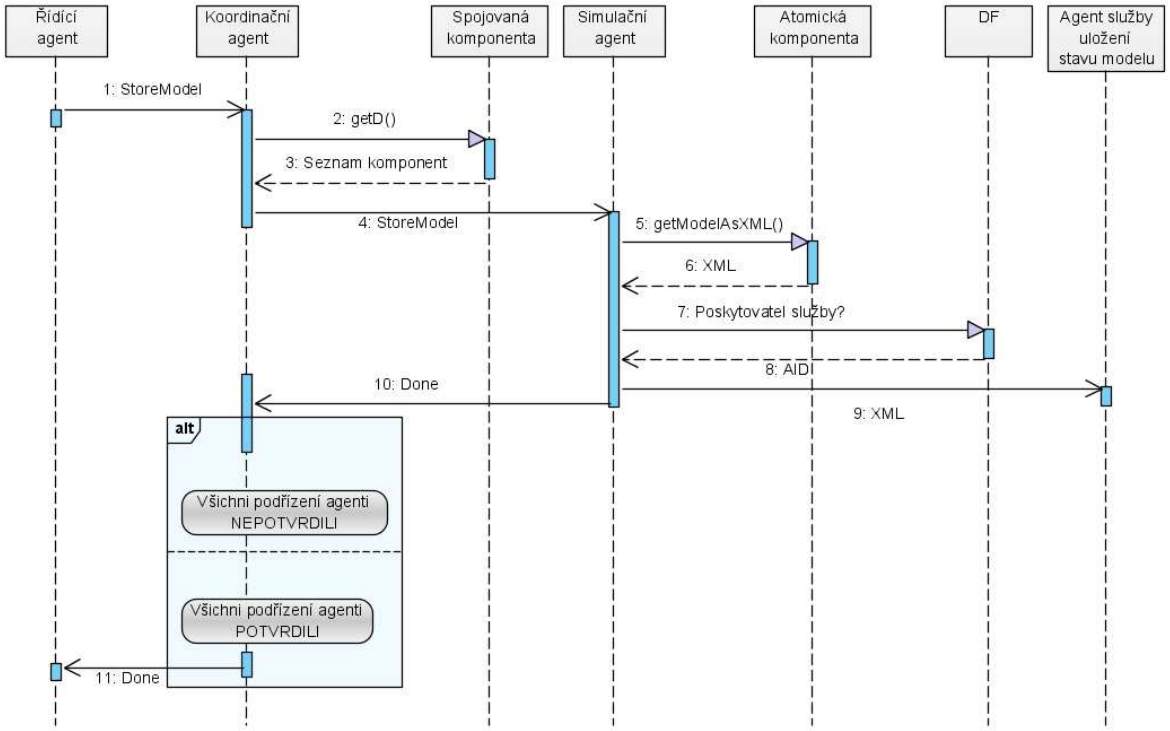
4.3.3 Uložení a obnovení stavu modelu

Díky této službě je možné perzistentně uložit stav celého modelu. Pokud simulační prostředí s touto službou navíc umožňuje obnovení stavu modelu z perzistentní zálohy, nabízí se dvě velmi užitečné využití těchto vlastností:

- pro časově velmi náročnou simulaci lze simulaci přerušovat a následně simulaci obnovit v bodě jejího přerušení,
- v průběhu simulace průběžně ukládat stav modelu, jako případnou zálohu při výpadku některého z uzlu distribuovaného systému.

Implementace je principielně opět velmi obdobná implementaci služby pro ukládání výsledků simulace. Pokyn pro uložení stavu modelu vydává řídící agent, každý abstraktní simulátor (koordinační a simulační agent) bude mít implementovanou podporu pro tento typ zprávy.

Simulační agent po obdržení takové zprávy zavolá metodu přiřazené atomické komponenty *getModelAsXML*, která vrátí řetězec obsahující XML dokument, který reprezentuje aktuální stav komponenty a příslušný modelový čas. Po získání dokumentu simulační agent v registru služeb DF vyhledá agenta implementujícího službu pro uložení stavu modelu a vygenerovaný XML dokument mu zašle. Po odeslání potvrdí nadřazenému koordinačnímu agentovi provedení operace. Chování koordinačního agenta spočívá v rozeslání této zprávy podřízeným agentům a čekání na jejich odpověď. Po obdržení odpovědí od všech podřízených agentů zašle koordinační agent potvrzení svému nadřízenému agentovi. Celá operace je zobrazena pomocí sekvenčního UML diagramu na obrázku 4.8.



Obrázek 4.8: *Sekvenční diagram uložení stavu modelu*

Služba pro obnovení stavu modelu bude také implementována pomocí agenta, který tuto službu zaregistruje prostřednictvím adresáře služeb (DF). Pokyn k obnovení stavu modelu dává řídící agent, každý abstraktní simulátor (koordinační a simulační agent) bude mít implementovanou podporu pro tento typ zprávy. Koordinační agent opět pouze rozešle tento pokyn svým podřízeným agentům a čeká na jejich potvrzení operace. Simulační agent po obdržení tohoto pokynu si pomocí DF nalezne agenta této služby, a zašle mu zprávu obsahující název přiřazené atomické komponenty a požadovaný simulační čas. Jako odpověď obdrží XML dokument s uloženým stavem komponenty. Tento dokument použije jako vstupní parametr funkce atomické komponenty *restoreModelFromXML*, která pomocí zadaného XML dokumentu obnoví stav komponenty, ve kterém se nacházela při jeho uložení. Sekvenční diagram této operace je velmi podobný diagramu z obrázku 4.8, proto zde nebude uveden.

Pro využití těchto dvou služeb uvedené v bodě 2, jako ochranu proti výpadkům uzlů distribuovaného systému, je nutné upravit algoritmus řídícího a koordinačního agenta. Algoritmus řídícího

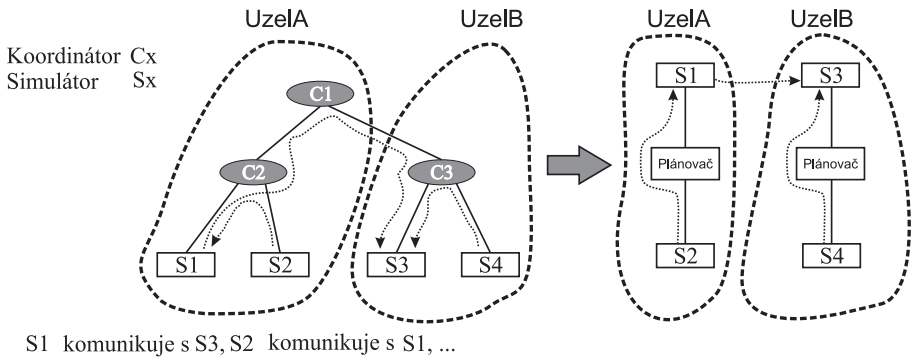
agenta je upraven tak, aby ve stanovený modelový čas zaslal koordinačnímu agentovi pokyn k uložení stavu modelu. Algoritmus koordinačního agenta lze upravit dvěma způsoby. Prvním způsobem je zavedení maximálního času pro potvrzení operace podřízeným agentem. Tato úprava však není příliš vhodná. Nelze jednoduše předem odhadnout, jak bude operace podřízeného agenta trvat. Při nastavení nedostatečného maximálního času by byla situace špatně vyhodnocena. Lepší úpravou algoritmu je zavedení podpory pro speciální typ zprávy. Pomocí této zprávy se nadřazený koordinátor informuje, zda je podřízený agent stále přístupný. Zprávu bude nadřazený koordinátor zasílat v předem stanovených časových intervalech při čekání na potvzení operace. Pokud koordinátor neobdrží odpoveď na tuto informační zprávu, vyhodnotí situaci jako výpadek uzlu a informuje nadřazeného agenta[1]. Po obdržení této zprávy řídící agent provede opětovnou inicializaci simulace (z důvody možné změny počtu uzlů systému) a obnovení stavu modelu z poslední známé zálohy.

4.4 Optimalizace

Tato sekce diskutuje některé možnosti optimalizace simulace v simulačním prostředí založené na navržené architektuře. Jejím úkolem je pouze tyto možnosti představit, mohou být předmětem dalšího výzkumu a vývoje.

4.4.1 Optimalizace modelu

Navržená architektura je založena na abstraktním simulátoru pro paralelní DEVS formalismus (kapitola 3.2.4). Tento abstraktní simulátor interpretuje dynamiku modelů specifikovanou P–DEVS formalismem. Simulátor má stejnou hierarchickou strukturu jako model, vyskytují se v něm dva typy abstraktního simulátoru (koordinátor spojované komponenty a simulátor atomické komponenty). Jak již bylo uvedeno v kapitole o DEVS formalismu (kapitola 3.2.2) spojovaná komponenta definuje vnitřní strukturu modelu. Spojovaná komponenta a přiřazený abstraktní simulátor–koordinátor má za úkol dvě hlavní funkce. První funkcí je správné plánování událostí pro jeho vnitřní komponenty. Druhá funkce je předávat externí vstupy vnitřním komponentám. Pokud odstraníme spojované komponenty z modelu a k abstraktnímu simulátoru atomické komponenty připojíme informace o propojení s ostatními komponentami, dynamika modelu se nezmění. Tím odstraníme hierarchickou strukturu. Situaci ukazuje obrázek 4.9.



Obrázek 4.9: *Transformace modelu na nehierarchickou strukturu*

[1]Tento přístup nemůže být využit v každém agentním systému. V případě, že je pro každého agenta přiřazeno pouze jedno vlákno, nelze tento způsob využít (například v JADE [36]).

Byly publikovány postupy a algoritmy, které tuto hierarchickou strukturu odstraňují a transformují model na nové schéma. Toto schéma redukuje počet abstraktních simulátorů potřebných k provedení simulace, a tím redukuje i režii systému. Další informace, algoritmy pro transformaci a simulaci lze nalézt v [15].

4.4.2 Maximalizace využití výpočetního uzlu

Na každý uzel distribuovaného systému lze umístit speciálního agenta, který bude mít za úkol průběžně monitorovat a analyzovat vytíženost uzlu. Zjištěné poznatky bude agent sdílet s ostatními agenty tohoto typu v systému. Na základě získaných informací bude provádět optimalizaci využití uzlu migrací simulačního agenta na jiný uzel distribuovaného systému, nebo naopak oznámením možnosti přijetím nového simulačního agenta na uzel.

Prvním problémem je způsob měření a analýzy vytíženosti výpočetního uzlu, dalším je implementace kooperativního rozhodování agentů o migracích simulačních agentů na základě znalosti o situaci v celém systému. Tyto dva problémy mohou být předmětem dalšího výzkumu.

4.5 Bezpečnost

Distribuované systémy, které pracují v otevřeném prostředí (jako je například internet), vyžadují zabezpečení jak na úrovni infrastruktury systému, tak i na úrovni aplikační. Distribuované systémy založené na autonomii agenta a jeho mobilitě vyžadují ještě vyšší pozornost pro bezpečnost systému. Úroveň zabezpečení takového systému je navrhována i s ohledem na jaho plánované využití. Například agentní systém využitý jako aukční systém, ve kterém agenti zastupují účastníky aukce a nabízejí cenu za prodávané zboží, vyžaduje jinou úroveň zabezpečení, než navrhovaný distribuovaný simulační systém. Dokonce i úroveň zabezpečení navrženého systému se může značně lišit. Například vysokou úroveň zabezpečení budou vyžadovat organizace pracující s utajovanými či komerčními informacemi. Naopak nižší úroveň zabezpečení vyžaduje akademické či personální využití. Aplikace prvků pro zvýšení bezpečnosti v systému ovlivní výkonnost celého systému, proto je nutné zvolit kompromisní řešení, které vyhovuje aktuálním požadavkům na simulaci.

Existuje několik typů útoků na agentní systém, některé útoky jsou uvedeny v následujících bodech jako krizové scénáře pro navrženou architekturu:

1. Představme si situaci, kdy pro simulaci poskytne útočník svůj výpočetní systém jako další uzel distribuovaného systému. Díky tomu se útočníkovi podaří do systému umístit agenta, který bude postupně rušit původní agenty v systému a nahrazovat je agenty vlastními se stejným identifikátorem. Tak lze poměrně jednoduše znehodnotit výsledky celé simulace. Tento útok je nebezpečný, nemusí být zjištěn a výsledky simulace mohou být považovány za korektní.

2. Tento scénář je obdobný scénáři uvedenému v prvním bodě. Útočník využije agenta, který bude rušit agenty simulačního systému. Tento útok je méně nebezpečný, než útok uvedený v prvním bodě. Cílem útočníka je pouze zrušit probíhající simulaci.

3. Útočník ze systému vymaže agenta pro ukládání dat a nahradí ho agentem vlastním, který si pomocí DF zaregistruje stejnojmennou službu. Tím útočník může potencionálně získat výsledky celé simulace.

4. Další možností útočníka je odposlouchávat nebo modifikovat zprávy mezi uzly distribuovaného systému. V tomto scénáři se útočník nestává aktivní součástí systému. V případě pouhého odposlouchávání zpráv je i těžce odhalitelným.

Následující odstavce popisují některé z technik zabezpečení distribuovaného systému.

Autentizace

Autentizace poskytuje garanci, že uživatel který se účastní distribuované simulace, je považován za legitimního účastníka. Legitimním účastníkem se rozumí účastník, o kterém má systém informace a kterého nepovažuje za útočníka. Takový účastník má v systému přiřazenou jednoznačnou identitu a přístupové heslo. Tento mechanismus zabezpečení poskytuje částečně zabezpečení proti útokům uvedeným v bodech 1 až 3. Toto zabezpečení je částečné ze dvou důvodů. Prvním důvodem je možnost zneužití cizí identity. Druhým důvodem je špatné prověření účastníka a následné přidělení nové identifikace a zařazení do seznamu legitimních účastníků. Z hlediska výkonnosti systému tento mechanismus zabezpečení minimálně zatěžuje systém.

Oprávnění

Díky authentizaci uživatelů se distribuovaný agentní systém stává víceuživatelským systémem, ve kterém jsou jednotlivé prvky systému (agenti, uzly distribuovaného systému) vlastněny uživateli. Všechny operace agentů v systému mohou být selektivně povoleny či zakázány na základě vlastníka agenta a dle množiny pravidel. Díky tomuto přístupu může být selektivně povolen či zakázán přístup jednotlivým uživatelům ke službám systému či k ostatním agentům. Tato technologie zlepšuje zabezpečení proti útokům uvedeným v bodech 1 až 3. Z hlediska výkonnosti systému mechanismus zabezpečení způsobuje citelné zatížení systému. Díky authentizaci uživatelů se distribuovaný agentní systém stává víceuživatelským systémem, ve kterém jsou jednotlivé prvky systému (agenti, uzly distribuovaného systému) vlastněny uživateli. Všechny operace agentů v systému mohou být selektivně povoleny či zakázány na základě vlastníka agenta a dle množiny pravidel. Díky tomuto přístupu může být selektivně povolen či zakázán přístup jednotlivým uživatelům ke službám systému či k ostatním agentům. Tato technologie zlepšuje zabezpečení proti útokům uvedeným v bodech 1 až 3. Z hlediska výkonnosti systému mechanismus zabezpečení způsobuje citelné zatížení systému. Díky authentizaci uživatelů se distribuovaný agentní systém stává víceuživatelským systémem, ve kterém jsou jednotlivé prvky systému (agenti, uzly distribuovaného systému) vlastněny uživateli. Všechny operace agentů v systému mohou být selektivně povoleny či zakázány na základě vlastníka agenta a dle množiny pravidel. Díky tomuto přístupu může být selektivně povolen či zakázán přístup jednotlivým uživatelům ke službám systému či k ostatním agentům. Tato technologie zlepšuje zabezpečení proti útokům uvedeným v bodech 1 až 3. Z hlediska výkonnosti systému mechanismus zabezpečení způsobuje citelné zatížení systému.

Integrita a důvěryhodnost zprávy

Podpis, kontrolní součet a šifrování zprávy garantuje spolehlivou úroveň zabezpečení komunikace agentů mezi výpočetními uzly systému. Podpis společně s kontrolním součtem u přenášené zprávy zaručuje její integritu (poskytuje ujištění, že s daty nebylo během přenosu manipulováno) a zaručuje identitu tvůrce zprávy. Na druhé straně, šifrování znemožňuje získávání dat ze zprávy během přenosu. Tyto technologie zabezpečují systém proti útokům uvedeným v bodě 4. Z hlediska výkonnosti systému tento mechanismus zabezpečení způsobuje vysoké zatížení. Zatížení systému způsobuje vytváření nebo čtení zabezpečené zprávy a velikost přenášené zprávy je zvýšena o velikost zabezpečující informace.

4.6 Porovnání navržené architektury se systémy z kapitoly 2.2

V této sekci bude uvedeno krátké porovnání navržené kapitoly s některými systému a architekturami uvedenými v kapitole 2.2.

Navržená architektura je nejvíce podobná nástroji James (kapitola 2.2.5). Tento nástroj byl navržen především pro simulace agentních systému. Nástroj James je postaven na přesně opačné architektuře. Distribuovanost zajišťuje paralelní DEVS formalismus, pomocí kterého se modelují i agenti, kteří jsou v tomto systému předmětem simulace. Hlaví rozdíly mezi architekturami jsou ve způsobu implementace distribuovanosti a také pro jaké účely byly architektury vytvořeny. Jak již bylo uvedeno v úvodu práce, k tomuto projektu nejsou dostupné všechny informace.

Dalším nástrojem, se kterým lze navrženou architekturu porovnávat, je DEVS/HLA (kapitola 2.2.2). Stejně jako navržená architektura používá tento simulační nástroj DEVS formalismus pro modelování a simulace. Pro implementaci distribuovaného zpracování využívá architekturu HLA. Hlavní výhodou DEVS/HLA je jeho kompatibilita s ostatními simulátory vyhovující normě HLA. To umožňuje tento nástroj zapojit do rozsáhlých simulačních systémů a právě pro tento účel byla HLA vytvořena. Hlavní využití pro HLA je stále především na půdě armády. Navržená architektura reprezentuje zcela odlišný přístup k realizaci distribuovaného zpracování, díky kterému lze poměrně jednoduše do systému zabudovat umělou inteligenci pro optimalizaci simulace.

Nástroje Swarm (kapitola 2.2.3), Mason (kapitola 2.2.4) a Farm (kapitola 2.2.6) jsou určeny především k simulacím, ve kterých jsou agenti využity jako modelovací prostředek. V navržené architektuře je agentní systém využit jako prostředek pro implementaci distribuovaného zpracování, což je zásadní odlišnost.

Kapitola 5

Prototyp simulačního prostředí

Kapitola velmi krátce seznamuje s prototypem simulačního prostředí, který je založen na navržené architektuře. V úvodu jsou vysvětleny důvody volby implementačního jazyka Java. Celá kapitola je rozdělena na tři základní sekce, v první sekci je vysvětlen princip implementace modelů a na jednoduchých příkladech je ukázána implementace atomické a spojované komponenty. Druhá sekce se zabývá implementací simulace. Simulace v navržené architektuře je založena na agentním systému, při implementaci prototypu byla využita kostra agentního systému z projektu JADE [36]. Jsou zde vysvětleny základní principy funkce tohoto agentního systému a je zde uveden způsob implementace simulace jako chování agentů. Třetí sekce popisuje použití prototypu a poslední sekce diskutuje stav implementace.

Jako implementační jazyk jsem zvolil jazyk *Java*. Jedná se o objektově orientovaný jazyk, který je v poslední době hojně využívaný především v souvislosti s internetem. Pro tento jazyk je volně k dispozici překladač[1] a existuje mnoho kvalitních vývojových prostředí zdarma[2], které usnadňují implementaci. Programy implementované v Javě jsou přenositelné mezi různými operačními systémy, jako jsou například MS Windows, Linux, Unix, MacOS a další. Hlavní a pravděpodobně jedinou nevýhodou tohoto jazyka je rychlost provádění programů, jedná se totiž (narozdíl například od C++) o jazyk interpretovaný. Jako platformu pro interpret jazyka Java lze, s určitým omezením z důvodu bezpečnosti, využít i prohlížeč www stánek. V dnešní době mají tuto podporu téměř všechny běžně využívané prohlížeče, či ji lze jednoduše do prohlížeče nainstalovat. Java je vhodný jazyk pro implementace prototypů systému.

5.1 Modelovaní

Implementace podpory modelování odpovídá návrhu uvedenému v kapitole o architektuře jako UML diagram tříd na obrázku 4.4. Při implementaci modelu bude uživatel využívat především abstraktní třídu *AbstractAtomicModel* pro implementaci atomické komponenty a třídu *CoupledComponent* pro implementaci spojované komponenty. Implementace atomické komponenty spočívá ve vytvoření nové třídy, která je potomkem abstraktní třídy *AbstractAtomicModel*. V této třídě jsou definovány abstraktní metody, které reprezentují přechodové a transformační funkce atomické komponenty. Korespondenci metod třídy a funkcí z definice atomické komponenty byla uvedena v tabulce 4.1. Uživatel pomocí těchto metod implementuje požadované chování komponenty. Při implementaci může využívat metody pro zjištění či nastavení stavu komponenty a pracovat se vstupními či výstupními porty.

[1] Překladač lze získat z `http://java.sun.com`

[2] Mezi nejznámější patří *NetBeans*, `http://www.netbeans.org` a *Eclipse*, `http://www.eclipse.org`

Implementace spojované komponenty je obdobná a také spočívá ve vytvoření vlastní třídy. Tato třída bude potomkem třídy *CoupledComponent*. Uživatel ve své třídě použije metody *addModel* a *addInfluence* pro definování seznamu vnitřních komponent a jejich propojení (nejčastěji v konstruktoru třídy). Zdůvodnění tohoto způsobu implementace bylo uvedeno v předchozí kapitole. Hlavní nevýhodou tohoto přístupu je snadná chyba uživatele při implementaci modelu, která se bude velmi špatně odhalovat. Při odhalování této chyby bude výhodné využít nástroj pro sledování průběhu simulace diskutovaného v kapitole 4.3.2.

Atomická komponenta

V kapitole 3.2.1 byl uveden příklad specifikace atomické komponenty, která modelovala semafor. Tuto komponentu využiji pro ukázku implementace modelu v prototypu simulačního prostředí. Ze všeho nejdříve je nutné vytvořit třídu komponenty, která ji bude implementovat a definovat všechny možné stavy komponenty. Komponenta má definováno pouze pět stavů. Z tohoto důvodu budou tyto stavy implementovány jako statické privátní proměnné třídy.

```
public class SemaphoreAtomicModel extends AbstractAtomicComponent {
    private static final ModelState STATE_RG = new ModelState("RG");
    private static final ModelState STATE_RY = new ModelState("RY");
    ...
}
```

Implementace interní přechodové funkce je velice jednoduchá a přesně koresponduje se specifikací komponenty.

```
public void internalFunction() {
    ModelState state = getModelState();
    if (STATE_RG.equals(state)) {
        setModelState(STATE_RY);
    } else if (STATE_RY.equals(state)) {
        setModelState(STATE_GR);
    } ...
}
```

Obdobným způsobem jsou definovány externí přechodové a výstupní funkce, i funkce času přechodu.

Spojovaná komponenta

Její implementace je jednodušší, než implementace atomické komponenty. Jako ukázku implementuji spojovanou komponentu, která bude obsahovat pouze dvě komponenty. První komponentou je atomická komponenta modelující generátor impulsů s jedním výstupem pojmenovaná *generator*, kterou implementuje třída *org.sjdevs.testing.AtomicGenerator*[3]. Druhou komponentou je opět atomická komponenta modelující přijímač s jedním vstupem pojmenovaná *acceptor*, kterou implementuje třída *org.sjdevs.testing.AtomicAcceptor*. Výstup první komponenty bude propojen na vstup druhé komponenty. Celá implementace spojované komponenty:

```
public class CoupledTest extends CoupledComponent {
   public CoupledTest(String name) {
      super(name);
      addComponent("generator", "org.sjdevs.testing.AtomicGenerator");
      addComponent("acceptor", "org.sjdevs.testing.AtomicAcceptor");
      addInfluence("generator", "out", "acceptor", "in");
   }
}
```

[3]Pro přehlednost jsou třídy v prog. jazyce Java seskupovány do tzv. balíčků. Balíčky tříd vytvářejí stromovou strukturu, jednotlivé balíčky se v zápise oddělují tečkou. Balíček *org.sjdevs* seskupuje celý prototyp.

5.2 Simulace

Dle návrhu simulaci zajišťují agenti agentního systému, který odpovídá normě FIPA. Implementace agentního systému je časově velmi náročná. Z tohoto důvodu jsem v prototypu simulačního systému využil projekt JADE.

JADE (*Java Agent DEvelopment Framework*) [36] je softwarový projekt, implementovaný v jazyce Java, jehož cílem je zjednodušit vývoj multiagentních systémů. Projekt nabízí vývojáři základní prostředí a dále několik nástrojů pro ladění a kontrolu běhu programu. JADE je především knihovna tříd, které zcela odpovídá specifikacím FIPA, proto usnadňuje vzájemnou spolupráci různých multiagentních systémů. Skládá se ze tří základních částí:

- Běhové prostředí (*runtime environment*) poskytuje základní infrastrukturu pro prostředí, kde agenti mohou „žít“. Spuštěné běhové prostředí je nutnou podmínkou ke spouštění a hostování agentů
- Knihovna tříd, které může programátor použít k vývoji vlastních agentů
- Sadu grafických nástrojů ke vzdálené administraci a monitorování aktivity běžících agentů

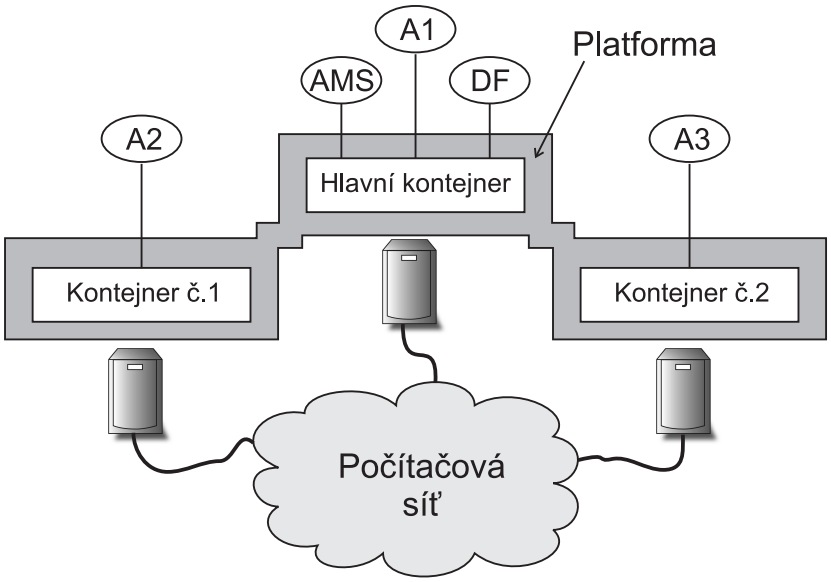
Celý projekt je distribuován jako open source pod LGPL licencí, vše je volně ke stažení[4]. JADE poskytuje infrastrukturu pro agenty i šablonu pro vytváření agentů, kteří využívají obecný vysoko–úrovňový jazyk a protokol. Nabízí uživateli množství předdefinovaných tříd, které usnadňují konstrukci a použití agentů. Prvky systému definované podle specifikace FIPA jsou tak z velké části programátorům skryty.

Základní prvek systému v JADE tvoří platforma, tvořená jedním nebo několika kontejnery. Každý kontejner představuje jednu Java aplikaci a mohou být distribuované mezi několik fyzických stanic. Příklad platformy tvořené třemi kontejnery na různých fyzických stanicích ukazuje obrázek 5.1. Prvky *Directory Facilitator* (na obrázku DF), *Agent Management System* (na obrázku AMS) i *Agent Communication Cannel* jsou součástí hlavního kontejneru platformy. Agenti jsou umístěny na kontejnery platformy ($A1$ je umístěn na hlavním kontejneru, $A2$ na kontejneru č. 1 atd.).

Agent je implementován jako samostatný proces a je potomkem třídy *jade.core.Agent*. Bylo nutné přetížit pouze metodu *setup()*, která je automaticky volána při vytváření agenta uvnitř kontejneru. Metoda se především využívá ke specifikaci chování agenta. Chování agenta je specifikováno instancemi tříd, které jsou potomky třídy *jade.core.behaviours.Behaviour*. Třída obsahuje metodu *action*, ve které je chování implementováno. Pro každý typ agenta v systému bylo nutné tyto třídy vytvořit. Například třída implementující chování koordinačního agenta obsahuje implementaci algoritmu koordinátora pro spojovanou P–DEVS komponentu.

Zajímavá je například implementace čekání spojované komponenty na potvrzení všech jejích podřízených agentů o ukončení operace (zpráva *Done*), například při zpracování zprávy *I*. Agent je v JADE implementován jako jediný proces. Tento proces je využit i pro zpracování chování agenta, které je provedeno jako celek. Tzn., průběh zpracování chování agenta nelze v určitém bodě přerušit a později z tohoto bodu pokračovat dále. Řešením by bylo cyklicky kontrolovat nové zprávy *Done* od podřízených agentů, toto řešení je ale velmi nevhodné. Agent by nebyl schopen zpracovávat ostatní zprávy, v tomto případě případné externí události vnitřních komponent. Vhodnějším řešením problému je rozdělení implementace chování agenta pro zprávu *I* na více dílčích chování, z nichž některé budou vytvořeny dynamicky v průběhu zpracování. Dynamicky bude vytvořeno chování, které přijímá zprávy typu *Done* od vnitřních komponent. Po obdržení všech zpráv automaticky vytvoří nové chování, které bude implementovat další kroky při zpracování zprávy *I*, které zaregistruje a samo sebe odstraní ze seznamu chování agenta. Tento způsob implementace neblokuje ostatní funkce agenta (reakce na zprávy *Y*).

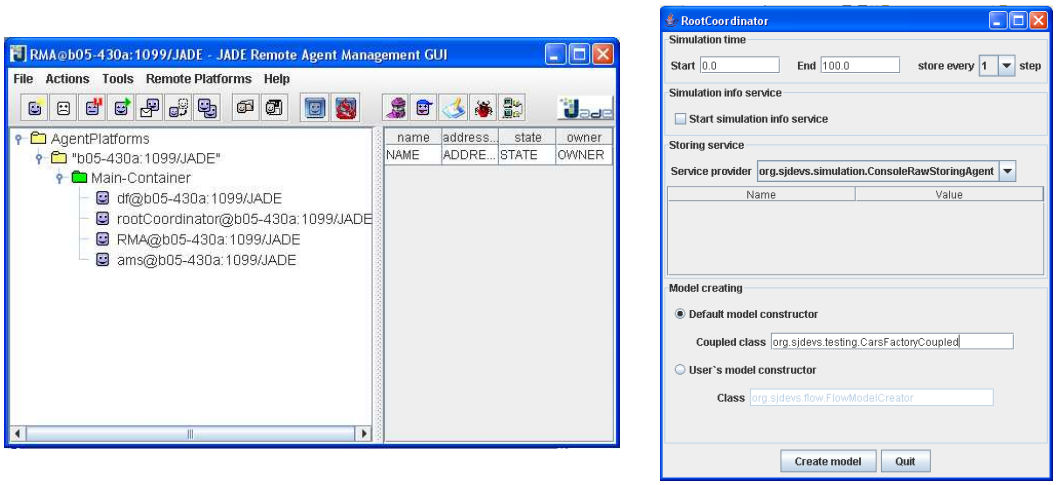
[4] Sadu knihoven a nástrojů JADE lze získat na `http://jade.tilab.com` po bezplatné registraci



Obrázek 5.1: *Platforma agentního systému v JADE*

5.3 Používání systému

Celý simulační systém je rozdělen na dvě části. První část reprezentuje jeden výpočetní uzel systému. Jedná se o implementaci kontejneru agentního systému. Druhá část je implementací hlavního kontejneru agentního systému, ke kterému se ostatní kontejnery agentního systému připojují. Uživatel spustí hlavní kontejner, po spuštění hlavního kontejneru se zobrazí grafické uživatelské rozhranní agentního systému a hlavního koordinátora (obrázek 5.2). Pomocí grafického uživatelského rozhranní hlavního koordinátora nastaví uživatel parametry simulace, jako je například simulační čas, způsob ukládání výsledků (konkrétní typ služby, diskutováno v kapitole 4.3.1) a způsob vytvoření modelu. K dispozici má uživatel dva způsoby, kterými může být model vytvořen.



Obrázek 5.2: *Grafické uživatelské rozhranní agentního systému JADE a kořenového koordinátora*

Prvním je tzv. *implicitní způsob*. U tohoto typu vytváření modelu zadá uživatel pouze název třídy,

která implementuje spojovanou komponentu modelu. Systém poté automaticky vytvoří instanci třídy a na základě seznamu komponent vytvoří i instance vnitřních komponenty a jejich simulátory či koordinátory.

Pokud uživatel potřebuje mít absolutní kontrolu nad průběhem vytváření modelu a jeho přípravy v agentním systému, může využít druhý způsob tzv. *uživatelský způsob*. U tohoto typu vytváření modelu zadá uživatel název třídy, která implementuje rozhranní *ModelCreator*, které obsahuje metodu *createModel*. V této metodě uživatel implementuje vlastní postup vytváření modelu. Tento přístup pro vytváření modelu byl využit pro simulace proudění kapaliny z důvodu dělení modelu a přípravy sítě modelu, pro všechny ostatní modely bylo využito implicitní vytváření modelu.

Po nastavení požadovaných parametrů simulace uživatel klikne na tlačítko pro vytvoření modelu. V agentním systému dojde k vytvoření simulačních a koordinačních agentů s přiřazenými komponentami modelu. Uživatel si může připojit další agentní kontejnery, které budou umístěny na jiných strojích. Pomocí grafického uživatelského rozhranní agentního systému může provést migraci agentů na tyto uzly a poté spustit simulaci.

Jak již bylo uvedeno v kapitole 4.4.2, v budoucnu by měla být provedena migrace automaticky s využitím umělé inteligence a racionálních agentů (kapitola 3.3), ale je třeba dalšího výzkumu v této oblasti. Prototypová verze tuto vlastnost nemá implementovánu.

5.4 Současný stav implementace

V současné době se stále jedná o prototyp simulačního prostředí. Prototyp nemá implementovány všechny funkce a služby popisované v kapitole o návrhu architektury. Prototyp má implementovány ty nejdůležitější funkce, k dispozici jsou třídy pro vytváření modelů a vše potřebné k provedení simulace. Tzn., jsou implementování všechny typy agentů, které jsou nezbytně nutné pro simulaci (řídící, koordinační a simulační agent). Dále je implementována služba pro ukládání výsledků simulace (služba byla popsána v kapitole 4.3.1). Je implementováno ukládání výsledků do souboru a výpis výsledků na standardní výstup. Pro simulace modelů z oblasti dynamiky tekutin (budou popsány v následující kapitole) bylo nutné implementovat speciální způsob ukládání výsledků. Další implementovanou službou je služba pro sběr informací o průběhu simulace (služba byla popsána v kapitole 4.3.2). Služba byla využívána pro odhalování chyb v implementaci simulačních algoritmů a při implementaci modelů. V prototypu není implementována žádná z optimalizací popisovaných v kapitole 4.4 a není implementována automatická migrace částí modelu mezi uzly distribuovaného systému. Budoucnost prototypu je diskutována v závěru práce v rámci dalšího možného výzkumu.

Kapitola 6

Ukázka simulace

Kapitola popisuje simulaci netriviálního modelu pomocí prototypu simulačního prostředí založeného na navržené architektuře. Jako netriviální model byl zvolen model z oblasti dynamiky tekutin. Na začátku kapitoly je krátký úvod do této problematiky a metodiky výpočtu tohoto složitého problému. Poté je představen simulovaný model a vysvětlen způsob implementace pomocí DEVS formalismu. V závěru kapitoly jsou prezentovány a diskutovány naměřené hodnoty při simulacích.

Pro ukázku simulace netriviálního modelu byl použit model z oblasti dynamiky tekutin. Počítačová dynamika tekutin je volný překlad anglického *Computational Fluid Dynamics (CFD)*, což je vědecká oblast aplikované mechaniky/dynamiky tekutin multidisciplinárního charakteru. Význam této oblasti výzkumu je obrovský. S problematikou proudění tekutin se dnes setkáváme na každém kroku. Využívá se k zdokonalování konstrukce strojů a zařízení, přes které proudí tekutiny. Optimalizovaná konstrukce stroje například snižuje množství energie, které spotřebuje stroj při svém chodu. Využívá se při konstrukci staveb či mostů, u kterých je nutné zohledňovat mnohé aerodynamické požadavky. Jako další oblast využití simulace proudění tekutin lze například uvést modelování a simulace ekosystémů. Existuje velmi dlouhá řada dalších oblastí, ve které se využívá simulace proudění tekutin například: letectví, vojenská technika, chemicko–technologické procesy apod.

Fyzikálním modelem a matematickým základem celé této oblasti je tzv. *Navier–Stokesova* rovnice. Tato rovnice je známa již 150 let. Rovnici odvodil Navier[1] v roce 1827 a nezávisle na něm a jiným způsobem ji odvodil Stokes[2] v roce 1845. *Navier–Stokesova* rovnice vyjadřuje rovnováhu sil na elementu tekutiny. Řešení této rovnice není do hloubky zcela zvládnuté dodnes[3]. Odborníci zabývající se touto oblastí hovoří o *Navier–Stokesovy* rovnici jako o rovnici velmi jednoduchou pro tekutinu, složitou pro lidi a nepřekonatelnou pro počítače. Hlavní motivací specialistů na CFD je zvládnout proudění tekutin stále složitějšími modely a vytvořit co nejvěrnější simulaci světa a procesů proudění tekutiny.

Ukázkový model je jednoduchý model obtékání tekutin kolem tuhých a statických těles. Simulace je pomocí formalismu DEVS paralelizována a simulována na několika výpočetních systémech. Následující sekce poskytuje krátký úvod k *Navier–Stokesovy* rovnici, ve kterém je rovnice představena a velmi stručně vysvětlena metodika jejícho výpočtu. Podrobné vysvětlení lze nalézt v [20] či [13]. Po tomto úvodu následuje popis simulovaného modelu a princip implementace pomocí DEVS formalismus. V závěru kapitoly jsou prezentovány a diskutovány naměřené hodnoty při simulacích v porovnání s klasickou simulací na jednom výpočetním systému.

[1] Claude Louise M. H. Navier (1787–1836) byl francouzský matematik

[2] George Gabriel Stokes (1819–1903) byl anglický matematik a fyzik

[3] Tento fakt je publikován v [20], kde lze nalézt velmi podrobně zpracované téma CFD.

6.1 Modelování proudění tekutin

Jak již bylo uvedeno v úvodu kapitoly, fyzikálním a matematickým modelem je tzv. *Navier–Stokesova* rovnice. Rovnice vyjadřuje rovnováhu sil na elementu tekutiny proudící laminárně, tedy s uvažováním třecích sil. V místě elementu působí vnější objemové zrychlení, dále na stěny elementu působí síly tlakové třecí, které jsou způsobeny viskozitou kapaliny. Na tento element je aplikován základní vztah silové rovnováhy (součet všech působících sil je roven součinu hmoty a jeho zrychlení) a z této rovnice se při odvozování *Navier–Stokesovy* rovnice vychází, celý postup odvozování lze nalézt v [13]. Pro skutečnou kapalinu, která je navíc stlačitelná, má rovnice obecný vektorový tvar (6.1):

$$\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot grad\, \vec{v} = \vec{A} - \frac{1}{\rho} \cdot grad\, p + \nu \cdot \Delta \vec{v} + \frac{\nu}{3}\, grad\, div\, \vec{v} \tag{6.1}$$

Význam jednotlivých členů rovnice:

$\frac{\partial \vec{v}}{\partial t}$ lokální zrychlení,

$\vec{v} \cdot grad\, \vec{v}$ konvektivní zrychlení,

$\vec{A}$ objemové zrychlení,

$\frac{1}{\rho} \cdot grad\, p$ zrychlení způsobené tlakovým spádem (gradientem),

$\nu \cdot \Delta \vec{v}$ zrychlení potřebné k překonání viskozního tření tekutiny,

$\frac{\nu}{3} grad\, div\, \vec{v}$ je zrychlení, způsobené dalším účinkem vazkosti vzhledem ke stlačitelnosti tekutiny.

Rovnici lze doplnit o rovnici kontinuity. Tvar rovnice kontinuity je závislý na druhu tekutiny, například pro nestlačitelnou tekutinu má tvar $div\, \vec{v} = 0$. Simulovaný model je model proudění nestlačitelné tekutiny ve 2D. Rovnice jsou pro implementaci výhodnější v diferenciální formě rozepsané pro 2D prostor. Pro 2D jsou uvedeny dvě rovnice. Moment ve směru osy x (6.2) a moment ve směru osy y (6.3) a rovnice kontinuity (6.4).

$$\rho \frac{\partial u}{\partial t} + \rho\, u \frac{\partial u}{\partial x} + \rho\, \nu \frac{\partial u}{\partial y} = -\frac{\partial p}{\partial x} + \mu \left(\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial x^2}\right), \tag{6.2}$$

$$\rho \frac{\partial v}{\partial t} + \rho\, u \frac{\partial v}{\partial x} + \rho\, \nu \frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + \mu \left(\frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial x^2}\right), \tag{6.3}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{6.4}$$

Rovnice jsou velmi složité, proto je provedena úprava na bezrozměrné veličiny[4]. Díky této úpravě je možné vypuštění některých členů rovnice. Jedná se o členy, které mají velmi malý vliv na výsledek. Po těchto úpravách dostáváme mnohem přijatelnější výsledek. Pro urychlení výpočtu se vypočítává moment pouze v jedné ose, druhý moment je vypočítán pomocí rovnice kontinuity. Upravené rovnice mají tvar:

$$\frac{\partial u'}{\partial t'} + u \frac{\partial u'}{\partial x'} + \nu \frac{\partial u'}{\partial y'} = -\frac{\partial p'}{\partial x'} + \frac{1}{Re} \frac{\partial^2 u'}{\partial y'^2}, \tag{6.5}$$

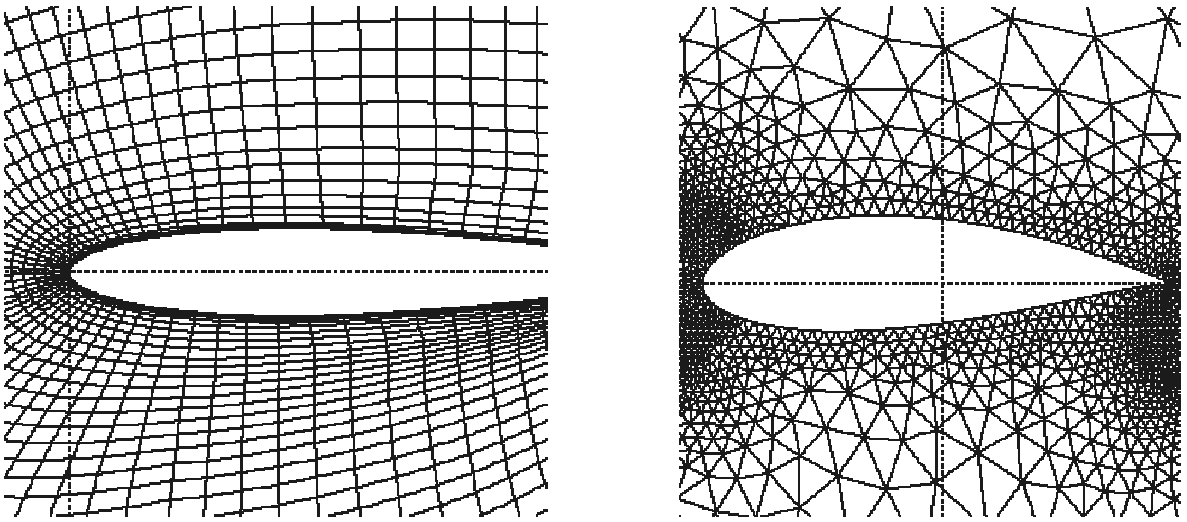
$$\frac{\partial u'}{\partial x'} + \frac{\partial v'}{\partial y'} = 0 \tag{6.6}$$

[4] Postup a vysvětlení úpravy uvádí [17]

Kde Re jde Reynoldsovo číslo, které udává poměr vnitřních sil tekutiny k viskozním silám[5]. Tyto rovnice jsou již mnohem jednodušší a jsou využívány pro výpočet. Postup výpočtu uvádí následující sekce.

6.1.1 Mřížka

Analytické řešení *Navier–Stokesovy* rovnice je možné pouze pro velice jednoduché modely (jednoduché laminární proudění). Složitější modely se řeší pomocí numerických metod, především metodou konečných diferencí. Toto řešení spočívá ve výpočtu vlastností proudící tekutiny pouze v několika bodech řešeného prostoru. Tyto body tvoří tzv. mřížku. Prvním krokem analýzy problému je stanovení těchto bodů. V současné době se využívají dva základní typy mřížky. Prvním typem je tzv. *strukturovaná mřížka* (obrázek 6.1 vlevo). Konstrukce strukturované mřížky je složitá, nicméně výpočet *Navier–Stokesovy* rovnice na této mřížce je jednodušší. V tomto typu mřížky se velmi jednoduše vyhledávají okolní body zvoleného bodu. Druhým typem je *mřížka nestrukturovaná* (obrázek 6.1 vpravo). Mřížka tohoto typu se jednodušeji generuje a poskytuje výhodnější rozložení bodů v prostoru. Výpočet *Navier–Stokesovy* rovnice na této mřížce je však složitější, seznam bodu je nutné upravovat a složitěji se v tomto seznamu vyhledávají okolní body zvoleného bodu.



Obrázek 6.1: *Typy diskretizací prostoru pro výpočet NS rovnic pomocí konečných diferencí*

Generováním a optimalizací mřížky se zabývá mnoho výzkumných skupin, jedná se o velmi složitý a zajímavý problém. Velice často se využívá metod a postupů využívaných v počítačové grafice. Pro experiment byl zvolen model se strukturovanou mřížku, pro její generování byl využit nástroj GenGrid[6].

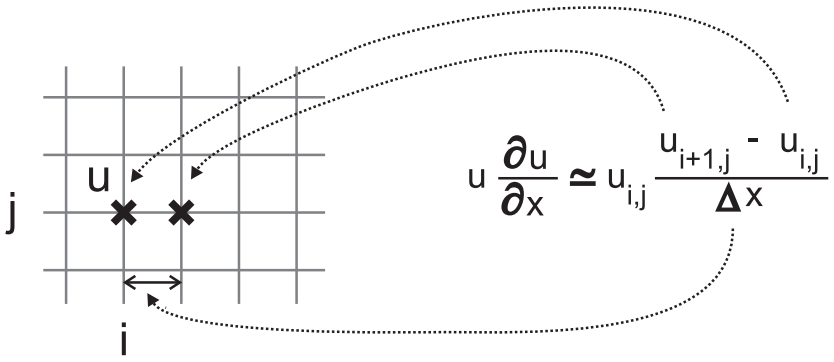
[5] Další informace lze nalézt na `http://en.wikipedia.org/wiki/Reynolds_number`

[6] Nástroj lze získat z `http://caflab.yeungnam.ac.kr/down/download1_e.html`

6.1.2 Metodika výpočtu

Diskretizace na mřížce

Parciální derivace z rovnice 6.5 nahradíme diferencemi vypočtenými na mřížce. Například na obrázku 6.2 je uveden příklad diskretizace jednoho člena rovnice s grafickým znázorněním na mřížce. Tabulka 6.1 uvádí přehled diskretizace derivací pro ostatní členy rovnice.



Obrázek 6.2: *Příklad diskretizace parciální derivace na mřížce*

Postup výpočtu

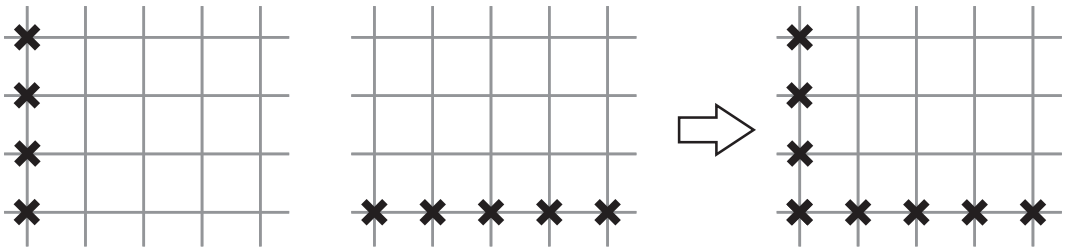
V prvním kroku vygenerujeme mřížku. Počáteční hodnoty pro výpočet známe pouze v několika bodech mřížky (obrázek 6.3, jsou označeny tučným křížkem). Jedná se o počáteční podmínky–vstup proudění (levá strana) a stěna (v modelu uvažuji proudění zleva–doprava, po stranách mřížky jsou pevné stěny). Výpočet probíhá zleva doprava a postupně jsou vypočítávány body celé mřížky. Další

Derivace	Diskretizace na mřížce
$\frac{\partial u}{\partial t}$	$\frac{u^{n-1}-u^n}{\Delta t}$
$\frac{\partial u}{\partial x}$	$\frac{u_{i+1,j}-u_{i,j}}{\Delta x}$
$\frac{\partial p}{\partial x}$	$\frac{p_{i+1,j}-p_{i,j}}{\Delta x}$
$\frac{\partial^2 u}{\partial y^2}$	$\frac{u_{i-1,j}-2\cdot u_{i,j}+u_{i+1,j}}{(\Delta x)^2}$
$\frac{\partial^2 u}{\partial y^2}$	$\frac{u_{i,j-1}-2\cdot u_{i,j}+u_{i,j+1}}{(\Delta y)^2}$

Tabulka 6.1: *Diskretizace členů NS rovnice na na mřížce*

cykly výpočtu jsou prováděny vždy pomocí hodnot z předešlém cyklu, jedná se tedy o metodu „kráčení v čase" (název použit v [20]). Diferenciální rovnici 6.5 nahradíme dle tabulky 6.1. V takto získané rovnici obsahuje pouze jediný člen hodnotu u $(i+1)$–tém sloupci (člen z obrázku 6.2), proto lze z této rovnice velmi jednoduše vypočítat hodnotu $u_{i+1,j}$. Druhou složku vektoru rychlosti $\vec{V}_{i,j} = (u_{i,j}, v_{i,j})$ vypočteme obdobným způsobem pomocí rovnice kontinuity (6.6) a složky vektoru rychlosti na ose x vypočtené v předešlém kroku výpočtu. Tímto postupem vypočteme hodnoty ve všech bodech mřížky.

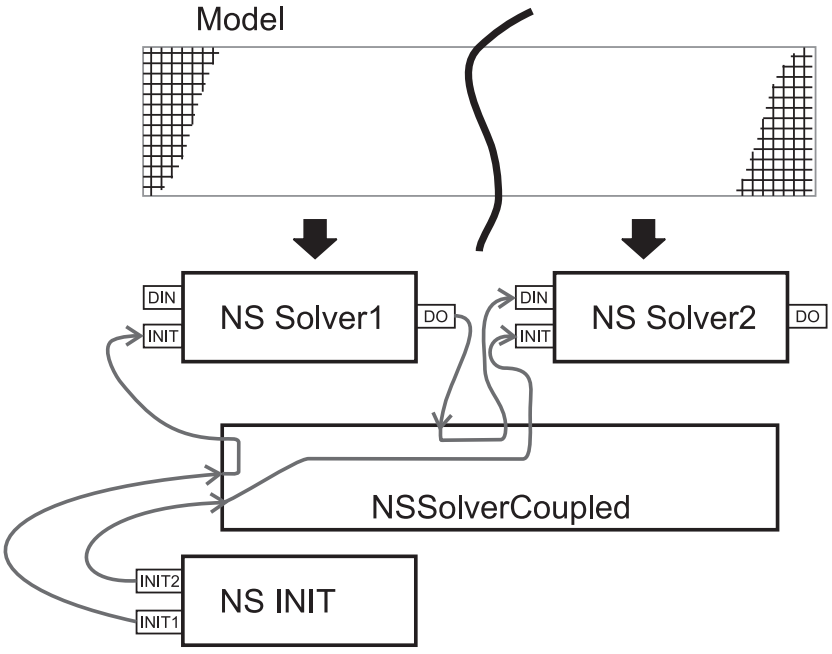
Při nevhodně zvolených parametrech výpočtu (především se jedna o Δt) se metoda může chovat značně nestabilně. Stanovení těchto výpočtu se věnuje, stejně jako generování sítě, velká část výzkumu v této oblasti. Parametry modelu byly nalezeny empiricky.



Obrázek 6.3: *Známé hodnoty vlastností tekutin na mřížce pro nastartování výpočtu*

6.2 Implementace pomocí DEVS formalismu

Simulovaný model je rozdělen na požadovaný počet částí dle osy x. Jednotlivé části se umístí na speciální atomické komponenty. Atomická komponenta obsahuje implementaci algoritmu pro řešení rovnic popsaného v předchozí sekci. Každá taková komponenta má dva vstupy. První vstup slouží k inicializaci, na tento vstup v prvním kroku simulace je vložena přidělená část mřížky a inicializační parametry. Druhý vstup slouží k získávání hodnot vlastností kapaliny v sousedních bodech přidělené mřížky (v případě modelu body nalevo od přidělené části mřížky). Komponenta má jeden výstup, který slouží k předávání vypočtených hodnot na pravém okraji přidělené mřížky další navazující komponentě. Celý model je zastřešen spojovanou komponentou, která definuje propojení jednotlivých částí mřížky. Inicializaci modelu provádí speciální atomická komponenta (v dalším textu ji budu označovat jako inicializační), která rozdělí mřížku modelu na požadovaný počet částí a jednotlivé části umístí na výstupní porty. Počet výstupních portů odpovídá počtu částí, na které je model dělen. Spojovaná komponenta propojuje tyto výstupy s inicializačními vstupy jednotlivých komponent pro výpočet. Po provedení inicializace již dále tato komponenta do simulace nezasahuje. Situaci ukazuje obrázek 6.4. V prvním kroku simulace provede inicializační komponenta rozdělení modelu na požadovaný počet



Obrázek 6.4: *DEVS model pro výpočet NS rovnic*

částí a jednotlivé části rozdělené mřížky umístí na své výstupy. Ty jsou dle propojení definované spojovanou komponentou přesunuty na příslušné inicializační vstupy atomických komponent pro výpočet NS rovnic (NS Solver 1 a 2). Tyto komponenty provedou pomocí externí přechodové funkce inicializaci dat potřebnou k výpočtu. V každém kroku výpočtu (simulační čas plyne po 1 a reprezentuje zde krok výpočtu) provede atomická komponenta pro výpočet umístění vypočtených hodnot v krajních bodech mřížky na svůj výstup DO. Pomocí propojení se tyto výstupní data dostanou na vstup DIN sousední pravé komponenty, která je pomocí externí přechodové funkce zapracuje do své mřížky. Poté je proveden vlastní výpočet jednoho kroku pomocí interní přechodové funkce a cyklus se opakuje pro stanovený počet opakování (hodnota simulačního času dosáhne určité hodnoty). V každém kroku kořenový simulátor zašle atomickým komponentám pro výpočet pokyn pro uložení aktuálních hodnot. Komponenta uloží hodnoty z mřížky (hodnoty rychlosti a tlaku v každém bodu mřížky) a zašle je agentovi pro ukládání, který je zpracuje a rekonstruuje celou mřížku.

6.3 Experiment

Abychom mohli hodnotit naměřené výsledky experimentu, je třeba stanovit metriky pro hodnocení. Pro paralelní systémy se využívá tzv. *Amdahlův zákon*. Tento zákon si v následující sekci definujeme.

6.3.1 Amdahlův zákon

V [1] formuloval Gene Myron Amdahl vztahy pro zrychlení výpočtu dosažitelné paralelizací. Tyto vztahy známe jako *Amdahlův zákon* a měly výrazný vliv na rozvoj v oblasti paralelních výpočtů. Amdahlův zákon vychází z předpokladu, že každá aplikace má určitou část, kterou nelze paralelizovat a je tedy nutné jí provést sekvenčně. Pro další použití označme:

T_c celkový čas potřebný pro zpracování úlohy na jednom procesoru

T_p čas potřebný pro zpracování části úlohy, kterou lze rozdělit na nezávislé podúlohy

T_s čas potřebný pro zpracování části úlohy, kterou je nutné zpracovat sekvenčně.

Z definice plyne, že $T_c = T_p + T_s$. Předpokládejme, že paralelní část lze rozdělit na k stejně velkých a nezávislých částí a že máme k dispozici alespoň k procesorů. Pak mohou být všechny podúlohy zpracovány současně a celá úloha pak bude zpracována v čase $T(k)$, který je dán vztahem:

$$T(k) = T_s + \frac{T_p}{k}$$

Jedním z nejdůležitějších parametrů, které udávají zvýšení výkonnosti získané paralelizací, je *zrychlení* $S(k)$, které je definováno vztahem:

$$S(k) = \frac{T_c}{T(k)} \tag{6.7}$$

Jestliže symbolem β označíme podíl paralizovatelné části úlohy, lze vztah pro zrychlení upravit do následující podoby:

$$S(k) = \frac{T_c}{T(k)} = \frac{1}{(1-\beta) + \frac{\beta}{k}}, \quad kde \quad \beta = \frac{T_p}{T_c}$$

Jednoduše lze dokázat, že pro zrychlení platí $S(1) = 1$ a $1 \leq S(k) \leq k$. Nejdůležitější, co lze ze vztahu odvodit, je, že ať použijeme sebevětší počet procesorů, nikdy nedosáhneme zrychlení větší než:

$$\frac{1}{1-\beta}$$

Další důležitou metrikou určující úspěšnost paralelizace je *účinnost* $\epsilon(k)$, která je definována vztahem:

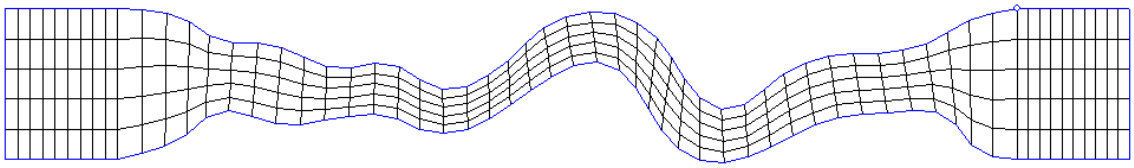
$$\epsilon(k) = \frac{S(k)}{k} \tag{6.8}$$

Pro naše potřeby nám plně postačí tyto dvě metriky.

6.3.2 Model experimentu

Jako experimentální model byl použit model obtékání tekutiny kolem pevných a statických těles. Fyzikální podstata a způsob řešení tohoto problému byly popsány v prvních sekcích této kapitoly. Jak již bylo uvedeno v 6.1.1, pro generování strukturované mřížky byl využit nástroj GenGrid.

Bylo vytvořeno sedm variant modelu. Nejjednodušší model je zobrazen na obrázku 6.5. V tomto modelu vtéká tekutina z levé strany modelu a na pravé straně z modelu vytéká, spodní a horní okraj modelu reprezentuje stěnu. Jednotlivé varianty se liší počtem bodů mřížky. Počet bodů mřížky má



Obrázek 6.5: *Model pro experiment (366 bodů)*

přímý vliv na náročnost výpočtu a tedy i na jeho délku. Z porovnání výsledků pro různé varianty bude ukázán vliv komunikace zasíláním zpráv na výkonnost celé simulace. Parametry jednotlivých variant modelu pro experiment shrnuje tabulka 6.2. Experiment byl prováděn v laboratořích výpočetní

Označení modelu	Bodů v ose x	Bodů v ose y	Celkově bodů	Velikost souboru [kB]
366 bodů	61	6	366	23.24
8091 bodů	261	31	8091	513.56
10941 bodů	521	21	10941	694.51
18971 bodů	311	61	18971	1204.22
32181 bodů	631	51	32181	2042.74
61641 bodů	716	81	61641	3912.77

Tabulka 6.2: *Parametry modelů (počet bodů mřížky) použitých v experimentu*

techniky Fakulty informačních technologií VUT v Brně, kde jsou k dispozici pracovní stanice s konfigurací: CPU AMD Athlon64 3200+, 512 MB RAM, pevný disk 160GB SATA, grafická karta GeForce 6600, kde každá pracovní stanice reprezentovala jeden výpočetní uzel distribuovaného systému.

6.3.3 Výsledky experimentu

Simulace byla prováděna v distribuovaném simulačním prostředí s jedním až deseti uzly. Experiment spočíval v provedení simulace modelu a měřením doby, za který byla simulace provedena. Pro každou variantu modelu a počet uzlů bylo provedeno jedno měření. Výsledky jsou shrnuty v tabulce 6.3. Grafická reprezentace naměřených hodnot je zobrazena grafy na obrázcích 6.6 a 6.7.

Výsledky jsou zobrazeny pomocí dvou grafů z důvodu velmi rozdílných hodnot doby trvání pro model s 336 body mřížky a s 61641 body mřížky. Z prvního grafu (obrázek 6.6) pro dobu trvání modelu s 366 body, 8091 body a 10941 body je vidět vliv komunikace na výkonnost celé simulace. Pro model s nejmenším počtem bodů mřížky (366) je distribuované zpracování kontraproduktivní a má zcela opačný vliv. Počet uzlů mřížky je příliš malý a výpočet je takto výkonnými stanicemi proveden velmi rychle. Podstatně větší část celkové doby simulace zabere komunikace, než samotný výpočet. A to již při dvou výpočetních uzlech. Z tohoto důvodu čas potřebný k simulaci paradoxně roste se s zvyšujícím se počtem výpočetních uzlů.

Příznivější naměřené hodnoty již vykazuje model s 8091 body mřížky. Pro počet výpočetních uzlů menší jak deset, čas s rostoucím počtem výpočetních uzlů klesá, pro deset výpočetních uzlů nastává stejný efekt jako pro předešlý model. Komunikace zabere více času, než vlastní výpočet na jednom výpočetním uzlu, a proto by se zvyšujícím počtem výpočetních uzlů zvyšoval i celkový čas simulace. Pokud mezi těmito modely porovnáme počet bodů mřížky na jeden výpočetní uzel distribuovaného

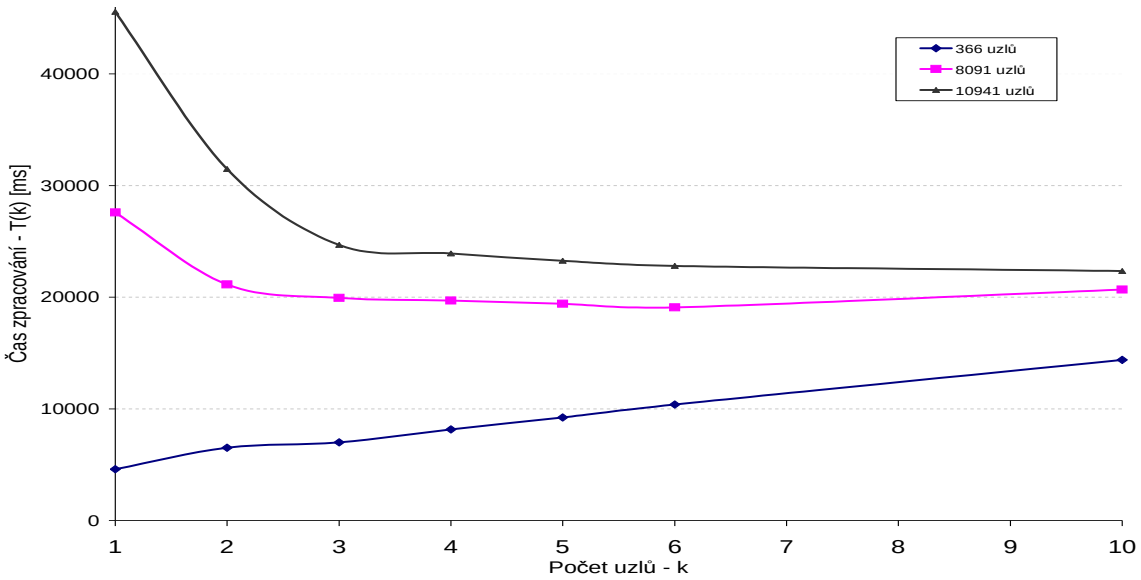
	Čas zpracování [ms]					
Uzlů	366 bodů	8091 bodů	10941 bodů	18971 bodů	32181 bodů	61641 bodů
1	4581	27598	45554	69203	159158	272404
2	6513	21140	31472	42255	82649	133487
3	7002	19950	24683	35140	59066	90870
4	8146	19705	23922	31749	48278	76680
5	9230	19405	23270	32372	43933	71390
6	10389	19093	22804	29669	41500	66878
10	14385	20677	22340	26350	39798	59020

Tabulka 6.3: *Přehled naměřených hodnot při experimentu*

systému při využití všech deseti výpočetních uzlů, dostaneme tyto výsledky: pro model s celkovým počtem 366 bodů mřížky máme přibližně 37 bodů, pro model s 8091 body mřížky máme přibližně 809 bodů a pro model s 10941 body mřížky máme 1094 bodů. Z naměřených hodnot je patrné, že simulace se stává neefektivní, pokud na jeden výpočetní uzel distribuovaného systému připadá přibližně méně jak 800 bodů mřížky. Proto lze předpokládat, že pro model s 10941 body mřížky by byla distribuovaná simulace neefektivní už při třinácti výpočetních uzlech.

Ve druhém grafu (obrázek 6.7) jsou zobrazeny naměřené hodnoty pro modely s větším počtem bodů mřížky. Ani při využití všech deseti výpočetních uzlů distribuovaného systému není dosažena empiricky nalezená hodnota 800 uzlů sítě na jeden výpočetní uzel systému. Proto u těchto modelů nenastává efekt popsaný v předchozím odstavci. Simulace by měla být nejefektivnější pro model s nejvíce body sítě (model s 61641 body mřížky), u kterého bude značně převažovat doba výpočtu nad dobou komunikace.

S využitím Amdahlova zákona (sekce 6.3.1) vypočteme zvýšení výkonnosti získaného paralelizací–zrychlení $S(k)$. Výpočet je proveden dle vztahu 6.7 Amdahlova zákona. Vypočtené hodnoty jsou uvedeny v tabulce 6.4, graf vypočtených hodnot je zobrazen na obrázku 6.8. Z tabulky a grafu hodnot je vidět, že pro model s počtem 366 bodů mřížky je zrychlení $S(k) < 1$, což v podstatě znamená zpomalení simulace. Důvod byl vysvětlen v předešlém odstavci. Nejlepších výsledků zrychlení paralelizací, dle očekávání, bylo dosaženo pro model s 61641 body mřížky.



Obrázek 6.6: *Závislost doby simulace na počtu výpočetních uzlů pro jednotlivé modely (1/2)*

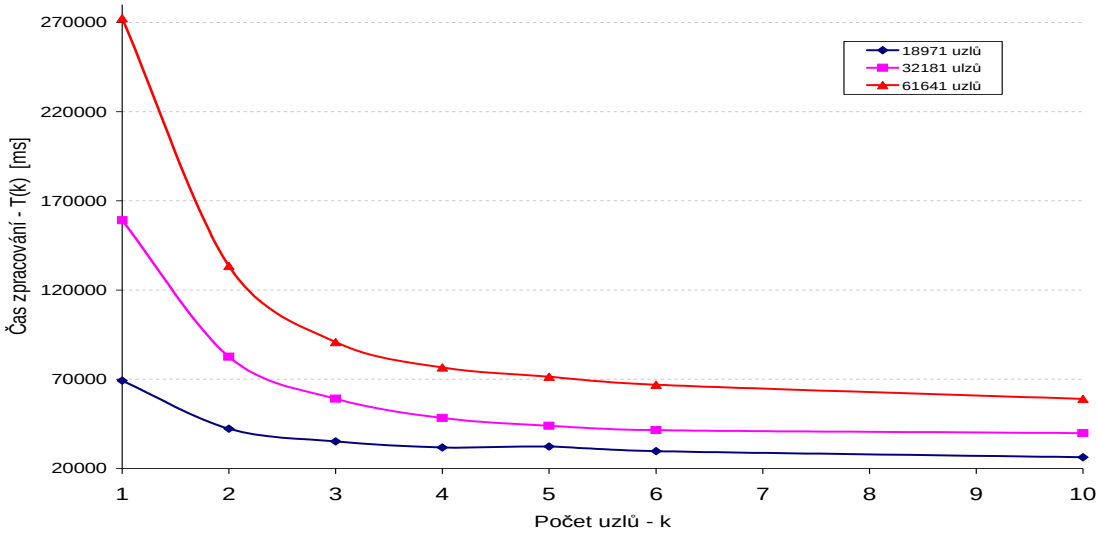
	Zrychlení $S(k)$					
Uzlů–k	366 bodů	8091 bodů	10941 bodů	18971 bodů	32181 bodů	61641 bodů
1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
2	0.7034	1.3055	1.4474	1.6377	1.9257	2.0407
3	0.6542	1.3834	1.8456	1.9694	2.6946	2.9977
4	0.5624	1.4006	1.9043	2.1797	3.2967	3.5525
5	0.4963	1.4222	1.9576	2.1377	3.6227	3.8157
6	0.4409	1.4455	1.9976	2.3325	3.8351	4.0731
10	0.3185	1.3347	2.0391	2.6263	3.9991	4.6155

Tabulka 6.4: *Tabulka hodnot zrychlení paralelního zpracování*

Poslední tabulkou s výsledky experimentu je tabulka 6.5. Tabulka obsahuje výsledky účinnosti paralelizace získané pomocí vztahu 6.8 Amdahlova zákona. Graf hodnot účinnosti paralelizace je na obrázku 6.9. Pouze u modelu s 61641 body mřížky a dvěma výpočetními uzly je tato hodnota vyšší než 1. Aby bylo dosaženo více hodnot vyšších než 1 bylo by zapotřebí ještě zvýšit počet bodů mřížky modelu, nebo pro testování využít méně výkonné pracovní stanice. Dle předpokladů, nejvyšší účinnost byla dosažena pro model s nejvyšším počtem bodů mřížky, u kterého byl nejvyšší poměr časů výpočtu ku času komunikace. Účinnost s rostoucím počtem výpočetních uzlů klesala. Vizualizace výsledků jsou uvedeny v příloze A.

Odhad chyby měření

Výsledky měření mohou být zkresleny dalšími úlohami, které mohly být v průběhu simulace spuštěny na jednotlivých stanicích. Měření bylo prováděno v operačním systému Linux. Na používané pracovní



Obrázek 6.7: *Závislost doby simulace na počtu výpočetních uzlů pro jednotlivé modely (2/2)*

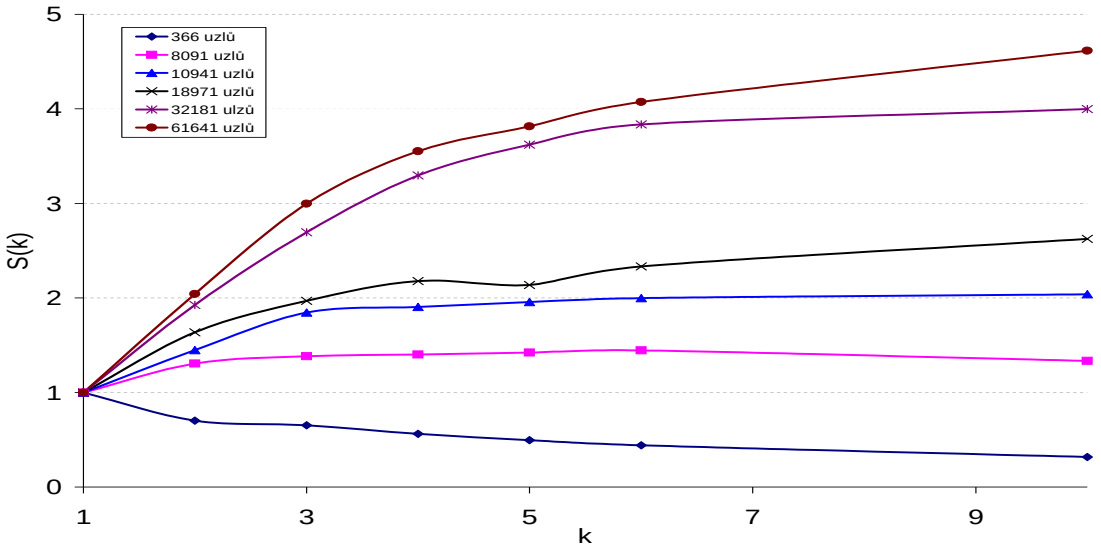
	Účinnosti paralelizace $S(k)$					
Uzlů–k	366 bodů	8091 bodů	10941 bodů	18971 bodů	32181 bodů	61641 bodů
1	1.000	1.000	1.000	1.000	1.000	1.000
2	0.352	0.653	0.724	0.819	0.963	1.020
3	0.218	0.461	0.615	0.656	0.898	0.999
4	0.141	0.350	0.476	0.545	0.824	0.888
5	0.099	0.284	0.392	0.428	0.725	0.763
6	0.073	0.241	0.333	0.389	0.639	0.679
10	0.032	0.133	0.204	0.263	0.400	0.462

Tabulka 6.5: *Tabulka hodnot účinnosti paralelizace*

stanice se mohou studenti a zaměstnanci fakulty připojit dálkově pomocí SSH, z tohoto důvodu byl před každou simulací na každé stanici kontrolován seznam aktuálně připojených uživatelů. V průběhu experimentu tyto stanice nebyly využívány žádným dalším uživatelem, a proto se lze domnívat, že v průběhu měření byly na stanicích spuštěny pouze systémové úlohy.

Velikost chyby měření lze odhadnout z rozdílů mezi jednotlivými simulacemi stejného modelu. V tabulce 6.6 jsou uvedeny časy naměřené pro model s minimálním a maximálním počtem bodů mřížky s různými počty výpočetních systémů.

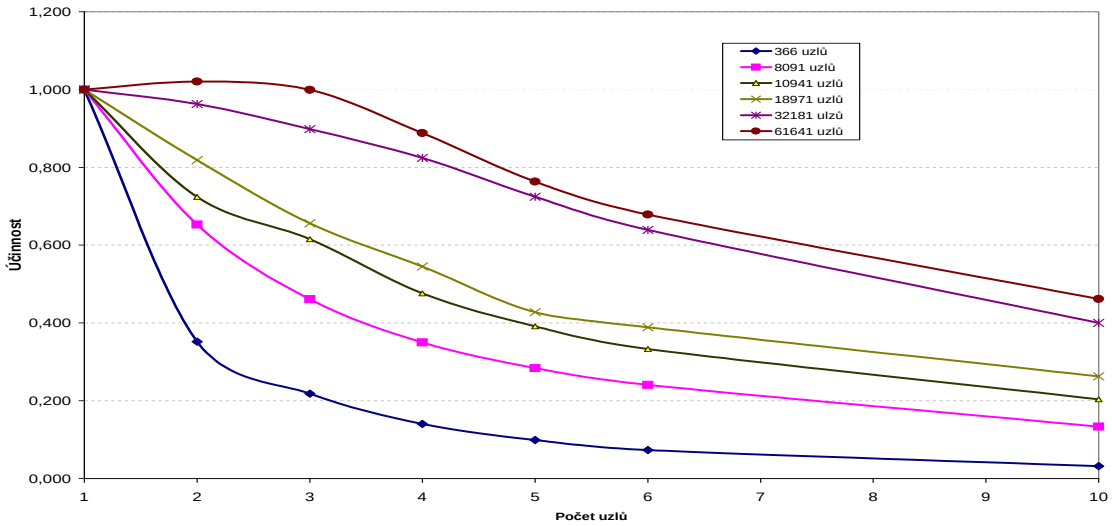
Všechny odchylky jsou v řádu procent. Výše uvedené výsledky považovat za směrodatné, pro účely demonstrace paralelizace jsou dostatečně přesné.



Obrázek 6.8: *Závislost zrychlení získaného paralelizací na počtu výpočetních uzlů*

	Čas zpracování [ms]			
	366 bodů		61641 bodů	
	1 stanice	4 stanice	1 stanice	4 stanice
1. simulace	4581	7447	272404	75354
2. simulace	4724	8146	274154	76680
3. simulace	4298	8458	268957	78521
Průměrná hodnota $\bar{x}$	4534.33	8017.00	271838.33	76851.67
Směrodatná odchylka s	177.02	422.70	2159.04	1298.61
Kolik % činní s z $\bar{x}$	3.904%	5.273%	0.794%	1.690%

Tabulka 6.6: *Tabulka odhadu chyby měření*



Obrázek 6.9: *Závislost účinnosti paralelizace na počtu výpočetních uzlů*

Kapitola 7

DEVSML

Cílem kapitoly je prezentovat navržený jazyk DEVSML (*DEVS formalism MetaLanguage*) pro specifikace simulačních modelů založených na DEVS formalismu. Modely implementované pomocí DEVSML jsou nezávislé na konkrétním simulačním prostředí pro DEVS formalismus. V první části je vysvětlena motivace vzniku jazyka, ve druhé části popisuje jazyk DEVSML včetně transformace modelů pro různá simulační prostředí, jako je například *DEVSJava* [47] či *DEVS/C++* [49]. V závěru kapitoly jsou diskutovány výhody využití jazyka DEVSML.

V současné době je v simulačních systémech trend seskupovat modely a jejich simulátory do těsně provázaných balíčků. Všeobecně platí, že při vytváření nového simulačního prostředí jsou všechny modely opakovaně konstruovány od základů, implementované simulační modely není možné sdílet s jinými simulačními systémy. Z tohoto důvodu by bylo vhodné vytvořit standard pro specifikace simulačních modelů, který by tento problém odstranil. Tento standard by umožnil sdílet implementace simulačních modelů mezi různými simulačními prostředími, což by ušetřilo čas a finanční prostředky vynaložené na implementaci modelu a simulačního prostředí. Bylo by možné vytvářet knihovny běžně využívaných komponent modelů.

Celá disertační práce se zabývá DEVS formalismem, proto jsem se při řešení tohoto problému soustředil na DEVS formalismus. V dnešní době existuje mnoho implementací tohoto formalismu jako například *DEVS/C++* [49], *DEVSJava* [47] apod. Problémem těchto implementací je nemožnost sdílení modelů. Modely jsou obvykle implementovány prostředky jazyka, ve kterém je implementován simulátor. Simulační modely implementované v těchto simulačních prostředích jsou úzce spjaty s těmito simulátory a nelze je využít v jiné implementaci DEVS formalismu.

Byl navržen metajazyk pro specifikaci simulačních DEVS modelů. Modely specifikované tímto metajazykem lze potencionálně transformovat do jakéhokoliv simulačního prostředí pro DEVS bez nutnosti dalších úprav simulačního modelu. Po návrhu tohoto jazyka jsme začali vyvíjet prototyp modelovacího nástroje, který zjednoduší implementaci simulačních modelů pomocí navrženého jazyka. Modelovací nástroj umožňuje graficky specifikovat model, a to včetně přechodových a transformačních funkcí atomické komponenty.

Následující sekce zařazuje tuto práci do oblasti standardizace DEVS formalismu výzkumné skupiny zabývající se přímo standardizací DEVS formalismu [40]. V sekci se také zabývám některými existujícími nástroji a diskutuji jejich přednosti či nevýhody. Jádrem kapitoly je sekce popisující navržený metajazyk pro specifikaci simulačních DEVS modelů a krátký popis prototypové implementace modelovacího nástroje. V závěru práce je diskutována další budoucnost projektu, v závěru kapitoly jsou diskutovány výhody, které tento způsob specifikace simulačních DEVS modelů přináší.

7.1 Existující systémy a nástroje

Existuje řada implementací DEVS formalismu ([39] uvádí seznam nejpoužívanějších implementací), z nejznámějších můžeme jmenovat *DEVS/C++* [49], *DEVSJava* [47] a *PythonDEVS* [3]. V dnešní době je za referenční implementaci DEVS formalismu považován simulační nástroj *DEVSJava*.

Výzkumná skupina *DEVS Standardization Group* [40] se zabývá výzkumem v oblasti standardizace DEVS formalismu. Její výzkum je rozdělen do čtyř základních oblastí: první oblastí (1) je standardizace DEVS formalismu, druhou oblastí (2) je standardizace reprezentace DEVS modelů, třetí oblastí (3) je standardizace rozhraní DEVS simulátoru a čtvrtou oblastí (4) je standardizace knihoven DEVS modelů. Její členové publikovali práce (například [10], [38]) především z oblasti (2), které navrhují způsoby specifikace simulačních modelů DEVS formalismu. Popis struktury modelu je založen na využití XML (*Extensible Markup Language*) [42], popis přechodových funkcí atomických komponent je vypuštěn či jsou implementovány pomocí pseudo kódu. Pro obecné modelování vyvinuli modelovací nástroj AToM^3 [16], který je založen na meta modelování a transformacích modelu. Prvním krokem v tomto nástroji je specifikace modelovacího formalismu, poté následuje specifikace vlastního modelu. Nástroj je založen na grafových gramatikách a umožňuje transformace modelů na stejné (či jiné) formalismy a automatické generování simulačních nástrojů. Jedná se o velmi vyspělý teoretický nástroj.

Tato práce zapadá do výzkumných oblastí (2) a (4). Na internetu jsem nalezl dva projekty, které s touto problematikou úzce souvisí. Prvním projektem je DEVSW [45]. V tomto projektu je popis struktury simulačního modelu založen na XML, popis přechodových a transformačních funkcí atomických komponent je založen na pseudo kódu. Druhý projekt [29] k popisu simulačního DEVS modelu využívá pouze XML. Nejzajímavější je popis transformačních funkcí atomických komponent, jsou popsány pouze pomocí XML. Popis funkcí je tvořen pomocí pravidel s konečně mnoha stavy, tímto přístupem lze popsat pouze chování konečného automatu.

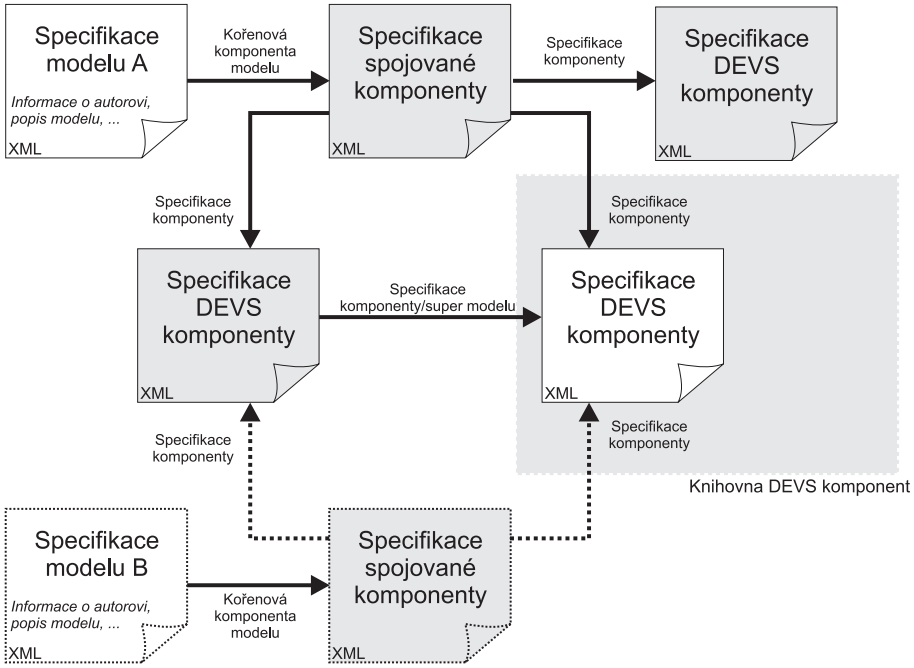
7.2 Návrh jazyka DEVSML

Velmi elegantní řešení představuje projekt [29], v tomto projektu je jeden zásadní problém. Definice funkcí pouze pomocí dvou uživatelských značek degraduje jejich vyjadřovací schopnost na regulární jazyky. Přičemž z definice DEVS formalismu vyplývá, že ve funkcích atomické komponenty lze implementovat jakoukoliv parciálně rekurzivní funkci. Jejich vyjadřovací schopnost je tedy stejná jako Turingova stroje [21]. Vyjadřovací schopnost těchto funkcí definuje vyjadřovací schopnost celého DEVS formalismu. Pokud bychom jejich schopnost omezili, omezíme tím vyjadřovací schopnost celého DEVS formalismu. Proto volba způsobu definice těchto funkcí je velmi důležitá.

DEVSML (*DEVS formalism MetaLanguage*) je založen na jazyce XML. Dokument založený na XML je hierarchická struktura, stejně tak jako struktura spojovaného DEVS modelu. Proto specifikace spojované komponenty pomocí XML nepředstavuje velký problém. Hlavní problém představuje specifikace atomické komponenty. A to především způsob implementace přechodových a transformačních funkcí této komponenty. Pro popis sémantiky funkcí bylo také využito pouze XML. Dalším důležitým úkolem bylo navrhnout způsob ukládání modelu do XML dokumentů. Například projekt [45] ukládá celý model do jediného XML dokumentu. Pro zpracování je to velmi pohodlný přístup, ale není příliš vhodný pro znovu použití některých komponent v jiném modelu. Především z tohoto důvodu bylo navrženo ukládání každé komponenty v samostatném XML dokumentu. Model je definován pomocí jednoduchého XML dokumentu, který obsahuje základní informace o modelu jako je název modelu, popis modelu a informace o autorovi. Obsahuje také odkaz na kořenovou spojovanou komponentu reprezentující daný model. Kořenová spojovaná komponenta obsahuje odkazy na definice vnořených komponent. Obrázek 7.1 ukazuje příklad popsaného způsobu ukládání modelů pomocí několika XML

dokumentů. Tento přístup umožní velmi jednoduše vytvářet a využívat knihovny umístěné na internetu. Běžně využívané komponenty se nemusí znovu implementovat, což zjednoduší a urychlí vývoj komplexních modelů.

Hlavní XML dokument specifikace modelu A na obrázku 7.1 obsahuje informace o modelu (autor, popis apod.). Uvnitř dokument je odkaz na další XML dokument, který obsahuje definici spojované komponenty (na obrázku jsou odkazy reprezentovány šipkou). V dokumentu se specifikací spojované komponenty se lze odkazovat na dokumenty se specifikacemi dalších komponent, které jsou využity pouze v tomto modelu (na obrázku dokument vpravo nahoře), nebo na dokument z knihovny DEVS komponent či na dokument s definicí komponenty, který je využit i v jiném model (na obrázku model B).



Obrázek 7.1: *Uložení modelů pomocí DEVSML*

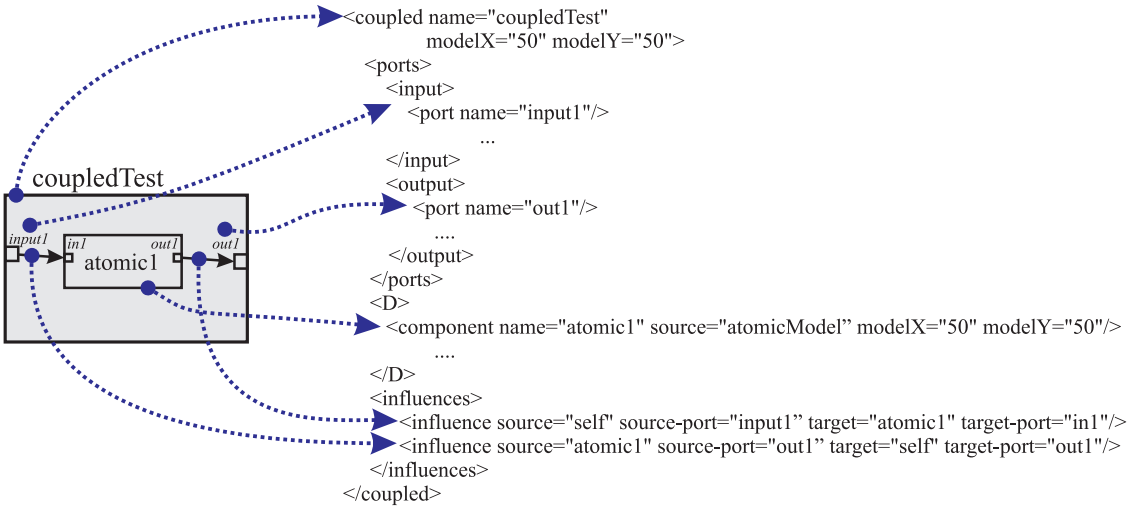
Příloha C obsahuje kompletní DTD (*Document Type Definition*) [42] jazyka DEVSML.

7.2.1 Spojovaná komponenta

XML se specifikací spojované DEVS komponenty obsahuje definici vstupních a výstupních portů komponenty, seznam jmen vnitřních komponent včetně odkazů na dokumenty se specifikacemi komponent a definici propojení komponent. Každá DEVS komponenta je tedy definována v samostatném XML dokumentu, což umožňuje velmi jednoduše do vytvářených modelů připojit již připravené komponenty. Připojen může být jak lokální soubor, tak i soubor umístěný na internetu. Na obrázku 7.2 je uveden příklad popisu spojované komponenty pomocí DEVSML.

7.2.2 Atomická komponenta

Definice vstupních a výstupních portů atomické komponenty je obdobná jako definice portů u spojované komponenty. Problém je implementace transformačních a přechodových funkcí komponenty. Při návrhu zápisu funkcí jsme se nechali inspirovat projektem *JavaML* [2]. Projekt převádí zdrojový kód



Obrázek 7.2: *Ukázka specifikace spojované komponenty pomocí DEVSML*

programu zapsaného v programovacím jazyce Java do popisu pomocí jazyka XML. Na tomto XML jsou poté prováděny optimalizace kódu a verifikace. Z tohoto projektu jsem převzal způsob zápisu základních programových konstrukcí a zápisu výrazů.

Mezi základní konstrukční prvky jazyka patří prvky pro nastavení a získání hodnoty proměnné (*setVar*, *getVar*), větvení programu na základě logické podmínky (*if–then–else*). Jazyk obsahuje tři typy cyklů. Cyklus s podmínkou na začátku (*while*), cyklus s podmínkou na konci (*until*) a cyklus se známým počtem průchodů (*for*). Dále obsahuje prvky pro získání a nastavení hodnoty stavové proměnné (*getStateVar*, *setStateVar*). Přičemž za stavovou proměnnou modelu je považována každá globální proměnná komponenty. Stavové proměnné se definují ve značce *state–variables* pomocí značky *state–variable*. Tato značka má dva povinné atributy, jméno proměnné a datový typ. Nepovinným parametrem je výchozí hodnota proměnné. V tuto chvíli jsou definovány pouze čtyři datové typy, celé číslo, reálné číslo, řetězec znaků a logický datový typ.

Pro zjednodušení specifikace atomických komponent jsme zavedli pojem *super model* (analogie super třídy v objektově orientovaném programování). Super model představuje rodičovský model, ze kterého se dědí vlastnosti atomické komponenty (definice portů, stavové proměnné, funkcí, . . .).

Následuje fragment implementace funkce pomocí DEVSML:

```
<code>
  <if>
    <test>
      <binary-expr op=">">
        <var-ref name="A"/>
        <literal-number value="10"/>
      </binary-expr>
    </test>
    <true-case>
      <loop kind="for">
        <init>
          <local-variable name="i" type="integer"/>
        </init>
        <update>
          <unary-expr op="++" post="true">
            <var-ref name="i"/>
          </unary-expr>
        </update>
        <test>
```

```
            <binary-expr op="<=">
              <var-ref name="i"/>
              <literal-number value="5"/>
            </binary-expr>
          </test>
          <set-state-var name="B[i]">
            <literal-number value="0"/>
          </set-state-var>
        </loop>
      </true-case>
    </if>
    <set-state-var name="A">
      <literal-number value="5"/>
    </set-state-var>
  </code>
```

7.2.3 Zdrojové jazyky pro DEVSML

Implementace modelu přímo pomocí jazyka DEVSML je možná, ale také velmi zdlouhavá a složitá. Pro snadnější implementaci modelů bude pro uživatele výhodnější využít běžný programovací jazyk či jednoduchý pseudojazyk.

Hierarchická struktura modelu může být specifikována pomocí grafického nástroje. Například na obrázku 7.2 je definice spojované komponenty, která obsahuje jednu atomickou komponentu. Transformace hierarchické struktury specifikované pomocí grafické notace je poměrně jednoduchou záležitostí. Grafická specifikace modelu a jejich transformace je jednou z hlavních vlastností našeho experimentálního modelovacího nástroje.

Mnohem složitějším problémem je specifikace chování atomické komponenty. Jednou z možností specifikace těchto funkcí je využití jednoduchého pseudojazyka. V současné době probíhají experimenty s jazykem, který se podobá jazyku *Lisp*[1].

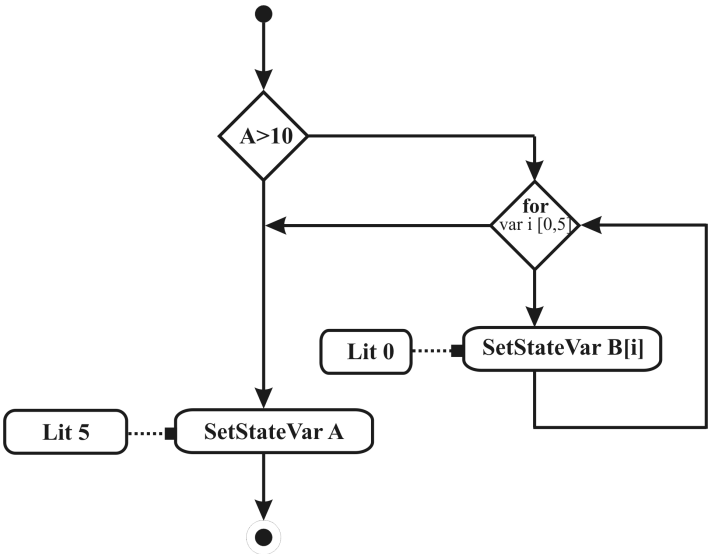
```
(if (> A 10)
  (for i from 0 to 5 do
      (set B[i] 0)))
(set A 5)
```

Implementace tohoto fragmentu pseudojazyka byla použita v ukázce implementace funkcí atomické komponenty pomocí DEVSML v předešlé sekci.

Pokud budeme mít k dispozici překladač jazyka na DEVSML, lze DEVSML využít pouze pro ukládání modelu a jeho transformaci. Specifikace funkcí lze provádět pomocí běžného jazyka či pseudojazyka. Pokud je k dispozici i překladač z DEVSML do příslušného jazyka, mohou být uložené DEVSML modely mnohem komfortněji prohlíženy a upravovány.

Jednou z okrajových oblastí našeho současného výzkumu je využití grafické specifikace nejen pro hierarchickou strukturu modelu, ale i pro definici chování atomické komponenty. Zkoumáme možnost využití grafického jazyka vycházejícího z UML [12], zvláště potom z diagramu aktivit. Ukázka implementace chování atomické komponenty pomocí tohoto grafického jazyka je zobrazena na obrázku 7.3. Nicméně, tento způsob implementace je stále velice experimentální a výzkum se především soustředí na využití pseudokódu a jiných programovacích jazyků pro implementaci chování atomické komponenty.

[1] Lisp programming language, `http://en.wikipedia.org/wiki/Lisp_programming_language`



Obrázek 7.3: *Ukázka implementace funkce atomické komponenty pomocí grafické notace*

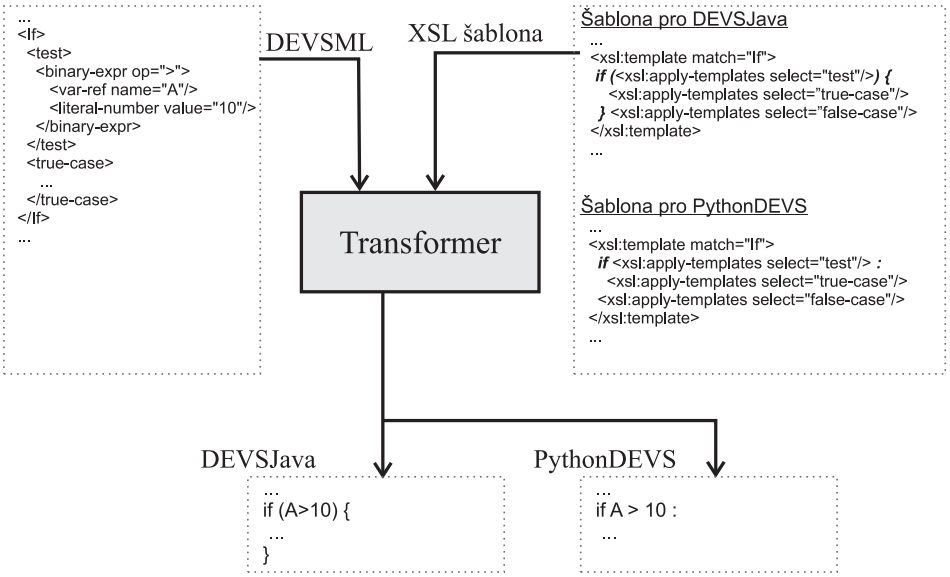
7.3 Výpočetní síla funkcí popsaných pomocí DEVSML

Cílem této sekce je dokázat, že třída problémů řešitelných v tomto jazyce je totožná s třídou problémů řešitelných pomocí Turingova stroje [21]. Nebude tedy docházet k omezení vyjadřovací síly modelu popsaných pomocí tohoto jazyka. Důkaz bude proveden pomocí jednoduchého skeletového jazyka (též nazývaného *Bare–Bones Programming Language*) [21, strana 64]. Je navržen pro výpočet parciálně rekurzivních funkcí a je vytvořen tak, aby byl co nejjednodušší. Obsahuje pouze jediný datový typ, kterým je přirozené číslo. Dále obsahuje tři základní programové konstrukce, inkrementace proměnné, její dekrementace a cyklus s podmínkou. Tento cyklus je prováděn tak dlouho, dokud hodnota proměnné v podmínce je různá od nuly. Je dokázané [21, strany 66–68], že pomocí tohoto skeletového jazyka je možné implementovat libovolnou parciálně rekurzívní funkci a naopak funkce implementovatelné ve skeletovém jazyce jsou parciálně rekurzivní. Protože parciálně rekurzívní funkce jsou vyčíslitelné pomocí Turingova stroje a naopak, libovolný proces Turingova stroje je procesem výpočtu parciálně rekurzívní funkce, platí, že skeletový jazyk umožňuje výpočet libovolného algoritmicky řešitelného problému.

Jazyk DEVSML pokrývá všechny konstrukce skeletového jazyka a datové typy DEVSML lze zakódovat pomocí přirozeného čísla, lze prohlásit, že třída problémů řešitelných v jazyce DEVSML je rovná třídě problémů řešitelných pomocí Turingova stroje. Tedy navržený jazyk DEVSML nijak neomezuje vyjadřovací sílu DEVS formalismu.

7.4 Transformace modelu

Pojem transformace modelu zde reprezentuje proces, ve kterém je model specifikovaný pomocí jazyka DEVSML transformován na jiný ekvivalentní zápis použitelný ve vybraném DEVS simulačním prostředí. Jedná se tedy o transformaci syntaxe specifikace modelu na jinou formu syntaxe při zachování sémantiky modelu. Celý systém transformace se skládá ze dvou základních stavebních prvků. Prvním prvkem je proces, který řídí celou transformaci, druhým prvkem je XSLT procesor [43] provádějící vlastní transformaci modelu na základě XSLT šablony [43]. Šablona obsahuje pravidla



Obrázek 7.4: *Princip transformace modelu specifikovaného pomocí DEVSML pro požadované sim. prostředí*

pro transformaci syntaxe modelu, to znamená, že pro každé simulační prostředí musí být definována XSLT šablona.

Řídící proces transformace postupně načítá definiční XML dokumenty jednotlivých komponent modelu a aplikuje na tyto jednotlivé dokumenty XSL transformaci s příslušnou šablonou. Ukázka transformace je na obrázku 7.4. Přidání podpory pro další DEVS simulační prostředí spočívá v implementaci XSLT šablony. Do systému není potřeba nijak zasahovat.

7.5 Přínosy jazyka DEVSML

DEVSML odstraňuje závislost simulačních DEVS modelů od implementace DEVS simulátoru. Model specifikovaný tímto jazykem je potenciálně přenositelný do jakéhokoliv DEVS simulátoru, což je hlavním přínosem této práce. Model lze také přenést do speciálního simulačního prostředí, jako je například distribuované či real–time simulační prostředí. Vývoj modelu může probíhat v sekvenčním simulačním prostředí, ve kterém se obvykle jednodušeji provádí testování. Výsledný model poté může být simulován ve specializovaném simulačním prostředí. DEVSML také umožňuje vytvářet knihovny běžně využívaných modelů a tím urychlit specifikaci modelů. Další výhoda používání DEVSML, je možnost automatické verifikace nového simulačního prostředí. Tu lze provádět pomocí formálního důkazu, ale to je obvykle velmi složité a zdlouhavé. V [40] byl představen jiný způsob, který spočívá v automatickém porovnávání výsledků simulace s výsledky simulace stejného modelu v referenčním simulačním prostředí. DEVSML umožňuje model sdílet mezi testovaným a referenčním simulačním prostředím.

Kapitola 8

Závěr

Předložená disertační práce se zabývá problematikou distribuované simulace a portability simulačních modelů mezi různými simulačními nástroji. Celá práce zde bude shrnuta v několika krocích. Nejdříve bude uveden postup řešení, poté budou zopakovány nejvýznamnější dosažené výsledky, na něž logicky navazuje část vymezující další možnosti výzkumu v této oblasti. Kapitolu uzavírá přehled publikací autora, které přímo souvisejí s touto disertační prací.

Zvolený přístup k řešení

V oblasti distribuované simulace je řada úkolů, přičemž nebylo pochopitelně možné se věnovat v této práci všem. Po prostudování existujících architektur pro simulační systémy a existujících simulačních systémů, z nichž nejzajímavější byly uvedeny v kapitole 2, bylo největší úsilí věnováno dvěma vybraným oblastem. Tyto dvě oblasti se posléze staly stěžejními částmi práce. Jedná se o návrh architektury distribuovaného simulačního prostředí a portabilitu simulačních DEVS modelů. Obě tyto oblasti byly důkladně zkoumány. Navržená architektura je založena na dvou základních stavebních kamenech, které byly popsány v kapitole 3. Distribuovanost v navržené architektuře zajišťuje agentní systém, modelování a simulace zajišťuje DEVS formalismus. Pro zajištění portability simulačních modelů byl navržen metajazyk DEVSML. DEVSML je metajazyk založený na technologii XML. Transformace modelů specifikovaných pomocí DEVSML je pro různá simulační prostředí prováděna pomocí XSL transformace. Metajazyk DEVSML a princip transformace byl prezentován v kapitole 7.

Dosažené výsledky

Dosažené výsledky lze rozdělit do dvou oblastí. První oblastí je návrh architektury distribuovaného simulačního systému. Na základě této architektury byl implementován prototyp distribuovaného simulačního nástroje, který implementoval podstatné části architektury. Pomocí prototypu byla úspěšně provedena simulace komplexního modelu. Druhou oblastí je návrh metajazyka DEVSML pro specifikaci simulačních DEVS modelů. V úvodní kapitole práce byly prezentovány tři stanovené cíle disertační práce. Nyní bude zhodnoceno jejich splnění:

1. *Navrhnout architekturu distribuovaného simulačního systému. Systém založený na této architektuře bude schopen reagovat na dynamické změny prostředí, jako je například změna počtu výpočetních uzlů v průběhu simulace.*
   Návrh architektury byl prezentován v kapitole 4. Pro dosažení požadovaných vlastností byla využita architektura založená na agentním systému. V rámci této kapitoly byly prezentovány i metody, které zajistí odolnost distribuovaného simulačního prostředí vůči výpadkům uzlů a jakými způsoby lze implementovat metody pro optimalizaci distribuované simulace. Obě tyto

metody implementují schopnost distribuovaného simulačního systému dynamicky reagovat na změny prostředí, ve kterém systém pracuje.

2. *Navrhnout metajazyk pro specifikaci simulačních modelů založených na DEVS formalismu. Implementované modely nebudou závislé na konkrétním simulačním prostředí pro DEVS. Tyto modely bude potenciálně možné transformovat pro jakékoliv simulační prostředí určené pro DEVS formalismus.*
   Návrh jazyka byl prezentován v kapitole 7. Navržený metajazyk pro popis simulačních DEVS modelů, nazvaný jako DEVSML, umožňuje implementovat simulační modely, které jsou zcela nezávislé na konkrétním DEVS simulátoru. Tyto modely lze potencionálně transformovat do jakéhokoliv simulačního prostředí pro DEVS, bez nutnosti dalších úprav simulačního modelu. V současné chvíli je rozpracována transformace pro simulátory DEVSJava, DEVS/C++ a PythonDEVS a také pro prototyp distribuovaného simulačního nástroje popsaného v kapitole 5.

3. *Implementovat prototyp simulačního prostředí založený na navržené architektuře a provést simulaci netriviálního modelu.*
   Prototyp byl implementován v programovacím jazyce Java. Jeho popis je uveden v kapitole 5. Pomocí tohoto prototypu byla provedena simulace komplexního modelu, jejíž výsledky byly prezentovány a diskutovány v kapitole 6.

Možnosti dalšího výzkumu

Následující výzkum v oblasti distribuované simulace bude navazovat na výsledky a úvahy prezentované v této práci. Výzkum lze rozdělit na dvě části, první část se bude zabývat distribuovanou simulací obecně. Zásadním problémem v této části výzkumu je dokončení implementace simulačního prostředí, tj. přejít z prototypové realizace na plně funkční a použitelné distribuované simulační prostředí. Dalším velice zajímavým směrem výzkumu je využití umělé inteligence pro optimalizaci simulace a optimalizace modelu pro distribuované zpracování. V této oblasti výzkumu bude pokračovat Michal Cerhák[1]. Druhou částí výzkumu je problematika portability simulačních DEVS modelů. V současné chvíli je tento výzkum na svém počátku. Kapitola 7 představila navržený metajazyk pro specifikace simulačních DEVS modelů. V dalším výzkumu v této oblasti bude pokračovat kolega Ing. Petr Polášek[2], který bude dále rozvíjet navržený metajazyk a bude pokračovat v implementaci prototypu modelovacího nástroje, jehož implementace byla zahájena v rámci této práce.

Související publikace

Předložená disertační práce byla podpořena celou řadou publikací autora, které zde budou uvedeny v chronologickém pořadí. V práci byly využity výsledky dosažené již v rámci inženýrského studia [58, 57]. Koncepce DEVS–BUS pro heterogenní modelování pomocí DEVS formalismu byla popsána v [53]. Pří vývoji distribuovaného simulačního prostředí je výhodné využívat různé vizualizační nástroje, popis této problematiky lze nalézt v [51]. V [54] byl představen navržený koncept pro distribuovanou simulaci dopravy pomocí DEVS formalismu a ve stejném roce byla představena navržená architektura distribuovaného simulačního prostředí v [59]. Návrh architektury byl později rozšířen a nová verze architektury společně s ukázkou distribuované simulace byla publikována v [60]. První verze metajazyka pro specifikace simulačních DEVS modelů byla publikována v [55], rozšířená a upravená verze byla publikována na vědecké konferenci v [56].

[1] Nyní student 5. ročníku magisterského studia, v září 2006 nastupuje do 1. ročníku doktorského stud. programu na Ústav inteligentních systémů, FIT VUT v Brně

[2] Nyní student 1. ročníku doktorského studijního programu na Ústavu inteligentních systémů, FIT VUT v Brně

Literatura

[1] Amdahl, G., M. Validity of the single–processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Atlantic City, New Jersey, United States of America, 1967.

[2] Badros, G. JavaML: a markup language for java source code. In *Proceedings of the 9^{th} international World Wide Web conference on Computer networks: the international journal of computer and telecommunications netowrking*, pages 159–177, Amsterdam, Netherlands, 2000.

[3] Bolduc, J.S. and Vangheluwe, H. The modelling and simulation package PythonDEVS for classical hierarchical devs. Technical report, McGill University, 2001. MSDL technical report MSDL-TR-2001-01.

[4] Bryan, H., Mailler, R. and Lesser, V. The Farm Distributed Simulation Environment. Computer Science Technical Report 2004-12, University of Massachusetts, March 2004.

[5] Chandy, K. and Misra, J. Distributed simulation: A case study in design and verification of distributed system. In *IEEE Transactions on Software Engineering*, pages 440–452, 1979.

[6] Dahmann, J. S., Fujimoto, R. and Weatherly, R. M. The department of defense high level architecture. In *WSC '97: Proceedings of the 29^{th} conference on Winter simulation*, pages 142–149, New York, USA, 1997. ACM Press.

[7] Department of Artificial Intelligence, Faculty of Computer Science, University of ULM. JAMES. [online] <`http://www.informatik.uni-ulm.de/ki/james.html`>. poslední revize 21. březen 2001, [cit. 16.5.2006].

[8] Filippi, J.B. JDEVS, DEVS OO modeling and simulation toolkit for ecosystem modeling. [online] <`http://spe.univ-corse.fr/filippiweb`>, poslední revize únor 2004 [cit. 10.5.2006].

[9] Filippi, J.B., Bernardi, F. and Delhom, M. The JDEVS hybrid modelling and simulation environment. In *Predicting hydrologic response from physio–climatic attributes: an application to ungauged sub–catchments of the Burdekin River, North Queensland*, volume 3, Lugano, Switzerland, 2002. iEMSs.

[10] Fishwick, P. XML based modeling and simulation using XML for simulation modeling. In *WSC '02: Proceedings of the 34^{th} conference on Winter simulation*, pages 616–622, San Diego, California, 2002. Winter Simulation Conference.

[11] Arizona Center for Integrative Modeling & Simulation. ACIMS Software Development. [online] <`http://www.acims.arizona.edu/SOFTWARE/software.shtml`>. poslední revize září 14 2003, [cit. 16.5.2006].

[12] France, R., Evans, A. and Lano, K. The UML as a formal modeling notation. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.

[13] Haluza, M. Základní kapitoly z hydromechaniky. [online] <`http://khzs.fme.vutbr.cz/~vvhal/SKRIPTA/SK9/Sk9.html`>. poslední revize 2002, [cit. 16.5.2006].

[14] Janoušek, V. *Modelování objektů Petriho sítěmi*. PhD thesis, Ústav informatiky a výpočetní techniky, Fakulta elektrotechniky a informatiky, Vysoké učení technické v Brně, 1998. s. 137.

[15] Kihyung, K. and Wonseok, K. Efficient distributed simulation of hierarchical DEVS models: Transforming Model Structure into a Non–Hierarchical one. *Proceedings of the 33rd Annual Simulation Symposium*, 2000.

[16] Lara, J. and Vangheluwe, H. AToM^3: A tool for multi-formalism modelling and meta-modelling. In *Lecture Notes in Computer Science 2306, FASE/ETAPS'02*, pages 174–188, Grenoble, France, 2002.

[17] Littlejohn, A., Badcock, K. and Baldwin, A. CFD Illustration–Boundary Layers. [online] <http://www.aero.gla.ac.uk/Research/CFD/education/course/CALF>. [cit. 16.5.2006].

[18] Lockheed Martin Advanced Technology Laboratories. CSIM – Performance Simulator. [online] <`http://www.atl.external.lmco.com/projects/csim`>. poslední revize 15. června 2006, [cit. 16.5.2006].

[19] Luke, S. and Balan, C. Multi–Agent Simulator Of Neighborhoods. [online] <`http://cs.gmu.edu/~eclab/projects/mason`>. [cit. 16.5.2006].

[20] Molnár, V. Počítačová dynamika tekutín. [online] <`http://www.cfd.sk/cfd-book`>. poslední revize 2002, [cit. 16.5.2006].

[21] Motyčková, L., Češka, M. and Hruška, T. *Vyčíslitelnost a složitost*. Ediční středisko VUT Brno, 1992. ISBN 80-214-0441-8.

[22] Pegden, C. and Pritsker, B. SLAM tutorial. In *Proceedings of the 13th conference on Winter simulation*, pages 91–100, Atlanta, Georgia, 1981.

[23] Pečiva, L. Prototyp multiagentního systému. Master's thesis, Ústav inteligentních systémů, Fakulta informačních technologií, Vysoké učení technické v Brně, 2005.

[24] Peringer, P. *Hierarchické modelování na bázi komunikujících objektů*. PhD thesis, Ústav informatiky a výpočetní techniky, Fakulta elektrotechniky a informatiky, Vysoké učení technické v Brně, 1996. s. 82.

[25] Pitch Technologies AB. High Level Architecture: High–Level Architecture for beginners. [online] <`http://www.pitch.se/hla/hlaforbeginners.asp`>. [cit. 16.5.2006].

[26] Rábová, Z., Zendulka, J., Češka M., Peringer P., Janoušek, V. *Modelování a simulace*. Nakladatelství VUT, 1992. ISBN 80-214-0480-9.

[27] Server Root.cz. Lehký úvod do LDAP. [online] <`http://www.root.cz/clanky/lehky-uvod-do-ldap`>. ISSN 1212-8309, poslední revize 24. července 2000, [cit. 16.5.2006].

[28] Schattenberg, B. and Uhrmacher, A. Planning agents in James. In *Proceedings of the IEEE*, volume 89, pages 158–173, 2001.

[29] Schäfer, A. Visualisierung und XML–Darstellung von DEVS–Modellen. Master's thesis, Universität der Bundeswehr München, 2003.

[30] UMBC Computer Science and Electrical Engineering Department. UMBC AgentWeb. [online] <`http://www.csee.umbc.edu/kqml`>. poslední revize prosinec 2005, [cit. 10.4.2006].

[31] IEEE Computer Society standards organization. FIPA Abstract Architecture Specification. [online] <`http://www.fipa.org/specs/fipa00001`>. poslední revize 6. prosinec 2002, [cit. 16.5.2006].

[32] IEEE Computer Society standards organization. FIPA Agent Management Specification. [online] <`http://www.fipa.org/specs/fipa00023`>. poslední revize 18. duben 2002, [cit. 16.5.2006].

[33] IEEE Computer Society standards organization. FIPA (Foundation for Intelligent Physical Agents). [online] <`http://www.fipa.org`>. [cit. 10.4.2006].

[34] Inc. Sun Microsystems. Java Remote Method invocation (Java RMI). [online] <`http://java.sun.com/products/jdk/rmi`>. [cit. 16.5.2006].

[35] Swarm Development Group. Swarm. [online] <`http://www.swarm.org`>. poslední revize 4. červen 2006, [cit. 16.5.2006].

[36] Tilab. Java Agent DEvelopment Framework. [online] <`http://jade.cselt.it`>. [cit. 17.5.2006].

[37] University of Massachusetts at Amherst, Multi–Agent Systems Laboratory, Leser, V. Multi–Agent Systems Lab homepage. [online] <`http://mas.cs.umass.edu`>. [cit. 16.5.2006].

[38] Vangheluwe, H. and Lara, J. Meta–Models are models too. In *Proceedings of the 34th Winter Simulation Conference: Exploring New Frontiers*, pages 597–605, San Diego, California, 2002.

[39] Wainer, G. Devs standardization group : DEVS tools. [online] <`http://www.sce.carleton.ca/faculty/wainer/standard`>. poslední revize 21. listopadu 2005, [cit. 16.5.2006].

[40] Wainer, G. Devs standardization study group. Technical report, The SISO Standards Activities Committee (SAC), 2005. Interim final report.

[41] Woodridge, M. and Jennings, N. *Intelligent Agents: Theory and Practice*. Cambridge University Press, 1994.

[42] World Wide Web Consortium W3C. Extensible markup language (XML). [online] <`http://www.w3.org/XML`>. poslední revize 22. dubna 2006, [cit. 16.5.2006].

[43] World Wide Web Consortium W3C. The extensible stylesheet language family XSL. [online] <`http://www.w3.org/Style/XSL`>. poslední revize 22. dubna 2006, [cit. 16.5.2006].

[44] Young, J. K. and Tag, G. K. A heterogeneous simulation framework based on the DEVS BUS and the high level architecture. In *Proceedings of the 30th conference on Winter simulation*, pages 412–428, Los Alamitos, CA, USA, 1998.

[45] Yung–Hsin, W. and Lung-Hsiung, W. A modeling and simulation example using DEVSW. In *The 31st Annual Simulation Symposium*, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[46] Zbořil, F. *Plánování a komunikace v multiagentních systémech*. PhD thesis, Ústav inteligentních systémů, Fakulta informačních technologií, Vysoké učení technické v Brně, 2004. s. 105.

[47] Zeigler, B. P. DEVS–JAVA User's guide. Technical report, AI & Simulation Lab., Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ., 1997.

[48] Zeigler, B. P. and Chow, A. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, pages 716–722, San Diego, CA, USA, 1994.

[49] Zeigler, B. P. and Kim, J. G. DEVS/C++ a high performance modelling and simulation environment. In *Hawaii international conference on system sciences–HICSS*, pages 350–359, Waikoloa, Big Island Hawaii, 1996.

[50] Zeigler, B. P. and Praehofer, H. *Theory of Modeling and Simulation, second edition*. Academic Press, 2000. ISBN 0-12-778455-1.

Publikace autora

[51] Florián, V. and Slavíček, P. Roles of visualiazation tools in distributed simulations. In *Proceedings of the 10^{th} conference STUDENT EEICT 2004*, pages 339–343, 2004. Brno, Czech Republic.

[52] Florián, V., Hanáček, P. and Slavíček, P. Analysing methods for modelling attacks on security devices. In *Proceedings of 38^{th} International Conference MOSIS'04*, pages 261–265, 2004. Ostrava, Czech Republic.

[53] Janoušek, V. and Slavíček, P. Heterogenní simulace na bázi DEVS. In *Proceedings of XXVth International Autumn Colloquium ASIS 2003*, pages 213–218, 2003. Sv. Hostýn, Czech Republic.

[54] Janoušek, V. and Slavíček, P. Concept for the parallel road–traffic simulation. In *Proceedings of 39^{th} Spring International Conference Modelling and Simulation of Systems MOSIS'05*, page 6, 2005. Hradec nad Moravicí, Czech Republic.

[55] Janoušek, V., Polášek, P. and Slavíček, P. Metajazyk pro popis DEVS formalismu. In *NETSS 2006*, pages 19–25, 2006. Přerov, Czech Republic.

[56] Janoušek, V., Polášek, P. and Slavíček, P. Towards DEVS meta language. In *The Industrial Simulation Conference 2006*, 2006. Palermo, Italy.

[57] Slavíček, P. Discrete Event system Specification–DEVS. In *Proceedings of the 9^{th} Conference and Competition STUDENT EEICT*, page 3, 2003. Brno, Czech Republic.

[58] Slavíček, P. Distribuovaná a heterogenní simulace. Master's thesis, Ústav inteligentních systémů, Fakulta informačních technologií, Vysoké učení technické v Brně, 2003.

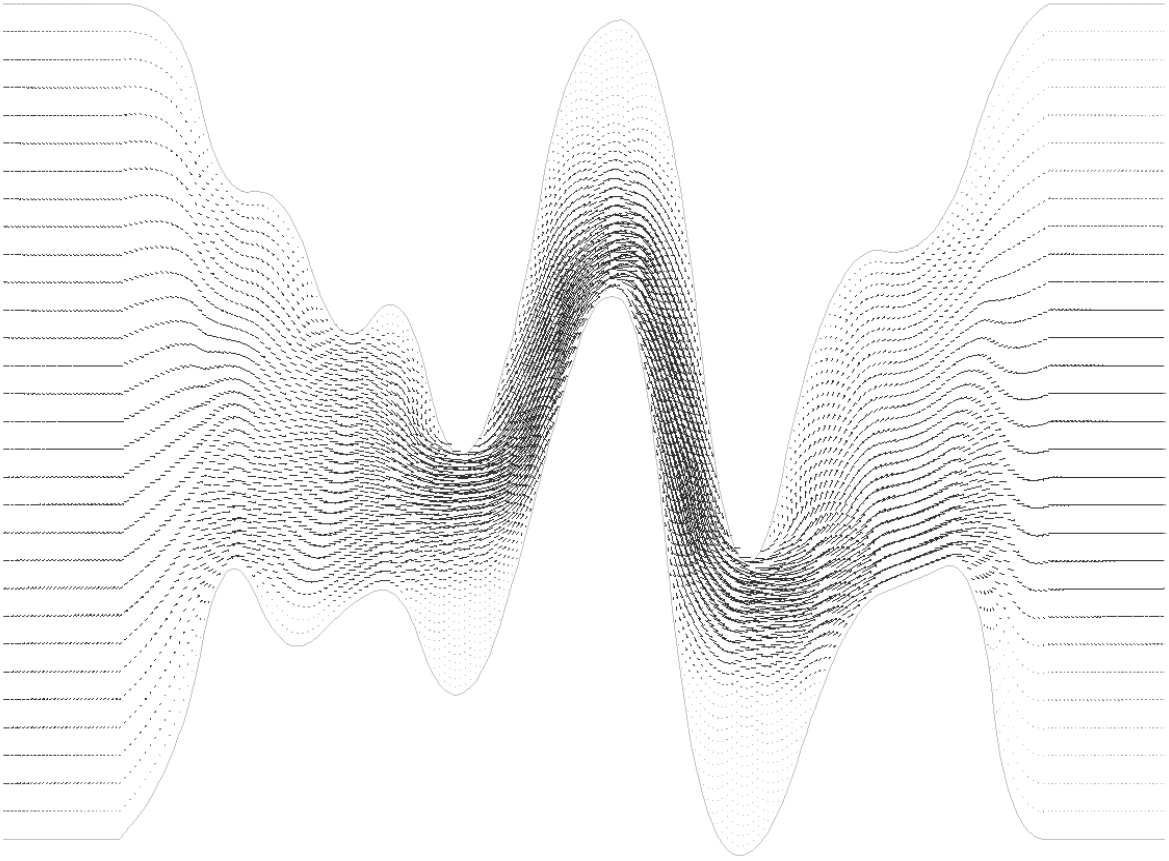
[59] Slavíček, P. Distribuované simulační prostředí. In *Proceedings of XXVIIth International Autumn Colloquium ASIS 2005*, page 6, 2005. Přerov, Czech Republic.

[60] Slavíček, P. Distributed simulation environment. In *1^{st} Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 114–120, 2005. Znojmo, Czech Republic.

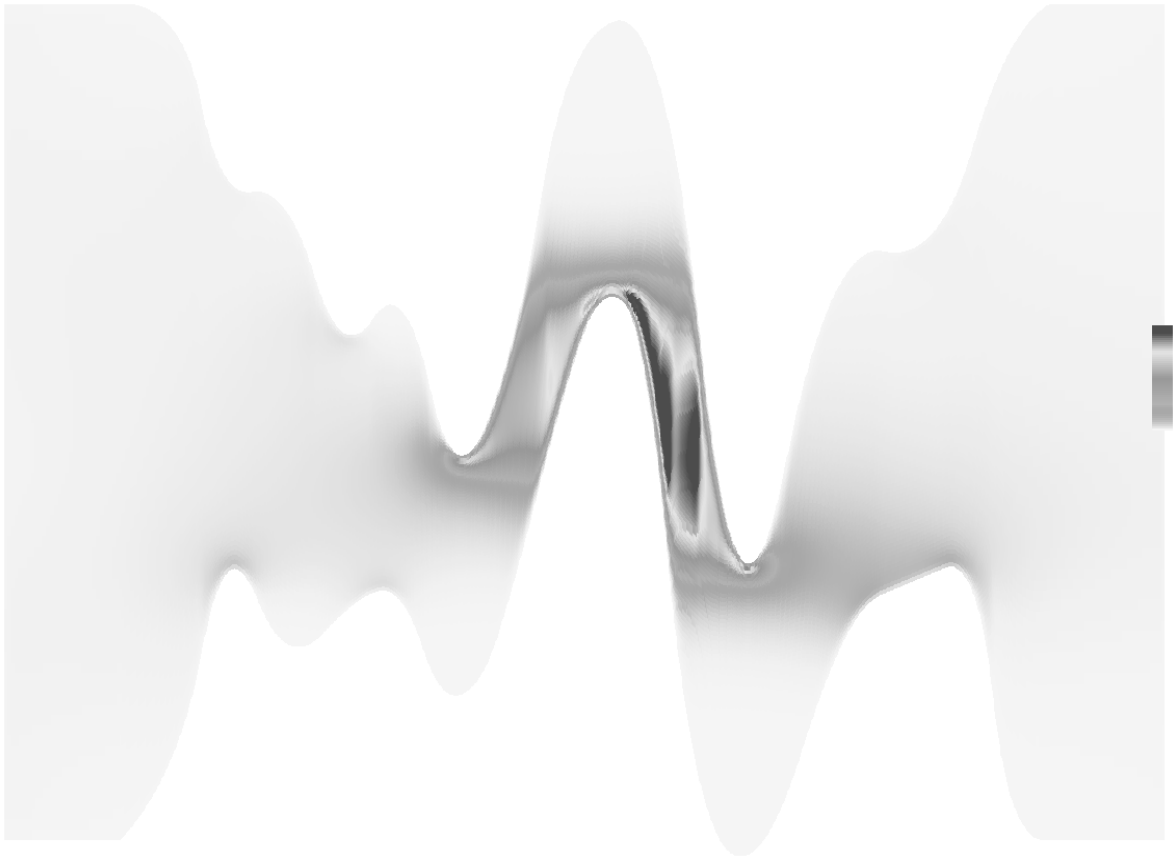
Příloha A

Ukázky vizualizace výsledků simulace

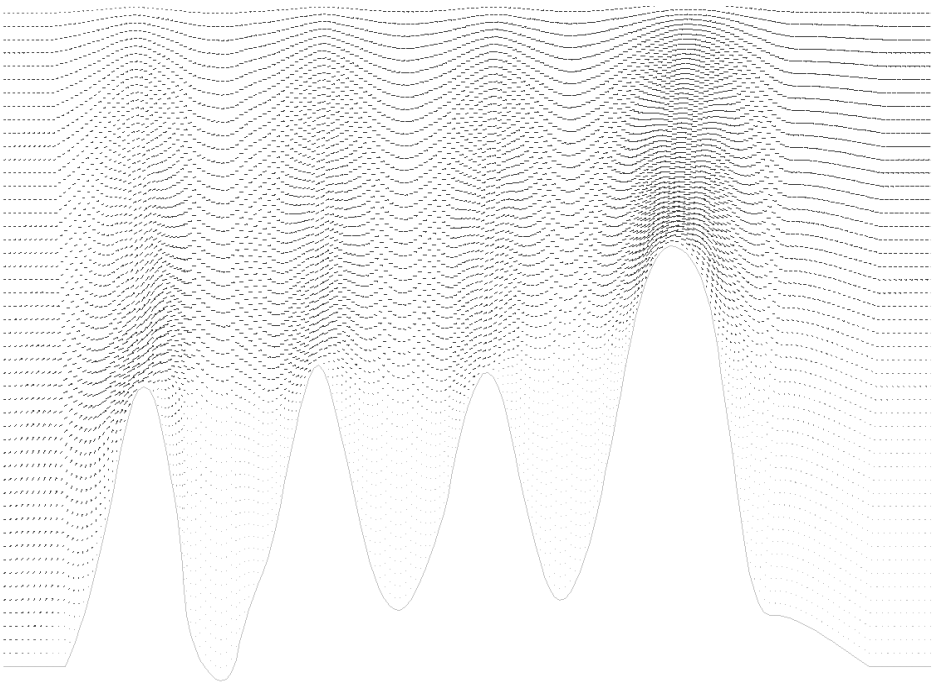
Příloha obsahuje ukázky vizualizace výsledků simulací z oblasti dynamiky tekutin. Na obrázcích A.1 a A.2 jsou vizualizace výsledků simulace modelu, který byl použit v kapitole 6 (pro lepší zobrazení bylo použito jiné měřítko os x a y, než na obrázku 6.5).



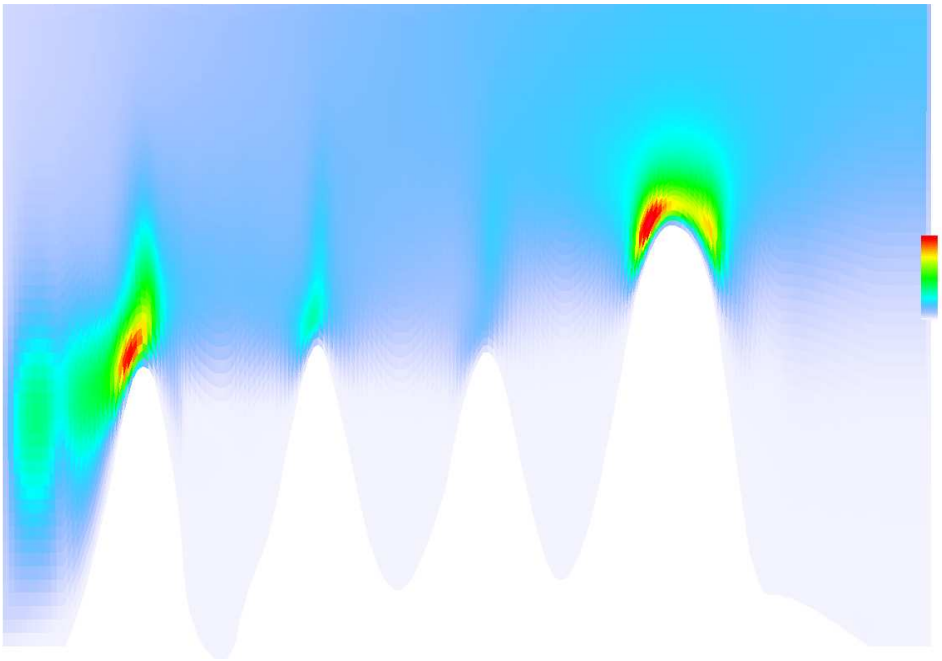
Obrázek A.1: *Model s 8091 body sítě, vektory rychlosti*



Obrázek A.2: *Model s 8091 body sítě, velikost rychlosti*



Obrázek A.3: *Model s 10251 body sítě, vektory rychlosti*



Obrázek A.4: *Model s 10251 body sítě, velikost rychlosti*

Příloha B

Další ukázkový model

Příloha obsahuje další ukázkové modely, na kterých byla testována funkčnost prototypu distribuovaného simulčního prostředí.

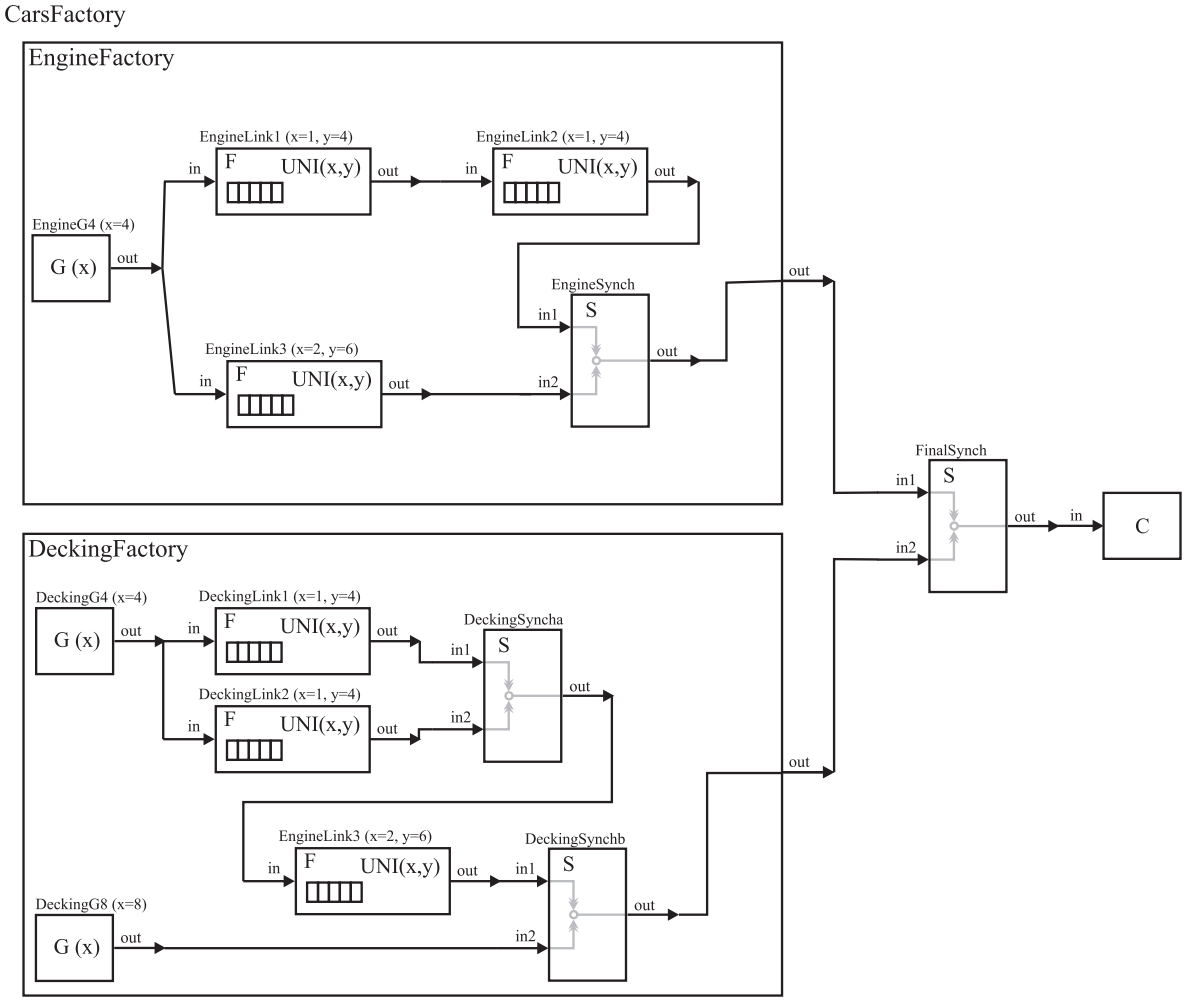
Jako další ukázkový model uvažujme značně zjednodušený model továrny na automobily. Továrna se skládá ze dvou divizí, první divize (v modelu označená jako $EngineFactory$) sestavuje motory pro vozidla, druhá divize (v modelu označená jako $DeckingFactory$) vyrábí karosérie. Do divize pro motory jsou každé 4 minuty dodávany potřebné komponenty pro sestavení jednoho motoru. Divize obsahuje dvě linky pro montáž komponent, které pracují paralelně ($EngineLink1$ a $EngineLink2$, duhá linka $EngineLink3$). Výstupní sestavené komponenty obou linek se montují v jeden celek, který je výsledkem této divize. Do divize pro karosérie jsou komponenty pro karosérii dodávány dvěma zdroji. První zdroj dodává kažé 4 minuty komponenty do dvou různých linek pro další montáž ($DeckingLink1$ a $DeckingLink2$). Sestavené komponenty z obou linek se sestavují v jeden celek, který se předává k dalšímu zpracování na lince $DeckingLink3$. Druhý zdroj dodává komponenty každých 8 minut, které se sestavují v jeden celek společně s komponentami z linky $DeckingLink3$. Celek je výstupem z této divize.

Sestavené části z obou divizí jsou sestavovány v jeden celek, který je výsledným produktem. Model popsané továrny je na obrázku B.1. Jedná se o spojovanou komponentu, která obsahuje dvě atomické komponenty a dvě spojované komponenty. V dalším textu budou tyto komponenty definovány.

CarsFactory

$$CarsFactory = \langle X_{self}, Y_{self}, D, \{M_i | i \in D\}, \{I_i | i \in D \cup \{self\}\}, \{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}\rangle$$

$$\begin{aligned}
X_{self} &= \{\emptyset\} \\
Y_{self} &= \{\emptyset\} \\
D &= \{EngineFactory, DeckingFactory, FinalSynch, C\} \\
& M_{EngineFactory} = EngineFactoryCoupled, M_{DeckingFactory} = DeckingFactoryCoupled, \\
& M_{FinalSynch} = SynchAtomic, M_C = CounterAtomic \\
& I_{EngineFactory} = \{FinalSynch\}, I_{DeckingFactory} = \{FinalSynch\}, I_{FinalSynch} = \{C\} \\
& Z_{EngineFactory,FinalSynch} = \{(out, in1)\}, Z_{DeckingFactory,FinalSynch} = \{(out, in2)\}, \\
& Z_{FinalSynch,C} = \{(out, in)\}
\end{aligned}$$



Obrázek B.1: *Model továrny na automobily*

EngineFactoryCoupled

$$EngineFactoryCoupled = \langle X_{self}, Y_{self}, D, \{M_i | i \in D\}, \{I_i | i \in D \cup \{self\}\}, \{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}\rangle$$

$$\begin{aligned}
X_{self} &= \{\emptyset\} \\
Y_{self} &= \{out\} \\
D &= \{EngineG4, EngineLink1, EngineLink2, EngineLink3, EngineSynch\}
\end{aligned}$$

$M_{EngineG4} = GeneratorAtomic, M_{EngineLink1} = M_{EngineLink2} = M_{EngineLink3} = FacilityAtomic,$
$M_{EngineSynch} = SynchAtomic$

$I_{EngineG4} = \{EngineLink1, EngineLink3\}, I_{EngineLink1} = \{EngineLink2\},$
$I_{EngineLink2} = \{EngineSynch\}, I_{EngineLink3} = \{EngineSynch\}, I_{EngineSynch} = \{self\}$

$Z_{EngineG4,EngineLink1} = \{(out, in)\}, Z_{EngineG4,EngineLink3} = \{(out, in)\},$
$Z_{EngineLink1,EngineLink2} = \{(out, in)\}, Z_{EngineLink2,EngineSynch} = \{(out, in1),$
$Z_{EngineLink3,EngineSynch} = \{(out, in2)\}, Z_{EngineSynch,self} = \{(out, out)\}$

DeckingCoupled

$$DeckingCoupled = \langle X_{self}, Y_{self}, D, \{M_i | i \in D\}, \{I_i | i \in D \cup \{self\}\}, \{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}\rangle$$

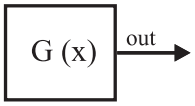
$$\begin{aligned} X_{self} &= \{\emptyset\} \\ Y_{self} &= \{out\} \\ D &= \{DeckingG4, DeckingG8, DeckingLink1, DeckingLink2, DeckingSyncha, DeckingLink3, DeckingSynchb\} \end{aligned}$$

$M_{DeckingG4} = M_{DeckingG8} = GeneratorAtomic,$
$M_{DeckingLink1} = M_{DeckingLink2} = M_{DeckingLink3} = FacilityAtomic,$
$M_{DeckingSyncha} = M_{DeckingSynchb} = SynchAtomic$

$I_{DeckingG4} = \{DeckingLink1, DeckingLink2\}, I_{DeckingLink1} = \{DeckingSyncha\},$
$I_{DeckingLink2} = \{DeckingSyncha\}, I_{DeckingSyncha} = \{DeckingLink3\},$
$I_{DeckingLink3} = \{DeckingSynchb\}, I_{DeckingG8} = \{DeckingSynchb\}, I_{DeckingSynchb} = \{self\}$

$Z_{DeckingG4,DeckingLink1} = \{(out, in)\}, Z_{DeckingG4,DeckingLink2} = \{(out, in)\},$
$Z_{DeckingLink1,DeckingSyncha} = \{(out, in1)\}, Z_{DeckingLink2,DeckingSyncha} = \{(out, in2),$
$Z_{DeckingSyncha,DeckingLink3} = \{(out, in)\}, Z_{DeckingLink3,DeckingSynchb} = \{(out, in1)\}$
$Z_{DeckingG8,DeckingSynchb} = \{(out, in2)\}, Z_{DeckingSynchb,sef} = \{(out, out)\}$

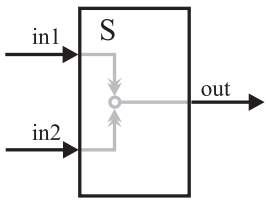
GeneratorAtomic



$$GeneratorAtomic = \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{con}, Y, \lambda\rangle$$

$S = \{on\} \quad X = \{\emptyset\} \quad Y = \{out\} \quad ta(s) = x \quad \delta_{int}(s) = on \quad \delta_{ext}((e, s), b) = on \quad \delta_{con}(s, b) = on \quad \lambda(on) = 1$

SynchAtomic



$$SynchAtomic = \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{con}, Y, \lambda\rangle$$

$S = \{ACTIVE, EMPTY, WAITING\} \quad X = \{in1, in2\} \quad Y = \{out\}$

$\delta_{con}(s, b) = \{\delta_{ext}((0, s), b); \delta_{int}(s)\}$

$ta(s) =$

```
{
   if (s==ACTIVE) {
     return 0;
   }
   return MAX_VALUE;
}
```

$\delta_{int}(s) =$

```
{
   if (in1.isEmpty() && in2.isEmpty()) {
     return EMPTY;
   } if (in1.isEmpty() || in2.isEmpty()) {
     return WAITING;
   } else {
     return ACTIVE;
   }
}
```

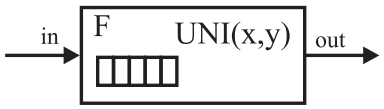
$\delta_{ext}((e,s),b) =$

```
{
  Object fin1 = getValueFromInput("in1");
  Object fin2 = getValueFromInput("in2");
  if (fin1!=null) {
    in1.add(fin1);
  }
  if (fin2!=null) {
    in2.add(fin2);
  }
  if (in1.isEmpty() || in2.isEmpty()) {
    return WAITING;
  }
  return ACTIVE;
}
```

$\lambda(s) =$

```
{
  Object v1 = in1.firstElement();
  Object v2 = in2.firstElement();
  in1.remove(v1);
  in2.remove(v2);
  setValueToOutput("out", 1);
}
```

FacilityAtomic



$$FacilityAtomic = \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{con}, Y, \lambda \rangle$$

$S = \{EMPTY\_QUEUE, ACTIVE\} \quad X = \{in\} \quad Y = \{out\}$

$\delta_{con}(s,b) = \{\delta_{ext}((0,s),b); \delta_{int}(s)\}$

$ta(s) =$

```
{
   return nextTime;
}
```

$\delta_{int}(s) =$

```
{
  if (queue.isEmpty()) {
    nextTime = MAX_VALUE;
    return EMPTY_QUEUE;
  } else {
    QueueItem q = queue.firstElement();
    nextTime = q.delayTime;
    return ACTIVE;
  }
}
```

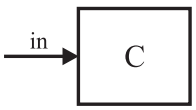
$\delta_{ext}((e,s),b) =$

```
{
   Object value = getValueFromInput("in");
   QueueItem item = new QueueItem();
   item.value = value;
   item.delayTime = generateUniRanTime(x,y);
   item.incomeTime = getActualTime();
   queue.add(item);
   if (s==EMPTY_QUEUE) {
     return ACTIVE;
   } else {
     nextTime = nextTime - e;
   }
}
```

$\lambda(s) =$

```
{
  QueueItem q = queue.firstElement();
  queue.remove(q);
  setValueToOutput("out", q.value);
}
```

CounterAtomic



$$CounterAtomic = \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{con}, Y, \lambda \rangle$$

$S = \{on\} \quad X = \{\emptyset\} \quad Y = \{out\} \quad ta(s) = MAX\_VALUE \quad \delta_{int}(s) = on \quad \delta_{ext}((e,s),b) = counter++$
$\delta_{con}(s,b) = on \quad \lambda(s) = \emptyset$

Výsledky simulace

Cílem této simulace nebyla optimalizace výroby v továrně, cílem byla pouze demonstrace funkčnosti simulačního prostředí, především komunikace mezi komponentami modelu v průběhu simulace.

Simulace byla prováděna v intervalu simulačního času 0 až 480. Při testování byl pro každou simulaci změněn interval dodávek komponent do divize pro karosérie, který je modelován komponentou $DeckingG8$. Z modelu vyplývá, že tato komponenta bude mít značný vliv na celkovém počtu vyrobených vozů. Pokud budou intervaly dodávek krátké, komponenta $DeckingG8$ nebude ovlivňovat celkový počet vyrobených vozů a nezpracované komponenty se budou hromadit na vstupu $in2$ komponenty $DeckingSynchb$. Naopak, pokud bude interval dodávek velký, dramaticky se sníží počet vyrobených vozů a na vstupu $in2$ komponenty $DeckingSynchb$ nebude žádná komponenta. Z tohoto důvodu byl měněn interval dodávek a bylo sledováno celkové množství vozů a počet komponent, které čekají na zpracování vstupu $in2$ komponenty $DeckingSynchb$.

Při intervalu dodávek 4 minuty bylo celkové množství vyrobených vozů 108, na konci simulace čekalo na zpracování na vstupu $in2$ komponenty $DeckingSynchb$ 8 komponent karosérie. Z toho jasně vyplývá, že celkový počet vyrobených vozů ovlivňovaly ostatní komponenty modelu. Pokud byl interval dodávek zvýšen na 15 minut, celkové množství vyrobených vozů kleslo na 31 a na vstupu vstupu $in2$ komponenty $DeckingSynchb$ čekala 1 komponenta. Nakonec byl interval dodávek zvýšen na 25 minut, celkové množství vyrobených vozů kleslo na 18 a na vstupu vstupu $in2$ komponenty $DeckingSynchb$ nečekala žádná komponenta. Z toho jasně výplývá, že tento interval dodávek komponent je nedostatečný a brzdí celou výrobu. Výsledky simulace dopovídají předpokladům.

Příloha C

DTD jazyka DEVSML

C.1 Definice modelu

```
<!ELEMENT model (name, description, authors*, root-model)>

<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT authors (author+)>
<!ELEMENT author (name, email?, institute?, icq?, contact?, phone?, note?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT institute (#PCDATA)>
<!ELEMENT icq (#PCDATA)>
<!ELEMENT contact (#PCDATA)>

<!ELEMENT root-model (source)>
<!ELEMENT source (#PCDATA)>
```

Ukázka definice modelu

```
<model>
  <name>Example model A</name>
  <description>This is an example model definition</description>
  <authors>
    <author>
      <name>Pavel Slavíček</name>
      <email>slavicek@fit.vutbr.cz</email>
      <contact>FIT, BUT, Božetěchova 2, 612 66 Brno, Czech Republic</contact>
    </author>
  </authors>
  <root-model>
    <source>file://coupledModelA.xml</source>
  </root-model>
</model>
```

C.2 Definice spojovaného modelu

```
<!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 incluences (incluence*)>

<!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>
```

Ukázka definice spojovaného modelu

```
<coupled name="coupledTest" modelX="50" modelY="50">
  <ports>
    <input>
      <port name="input1"/>
      <port name="input2"/>
    </input>
    <output>
      <port name="output2"/>
    </output>
  </ports>

  <D>
    <component name="atomic1" source="file://atomicModelA.xml"
               modelX="50" modelY="50"/>
    <component name="atomic2" source="file://atomicModelA.xml"
               modelX="250" modelY="30"/>
    <component name="atomic3" source="file://atomicModelB.xml"
               modelX="130" modelY="150"/>
    <component name="atomic4" source="file://atomicModelB.xml"
               modelX="250" modelY="200"/>
  </D>

  <influences>
    <influence source="atomic1" source-port="out1"
               target="atomic2" target-port="in2"/>
    <influence source="atomic1" source-port="out1"
               target="atomic3" target-port="in1"/>
    <influence source="atomic3" source-port="out1"
               target="atomic2" target-port="in1"/>
    <influence source="atomic3" source-port="out1"
               target="atomic4" target-port="in1"/>
  </influences>
</coupled>
```

C.3 Definice atomického modelu

```
<!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 ta (block)>
<!ELEMENT internal-transition-function (block)>
<!ELEMENT external-transition-function (block)>
<!ELEMENT output-function (block)>
<!ELEMENT functions (function*)>

<!ELEMENT state-variable EMPTY>
<!ATTLIST state-variable name CDATA #REQUIRED>
<!ATTLIST state-variable type (integer|string|real|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>
```

Ukázka definice spojovaného modelu

```
<atomic name="atomicModel" modelX="50" modelY="50" super-model="superAtomicModel">
  <state-variables>
     <state-variable name="state1" type="integer" initial-value="0"/>
  </state-variables>

  <ports>
    <input>
      <port name="in1"/>
      <port name="in2"/>
    </input>
    <output>
      <port name="out1"/>
    </output>
  </ports>

  <ta>
    <block> .... </block>
  </ta>

  <internal-transition-function>
    <block> .... </block>
  </internal-transition-function>

  <external-transition-function>
    <block> .... </block>
  </external-transition-function>

  <output-function>
    <block> .... </block>
  </output-function>

  <functions>
    <function name="fnc1" type="integer" visibility="protected">
      <parameters>
        <parameter name="param1" type="integer"/>
      </parameters>
      <block> .... </block>
```

```
    </function>
  </functions>
</atomic>
```

Definice funkcí atomického modelu

```
<!ELEMENT block ((%stmt-elements;)*)>

<!ENTITY %expr-elems "conditional-expr|binary-expr|unary-expr|literal-number|
                      literal-string|literal-boolean|literal-null">
<!ENTITY % stmt-elems "block|local-variable|if|loop|do-loop|return|
                       function-call|get-state-var|continue|break|%expr-elems;">

<!ELEMENT local-variable (type,(%expr-elems;)?)>
<!ATTLIST local-variable name CDATA #REQUIRED>

<!ELEMENT var-ref EMPTY>
<!ATTLIST var-ref name CDATA #REQUIRED>

<!ELEMENT assignment-expr (lvalue,(%expr-elements;))>

<!ELEMENT lvalue (var-set)>

<!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 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 get-state-var EMPTY>
<!ATTLIST get-state-var name CDATA #REQUIRED>

<!ELEMENT set-state-var (%expr-elements;)>
```

```
<!ATTLIST set-state-var name CDATA #REQUIRED>
```

Ukázka implementace bloku kódu

```
<code>
  <local-variable name="A" type="integer">
    <get-state-var value="state1"/>
  </local-variable>
  <if>
    <test>
      <binary-expr op=">">
        <var-ref name="A"/>
        <literal-number value="10"/>
      </binary-expr>
    </test>
    <true-case>
      <code>
        <loop kind="for">
          <init>
            <local-variable name="i" type="integer"/>
          </init>
          <update>
            <unary-expr op="++" post="true">
              <var-ref name="i"/>
            </unary-expr>
          </update>
          <test>
            <binary-expr op=">">
              <var-ref name="i"/>
              <literal-number value="100"/>
            </binary-expr>
          </test>
          <code>
            <unary-expr op="++" post="true">
              <var-ref name="A"/>
            </unary-expr>
          </code>
      </code>
    </true-case>
  </if>
  <set-state-var name="state1">
    <var-ref name="A"/>
  </set-state-var>
</code>
```