VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DISERTAČNÍ PRÁCE

k získání akademického titulu Doktor (Ph.D.)

ve studijním programu (oboru)

INFORMAČNÍ TECHNOLOGIE

Ing. Radek Kočí

METODY A NÁSTROJE PRO IMPLEMENTACI OTEVŘENÝCH SIMULAČNÍCH SYSTÉMŮ

Školitelka: doc. Ing. Zdena Rábová, CSc.
Datum státní doktorské zkoušky: 24. června 2002
Datum odevzdání práce: 12. srpna 2004
Práce je k dispozici: Vědecké oddělení FIT, Božetěchova 2, 612 66 Brno

**METODY A NÁSTROJE PRO IMPLEMENTACI
OTEVŘENÝCH SIMULAČNÍCH SYSTÉMŮ**

Abstrakt

Modelování a simulace tvoří širokou oblast informačních technologií, která se věnuje zkoumání reálného světa prostřednictvím jeho abstrakcí a experimentů nad těmito abstrakcemi. Modelování obsahuje obecně dvě úrovně: modelování statické struktury systémů a modelování jejich chování. Simulaci, tj. experimentování s modelem, pak chápeme jako interpretaci chování modelu na základě popisu chování jeho struktur. Pro popis dynamických aspektů modelu lze použít různá paradigmata a formalismy. Tato práce vychází z Objektově orientovaných Petriho sítí (OOPN), které byly vyvinuty na Fakultě informačních technologií Vysokého učení technického v Brně. OOPN s výhodou kombinují přednosti Petriho sítí (formální základ, přirozený popis paralelismu) a objektové orientace (strukturalizace modelu, přirozená tvorba instancí podstruktur).

Moderní simulační systémy jsou stále komplikovanější a komplexnější. S tímto trendem roste i potřeba prostředků pro snadnou, příp. automatickou konfiguraci a adaptaci schopností systémů pro různé aplikační podmínky. Jako použitelné řešení se jeví koncept tzv. otevřených implementací, které zpřístupňují některé aspekty interní implementace systémů a tím umožňují jejich kontrolu. Základním prostředkem pro vyjádření otevřené implementace je zajištění obecného implementačního rámce (*framework*), který usnadňuje práci uživatelů, ať už v podobě navrhování či používání otevřených implementací. Jedním z takových rámců je metaúrovňová architektura a s ní spojený metaobjektový protokol (MOP). MOP zajišťuje řešení založené na objektové orientaci, které může být jednoduše integrováno do klasických vývojových procesů. MOP je velmi široce uplatňován v moderních operačních systémech a jazycích, kde nabízí elegantní a uniformní způsob programování založený na reflektivních konceptech.

Předkládaná práce vychází z myšlenky otevřené implementace simulačního systému konvergujícího do zjednodušené formy operačního systému, který slouží pro modelování, simulaci a prototypování složitých problémů. Výhody otevřené implementace se projeví především ve velké flexibilitě systému umožňující inkrementální vývoj modelů, jednoduchou aplikovatelnost speciálních simulačních technik či možnost adaptace prostředí vzhledem k měnícím se požadavkům, a to včetně možnosti měnit či kombinovat různé vyjadřovací prostředky (OOPN, Smalltalk apod.)

Poděkování

Rád bych poděkoval své školitelce Zdeně Rábové za její podporu při řešení disertační práce a také členům Ústavu inteligentních systémů, zejména Vladimíru Janouškovi za praktické rady a konzultace při řešení mé práce, dále Petru Peringerovi, Martinu Hrubému a Tomáši Vojnarovi za jejich podněty a rady.

Výzkum spojený s disertační prací byl podporován těmito granty: Modelování, verifikace a prototypování distribuovaných aplikací s využitím Petriho sítí (GA102/00/1017), Prostředí pro vývoj, modelování a aplikaci heterogenních systémů (GA102/01/1485), Systém PNtalk pro prototypování aplikací (FR1959/2002/G1) a Automatizované metody a nástroje pro vývoj spolehlivých paralelních a distribuovaných systémů (GA102/04/0780).

Vědomosti bez přemýšlení jsou zbytečné,
myšlení bez poznatků je prázdné.

Robert J. Sternberg

Obsah

1 Úvod **1**
- 1.1 Motivace 1
- 1.2 Objektově orientované Petriho sítě 1
- 1.3 Otevřené implementace 2
- 1.4 Související práce 3
- 1.5 Cíle disertační práce 4
- 1.6 Struktura práce 4

I Koncepce otevřeného simulačního systému **5**

2 Objektově orientované přístupy **6**
- 2.1 Objektová orientace 6
  - 2.1.1 Revoluce a evoluce v počítačových vědách 6
  - 2.1.2 Základy objektové orientace 6
  - 2.1.3 Pojmy objektové orientace 7
- 2.2 Přehled OO jazyků 8
  - 2.2.1 Historie objektově orientovaných jazyků 8
  - 2.2.2 Smalltalk 9
  - 2.2.3 Simula 10
  - 2.2.4 CLOS 10
  - 2.2.5 Self 10
  - 2.2.6 C++ 10
  - 2.2.7 Java 11
- 2.3 Koncept objektově orientovaného modelování a prototypování 11
  - 2.3.1 Evoluční cyklus modelování a prototypování 11
  - 2.3.2 Modelování a prototypování v simulačním systému 12

3 Koncepce otevřených systémů **13**
- 3.1 Evoluční proces objektu 13
  - 3.1.1 Chování objektů 13
  - 3.1.2 Aktivita objektu 14
- 3.2 Klasifikace modelů 15
- 3.3 Otevřené systémy 15
  - 3.3.1 Koncepce otevřených implementací 16
  - 3.3.2 Metaúrovňové architektury 16
- 3.4 Reflektivita 17
  - 3.4.1 Výpočetní a strukturální reflektivita 18

3.4.2 Časová klasifikace reflektivity . . . 19
3.4.3 Typová klasifikace reflektivity . . . 19
3.5 Metaobjektový protokol . . . 19
3.5.1 Explicitní metaobjektový protokol . . . 19
3.5.2 Implicitní metaobjektový protokol . . . 20
3.5.3 Interní metaobjektový protokol . . . 20
3.6 Systémy s metaúrovňovou architekturou . . . 20
3.6.1 OpenC++ v1 . . . 21
3.6.2 CodA . . . 21
3.6.3 Apertos . . . 21
3.6.4 AcTalk . . . 22

4 Petriho sítě jako prostředek pro modelování, simulaci a prototypování 23
4.1 Objektově orientované přístupy k Petriho sítím . . . 23
4.1.1 Renew . . . 23
4.1.2 Petri Net Kernel . . . 23
4.1.3 Objektové Petriho sítě . . . 24
4.1.4 Kooperativní sítě . . . 24
4.2 Objektově orientované Petriho sítě . . . 24
4.2.1 Struktura OOPN . . . 24
4.2.2 Systém sítí . . . 25
4.2.3 Metody . . . 25
4.2.4 Predikáty a synchronní porty . . . 25
4.2.5 Systém tříd . . . 26
4.2.6 Dědičnost . . . 26
4.3 Evoluce objektů popsaných OOPN . . . 26
4.4 Interpret OOPN . . . 27
4.4.1 Proces objektu . . . 27
4.4.2 Reprezentace OOPN . . . 28
4.4.3 Testování přechodů . . . 29
4.5 Modelování a prototypování v OOPN . . . 30
4.5.1 Systém PNtalk . . . 30
4.5.2 Modelování v systému PNtalk . . . 30
4.5.3 Objektově orientovaný návrh a prototypování . . . 32

II Architektura otevřeného simulačního systému 34

5 Architektura systému 35
5.1 Neformální definice . . . 35
5.2 Úrovně systému . . . 36
5.2.1 Klasifikace úrovní systému Smalltalk . . . 36
5.2.2 Klasifikace úrovní simulačního systému . . . 37
5.2.3 Hierarchie metaobjektů doménových objektů . . . 38
5.2.4 Hierarchie metaobjektů doménových tříd . . . 40
5.3 Reprezentace doménového objektu . . . 40
5.3.1 Vlákno . . . 40
5.3.2 Systémový proces . . . 41
5.3.3 Procesní vazba . . . 42

5.3.4 Synchronizace zpráv . . . 43
5.3.5 Struktura doménového metaobjektu . . . 44
5.4 Zpracování asynchronní zprávy . . . 45
5.4.1 Zprávy atomické v čase . . . 45
5.4.2 Fáze zpracování zprávy . . . 45
5.4.3 Příjem zprávy . . . 46
5.4.4 Aktivace kontextu metody . . . 47
5.4.5 Zpracování kontextu metody . . . 47
5.4.6 Zasílání zpráv z kontextu metody . . . 48
5.4.7 Ukončení kontextu metody . . . 49
5.5 Evoluce systému . . . 50
5.5.1 Události v OOPN . . . 50
5.5.2 Struktura světa . . . 51
5.5.3 Řízení objektů světa . . . 52
5.5.4 Řízení událostí objektu . . . 53
5.5.5 Čas v systému . . . 55
5.6 Transparentnost komunikace . . . 56
5.6.1 Interoperabilita . . . 56
5.6.2 Vytvoření proxyobjektu . . . 57
5.6.3 Komunikace objektů a proxyobjektů . . . 57

6 Význam otevřené architektury **61**
6.1 Kombinace paradigmat v modelování . . . 61
6.1.1 Smalltalk a OOPN . . . 61
6.1.2 Řízení spojených světů . . . 62
6.1.3 Změna doménového paradigmatu . . . 64
6.2 Systém DEVS . . . 64
6.2.1 Atomický DEVS . . . 64
6.2.2 Spojovaný DEVS . . . 65
6.2.3 DEVS simulátory . . . 66
6.2.4 Kombinace simulačního systému a DEVS . . . 68
6.3 Reprezentace stavu objektů . . . 69
6.3.1 Identifikace metaobjektů . . . 69
6.3.2 Stav místa . . . 70
6.3.3 Stav vlákna . . . 71
6.3.4 Stav procesu . . . 72
6.3.5 Stav objektu . . . 73
6.3.6 Stav světa . . . 73
6.3.7 Stav objektů Smalltalku . . . 74
6.4 Speciální techniky pro simulaci a prototypování . . . 74
6.4.1 Vnořená simulace . . . 74
6.4.2 Migrace objektů . . . 77
6.4.3 Prototypování . . . 78

7 Závěr **79**

A Syntax jazyka PNtalk **86**

B Základní syntaxe a sémantika jazyka Smalltalk **89**

C Objektově orientované modelování s OOPN **91**

D Metaprotokol **94**
D.1 Společný metaprotokol 94
D.2 Metaprotokol systémových metaobjektů 94
D.3 Metaprotokol komponent metaobjektu PNObject 96
D.4 Metaprotokol proxy metaobjektů 97

Seznam obrázků

2.1 Strom vývoje objektově orientovaných jazyků . . . 9

3.1 Kategorie metaobjektového protokolu . . . 20

4.1 Popis prvků OOPN . . . 25
4.2 Vztah mezi třídou, vzorem a instancí . . . 28
4.3 Reprezentace OOPN . . . 29
4.4 Ukázka OOPN metody . . . 31

5.1 Vztah tříd a metaobjektů ve Smalltalku . . . 36
5.2 Vztah objektů a tříd ve Smalltalku . . . 37
5.3 Vztah mezi úrovněmi Smalltalku a simulačního systému . . . 38
5.4 Základní pohled na hierarchii dynamické části systému . . . 39
5.5 Základní pohled na hierarchii statické části systému . . . 40
5.6 Procesy a vlákna v OOPN . . . 41
5.7 Vztah Smalltalku a úrovní simulačního systému . . . 58

6.1 Hierarchie simulátorů DEVS . . . 67

A.1 Příklad metody OOPN . . . 88

C.1 Schéma dědičnosti v OOPN . . . 91
C.2 Diagram aktivit pro vyhledávání metod v OOPN . . . 92
C.3 Model vyhledávání metod v kořenové třídě PN . . . 92
C.4 Model vyhledávání metod v třídě OOPN . . . 92

Kapitola 1

Úvod

1.1 Motivace

Modelování a simulace systémů tvoří širokou oblast informačních technologií, která se věnuje zkoumání reálného světa prostřednictvím jeho abstrakcí a experimentů nad těmito abstrakcemi. Modelování obsahuje obecně dvě úrovně: modelování statické struktury systémů a modelování jejich dynamických aspektů, tj. chování modelovaných systémů. Simulaci pak chápeme jako experimentování s modelem systému, tj. interpretaci chování modelu na základě popisu chování jeho struktur. Pro popis dynamických aspektů modelu lze použít různá paradigmata a formalismy.

Od poloviny 90. let se na Fakultě informačních technologií Vysokého učení technického v Brně vyvíjí systém PNtalk a s ním spojené paradigma Objektově orientovaných Petriho sítí (OOPN). PNtalk byl od počátku koncipován jako systém pro modelování, simulaci a prototypování složitých paralelních systémů prostřednictvím OOPN, které kombinují výhody objektové orientace a Petriho sítí. Základní formalismus paradigmatu OOPN je tvořen vysokoúrovňovou variantou Petriho sítí, které umožňují přirozeným způsobem popsat tok řízení modelu a paralelismus. Objektová orientace umožňuje především strukturovanost modelu. OOPN jsou založeny na stejném chápání objektu jako objektově orientovaný jazyk Smalltalk [GR89] s jedním podstatným rozdílem – procedura (tj. metoda objektu) není popsána jako sekvence příkazů, ale prostřednictvím vysokoúrovňových Petriho sítí. Metodu pak chápeme jako vzor struktury. Při vyvolání metody dojde k vytvoření instance příslušné sítě (vzoru), tzn. že se vytvoří její kopie, nad kterou pak probíhá evoluce dané sítě.

Nad těmito strukturami pak můžeme uvažovat o jejich dynamických změnách, ať vzoru či kopie vzoru. Změnou vzoru se změní struktura a chování jeho nových kopií. Změnou kopie vzoru můžeme dosáhnout změn sémantiky vyvolané metody za jejího běhu. Změna struktury se nemusí týkat pouze modifikací Petriho sítí – zajímavým případem změny struktury je možnost nahrazení paradigmatu např. jinou variantou Petriho sítí či sekvenčním výpočtem ve Smalltalku.

Problémy, před kterými současný vývoj systému PNtalk stojí, jsou v zásadě dva (do jisté míry spolu svázané). Prvním je otázka realizace systému, tedy navržení architektury, která by umožňovala výše prezentované dynamické změny struktur. Druhý problém je metodologie systému PNtalk a s ní spojená potřeba vytvářet případové studie problémů a jejich řešení s využitím specifických vlastností systému PNtalk. V této práci se budeme věnovat především prvnímu problému a v závěru se zamyslíme nad možnostmi prezentované architektury.

1.2 Objektově orientované Petriho sítě

Objektově orientované Petriho sítě (OOPN) představují splynutí dvou paradigmat – objektové orientace a Petriho sítí. Tyto přístupy mají několik svých výhod a nevýhod.

Objektová orientace. Pokud při řešení (modelování) problému použijeme objektově orientovaný přístup, znamená to především nalezení samostatných entit v řešeném problému, které mají nějaký stav a poskytují služby. Jedné takové entitě pak říkáme objekt. Každý objekt má definovanou množinu atributů, stav objektu je určen stavem těchto atributů. Objekt poskytuje služby prostřednictvím metod, což je pojmenovaná sekvence příkazů, která může mít vstupní parametry. Objekty spolu komunikují zasíláním zpráv, na které reagují vyvoláním a vykonáním příslušné metody. Provádění objektově orientovaného programu je definováno sekvencí zasílaných zpráv mezi objekty. Jedné sekvenci říkáme proces. Program může běžet ve více sekvencích současně (paralelní běh) – v tomto případě se procesy mohou vzájemně synchronizovat (pokud je to nutné) pomocí různých synchronizačních nástrojů.

Petriho sítě. Petriho sítě vznikly jako nástroj pro modelování paralelních procesů. Jednoduchým mechanizmem dokáží popsat sekvenci procesu a synchronizaci mezi procesy. Evoluce Petriho sítí není dána sekvencí příkazů v procesu, ale událostmi.

Objektově orientované Petriho sítě. OOPN kombinují dva odlišné přístupy k běhu simulace – procesní (objektová orientovanost) a událostní (Petriho sítě). Modelovací jednotkou je objekt, stejně jako v objektově orientovaných systémech. Jeho chování a atributy jsou definovány třídou. Metoda objektu (třídy) není popsána sekvencí příkazů, ale Petriho sítí. Objekty spolu komunikují zasíláním zpráv, na přijímanou zprávu objekt reaguje vytvořením příslušné metody.

Model lze chápat jednak jako abstrakci systému, nad kterou lze provádět simulaci, a současně jako součást návrhového postupu při vývoji nových systémů. Jednou z fází při návrhu a vytváření modelovaného systému je vytvoření prototypu. Prototyp je zjednodušená verze vyvíjené aplikace, obvykle realizovaná v prostředí určeném pro rychlou implementaci (např. Smalltalk). Vytvoření prototypu je důležitou součástí návrhu, protože prototyp lze rychle vytvořit, dá se s ním pracovat jako s výslednou aplikací, a v této fázi často dochází ke změně specifikace a požadavků na systém. Tyto změny lze jednoduchou cestou uplatnit přímo v modelu a po získání požadovaných výsledků simulace pak v prototypu. Pro vývoj aplikace by bylo výhodnější, kdyby se tyto dva kroky (modelování a prototypování) spojily, kdyby vzniklý model mohl, s drobnými úpravami, přímo sloužit jako funkční prototyp aplikace. Jedním z cílů projektu PNtalk je spojit výhody rychlé tvorby prototypů, jaké nabízí systém Smalltalk, a paradigmatu založeného na formálním popisu modelu.

1.3 Otevřené implementace

Pro architekturu systému s dynamickými změnami struktur a chování se jako nejlepší přístup jeví aplikace principů otevřených implementací, zejména pak principů reflektivních a metaúrovňových architektur. Přirozeným prostředím pro aplikaci těchto principů jsou objektově orientovaná prostředí, protože ta s sebou nesou jasně definovanou strukturu a jasně definované hranice mezi těmito strukturami. Struktury jsou zapouzdřeny v objektech, které mohou ovlivňovat struktury jiných objektů pouze prostřednictvím daného a jasného rozhraní. Rozhraní bývá někdy označováno jako protokol (termín protokol zavedl jazyk a systém Smalltalk [GR89]).

Základní ideou otevřené implementace je zpřístupnění vnitřních struktur objektů, abychom mohli systém modifikovat a rozšiřovat. Klasickým případem otevřené implementace jsou metaúrovňové architektury. Můžeme si je představit tak, že ke každému klasickému objektu existuje nějaký jiný (meta) objekt, který poskytuje protokol pro inspekci a změnu vybraných aspektů klasického objektu.[1] Obecně v metaúrovňových architekturách existují různé úrovně, kde vždy jedna úroveň je řízena jinou úrovní.

[1] Takto pojatému klasickému objektu budeme nadále říkat *doménový objekt*. S tímto objektem je spojeno *doménové paradigma*, které je použito pro jeho definici (např. Smalltalk nebo OOPN).

Principy otevřených implementací umožňují nejen ovlivňovat vlastní struktury objektů, ale i jejich výpočetní chování, tzn. způsob, jak reagují na zprávy, jaké další operace se vykonají v důsledku zaslání či přijetí zprávy apod. Použitelnost otevřených implementací závisí na úrovni jejich implementace a na paradigmatu, který je použit při implementaci. Architektura simulačního systému je založena na myšlence maximální úrovně otevřenosti. Výhody takto koncipovaného systému se projeví především ve velké flexibilitě umožňující adaptovat systém podle požadovaných potřeb, a to včetně možnosti měnit či kombinovat různé vyjadřovací prostředky.

Na tomto místě může mnohým vytanout na mysli otázka, jaká je reálná uplatnitelnost systému s tak velkým stupněm otevřenosti, navíc pracujícím s paradigmatem OOPN. Cílem výzkumu OOPN není pouze vytvořit nástroj pro modelování a simulaci, ale nástroj umožňující vyvíjený model začlenit do reálného prostředí. Takto pojatý model pak může sloužit jako část prototypu či přímo aplikace. S využitím vlastností dynamických struktur pak lze tento systém chápat jako platformu pro tvorbu modelů a prototypů aplikací s možností vyvíjet vlastní paradigmata a jednoduchým způsobem je začleňovat do systému. Pak můžeme na takto pojatý systém pohlížet jako na zjednodušenou formu (ve smyslu bezpečnosti) operačního systému. Smyslem celého projektu PNtalk je ověřit diskutované vlastnosti otevřených systémů v praxi a jejich uplatnění v oblasti modelování a prototypování agentů, umělé inteligence či návrhu komplexních systémů.

1.4 Související práce

Problematikou modelování, objektově orientovaných Petriho sítí a reflektivních architektur se zabývá mnoho projektů a prací ve světě. Následující výčet lze chápat jako inspirativní zdroje, které jsem využil ve své práci. V oblasti modelování můžeme najít různé varianty metod a přístupů, např. z aktivit na Fakultě informačních technologií Vysokého učení technického v Brně můžeme zmínit techniky a nástroje pro tvorbu multimodelů [HK03, HKPR02], simulační jazyk HELEF [Hru04] nebo simulační knihovnu SIMLIB [Per96] pro jazyk C++. V oblasti objektově orientovaných Petriho sítí můžeme najít shrnutí různých přístupů, variant a použití v [EJN97, ZH]. Myšlenka spojení objektové orientace a Petriho sítí vznikla v první polovině devadesátých let. Jednotlivé přístupy k vytvoření různých variant objektových Petriho sítí si můžeme uvést podle jejich představitelů: dr. Charles A. Lakos [Lak95a], Christophe Sibertin-Blanc [SB94], dr. Daniel Moldt [Mol95] nebo profesor Rüdiger Valk [Val98]. Objektově orientované Petriho sítě společně se systémem PNtalk byly poprvé představeny v polovině devadesátých let 20. století [ČJV97, Jan95, Jan98]. Další aktivní projekty v současné době jsou např. Renew [NF98, Ren04] nebo Petri Net Kernel [KW01].

Myšlenka objektově orientované výpočetní reflektivity[2] (*computational reflection*), včetně strukturálních změn a změn chování během evoluce, byla představena v sedmdesátých letech [GR89], ale kořeny těchto konceptů můžeme vysledovat mnohem dříve (jazyk Lisp, Univerzální Turingův stroj apod.) V současné době se problematikou reflektivních systémů zabývají profesor Gregor Kiczales [KLM$^+$97] (aspektově orientované programování) a dr. Pattie Maes [Mae87] (umělá inteligence), projekty Actalk [Bri99] (souběžná práce ve Smalltalku prostřednictvím reflektivity, představení aktorů), Apertos [Yok92a] a TUNES [Tun04] (vývoj vysoce flexibilních operačních systémů). V oblasti modelování a simulace můžeme zmínit přístupy k zavedení reflektivity (umožňující strukturální změny modelů) do systému DEVS [BZP00], a to např. Fernando J. Barros [Bar97] nebo profesor Uhrmacher [Uhr01]. Ve světě Petriho sítí můžeme nalézt aktivity související s kombinací objektové orientace, Petriho sítí a reflektivních vlastností např. u dr. Lakose [Lak96], který zdůrazňuje výhody takového přístupu v čisté, flexibilní a efektivní implementaci a také v možnosti představit alternativní schémata strategií plánování apod.

[2] Někdy bývá ve spojitosti s reflektivními systémy použit pojem *reflexe*. Podle mého mínění je český termín *reflektivita* korektní ekvivalent anglického termínu *reflection*. Reflexe pak vyjadřuje jednu z vlastností reflektivních systémů, tj. schopnost odrážet (reflektovat) vybrané aspekty objektů (či jiných entit).

Prezentovaná disertační práce se zabývá reflektivními přístupy k Objektově orientovaným Petriho sítím (OOPN). Oproti jiným přístupům představuje obecnější metaúrovňovou architekturu, která se neomezuje pouze na formalismus OOPN.

1.5 Cíle disertační práce

Disertační práce vychází z konceptu systému PNtalk pro modelování a simulaci prostřednictvím doménového paradigmatu Objektově orientovaných Petriho sítí (OOPN). Jejím cílem je rozšířit výzkum OOPN a vytvořit nástroj umožňující začlenit vyvíjený model do reálného prostředí, tj. nástroj pro prototypování. Práce se věnuje přístupům k vyjádření dynamických aspektů modelování, simulace a prototypování. Jako základní přístup je diskutován princip otevřených implementací, který je aplikován na architekturu otevřeného simulačního systému. Otevřený simulační systém umožňuje uvažovat dynamické změny modelů a dynamické manipulace s modely, včetně spojení modelů popsaných OOPN s jinými paradigmaty. Cíle disertační práce můžeme shrnout do tří bodů:

1. *Návrh nové koncepce otevřené architektury simulačního systému.* Architektura vychází z principů otevřených implementací a objektové orientace. Koncepce návrhu simulačního systému má umožnit spojovat různá paradigmata, provádět dynamické manipulace s modely i simulačním systémem a provádět různé druhy simulačních experimentů.

2. *Zobecnění přístupu k reprezentaci doménových paradigmat.* Otevřený simulační systém má umožnit kombinaci různých paradigmat na úrovni objektů i metod a jejich přizpůsobení měnícím se požadavkům modelování. Přístup k popisu doménových paradigmat vychází z vlastností popisu a evoluce OOPN.

3. *Návrh metodiky a zhodnocení aplikovatelnosti navrženého otevřeného simulačního systému.* Výhody otevřeného simulačního systému se projeví především v jeho velké flexibilitě umožňující adaptovat systém podle požadovaných potřeb, kombinovat různé vyjadřovací prostředky pro specifikaci modelů a vytvářet prototypy reálných aplikací.

1.6 Struktura práce

Práce je složena ze dvou částí a sedmi kapitol, které se postupně věnují jednotlivým oblastem řešeného problému. První část uvádí čtenáře do světa objektové orientace, simulačních nástrojů a jazyků a otevřených implementací (tj. reflektivních a metaúrovňových systémů). Druhá část popisuje návrh otevřeného simulačního systému a jeho možnosti.

Po úvodní kapitole následuje kapitola druhá, která přináší základní shrnutí objektově orientovaného přístupu k modelování a tvorbě aplikací. Diskutuje současné objektově orientované nástroje, systémy a jazyky. V závěru uvádí do spojitosti pojmy objektová orientace, modelování, simulace a prototypování. Třetí kapitola rozebírá pojem otevřená implementace. Vysvětluje její význam a použití, definuje základní pojmy a koncepci otevřených implementací. Čtvrtá kapitola si všímá různých spojení objektové orientace a Petriho sítí a provádí diskuzi jejich výhod a nevýhod. Dále popisuje Objektově orientované Petriho sítě (OOPN), které tvoří základní formalismus prezentovaného simulačního systému. Věnuje se především rozboru základní struktury a možností, které OOPN nabízejí.

Kapitola pátá, zahrnutá již do druhé části práce, definuje architekturu otevřeného simulačního systému, který využívá prostředků popsaných v první části – tj. otevřené implementace a formalismu OOPN. Šestá kapitola diskutuje možnosti této architektury, její možná rozšíření a použitelnost. V závěru práce provedeme diskuzi navržených principů otevřené architektury simulačního systému, výhody a nevýhody těchto principů a další směřování vývoje v této oblasti.

Část I

Koncepce otevřeného simulačního systému

Kapitola 2

Objektově orientované přístupy

V této kapitole se budeme věnovat objektové orientaci a objektově orientovaným přístupům k modelování, prototypování a simulaci. Vysvětlíme si základní pojmy objektové orientace, souvislost objektové orientace s modelováním a prototypováním systémů a uvedeme si přehled existujících objektově orientovaných jazyků a systémů. V poslední části kapitoly rozvedeme objektově orientované přístupy k modelování a prototypování.

2.1 Objektová orientace

2.1.1 Revoluce a evoluce v počítačových vědách

Téměř všechna vědní snažení obsahují ve své historii periody evoluce následované periodami revoluce. V jistém okamžiku je prezentována nová revoluční teorie, kterou obvykle zastává malá skupina lidí. Během času se tato nová teorie ukáže být buď špatná, nebo akceptovatelná (i když nemusí být nutně pravdivá). Pokud je akceptovatelná, přejde revoluční myšlenka do své evoluční fáze. Během této periody dochází k vylepšování, kultivování teorie úspěšnými experimenty, objevy atd.

Počítačová věda a programování můžeme považovat za příklad revolučně–evolučního cyklu [Hun97]. Přesun od nízkoúrovňového k vysokoúrovňovému programování byla jedna taková revoluce. Objektová orientace je další ukázkou revoluce v programování a přístupu k přenosu konceptů reálného světa do světa vymezeného počítačovou vědou. Během let se objektová orientace stala široce akceptovanou, i když stále existují lidé, kteří se ptají – k čemu objektová orientace? Přesto můžeme říci, že objektová orientace přečkala periodu revoluce v počítačové vědě a přešla do fáze evoluce (zpřesňování a vývoj), a to především v oblasti metodologie návrhu, modelování a tvorby softwarových produktů. Objektová orientace se stala významnou silou v modelování a tvorbě softwaru.

2.1.2 Základy objektové orientace

Existuje mnoho objektově orientovaných programovacích nástrojů (např. Smalltalk, Simula, C++, Object-Pascal nebo Java – viz kapitola 2.2) a technik pro objektově orientované modelování, z nichž nejznámější je UML [UML04]. Přestože mnoho uživatelů tyto nástroje a technologie využívá, výsledný efekt není takový, jak jej objektová technologie prezentovala a jaké objektová technologie skutečně nabízí. Za klíčový důvod těchto neúspěchů lze považovat nepochopení základního významu objektové technologie a odpoutání se od vžitých představ tvorby výpočetních modelů. V tomto pojetí je objekt většinou chápán pouze jako jakási struktura, která umožňuje částečně modularizovat vyvíjenou aplikaci. Princip a filozofie objektově orientované technologie je však radikálně odlišný.

Základní problém přístupu k tvorbě aplikací (či modelování obecně) by se dal charakterizovat větou: Až dosud se vývojáři snažili sladit své vyjadřovací prostředky s požadavky strojů místo opačného přístupu

– přiblížit vyjadřovací prostředky strojů myšlení a chápání světa člověkem. Objektová technologie otevírá jednu z cest, jak tohoto cíle dosáhnout. Velmi důležitou podmínkou správného použití objektových přístupů je nejen zvládnutí vyjadřovacích prostředků, ale především metod jejich použití a nasazení na řešení konkrétních problémů. Objektově orientované přístupy lze považovat za soubor objektově orientovaných prostředků (třídy, objekty atd.), jejich vzájemných vazeb a metodologie. Velmi podstatné je uvědomit si (tak jako i v jiných technologiích), že objektově orientované přístupy nejsou *cílem* našeho snažení, ale *prostředkem* k dosahování cílů.

Základním principem objektově orientovaného návrhu je myslet v reálných objektech (tj. uvědomit si *jaké* objekty máme, *jak* se chovají, *co* je třeba o nich vědět), nikoli v datových strukturách a procedurách. Na reálný svět můžeme nahlížet jako na množinu objektů, které reagují na vnější podněty, a některé z nich mohou vyvíjet i vlastní aktivity. Objekt vytvořený v objektově orientovaném programovacím prostředí je pak abstrakcí reálného objektu a modelování nějaké části reálného světa je vytváření abstrakcí nad objekty dané části reality. Tady je vhodné uvést malou poznámku – dosavadní praxe je taková, že se odděluje modelování (tj. nalézání tzv. konceptuálních objektů v řešené problematice a jejich vazeb) od programování, vytváření tzv. softwarových objektů, které korespondují s konceptuálními objekty jen zčásti. Rozlišuje se tedy hranice mezi modelováním (fáze návrhu) a programováním (fáze implementace). V současnosti existují snahy o přiblížení těchto dvou fází tvorby systémů či aplikací. Vztah této problematiky k prezentovanému systému bude diskutován v kapitole 4.5.3. Podrobnější diskuzi této problematiky lze také najít v [Koč01c].

2.1.3 Pojmy objektové orientace

Existuje několik základních pojmů objektové orientace a současně existuje několik variant jejich výkladu. Pro zamezení možných nedorozumění si nyní tyto pojmy uvedeme a definujeme jejich význam tak, jak budou chápány v dalším textu. Nejprve esenciální pojmy (v závorce je vždy uveden používaný anglický termín):

Objekt (Object). Na objekt můžeme pohlížet jako na černou skříňku (*black-box*), jejíž vnitřní realizace je nám skryta. Objekt má svůj stav, chování a identitu. Stav objektu je množina podmínek, ve kterých může objekt existovat. Změny stavu reflektují chování objektu. Chování objektu definuje, jak objekt reaguje na vnější či vnitřní podněty. Chování je popsáno množinou operací.

Třída (Class). Třída je generická definice pro množinu podobných objektů. Třída zajišťuje specifikace atributů a chování objektů, jde tedy o abstrakci nad objekty. Objekt je pak instance příslušné třídy.

Komunikace objektů (Message passing). Objekty mezi sebou (případně objekt sám se sebou) komunikují prostřednictvím zasílání zpráv. Zpráva je příjemcem (*receiver*) chápána jako požadavek na provedení indikovaného chování (metody) a návrat výsledku akce volajícímu objektu (*sender*).

Základní charakteristika objektové orientace je uváděna v následujících čtyřech pojmech: *abstrakce*, *zapouzdření*, *dědičnost* a *polymorfismus* [Hun97]. Podle mého názoru je mnohem lepší místo o dědičnosti mluvit o *hierarchii*, tj. hierarchickém vztahu objektů. Hierarchie se pak dá realizovat prostřednictvím *dědičnosti* nebo *skládání* objektů, přičemž je nutno podotknout, že skládání objektů je podporováno pouze některými dynamickými jazyky.

Abstrakce (Abstraction). Vytvářený systém objektů a tříd je abstrakcí řešeného problému. Jinými slovy z doménové oblasti řešeného problému vybereme pouze ty relevantní informace, na jejichž základě pak vytvoříme příslušné struktury a definujeme operace nad nimi a vztahy mezi nimi. Ostatní informace zanedbáme, *abstrahujeme*.

Zapouzdření (Encapsulation). Hlavní vlastností zapouzdření je ukrytí implementačních detailů objektu. Jedinou možností, jak může okolí objektu modifikovat jeho stav nebo přistupovat k interním informacím, je prostřednictvím operací, resp. zasíláním zpráv.

Polymorfismus (Polymorphism). Polymorfismus lze přeložit jako mnohotvarost, schopnost výskytu v mnoha formách. V objektově orientované technologii polymorfismus znamená výskyt různých typů chování na základě jedné zprávy. Jinak řečeno, my můžeme objektům zasílat stejné zprávy, aniž známe jejich typ (tj. třídu), a zpráva bude zpracována příslušným způsobem právě podle typu objektu.

Dědičnost (Inheritance). Dědičnost je mechanismus vytváření nových objektů ze starých mírnou modifikací jejich chování a atributů. Objekt odvozený z jiného objektu dědí (sdílí) jeho chování a množinu stavů, které může modifikovat. Dědičnost lze chápat jako specializaci na úrovni objektů, tj. z obecnějšího objektu generujeme objekt konkrétnější. Dědičnost je většinou realizována jako vztah mezi třídami, nikoliv přímo mezi objekty (tzn. že existuje hierarchie tříd s relací dědičnosti, ze kterých pak vznikají jejich instance – vlastní objekty).

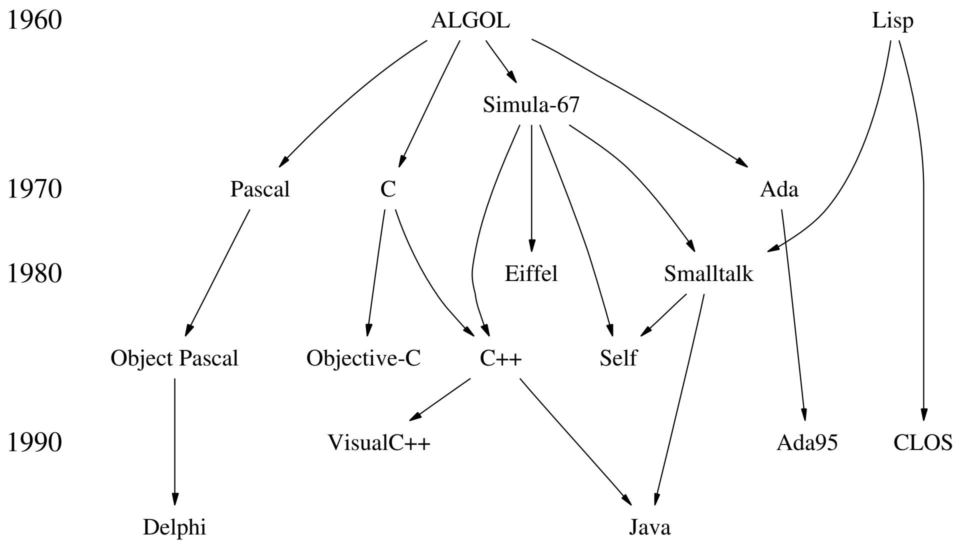
Skládání (Composition). Dalším podstatným rysem objektově-orientovené technologie je skládání. Skládání objektů znamená, že vnitřními daty objektů jsou opět nějaké jiné objekty. Objekt, který je složen z jiných objektů, pak může využívat funkčnost složených objektů, přičemž tato funkčnost je vlivem zapouzdření ukrytá z pohledu okolí objektu. V takovém případě mluvíme o delegaci funkčnosti (tj. složený objekt deleguje některé zprávy na některý ze svých podobjektů).

2.2 Přehled OO jazyků

2.2.1 Historie objektově orientovaných jazyků

Ve světě programovacích jazyků nemůžeme jednoznačně definovat rodokmen a z něj vyplývající vlastnosti jazyka. Nemůžeme říci, jak dobrý je nějaký jazyk na základě jeho předchůdců. Přesto současné objektově orientované jazyky byly ovlivňovány a inspirovány předchozími jazyky. Na základě těchto vlivů si můžeme definovat vztah mezi různými jazyky (viz obrázek 2.1 – obrázek se věnuje především objektově orientovaným jazykům a jejich vztahu [Hun97]). Na obrázku můžeme vidět dva jazyky, které představují dva odlišné přístupy k programování – funkcionální (Lisp) a imperativní (Algol). Tyto jazyky si můžeme představit jako kořeny vývoje v uvedených přístupech k programování.

Imperativní přístup můžeme rozdělit do čtyř větví (z hlediska objektově orientovaných jazyků) – větve ovlivněné jazyky Simula-67, C, Pascal a Ada. Objektově orientované jazyky, které přebírají principy těchto jazyků, jsou Eiffel, C++, Objective-C, Object Pascal, Delphi a Ada95. Funkcionální přístup lze rozdělit do dvou větví – větev reprezentovaná jazykem CLOS a větev reprezentovaná jazykem Smalltalk. Tato druhá větev je však z pohledu čistého rozdělení na imperativní a fukcionální jazyky větví hybridní, neboť vychází z konceptů, které zavedl jazyk Simula-67 (objektová orientace) a současně jazyk Lisp (funkcionální přístup k objektům). Takto lze častečně chápat i v současné době velmi rozšířený jazyk Java, který je kombinací přístupů jazyků C++ a Smalltalk. Specifické postavení zaujímá jazyk Self, který můžeme chápat jako pokračování myšlenek stojících u zrodu jazyka Smalltalk. Z výše uvedených jazyků si nyní některé přiblížíme. Komplexnější přehled objektově orientovaných jazyků je uveden např. v [Phi95].



Obrázek 2.1: Strom vývoje objektově orientovaných jazyků

2.2.2 Smalltalk

Programovací jazyk a systém Smalltalk [GR89] patří v programátorské praxi mezi poměrně nové systémy,[1] ale stále zřetelněji se ukazuje, že ho nepostihne osud jiných jazyků a systémů, které na čas zazářily na softwarovém nebi a poté upadly v zapomnění. Smalltalk se postupně stal nástrojem především profesionálních programátorů a počítačových výzkumníků. Smalltalk je určen zejména pro výzkumné účely a pro rychlý návrh programů využívající grafické uživatelské rozhraní ovládané myší s okny, ikonami a menu. Je mimořádně vhodný pro tvorbu aplikací z oblasti umělé inteligence a z oblasti objektově orientovaných databázových systémů. Smalltalk není jen programovací jazyk, neboť kromě překladače vlastního jazyka je jeho nedílnou součástí vlastní grafické prostředí pro podporu tvorby a běhu programů, které tvoří vlastní kompaktní objektově orientovaný operační systém využívající bitmapovou grafiku, okna, menu a ovládání myší.

Smalltalk byl vyvíjen v průběhu 70. let 20. století v Palo Alto Research Center (PARC) kolektivem vědců vedených dr. Alanem Kayem. Na výzkumu se také podílel tým System Concepts Laboratory. Předmětem výzkumu byl projekt Dynabook pro vývoj osobního počítače budoucnosti s jednotným programovým a vývojovým prostředím, kde nebude rozdílu (např. ve vzhledu, funkčnosti a syntaxi příkazů) mezi operačním systémem a aplikačním programovacím jazykem.

Architektura stroje byla navržena jako tzv. Smalltalk Virtual Machine (SVM) implementovaná mikroprogramy na výkonném větším počítači. Na této architektuře pak bylo vytvořeno kompletní prostředí skládající se z operačního systému, překladače programovacího jazyka, uživatelského interface a vývojového i ladicího prostředí včetně počítačové sítě. Vše bylo jednotně naprogramováno pouze v jazyce Smalltalk.

První, pracovní a pokusná verze systému vznikla v roce 1972 a nazývala se Smalltalk-72. V navrhovaném jazyce systému se nejvíce odrazily prvky z prvního neprocedurálního funkcionálního jazyka LISP a prvního objektově orientovaného jazyka Simula. Z této verze jazyka se vyvinula po dalších úpravách vývojová verze jazyka Smalltalk-76 a po následných úpravách verze Smalltalk-80, kterou lze považovat za současný standard.

[1] Smalltalk byl dlouhá léta používán pouze malým počtem uživatelů, především z důvodů nutnosti výkonného stroje pro provoz virtuálního stroje a licenční politiky, avšak v dnešní době se stává stále častěji používaným nástrojem pro tvorbu aplikací.

2.2.3 Simula

Simula (Simula 67) [Poo87] je objektově orientovaný jazyk se systémovými třídami podporující diskrétní procesně orientovanou simulaci. Jazyk Simula můžeme považovat za první jazyk, který definoval objektově orientovaný přístup k tvorbě programů. Zavedl pojmy jako jsou objekt, třída, dědičnost atd. Základní myšlenka, kterou se jazyk Simula odlišuje od jiných objektově-orientovaných jazyků, spočívá v tom, že každý objekt konkrétní třídy není pouze statickou strukturou nesoucí data a metody, ale může v čase (tedy od vzniku objektu) vyvíjet aktivitu (projevovat chování popsané sekvencí příkazů) – objekt je tedy asociován s procesem a jako proces jej můžeme také chápat. Tato sekvence příkazů bývá nazývána životní proces (*life process*), někdy také životní kód (*life code*) třídy. Každý nový vytvořený objekt třídy začne bezprostředně po svém vytvoření provádět svá životní pravidla. Procesy mohou být synchronizovány (pozastaveny, uvolněny podle dané podmínky apod.) speciálními příkazy jazyka.

2.2.4 CLOS

CLOS (The Common Lisp Object System) [KdRB91] představuje silný a flexibilní typovaný systém. CLOS vychází z jazyka Common Lisp, do kterého zavádí prostředky pro tvorbu a manipulaci s třídami a objekty. CLOS zavádí pojmy jako jsou metaobjekt a metaobjektový protokol (viz kapitola 3.5). Pro každý typ elementu v programu existuje korespondující základní metaobjektová třída, např. pro třídu, definici slotu, generickou funkci, metodu atd. Velkou výhodou systému CLOS je velká dynamičnost pramenící z jeho funkcionální podstaty.

2.2.5 Self

Self [Sel04] je objektově orientovaný jazyk, určený pro programování zkoušením a ověřováním kódu (*exploratory programming*), založený na malém počtu jednoduchých myšlenek: prototypy, sloty a chování. Prototypy kombinují dědičnost a tvorbu instancí a zajišťují pracovní prostor, který je jednodušší a flexibilnější než v jiných objektově orientovaných jazycích. Sloty unifikují přístup k proměnným a procedurám (metodám) – obsahem slotu je vždy objekt, přičemž metody se chápou jako speciální objekt. Self neodděluje popis stavu od popisu chování. V jazyku Self neexistuje pojem program či počáteční stav aplikace, ale objekty (a tedy aplikace) jsou vyvíjeny inkrementálně za běhu celého systému.

Výchozí myšlenky Selfu jsou inspirovány jazykem Smalltalk (objektová orientace, virtuální stroj, kompaktní prostředí). Self byl primárně navržen Davidem Ungarem a Randallem Smithem v roce 1986 ve společnosti Xerox PARC. Poté, co se oba autoři přesunuli na Stanfordskou univerzitu (*Stanford University*), vyvinuli a představili v roce 1987 první překladač. První veřejně dostupná verze byla publikována v roce 1990. V současné době existuje verze 4.2.1, která je určena pro operační systémy Mac OS X a Solaris. Jazyk a systém Self je inspirativní zdroj pro mnoho objektově orientovaných jazyků. Představuje jeden z nejčistších přístupů k objektovému modelování a programování.

2.2.6 C++

Oproti jazykům typu CLOS, Smalltalk nebo Self představuje jazyk C++ [Str00] dvě zásadní odlišnosti: je statický a objektová orientace byla dodána dodatečně (nebyl tedy od počátku navržen jako objektově orientovaný, ale vycházel z jazyka C). Nevýhodu jazyka C++ lze spatřovat především v jeho statickém přístupu k programu. Další jeho vlastnost, silná statická typová kontrola, může přinášet problémy v oblasti aplikace principů otevřených implementací.

2.2.7 Java

Historie programovacího jazyka Java [Eck02] sahá do první poloviny devadesátých let 20. století. V roce 1991 byl představen jazyk OAK, který byl původně navržen jako programovací prostředí pro spotřební elektrotechniku. V průběhu let se z něj vyvinul obecný objektově orientovaný jazyk, který vyústil v prezentaci první verze v roce 1995. V současné době existuje ve verzi 1.5.0 (beta). Jazyk a systém Java je vyvíjen společností SUN Microsystems. Java přebírá některé myšlenky systému Smalltalk (virtuální stroj, garbage collecting, částečně dynamická reflektivita) a jazyka C++ (objektově orientovaný přístup, typová kontrola). Výborné vlastnosti Javy odrážejí skutečnost, že velký tým vývojářů pracuje na vývoji rozsáhlé knihovny tříd pro různá aplikační prostředí. Nevýhody Javy lze spatřovat ve stále velkém stupni statičnosti.

2.3 Koncept objektově orientovaného modelování a prototypování

Model lze obecně chápat jako abstrakci uvažovaného systému. Smyslem modelu je ověřit požadované vlastnosti modelovaného systému či porozumět uvažované problémové doméně. Pro ověřování vlastností existuje více přístupů, od formálních verifikací a validací až po simulace. Z toho také vyplývá různorodost používaných modelů; pro formální verifikaci se většinou používají formální modely vyšší úrovně abstrakce, pro simulaci simulační modely provádějící abstrakci času, zpoždění, zařízení v systému apod. Pokud chceme pouze porozumět řešené problémové doméně (tj. nechceme provádět verifikaci vlastností), můžeme použít některý z formalismů pro modelování systémů jako např. UML [UML04] nebo přímo manipulovat s modelem systému, který není tak abstraktní jako simulační model, ale zároveň není plnohodnotnou implementací problému – tedy prototypovat.

2.3.1 Evoluční cyklus modelování a prototypování

Prototypování [Kri96] můžeme chápat jakou součást životního cyklu vývoje aplikace. Prototypování bylo poprvé představeno a použito v osmdesátých letech, tedy ve stejné době, kdy se objektově orientované přístupy staly populární. Základní myšlenkou prototypování je vývoj funkčního modelu aplikace, aniž jsou předem známy veškeré požadované aspekty a vlastnosti modelovaného systému. Přesto je někdy výhodné mít k dispozici odlehčenou verzi modelu, která umožňuje provádět experimenty, testovat a validovat různá možná řešení daného problému a lépe rozhodovat, co je skutečně zapotřebí a jak toho přesně dosáhnout.

Evoluční cyklus modelování a prototypování si můžeme představit následovně. Na základě řešeného problému se vytvoří jeho model, který slouží jako výchozí aspekt pro prototypování. Tento model může být buď abstraktní nebo simulační. Simulačním modelem lze ověřovat požadované aspekty, upřesňovat požadavky na řešený problém, a to s využitím principů inkrementálního vývoje. Tento simulační model lze pak použít jako základ pro tvorbu prototypu. Abstraktní model slouží především pro porozumění řešené problematice a vyjádření vztahu mezi jednotlivými entitami modelu (viz např. již zmiňované UML). Pokud by však některé z těchto abstraktních modelů byly realizovány s použitím striktně definovaného formalismu, mohly by tyto modely sloužit jako součást prototypové realizace problému. Blíže se touto problematikou zabývá kapitola 4.5.3.

Klasický postup při práci se simulačním modelem je vytvoření modelu v některém simulačním jazyce, jeho překlad (do strojového kódu, mezikódu, případně vnitřní reprezentace simulačního nástroje) a poté jeho simulace, experimentování s modelem s využitím daného simulačního nástroje. Při jakékoliv změně modelu nebo jeho atributů je nutné celou tuto proceduru opakovat. Pro lepší manipulaci se simulačním modelem (především pro experimentování) by však bylo výhodnější mít možnost uskutečňování dynamických změn, tj. změn chování či struktury modelu za běhu simulace. Vhodným prostředím pro realizaci těchto požadavků jsou dynamické systémy založené na objektově orientovaných principech. Modely jsou většinou vytvářeny s použitím jednoho typu paradigmatu (např. Petriho sítě [Lak95a, SB94, Mol95, Val98, Jan95], SIMLIB [Per96], Simula [Poo87] apod.) Občas je však vhodné určité části systému modelovat pomocí různých paradigmat, která jsou pro vyjádření dílčího problému vhodnější.

2.3.2 Modelování a prototypování v simulačním systému

Prototypování ke svému zdárnému použití potřebuje prostředí pro vývoj, metody, techniky, formalismy a nástroje umožňující a usnadňující tvorbu prototypů. Jako adekvátní základ se jeví právě použití objektově orientovaných přístupů a dynamických prostředků, jaké představuje například Self a Smalltalk. Architektura systému prezentovaného v této práci je založena na konceptech, které nabízí Smalltalk. Preference Smalltalku před Selfem vychází z následujících úvah. Formalismem diskutovaným jako základní paradigma simulačního systému jsou Objektově orientované Petriho sítě (viz kapitola 4), které jsou svou podstatou splynutím Petriho sítí a jazyka Smalltalk. Dalším podstatným důvodem je i dostupnost systémů Smalltalk a Self. Virtuální stroj Selfu je implementován pro podstatně menší počet počítačových architektur a operačních systémů než virtuální stroj Smalltalku. Smalltalk je také v programátorské praxi více rozšířený.

Jak vyplývá z výše uvedeného textu, hranice mezi některými modely UML, simulačním modelem a prototypováním nemusí být zcela jednoznačné. Prezentovaná práce se věnuje modelování, simulaci a prototypování a především systému pro práci s dynamickými simulačními modely a prototypy. Jednou z klíčových myšlenek je spojit nebo alespoň maximálně zjednodušit vazbu mezi modelem a prototypem. Tedy myšlenka, že model může sloužit (po případných malých změnách) současně jako prototyp nebo (v některých případech) i jako součást výsledné aplikace.

Kapitola 3

Koncepce otevřených systémů

V této kapitole se budeme věnovat pojmu *otevřený systém*. V prezentované práci se pod tímto pojmem rozumí takový systém, který využívá vlastností *otevřených implementací*. Nejprve se podíváme detailněji na evoluční proces objektu a rozdělení objektů podle jejich významu. Na základě prezentovaného rozboru si pak přiblížíme pojem otevřené implementace systémů, a to zejména ze dvou hledisek: *co* tento pojem znamená a *proč* je výhodné se jím zabývat. S otevřenými implementacemi souvisejí pojmy *reflektivita*, *metaúrovňové architektury* a *metaobjektový protokol*, které budou také prezentovány.

3.1 Evoluční proces objektu

V této kapitole se budeme věnovat aktivitám objektu během jeho života, a to z pohledu způsobu, *jakým* provádí své aktivity. Chování objektu je popsáno metodami a vlastní činnost je dána instancemi těchto metod (jinými slovy vyvoláním metod). Evoluce objektu pak probíhá na základě přijímaných zpráv, které evokují příslušné metody. Aktivita objektu říká, zda je objekt pasivní součástí systému (tj. zda jeho evoluce závisí pouze na příjmu zpráv ze svého okolí), nebo zda je schopen evoluce bez nutnosti vnějších podnětů (tj. zda jeho evoluce není přímo závislá na vnějších zprávách). Klíčovým aspektem životního procesu objektu je způsob zpracování zpráv, a proto se zaměříme především na různé přístupy ke zpracování zpráv.

3.1.1 Chování objektů

Pod pojmem chování objektu budeme rozumět způsob, jakým objekty zpracovávají vnější podněty (tj. jakým způsobem přijímají a zpracovávají zprávy), příp. jakým způsobem generují vnitřní podněty (tj. zasílání zpráv, synchronizace apod.) Evoluce objektů, a tedy i systému, je pak dána reakcemi na tyto podněty. Můžeme rozlišit tři základní vlastnosti, které definují chování objektu:

- *Komunikace*. Komunikace mezi objekty probíhá na základě zasílání zpráv. Klíčovou entitou je zpráva a způsob jejího zpracování příjemcem zprávy. Zpracování může být synchronní nebo asynchronní, zprávy mohou mít definované priority apod. Důležitou vlastností komunikace je přenos přímého výsledku zprávy (návratová hodnota metody) volajícímu objektu.

- *Zpracování zprávy*. Objekt může zprávu zpracovat implicitně (např. Smalltalk [GR89]) nebo explicitně (např. AcTalk [Bri99]). Implicitní zpracování znamená, že objekt ihned po přijetí zprávy vyvolá příslušnou akci, po jejímž ukončení předá zpět volající entitě výsledek akce. Volající entita přitom *čeká* na tento výsledek. Explicitní zpracování znamená, že objekt po příjmu zprávy sám rozhoduje, kdy a za jakých podmínek zprávu zpracuje.

- *Synchronizace*. Synchronizace vnitřních procesů objektu definuje, jakým způsobem objekt kontroluje souběžný výpočet nad svým vnitřním stavem, tj. explicitní zpracování zpráv, zpracování vnitřních událostí apod.

Proces zasílání zpráv (*message passing*) můžeme rozdělit na dva typy:

- *Synchronní*. Odesílatel zprávy čeká na dokončení zpracování zprávy příjemcem. Podle typu aktivity objektu je zpráva zpracována buď v kontextu procesu odesílatele, nebo příjemce (běžnější je v kontextu procesu odesílatele).

- *Asynchronní*. Odesílatel zprávy nečeká na dokončení zpracování zprávy příjemcem, ale pouze zasílá žádost o zpracování zprávy a okamžitě pokračuje ve své vlastní činnosti. Po ukončení zpracování zprávy příjemcem může být (nemusí) odesílateli zprávy předán přímý výsledek zprávy (nejčastěji opět asynchronní zprávou). Jak vyplývá z charakteru koncepce asynchronních zpráv, nejsou uplatnitelné v systému, který je tvořen pouze pasivními objekty.

3.1.2 Aktivita objektu

Chování objektu je popsáno sadou operací, které jsou asociovány s příslušnými metodami. Zpráva je pak chápána jako žádost o provedení příslušné operace. Na tomto místě musíme rozlišit (při prvním přiblížení) tyto základní fáze zpracování zprávy: *příjem zprávy*, *zpracování zprávy* a *přenos výsledné hodnoty*. Pokud je příjemci zaslána zpráva, dojde nejprve k jejímu přijetí, v dalším kroku ke zpracování (tj. vyvolání příslušné metody) a následně k přenosu výsledné (návratové) hodnoty metody zpět do místa volání. Pro implicitní zpracování zpráv platí, že všechny fáze proběhnou okamžitě a najednou. Pro explicitní zpracování zprávy platí, že tyto fáze nemusí následovat těsně za sebou, tzn. že po přijetí zprávy může dojít k jejímu zpracování v blíže neurčeném čase (podrobněji tuto problematiku popisuje kapitola 5.4). Podle způsobu zasílání a zpracování zpráv můžeme rozdělit objekty do dvou kategorií:

- *Pasivní objekt*. Objekt je *pasivní* součást modelu, který reaguje pouze na vnější podněty. Přijímá pouze synchronní zprávy, které zpracovává implicitně.

- *Aktivní objekt* (*aktor*). Zkusme nyní definici objektu doplnit o interní chování (životní proces), které není spojeno s nutností vnějších podnětů, ale pouze s objektem samotným. Objekt s interním chováním je objekt *živý* a může sám vysílat podněty do svého okolí nebo vyvolat modifikaci svého vnitřního stavu. Takový objekt nazvěme *aktor*. Nejčastějším prostředkem pro realizaci interního chování je *proces*. Objekt zapouzdřuje *chování*, *data* a *proces*. Aktor je schopen zpracovávat synchronní i asynchronní zprávy, a to implicitně nebo explicitně.

S přístupem aktivního objektu ovšem vzniká problém kooperace a synchronizace procesů nad objektem. V jednom časovém okamžiku může nad objektem běžet několik procesů, které mohou modifikovat jeho vnitřní stav – jde o souběh procesů. Proces vzniká jako reakce na zaslanou zprávu. To znamená, že vyvolanou metodu můžeme považovat za jeden *proces*. Základním předpokladem pro správnou interpretaci aktivního objektu je vzájemné vyloučení akcí jednotlivých procesů (pokud je to nutné). Existuje více variant vzájemného vyloučení:

Optimistická. Předpokladem je, že každý proces *může ovlivňovat* stav objektu a je nutné synchronizovat navzájem pouze tyto procesy. Základní myšlenka této varianty je, že každý proces si musí ohlídat svou kritickou sekci, která má být synchronizována s ostatními.

Konzervativní. Předpokladem je, že každý proces *ovlivňuje* stav objektu a je nutné zajistit, aby v daném okamžiku byl aktivní *právě* jeden proces. To samo o sobě ještě k zajištění vzájemného vyloučení nestačí. Pokud by docházelo k přepínání kontextů mezi procesy, může dojít k nekonzistencím stavu objektu a reprezentace stavu objektu v procesu. Proto je navíc k požadavku aktivity *právě* jednoho procesu přidán požadavek vykonání procesu *atomicky*. Základním principem tohoto přístupu je fronta zpráv. Každá příchozí zpráva je uložena do fronty příchozích zpráv. Objekt obsahuje plánovač, který rozhoduje, která zpráva bude v dalším kroku zpracována. Tímto principem je zajištěna invokace pouze jednoho procesu (metody) v daném čase. Jak vyplývá z podstaty zpracování zprávy, jde o *asynchronní* přístup. Tento přístup se také někdy nazývá *Object-oriented concurrent programming* (OOCP) [Bri99] a je zobecněním objektově orientovaného přístupu k modelování.

Událostní. Tato varianta je použita v případě OOPN a posléze zobecněna pro celý simulační systém. Předpokladem je, že základní výkonnou jednotkou není proces, ale *událost*. Proces je tvořen sekvencí událostí, které obecně nemusí být kauzálně svázané. Pro vzájemné vyloučení provádění událostí platí stejná pravidla jako pro konzervativní variantu s procesy.

3.2 Klasifikace modelů

Předchozí kapitola se věnovala chování objektu ve smyslu výpočetním, tj. jakým způsobem zpracovává zprávy, jakým způsobem probíhá evoluce objektu. To je jeden aspekt související s definicí objektu, který je relevantní z pohledu realizace (implementace) výpočetního prostředí a pouze částečně z pohledu řešení vlastního (doménového) problému. Nyní se podíváme podrobněji na modelování jako takové, tj. na specifikaci řešeného problému s využitím vybraného paradigmatu.

Modelování systémů spočívá ve vytváření tzv. *abstraktních modelů* reálného světa, ze kterých se pak vytvářejí *simulační modely*. Tyto modely jsou určeny k experimentům a ověřování charakteristik modelovaného problému empirickými a statistickými metodami. Tvorba modelů probíhá na *doménové úrovni*, tj. na úrovni sémanticky korespondující s řešeným (modelovaným) problémem. Naproti tomu model výpočetního chování definuje strukturu výpočetního systému a vztahy mezi jednotlivými komponentami tohoto systému. Jako příklad si můžeme uvést výpočetní systém Smalltalk: jistá množina tříd (resp. instance těchto tříd) tvoří výpočetní model celého systému, definuje strukturu vazeb, komunikační protokol a chování objektů. Prostřednictvím těchto objektů pak uživatel (či přímo doménový model) vytváří doménový model řešeného problému v tomto výpočetním systému.

Doménový model. Doménový model je model řešeného problému v daném doménovém paradigmatu. Popisuje strukturu, vazby a chování modelované části reality. Simulační model je model umístěný do reálného výpočetního prostředí – doménový simulační model je tedy reprezentován *výpočetním modelem*.

Výpočetní model. Výpočetní model je reprezentací doménového modelu v systému. Definuje výpočetní chování objektů, tzn. způsob zpracování zpráv, evokace metod, definuje rámec pro externí zásahy do doménového modelu apod. Tato práce se de facto zabývá výpočetními modely a jejich reprezentacemi. To pak tvoří jádro otevřené implementace simulačního systému.

3.3 Otevřené systémy

Otevřený systém si můžeme definovat různými způsoby. Jelikož se jedná o klíčovou vlastnost teoretických úvah a praktických aspektů prezentovaných v této práci, pokusíme se nyní popsat a vysvětlit tento pojem z několika hledisek. Půjde především o hledisko významové (tj. proč se vlastně zabýváme otevřenými systémy) a realizační (tj. výstavby rámce – *frameworku* – pro budování otevřených systému na jeho základě).

3.3.1 Koncepce otevřených implementací

Současný trend v systémech pro podporu komplexních aplikací (např. operační systémy) je umožnit aplikacím nejen používat služby nabízené daným softwarem, ale také poskytnout prostředky k řízení poskytovaných služeb. To znamená, že aplikace má možnost modifikovat způsob provádění služeb systému apod. Tento trend se může zdát být v protikladu k tradičnějšímu přístupu – tzv. abstrakci černé skříňky (*black-box abstraction*). Ta říká, že nějaký modul (objekt) by měl odkrývat svou funkcionalitu, ale skrývat její implementaci. Přístup ukrývání má nesporně mnoho výhod, především v oblasti portability, znovupoužitelnosti či jednoduchosti návrhového procesu. Naproti tomu lze najít i nevýhody spojené především s výběrem „vhodnější" implementace požadovaného chování (rozhraní), s přizpůsobením vybraných částí prostředí, ve kterém je modul nasazen, či s evolucí modelu při jeho běhu.

Principy otevřené implementace nabízejí řešení uvedených problémů. Je třeba podotknout, že tyto principy by měly být chápány spíše jako rámec (framework) pro více flexibilní návrh a použití „černých skříněk".[1] Otevřené implementace dodržují zásady kvalitního návrhu rozhraní [Hof88] :

- Rozhraní je *konzistentní*. Rozhraní poskytuje operace pojmenované podle jasného a neměnného klíče. Navíc podobné nebo stejné operace různých objektů mají stejné identifikátory.
- Rozhraní je *esenciální*. Rozhraní poskytuje operace, které zaručují plnou poskytovanou funkcionalitu objektu (komponenty).
- Rozhraní je *minimální*. Rozhraní poskytuje pokud možno minimální množinu operací. Složitější akce se pak dosahují provedením příslušných operací ze základního rozhraní.
- Rozhraní je *skrývající*. Rozhraní deklaruje bariéru mezi poskytovanou operací a její implementací.

Nejvíce používaným principem otevřené implementace je *metaúrovňová architektura* a s ní spojený *metaobjektový protokol*. Tento princip je využíván i v návrhu simulačního systému, a proto se mu v další části budeme věnovat.

3.3.2 Metaúrovňové architektury

Výraz *meta*[2] vyjadřuje vyšší sféru řízení, de facto informace o informacích. Tato definice je ovšem příliš obecná a může být aplikována na mnoho problémů. Proto si nyní výraz *meta* a spojení *meta-level* adaptujeme na naše pojetí ve světě objektů. Předpokládejme, že program je reprezentován množinou objektů a v některých případech je chování objektů popsáno třídami, tedy třídními objekty. My ovšem potřebujeme rozlišit řídicí sféru od sféry doménové. Abychom toho dosáhli, představíme si dvě různé úrovně toku řízení: *doménovou úroveň* (*domain level*)[3] a *metaúroveň* (*meta level*) [Mae87].

Jako příklad si můžeme uvést Smalltalk [GR89, FJ89]. Každá třída má svou metatřídu, která má stejný význam jako třídy pro běžné objekty. Jelikož metatřídy řídí chování tříd, tedy mohou být chápány jako metaobjekty pro třídy. Architektura Smalltalku zavádí meta-úroveň (reprezentovanou metatřídami) pro řízení tříd. Na základě rozlišování různých úrovní v systému můžeme definovat následující pojmy:

Doménová úroveň (domain level): *Aplikační úroveň, která popisuje (definuje) řešený problém prostřednictvím daného paradigmatu. Na této úrovni probíhá evoluce doménových objektů (tj. objektů, které mají přímou vazbu na reálný řešený problém).*

[1] Není dobré (a ani to principy otevřených implementací nedělají) zavrhnout „černé skříňky". V některých případech je však výhodné mít k dispozici mechanismy, jak tento koncept (čátečně) obejít.

[2] Výraz *meta* pochází z řečtiny a znamená *za* nebo *po*. V našem pojetí to lze chápat jako označení pro něco, co stojí *za* (tedy *v pozadí za*) nějakým objektem a vyjadřuje (popisuje) jeho aspekty.

[3] V literatuře bývá obvykle tato úroveň označována jako základní (*base level*). My se však přidržíme označení doménová úroveň (*domain level*), protože to podle mého názoru lépe vystihuje podstatu této úrovně – reprezentace řešeného problému (viz 3.2).

Meta úroveň (meta level): *Meta úroveň je generický pojem, který vytváří prostor pro evoluci systémových (řídicích, implementačních) metaobjektů. Metaobjekty popisují, řídí, implementují či modifikují doménové objekty. V případě víceúrovňové architektury může metaobjekt ovládat i jiné metaobjekty.*

Metaobjektový protokol (metaobject protocol): *Objektově orientované rozhraní umožňující komunikaci objektů doménových úrovní a meta úrovní. Metaobjekt je pevnou součástí systému, a tudíž metaprotokol je sestaven a znám předem. Přestože je metaprotokol pevně stanoven, existuje možnost jeho inkrementálního vývoje (viz časová a typová klasifikace reflektivity, kap. 3.4.2 a 3.4.3).*

Doménový protokol (domainobject protocol): *Doménový protokol definuje operace nabízené doménovým objektem. Doménový protokol je vytvářen uživatelem (příp. samotným systémem či modelem) buď staticky, nebo během života systému.*

Doménová operace (domain operation): *Operace z doménového protokolu.*

Metaobjektová operace (metaobject operation): *Operace z metaobjektového protokolu.*

3.4 Reflektivita

S otevřenými implementacemi a metaúrovňovými architekturami úzce souvisejí jejich reflektivní vlastnosti. V následujícím textu proto budeme tuto problematiku diskutovat. Nejprve si uvedeme základní definici reflektivity:

Reflektivita (reflection) *je proces uvažování systému o sobě samém a jednání systému nad sebou samým [MMWY92]. K odhalení interních aspektů objektu (reification) reflektivní systém včleňuje speciální data (reifiable data), která reprezentují nebo implementují jeho strukturální a výpočetní aspekty v sobě na metaúrovni.*

Jak je vidět z předchozí definice reflektivity (nebo lépe *reflektivního systému*), jde o takový systém, který má sám v sobě prostředky pro modifikaci své struktury či svého chování, a to *pouze prostřednictvím* těchto prostředků (viz návrh kvalitního rozhraní, kap. 3.3.1). Protože tato definice je příliš fádní a protože existuje celá řada variant reflektivních přístupů k tvorbě systémů, uvedeme si nyní praktičtější definice, se kterými budeme dále pracovat v tomto textu.

Vyjděme z předpokladu meta přístupu k vytvoření reflektivního systému. Obecně pak můžeme říci, že meta úroveň řídí doménovou úroveň ve dvou krocích [Paw98]:

- Doménový objekt volá metaobjekt s žádostí o změnu sémantiky svých operací nebo struktury. Tento přístup bývá nazýván *reifikace* (*reification*)[4] implementace nebo sémantických aspektů objektu.
- Po ukončení reifikace je řízení předáno z metaobjektu zpět doménovému objektu. Protože metaobjekt změnil část doménového objektu, jeho chování či reprezentace je nyní změněna. Tento proces se nazývá *reflektování* (*reflecting*) změn zpět do doménového objektu.

Vztah mezi doménovým objektem a metaobjektem (*meta relationship*) není symetrický. Zatímco metaobjekt *potřebuje* znát interní reprezentaci doménového objektu k tomu, aby mohl provádět operace

[4] *Reify* podle výkladového slovníku znamená *consider an abstract concept to be real*. V našem pojetí to můžeme chápat jako vytváření konkrétních dat či informací z abstraktních konceptů objektu, které mohou být kontrolovány metaobjektem a tím mohou být kontrolovány příslušné aspekty objektu.

nad sémantickými aspekty objektu (struktura, chování), doménový objekt *nepotřebuje* znát interní reprezentaci metaobjektu. Doménový objekt potřebuje pouze vědět, jak poslat reifikační zprávu (*reification message*) a které informace na doménové úrovni jsou relevantní pro metaobjekt.

Zkusme si nyní (na základě předchozích úvah) definovat následující dva pojmy, které se přímo vážou k termínu *reflektivní systém*:

Reifikace (reification): *Operace zahrnující explicitní vyjádření doménových konceptů nebo mechanizmů, které jsou transparentní pro programátora. Tyto reifikované koncepty jsou obvykle implementovány v metaobjektech (tedy na meta úrovni). Lze říci, že reifikační proces nastane v okamžiku, kdy doménová úroveň předá řízení meta úrovni.*

Reflektivita (reflection): *Operace zahrnující modifikace interpretace doménové úrovně k zajištění změn sémantiky objektů doménové úrovně. Lze říci, že proces reflexe nastane v okamžiku, kdy meta úroveň zpracuje některý speciální výpočet a předá řízení zpět základní úrovni.*

Nutno podotknout, že se občas slova *reification* a *reflection* používají nezávisle na tom, která meta komunikace právě nastává. Často záleží na úhlu pohledu autora, který si zvolí jako klíčový pro popis daného problému. Pokud je zdůrazněno, že je to právě *meta úroveň*, která modifikuje doménovou úroveň, můžeme hovořit o *reflektivitě* (*reflection point of view*). Naopak, pokud jako klíčovou myšlenku upřednostňujeme fakt, že je to *doménová úroveň*, která je modifikována meta úrovní, můžeme hovořit o *reifikaci* (*reification point of view*) [Paw98]. Jako příklad lze uvést jazyky a systémy *OpenC++* a *Inguana*. OpenC++ používá slovo *reflect* k získání reference na doménový objekt, zatímco Inguana používá slovo *reify* k vyjádření téhož významu. V této práci se přidržíme terminologie z pohledu reflektivity.

3.4.1 Výpočetní a strukturální reflektivita

V této kapitole budeme diskutovat dva různé pohledy na reflektivitu: strukturální (*structural reflection*) a výpočetní (*behavioral reflection*) [Fer89]. Strukturální reflektivita reifikuje strukturální aspekty doménového objektu, jako jsou např. dědičnost, množina operací nabízená objektem, implementace operací apod. Naproti tomu výpočetní reflektivita se zaměřuje na reifikaci výpočtu, tj. na způsoby a možnosti provádění evoluce objektů. Nicméně rozdělení reflektivity na dva nezávislé typy je extrémním pohledem. Ve většině případů implementace reflektivních systémů jde o kombinaci obou principů, přičemž je kladen větší důraz na jeden z nich.

Výpočetní reflektivita je typicky používaná v operačních systémech a komunikacích. Jako příklad si můžeme uvést reflektivní operační systém Apertos [Yok92b], implementující správu systémových objektů (*system object management*) na metaúrovni. Jedním z hlavních výhod tohoto přístupu je, že např. zmiňovaný systém Apertos umožňuje adaptovat systémové strategie řízení (*system policies*) na lepší možné služby. Další příklady systémů využívající vlastností výpočetní reflektivity jsou CodA [McA95], Iguana [GC96] nebo Actalk [Bri99]. Většina z těchto systému implementuje také strukturální reflektivitu.

Strukturální reflektivita implikuje, že na metaúrovni musí být popsána doménová úroveň (nebo její význačné vlastnosti). Příkladem systémů, které využívají strukturální reflektivitu, jsou CLOS [KdRB91] nebo OpenC++ v2 [Chi95]. Společná technika pro docílení strukturální reflektivitu je zavedení metatříd. Metatřída je metaobjekt, který řídí několik doménových objektů. Jako příklad si můžeme uvést *Run-Time Type Identification* v C++ [Str92].

Souhrn: Výpočetní reflektivita je více vhodná a přímo připravená k implementaci některých systémových mechanizmů, zatímco strukturální reflektivita je široce uplatňovaná pro jazyková rozšíření.

3.4.2 Časová klasifikace reflektivity

Metaúrovňové architektury jsou typicky vyvíjeny na poli interpretovaných jazyků. To pramení z jednoduchého důvodu – interpret programu musí zkonstruovat mnoho informací (*reify data*) o programu. Tyto informace nejsou používané vlastním programem, ale slouží interpretu pro vykonávání programu. V kontextu reflektivního programování to však může být přímo interpretace programu, která je podstatou výpočtu, tzn. že reflektivní výpočet slouží k určení toho, jak interpretovat program. Tedy podstatným rysem reflektivního programování je, že dokáže *dynamicky* adaptovat interpretaci programu (příp. sémantiku doménových objektů). Pokud se interpret rozšíří o reflektivní vlastnosti prostým přidáním podpory pro odkrytí meta-informací doménovému programu, umožní to programu ovlivňovat rozhodovací proces interpretu dynamickými změnami meta-informací. Tento přístup nazveme *dynamická reflektivita*.

V případě kompilovaných jazyků také existují metainformace o programu. Generování metainformací může být rozšířeno pro potřeby reflektivity a zpřístupněno programu. Tyto informace jsou vytvářeny během kompilace a oproti interpretovaným jazykům jsou implicitní a neměnné (tj. nelze je dynamicky měnit). Tento přístup nazveme *statická reflektivita*. Nicméně tento koncept přináší jednu zásadní výhodu oproti vytváření metainformací za běhu – představuje mnohem menší režii při evoluci programu. Současný trend ve vytváření reflektivních systémů je přesunout proces reflexe tak blízko době kompilace, jak je to jen možné (z hlediska útlumu výhod dynamické reflektivity).

3.4.3 Typová klasifikace reflektivity

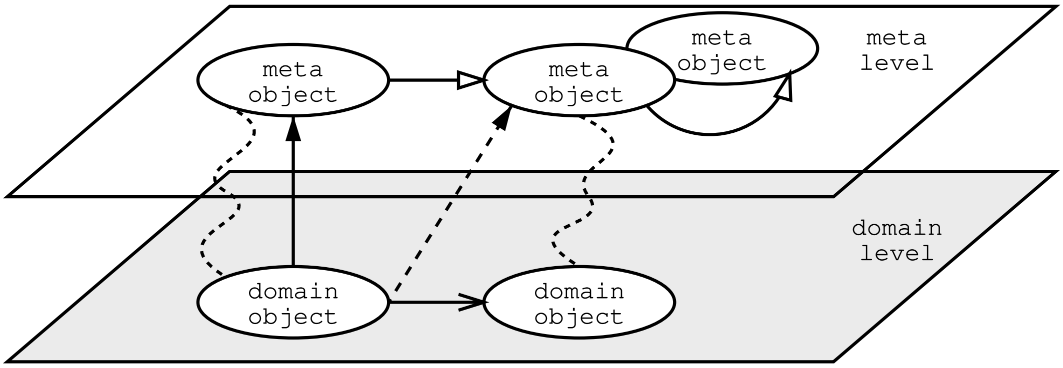
Jakkoliv by se mohlo zdát, že klasifikace reflektivity podle času (tedy zda způsob získávání reifikačních dat je dán staticky při kompilaci nebo dynamicky za běhu systému) je dostačující, zavádí se navíc ještě typová klasifikace metaúrovňových systémů [Chi96]. Ta rozlišuje, zda je takový systém založen na objektech nebo třídách. V třídně založených systémech jsou metaobjekty navrženy jako metaobjekty pro třídy (tedy jako metatřídy), tudíž existuje obvyklá instanční vazba mezi metaobjektem (tj. metatřídou) a doménovým objektem (tj. třídou). Jako typický příklad si můžeme uvést systém Smalltalk [GR89], kde třídy jsou chápány jako objekty, tj. instance svých metatříd. V objektově založených reflektivních systémech jsou metaobjekty navrženy jako metaobjekty pro objekty, tzn. že doménový objekt není instancí svého metaobjektu. V tomto případě jsou metaobjekty chápány spíše jako objekty, které zastupují doménové objekty v systému a při jejich evoluci.

3.5 Metaobjektový protokol

Nejpodstatnější aspekty, týkající se metaúrovňových architektur, jsou jednak struktura rozdělení řídicího toku a ovšem způsob komunikace mezi metaúrovněmi a doménovými úrovněmi. Proto často bývají objektově orientované metaúrovňové architektury nazývány jako *metaobjektový protokol* (MOP). Tento pojem byl poprvé použit v systému CLOS [KdRB91]. MOP znamená deklaraci rozhraní pro přizpůsobení systému či jazyka [GC95]. Tento protokol můžeme rozdělit do tří kategorií [Zim96]: explicitní (*explicit metaobject protocol*), implicitní (*implicit metaobject protocol*) a vnitřní (*inter metaobject protocol*).

3.5.1 Explicitní metaobjektový protokol

Rozhraní explicitního metaprotokolu je použito doménovými objekty pro komunikaci s metaobjekty k zajištění přímého řízení chování doménového objektu. Např. doménový objekt může požádat metaobjekt o serializaci svého stavu (tj. uložení stavu doménového objektu). Explicitní metaobjektový protokol *není* transparentní pro doménové objekty, ale je nesmírně užitečný pro *přizpůsobení* a *řízení* implementace. Na obrázku 3.1 jsou zobrazeny jednotlivé typy metaobjektového protokolu. Explicitní MOP je naznačen plnou čarou s šipkou ve tvaru plného trojúhelníku. Na uvedeném příkladu je naznačen doménový objekt,



Obrázek 3.1: Kategorie metaobjektového protokolu

který zasílá meta zprávu (tj. operaci z MOP) vybranému metaobjektu. Typicky může tato zpráva vyvolat replikaci objektu apod. Jelikož jde o explicitní zprávu, může vyvolat pozastavení evoluce doménového objektu do doby vyřízení metaoperace.

3.5.2 Implicitní metaobjektový protokol

V kontrastu s explicitním metaobjektovým protokolem, kdy objekt musí přímo evokovat operace metaobjektu, existuje implicitní metaobjektový protokol, který je z pohledu doménových objektů *transparentní*. K vyvolání operací z implicitního metaobjektového protokolu tedy dochází *bez* přímé a vědomé účasti doménového objektu. Pokud je nějaký kód (operace) vyvolán na metaúrovni, může automaticky, transparentně a bezprostředně ovlivňovat aspekty chování nebo strukturální aspekty doménového objektu. Tato implikace bývá v literatuře označována jako *kauzální konektivita* (*causal connectivity*) [Mae87].

Na obrázku 3.1 je implicitní metaobjektový protokol naznačen čárkovanou čarou s šipkou ve tvaru vyplněného trojúhelníku. Plná čára ukončená prostou šipkou pak reprezentuje komunikaci (zaslání zprávy) mezi doménovými objekty. V reakci na zaslání doménové zprávy dojde k vyvolání operace z implicitního metaobjektového protokolu. Obecně bývá tento protokol fixní, protože jeho volání je zabudováno v okamžiku překladu doménového kódu. Způsob implementace pak závisí na použitém výpočetním modelu, přičemž na něm také závisí náročnost implementace.

3.5.3 Interní metaobjektový protokol

Tento protokol je podobný explicitnímu metaprotokolu s tím rozdílem, že slouží pro vzájemnou komunikaci metaobjektů a je skryt z pohledu doménových objektů. Na obrázku 3.1 je interní metaobjektový protokol naznačen plnou čarou s šipkou ve tvaru prázdného trojúhelníku. Slouží primárně pro přenos informací mezi metaobjekty.

3.6 Systémy s metaúrovňovou architekturou

V této části si uvedeme několik dalších ukázek jazyků a systémů jako reprezentaci metaobjektového protokolu (či metaúrovňových architektur). Tyto jazyky a systémy slouží jako inspirace možností a použití jednotlivých přístupů k aplikaci reflektivních konceptů.

3.6.1 OpenC++ v1

OpenC++ v1 [CM93] reflektuje chování objektu v čase jeho běhu voláním metaúrovně za předpokladu, že nastane jedna ze tří událostí:

- volání metody,
- vytváření obektu,
- přístup k atributům objektu.

Základní myšlenka spočívá v tom, že programátor na doménové úrovni explicitně (pomocí speciální formy) označí třídu, metodu či atribut. Preprocesor přidá příslušný C++ kód, který zajišťuje volání metod metaobjektu. Takto vniklý kód je pak přeložen standardním překladačem C++. Při pokusu o přístup k nějaké takto označené komponentě pak dojde k evokaci příslušné metody metaobjektu.

Následující kód ukazuje příklad doménového programování v OpenC++ v1. Kód obsahuje definici třídy C, jejíž instance jsou reflektovány instancemi třídy MC (tedy instance třídy MC reprezentuje metaobjekt pro instanci třídy C). Při invokaci metod $func(int)$ a $update(int)$ dojde k reifikaci na doménovou úroveň. Reifikace metody $update(int)$ je navíc parametrizována kategorií s názvem $write$.

```
// MOP reflect class C: MC;
class C {
  public:
    C();
    // MOP reflect:
      int func(int);
    // MOP reflect(write):
      int update(int);
    ...
}
```

Třída MC musí být deklarovaná jako potomek speciální třídy $MetaObj$. Ta pak definuje operace zajišťující reifikaci doménové úrovně. OpenC++ v1 tedy implementuje reflektivitu chování, a to v čase kompilace (statická reflektivita).

3.6.2 CodA

CodA [McA95] je objektový systém, založený na metaobjektovém protokolu (MOP) s dynamickou (*run-time*) reflektivitou, implementovaný nad Smalltalkem. Je zaměřen především na usnadnění implementace distribuovaných objektů a změn chování objektů během jejich evoluce. Doménový objekt může být řízen množinou sedmi metaobjektů (*Send*, *Accept*, *Queue*, *Receive*, *Protocol*, *Execution*, and *State*). Uživatel může modifikovat chování doménových objektů změnou některých z metaobjektů či derivací nových typů metaobjektů.

3.6.3 Apertos

Apertos [Yok92b] je operační systém založený na reflektivním pracovním rámci (framework). V tomto modelu jsou systémové objekty podporovány skupinou metaobjektů, které jsou v systému Apertos nazývány metaprostor (*metaspace*). Objekty a metaobjekty jsou reprezentovány uvnitř metahierarchie, která je definovaná uživatelem. To znamená, že může vznikat neomezený počet metaprostorů (v souladu s požadavky uživatelů), přičemž jednotlivé metaobjekty mohou být sdíleny různými metaprostory. Apertos můžeme klasifikovat jako MOP s výpočetně orientovanou reflektivitou.

3.6.4 AcTalk

AcTalk (*Ac*tors in Small*Talk*) [Bri99, Bri88] je metaúrovňová architektura implementovaná ve Smalltalku. Představuje aktorově založené paralelní objekty k jinak pasivnímu objektovému prostředí. K tomu využívá metaúrovňové architektury a metaobjektový protokol. Jeho hlavním cílem je zajistit testovací framework pro popis chování objektů v prostředí, které primárně podporuje souběžný (*concurrent*) běh a interakci na základě zasílání zpráv (*message passing*). Tento framework přináší některé myšlenky (především aktivní objekty), které je možno brát jako inspiraci pro vývoj simulačního systému popisovaného v této práci.

Kapitola 4

Petriho sítě jako prostředek pro modelování, simulaci a prototypování

Petriho sítě byly vyvinuty v šedesátých a sedmdesátých letech 20. století a brzy se ukázaly být vhodným jazykem pro popis a analýzu synchronizace, komunikace a sdílení zdrojů mezi paralelními procesy. Přesto se při praktickém použití naráželo na dvě zásadní nevýhody. Vytvářený model byl často nepřiměřeně velký, protože veškerá manipulace s daty musela být reprezentována přímo v síťové struktuře. Petriho sítě také neobsahovaly koncept hierarchie, takže bylo nemožné vytvářet skutečně rozsáhlé modely jako množinu separátních submodelů. Tyto problémy byly odstraněny při vývoji vysokoúrovňových Petriho sítí a hierarchických Petriho sítí v osmdesátých letech. Jednou z nejznámějších variant vysokoúrovňových Petriho sítí je CPN (Coloured Petri Nets). CPN spojuje jak strukturování dat, tak i hierarchickou dekompozici beze ztráty síly původního formalismu.

V devadesátých letech se začaly objevovat myšlenky spojení vysokoúrovňových Petriho sítí s objektovou orientací [ZH]. V této kapitole si přiblížíme několik variant spojení objektové orientace a Petriho sítí a dále se budeme věnovat specifické formě objektově zaměřených Petriho sítí – Objektově orientovaným Petriho sítím (OOPN) [Jan98], které jsou vyvíjeny od poloviny devadesátých let na Fakultě informačních technologií Vysokého učení technického v Brně.

4.1 Objektově orientované přístupy k Petriho sítím

4.1.1 Renew

V oblasti objektově orientovaných systémů a Petriho sítí probíhá intenzivní výzkum na různých vědeckých pracovištích ve světě. Jako příklad si můžeme uvést *University of Hamburk* , kde probíhá výzkum a vývoj Objektových Petriho sítí (OPN) [Mol95], které se svou podstatou blíží OOPN (viz kapitola 4.2). Jejich nejzajímavější vlastností jsou synchronní kanály podobající se svými vlastnostmi synchronním portům (viz kapitola 4.2.4). Na *University of Hamburk* byl také vyvinut nástroj pro simulaci těchto sítí – *ReNew* [NF98, Ren04]. Nástroj ReNew byl implementován v jazyce Java a ten také tvoří inskripční jazyk OPN. Definované sítě (objekty) jsou překládány do kódu jazyka Java a poté překládány do mezikódu Javy jako kterékoliv jiné třídy. Třídy OPN tedy korespondují přímo s třídami Javy a oproti přístupu OOPN neposkytují takovou flexibilitu. Z toho vyplývá, že během evoluce OPN nelze provádět transparentní změny stavu aktivních sítí či změny chování objektů.

4.1.2 Petri Net Kernel

Petri Net Kernel [KW01] se od ostatních objektově orientovaných přístupů k Petriho sítím odlišuje v jednom zásadním aspektu – netvoří ucelený formalismus. Nástroj, který s tímto formalismem pracuje, však vytváří infrastrukturu pro tvorbu dalších nástrojů založených na Petriho sítích, včetně aplikací pro jejich analýzu, modifikaci, simulaci nebo verifikaci. Petri Net Kernel není omezen na jeden typ Petriho sítí, ale umožňuje uživatelům použít jejich specifická rozšíření a implementovat nové algoritmy pro analýzu, simulaci či verifikaci.

4.1.3 Objektové Petriho sítě

Jedním z mnoha přístupů k objektové orientaci Petriho sítí jsou také *Object Petri Nets* vyvinuté C. Lakosem [Lak95b, Lak95a, Lak97]. Tyto sítě jsou jistou formou zobecnění hierarchické Petriho sítě. Jedná se o poměrně univerzální a flexibilní řešení, avšak definice a použitý formalismus jsou komplikované.

4.1.4 Kooperativní sítě

Koncepce kooperativních sítí (*Cooperative Nets*) [SB94] se blíží definici a sémantice OOPN. Zásadní rozdíl spočívá v konceptu tvorby instancí sítí v objektu. Zatímco v pojetí OOPN existuje jedna síť jako objektová a pro každou metodu existuje zvláštní síť, v pojetí kooperativních sítí existuje pro každý objekt právě jedna síť. V okamžiku vyvolání metody vzniká instance příslušné sítě a argumenty metody se vloží do příslušných míst (OOPN). V pojetí koooperativních sítí se v reakci na zaslanou zprávu vloží argumenty zprávy do příslušných míst sítě objektu.

Souhrn: Všechny výše uvedené ukázky různých přístupů k objektově orientovaným Petriho sítím nebo objektovým prostředím pro vývoj různých variant Petriho sítí se svou podstatou blíží variantě Objektově orientovaných Petriho sítí (OOPN). Zásadní rozdíl mezi OOPN a jinými formalismy lze spatřovat v následujících bodech:

- Větší provázanost Petriho sítí a inskripčního jazyka, která umožňuje tvorbu prototypových aplikací s řízením na základě Petriho sítí, tvorbu workflow[1] aplikací apod.
- Vysoký stupeň dynamičnosti OOPN, který umožňuje vývoj sítí za běhu modelu (simulace, běh prototypu) samotným modelem nebo vnějšími zásahy.

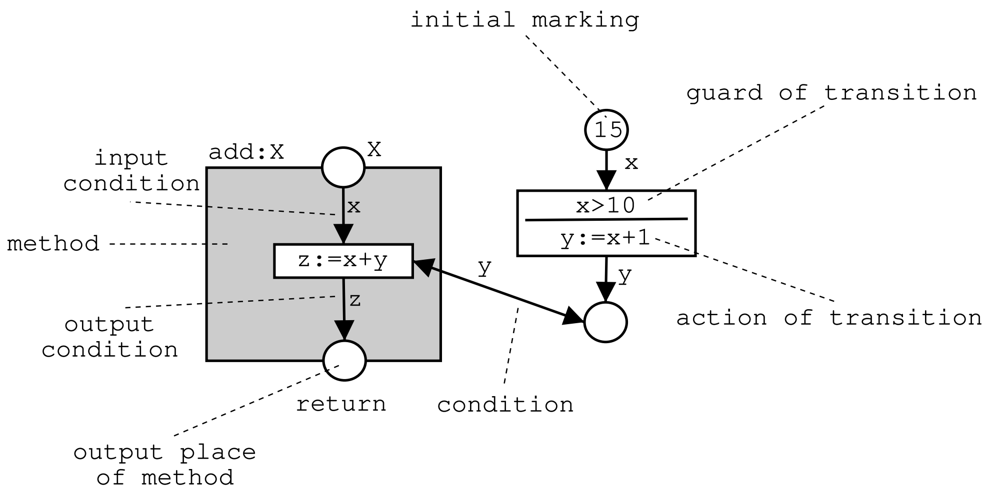
4.2 Objektově orientované Petriho sítě

V této kapitole si nastíníme základní pojmy a principy Objektově orientovaných Petriho sítí (OOPN). Definice OOPN [Jan98] má několik úrovní. Základní úroveň vymezuje primitivní sémantiku výrazů Petriho sítě. Další úrovní je *systém sítí* a *instance sítí*. Sítě jsou součástí metod a tříd. OOPN je tvořena *systémem tříd*. Dále je definován *objekt* a *systém objektů*, který určuje dynamiku a stav OOPN. OOPN byly od počátku koncipovány jako specifická varinta splynutí Petriho sítí a systému Smalltalk. S OOPN se také pojí jazyk PNtalk popisující strukturu a chování sítí objektů OOPN. Syntax jazyka PNtalk a ukázkový příklad je uveden v příloze A. Jazyk Smalltalk také tvoří inskripční jazyk OOPN.

4.2.1 Struktura OOPN

V definici OOPN rozlišujeme *konstantní* (nebo také *statické*) objekty a *neprimitivní* objekty. Neprimitivní objekty jsou specifikovány třídami. Mají svůj vnitřní stav, který se může měnit na základě vnějších či vnitřních podnětů. Statické objekty chápeme jako konstanty, jejichž jména lze ztotožnit s obsahem (hodnotou) objektu. Jedná se např. o čísla, znaky, symboly apod. Na obrázku 4.1 je popisován význam jednotlivých prvků třídy OOPN. V dalších podkapitolách je tento popis podrobněji rozveden.

[1]V otázce použití Petriho sítí pro definice workflow procesů existuje celá řada aktivit, a to např. i v rámci již zmiňovaného projektu Renew.



Obrázek 4.1: Popis prvků OOPN

4.2.2 Systém sítí

Neprimitivní objekty jsou definované třídami, které jsou popsány prostřednictvím Petriho sítí. Síť je složena z míst, přechodů a orientovaných hran mezi nimi, které reprezentují vstupní a výstupní podmínky přechodů. Vstupní podmínka je graficky reprezentovaná šipkou směřující od místa k přechodu, výstupní podmínka šipkou směřující od přechodu k místu. Každé místo a přechod mají své jedinečné jméno v daném prostoru jmen (tj. v síti). Součástí místa může být počáteční značení (multimnožina značek). Značka může reprezentovat statický objekt, jméno neprimitivního objektu, jméno třídy, nebo n-tici složenou z vyjmenovaných položek. Uvažují se i složené (vnořené) n-tice.

Hrany jsou ohodnoceny *hranovými výrazy*, které mohou obsahovat již vyjmenované prvky. Tyto prvky mohou být nahrazeny zástupnou, tzv. *volnou* proměnnou. Dále jsou zavedeny *podmínky přechodu* (bezpřívlastkové, rozdílné od vstupních a výstupních podmínek), graficky reprezentované oboustrannými šipkami.

Přechody obsahují *stráž* a *akci* přechodu. Stráž tvoří boolovské výrazy (výrazy s boolovským návratovým typem). Výrazem se rozumí zaslání zprávy objektu. Rozhodnutí o splnitelnosti stráže má charakter konjunkce všech výrazů ve stráži, tj. pro splnění stráže musí být splněny všechny výrazy. Akce je zaslání zprávy s možností přiřadit výsledek do proměnné.

Přechod může obsahovat i více akcí. V takovém případě se tento přechod chápe jako skupina přechodů, řazených sekvenčně za sebou podle jednotlivých akcí. Mezi jednotlivé přechody je vloženo nové místo (aby nově vzniklá síť odpovídala definici Petriho sítě), sloužící pro přenos příslušných objektů.

4.2.3 Metody

Metoda je definovaná jménem, sítí, vstupními (parametrovými) místy a výstupním místem *return*. Jména vstupních míst odpovídají parametrům metody. Při invokaci metody se do parametrových míst vloží objekty reprezentující příslušné parametry metody. Výstupní místo má metoda pouze jedno a nemůže být součástí vstupních podmínek sítě metody nebo objektové sítě.

4.2.4 Predikáty a synchronní porty

Součástí stráže přechodu je zasílání zpráv statickým objektům (např. $x > 10$), které testují hodnotu vstupujících statických objektů. Provedení přechodu však může být podmíněno i stavem libovolného objektu.

K tomuto účelu slouží *predikáty*. Predikát má tvar virtuálního přechodu, který má pouze testovací hrany a nemá žádnou stráž nebo akci. Predikát má přístup k místům sítě objektu, ovšem pouze nedestruktivní (nemůže měnit stav objektu, pouze jej testovat). Nutnou podmínkou pro predikáty je, že musí být proveditelné atomicky. Predikát je určen k dynamické fúzi s volajícím přechodem, tzn. že predikát se stává součástí stráže volajícího přechodu.

Rozšířením predikátu o vedlejší efekt, tj. možnost ovlivnit obsah míst objektové sítě, vznikne *synchronní port*. Při testování stráže přechodu, odkud se synchronní port volá, se stav objektu nemění. Pokud se přechod, který ze své stráže volal synchronní port, provede, provede se i synchronní port (jako přechod).

4.2.5 Systém tříd

Třída je definovaná *objektovou sítí* a množinou *metod*. Objektová síť je síť, která je instanciována v okamžiku vytvoření objektu (instance třídy) a její doba života je svázána s dobou života objektu (objektová síť tedy definuje *životní proces* objektu). Každá třída má právě jednu objektovou síť, která může být i prázdná. Metoda je tvořena sítí, která je instanciována v okamžiku vyvolání metody a zrušena v okamžiku ukončení metody. Přechody sítě metody mohou být podmíněny (vstupně i výstupně) místy objektové sítě.

4.2.6 Dědičnost

Třídy jsou definovány inkrementálně s využitím dědičnosti. Obecná třída všech tříd definovaných Petriho sítěmi je třída *PN*, která obsahuje prázdnou objektovou síť a prázdnou množinu sítí metod. Nicméně definuje základní chování všech aktivních objektů v systému OOPN. Dědičnost tříd je definována jako dědičnost objektových sítí (tj. zděděné přechody a místa objektové sítě mohou být předefinovány; nové přechody a místa mohou být přidány) společně s množinou sítí metod (implementace zděděných metod může být nahrazena novou; nová metoda může být přidána).

4.3 Evoluce objektů popsaných OOPN

Objekty popsané Petriho sítěmi spolu komunikují zasíláním zpráv. Na každou přijatou zprávu objekt reaguje tak, že vytvoří instanci metody korespondující se zprávou. Volající objekt čeká na dokončení metody, aby mohl převzít návratovou hodnotu (klasický *message passing* stejně jako např. v hostitelském prostředí Smalltalk).

Každý objekt může mít *objektovou síť*, která má pouze jednu instanci, jež vznikne při vytvoření objektu a zaniká nejpozději se zánikem objektu. Objektovou síť nikdo nevolá, nemá žádnou návratovou hodnotu (není to metoda) a reprezentuje aktivní životní proces objektu. Objektová síť a metody jsou popsány Petriho sítí – nadále budeme oba typy nazývat jednoduše *síť*.

Evoluce sítí je *událostně orientovaná*, což je základní rozdíl od evoluce kódu napsaného ve Smalltalku, která je *procesně orientovaná*. V Objektově orientovaných Petriho sítích (OOPN) můžeme za událost považovat provedení či proveditelnost přechodu. V každém okamžiku je definována množina proveditelných přechodů, a tedy událostí. Provedením přechodu (události) se může tato množina změnit. Podle typu proveditelnosti přechodu můžeme rozlišovat čtyři typy událostí [Jan98] v OOPN:

1. *A-proveditelný*. Atomické provedení přechodu. Akce přechodu reprezentuje zaslání zprávy statickému (konstantnímu) objektu.

2. *N-proveditelný*. Akcí přechodu je vytvoření nového neprimitivního objektu. Jedná se o atomické provedení přechodu.

3. *F-proveditelný*. Akcí přechodu rozumíme zaslání zprávy neprimitivnímu objektu. Operace je blokující, přechod čeká na ukončení vyvolané metody.

4. *J-proveditelný*. Dokončení přechodu po skončení vyvolané metody v akci přechodu. Ukončením metody rozumíme vložení alespoň jedné značky (objektu) do výstupního místa, přičemž metoda může mít více návratových hodnot (tj. více objektů v místě *return*). Přechod pak může být dokončen pro různá navázání, kde každé navázání je dáno jednou návratovou hodnotou metody. V okamžiku odebrání objektu volajícím přechodem dojde k ukončení a zrušení sítě volané metody.[2]

Provádění metody v OOPN není definováno jako proces, i když se o procesu v abstraktním pojetí mluví – objekt OOPN je tvořen množinou procesů, kde pod pojmem proces se rozumí instance sítě. Evoluce takové instance sítě není definována jako sekvence příkazů, ale jako sekvence událostí, kde událostí může být provedení přechodu podle jedné z variant proveditelnosti (A, N, F, J proveditelnost).

4.4 Interpret OOPN

Pro simulaci modelů popsaných Objektově orientovanými Petriho sítěmi (OOPN) slouží *interpret* OOPN. V této části si nastíníme ty základní problémy a jejich řešení, které jsou podstatné z pohledu architektury prezentované v této práci. Podrobnější informace jsou k dispozici v [Koč01a, PNt04].

4.4.1 Proces objektu

Objekty navzájem komunikují prostřednictvím zasílání zpráv. Pokud nějaký objekt (*sender*) zašle zprávu jinému objektu (*receiver*), tento objekt (receiver) zprávu zpracuje a předá zpět zdrojovému objektu (sender) návratovou hodnotu. Zdrojový objekt (sender) přitom čeká na dokončení vyvolané metody.

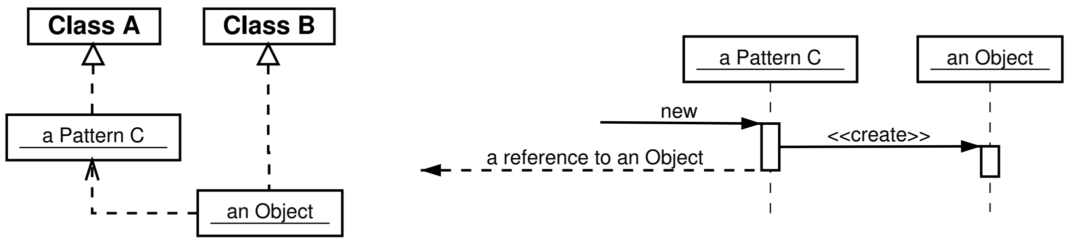
Zpracování vyvolané metody můžeme chápat jako (abstraktní) proces. V našem pojetí však chápeme proces objektu OOPN ne jako sekvenci operací, ale jako sekvenci událostí, které definují (abstraktní) podprocesy v procesu. Interpret OOPN je tedy založen na vnímání objektu jako procesu, který ovládá všechny abstraktní procesy v objektu [Koč01a, KR02].

Z hlediska návrhu systému přináší simulace metody OOPN problém, jak proces metody reprezentovat. Existuje několik přístupů k řešení tohoto problému:

- Proces metody reprezentovat reálným procesem. Při vzniku instance metody vznikne proces, ve kterém je metoda zpracována. Pokud je součástí zpracovávaného přechodu zaslání zprávy jinému neprimitivnímu objektu, dojde k suspendování procesu. Tento přístup má nespornou výhodu v realizaci – je velmi jednoduchá. Nevýhody však převažují nad výhodami – při zpracování události v metodě musí ostatní události čekat na dokončení této zpracovávané události; zpracovávaná událost musí čekat na dokončení volané metody, která je událostně orientovaná, takže událost není atomická – lze tedy hovořit o události? Šlo by uvažovat o simulaci OOPN jako o hybridní – spuštění přechodu je událost, se kterou je svázán proces, tj. při aktivaci této události se vyvolá proces.

- Každou instanci přechodu reprezentovat reálným procesem. Tento přístup odstraňuje nevýhody předchozího, ale přináší řadu omezení systému. Omezení se týkají především přenositelnosti objektů na jiný server (migrace objektů), uchování a znovu obnovení stavu simulace při přehrávání částí simulace, zpětný chod simulace apod.

[2] V rámci této práce se uvažuje zúžená varianta J-proveditelnosti, která ukončuje vyvolanou metodu ihned poté, co byl vložen objekt do místa *return*. Tento objekt se pak bere jako návratová hodnota metody.



a) b)

Obrázek 4.2: Vztah mezi třídou, vzorem a instancí

- Simulaci objektu reprezentovat reálným procesem. Nevýhodou je složitější návrh systému, protože zasílání zpráv musí být nahrazeno *delegováním*, tj. objekt po zaslání zprávy nečeká na výsledek (a tedy proces není suspendován), ale výsledek je doručen jinou cestou. Na druhou stranu tento přístup přináší plnou kontrolu nad simulačním objektem a řeší výše uvedené problémy. Navíc plně odpovídá událostem definovaným pro evoluci OOPN (4 typy proveditelnosti přechodu). Jak uvidíme z dalšího textu, tato koncepce přináší další výhody, které plně kompenzují náročnost návrhu. Pro návrh a implementaci interpretu OOPN byl zvolen právě tento přístup.

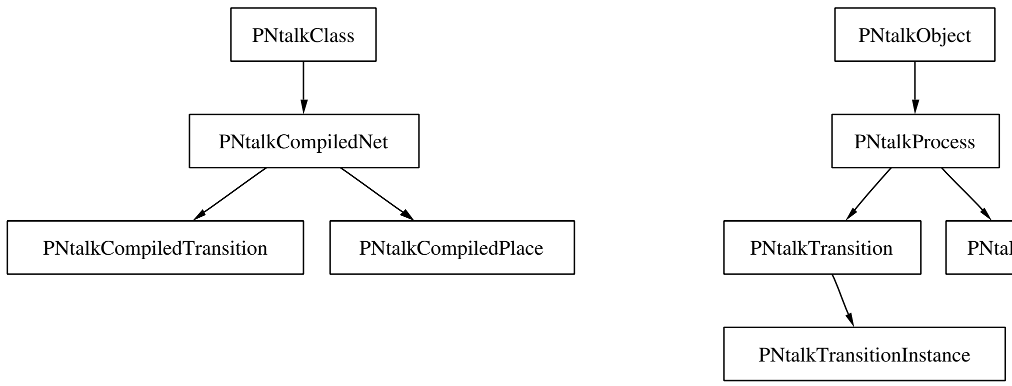
4.4.2 Reprezentace OOPN

Problém reprezentace OOPN může mít několik variant řešení. V zásadě můžeme rozlišovat dvě různé struktury – popis vzoru tříd a struktury reprezentující živé objekty během simulace. V první variantě použijeme jako vzor textový popis OOPN a při každém přístupu provádíme překlad a tvorbu struktur v dynamické části systému. Tento přístup je ovšem neefektivní. Další variantou je tento popis přeložit do nějaké vnitřní reprezentace a poté provádět simulaci nad touto reprezentací. Vnitřní reprezentaci nazveme statickou částí, struktury vzniklé při simulaci na základě statické části systému budeme nazývat dynamická část systému. V tomto okamžiku se můžeme inspirovat systémem Smalltalk, kde je statická část reprezentována objekty, které jsou chápány jako třídy, dynamickou část tvoří množina objektů chápaných jako instance tříd. Všechny objekty jsou uloženy v tzv. image (tzn. že jsou potenciálně perzistentní) přičemž textový popis je součástí každé třídy. Každá změna třídy je okamžitě zakompilována do image.

Nyní si uvedeme přehled sémantiky některých pojmů:

- *třída* je objekt, který definuje strukturu a chování jiného objektu,
- *instance třídy* je objekt, který vznikl jako instance třídy, tj. aplikací příslušných operací nad třídou (např. *new* v jazycích Simula nebo Smalltalk),
- *vzor* je objekt, který definuje strukturu a chování jiného objektu,
- *instance vzoru* je objekt, který vznikl aplikací příslušných operací nad vzorem.

Vztah mezi těmito pojmy ukazuje obrázek 4.2. Diagram tříd (obrázek 4.2a) zobrazuje dvě třídy (*A* a *B*), jeden vzor (*C*) a jednu instanci (*Object*) vzoru *C*, která je současně instancí třídy *B*. Instance třídy *B* musí umět poznat svůj vzor a všechny relevantní zprávy předat ke zpracování svému *vzoru*, nikoli třídě *B*. Třídy *A* a *B* nedefinují přímo chování vzoru a jeho instance, pouze implementují operace nutné pro zajištění očekávaného chování, tj. že objekt *Object* je instancí vzoru *C*. Třídy *A* a *B* jsou speciální třídy stejné pro libovolný vzor a všechny jeho instance. Můžeme tedy mluvit o *třídě vzoru* a *třídě*



Obrázek 4.3: Reprezentace OOPN

instance vzoru. Obrázek 4.2b zobrazuje posloupnost zasílání zpráv při vytváření instance třídy OOPN, tedy *instance vzoru*. Příslušnému vzoru je zaslána zpráva *new*, která iniciuje vznik nového objektu a poté vzor vrací referenci na nově vzniklý objekt. Tento objekt je instancí třídy *B* a reprezentuje *instanci vzoru C*.

Můžeme vidět, že uvedený koncept vyčlenění speciálních objektů pro OOPN třídy (vzory) a OOPN objekty (instance vzorů) odpovídá konceptu metaúrovňových architektur s dynamickou reflektivitou založenou na objektech. Vzor a instance vzoru představují *metaobjekty*, které reprezentují v systému *doménové objekty*, tj. třídy a objekty OOPN. OOPN jsou reprezentovány dvěma hierarchiemi metaobjektů,[3] jak je naznačeno na obrázku 4.3. V levé části je zobrazena struktura třídy OOPN (tak, jak je vygenerována překladem zdrojového kódu OOPN). Třída OOPN (*PNtalkClass*) je složena ze zkompilovaných metod, neboli sítí (*PNtalkCompiledNet*). Každá síť je složena ze zkompilovaných přechodů (*PNtalkCompiledTransition*) a míst (*PNtalkCompiledPlaces*). Reprezentace míst obsahuje struktury pro počáteční značení míst; reprezentace přechodů obsahuje struktury pro hrany, stráž a akce přechodu. Při vytvoření instance OOPN třídy vznikne odpovídající hierarchie živého objektu v systému (viz pravá část obrázku 4.3). Instance třídy OOPN (*PNtalkObject*) je složena z instancí sítí (*PNtalkProcess*), pokud je daná síť evokovaná. Každá instance obsahuje minimálně jednu objektovou síť. Instance sítě (tj. proces) je složena z reálných přechodů (*PNtalkTransition*) a míst (*PNtalkPlace*). Ty vzniknou jako instance zkompilovaných přechodů a míst během vytváření procesu. Při spuštění (*fire*) přechodu vznikne v reprezentaci jeho instance (*PNtalkTransitionInstance*). Nad těmito objekty pak probíhá evoluce OOPN.

4.4.3 Testování přechodů

Jednou z nejsložitějších částí interpretu OOPN je testování přechodů, tj. zjištění množiny potenciálních instancí přechodů. Instance přechodu pak představuje jeden stav přechodu po provedení jedné z událostí A, N nebo F (viz A,N,F-proveditelnost přechodu). Každý přechod může být svázán s místy (prostřednictvím *hranových výrazů*) a se synchronními porty objektů (prostřednictvím *zaslání zprávy objektu*). Výsledkem testování je seznam událostí určující přechody, které mohou být „provedeny”. Testování jednoho navázání probíhá ve dvou fázích. Nejprve se testují hrany, tzn. že se prohledává obsah příslušného místa a hledají se ty objekty, které vyhovují hranovému výrazu. Druhá fáze spočívá v testování stráže přechodu (*guard*). Ta má dva typy podmínek: operace porovnání primitivních objektů a volání synchronních portů. Pro úspěšné testování musí být pravdivé všechny podmínky.

[3] Podrobnější popis začlenění pojmu metaobjekt do architektury simulačního systému je uveden v kapitole 5.

- *Hranový výraz*. Hranový výraz je asociován s hranou mezi místem a přechodem a určuje, kolik objektů, případně jakých, se bude prostřednictvím této hrany testovat nebo přesouvat. Existují tři typy hran: *vstupní*, *testovací* (*vstupní bezpřívlastkové*) a *výstupní*. Z hlediska navazování přechodů jsou důležité první dva typy. Z hlediska testování přechodů není mezi nimi žádný rozdíl, při provedení přechodu se u vstupní hrany odeberou navázané objekty z místa, u testovací hrany tyto objekty v místě zůstávají.

- *Synchronní port*. Voláním synchronního portu objektu se zajišťuje synchronizace s daným objektem – synchronní port testuje pravdivost specifikované podmínky pro volaný objekt. Volání synchronního portu má charakter zaslání zprávy, zpráva může mít parametry. Volání synchronního portu může mít vedlejší efekt – pokud je jejím parametrem symbolické jméno, kterému nebyl v předchozích krocích testování navázán objekt, po úspěšném provedení synchronního portu (testovaná podmínka je pravdivá) se automaticky na toto symbolické jméno naváže přípustný objekt, vyhovující podmínce. Pokud takový objekt neexistuje, je testování synchronního portu neúspěšné.

Testování probíhá pro každý existující přechod (tím se rozumí přechod, který je součástí nějaké sítě, nikoliv aktivní přechod). Výsledkem testování je seznam přechodů, které mohou být aktivovány. Každý takový přechod může být navíc aktivován pro různá navázání proměnných. V dalším kroku evoluce se vybere jeden přechod s jedním jeho navázáním, který se provede (tzn. že se vytvoří instance přechodu pro vybrané navázání). Algoritmus navazování přechodů je podrobně popsán v [Koč01a].

4.5 Modelování a prototypování v OOPN

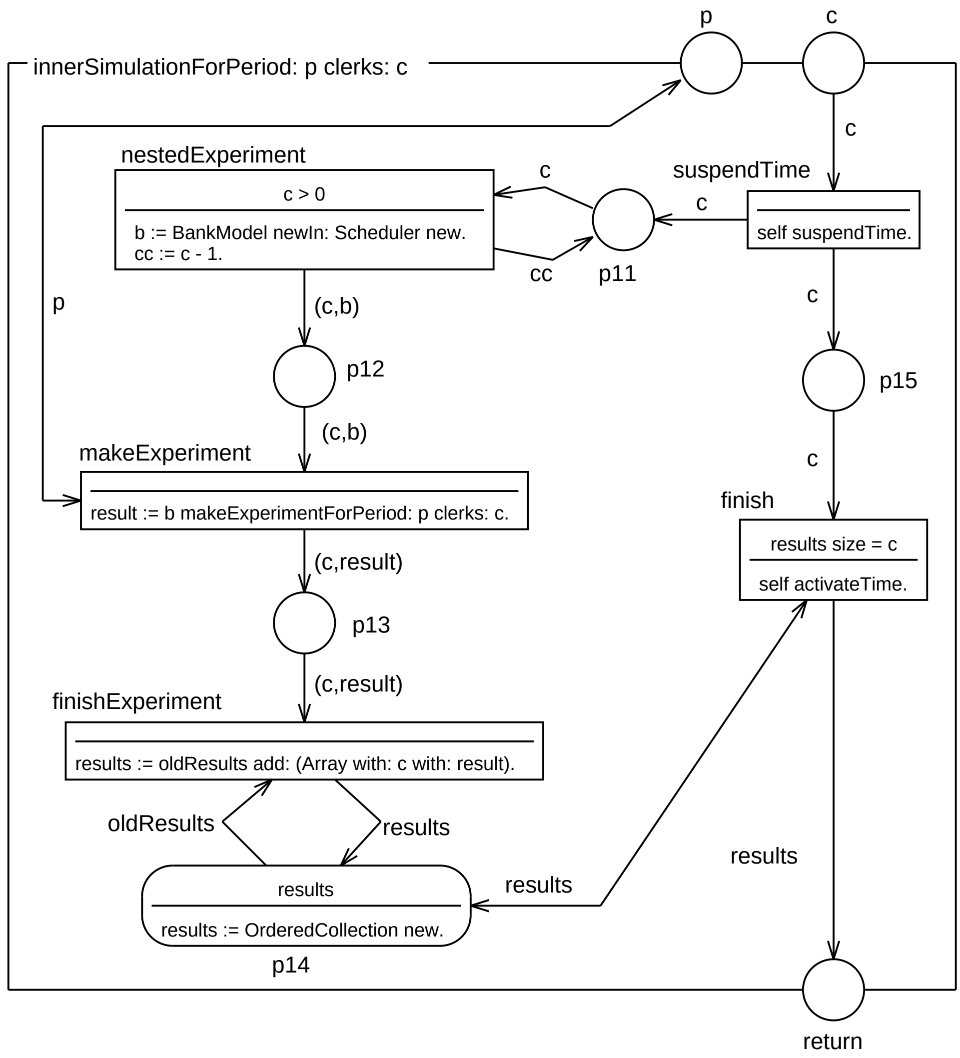
Objektově orientované Petriho sítě (OOPN) mohou být použity v mnoha oblastech informačních technologií, a to především v modelování paralelních systémů, jako součást vývojového procesu aplikací (skloubení formálního modelovaní a prototypování), pro aplikaci pokročilých simulačních technik apod. V této kapitole si přiblížíme podstatné rysy OOPN a jejich použitelnost.

4.5.1 Systém PNtalk

Objektově orientované Petriho sítě tvoří základ systému PNtalk. PNtalk je dlouhodobý projekt, který se zabývá modelováním, simulací, verifikací a prototypováním netriviálních programových systémů. Paradigma OOPN umožňuje využít spojení vlastností objektové orientace a Petriho sítí – tj. formální, hierarchický a objektově-orientovaný popis modelovaného problému. Jedním z mnoha cílů, které tento projekt sleduje, je vytvořit otevřený dynamický systém, který umožňuje za chodu modifikovat sám sebe, vyvíjet se a dynamicky se přizpůsobovat měnícím se požadavkům, využívat různá paradigmata a simulační techniky apod.

4.5.2 Modelování v systému PNtalk

Při modelování OOPN vytváříme třídy a metody podobně jako v klasických objektově orientovaných jazycích, ovšem s několika zásadními rozdíly. Každá metoda je popsána speciálními vysokoúrovňovými Petriho sítěmi (proto se zavádí spíše označení síť než metoda). Každá třída má jednu speciální nepojmenovanou síť (tzv. objektovou síť), jejíž instance vznikne v okamžiku vzniku objektu a je aktivní po celou dobu života objektu – jde tedy o aktivní objekt. Jelikož je metoda popsaná sítí OOPN, existuje paralelismus uvnitř metody, což je z hlediska klasického objektového přístupu věc nezvyklá. Na obrázku 4.4 je ukázka jedné sítě – metody se dvěma vstupními parametry. Oproti jiným objektovým Petriho sítím jsou formalismus OOPN a systém PNtalk ovlivněny objektovou orientací systému Smalltalk a umožňují vzájemnou interoperabilitu Petriho sítí a klasického objektově orientovaného přístupu Smalltalku. Na



Obrázek 4.4: Ukázka OOPN metody

obrázku 4.4 je vidět v místě *p14* volání *OrderedCollection new*, což je zaslání zprávy *new* standardní třídě Smalltalku.

Uvedený příklad je převzaný z článku [JK02], který ze zabývá využitím vybraných aspektů systému PNtalk. Prezentuje interoperabilitu OOPN se Smalltalkem a vnořenou simulaci. Na obrázku 4.4 je v přechodu *nestedExperiment* použito volání *BankModel newIn: Scheduler new*. *BankModel* je třída OOPN, jejíž metaobjekt rozumí metaoperaci *newIn:*, která způsobí vytvoření nové instance třídy *BankModel* a takto vzniklý objekt pak vloží do nového simulačního plánovače (viz *Scheduler new*). Naproti tomu v přechodu *makeExperiment* volání *b makeExperimentForPeriod: p clerks: c* je zaslání doménové zprávy objektu OOPN *b*. Koncept metaobjektů umožňuje mimo jiné i jednoduché (přímé) začlenění jiného paradigmatu do systému – rozhraní metaobjektu zůstává stejné, uvnitř může být interpretován jiný formalismus než OOPN. Přímé ovládání metaobjektu a jeho struktura umožňuje relativně jednoduchým způsobem ukládat živý objekt a znovu jej vkládat do světa (případně i jiného při migraci).

4.5.3 Objektově orientovaný návrh a prototypování

Výhodou objektově orientovaných přístupů k návrhu systému je především stabilita navrhovaných prvků a jednoduchost jejich změn. Klasické metodologie založené pouze na vnímání funkcí, které systém musí zajistit, jsou nestabilní z pohledu neustále se měnících požadavků uživatelů, zatímco metodologie založené na objektech vykazují větší míru odolnosti.[4] Proto se vývoj systémů zaměřuje na objektově orientovanou analýzu, návrh a také implementaci. V devadesátých letech bylo vyvinuto mnoho metodologií objektově orientované analýzy a návrhu. Každá z nich poskytuje různé metody a nástroje a definuje vlastní (různorodé) procesy tvorby systémů. Tento stav se ukázal jako nevyhovující, protože nezahrnuje komunikaci návrhářů, programátorů a ostatních zainteresovaných osob na vývoji aplikace. Proto se vyvíjela technologie, která by umožnila sjednotit jednotlivé procesy vývoje, pokud možno pro celou návrhovou etapu. Tato snaha vyústila v definici jazyka UML [UML04] (Unified Modelling Language) v roce 1995. UML ovšem sám o sobě nenabízí metodologii tvorby či návrhu systémů, ale nabízí pouze prostředky pro unifikovanou tvorbu modelů různých aspektů navrhovaného systému.

UML operuje s pojmem *pohled* (*view*). Pohled systému je projekce systému na jeden z jeho relevantních aspektů. Taková projekce se zaměřuje na příslušný aspekt a ignoruje ostatní. Jak jistě vyplývá z podstaty věci, je vhodné vytvářet několik různých pohledů na tentýž systém. Pohledy nad systémem jsou pak modelovány prostřednictvím vhodných nástrojů (modelů) poskytovaných UML. Můžeme rozlišit tyto základní pohledy:

- *Strukturální pohled (structural view)* popisuje vrstvu mezi objekty a třídami, jejich asociace a možné komunikační kanály.
- *Pohled chování (behavior view)* popisuje, jak systémové komponenty (objekty) interagují, a charakterizuje reakce na vnější systémové operace.
- *Datový pohled (data view)* popisuje stavy systémových komponent (objekty) a jejich vazby.
- *Pohled rozhraní (interface view)* je zaměřeno na zapouzdření systémových částí a jejich potenciální použití okolím systému.

Vlastní frazeologie UML nemá dostatečnou sílu, aby umožňovala fundamentálně popsat vztahy a pravidla, která jsou součástí každého modelovaného systému. Např. nemůžeme jednoznačně definovat v UML podmínku, že nejméně jedna položka musí být na zásobníku před voláním operace *pop*. Podobné situace se často řeší textovým komentářem k příslušnému modelu, čímž ovšem klesá formální úroveň modelu

[4] Objektově orientovaný návrh nutně neimplikuje implementaci navrhovaného systému v objektově orientovaném jazyce. Pro složitější systémy však přináší objektově orientovaný návrh značné výhody i v případech neobjektových implementací.

a vzniká otázka, zda to přispívá k lepší srozumitelnosti modelu. Další problémy vznikají rozdělením vývojového procesu na několik hlavních fází: analytickou, návrhovou, implementační a příp. prototypovou. Protože všechny modely cyklicky procházejí fázemi vytvoření, ověření a revidování, musí existovat spolupráce mezi jednotlivými fázemi návrhu a realizace systému. To s sebou přináší komplikace spojené především s komunikací. V současné době jsou tendence řešit výše uvedené problémy spojené s modelováním, návrhem a implementací systémů. Jednou z možných cest řešení některých problémů je použití Petriho sítí, resp. některé z jejich vysokoúrovňových variant, např. Objektově orientované Petriho sítě (OOPN).

Obecně se dá říci, že při použití grafických modelů (diagramů) vzniká problém porozumění pramenící právě z jejich grafické podstaty. Tento problém by se dal vyjádřit následovně: Krásná vlastnost techniky grafického popisu je, že jí rozumí každý. Špatná vlastnost je, že jí rozumí každý jinak. Tento problém u Petriho sítí (včetně OOPN) není. Grafické vyjádření má svou podstatu ve formální definici. OOPN lze použít pro modelování statických i dynamických aspektů systémů a díky svému formálnímu základu je možné provádět *přímo s těmito modely* experimenty (simulace), příp. verifikace vybraných částí modelu. Ukázka použití OOPN v procesu návrhu a prototypování, včetně analogie s UML, je uvedena v příloze C. Bližší diskuzi na dané téma a případovou studii OOPN modelování lze nalézt v [Koč01c, KV01].

Část II

Architektura otevřeného simulačního systému

Kapitola 5

Architektura systému

Tato kapitola se zabývá klíčovým aspektem disertační práce, tedy otevřenou architekturou systému pro simulaci a prototypování. Principy a význam otevřených implementací byly popsány v kapitole 3. Aplikováním těchto principů na systém pro podporu simulace vznikne nový simulační systém s vlastnostmi otevřených implementací. Nejprve si ukážeme obecnější pohled na architekturu takto koncipovaného simulačního systému a postupně budeme přecházet ke konkrétním aspektům a vazbám systému. Celá koncepce je od počátku navrhovaná pro potenciální práci v distribuovaném a heterogenním prostředí.

5.1 Neformální definice

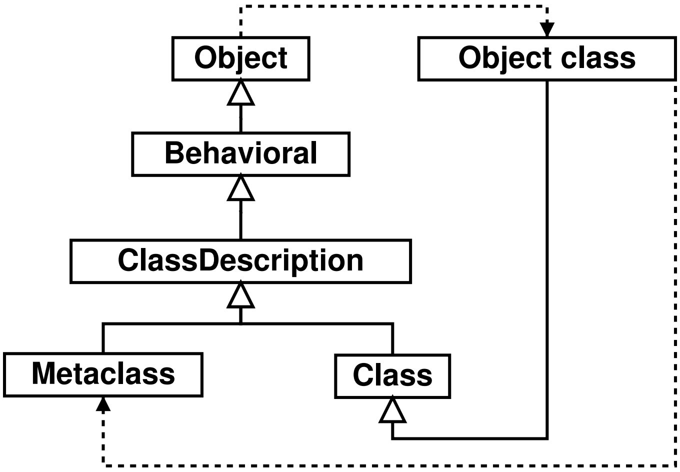
Abychom byli schopni definovat architekturu otevřeného simulačního systému, musíme si nejprve říci, jaké požadavky a vlastnosti na tento systém klademe a co od něj očekáváme. Jako výchozí doménové paradigma jsou uvažovány Objektově orientované Petriho sítě (OOPN), jejichž definice a význam jsou popsány v kapitole 4. Jak z ní vyplývá, doménový objekt OOPN je chápán jako *aktor*. V objektu existují procesy (procesy ve smyslu instance struktur – metod), které mohou být vzájemně nezávislé, a navíc mohou uvnitř procesů vznikat *podprocesy*. Evoluce doménových objektů je definovaná jako posloupnost událostí v jednotlivých objektech. Koncepce řízení a evoluce modelů představených OOPN je natolik obecná, že může být (s případnými modifikacemi) použita i na jiná paradigmata. V popisu otevřené architektury simulačního systému, uváděném v této kapitole, budeme proto uvažovat zobecněné principy evoluce a řízení objektů OOPN.

Velmi důležitým a podstatným rysem simulačního systému je právě jeho „otevřenost”, tedy vlastnost, která umožňuje vytvářet dynamické modely, provádět simulaci těchto modelů, a to včetně různých testovacích technik (např. modifikace rozpracovaného stavu a následný běh, ukládání živých objektů a jejich následné znovuoživení), vytvářet *prototypy* modelovaných aplikací prostým spojením modelu s jinými výrazovými prostředky apod. Požadované vlastnosti si můžeme shrnout do několika bodů:

- *Doménový objekt je aktor*. Protože primárně pracujeme s OOPN a princip evoluce či řízení objektů v systému je zobecněním prinicipů OOPN i na jiná paradigmata, musíme brát v úvahu, že každý objekt může mít svůj vlastní proces, a tedy může vyvíjet aktivitu bez nutnosti vnějších podnětů.

- *Nejmenší procesní jednotkou je vlákno*. Při vyvolání metody dojde k vytvoření její instance, tj. *procesu*. Součástí procesu mohou být podprocesy, které nazveme *vlákna*. Vlákno definuje jednoznačnou sekvenci výpočtu v procesu.

- *Simulace modelu.* Evoluce modelu může probíhat ve dvou fázích – plynulá simulace nebo krokování. Mezi těmito fázemi lze bez problému přecházet. Je možné ukládat stav simulace a poté se k tomuto uloženému stavu vracet.



Obrázek 5.1: Vztah tříd a metaobjektů ve Smalltalku

- *Možnost manipulace se stavem simulace objektů.* Jde především o možnost ukládat stav simulace, objektu či množiny objektů a jeho znovuoživení v zadaném kontextu. S tím souvisejí i možnosti migrace objektů v distribuovaném výpočetním prostředí. Touto problematikou se více zabývá kapitola 6.3.
- *Heterogenita*. Spojení různých paradigmat (neomezit se pouze na OOPN). Možnost přizpůsobení vyjadřovacích prostředků modelů či systému vlastním požadavkům. Tuto problematiku si více přiblížíme v kapitole 6.1.
- *Možnost aplikace různých simulačních technik.* Jako ukázku si můžeme uvést vnořenou simulaci, větvení simulace, klonování simulací či objektů atd. Touto problematikou se podrobněji zabývá kapitola 6.4.

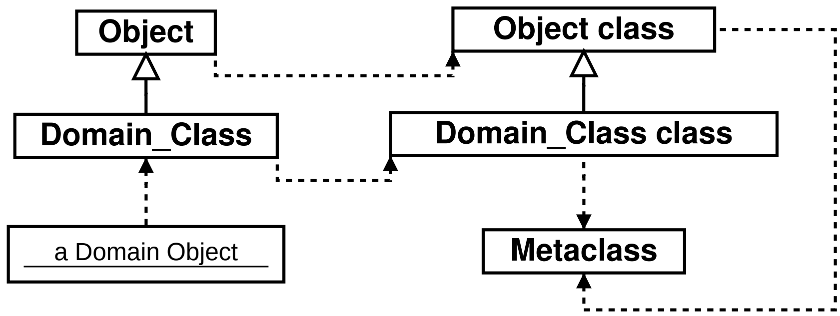
5.2 Úrovně systému

Objektově orientované Petriho sítě (OOPN) představují splynutí Petriho sítí (formální základ) a jazyka Smalltalk (objektová orientovanost). OOPN tvoří základní paradigma simulačního systému. Z požadavků uvedených v předchozí kapitole vyplývá, že tento systém musí poskytovat metainformace o doménových objektech, ale také (alespoň částečně) metainformace o systému samotném. Návrh simulačního systému je proto koncipován jako metaúrovňový (viz kapitola 3.3), jehož nejvyšší úroveň zapadá do metaúrovňové architektury systému Smalltalk [JK03, Koč03, KR03].

5.2.1 Klasifikace úrovní systému Smalltalk

Smalltalk chápe všechny své komponenty jako objekty, kde objekt je vždy instancí své třídy. To znamená, že i třída je instancí jiné třídy, která se nazývá metatřída. Všechny metatřídy jsou instancemi speciální metatřídy *Metaclass*. Vztah mezi těmito třídami je znázorněn na obrázku 5.1 (plná čára označuje vztah generalizace–specializace, tečkovaná vztah třída–instance; šipka označuje orientaci vztahu, např. *Object class* je instance třídy *Metaclass*). Smalltalk tedy zavádí jednu metaúroveň pro řízení chování tříd.

Uvedený koncept si přiblížíme na malém příkladu. Každý objekt ve Smalltalku rozumí zprávě *class*, která zjišťuje třídu daného objektu. Na obrázku 5.2 je znázorněna uživatelsky definovaná třída



Obrázek 5.2: Vztah objektů a tříd ve Smalltalku

Domain_Class a její instance *a Domain_Object*. Třída *Domain_Class* definuje chování objektu *a Domain_Object*. Třída *Domain_Class* je asociovaná se svou metatřídou *Domain_Class class* (je její instancí), která definuje chování třídy.

Souhrn: Objekt je řízen svou třídou, která určuje operace a vnitřní strukturu objektu. Třída je řízená svou metatřídou, která určuje chování třídy, tj. způsob zpracování přicházejících zpráv, způsob tvorby instancí apod.

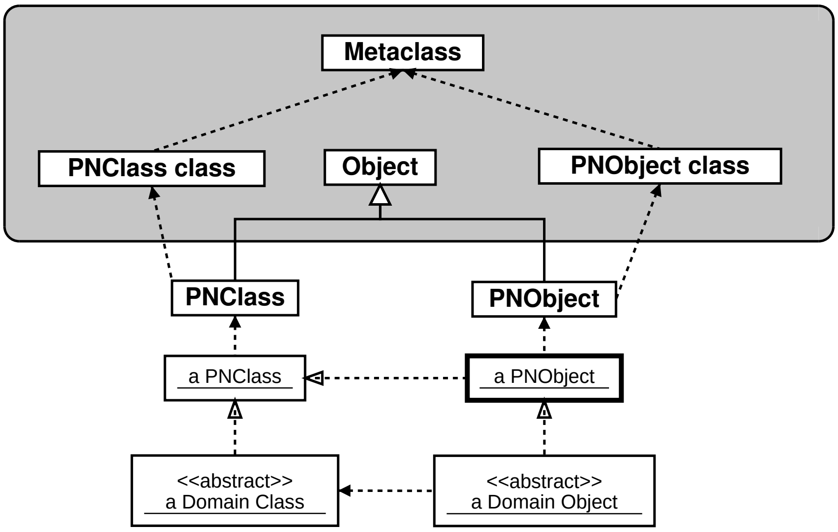
5.2.2 Klasifikace úrovní simulačního systému

Nyní si ukážeme, jakým způsobem zapadá základní koncept architektury otevřeného simulačního systému do prezentovaného metasystému Smalltalku. Na obrázku 5.3 reprezentují třídy *PNClass* a *PNObject* základní subjekty architektury. Instance třídy *PNClass* reprezentují doménové třídy OOPN a instance třídy *PNObject* reprezentují metaobjekty pro doménové objekty OOPN. Představme si nyní jednotlivé úrovně v systému a jejich význam (úrovně Smalltalku jsou uzavřeny ve vyplněném obdélníku):

Doménová úroveň (domain level): Tato úroveň je představována třídami OOPN a jejich instancemi (objekty). Z hlediska doménové úrovně jde o klasický objektově orientovaný přístup k tvorbě modelů. Doménové objekty a třídy jsou *konceptuální objekty*, se kterými se manipuluje při tvorbě modelů v doménovém paradigmatu (v našem případě OOPN). V simulačním systému však tyto objekty reálně neexistují a jsou zastoupeny svými *metaobjekty* (např. výsledkem překladu doménové třídy je metaobjekt *a PNClass*, který reprezentuje v systému danou doménovou třídu).

Metaúroveň (metalevel): Metaúroveň nad doménovou úrovní je reprezentovaná množinou instancí tříd *PNClass* (popisuje chování tříd OOPN) a *PNObject* (popisuje chování živých objektů, tj. instancí tříd OOPN). Tyto instance budeme obecně nazývat *metaobjekty*. Pro každý doménový objekt existuje právě jeden metaobjekt na metaúrovni (a naopak každý metaobjekt na metaúrovni reprezentuje právě jeden doménový objekt). Metaobjekt nad doménovým objektem je *aktor*. Jeho součástí je proces, který serializuje všechny přístupy a požadavky na objekt. Tento proces nazvěme *řídicí proces metaobjektu* (*metaobject control process*).

Meta metaúroveň (meta metalevel): Metaúroveň nad metaúrovní. Je reprezentována třídami *PNClass* a *PNObject* (tj. instancemi svých metatříd ve Smalltalku). Jak už bylo řečeno, systém předpokládá možnost změny doménového paradigmatu (tato možnost je diskutována v kapitole 6.1). V takovém případě budeme rozlišovat *typ doménového objektu*. Pro každý *typ doménového objektu* existuje právě jedna třída na meta metaúrovni. Pro daný typ doménového objektu jsou pak všechny metaobjekty doménových objektů instancemi příslušné třídy na meta metaúrovni.



Obrázek 5.3: Vztah mezi úrovněmi Smalltalku a simulačního systému

Architektura simulačního systému zavádí novou metaúroveň mezi doménovou třídou a objektem na jedné straně a Smalltalkem na straně druhé. Metaúrovňovou architekturu simulačního systému si můžeme klasifikovat následujícím způsobem (viz klasifikace reflektivity v kapitole 3.4):

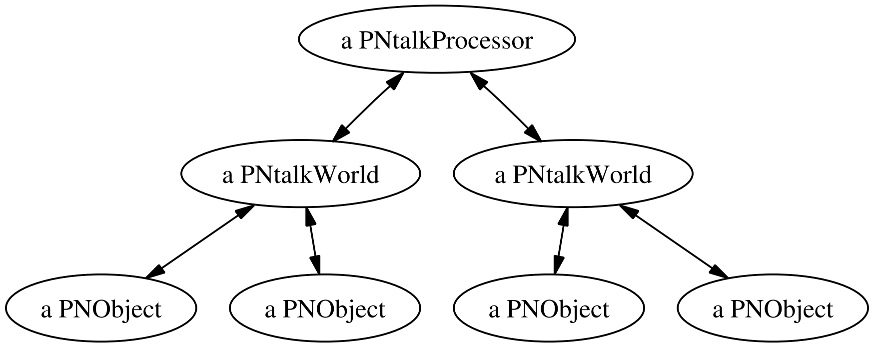
- Architektura kombinuje statickou a dynamickou reflektivitu. Veškeré operace mohou být kdykoliv během evoluce systému inkrementálně změněny a rekompilovány.
- Architektura kombinuje strukturální reflektivitu a reflektivitu chování (i když primárně se zaměřuje na reflektivitu chování).
- Architektura implementuje reflektivitu založenou na objektech. V kombinaci se Smalltalkem pak lze využít i reflektivitu založenou na třídách.

Doménové objekty jsou v systému zastupované svými metaobjekty. Metaobjekt řídí operace nad doménovým objektem na základě *řídicích zpráv*. Komunikace doménových objektů (zasílání *synchronních zpráv*) je adaptována na komunikaci metaobjektů prostřednictvím řídicích *asynchronních zpráv*. V kapitole 3.3.2 jsme definovali dvě kategorie operací, *doménové* a *metaobjektové*. Nyní si tyto definice rozšíříme.

1. *Doménová operace* (domain operations). Operace nabízené doménovým objektem z doménového protokolu. Nejsou dostupné přímo, ale prostřednictvím speciálního (*meta*) protokolu.
2. *Metaobjektová operace* (metaobject operations). Operace nabízené metaobjektem z metaobjektového protokolu. Jsou přímo dostupné, umožňují manipulaci jak s doménovým objektem, tak i metaobjektem. Mimo jiné implementují metody pro příjem, vyhledání a aktivace doménových operací (tj. implicitní metaobjektový protokol).

5.2.3 Hierarchie metaobjektů doménových objektů

Celý systém si můžeme rozdělit na dvě základní části: na část definující prostor pro doménové objekty a na část definující prostor pro doménové třídy. Doménový objekt i doménová třída jsou pak reprezentovány



Obrázek 5.4: Základní pohled na hierarchii dynamické části systému

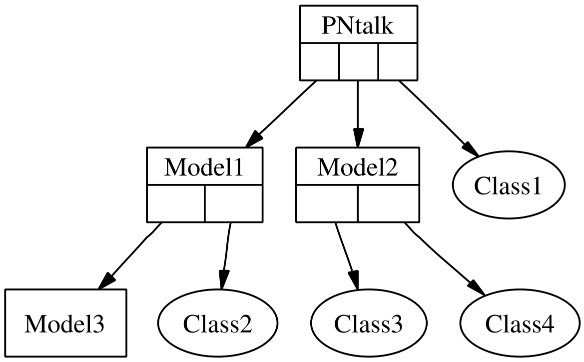
metaobjekty z příslušných částí. Základní popis hierarchie metaobjektů doménových tříd je uveden v kapitole 5.2.4. V této kapitole se budeme věnovat hierarchii metaobjektů doménových objektů.

Jednotlivé metaobjekty reprezentují přesně specifikovanou část a funkcionalitu systému. Na obrázku 5.4 můžeme vidět tři základní vrstvy (*layers*) metaobjektů v systému a jejich vzájemný vztah:

- Živé objekty (metaobjekty *PNObject*) představují doménové objekty v systému. Každý metaobjekt *PNObject* obsahuje referenci na metaobjekt korespondující doménové třídy.
- Světy (metaobjekty *PNtalkWorld*) zapouzdřují množinu objektů (*PNObject*) tvořících sémanticky uzavřený celek. Klíčovým aspektem světa je jedinečná a nezávislá časová osa. Z hlediska simulace tedy svět zapouzdřuje nezávislou simulaci.
- Koordinační uzel (metaobjekt *PNtalkProcessor*) zapouzdřuje celý systém. Uzel provádí koordinaci světů a reprezentuje danou část v potenciálně distribuovaném prostředí.

Všechny výše uvedené metaobjekty jsou zařazeny do metaúrovňového systému Smalltalk stejným způsobem, jak je naznačeno v kapitole 5.2.2, včetně jejich aktivity. Jediným rozdílem je složení metaobjektů a jejich protokol – pouze metaobjekty reprezentující živé (doménové) objekty jsou schopny obsluhovat chování doménového objektu. Protokol metaobjektů reprezentujících dynamickou část simulačního systému je uveden v příloze D.2. V systému existují ještě další typy metaobjektů, které reprezentují dílčí komponenty objektů a nemají definovanou aktivitu. Metaobjekty tedy můžeme rozlišit na následující typy:

- *Slave*. Jedná se o metaobjekt, který reprezentuje dílčí část objektu (např. *proces*, *přechod* atd.) Tento metaobjekt *není* aktor a umožňuje zpracovávat pouze *synchronní zprávy*.
- *Master*. Jedná se o metaobjekt, který reprezentuje příslušný objekt. Tento metaobjekt *je* aktor a umožňuje zpracovávat *synchronní* i *asynchronní zprávy*. Metaobjekty *a_PNObject*, *a_PNtalkWorld* a *a_PNtalkProcessor*, uvedené na obrázku 5.4, jsou objekty typu *master*.
- *Doménový metaobjekt*. Jedná se o speciální metaobjekt typu *master*, který reprezentuje doménový objekt. Definuje řídicí operace nutné pro ovládání evoluce doménového objektu, včetně implicitního metaobjektového protokolu. Na obrázku 5.4 je doménový metaobjekt reprezentován metaobjektem *a_PNObject*.



Obrázek 5.5: Základní pohled na hierarchii statické části systému

5.2.4 Hierarchie metaobjektů doménových tříd

Koncepce hierarchie metaobjektů v části systému popisující doménové třídy je podobná koncepci hierarchie tříd ve Smalltalku. Lze ji charakterizovat jako strom, jehož uzly mohou být buď metaobjekty reprezentující doménové třídy, nebo speciální metaobjekty reprezentující prostory jmen. Kořenový uzel reprezentuje prostor jmen, který má název *PNtalk*. V systému pak rozlišujeme relativní jméno třídy a absolutní jméno třídy. Relativní jméno je vždy vztaženo k prostoru jmen, ve kterém se nachází kontext přístupu k danému jménu (tzn. že záleží na tom, z kterého prostoru jmen se přistupuje k danému jménu třídy). Absolutní jméno je dáno plně kvalifikovaným jménem, které obsahuje sekvenci prostorů jmen, začínající prostorem s názvem *PNtalk*. Ukázkový příklad hierarchie je uveden na obrázku 5.5 (obdélníky reprezentují prostory jmen, elipsy metaobjekty doménových tříd). Plně kvalifikované (absolutní) jméno třídy *Class3* je *PNtalk.Model2.Class3*. Podle rozlišení metaobjektů, uvedených v předchozí kapitole, jsou všechny metaobjekty doménových tříd typu *slave*. Protokol metaobjektů reprezentujících doménové třídy je uveden v příloze D.2.

5.3 Reprezentace doménového objektu

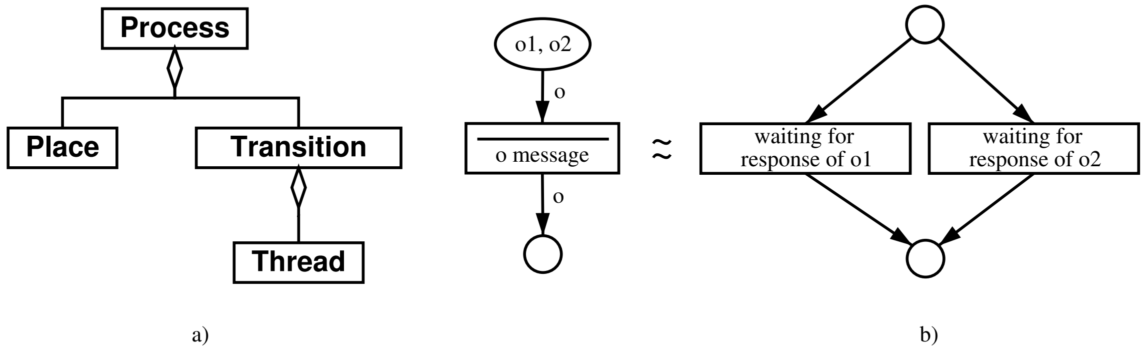
V této kapitole se budeme věnovat struktuře základní vrstvy architektury otevřeného simulačního systému – doménového objektu. Pro jeho popis formálně zavedeme několik základních pojmů a struktur.[1] Obsahem struktur jsou většinou seznamy jiných metaobjektů. Prezentované struktury předpokládají, že každý objekt existuje v systému pouze jednou a všechny přístupy k nim (tj. i prvky seznamů) jsou realizovány prostřednictvím *referencí*. V dalším textu se proto bude používat pojem *reference na objekt*.

5.3.1 Vlákno

Základní úrovní při evoluci[2] modelu je *vlákno*, které je součástí *procesu*. Princip procesů a vláken vychází z předpokladu, že metoda tvoří vzor pro instanciaci nějaké struktury (tj. sítě OOPN, kódu ve Smalltalku apod.) Instance této struktury, tedy *proces*, pak může být složena z více podprocesů, tedy *vláken*. Vlákno můžeme považovat za sekvenci kauzálně svázaných událostí, kdy jedna událost je následována právě

[1] Tyto struktury jsou reprezentovány instancemi tříd z meta metaúrovně simulačního systému. Struktura je zapisována ve tvaru $name = (item1, item2, \ldots)$, kde $name$ pak odpovídá jménu třídy a $item1, item2, \ldots$ jsou její instanční proměnné. K uváděným strukturám se vážou operace nad danými strukturami, které jsou demonstrovány formou metod v kapitole 5.4.

[2] Evolucí modelu budeme rozumět běh (či simulaci) modelu a evolučním krokem budeme rozumět posun v simulaci modelu.



Obrázek 5.6: Procesy a vlákna v OOPN

jednou jinou událostí. V pojetí OOPN je vlákno chápáno jako *instance přechodu*, tj. přechod popisuje chování vlákna. Událost je tedy vždy svázána s přechodem nebo vláknem a provedení události může mít za následek posunutí evoluce vlákna o jeden krok, vytvoření vlákna nového nebo ukončení existujícího vlákna.[3] V OOPN můžeme rozlišit tyto čtyři základní události:

- *Událost „provedení přechodu"*. Při této události vznikne *instance* přechodu. Tím vzniká nové vlákno procesu, které reprezentuje instanci přechodu.
- *Událost „zaslání zprávy"*. Instance přechodu (vlákno) zašle zprávu objektu.
- *Událost „ukončení zprávy"*. Instance přechodu (vlákno) vyzvedne návratovou hodnotu zprávy.
- *Událost „ukončení přechodu"*. Provádění přechodu je ukončeno, instance přechodu (vlákno) je zrušena.

Vlákno tedy vždy provádí události *„zaslání zprávy"*. Pokud je ve vlákně více operací, pak je vlákno po provedení události ve stavu *„čekání na výsledek operace"*. V okamžiku dokončení volané operace se vlákno uvolní a může pokračovat ve své evoluci, tj. provést událost *„ukončení zprávy"*. Po provedení této události se může vyskytnout jedna z událostí *„zaslání zprávy"* nebo *„ukončení přechodu"*. Událost *„zaslání zprávy"* znamená, že ve vlákně existuje další operace, událost *„ukončení přechodu"* znamená, že ve vákně již neexistuje další operace a vlákno se ukončí.

Princip vláken názorně prezentuje obrázek 5.6. V části (b) obrázku 5.6 je zobrazena jednoduchá síť s jedním přechodem, jedním vstupním místem a jedním výstupním místem. Vstupní místo obsahuje dva objekty *o1* a *o2*, akce přechodu je zaslání zprávy *message* objektům ze vstupního místa. Při evoluci je přechod sítě rozvinut do dvou vláken pro objekt *o1* a *o2* (obrázek ukazuje stav, kdy obě vlákna čekají na dokončení volané operace v objektech).

5.3.2 Systémový proces

Na obrázku 5.6a můžeme vidět hierarchii tříd metaobjektů tvořících systémový proces simulačního systému. Proces je sestaven z míst a přechodů, přechod může obsahovat své *instance* (*vlákna*). S každým procesem se váže implementace použitého paradigmatu, tj. *interpret*. Interpret provádí interpretaci paradigmatu, generuje události, provádí vybrané události atd. Proces je pak tvořen množinou vláken a množinou událostí, které jsou potenciálně proveditelné. Nyní si definujeme systémový proces jako komponentu na meta metaúrovni.

[3] V reakci na provedení události se vyvolá obsluha události, tj. příslušná metoda metaobjektu.

Systémový proces *je struktura*

$$P = (Calendar, Threads, Events)$$

kde

- $Calendar$ *– seznam aktivačních záznamů událostí,*[4]
- $Threads$ *– seznam vláken procesu,*
- $Events$ *– seznam aktuálně proveditelných událostí procesu.*

Třídu všech systémových procesů označíme Π.

◁ *5.3.1*

Takto specifikovaný systémový proces P je obecný. Každé použité paradigma má definovaný svůj vlastní proces, který musí splňovat požadavky kladené na obecný proces (základní struktura a operace nad strukturou). Ostatní části pak reflektují požadavky použitého paradigmatu a jeho interpretu. Můžeme tedy rozlišovat *typ systémového procesu*. Nyní si uvedeme definici systémového procesu OOPN.

Systémový proces OOPN *je struktura*

$$PNtalkNet = (Calendar, Threads, Events, Transitions, Places)$$

kde

- $Calendar, Threads$ *a* $Events$ *– mají stejný význam jako ve specifikaci 5.3.1,*
- $Transitions$ *– seznam přechodů sítě procesu,*
- $Places$ *– seznam míst sítě procesu.*

◁ *5.3.2*

5.3.3 Procesní vazba

Doménové objekty navzájem komunikují prostřednictvím zasílání zpráv. Pokud nějaký objekt (*sender*) zašle zprávu jinému objektu (*receiver*), objekt *receiver* zprávu zpracuje a předá zpět zdrojovému objektu *sender* návratovou hodnotu. Objekt *sender* přitom čeká na dokončení vyvolané metody. Zpracování vyvolané metody chápeme jako proces, který je tvořen množinou vláken. V každém procesu v daném čase existuje konečná množina událostí. Evoluce modelů simulačního systému je založena na komunikaci doménových metaobjektů prostřednictvím asynchronních zpráv. Synchronní komunikace doménových objektů je tedy konvertována na asynchronní komunikaci příslušných metaobjektů.

Uvažujme nyní následující situaci. Mějme objekty A a B, uvažujme libovolný proces P_A objektu A a událost e_A jako událost ve vlákně t_A procesu P_A. Událost e_A představuje zaslání zprávy $message$ objektu B. Ten na základě zprávy vytvoří proces P_B a začne provádět jeho evoluci. Vlivem události e_A dojde k suspendování vlákna t_A v procesu P_A, které čeká na dokončení vyvolaného procesu P_B. Můžeme si tedy definovat vazbu závislosti mezi vláknem t_A a procesem P_B. Pokud zobecníme náš myšlenkový postup na obecnou událost, vlákno a proces, můžeme definovat *procesní vazbu* jako dvojici $(thread, process)$, kde $thread$ reprezentuje závislé vláko a $process$ reprezentuje vyvolaný proces.

Klíčem pro uchování procesní vazby v doménovém metaobjektu je speciální metaobjektu typu slave – *systémová zpráva*. Metaobjekt *systémová zpráva* zajišťuje potřebné informace nutné pro korektní zpracování asynchronní zprávy mezi metaobjekty. Systémová zpráva je v rámci systému jedinečná (má jedinečnou identifikaci).

[4]Význam kalendáře a práce s časem jsou popsány k kapitole 5.5.5.

Systémová zpráva *je struktura*

$$PNtalkMessage = (message, args, receiver, sender, type)$$

kde

- $message$ *– řetězec identifikující zprávu,*
- $args$ *– seznam argumentů zprávy,*
- $receiver$ *– reference na příjemce zprávy,*
- $sender$ *– reference na odesílatele zprávy,*
- $type \in \{domain, meta\}$ *– typ zprávy (doménová nebo metaobjektová zpráva).*

$\triangleleft$ *5.3.3*

Procesní vazba je v doménovém metaobjektu uložena prostřednictvím dvou seznamů – *seznamu závislých vláken* a *seznamu vyvolaných procesů* (viz $depT$ a $invP$ ve specifikaci 5.3.4). Prvkem seznamu závislých vláken je dvojice $(message, thread)$, kde $message$ je systémová zpráva, která reprezentuje zprávu zaslanou z vlákna $thread$. Prvkem seznamu vyvolaných procesů je dvojice $(message, process)$, kde $message$ je systémová zpráva, která byla přijata doménovým metaobjektem a $process$ je proces vyvolaný na základě přijaté zprávy.

5.3.4 Synchronizace zpráv

Jak již bylo naznačeno, komunikace doménových objektů je konvertována v systému jako asynchronní komunikace metaobjektů. Podle použitých operací z různých protokolů (viz kategorie protokolů, kapitoly 3.3.2 a 3.5) můžeme odlišit tyto varianty komunikace:

- *doménový protokol* (*doménový objekt* $\Leftrightarrow$ *doménový objekt*) – synchronní komunikace,
- *implicitní metaobjektový protokol* (*doménový metaobjekt* $\Leftrightarrow$ *doménový metaobjekt*) – asynchronní komunikace,
- *vnitřní metaobjektový protokol* (*metaobjekt* $\Leftrightarrow$ *metaobjekt*) – synchronní i asynchronní komunikace (podle implementace operace),
- *explicitní metaobjektový protokol* (*doménový objekt* $\Leftrightarrow$ *metaobjekt*) – synchronní komunikace.

Synchronnost zprávy záleží na charakteru přenosového kanálu a na konečné implementaci metody, která tvoří vzor procesu vytvořeného na základě přijaté zprávy. Pokud komunikujeme s objekty na doménové úrovni[5] jde o synchronní komunikaci, na metaúrovni o asynchronní komunikaci. Operace na metaúrovni většinou manipulují s interními daty metaobjektů, a proto musí být zajištěna synchronizace těchto operací, nejlépe prostřednictvím vzájemného vyloučení. Způsob synchronizace metaobjektových operací nad metaobjektem si rozdělíme na dvě kategorie:

- *implicitní*. Implicitní synchronizace je implementována jako vzájemné vyloučení (mutual exclusion) jednotlivých *obsluh událostí* v objektu a vnějších *asynchronních zpráv*. Každá událost v objektu může být provedena pouze na základě speciální asynchronní řídicí zprávy. Pro zajištění vzájemného vyloučení byl proto zvolen způsob serializace příchozích *asynchronních* zpráv, které jsou řazeny do fronty. Metaobjekt master obsahuje řídicí proces, který neustále vybírá asynchronní zprávy z fronty a zpracovává je.

[5] Na doménové úrovni uvnitř systému. Je možné spojit objekty systému s objekty okolí systému (tedy spojit doménovou úroveň). Tento případ je specifický a je popsán v kapitole 5.6.

- *explicitní*. Explicitní synchronizace není přímo implementována a je záležitostí každé metody objektu typu master nebo slave, jakým způsobem ji zajistí.

Na základě variant komunikace mezi metaobjekty a způsobu synchronizace můžeme definovat následující typy zpráv, které se mohou v systému objevit. Toto rozdělení je podstatné v dalším textu, kde se prezentují algoritmy a procesy řízení objektů prostřednictvím kódu příslušných metod.

- *synchronní*. Synchronní zprávy jsou takové zprávy, které jsou přijímány přímo objektem master, případně objektem slave.
- *asychronní* (*delegované*). Delegované zprávy jsou takové zprávy, které jsou serializovány přes frontu zpráv (viz *implicitní synchronizace*). Doménové zprávy jsou delegovány prostřednictvím speciálních synchronních zpráv metaobjektu master. Řídicí zprávy jsou implicitně chápány jako synchronní, mohou však být delegovány tak, že příslušná synchronní metoda vytvoří odpovídající strukturu systémové zprávy (viz specifikace 5.3.3) a vloží ji do fronty zpráv.
- *interní*. Interní zprávy jsou zprávy vybírané z fronty zpráv a zasílané v rámci interních metaobjektů typu slave metaobjektu master. Interní zprávy se vykonávají vždy v kontextu řídicího procesu metaobjektu master.

5.3.5 Struktura doménového metaobjektu

Na tomto místě si shrneme dosud popisované entity a atributy metaobjektů a na jejich základě definujeme strukturu doménového metaobjektu. Každému doménovému objektu přísluší množina metaobjektů v systému. Vždy existuje metaobjekt, který reprezentuje doménový objekt svému okolí a který zapouzdřuje další metaobjekty, na které lze delegovat operace příslušející oblasti správy daného metaobjektu. Metaobjekt, který reprezentuje objekt svému okolí, jsme nazvali objekt typu *master*, ostatní metaobjekty jsou typu *slave*.

Doménový metaobjekt *je struktura*

$$PNObject = (Queue,\, invP,\, depT,\, class,\, LP,\, Processes)$$

kde

- $Queue$ *– fronta delegovaných zpráv,*
- $invP$ *– seznam vyvolaných procesů,*[6]
- $depT$ *– seznam závislých vláken,*[6]
- $class$ *– reference na třídu doménového objektu,*
- $LP \in \Pi$ *– životní proces objektu,*
- $Processes \subset \Pi$ *– seznam procesů doménového objektu.*

Třídu všech doménových metaobjektů označíme Θ.

◁ *5.3.4*

[6] Viz procesní vazba, kapitola 5.3.3.

5.4 Zpracování asynchronní zprávy

V této kapitole si představíme princip zpracování asynchronních zpráv *doménovým metaobjektem*. Asynchronní zprávou může být doménová zpráva z doménového protokolu, která je přijatá metaobjektem prostřednictvím speciální synchronní řídicí zprávy, nebo asynchronní řídicí zpráva z metaobjektového protokolu. Nejprve si definujeme jednotlivé kroky a poté je popíšeme pomocí kódu ve Smalltalku. Tento přístup s sebou přináší dvě výhody: dobře zapsaný algoritmus v jazyce Smalltalk je dobře a intuitivně pochopitelný, a navíc může být přímo aplikován. Uvedené ukázky jsou převzaty z prototypové implementace simulačního systému [PNt04]. V příloze B je popis syntaxe a speciálních symbolů používaných v zápise kódu pro ty čtenáře, kteří nejsou seznámeni s jazykem Smalltalk. Uvedené části kódu mají pouze popisný charakter, proto neuvádíme plnou definici tříd. Pro lepší přehlednost jsou instanční proměnné (tj. proměnné objektu) v kódu metod zvýrazněny. Uvažované metody popisují operace nad základními třídami systémových komponent, viz specifikace 5.3.2, 5.3.3 a 5.3.4.

Před každou metodou je v závorce uveden symbol, který určuje typ zprávy (viz kapitola 5.3.4): synchronní (S), asynchronní nebo také delegovaná (D) a interní (I). V případě doménového metaobjektu je na několika místech zmíněn speciální metaobjekt *slave*, který je přímo podřízen doménovému metaobjektu. Mezi tyto dva metaobjekty jsou rozděleny operace nutné pro zpracování zpráv. *Doménový metaobjekt* přijímá a zpracovává synchronní zprávy (S). Prostřednictvím synchronních zpráv také přijímá asynchronní zprávy. *Metaobjekt slave* zpracovává asynchronní zprávy (D) a interní zprávy (I).

5.4.1 Zprávy atomické v čase

Tyto zprávy zaujímají zvláštní postavení při jejich zpracování. Jde o takové zprávy, které musí být provedeny atomicky v modelovém čase, tzn. že modelový čas se nemůže posunout během jejich zpracování. Jejich základní využití je v komunikaci modelu s těmi částmi modelu, které nemusejí být popsány Objektově orientovanými Petriho sítěmi. Zpracování zpráv v těchto objektech není zahrnuto do časového evolučního cyklu simulačního systému (viz kapitola 5.5.5). Architektura nabízí prostředky pro zajištění správné interpretace atomických zpráv, a to jednoduchou modifikací původních algoritmů. Tato modifikace spočívá v označení atomických zpráv (viz metoda 5.4.7) a kontrole počtu právě zpracovávaných atomických zpráv. Pokud právě existuje nějaká atomická zpráva, pak je blokován posun času (viz metoda 5.5.15).

5.4.2 Fáze zpracování zprávy

Asynchronní zpráva může být zpracována v několika krocích (tj. asynchronní zpráva *nemusí* být atomická). Z toho vyplývá nutnost mít mechanizmy pro dočasné uložení rozpracované metody odpovídající volané zprávě, její znovu zavedení a vykonání následujícího kroku. Proces zpracování asynchronní zprávy můžeme rozdělit do několika kroků:

1. *Příjem zprávy*. Zpráva je přijata doménovým metaobjektem a zařazena do fronty. Zpráva je konvertována do struktury *PNtalkMessage* (tj. je vytvořena instance třídy *PNtalkMessage*, viz specifikace 5.3.3). Přímý výsledek nositelské zprávy je vytvořená struktura *PNtalkMessage.*

2. *Aktivace kontextu metody*. Řídicí proces (*metaobject control process*) doménového metaobjektu řídí vlastní zpracování delegovaných zpráv. Je aktivován přítomností zprávy ve frontě. Po aktivaci zprávu vyzvedne z fronty, vytvoří kontext metody a začne vlastní zpracovávání zprávy. Vyvolaná metoda je vždy zpracována v nějakém kontextu metody. Kontext metody zajišťuje uchování informací o běžící metodě v případě, že její evoluce není atomická. Podle typu zprávy a paradigmatu, které jsou použity pro popis metody, je pak vytvořen příslušný kontext metody.

   (a) *Doménová zpráva*. Kontext metody je vytvořen jako instance třídy *PNtalkNet*,[7] která je asociovaná s doménovou metodou objektu. Řízení je předáno přímo kontextu metody tak, že se provede jeho aktivace.
   (b) *Řídicí zpráva*. Kontext metody je vytvořen jako struktura *PNtalkMethodContext*. V rámci tohoto kontextu je pak předáno řízení speciálnímu metaobjektu *slave* tak, že se vyvolá příslušná metoda tohoto objektu slave. Asynchronní (delegované) řídicí zprávy jsou tedy implementovány jako metody speciálního metaobjektu slave.

3. *Zpracování kontextu metody*. Zpracování kontextu metody záleží na typu kontextu. Jak vyplývá z předchozího bodu, můžeme v základní architektuře rozlišit tyto dva typy:
   (a) *PNtalkMethodContext*. Aktivovaná zpráva je zpracována standardním kódem Smalltalku v metodě speciálního metaobjektu *slave*.
   (b) *PNtalkNet*. Aktivovaná zpráva je zpracována interpretem doménového paradigmatu (v našem případě OOPN).

4. *Ukončení kontextu metody*. Po zpracování události v kontextu metody může být kontext ve dvou stavech:
   (a) *čekající*. Čekající znamená, že metoda není atomická a čeká na provedení dalšího kroku. V tomto případě se uloží kontext metody do seznamu invokovaných metod.
   (b) *ukončený*. Evoluce metody byla ukončena. Po ukončení metody je nutné přenést vysílajícímu objektu přímý výsledek zprávy.

Zvláštní kapitolu tvoří *zasílání zpráv* z běžícího kontextu metody. Zpráva je vždy zasílána z vlákna, které je součástí kontextu metody. Zaslání zprávy způsobí registraci vlákna v procesní vazbě, viz kapitola 5.4.6.

5.4.3 Příjem zprávy

Doménový metaobjekt rozlišuje dvě metody pro příjem asynchronních zpráv. Jsou to metody *performMessage:* a *performDomainMessage:*. První slouží pro příjem zpráv z metaprotokolu, druhá pro příjem zpráv z doménového protokolu. Argumentem obou metod je systémová zpráva PNtalku (viz specifikace 5.3.3). V případě doménové zprávy je nastaven příznak (*msg domain*) a poté je předána k zpracování metodě *performMessage:*.

***(S)* PNObject » performDomainMessage: msg**
    msg domain.
    ↑ *self performMessage: msg.*

◁ *5.4.1*

Metoda *performMessage:* nastaví příjemce zprávy a uloží zprávu do vstupní fronty (viz instanční proměnná *inputQueue*). Vrací objekt systémové zprávy, která byla přijata. Tento objekt pak slouží jako informační zdroj pro uchování procesní vazby.

***(S)* PNObject » performMessage: msg**
    msg receiver: self.
    inputQueue *nextPut: msg.*
    ↑ *msg.*

◁ *5.4.2*

[7] V obecné rovině se jedná o instanci procesu (viz definice 5.3.1), tj. pro doménovou zprávu je $kontext\ metody \in \Pi$. V této části se omezíme na použití OOPN s tím, že veškerý protokol je uniformní a použitelný beze změn i pro jiná paradigmata. Pouze v reprezentaci procesu (v našem případě *PNtalkNet*) dojde ke změně vnitřní implementace (protokol zůstává opět stejný).

5.4.4 Aktivace kontextu metody

Řídicí proces doménového metaobjektu předá po vyzvednutí zprávy z fronty řízení metodě *processMessage:*. Ta vytvoří kontext metody asociované s přijatou zprávou a předá řízení speciálnímu metaobjektu *slave* (viz instanční proměnná *slave* v kódu metody) pro zpracování zprávy ve vytvořeném kontextu. Po návratu řízení zpět je kontext metody buď ukončen (*methodContext finish: return*), nebo uložen do seznamu invokovaných metod (*listInvocatedMethods*). Tento seznam reprezentuje seznam vyvolaných procesů $invP$ (viz specifikace 5.3.4). Po vytvoření kontextu metody je tento kontext zařazen do seznamu existujících procesů v objektu (*listProcesses add: methodContext*).

***(I)* PNObject » processMessage: msg**
| *methodContext return* |
methodContext ← *msg getMethodContextForObject: self.*
return ← slave *performMessage: msg in: methodContext.*
listProcesses *add: methodContext.*
methodContext isWaiting
ifFalse: [*methodContext finish: return*]
ifTrue: [listInvocatedMethods *at: (methodContext message) put: methodContext*].
◁ *5.4.3*

Proces aktivace kontextu metody získá příslušný kontext tak, že o něj požádá zpracovávanou zprávu (*msg getMethodContextForObject: self*). Ta na základě svého interního stavu vrátí příslušný kontext (tj. *PNtalkMethodContext* nebo *PNtalkNet*). Interním stavem se v tomto případě rozumí fakt, zda je zpráva doménová či nikoliv. Pokud je doménová, zjistí se metaobjekt reprezentující doménovou třídu objektu a požádá se o vrácení reprezentace procesu, viz podmíněný výraz v části *ifTrue:*. Pokud zpráva není doménová, vytvoří se přímo kontext zprávy, viz podmíněný výraz v části *ifFalse:*.

***(I)* PNtalkMessage » getMethodContextForObject: obj**
↑ *self isDomain*
ifTrue: [*obj yourClass performMessage: self forObject: obj*]
ifFalse: [*PNtalkMethodContext forObject: obj*].
◁ *5.4.4*

5.4.5 Zpracování kontextu metody

Po získání kontextu je řízení předáno speciálnímu metaobjektu *slave*, který podle typu zprávy předá řízení příslušné metodě příslušného metaobjektu. Pokud je zpráva doménová (*msg isDomain*) zavolá se speciální řídicí metoda *performDomainMessage:*. Pokud zpráva není doménová, jedná se o asynchronní zprávu z metaobjektového protokolu, která bude zpracována metaobjektem *slave*. V tomto případě se aktivuje metoda metaobjektu *slave*, jejíž jméno koresponduje se zpracovávanou zprávou (zaslání zprávy *perform:withArguments:* z protokolu základní třídy Smalltalku *Object*). Kontext metody je po celou dobu zpracování příslušné asynchronní zprávy uložen v instanční proměnné *methodContext*.

***(I)* PNObject (Slave) » performMessage: msg in: mc**
methodContext ← *mc.*
↑ *msg isDomain*
ifTrue: [*self performDomainMessage: msg*]
ifFalse: [*self perform: (msg selector) withArguments: (msg arguments)*].
◁ *5.4.5*

Metoda *performDomainMessage:* zpracovává doménovou zprávu. Existují dvě možné varianty: doménová metoda příslušející zprávě byla buď nalezena, nebo nebyla nalezena. V prvním případě dojde k zaslání zpráv *wait* a *activate*. Tyto zprávy způsobí, že vyvolaná metoda se nebude považovat za ukončenou (implicitně se chápe jako atomická) a dojde k její aktivaci. Aktivace vyvolané metody (tj. procesu) spočívá ve zjištění aktuálních událostí procesu. V kontextu OOPN to znamená, že se provede testování přechodů a zjistí se množina potenciálních instancí přechodů.

V případě nenalezení metody se aktivuje příslušná metoda metaobjektu *slave*. Tedy některé doménové zprávy mohou (nemusí) být implementovány jako zprávy metaobjektu a jsou pro všechny doménové instance stejné. Jako příklad si můžeme uvést metodu *hold:* (viz kapitola 5.5.5). Doménový objekt (např. objekt OOPN) může tyto operace překrýt.

***(I)* PNObject (Slave) » performDomainMessage: msg**

↑ methodContext *isNil*
 ifFalse: [methodContext *wait; activate.*]
 ifTrue: [*self perform: (msg selector) withArguments: (msg arguments).*].

◁ *5.4.6*

5.4.6 Zasílání zpráv z kontextu metody

Během evoluce objektů dochází k zasílání zpráv mezi objekty. Vlivem toho vznikají a zanikají *procesní vazby* mezi vlákny procesů a nově evokovanými procesy. V rámci procesu se zprávy zasílají z vláken. Pro zajištění potřebných akcí spojených s procesními vazbami jsou implementovány operace 5.4.7 a 5.4.8.

Na tomto místě trošku předběhneme vývoj popisovaných algoritmů. Každá externí reference na libovolný doménový objekt je v systému realizována prostřednictvím speciálních *proxyobjektů* (viz kapitola 5.6). Význam proxyobjektů spočívá především v tom, že obaluje skutečný (zastupovaný) objekt a může přizpůsobovat jeho výpočetní chování podle různých situací (např. při migraci objektů apod.) Mimo jiné zná i typ zastupovaného objektu (viz zaslání zpráv *receiver isPrimitive* a *receiver isSmalltalk* v metodě 5.4.7), což je důležité pro způsob odeslání zprávy (viz metoda 5.4.7).

Metoda 5.4.7 slouží pro zaslání zprávy objektu. Má tři argumenty – selektor zprávy (*selector*), pole argumentů (*args*) a příjemce zprávy (*receiver*). V případě, že příjemce zprávy je primitivní objekt (tj. konstantní objekt nebo objekt vybraných tříd, viz kapitola 5.6.2), zašle se mu zpráva přímo (prostřednictvím metody *perform:withArguments:* z protokolu základní třídy Smalltalku *Object*). Pokud je příjemce neprimitivní objekt, vytvoří se instance třídy *PNtalkMessage*, vlákno se registruje (*self master thread: self willContinueAfter: message*) a nakonec se zašle zpráva příjemci (*receiver performMessage:*). Konstrukce *self master* při registraci procesní vazby vlákna představuje získání reference na doménový metaobjekt.

Speciální případ nastává v okamžiku, kdy je příjemce zprávy neprimitivní objekt Smalltalku. V tomto případě dojde k označení zprávy jako atomické v čase (*msg atomic*). Označení způsobí vyvolání metody *PNtalkWorld » suspendTime*, která inkrementuje počet atomických zpráv ve světě. Při ukončení zpracování zprávy se vyvolá metoda *PNtalkWorld » activateTime*, která tento počet dekrementuje. Význam této informace je popsán u metody 5.5.15.

***(I)* PNtalkThread » sendMessage: selector withArguments: args to: receiver**

receiver isPrimitive ifTrue: [↑ *receiver perform: selector withArguments: args*].
message ← *PNtalkMessage selector: selector arguments: args*
 sender: (self master asProxy) receiver: receiver.
receiver isSmalltalk ifTrue: [*message atomic*].
self master thread: self willContinueAfter: message.
↑ *receiver performMessage: message.*

◁ *5.4.7*

Registrace zprávy je nutná pro aktivaci vlákna poté, co se ukončí volaná metoda. Registrace spočívá v uložení vlákna do slovníku *listDependencies*. Klíčem k záznamu je zpráva, neboť určuje procesní vazbu. Tento slovník pak reprezentuje seznam závislých vláken $depT$ (viz specifikace 5.3.4).

***(I)* PNObject » thread: aThread willContinueAfter: message**
listDependencies *at: message put: aThread.*

◁ *5.4.8*

V okamžiku ukončení volané doménové metody (viz metoda 5.4.13) dojde k vyvolání metody 5.4.9, čímž dojde k aktivaci vlákna (tzn. že vznikne nová událost). Vlákno si nejprve uloží návratovou hodnotu (objekt) ukončené metody a poté se registruje v procesu (*self process newEvent: self*) jako nová událost. Konstrukce *self process* představuje získání reference na proces, který obsahuje dané vlákno. Přímý výsledek ukončené doménové metody je uložen pro další zpracování v instanční proměnné *result* vlákna.

***(I)* PNtalkThread » continueWith: aValue fromMessage: message**
result *resultValue: aValue.*
self process newEvent: self.

◁ *5.4.9*

5.4.7 Ukončení kontextu metody

Po zpracování zprávy speciálním metaobjektem *slave* a vrácení řízení metodě *processMessage:* doménového metaobjektu se testuje stav kontextu metody (viz metoda 5.4.3). Pokud je kontext metody ukončen, volá se jeho metoda *finish* (v případu bez návratové hodnoty) nebo *finish:* (v případu s návratovou hodnotou). Kontext metody může být buď *PNtalkMethodContext* nebo *PNtalkNet* (viz *Fáze zpracování zprávy – bod 2*), takže následující kód je shodný pro objekty obou struktur:

***(I)* finish**
↑ *self finish: nil.*

◁ *5.4.10*

***(I)* finish: retVal**
message *finishFor: retVal.*
self destroy.

◁ *5.4.11*

Prvním krokem při ukončování kontextu metody je přenesení této informace přes nositelskou zprávu k vysílateli zprávy (*message finishFor:*). Součástí přenosu je i přímý výsledek (návratový objekt) doménové metody. Metoda metaobjektu *PNtalkMessage » finishFor:* předá tyto informace vysílateli zprávy (*sender*), pokud tento existuje. Pokud neexistuje, znamená to, že doménová zpráva se považuje za *asynchronní* a není potřeba zasílat informace o jejím ukončení vysílateli zprávy.

***(I)* PNtalkMessage » finishFor: retVal**
sender *isNil ifFalse:* [sender *thisMessageFinished: (self) with: retVal*].

◁ *5.4.12*

Zpráva *thisMessageFinished:with:* je implementována jako asynchronní (delegovaná), což zajišťuje její vzájemné vyloučení s řídicími procesy nad doménovým metaobjektem. Synchronizace této zprávy je nutná, protože v opačném případě by manipulovala s vnitřními strukturami doménového metaobjektu *vně* jeho řídicího procesu. Metoda *thisMessageFinished:with:* nejprve zjistí vlákno, které je v procesní vazbě s ukončeným procesem. Jako informační zdroj je použit objekt zprávy, viz první řádek v metodě. Vlákno je poté aktivováno (*thread continueWith: aValue fromMessage: msg*, viz metoda 5.4.9). Nakonec je zrušen záznam o procesní vazbě.

***(D)* PNObject » thisMessageFinished: msg with: retValue**
| thread |
thread ← self master listDependencies at: msg.
thread continueWith: retValue fromMessage: msg.
self master listDependencies removeKey: msg.
◁ *5.4.13*

Dalším krokem při ukončování kontextu metody je zrušení kontextu (viz volání *self destroy* v metodě 5.4.11). Kontext může být reprezentován buď metaobjektem *PNtalkMethodContext*, nebo metaobjektem *PNtalkNet*. Základní funkce těchto metod je zrušení registrace procesu. Navíc mohou řešit situace specifické pro interpretovaný proces.

***(I)* PNtalkMethodContext » destroy**
self master removeMethod: self.
◁ *5.4.14*

***(I)* PNtalkNet » destroy**
listPlaces *do:* [*:place* | *place destroy*].
listTransitions *do:* [*:trans* | *trans destroy*].
self master removeMethod: self.
◁ *5.4.15*

Metoda *removeMethod:* doménového metaobjektu zruší procesní vazbu vyvolané metody a příslušné struktury svázené s vyvolanou metodou (procesem).

***(I)* PNObject » removeMethod: aMethod**
self removeInvocation at: (aMethod message).
listProcesses *remove: aMethod.*
◁ *5.4.16*

5.5 Evoluce systému

V předchozí kapitole jsme se věnovali procesům a algoritmům, které zajišťují základní funkcionalitu doménového metaobjektu, a to na základě zpracování zpráv. V následující kapitole se budeme věnovat evolučním procesům systému, tedy algoritmům pro řízení jednotlivých objektů a světa.

5.5.1 Události v OOPN

Základní problém při řízení simulace objektů, které jsou tvořeny OOPN, spočívá ve zvláštním charakteru provádění jednotlivých událostí. Při jakékoliv změně stavu je nejprve nutné provést *testování přechodů*. Výsledkem testování je množina proveditelných přechodů, tj. událostí typu A, N a F podle definice OOPN (viz kapitola 4.3). Poslední z prezentovaných událostí (událost typu J) představuje následující situaci. Již spuštěný přechod (tj. instance přechodu) čeká na dokončení zpracování zprávy, kterou zaslal nějakému objektu. Zpráva byla zpracována a byla vygenerována výsledná hodnota zprávy, tedy objekt, který reprezentuje přímý výsledek zpracování zprávy. V tomto okamžiku může být čekající instance přechodu uvolněna a pokračovat ve své evoluci (viz metody 5.4.13 a 5.4.9) – tj. nastává událost typu J. Události, které mohou nastat při evoluci OOPN, si můžeme rozdělit do dvou kategorií:

- *A, N* a *F* události jsou zjišťovány explicitně prostřednictvím testování aktuálního stavu sítě (do této kategorie patří události *„provedení přechodu"* a *„zaslání zprávy"*, viz kapitola 5.3.1),

- *J* událost je generována implicitně při vlastní evoluci sítí (do této kategorie patří události *„ukončení zprávy"* a *„ukončení přechodu"*, viz kapitola 5.3.1).

V každém kroku evoluce objektu se z množiny aktuálních událostí vybere jedna, která se poté provede. Provedení události může způsobit změnu stavu konečné množiny míst, které podmiňují konečnou množinu přechodů. Při změně takového místa dochází k automatickému otestování příslušných přechodů. Velkou komplikací při zajišťování konzistence množiny proveditelných událostí s reálným stavem jsou synchronní porty. Pokud součástí podmínky nějakého přechodu je volání synchronního portu, můžeme vzniklé problémy charakterizovat takto:

- provedení události v objektu může vést ke změně stavu proveditelnosti synchronního portu, aniž má provedení události přímou vazbu na náš uvažovaný přechod,
- nelze snadno predikovat, které synchronní porty jsou součástí testování přechodu.

Bezpečné řešení tohoto problému spočívá v testování těch sítí, které synchronní porty obsahují, po každém evolučním kroku celého světa. Tím dojde k zajištění konzistence mezi stavem sítí a množinou proveditelných událostí. Algoritmy testování přechodů mohou být podrobeny optimalizačním metodám, které jsou diskutovány např. v disertační práci dr. Vojnara [Voj01].

5.5.2 Struktura světa

Každý objekt je součástí světa, který řídí jeho evoluci v koordinaci s ostatními objekty světa. Svět zajišťuje synchronizaci modelových časů jednotlivých objektů a je iniciátorem podnětů pro kroky evoluce.

> ***Svět*** *je struktura*
>
> $$PNtalkWorld = (Objects, Active, Busy, Executable, Calendar)$$
>
> *kde*
>
> - $Objects \subset \Theta$ *– seznam objektů světa,*
> - $Active \subseteq Objects$ *– seznam aktivních objektů světa,*
> - $Executable \subseteq Objects$ *– seznam objektů obsahujících alespoň jednu událost,*
> - $Busy \subseteq Objects$ *– seznam pracujících objektů,*
> - $Calendar$ *– seznam aktivačních záznamů událostí ve světě.*
>
> *Třídu všech světů simulačního systému označíme* Ξ*.*
>
> $\triangleleft$ *5.5.1*

Metaobjekty typu *master* (tj. doménové metaobjekty nebo světy) mohou být, z hlediska jejich evoluce, v některém z následujících dvou stavů. Stav je odvozen od aktivity při zpracovávání asynchronních (tj. delegovaných) zpráv.

- *pracující* (*busy*) – metaobjekt zpracovává řídicí zprávu / zprávy
- *připravený* (*ready*) – metaobjekt je ustálený, tzn. že nemá žádnou zprávu ke zpracování

Dále si definujeme následující dva pojmy:

- *Aktivita*. Každý doménový metaobjekt může být aktivní nebo neaktivní. Aktivita objektu znamená pouze jeho označení, zda je zahrnut do evolučního cyklu světa nebo ne. Podobně je definovaná aktivita i pro celý svět – pokud je svět neaktivní, žádný objekt nemůže provádět kroky ve své evoluci.
- *Živost*. Pokud neexistuje v doménovém metaobjektu proveditelná událost, říkáme, že je daný objekt *neživý*. Pokud neexistuje ve světě objekt, který by byl živý, a kalendář událostí je prázdný, pak říkáme, že je svět *neživý*.

5.5.3 Řízení objektů světa

Algoritmus řízení evoluce objektů světa si můžeme demonstrovat následujícím způsobem (algoritmus předpokládá, že uvažovaný svět je aktivní). Algoritmus je prezentován deklarativní formou, kde pro každou splněnou podmínku $[\ldots]$ (případně konjunkci podmínek $[\ldots] \wedge [\ldots]$) se provede akce $\Rightarrow$.

- $[\exists\, o \in Objects : o \in Busy] \Rightarrow$ svět je ve stavu $busy$
- [svět není ve stavu $busy$] $\wedge$ $[\neg\exists\, o \in Objects : o \in Executable] \wedge [\neg\exists\, t \in Calendar] \Rightarrow$ svět je *neživý*
- [svět není ve stavu $busy$] $\wedge$ $[\neg\exists\, o \in Objects : o \in Executable] \wedge [\exists\, t \in Calendar] \Rightarrow$ svět vybere nejmenší čas z kalendáře, posune svůj čas a zašle jeho hodnotu všem svým objektům
- [svět není ve stavu $busy$] $\wedge$ [svět je živý] $\wedge$ $[\exists\, o \in Objects : o \in Active \wedge o \in Executable] \Rightarrow$ svět zašle zprávu *step* každému objektu $o \in Active \wedge o \in Executable$

Pokud objekt po otestování proveditelnosti přechodů zjistí, že se nemůže vyskytnout v daném stavu žádná událost, zašle metaobjektu *world* řídicí zprávu *self thisObjectIsNotExecutable:*. Asociovaná metoda zruší tento objekt z množiny $Executable$. Z hlediska řízení evoluce světa to znamená, že tomuto objektu není potřeba zasílat řídicí zprávu *step*.

***(S)* PNtalkWorld » thisObjectIsNotExecutable: anObj**
executable *removeKey: (anObj id).*

◃ *5.5.2*

V objektu, který má prázdnou množinu událostí (a tedy není možné provést evoluční krok objektu), může vzniknout nová událost dvěma způsoby. Buď objekt přijme novou zprávu nebo se ukončí proces vyvolaný z vlákna objektu. V prvním případě se vytvoří nový systémový proces objektu a provede se testování jeho přechodů (viz metoda 5.4.6). Výsledkem testování mohou být události *„provedení přechodu"*.[8] V druhém případě se automaticky vytvoří událost *„ukončení zprávy"*, viz metody 5.4.13 a 5.4.9. Pokud v objektu vznikne nová událost, zašle metaobjektu *world* řídicí zprávu *self thisObjectIsExecutable:*. Asociovaná metoda vloží tento objekt do množiny $Executable$.

***(S)* PNtalkWorld » thisObjectIsExecutable: anObj**
executable *at: (anObj id) put: anObj.*

◃ *5.5.3*

Před započetím zpracování zpráv zašle objekt světu informativní (synchronní) řídicí zprávu *thisObjectIsBusy:*. Asociovaná metoda vloží tento objekt do množiny pracujících objektů. Z hlediska řízení evoluce světa to znamená, že nelze zasílat objektům řídicí zprávu *step*:

[8] Viz typy událostí v kapitole 5.3.1.

(S) **PNtalkWorld » thisObjectIsBusy: anObj**
busy *at: (anObj id) put: anObj.*

◁ *5.5.4*

Pokud je objekt ve stavu *připravený* (tedy zpracuje všechny zprávy), zašle světu informativní řídicí zprávu *thisObjectIsReady:*. Asociovaná metoda vyřadí objekt z množiny pracujících objektů. Pokud bude množina po této operaci prázdná, znamená to, že je poteciálně možné provést další krok evoluce světa a metoda zašle zprávu *step* metaobjektu *world* (*self step*):

(S) **PNtalkWorld » thisObjectIsReady: anObj**
busy *removeKey: (anObj id).*
busy *isEmpty ifTrue:* [*self step*].

◁ *5.5.5*

Metoda *step* světa provádí další krok evoluce světa. Prvním krokem je test aktivnosti světa. Pokud je svět neaktivní, metoda skončí a ke kroku evoluce nedojde. Pokud je aktivní, vytvoří se množina objektů připravených provést evoluční krok jako průnik množin aktivních a připravených (*executable*) objektů.[9] Pokud je výsledná množina (*runnableObjects*) prázdná, dojde k posunu času (*self stepTime* – metoda 5.5.15). V opačném případě se z množiny vybere libovolný objekt a provede se jeden krok jeho evoluce (*object step*).

(D) **PNtalkWorld » step**
| *runnableObjects* |
(self isActive) ifFalse: [↑ *nil*].
runnableObjects ← executable *select:* [*:item* | active *includes: item*].
runnableObjects isEmpty
    ifTrue: [*self stepTime*]
    ifFalse: [| *object* |
        object ← *runnableObjects removeFirst.*
        object step.
    ].

◁ *5.5.6*

5.5.4 Řízení událostí objektu

Objekt obsahuje množinu procesů. Při přijetí zprávy *step* od řídicího metaobjektu *world*, provede objekt jeden krok své evoluce. Tento krok spočívá ve vybrání jedné události a následně v jejím provedení. Událost (resp. záznam o události) je vybrána náhodně z procesů objektu. Při vybrání události se zruší její záznam v seznamu událostí. V metodě *step* doménového metaobjektu je tato situace reprezentována zasláním zprávy *chooseEvent*. Záznam o události je reprezentován buď strukturou, která vznikla jako výsledek testování přechodu (instance třídy *PNtalkEvent*), nebo vláknem. Vybranému objektu se pak zašle zpráva *perform*.

(D) **PNObject » step**
| *event* |
event ← *self chooseEvent.*
event perform.

◁ *5.5.7*

[9] Tento algoritmus je ilustrativní, v reálné implementaci je výhodnější použít jeho optimalizaci.

Pokud je událost reprezentovaná instancí třídy *PNtalkEvent*, znamená to, že se jedná o událost provedení přechodu. Součástí instance třídy *PNtalkEvent* je odkaz na přechod (viz proměnná *component* v metodě 5.5.8), který se má provést, a navázání (viz proměnná *bindings* v metodě 5.5.8), pro které se má přechod provést. Zasláním zprávy *component performFor:* dojde ke vzniku nového vlákna (viz metoda 5.5.9). Pokud je událost reprezentovaná přímo vláknem, jedná se o další krok v evoluci vlákna (viz metoda 5.5.10).

***(I)* PNtalkEvent » perform**
component *performFor:* bindings.
◁ *5.5.8*

Součástí zpracování události provedení přechodu (metoda 5.5.9) jsou i akce spojené s odebráním objektů z příslušných míst a s provedením stráží přechodů. Tyto akce nejsou z pohledu popisu řízení událostí podstatné, a proto je v metodě neuvádíme. Výsledkem těchto akcí je slovník reprezentující hodnoty navázaných proměnných (instanční proměnná *variables* ve vlákně).

***(I)* PNtalkTransition » performFor: aBindings**
| *thread* |
"akce spojené s provedením hran a stráže přechodu na základě navázání aBindings"
thread ← *PNtalkThread newIn: self.*
thread variables: variables.
thread perform.
◁ *5.5.9*

Vlákno může být ve dvou stavech. Buď je schopno provést další operaci ze seznamu akcí, nebo čeká na dokončení operace. Objekt reprezentující návratovou hodnotu operace je uložen v instanční proměnné *result*. Pokud je obsah objektu prázdný (*result isEmpty*), znamená to, že vlákno je ve stavu, kdy je schopno provést další operaci. V opačném případě vlákno dokončí operaci, tj. vyzvedne návratovou hodnotu.

***(I)* PNThread » perform**
result *isEmpty*
ifTrue: [*self performAction*]
ifFalse: [*self finishAction*]
◁ *5.5.10*

Operace vlákna jsou uloženy v instanční proměnné *actions*, což je seřazená kolekce. Index prováděné operace je uložen v instanční proměnné *actionIndex*. Index operace určuje, která operace se právě provádí nebo která operace se má v dalším kroku provést. Metoda *performAction* provede operaci, která je uložena na indexu *actionIndex* v kolekci *actions*. Výsledkem je objekt reprezentující návratovou hodnotu operace (viz instanční proměnná *result*). Pokud tento objekt není prázdný, znamená to, že operace byla okamžitě provedena (byla tedy atomická). V tomto případě se provede ukončení operace, viz metoda *finishAction*. Pokud je objekt *result* prázdný, znamená to, že vlákno je ve stavu čekání na výsledek operace. Výsledek operace je doručen řídicí zprávou (viz metoda 5.4.13). V dalším kroku evoluce tohoto vlákna se opět vyvolá metoda *perform*, tentokrát však instanční proměnná *result* nebude prázdná. To způsobí vyvolání metody *finishAction*, čímž se provede ukončení doménové operace.

***(I)* PNThread » performAction**
result ← *(*actions *at:* actionIndex*) valueInThread: self variables:* variables.
result *isEmpty ifFalse:* [*self finishAction*].
◁ *5.5.11*

Při ukončení operace se zpracuje návratová hodnota operace (*finishInThread: self variables: variables result: result*). Obsah objektu reprezentujícího návratovou hodnotu operace (*result*) je při tom nastaven jako prázdný. Následně se inkrementuje index operace. Pokud hodnota indexu přesáhne velikost kolekce operací, znamená to, že byly provedeny všechny operace vlákna a vlákno se ukončí. V opačném případě došlo k výskytu nové události (operace zaslání následující zprávy). Vlákno se registruje jako reprezentace (záznam) této události, což znamená, že vlákno je připraveno provést další krok.

***(I)* PNThread » finishAction**

(actions *at:* actionIndex) *finishInThread: self variables:* variables *result:* result.
actionIndex ← actionIndex + *1*.
actionIndex > (actions *size)*
ifTrue: [*self destroy*]
ifFalse: [*self process newEvent: self*].

◁ *5.5.12*

5.5.5 Čas v systému

Důležitým aspektem simulace je modelový čas. V prezentovaném simulačním systému si každý objekt udržuje vlastní kalendář událostí, který uchovává informace o časově vázaných událostech. Manipulace s časovými událostmi v doménovém modelu jsou prováděny jako zasílání zpráv vlastnímu objektu (*self*). Např. *self hold: 100* představuje suspendování aktivního vlákna na dobu 100 časových jednotek. Tato zpráva je na doménové úrovni chápána jako synchronní, při jejímž zpracování dojde k suspendování vlákna. Na metaúrovni je tato zpráva zpracována jako zpráva z doménového protokolu. Pokud daná zpráva není nalezena, je předána ke zpracování speciálnímu metaobjektu *slave* (viz metoda 5.4.5). Z principu zpracování této zprávy vyplývá, že vytváří kontext metody. Tento kontext je v obsluze zprávy *hold:* asociován s uvedeným časem a poté vložen do kalendáře. V okamžiku dosažení požadovaného času se tento kontext uvolní a ukončí zpracování metody. Tím se se uvolní původní čekající vlákno, tj. čekání vlákna je ukončeno.

Koncepci práce s časem si ukážeme na vlastním kódu systému. Metoda *hold:* speciálního objektu *slave* doménového metaobjektu zjistí čas následné aktivace události, uloží kontext metody do kalendáře a informuje svět o čase vzniku nové události:

***(D)* PNObject (Slave) » hold: aTime**

| *t* |
t ← *(self world time) + aTime.*
calendar *newEvent: methodContext atTime: t.*
self world newEventAtTime: t.
methodContext block.

◁ *5.5.13*

Svět zpracuje informaci o nové události tak, že si uloží *čas* této události do kalendáře:

***(I)* PNtalkWorld » newEventAtTime: aTime**

calendar *newEventAtTime: aTime.*

◁ *5.5.14*

V okamžiku ustálení světa (tj. svět je ve stavu *připravený*) se svět pokusí posunout čas, pokud je to možné. Metoda *stepTime* (viz volání v metodě 5.5.6) vybere z kalendáře nebližší čas a zašle všem objektům zprávu *releaseEventForTime:*, čímž se jednak přenese informace o změně modelového času do všech objektů a jednak umožní uvolnění příslušných událostí v objektech. Nejprve se ovšem otestuje, zda se ve světě nezpracovává atomická zpráva (*self atomicEvents* $<$ *1*). Pokud ano, nemůže dojít k posunu času (viz koncepce atomických zpráv v čase, kapitola 5.4.1).

(I) **PNtalkWorld » stepTime**
| *newTime* |
(self atomicEvents $<$ *1)*
ifTrue: [
newTime := calendar *nextTime.*
newTime isNil
ifFalse: [
listObjects *do:* [*:obj* | *obj releaseEventsForTime: newTime*].
].
].

◁ *5.5.15*

Objekt zpracuje řídicí zprávu *releaseEventsForTime:* tak, že vybere množinu událostí, které přísluší danému času. Každá taková událost je reprezentována kontextem metody (tj. instancí třídy *PNtalkMethodContex* nebo instancí třídy *PNtalkNet*), uvolnění je tedy provedeno jako ukončení těchto metod (zpráva *finish*, viz metoda 5.4.10). Příslušná vlákna se tímto aktivují a mohou pokračovat v další evoluci (viz metoda 5.4.13).

(D) **PNObject » releaseEventsForTime: aTime**
| *contexts* |
contexts ← calendar *nextEventAtTime: aTime.*
contexts isNil ifFalse: [*contexts do:* [*:context* | *context finish*]].

◁ *5.5.16*

5.6 Transparentnost komunikace

V této kapitole se zamyslíme nad transparentní komunikací mezi objekty, tj. nad způsobem, jak zajistit požadované vlastnosti procesu zpracování zpráv bez ohledu na fakt, jaký je typ cílového objektu. Nejprve si definujeme problém interoperability, který s transparentností komunikace souvisí a navíc je nutný z hlediska využití systému pro prototypování. Poté se zaměříme na řešení daného problému s využitím speciálních metaobjektů – proxy objektů. Prezentovaná problematika je také diskutována v [Koč01b].

5.6.1 Interoperabilita

Interoperabilitu můžeme chápat jako plně transparentní přístup mezi objekty různých prostředí, v našem případě mezi objekty OOPN a Smalltalku. Hlavní motivací této vlastnosti je možnost použít Petriho sítě (OOPN) jako část programu napsaného ve Smalltalku (a naopak). Problém interoperability mezi objekty OOPN a Smalltalku spočívá v různém výpočetním chování těchto objektů. Ve Smalltalku probíhá komunikace prostřednictvím přímého zasílání zpráv, tzn. že objekt, který zaslal zprávu jinému objektu, čeká na dokončení zpracování zprávy. Objekty OOPN jsou reprezentovány svými metaobjekty, které spolu komunikují prostřednitvím asynchronních (delegovaných) zpráv. Doménové zprávy objektu ve Smalltalku jsou k dispozici přímo jako metody objektu, doménové zprávy objektu OOPN jsou k dispozici prostřednictvím řídicí zprávy (viz metoda 5.4.1). Při komunikaci objektů Smalltalku a OOPN zde proto musí existovat vrstva upravující výpočetní chování na požadovanou podobu. Tuto vrstvu reprezentuje metaobjekt, který nazveme *proxyobjekt*.

Proxyobjekty s sebou nesou ještě jednu neméně důležitou vlastnost. Budují vrstvu mezi vysílajícím objektem a jeho příjemcem. Tato vlastnost se uplatní především v okamžiku distribuovaného běhu modelu, kde spolu mohou komunikovat vzdálené objekty, případně kde objekty mohou migrovat mezi jednotlivými uzly simulačního systému (viz kapitola 6.4). Z tohoto důvodu zavedeme dva typy proxyobjektů, které umožňují komunikovat s objekty na doménové úrovni nebo na metaúrovni:

- *domainproxy*. Tento objekt obaluje původní doménový metaobjekt a zpřístupňuje doménové operace reprezentovaného doménového objektu. Doménový proxyobjekt musí implementovat část rozhraní původního metaobjektu, ale navíc umožňuje zasílání doménových zpráv *přímo* proxyobjektu, bez nutnosti znát skutečnou realizaci přenosu zpráv, a to *synchronním* způsobem.
- *metaproxy*. Tento objekt má podobné chování jako doménový proxyobjekt s tím rozdílem, že umožňuje přímé synchronní zasílání zpráv z metaobjektového protokolu.

5.6.2 Vytvoření proxyobjektu

Proxyobjekt je vytvořen na základě zprávy *newFor:*, která je zaslána příslušné třídě proxyobjektu. Ta vytvoří svou instanci, zašle jí zprávu *yourObject:* a vrací vytvořený objekt. Metoda *yourObject:* proxyobjektu provede jeho inicializaci. Inicializace proxyobjektu spočívá v uložení reference na zastupovaný objekt (viz instanční proměnná *tally*) a zjištění typu zastupovaného objektu (viz instanční proměnná *tallyType*). Pro zjištění typu je využito reflektivních vlastností Smalltalku – metoda *isKindOf:* zjišťuje, zda je objekt (příjemce zprávy) instancí zadané třídy. V simulačním systému rozlišujeme tři typy objektů – konstantní objekty, neprimitivní objekty Smalltalku a neprimitivní objekty reprezentované doménovými metaobjekty. Za konstantní objekty jsou považovány ty, které jsou instancemi třídy *Magnitude* (čísla, znaky). Kvůli optimalizaci při běhu zavedeme speciální typ objektů, u kterých můžeme předpokládat, že zpracují zprávy atomicky. Tento typ nazveme *primitivní objekty Smalltalku*. Za primitivní objekty jsou považovány ty, které jsou instancemi třídy *Collection* (kolekce a řetězce). Instance třídy *PNObject* jsou neprimitivní doménové objekty systému a ostatní objekty jsou považovány za neprimitivní objekty Smalltalku.

```
(S) yourObject: anObject
    tally ← anObject.
    (tally isKindOf: Magnitude) ifTrue: [ tallyType ← #const. ↑ self ].
    (tally isKindOf: Collection) ifTrue: [ tallyType ← #primSmalltalk. ↑ self ].
    (tally isKindOf: PNObject)
        ifTrue: [ tallyType ← #oopn. ↑ self ]
        ifFalse: [ tallyType ← #smalltalk. ↑ self ].
```

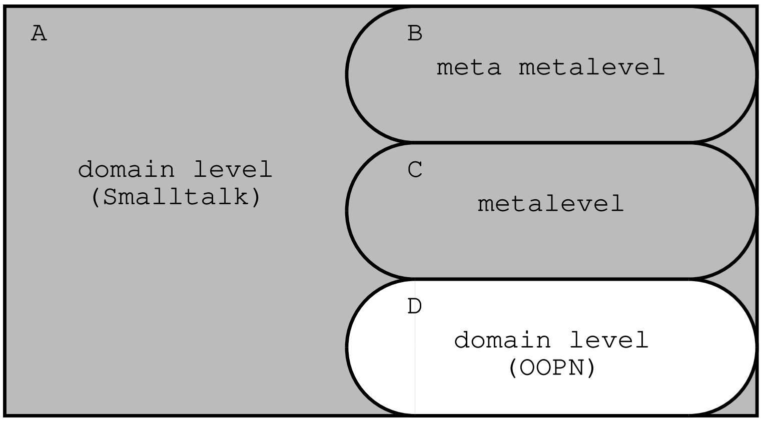
◁ *5.6.1*

Proxyobjekt rozumí zprávám testujícím typ zastupovaného objektu (typ je určen podle obsahu instanční proměnné *tally*): *isConst*, *isSmalltalk*, *isPrimSmalltalk* a *isOOPN*. Zpráva *isPrimitive* testuje, zda je zastupovaný objekt primitivní, tj. zda se jedná o konstantní objekt nebo primitivní objekt Smalltalku.

5.6.3 Komunikace objektů a proxyobjektů

Uvažované objekty spolu mohou komunikovat několika způsoby – každý způsob představuje propojení výpočetního chování různého typu.

Na doménové úrovni můžeme rozlišit objekty dvou typů: objekty přímo podřízené simulačnímu systému (tj. objekty reprezentované doménovými metaobjekty) a objekty Smalltalku. Metaúroveň je tvořena objekty Smalltalku, které mají speciální úlohu – jsou to *metaobjekty* reprezentující doménové



Obrázek 5.7: Vztah Smalltalku a úrovní simulačního systému

objekty podřízené simulačnímu systému. Na obrázku 5.7 jsou naznačeny jednotlivé úrovně v systému. Šedě vyplněné části představují prostor pro objekty Smalltalku. Část těchto objektů tvoří meta metaúroveň a metaúroveň simulačního systému. Ostatní objekty jsou chápány jako doménové objekty Smalltalku.

V následujícím výčtu jsou popsány jednotlivé varianty komunikace objektů. V závorce je uveden prostor, ve kterém se příslušný objekt nachází, viz znaky *A*, *B*, *C* a *D* na obrázku 5.7). Protože objekty na meta metaúrovni se v této komunikaci nevyskytují a doménové objekty jsou zastoupeny svými metaobjekty, jsou přímými účastníky komunikace pouze objekty z prostorů *A* a *C*.

1. doménový metaobjekt (*C*) komunikuje s jiným doménovým metaobjektem (*C*) – zpráva je *delegována* prostřednictvím *řídicí* zprávy.

2. doménový metaobjekt (*C*) komunikuje s konstantním objektem Smalltalku (*A*) nebo s objektem (*A*), který je instancí vybraných tříd ze standardní knihovny Smalltaku, např. kolekce (*Dictionary*, *OrderedCollection*, atd.) Zpracování zprávy je atomické a zpráva je zaslána přímo z řídicího procesu metaobjektu.

3. doménový metaobjekt (*C*) komunikuje s objektem Smalltalku (*A*), u kterého není atomičnost zpracování zprávy zaručena – zpráva nemůže být zaslána přímo z řídicího procesu metaobjektu, ale musí být „obalena” zvláštním procesem. Z pohledu metaobjektu jde o *delegování zprávy*.

4. objekt Smalltalku (*A*) komunikuje s doménovým metaobjektem (*C*) – zpráva musí respektovat způsob zpracování zpráv v systému (tzn. že musí být delegována), z pohledu objektu Smalltalku jde však o *synchronní zaslání* zprávy.

Nyní si jednotlivé případy popíšeme podrobněji, včetně použití proxyobjektů. Jelikož mezi *domainproxy* a *metaproxy* je rozdíl pouze v tom, nad kterým protokolem pracují (vše ostatní je shodné), budeme v dalším textu pracovat pouze s pojmem *proxyobjekt*. V místech, kde dochází k rozdílnosti, tento fakt osvětlíme blíže.

V případu 1 je zasílaní a zpracování zpráv plně v režii simulačního systému. V případech 2–4 dochází ke změnám ve výpočetním chování (odesílatel a příjemce zprávy očekávají jiný model výpočetního chování). V případě 2 jde o změnu přímé zaslání / delegace zprávy, v případech 3 a 4 jde navíc o změnu mechanismu vyvolání metody – objekt Smalltalku předpokládá prosté vyvolání metody třídy, doménový metaobjekt předpokládá vyvolání metody přes řídicí zprávu. Mechanizmus zpracování zprávy doménovým metaobjektem navíc předpokládá, že dostane informaci o ukončení zpracování vyvolané metody a poté si převezme návratovou hodnotu.

případ 1) Tento případ je řešen přímo tak, že proxyobjekt předá argumenty přijaté řídicí zprávy cílovému metaobjektu. V případě umístění cílového metaobjektu na vzdáleném uzlu zajistí přenos zprávy na vzdálený uzel.

případ 2) Tento případ je řešen přímo vysílajícím doménovým metaobjektem. Zde je učiněna výjimka v procesu zpracování zpráv pomocí delegování – zpráva je zpracována přímo i z pohledu metaobjektu (viz metoda 5.4.7). Tento ústupek je proveden především z důvodů rychlosti zpracování, neboť komunikace s konstantními objekty probíhá nejčastěji.

případ 3) Vysílající metaobjekt (sender) vytvoří proxyobjekt pro cílový objekt (receiver). Zasílanou zprávu pak nepošle cílovému objektu, ale proxyobjektu prostřednictvím řídicí zprávy *performMessage:*. Vlastní doménová zpráva (zasílání meta zpráv nemá v tomto kontextu význam) je zaslána cílovému objektu z prostředí speciálního procesu. Současně je simulována *delegace* zprávy z pohledu doménového metaobjektu. Vytvořený proces simuluje chování metaobjektu a po ukončení zpracování zprávy cílovým objektem zašle receiver řídicí zprávu *thisMessageFinished:with:* objektu sender (sender je také zastupovaný svým proxyobjektem, viz metoda 5.6.5). Ukázka příslušné části kódu metody *performMessage:* proxyobjektu, která je invokována při zaslání zprávy podle případu 3 (zastupovaný objekt je uložen v instanční proměnné *tally* proxyobjektu):

***(S)* performMessage: msg**

```
...
[   result ← tally perform: (msg selector) withArguments: (msg arguments).
    (msg sender) thisMessageFinished: msg with: result.
] forkAt: (Processor userSchedulingPriority).
↑ msg.
```

◁ *5.6.2*

případ 4) Vysílající objekt má k dispozici proxyobjekt pro cílový objekt OOPN. Zpráva je zaslána klasickým způsobem. Protože jde o proxyobjekt, který tuto zprávu nezná, je odchycena a předána jako parametr speciální metody *doesNotUnderstand:*. Tato metoda předá odchycenou zprávu ke zpracování metodě *performMessage:*.

***(S)* doesNotUnderstand: msg**

```
↑ self performMessage: msg.
```

◁ *5.6.3*

Metoda *performMessage:* zprávu zpracuje tak, že ji předá cílovému objektu prostřednictvím řídicí zprávy a poté *čeká* na její zpracování. Čekání je simulováno pomocí sdílené fronty *waitQueue*, která suspenduje proces do té doby, než do ní bude vložen objekt. Fronta je sdílená procesem, z kterého je zasílána zpráva, a řídicím procesem cílového objektu. Podle toho, zda se jedná o *domainproxy* nebo *metaproxy*, je volání řídicí zprávy buď *tally performDomainMessage:* (pro *domainproxy*), nebo *tally performMessage:* (pro *metaproxy*). Protože vysílající objekt (sender) nezná metaobjektový protokol, musí se současně simulovat i metaprotokol vysílajícího objektu. Této situace je dosaženo tak, že jako odesílatel se označí proxyobjekt příjemce (viz označení odesílatele *self* při vytvoření systémové zprávy *PNtalkMessage message: msg sender: self*). Ukázka příslušné části kódu metody *performMessage:* proxyobjektu, která je invokována při zaslání zprávy podle případu 4 (ukázka uvažuje variantu *domainproxy*):

(S) **performMessage: msg**

...
tally *performDomainMessage: (PNtalkMessage message: msg sender: self).*
return ← waitQueue *next.*
↑ *return.*

◁ *5.6.4*

dokončení případů 3 a 4) V okamžiku ukončení volané metody zašle cílový objekt (receiver) zprávu *thisMessageFinished:with:*, která je zachycena proxyobjektem původního vysílajícího objektu (sender). Tady mohou nastat dvě varianty podle uvažovaného případu 3 nebo 4. V případu 3 zasílá řídicí zprávu *thisMessageFinished:with:* proxyobjekt zastupující objekt Smalltalku. V kódu metody *thisMessageFinished:with:* je tato varianta reprezentovaná podmíněnou částí *ifFalse:*, která tuto zprávu pouze předá zastupovanému objektu. V případu 4 zasílá řídicí zprávu *thisMessageFinished:with:* původní cílový objekt (*doménový metaobjekt*) svému proxyobjektu (viz *PNtalkMessage message: msg sender: self* v metodě 5.6.4). V kódu metody *thisMessageFinished:with:* je tato varianta reprezentovaná podmíněnou částí *ifTrue:*, která převezme návratovou hodnotu a předá ji do sdílené fronty. Tím se oživí suspendovaný proces (viz volání *waitQueue next* v metodě 5.6.4), který pak vrací získanou návratovou hodnotu.

(S) **thisMessageFinished: msg with: value**

(msg sender = self)
  ifTrue: [waitQueue *nextPut: value.*]
  ifFalse: [tally *thisMessageFinish: msg with: aValue*].

◁ *5.6.5*

Kapitola 6

Význam otevřené architektury

V této kapitole provedeme diskuzi prezentované architektury a jejího významu v různých aplikacích. Budeme se věnovat především různým přístupům k modelování, simulačním technikám a práci se stavy metaobjektů simulačního systému. K tvorbě modelů můžeme přistupovat různým způsobem, např. *kombinované modelování* spojuje diskrétní modely a modely spojité, *multimodelování* využívá pro popis modelované domény spojování menších modelů a *multiparadigma* využívá pro popis modelované domény různá paradigmata, např. Petriho sítě, Smalltalk apod. Multiparadigma je přirozeně uplatnitelné v multimodelování, kde dílčí modely mohou být popsány různými paradigmaty. V následujícím textu si přiblížíme především přístup založený na multiparadigmatu, některé speciální simulační techniky a manipulaci se stavem metaobjektů otevřeného simulačního systému.

6.1 Kombinace paradigmat v modelování

Jednou z klíčových myšlenek prezentované otevřené architektury je kombinace různých vyjadřovacích prostředků, tj. přístup založený na multiparadigmatu. Při modelování v simulačním systému můžeme použít multiparadigma dvěma různými způsoby. Buď zařadíme zvolené paradigma přímo do simulačního systému, nebo spojíme objekty simulačního systému s objekty jiného systému (tj. vytvoříme *multimodel*). Pro potřeby předložené disertační práce si nyní představíme tyto přístupy v úvahách nad kombinací se Smalltalkem a se systémem DEVS a rovněž v možnosti začlenění jiného paradigmatu přímo do prostředků simulačního systému.

6.1.1 Smalltalk a OOPN

V této kapitole se blíže podíváme na možnost spojení objektů simulačního systému s objekty Smalltalku. Hlavní problémy se spojením objektů modelu v systému s jeho okolím jsou v zásadě tyto dva: Protože okolí systému nepodléhá jeho evolučnímu řízení, je nutné zajistit *kompatibilitu výpočetního chování* (tj. zasílání a zpracování zpráv) a je nutné zajistit *synchronizaci modelového času*.

Kompatibilita výpočetního chování Pro zajištění transparentní komunikace mezi objekty simulačního systému a objekty Smalltalku byly zavedeny speciální proxyobjekty (viz kapitola 5.6). Každá reference na objekt, který je součástí simulačního systému, nebo se kterým komunikují objekty simulačního systému, je realizována prostřednictvím proxyobjektů.

Synchronizace časů Možnost ovlivňovat modelový čas v objektech, případně z objektů, které jsou vně simulačního systému, je poměrně malá. Daný objekt by musel implementovat základní rozhraní *doménového metaobjektu* a být zařazen do vybraného světa. Pak může pro manipulaci s časem využít metaobjektový protokol světa. Pokud však budeme uvažovat situaci, kdy je množina externích objektů řízena speciálním objektem (např. DEVS, viz kap. 6.2), můžeme tento objekt abstrahovat nejen jako speciální doménový metaobjekt, ale i jako svět v simulačním systému a zavést synchronizaci času světů. Pro zajištění této vlastnosti zavedeme koncepci *spojených světů* (tj. světů sdílejících modelový čas). Koncepce spojených světů se uplatní také při distribuovaném běhu modelu (viz kapitola 6.4.2).

6.1.2 Řízení spojených světů

Algoritmus evoluce světa, který je součástí spojených světů, si ukážeme na původním algoritmu evoluce světa (viz kapitola 5.5.2) jeho mírnou modifikací. Světy jsou sdružené pod řídící objekt, který nazveme $WorldController$. Každý svět obsahuje navíc referenci na řídicí svět ($controller$). Ostatní definice zůstávají stejné, pouze pro připomenutí: svět obsahuje množiny všech objektů ($Objects$), aktivních objektů ($Active$), pracujících objektů ($Busy$), připravených objektů ($Executable$) a kalendář událostí naplánovaných v čase ($Calendar$). V okamžiku, kdy dojde k ustálení všech objektů (tj. množina $Busy$ je prázdná), pokusí se svět provést další krok evoluce. *Živost světa* říká, zda je možné provést evoluční krok ve světě. Když svět není živý, znamená to, že neexistuje objekt, který by mohl provést krok, a kalendář událostí je prázný. Algoritmus řízení světa (viz kapitola 5.5.2) si nyní upravíme (algoritmus předpokládá, že uvažovaný svět je aktivní). Algoritmus je prezentován deklarativní formou, kde pro každou splněnou podmínku [...] (případně konjunkci podmínek [...] $\wedge$ [...]) se provede akce $\Rightarrow$.

- $[\exists\, o \in Objects : o \in Busy]$ $\Rightarrow$ svět je ve stavu $busy$
- [svět není ve stavu $busy$] $\wedge$ $[\neg\exists\, o \in Objects : o \in Executable] \wedge [\neg\exists\, t \in Calendar]$ $\Rightarrow$ svět je *neživý*
- [svět není ve stavu $busy$] $\wedge$ $[\neg\exists\, o \in Objects : o \in Executable] \wedge [\exists\, t \in Calendar]$ $\Rightarrow$ svět vybere nejmenší čas z kalendáře a zašle jeho hodnotu *řídicímu světu* (*controller*)
- [svět není ve stavu $busy$] $\wedge$ [svět není živý] $\wedge$ $[\exists\, o \in Objects : o \in Active \wedge o \in Executable]$ $\Rightarrow$ svět se stává *živý* a zašle tuto informaci *řídicímu světu* (*controller*)
- [svět není ve stavu $busy$] $\wedge$ [svět je živý] $\wedge$ $[\exists\, o \in Objects : o \in Active \wedge o \in Executable]$ $\Rightarrow$ svět zašle zprávu *step* každému objektu $o \in Active \wedge o \in Executable$
- [svět přijal informaci o novém času od řídicího světa] $\Rightarrow$ svět posune svůj čas a zašle hodnotu nového času všem objektům $o \in Objects$

Prezentovaný algoritmus se od své původní verze odlišuje ve dvou bodech: při možnosti posunu času pouze informuje řídicí svět a skutečný posun nastává až na základě podnětu od řídicího světa. Řídicí svět má tedy jediný úkol: synchronizace modelového času spojených světů.

Řídicí svět *je struktura*

$$WorldController = (Worlds, Busy, Calendar)$$

kde

- $Worlds \subset \Xi$ *– seznam spojených světů,*
- $Busy \subseteq Worlds$ *– seznam pracujících světů,*
- $Calendar$ *– seznam aktivačních záznamů událostí spojeného světa.*

$\triangleleft$ *6.1.1*

Algoritmus řízení spojených světů je podobný algoritmu řízení objektů, tj. pokud je spojený svět ve stavu $busy$, nemůže dělat nic, pokud není ve stavu $busy$ a existuje alespoň jeden záznam v kalendáři událostí, vybere záznam s nejmenší hodnotou a vybranou hodnotu zašle všem svým světům.

- $[\exists w \in World : w \in Busy] \Rightarrow$ spojený svět je ve stavu $busy$
- [spojený svět není ve stavu $busy$] $\wedge$ $[\exists t \in Calendar] \Rightarrow$ řídicí svět zašle hodnotu času $t_{min} = min\,\{t|t \in Calendar\}$ všem světům $w \in World$

Upravený algoritmus si nyní představíme na modifikaci metody *stepTime* (viz metoda 5.5.15). V okamžiku, kdy v původní metodě dochází k zasílání zprávy *releaseEventsForTime:*, dochází nyní k informování řídicího světa (*controller*) zprávou *newTime:in:*.

(I) **PNtalkWorld » stepTime**

```
| newTime |
(self atomicEvents < 1)
  ifTrue: [
    newTime ← calendar nextTime.
    newTime isNil ifFalse: [ controller newTime: newTime in: self master ].
  ].
```

◁ *6.1.2*

Metoda *newTime:in:* registruje nové časy těch světů, které jsou připraveny k posunu času. Informace o čase je uložena jako dvojice $(world, time)$ v kolekci *timeInCoupledWorlds* (tato kolekce je seřazená podle hodnoty času událostí).

(D) **WorldController » newTime: aTime in: aWorld**

```
timeInCoupledWorlds at: aWorld put: aTime.
```

◁ *6.1.3*

Pokud se svět překlopí do stavu $busy$, pak zašle tuto informaci řídicímu světu prostřednictvím zprávy *thisWorldIsBusy:*. Tato metoda uloží příslušný svět do kolekce *busyWorlds*.

(D) **WorldController » thisWorldIsBusy: aWorld**

```
busyWorlds at: (aWorld id) put: aWorld.
```

◁ *6.1.4*

Pokud se svět překlopí ze stavu $busy$ do stavu $ready$, pak zašle tuto informaci řídicímu světu prostřednictvím zprávy *thisWorldIsReady:*. Tato metoda zruší příslušný svět z kolekce *busyWorlds*. Pokud je nyní kolekce prázdná, znamená to, že všechny spojené světy čekají na posun času. Metoda vybere z kalendáře nejmenší čas a zašle jeho hodnotu všem spojeným světům prostřednictvím zprávy *releaseEventsForTime:*. Můžeme vidět, že tento algoritmus je podobný algoritmu řízení evoluce objektů ve světě.

(D) **WorldController » thisWorldIsReady: aWorld**

```
| newTime |
busyWorlds removeKey: (aWorld id).
(busyWorlds isEmpty)
  ifTrue: [
    newTime ← timeInCoupledWorlds first.
    worlds do: [ :world | world releaseEventsForTime: newTime ].
  ].
```

◁ *6.1.5*

Metoda *releaseEventsForTime:* posune čas světa a zašle novou hodnotu času všem svým objektům. Další kroky simulace jsou pak stejné jako u nemodifikovaného algoritmu.

***(D)* PNtalkWorld » releaseEventsForTime: aTime**
listObjects *do:* [*:obj* | *obj releaseEventsForTime: newTime*].

◁ *6.1.6*

Informace o živosti či neživosti světů se zasílají prostřednictvím řídicích zpráv. Jejich implementace není z pohledu popisu algoritmu důležitá, a proto ji zde neuvádíme.

6.1.3 Změna doménového paradigmatu

Architektura systému, tak jak byla prezentována, umožňuje poměrně jednoduchým způsobem začlenit do systému nové doménové paradigma, případně provést modifikaci stávajících paradigmat. Začlenění nového paradigmatu má význam v případě, že chceme běh části modelu, která je popsána tímto paradigmatem, plně podřídit simulačnímu systému.

Každá interpretace doménového paradigmatu simulačního systému je svázána se systémovým procesem (viz kapitola 5.3.2). Na tomto místě se zaměříme na změnu doménového paradigmatu na základě změny systémového procesu. Každý proces objektu je vytvořen v reakci na zaslání zprávy, a tedy invokaci příslušné metody. Každá metoda je svázána se svým doménovým paradigmatem, a tudíž dokáže determinovat systémový proces, který se má vytvořit. Takže pouhým začleněním příslušných informací do reprezentace metody[1] můžeme dosáhnout multiparadigmatu na úrovni jednotlivých metod objektu. S každým typem doménového paradigmatu je svázán jeho překladač a interpret příslušného systémového procesu, který musí dodržovat protokol definovaný pro systémový proces (viz *PNtalkProcess metaprotokol* v příloze D.3). Interpret a překladač musí být implementovány a zařazeny do meta metaúrovně simulačního systému. Při vytvoření instance nového procesu je pak nový systémový proces automaticky zařazen do struktur doménového metaobjektu (viz definice 5.3.4, str. 44).

Jako ukázku si můžeme uvést začlenění podmnožiny jazyka Smalltalk jako systémového procesu simulačního systému. V porovnání s OOPN je tento systémový proces Smalltalku tvořen právě jedním vláknem. Při překladu zdrojového textu metody se vytvoří podobná struktura jako pro akce přechodu v OOPN. Při vyvolání metody se ihned po vytvoření procesu vytvoří a aktivuje vlákno. Sekvenční proces je pak tvořen sekvencí událostí, kde událostí se rozumí (stejně jako v OOPN) zaslání zprávy nebo příjem výsledné hodnoty operace a možnost pokračování v evoluci.

6.2 Systém DEVS

Aplikovatelnost prezentované otevřené architektury simulačního systému si nejlépe přiblížíme srovnáním s léty prověřeným formalismem pro diskrétně-událostní specifikaci systémů DEVS [BZP00]. V této kapitole uvedeme definici vybraných systémů DEVS, princip jejich interpretace a možnost kombinace DEVS a simulačního systému.

6.2.1 Atomický DEVS

Základem DEVS formalismů je komponenta, která se nazývá *atomický DEVS* (*atomic DEVS*). Existují další modifikace a varianty systémů DEVS jako např. spojovaný DEVS, paralelní DEVS, dynamický DEVS, spojitý DEVS apod. Pro potřeby prezentace vlastností simulačního systému si nyní představíme atomický DEVS (v této kapitole) a spojovaný DEVS (v následující kapitole). Popis ostatních systémů DEVS přesahuje požadavky této práce.

[1] Reprezentace metody, viz kapitola 4.4.2.

DEVS *je struktura*

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

kde

- X *je množina vstupních hodnot,*
- S *je množina stavů,*
- Y *je množina výstupních hodnot,*
- $\delta_{int} : S \to S$ *je interní přechodová funkce,*
- $\delta_{ext} : Q \times X \to S$ *je externí přechodová funkce, kde*
  - $Q = \{(s, e)\,|s \in S,\, 0 \le e \le ta(s)\}$ *je množina totálních stavů,*
  - e *je uplynulý čas od posledního přechodu,*
- $\lambda : S \to Y$ *je výstupní funkce,*
- $ta : S \to R^{+}_{0,\infty}$ *je funkce určující dobu čekání ve stavu* $s \in S$ *(*$R^{+}_{0,\infty}$ *je množina kladných reálných čísel s* 0 *a* ∞*).*

$\triangleleft$ *6.2.1*

Princip interpretace DEVS modelu je následující. V libovolném čase t je systém ve stavu s. Pokud nepřijde vnější (externí) událost, systém čeká po dobu $ta(s)$. Hodnotou $ta(s)$ může být kladné reálné číslo, číslo 0 nebo ∞. Pokud nabyde hodnoty 0, znamená to, že systém nemůže přijmout žádnou externí událost. V případě hodnoty ∞ je systém pasivní, dokud externí událost nepřeruší jeho čekání. V okamžiku, kdy čas čekání uplyne (tj. $e = ta(s)$), systém vygeneruje výstup (funkce $\lambda(s)$) a změní vnitřní stav na $\delta_{int}(s)$.

V případě příchodu externí události před vypršením doby čekání $ta(s)$ je systém v totálním stavu (s, e), kde $e \le ta(s)$. V reakci na příchod události systém provede funkci $\delta_{ext} : (s,\, e,\, x)$, čímž se překlopí do nového stavu s' s novou hodnotou $ta(s$'$)$.

Interní přechodová funkce tedy definuje nový stav, pokud nepřijde externí událost od posledního přechodu a externí přechodová funkce definuje nový stav, pokud tato událost přijde.

6.2.2 Spojovaný DEVS

DEVS formalismus není omezen pouze na jednu komponentu, ale obsahuje prostředky pro vytváření modelů z jednotlivých komponent. Tomuto přístupu se říká *spojovaný DEVS* (*coupled DEVS*).

Spojovaný DEVS *je struktura*

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

kde

- $X = \{(p, v)\,|p \in IPorts,\, v \in X_p\}$ *je množina vstupních portů a hodnot, kde*
  - $IPorts$ *je množina vstupních portů,*
  - X_p *je množina vstupních hodnot portu* $p \in IPorts$*,*
- $Y = \{(p, v)\,|p \in OPorts,\, v \in Y_p\}$ *je množina výstupních portů a hodnot, kde*
  - $OPorts$ *je množina výstupních portů,*
  - Y_p *je množina výstupních hodnot portu* $p \in OPorts$*,*

- *D je množina jmen komponent, kde komponenty jsou DEVS modely a pro každé $d \in D$ platí*
  - $M_d = (X_d,\, Y_d,\, S,\, \delta_{int},\, \delta_{ext},\, \lambda,\, ta)$ *je DEVS, kde*
    * $X_d = \{(p, v)\, |p \in IPorts_d,\, v \in X_p\}$ *a*
    * $Y_d = \{(p, v)\, |p \in OPorts_d,\, v \in Y_p\}$,
- $EIC \subseteq \{((N,\, ip_N)\,,\, (d,\, ip_d))\, |\, ip_N \in IPorts,\, d \in D,\, ip_d \in IPorts_d\}$ *je externí vstupní vazba spojující externí vstupy se vstupy komponent,*
- $EOC \subseteq \{((d,\, op_d)\,,\, (N,\, op_N))\, |\, op_N \in OPorts,\, d \in D,\, op_d \in OPorts_d\}$ *je externí výstupní vazba spojující výstupy komponent s externími výstupy,*
- $IC \subseteq \{((a,\, op_a)\,,\, (b,\, ip_b))\, |\, a, b \in D,\, op_a \in OPorts_a,\, ip_b \in IPorts_b\}$ *je interní vazba spojující výstupy komponent se vstupy komponent, kde*
  - *nejsou povoleny přímé zpětnovazební cykly, tj.* $((a,\, op_a)\,,\, (b,\, ip_a)) \in IC$ *implikuje* $a \neq b$,
- $Select\,:\, 2^D \setminus \{\} \rightarrow D$ *je funkce (*tie-breaking function*) určující komponentu z množiny komponent, u které se při nejednoznačnosti dalšího kroku preferuje provedení interní přechodové funkce a výstupní funkce (např. pokud není jasné, zda v daném okamžiku provést interní nebo externí přechodovou funkci komponenty).*

$\triangleleft$ *6.2.2*

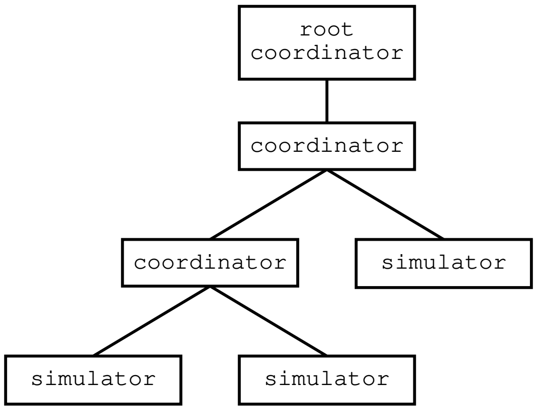
Spojovaný DEVS je tedy tvořen množinou atomických DEVS, kde každý takový DEVS má definovanou množinu vstupních portů a množinu výstupních portů (viz $IPorts_d$ a $OPorts_d$). Dále je definována množina vstupních a výstupních portů pro celý spojovaný DEVS (viz $IPorts$ a $OPorts$), které tvoří vstupy a výstupy hodnot z okolí do spojovaného DEVS a ze spojovaného DEVS do okolí. Vazby mezi komponentami DEVS jsou pak reprezentovány vazbami mezi příslušnými porty (viz EIC, EOC a IC). Např. $((N,\, ip_N)\,,\, (a,\, ip_a)) \in EIC$ znamená, že vstupní port ip_N spojovaného DEVS je spojen s portem ip_a komponenty a. V okamžiku, kdy se na portu ip_N objeví vstupní hodnota, se tato hodnota přenese na vstupní port ip_a, což má za následek vyvolání externí události v DEVS komponentě.

6.2.3 DEVS simulátory

Pro každý typ DEVS komponenty existuje její simulátor zajišťující posun času, koordinaci času u spojovaných DEVS apod. Simulátory jsou zařazeny do hierarchie, jak ukazuje obrázek 6.1. Modelu, který je tvořen spojeným DEVS, odpovídá *koordinátor* (*coordinator*). Modelu, který je tvořen atomickým DEVS, odpovídá *simulátor* (*simulator*). Koordinátor může sdružovat další koordinátory nebo simulátory. Na vrcholu celé hierarchie je *kořenový koordinátor* (*root coordinator*).

Simulace probíhá na základě čtyř zpráv, které si vyměňují podřízené simulátory nebo koordinátory s nadřazeným koordinátorem. Tyto zprávy jsou označovány *i-zpráva*, *x-zpráva*, **-zpráva* a *y-zpráva*. První tři jsou posílány podřízeným koordinátorům nebo simulátorům, poslední z nich je posílána nadřazenému koordinátoru. Sémantika těchto zpráv je následující:

- (i, t) – inicializace podřízeného koordinátoru nebo simulátoru,
- (x, t) – zaslání vstupních hodnot podřízenému koordinátoru nebo simulátoru,
- $(*, t)$ – změna modelového času podřízeného koordinátoru nebo simulátoru,
- (y, t) – zaslání výstupních hodnot koordinátoru nebo simulátoru nadřazenému koordinátoru.



Obrázek 6.1: Hierarchie simulátorů DEVS

Pro naši potřebu srovnání evoluce modelů popsaných DEVS a modelů v simulačním systému je důležitá *-zpráva, která je iniciátorem posunu času. Nyní si představíme algoritmy pro zpracování **-zprávy* v jednotlivých typech DEVS simulátorů.

DEVS simulátor

1. vygeneruje výstup $y = \lambda(s)$
2. zašle y-zprávu (y, t) nadřazenému koordinátorovi (ten zajistí přenos hodnot výstupu y do příslušných komponent prostřednictvím *x-zpráv*)
3. vygeneruje nový stav $s = \delta_{int}(s)$
4. nastaví čas poslední události $tl = t$ (tj. právě aktuální modelový čas)
5. nastaví čas následující události $tn = tl + ta(s)$

DEVS koordinátor

1. vybere podřízenou komponentu $d* = first(eventlist)$
2. zašle *-zprávu $(*, t)$ komponentě $d*$
3. seřadí seznam $eventlist$ podle časů tn_d podřízených komponent d a množiny $Select$
4. nastaví čas poslední události $tl = t$
5. nastaví čas následující události $tn = min\{tn_d | d \in D\}$

DEVS kořenový koordinátor

1. nastaví čas následující události celého DEVS modelu $t = tn$ podřízené komponenty $child$
2. $loop$:
   (a) zašle *-zprávu $(*, t)$ podřízené komponentě $child$
   (b) nastaví čas následující události $t = tn$ podřízené komponenty $child$

6.2.4 Kombinace simulačního systému a DEVS

Nyní si ukážeme možnost začlenění systému DEVS do otevřené architektury simulačního systému a naopak začlenění části simulačního systému do systému DEVS. V popisu budeme uvažovat svět simulačního systému, který může tvořit komponentu spojeného světa (má tedy upravené algoritmy evoluce, viz kapitola 6.1.2).

Začlenění DEVS do simulačního systému. V kapitole 5.2.2 jsme definovali pojem *typ doménového objektu*. Ke každému typu doménového objektu existuje speciální doménový metaobjekt. Atomický nebo spojovaný DEVS může být chápán jako speciální typ doménového objektu, který je reprezentován speciálním doménovým metaobjektem (nazveme jej *doménový metaobjekt DEVS*). Tento metaobjekt, který je začleněn do světa jako jeho komponenta, nahrazuje kořenový koordinátor v DEVS. Při komunikaci se svou podřízenou komponentou ($child$) uplatňuje původní zprávy DEVS simulátorů. Neprovádí však rozhodování o posunu času, ale tvoří spojovací článek mezi DEVS modelem a světem. Pro komunikaci se světem pak platí následující pravidla:

- Po simulačním kroku zjistí doménový metaobjekt DEVS čas následující události (viz bod 2b v algoritmu kořenového koordinátoru) a předá ji světu prostřednictvím zprávy *newEventAtTime:* (viz metoda 5.5.14).
- Pokud svět rozhodne o posunu času, informuje všechny své komponenty zprávou *releaseEventsForTime:* (viz metoda 5.5.16). Pokud je nový čas světa t_w roven času t doménového metaobjektu DEVS, pak zašle **-zprávu* $(*, t)$ podřízené komponentě $child$ (viz bod 2a v algoritmu kořenového koordinátoru).

Začlenění simulačního systému do DEVS. V tomto případě musíme uvažovat spojovaný DEVS (*coupled DEVS*), do kterého se začlení *svět* simulačního systému jako jeden z atomických DEVS. Protokol světa musí být upraven tak, aby odpovídal dané implementaci spojovaného DEVS. Evoluce času světa by se podřídila spojovanému DEVS, tzn. že impulzy k posunu času musí vycházet od řídicího objektu spojovaného DEVS (tj. od jednoho z DEVS koordinátorů). Pro komunikaci světa s nadřazeným koordinátorem DEVS pak platí následující pravidla:

- Zpráva $(*, t)$ odpovídá zprávě *releaseEventsForTime:* (viz metoda 6.1.6).
- Svět zasílá informaci o čase následující události prostřednictvím zprávy *newTime:in:* (viz metoda 6.1.3).

Mapování portů a zpráv. Předchozí popis uvažoval začlenění příslušných objektů do hierarchie simulačního systému, případně do hierarchie systému DEVS, a synchronizaci časů. Neuvažoval možnost komunikace DEVS komponent a objektů simulačního systému. Problém spočívá v různém přístupu k předávání hodnot mezi komponentami: DEVS komponenty je zasílají prostřednictvím svých nadřazených koordinátorů a *y-zpráv* (koordinátor pak tyto hodnoty rozešle prostřednictvím *x-zprávy* příslušným komponentám), zatímco objekty simulačního systému spolu komunikují přímo prostřednictvím svých doménových zpráv. Tento problém lze řešit mapováním portů DEVS komponent (viz specifikace 6.2.2). Komunikace mezi objekty simulačního systému a DEVS komponentami je pak založena na přenosu informací prostřednictvím asynchronních zpráv.

- *Přenos informace z DEVS komponenty do objektu simulačního systému.* Koordinátory DEVS přijímají *y-zprávy* nesoucí informaci z výstupních portů DEVS komponent. Pokud má být tato informace předána portu $op_N \in OPorts$ (tj. portu z množiny výstupních portů spojovaného

DEVS), projde *y-zpráva* až do doménového metaobjekt (příp. světa), který reprezentuje kořenový koordinátor. Metaobjekt předá uvedenou zprávu řídicímu světu, který uvedený port op_N namapuje na objekt simulačního systému a jeho doménovou zprávu. Tomuto objektu se poté zašle příslušná doménová zpráva *asynchronně*. Předpokladem je, že objekt simulačního systému rozumí zprávě odpovídající příslušnému portu.

- *Přenos informace z objektu simulačního systému do DEVS komponenty*. Kořenový koordinátor DEVS (resp. jeho metaobjekt) je v simulačním systému dostupný prostřednictvím proxyobjektu. Při příjmu zprávy z doménového protokolu předá proxyobjekt příslušnou informaci kořenovému koordinátoru prostřednictvím speciální *asynchronní* řídicí zprávy. Metaobjekt kořenového koordinátoru namapuje příslušnou zprávu na porty z množiny vstupních portů (tj. porty $ip_N \in IPorts$). Zasílaná doménová zpráva se chápe jako asynchronní.

6.3 Reprezentace stavu objektů

Na tomto místě si přiblížíme práci se stavy objektů. Pod pojmem práce se stavy objektů budeme rozumět ukládání stavu (tento proces bývá nazýván *serializace*) a obnovení stavu (tento proces bývá nazýván *deserializace*), přičemž se zaměříme na stavy *světů* a *objektů*. Práce se stavem *procesoru* (tj. metaobjektu, který reprezentuje celý systém, viz kapitola 5.2.3) je podobná práci se stavem světa, a proto ji nebudeme uvádět. Součástí stavu objektu jsou stavy *procesů*. Reprezentace stavu procesů závisí na paradigmatu, který reprezentují, a na simulátoru, jenž jej implementuje. Tato práce se věnuje Objektově orientovaným Petriho sítím (OOPN), a proto se následující popis stavu procesů bude týkat právě OOPN. Součástí stavu procesu OOPN jsou stavy *vláken* a *míst*.

Důležitá vlastnost, která vyplývá z otevřené implementace simulačního systému a práce se stavy jeho komponent, je možnost *reifikace stavu*. Tedy možnost změnit stav komponent tak, že se nejprve získá stav komponenty (v textové formě či ve formě kolekce), poté se provede změna vygenerovaného stavu (prostou změnou kolekce nebo textu) a nakonec se komponentě vnutí nový stav.

6.3.1 Identifikace metaobjektů

Identifikátor každého metaobjektu je tvořen celým kladným číslem. Každý metaobjekt obsahuje množinu metaobjektů (např. *svět* obsahuje množinu *objektů*). Identifikátory metaobjektů, které jsou součástí jiného (nadřazeného) metaobjektu, jsou jedinečné v rámci tohoto nadřazeného metaobjektu. Těmto identifikátorům říkáme *lokální*. Pro jednoznačnou globální identifikaci metaobjektu jsou definovány *globální* identifikátory, které vzniknou konkatenací řetězců identifikátorů nadřazených entit včetně identifikátoru entity. Nyní si uvedeme množiny identifikátorů procesoru, světa, objektu, procesu, místa a vlákna. Operátor $\langle\rangle$ reprezentuje vytvoření řetězce a . operátor konkatenace řetězců (s přidáním oddělovače řetězců „:“).

- $ID_{PW} = \{\, \langle id \rangle \mid id \in \mathbb{N}\}$ reprezentuje identifikátory procesorů,
- $ID_W = \{\, a.\langle id \rangle \mid a \in ID_{PW} \;\wedge\; id \in \mathbb{N}\}$ reprezentuje identifikátory světů,
- $ID_O = \{\, a.\langle id \rangle \mid a \in ID_W \;\wedge\; id \in \mathbb{N}\}$ reprezentuje identifikátory objektů,
- $ID_P = \{\, a.\langle id \rangle \mid a \in ID_O \;\wedge\; id \in \mathbb{N}\}$ reprezentuje identifikátory procesů,
- $ID_{PCm} = \{\, a.\langle id \rangle \mid a \in ID_P \wedge id \in \mathbb{N}\}$ reprezentuje identifikátory komponent procesu (tj. místa, přechody a vlákna).

Jako názornou ukázku si uvedeme lokální a globální identifikátor objektu ($ID = 23$), který je umístěn ve světě ($ID = 2$) a procesoru ($ID = 1$):

- lokální identifikátor $LID = 23$,
- globální identifikátor $GID = "1:2:23"$.

6.3.2 Stav místa

Místo je reprezentováno multimnožinou, která může obsahovat dva typy prvků: prvek *objektového typu* nebo prvek *konstantního typu*. Konstantní typ představuje objekty, které jsou statické (nemění se jejich stav), jako jsou čísla, statické řetězce, znaky a symboly. Množinu všech takových objektů si označme Ω. Objektový typ pak reprezentuje všechny ostatní objekty. Proto si v souvislosti se stavem místa definujeme dva pojmy: *stav položky místa* a *stav místa*.

Stav položky místa *je*

$$PI = (type, value, mult)$$

kde

- $type \in \{object, const\}$ *– typ prvku položky místa (objektový nebo konstantní typ),*
- *pokud* $type = object$*, pak* $value \in ID_O$ *je OID neprimitivního objektu (prvku místa),*[2]
- *pokud* $type = prim$*, pak* $value \in \Omega$ *je hodnota konstatního objektu (prvku místa),*
- $mult \in \mathbb{N}$ *– násobnost prvku v místě,*

Operace nad stavem položky místa: Získání a obnovení stavů jednotlivých metaobjektů si ukážeme na kódu příslušných metod. Obecně pro získání stavu libovolného metaobjektu (pokud má smysl ukládat tento stav) slouží operace *state*. Výsledkem operace je kolekce (s možností indexování) obsahující stav místa. Pro obnovení stavu metaobjektu slouží operace *state:*, která má jako parametr kolekci se stavem metaobjektu.[3] Pro stav položky místa PI jsou definovány uvedené operace takto:

PlaceItem » state
↑ *Collection*
with: type *with: (*value *isConstant ifTrue:* [value] *ifFalse:* [value *id*]*) with:* mult.
◁ *6.3.1*

PlaceItem » state: aCollection
type ← *aCollection at: 1.*
value ← *self isConstant ifTrue:* [*aCollection at: 2*]
ifFalse: [*PNtalkObjectProxy newForOID: (aCollection at: 2)*].
mult ← *aCollection at: 3.*
◁ *6.3.2*

[2] Pokud je prvek místa neprimitvní objekt Smalltalku, je situace obdobná jako u neprimitivního objektu OOPN. Ukládáním stavu objektů Smalltalku se zabývá kapitola 6.3.7.

[3] Každý takový metaobjekt obsahuje i metody pracující s textovou reprezentací stavu metaobjektu (viz *xmlState* a *xmlState:* v příloze D.1). Převod mezi reálnou reprezentací a textovou reprezentací je předmětem algoritmů pro práci s XML dokumenty a není zajímavý z pohledu prezentované práce se stavy metaobjektů.

Stav místa *je*

$$S_{Pl} = (id, listPI)$$

kde

- $id \in ID_{PCm}$ *– jedinečná identifikace místa,*
- $listPI$ *– seznam položek místa, prvkem kolekce je stav položky místa* (PI)*.*

Operace nad stavem místa: Operace *state* volá postupně operaci *state* na všechny položky místa. Výsledkem je kolekce kolekcí, kde vnořené kolekce reprezentují stav příslušných položek místa.

PNtalkPlace » state
$\uparrow$ *(content collect: [:item | item state]) addFirst: (self id); yourself.*
$\triangleleft$ *6.3.3*

PNtalkPlace » state: aCollection
(aCollection at: 2) collect: [:item | content add: (PI new state: item)].
$\triangleleft$ *6.3.4*

6.3.3 Stav vlákna

Vlákno v OOPN reprezentuje prováděný přechod, který obsahuje sekvenci operací a množinu proměnných. Každá operace má přidělený index, který odpovídá pořadí operace v sekvenci vlákna. Součástí stavu vlákna jsou stavy proměnných a index právě prováděné operace.

Stav proměnné *je*

$$V = (name, type, value)$$

kde

- $name$ *– název proměnné,*
- $type \in \{object, const\}$ *– typ obsahu proměnné (objektový nebo konstantní typ),*
- *pokud* $type = object$*, pak* $value \in ID_O$ *je OID neprimitivního objektu (obsah proměnné),*
- *pokud* $type = prim$*, pak* $value \in \Omega$ *je hodnota konstatního objektu (obsah proměnné).*

Operace nad stavem proměnné: Operace *state* a *state:* nad stavem proměnné jsou podobné operacím nad stavem položky místa (viz metody 6.3.1 a 6.3.2), a proto je zde neuvádíme.

Stav vlákna *je*

$$S_T = (id, index, transition, variables)$$

kde

- $id \in ID_{PCm}$ *– jedinečná identifikace vlákna (instance přechodu),*
- $index \in \mathbb{N}$ *– index právě prováděné operace ve vlákně,*
- $transition$ *– název přechodu, jehož je vlákno instancí,*
- $variables$ *– seznam stavu proměnných vlákna* (V)*.*

Operace nad stavem vlákna: Operace *state* vytvoří kolekci stavu proměnných, pak přidá jméno přechodu (jehož je vlákno instancí), informaci o aktuálním indexu operace (*self actionIndex*) a nakonec identifikaci vlákna. Výsledkem je kolekce: $(id\, index\, transname\ (var1\, var2\, \ldots))$.

PNtalkThread » state
↑ *(listVariables collect: [:item | item state])*
addFirst: (self transition name);
addFirst: (self actionIndex);
addFirst: (self id);
yourself.

◁ *6.3.5*

PNtalkThread » state: aCollection
actionIndex ← aCollection at: 2.
self transition: (aCollection at: 3).
(aCollection at: 4) collect: [:item | listVariables add: (Variable new state: item)].

◁ *6.3.6*

6.3.4 Stav procesu

Stav procesu je tvořen kolekcí obsahující identifikaci procesu, stav všech vláken, stav všech míst a název sítě reprezentující strukturu procesu. Přechody není třeba ukládat, protože jejich definice je uložena v reprezentaci třídy OOPN. Pokud bychom ovšem uvažovali možnost dynamických změn nejen metod, ale i *procesů*, museli bychom jakou součást stavu procesu ukládat i reprezentaci sítě daného procesu. Pro potřeby serializace procesů by postačila textová reprezentace sítě. Při rekonstrukci procesu by se základní struktura netvořila podle *třídy* objektu, ale podle uložené struktury sítě.

***Stav procesu** je*

$$S_P = (id, net, threads, places)$$

kde

- $id \in ID_P$ – *jedinečná identifikace procesu,*
- net – *jméno metody (sítě) tvořící strukturu procesu,*
- $threads$ – *seznam stavů vláken* (S_T)*,*
- $places$ – *seznam stavů míst* (S_{Pl})*.*

Operace nad stavem procesu: Operace *state* vytvoří kolekci obsahující (postupně od indexu 1) identifikaci procesu, název struktury, kolekci vláken a kolekci míst.

PNtalkProcess » state
↑ *OrderedCollection new;*
add: (self methodName);
add: (self id);
add: (listThreads collect: [:item | item state]);
add: (listPlaces collect: [:item | item state]);
yourself.

◁ *6.3.7*

PNtalkProcess » state: aCollection
self methodName: (aCollection at: 2).
(aCollection at: 3) collect: [:item | listThreads add:
((PNtalkThread newWithID: (item at: 1)) state: item)].
(aCollection at: 4) collect: [:item | listPlaces add:
((PNPlace newWithID: (item at: 1)) state: item)].

◁ *6.3.8*

6.3.5 Stav objektu

Stav objektu je tvořen identifikací objektu, množinou vazeb závislostí vláken a procesů, stavem všech procesů a identifikací třídy. Pokud bychom zobecnili předchozí úvahy o dynamických změnách i přímo na objekt, museli bychom ukládat i definici struktur (tj. metod) v rámci objektů. Tento koncept však evokuje otázku beztřídního přístupu k objektově orientovaným systémům (viz např. Self [Sel04] nebo HELEF [Hru04]) a přesahuje cíle této práce. Tato možnost se však bere v úvahu pro další vývoj systému.

Stav objektu *je*

$$S_O = (id,\, class,\, processes,\, dpT,\, evP)$$

kde

- $id \in ID_O$ *– jedinečná identifikace objektu,*
- $class$ *– název třídy objektu,*
- $processes$ *– seznam stavů procesů objektu (*S_P*),*
- $dpT = \{\, (t, p) \mid t \in ID_{PCm},\, p \in ID_P \}$ *– množina procesních vazeb mezi vláknem* t *objektu* id *a vyvolaným procesem* p*,*[4]
- $dpP = \{\, (p, t) \mid p \in ID_P,\, t \in ID_{PCm} \}$ *– množina procesních vazeb mezi procesem* p *objektu* id *a vláknem* t*.*[4]

Operace nad stavem objektu: Operace nad stavem objektu jsou podobné operacím nad stavy míst, vláken apod., a proto je nebudeme podrobně uvádět.

6.3.6 Stav světa

Stav světa je tvořen identifikací světa a stavem všech jeho objektů.

Stav světa *je*

$$S_W = (id,\, objects)$$

kde

- $id \in ID_W$ *je jedinečná identifikace světa,*
- $objects$ *je seznam stavů objektů světa (*S_O*).*

Operace nad stavem světa: Operace nad stavem světa jsou podobné operacím nad stavy míst, vláken apod., a proto je nebudeme podrobně uvádět.

[4] Procesní vazba, viz kapitola 5.3.3 a specifikace doménového metaobjektu 5.3.4 (strana 44).

6.3.7 Stav objektů Smalltalku

Abychom docílili skutečně transparentní manipulace se stavem objektů modelu, musíme uvažovat i stavy těch neprimitivních objektů Smalltalku, které jsou součástí modelu. Pokud je součástí místa např. kolekce, tj. instance třídy *OrderedCollection* (viz obrázek 4.4 na straně 31), pak se musí společně se stavem objektu OOPN ukládat i stav této kolekce.

Pro uložení stavu těchto objektů lze s výhodou využít existující nástroje pro serializaci a deserializaci objektů prostřednictvím XML. Jedním z takových nástrojů je např. SIXX.[5] Každý objekt je v simulačním systému dostupný prostřednictvím svého proxyobjektu (viz kapitola 5.6). Proxyobjekt rozumí operacím, které jsou určené pro práci nad stavem objektu, a současně má přidělenou jednoznačnou identifikaci v rámci simulačního systému (pro identifikátor id proxyobjektu platí $id \in ID_O$). Na již uvedených algoritmech pro práci nad stavy metaobjektů se nezmění nic s jednou výjimkou. Pokud proxyobjekt, který zastupuje objekt Smalltalku, příjme zprávu *xmlState*, pak využije např. již zmiňovaný nástroj SIXX a vygeneruje XML reprezentaci objektu Smalltalku, přičemž identifikace objektu v uložené reprezentaci musí odpovídat identifikaci proxyobjektu.

6.4 Speciální techniky pro simulaci a prototypování

V této kapitole se zamyslíme nad možnostmi využití speciálních simulačních technik v simulaci a prototypování z pohledu otevřeného simulačního systému. Speciální technikou se v kontextu této práce rozumí takové manipulace se simulačním modelem, které se vymykají klasickému pojetí simulačního cyklu *vytvoření / změna modelu – spuštění modelu v jedné časové ose – ověřování výsledků*. Simulační model může během své evoluce spouštět jiné simulační modely, může být ovlivňován jinými evolučními cykly, může modifikovat sám sebe, příp. vytvářet své (modifikované) klony a testovat optimální cestu další evoluce (tedy provádět simulaci v simulaci) apod. V neposlední řadě je možné vytvořit část modelu v jiném paradigmatu či zapojit model do daného prostředí – tj. vytvářet prototypy aplikací. Některé z uvedených praktik si nyní podrobněji popíšeme.

6.4.1 Vnořená simulace

Vnořenou simulaci si můžeme představit jako simulaci v simulaci. Simulační model v každém okamžiku své evoluce rozhoduje o dalším kroku podle předem daných pravidel a kritérií. V některých případech však může být výhodné provádět tato rozhodnutí na základě výsledků experimentů nad sebou samým – tedy provádět jakési „myšlenkové experimety” a podle nich rozhodnout o dalším evolučním kroku. Pokud se na tento problém podíváme z pohledu realizace „myšlenkových experimentů” v simulačním prostředí, zjistíme, že je nutné vytvořit speciální simulační prostor, ve kterém by probíhala vnořená simulace. Vlastnosti takového prostoru si můžeme shrnout do několika bodů:

- simulační model může být vytvořen dynamicky,
- simulace musí probíhat v nezávislé časové ose,
- simulační model může ovlivňovat jiné simulační modely a současně může být ovlivňován jinými simulačními modely.

Uvedeným požadavkům vyhovuje koncepce světů, která byla prezentována v kapitole 5.2.3, umožňující jednoduchým a unifikovaným způsobem pracovat s různými simulacemi. Vazba mezi jednotlivými světy je daná jejich použitím v modelu (tedy pokud simulační model získá referenci na libovolný svět, může

[5] Smalltalk Instance eXchange in Xml, http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/index.html

ovlivňovat jeho evoluci i strukturu jeho objektů). V tomto okamžiku se uplatňuje metaúrovňová architektura, neboť světy jsou metaobjekty, které řídí doménové metaobjekty. Simulační model tedy musí provést reifikační operace (čímž se řízení přepne do metaúrovně, viz kapitola 3.4), jejichž vliv se pak reflektuje do příslušných objektů. V současné době systém postrádá bezpečnostní prvky (ve smyslu *safety*), a je tedy nutné dodržovat jistá pravidla pro zajištění konzistentního stavu procesu reflexe (především při reifikačním přístupu k *jinému* metaobjektu než přísluší volajícímu objektu). Proces reifikace a následně reflexe musí být prováděn za následujících podmínek:

- objekt musí být suspendován (tzn. že při započetí reifikačního procesu musí dojít k suspendování objektu, nebo toto suspendování musíme zajistit předem),
- během procesu reifikace a následně reflexe se nesmí jiným způsobem změnit stav objektu,
- po ukončení všech uvedených procesů se objekt musí vrátit do původního evolučního stádia (tzn. pokud byl např. aktivní, po provedených změnách musí být opět aktivní).

Příklad 6.1: *Ukázka aplikace vnořené simulace*
Podstatným důsledkem otevřeného návrhu simulačního systému je možnost kontroly doménových objektů prostřednictvím jejich metaobjektů a možnost tuto kontrolu provádět z doménové úrovně. Tento koncept umožňuje provádět (mimo jiné) různé simulační techniky. Jako ukázku aplikace vnořené simulace si můžeme uvést následující kód akce přechodu:

world ← *self meta world.*
newWorld ← *world clone.*

Operace *self meta* představuje získání reference na metaobjekt daného doménového objektu, a tedy doménový objekt může provádět reifikační operace nad metaobjekty.[6] Zprávou *world* z metaobjektového protokolu pak získáme referenci na metaobjekt reprezentující svět, do kterého je objekt vložen. Voláním metaobjektové operace *world clone* vytvoříme *klon* našeho prostředí, které po dokončení vytvoření pokračuje v evoluci jako původní svět. Protože v průběhu reifikačního procesu, vyvolaného metodou *clone*, nelze reflektovat stav této operace do nově vytvářeného klonu, je v klonu tato operace ukončena s hodnotou *nil*. Pro úplnost dodáváme, že operace *clone* je asynchronní (tj. delegovaná), čímž je zajištěno výlučné zpracování této operace.

Pokud bychom chtěli dosáhnout jistých změn před vlastní evolucí klonu, můžeme místo operace *clone* použít operaci *cloneAndSuspend*, která po dokončení operace klonování zastaví evoluci vytvořeného světa. Následující ukázka vytváří suspendovaný klon světa, získává referenci na klon objektu, ve kterém je tento kód obsažen (proměnná *clone*, lze toho dosáhnout díky tomu, že *lokální* identifikace objektů v původním a nově vytvořeném světě je stejná), a vkládá novou instanci objektové sítě. Poté nově vytvořený svět aktivuje.

world ← *self meta world.*
newWorld ← *world cloneAndSuspend.*
newObject ← *newWorld componentID: (self meta id).*
newObject compile: 'zdrojový kód objektové sítě'.
newWorld resume.

□

[6] Při procesu reifikace může dojít ke změně chování či struktury doménového objektu (tj. k procesu reflexe).

Příklad 6.2: *Ukázka vnořené simulace s ovlivňováním simulace I*
Pokud bychom chtěli zajistit vazbu nově vzniklého světa na původní svět, můžeme buď využít předem připravených doménových metod modelu, nebo reflektivních vlastností systému. My si teď ukážeme druhou variantu. Vyjdeme z předchozí sekvence volání operací, kterou doplníme:

```
...
newObject compile: 'zdrojový kód objektové sítě'.
newWorld resume.
self hold: 200.
newWorld suspend.
```

V této ukázce jsme aktivovali svět, čekali 200 časových jednotek a poté jsme klonovaný svět opět suspendovali. Jinými slovy, provedli jsme vnořenou simulaci trvající 200 časových jednotech *našeho času* (v klonovaném světě mohl uběhnout jiný čas). Po suspendování klonovaného světa můžeme zjistit výsledky této simulace a provést rozhodnutí o dalším vývoji naší simulace (viz metaobjektový protokol v příloze D).

□

Příklad 6.3: *Ukázka vnořené simulace s ovlivňováním simulace II*
Jiný přístup k zajištění vazby nově vzniklého světa na původní svět je využít doménových metod modelu, ať již předem připravených, nebo dodatečně doplněných (s využitím operací z metaobjektového protokolu). Opět vyjdeme ze sekvence volání operací z příkladu 6.4.1, kterou doplníme:

```
...
newObject compile: 'object ... place parent() ...'.
newObject compile: 'method parent: aParent
      place aParent()
      trans t
         precond aParent(parent)
         postcond parent(parent)'.
newObject parent: self.
newWorld resume.
```

V této ukázce jsme zakompilovali do nového klonovaného objektu novou objektovou síť, které jsme mj. přidali místo se jménem *parent*. Dále jsme zakompilovali novou metodu, která uloží objekt ze vstupního parametru do místa *parent* objektové sítě. Nakonec jsme pomocí této metody předali novému objektu referenci na původní objekt (tj. *self*) a aktivovali svět. Poté, co simulace dosáhne určitého stavu, se může aktivovat přechod objektové sítě, který zašle zprávu původnímu objektu s případnými výsledky.

□

Příklad 6.4: *Ukázka vnořené simulace s ovlivňováním simulace III*
Dalším přístupem k zajištění vazby nově vzniklého světa na původní svět je využít doménových metod modelu v kombinaci s metaobjektovým protokolem a operacemi nad stavem objektů. Opět vyjdeme ze sekvence volání operací z příkladu 6.4.1, kterou doplníme:

```
...
newObject compile: 'object ... place parent() ...'.
place ← (newObject componentID: 1) placeNamed: 'parent'.
place addObject: self mult: 1.
newObject (newObject componentID: 1) addComponent: place.
newWorld resume.
```

V této ukázce jsme zakompilovali do nového klonovaného objektu novou objektovou síť, které jsme mj. přidali místo se jménem *parent*. Prostřednictvím operací z metaobjektového protokolu jsme pak do tohoto místa vložili referenci na původní objekt (tj. *self*) a aktivovali svět. Poté, co simulace dosáhne určitého stavu, se může aktivovat přechod objektové sítě, který zašle zprávu původnímu objektu s případnými výsledky.

□

Podobným způsobem jako u vnořené simulace můžeme dosáhnout větvení simulace, tj. vytvořit několik klonů existujícího stavu simulace světa a nechat probíhat nezávislé evoluce. Na základě dosažených výsledků se pak může vybrat jeden z klonovaných světů, který pokračuje dále ve své evoluci (ostatní klony se zruší). Tento koncept umožňuje vybírat optimální varianty hodnot proměnných v simulačním modelu nebo optimální strukturu vybrané části simulačního modelu.

6.4.2 Migrace objektů

Jedním z dalších možností popisovaného systému je použití migrace objektů při tvorbě a simulaci modelů. Význam migrace objektů se projeví v distribuovaném výpočetním prostředí [Ben87a, Ben87b], kdy objekt potřebuje silně komunikovat s jiným vzdáleným objektem. Komunikace přes síť bývá často velice časově drahá, a proto je výhodnější v takovýchto případech poslat objekt na cílový uzel, kde pak komunikace probíhá v rámci jednoho fyzického prostoru. Dalším důležitým důvodem migrace objektů bývá optimální rozdělení výpočetní zátěže v distribuovaném prostředí.

Migrace objektů v prezentovaném simulačním systému je realizována jako migrace příslušného metaobjektu (včetně všech jeho komponent) z jeho nadřazeného metaobjektu, tedy světa (viz hierachie metaobjektů 5.2.3). Operace migrace objektů představuje vytvoření klonu v cílovém světě, přesměrování referencí na klon a zrušení původního metaobjektu. Světy mohou tvořit spojený celek (viz koncepce spojených světů, kapitola 6.1.2) a tedy objekty mohou migrovat v rámci jednoho simulačního prostoru, nebo mezi různými simulačními prostory. Operace spojené s migrací objektu si můžeme rozdělit na několik fází:

1. *Příprava objektu*. Metaobjekt musí být suspendován a uspán. Operace uspání je velice důležitá, neboť během fáze přesunu musí být zajištěno blokování příjmu zpráv, než bude možné provést přesměrování zpráv na cílový klon. Jak již bylo řečeno v kapitole 5.6.3, každá reference na nekonstantní objektový typ je zajištěna přes proxyobjekty. V okamžiku uspání metaobjektu se veškeré příchozí zprávy ukládájí v těchto proxyobjektech.

2. *Transfer stavu objektu*. Stav metaobjektu je uložen (viz metoda *state*) a poslán do cílového prostoru.

3. *Vytvoření klonu*. V cílovém prostoru dojde k vytvoření daného metaobjektu a k jeho začlenění do příslušného nadřazeného metaobjektu. Poté se přenesený stav reflektuje na nově vytvořený metaobjekt (viz metoda *state:*), čímž vznikne klon původního metaobjektu.

4. *Přesměrování zpráv*. V tomto okamžiku je klon probuzen, čímž přebírá plnou kontrolu od původního metaobjektu. Zprávy, které byly až dosud blokovány v proxyobjektech, mohou být uvolněny. Proxyobjekty zajistí doručení zpráv probuzenému metaobjektu.

5. *Aktivace klonu a zrušení původního objektu*. Konečnou operací je aktivace probuzeného metaobjektu a zrušení původního metaobjektu.

Každá jedna reference na objekt je tvořena jedním proxyobjektem. Po probuzení začne klon přijímat zprávy prostřednictvím proxyobjektů. Při prvním zaslání zprávy po migraci objektu se proxyobjekt pokouší doručit zprávu původnímu objektu. Během této operace zjistí, že objekt byl přesunut do nového pracovního prostoru a získá jeho novou adresu. Od této chvíle jsou všechny zprávy přesměrovány na vzniklý klon objektu. Stejným způsobem se zajistí přesměrování zpráv objektům, které zasílá klon prostřednictvím svých uložených referencí.

6.4.3 Prototypování

Koncepty otevřené reflektivní architektury simulačního systému ve spojení s konceptem proxyobjektů umožňuje vytvářet prototypy aplikací a poměrně jednoduchým způsobem přecházet od modelů k jejich prototypům. Prototypování umožňuje reálné nasazení systému v oblastech s vysokým stupněm dynamičnosti vazeb, jako např. agentní systémy [MW97] či proces návrhu aplikací [Koč01c].

Při přechodu od modelu k prototypu jsou klíčové dva aspekty: reálný čas a spojení s objekty programovacího prostředí (v našem případě Smalltalku). Pro spojení objektů simulačního systému se Smalltalkem byly představeny proxyobjekty v kapitole 5.6, které zajišťují vzájemně transparentní komunikaci mezi simulačním systémem a Smalltalkem. Zasazení běhu modelu do reálného času spočívá ve změně časového chování modelu, tj. nahrazení manipulace s modelovým časem skutečnými akcemi. V současné době postrádá simulační systém metodologii tvorby prototypů, tj. množinu pravidel a kroků, které je potřeba vykonat během vývojového procesu. Metodologie prototypování je předmětem dalších výzkumů v oblasti reflektivního simulačního systému PNtalk. Jeho otevřená architektura, definující základní struktury a algoritmy, vytváří vhodné výchozí prostředí pro tento výzkum.

Kapitola 7

Závěr

Předkládaná disertační práce se zabývá modelováním, simulací a prototypováním složitých systémů s ohledem na dynamické aspekty vytvářených modelů a prototypů. Diskutuje objektově orientované přístupy k modelování a prototypování. Jedním z možných přístupů k řešení této problematiky je využití objektově orientované modifikace Petriho sítí, která kombinuje prostředky pro popis struktury (objektová orientace) a prostředky pro popis chování na formálním základě (Petriho sítě). Objektově orientované Petriho sítě (OOPN), díky své obecnosti, tvoří základní paradigma tvorby modelů, které bylo uvažováno v této práci.

Celým textem práce se prolínají myšlenky dynamické evoluce modelů (tj. schopnost modifikovat strukturu a chování modelu během jeho evoluce) a prototypování. Dynamická evoluce představuje výborný a jednoduchý prostředek pro aplikaci speciálních simulačních technik, modelování agentů apod. Ve spojení s prototypováním umožňuje inkrementální vývoj modelů softwarových systémů, jednoduchou převoditelnost formálního modelu na funkční prototyp apod. Stěžejním cílem práce byla diskuze principů otevřených systémů, jejich aplikace na principy evoluce OOPN a návrh otevřené architektury simulačního systému, který umožňuje uvažovat dynamické manipulace s modely.

Zvolený přístup k řešení. Práce prezentuje obecný přístup k architektuře simulačního systému pro dynamickou evoluci modelů založených na zobecněných principech evoluce OOPN. Vhodným přístupem pro návrh uvažované architektury se jeví principy otevřených implementací, zejména metaúrovňových architektur. Na jejich základě byla provedena analýza a zobecnění principů evoluce OOPN. Principy otevřených implementací a evoluce OOPN pak tvoří základ pro návrh otevřené architektury simulačního systému.

Dosažené výsledky. V disertační práci byla provedena diskuze a návrh architektury otevřeného reflektivního simulačního systému. Za zásadní výsledek lze považovat vybudování konceptuálního základu reflektivního systému, který umožňuje dynamický vývoj modelů, dynamickou evoluci modelů a kombinaci různých vyjadřovacích prostředků. Otevřená architektura navíc umožňuje zásahy do řídicích algoritmů a struktur, a to i z běžícího modelu. Navržené algoritmy a principy byly prototypově ověřeny. Nyní si zhodnotíme splnění tří cílů disertační práce, které byly stanoveny v úvodu.

1. *Návrh nové koncepce otevřené architektury simulačního systému.* Pro dosažení požadovaných vlastností byly využity reflektivní vlastnosti otevřených implementací a zavedena interoperabilita objektů OOPN se svým okolím, tj. s objekty Smalltalku. Prezentovaná architektura zahrnuje klasifikace výpočetních úrovní v systému, entit jednotlivých úrovní a vztahů mezi těmito entitami. Definuje tři základní výpočetní úrovně simulačního systému: doménovou úroveň, metaúroveň a meta metaúroveň. Doménová úroveň definuje evoluční prostor pro vlastní model, který je popsán prostřednictvím doménového paradigmatu. Metaúroveň vytváří evoluční prostor pro řízení

doménové úrovně. Meta metaúroveň definuje pravidla pro implementaci doménového paradigmatu a pro řízení meta úrovně. Speciální úroveň proxyobjektů umožňuje komunikaci objektů doménové úrovně s objekty systému Smalltalk.

2. *Zobecnění přístupu k popisu doménových paradigmat.* Klasifikace struktur popisujících běžící model (a tomu odpovídajících struktur popisujících model) na objekty, procesy a vlákna, včetně metaobjektového protokolu a algoritmů evoluce, umožňuje začleňovat nová paradigmata pod plnou kontrolu simulačního systému, kombinovat tato nová paradigmata se stávajícími (na úrovni objektů i metod) a dynamicky je přizpůsobovat měnícím se požadavkům.

3. *Návrh metodiky a zhodnocení aplikovatelnosti navrženého otevřeného simulačního systému.* Klasifikace struktur simulačního systému a klasifikace výpočetního chování simulačního systému přináší možnost multimodelování, kde jsou submodely vytvářeny s využitím různých prostředků a systémů (např. PNtalk, procesně orientovaná simulace ve Smalltalku, DEVS). Současně je tento koncept použitelný při tvorbě prototypů, kdy je výhodné část prototypu implementovat např. objekty ve Smalltalku a část prototypu modelovat např. Petriho sítěmi. Reflektivní architektura simulačního systému (dynamické změny chování a struktur) ve spojení s prototypováním umožňuje reálné nasazení systému v oblastech s vysokým stupněm dynamičnosti vazeb jako např. agentní systémy [MW97], proces návrhu aplikací [Koč01c], prototypování aplikací apod.

Další výzkum. Následující výzkum v oblasti dynamických simulačních systémů bude navazovat na výsledky a úvahy prezentované v této práci. Zásadním problémem je dokončení implementace systému, tj. přejít z prototypové realizace na plně funkční a použitelnou aplikaci. Dalším podstatným problémem je metodologie modelování prostřednictvím OOPN a metodologie tvorby heterogenních modelů (tj. kombinace různých paradigmat). S tím souvisí i výzkum spojený právě s kombinací různých vyjadřovacích prostředků a simulačních systémů, jako např. diskrétní i spojitá simulace ve Smalltalku, DEVS modely apod.

Další oblastí výzkumu aplikovatelnosti OOPN a navrhnuté otevřené architektury mohou být speciální simulační techniky (vnořená simulace, vícenásobná simulace apod.) Ve spojení s prototypováním jsou velmi zajímavými oblastmi výzkumu aplikace navrhnutého systému v softwarovém inženýrství, workflow modelování a aplikace umělé inteligence, např. modelování inteligentních agentů s využitím Petriho sítí.

Rejstřík pojmů

aktor, *aktivní objekt, který nepotřebuje externí podněty pro svou evoluci*, *14*

doménové paradigma, *paradigma popisující doménové objekty*, *2*

interoperabilita, *transparentní komunikace objektů různých prostředí*, *56*

metaúrovňová architektura, *klasifikace řízení komponent systému na vrstvy*, *16*
metaobjekt, *objekt řídící nebo reprezentující jiný objekt*, *17*
    doménový, *metaobjekt (master) reprezentující doménový objekt*, *39*, *44*
    master, *aktivní metaobjekt (viz aktor)*, *39*
    slave, *pasivní metaobjekt*, *39*

OOPN, *Objektově orientované Petriho sítě*, *24*
objekt
    doménový, *objekt reprezentující entitu modelovaného problému*, *16*
    konstantní, *objekt neměnící svůj stav*, *24*
    neprimitivní, *objekt, který může měnit svůj stav*, *24*
operace
    doménová, *operace z doménového protokolu*, *17*, *38*
    metaobjektová, *operace z metaobjektového protokolu*, *17*, *38*

PNtalk, *projekt zabývající se výzkumem reflektivních systémů a OOPN*, *30*
proces
    řídicí, *proces metaobjektu master*, *37*
    životní, *proces aktivního objektu*, *10*
    reflexe, *reflektování změn sémantiky či struktury objektu na doménové úrovni*, *18*
    reifikace, *změna sémantiky či struktury objektu*, *18*
    systémový, *struktura komponenty reprezentující doménový proces*, *42*
        OOPN, *struktura komponenty reprezentující proces OOPN*, *42*
procesní vazba, *vztah mezi vláknem a vyvolaným procesem*, *42*
protokol
    doménový, *rozhraní určené pro komunikaci s doménovým objektem*, *17*
    metaobjektový, *rozhraní určené pro komunikaci s metaobjektem*, *17*
        explicitní, 19
        implicitní, 20
        interní, 20
proxyobjekt, *metaobjekt zajišťující transparentnost komunikace*, *56*

reflektivita, *proces uvažování systému nad sebou samým*, *18*
    dynamická, statická, strukturální, výpočetní, založená na objektech nebo třídách, *18*
reifikace, *explicitní vyjádření doménových konceptů nebo mechanizmů*, *18*

svět, *metaobjekt zapouzdřující množinu doménových metaobjektů*, *39*, *51*
    spojený, *světy sdílející modelový čas*, *62*

úroveň
    doménová úroveň, *evoluční prostor pro doménové objekty*, *16*, *37*
    metaúroveň, *evoluční prostor pro doménové metaobjekty*, *17*, *37*
    meta metaúroveň, *evoluční prostor pro metaobjekty řídící objekty metaúrovně*, *37*

vlákno, *nejmenší jednotka evoluce*, *40*

zpráva systémová, *struktura zajišťující procesní vazbu*, *42*

Literatura

[Bar97] F. J. Barros. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(4):501–515, 1997.

[Ben87a] J. K. Bennett. Distributed Smalltalk: Inheritance and Reactiveness in Distributed Systems. Technical Report 87-12-04, Seattle, WA (USA), 1987.

[Ben87b] J. K. Bennett. The Design and Implementation of Distributed Smalltalk. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 318–330, New York, NY, 1987. ACM Press.

[Bri88] J. P. Briot. From Objects to Actors: Study of a Limited Symbiosis in Smalltalk 80. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 69–72, New York, NY, 1988. ACM Press.

[Bri99] J. Briot. Actalk : A Framework for ObjectOriented Concurrent Programming - Design and Experience, 1999.

[BZP00] T. Kim B. Zeigler and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, 2nd edition, 2000.

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'95)*, volume 27, pages 285–299. ACM Press, 1995.

[Chi96] S. Chiba. *A study of Compile-Time Metaobject Protocol*. PhD thesis, University of Tokyo, 1996.

[ČJV97] M. Češka, V. Janoušek, and T. Vojnar. PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling. *Computer Aided Systems Theory and Technology – EUROCAST'97*, 1333:591–610, 1997.

[CM93] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. *Lecture Notes in Computer Science*, 707, 1993.

[Eck02] B. Eckel. *Thinking in Java*. Prentice Hall PTR; 3rd edition, 2002.

[EJN97] R. Esser, J. Janneck, and M. Naedele. Applying an object-oriented petri net language to heterogeneous systems design. In *Petri Nets in Systems Engineering*, 1997.

[Fer89] J. Ferber. Computational reflection in class based object oriented languages. In *Proceedings of the 4th Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89)*, volume 24, pages 317–326. ACM Press, 1989.

[FJ89] B. Foote and R. E. Johnson. Reflective Facilities in Smalltalk-80. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 327–336, New York, NY, 1989. ACM Press.

[GC95] B. Gowing and V. Cahill. Making Meta-Object Protocols Practical for Operating Systems. In *4th International Workshop on Object Orientation in Operating Systems*, pages 52–55, Lund, Sweden, 1995.

[GC96] B. Gowing and V. Cahill. Meta-Object Protocol for C++: The Inguana Approach. In *Proceedings of Reflection'96*, 1996.

[GR89] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, 1989.

[HK03] M. Hrubý and R. Kočí. Modern Simulation Techniques and Tools. In *Proceedings of the International Workshop MOSMIC'2003*, pages 7–15, Zilina, SK, 2003. FRI ZU.

[HKPR02] M. Hrubý, R. Kočí, P. Peringer, and Z. Rábová. Tools for Creating of Multimodels. *Kybernetes: The International Journal of Systems & Cybernetics*, 31(9):1391–1400, 2002.

[Hru04] M. Hrubý. *Prostředí pro modelování heterogenních systémů*. PhD thesis, Brno University of Technology, 2004.

[Hun97] J. Hunt. *Smalltalk and Object Orientation: An Introduction*. Springer Verlag, 1997.

[Jan95] V. Janoušek. PNtalk: Object Orientation in Petri nets. In *Proc. of European Simulation Multiconference ESM'95*, pages 196–200, Prague, Czech Republic, 1995.

[Jan98] V. Janoušek. *Modelování objektů Petriho sítěmi*. PhD thesis, Brno University of Technology, 1998.

[JK02] V. Janoušek and R. Kočí. PNtalk - An Open System for Prototyping and Simulation. In *Proceedings of The 28th ASU Conference*, pages 133–146, Brno, CZ, 2002. FIT VUT.

[JK03] V. Janoušek and R. Kočí. PNtalk: Concurrent Language with MOP. In *Proceedings of the CS&P'2003 Workshop*, pages 271–282, Warsawa, PL, UW, 2003.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metalevel Protocol*. MIT Press, 1991.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, Ch.Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Koč01a] R. Kočí. Distribuovaná simulace OOPN ve Smalltalku. Technical report, Brno University of Technology, 2001.

[Koč01b] R. Kočí. Prospects of the PNtalk system in the Smalltalk environment. In *Proceedings of 7th Conference Student FEI 2001*, pages 338–342, Brno, CZ, 2001. VUT.

[Koč01c] R. Kočí. The PNtalk System - a Technique for Object Oriented Modelling. In *Proceedings of XXIIIrd International Autumn Colloquium*, pages 151–158, Ostrava, CZ, 2001. MARQ.

[Koč03] R. Kočí. The Open Architecture of the PNtalk System. In *Proceedings of the International Conference and Competition - Student EECIT 2003*, pages 358–362, Brno, CZ, 2003.

[KR02] R. Kočí and Z. Rábová. The PNtalk System and Interoperability. In *Proceedings of International Conference MOSIS '02*, pages 73–80, Ostrava, CZ, 2002. MARQ.

[KR03] R. Kočí and Z. Rábová. Purposes of the PNtalk System. In *Proceedings of International Conference MOSIS '03*, pages 149–156, Ostrava, CZ, 2003. MARQ.

[Kri96] P. Krief. *Prototyping with objects*. T.J.Press Ltd, Padstow, Cornwall, UK, 1996.

[KV01] R. Kočí and T. Vojnar. A PNtalk-based Model of a Cooperative Editor. In *Proceedings of the 35th Spring International Conference on Modelling and Simulation of Systems – MOSIS 2001*, pages 165–172, Hradec nad Moravicí, CZ, 2001. MARQ.

[KW01] E. Kindler and M. Weber. The Petri Net Kernel - An infrastructure for building Petri net tools. *International Journal on Software Tools for Technology Transfer*, 3(4):486–497, 2001.

[Lak95a] C. A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *Proceedings of the Application and Theory of Petri Nets 1995*, volume 935, pages 278–297. Springer-Verlag, Berlin, Germany, 1995.

[Lak95b] C. A. Lakos. The Object Orientation of Object Petri Nets. In *Proceedings of the first international workshop on Object-Oriented Programming and Models of Concurrency - 16th International Conference on Application and Theory of Petri Nets*, pages 1–14, 1995.

[Lak96] C. Lakos. Towards a Reflective Implementation of Object Petri Nets. In *Proceedings of TOOLS Pacific*, pages 129–140, Melbourne, Australia, 1996.

[Lak97] C. Lakos. Object Oriented Modelling with Object Petri Nets. In *Advances in Petri Nets, LNCS*, Berlin, 1997. Springer.

[Mae87] P. Maes. Computational reflection. Technical report, Artifical Inteligence Laboratory, Vrije Universiteit Brusel, 1987.

[McA95] J. McAffer. Meta-level programming with CodA. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952, pages 190–214. Springer-Verlag, 1995.

[MMWY92] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 127–144, New York, NY, 1992. ACM Press.

[Mol95] D. Moldt. OOA and Petri Nets for System Specification. In *Application and Theory of Petri Nets; Lecture Notes in Computer Science, 16th International Conference*. Italy, 1995.

[MW97] D. Moldt and F. Wienberg. Multi-Agent-Systems Based on Coloured Petri Nets. In *ICATPN*, pages 82–101, 1997.

[NF98] N. Neves and W. K. Fuchs. RENEW: A Tool for Fast and Efficient Implementation of Checkpoint Protocols. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1998.

[Paw98] R. Pawlak. Metaobject Protocols For Distributed Programming. Technical report, Laboratoire CNAM-CEDRIC, Paris, 1998.

[Per96] P. Peringer. *Hierarchické modelování na bázi komunikujících objektů*. PhD thesis, Brno University of Technology, 1996.

[Phi95] M. Philippsen. Imperative Concurrent Object-Oriented Languages: An Annotated Bibliography. Technical Report TR-95-049, Berkeley, CA, 1995.

[PNt04] PNtalk Home Page, http://www.fit.vutbr.cz/~koci/pntalk, 2004.

[Poo87] R.J. Pooley. *An Introduction to Programming in Simula (Computer Science Texts)*. Alfred Waller Ltd, 1987. Kniha dostupná na: http://www.cee.hw.ac.uk/~rjp/bookhtml/.

[Ren04] The Renew Home Page, http://www.renew.de/, 2004.

[SB94] C. Sibertin-Blanc. Cooperative Nets. In Robert Valette, editor, *Proceedings of Application and Theory of Petri Nets*, volume 815, pages 471–490. Springer-Verlag, Berlin, Germany, 1994.

[Sel04] The Self Home Page, http://research.sun.com/self/, 2004.

[Str92] B. Stroustrup. Run Time Type Identification for C++. In *Proceedings of the USENIX C++ Conference*, 1992.

[Str00] B. Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley Pub Co;, 2000.

[Tun04] The TUNES Project for a Free Reflective Computing System, http://tunes.org, 2004.

[Uhr01] A. M. Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach. In *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, volume 11, pages 206–232. 2001.

[UML04] The UML Home Page, http://www.uml.org, 2004.

[Val98] R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, volume 1420, pages 1–25. Springer-Verlag, Berlin, 1998.

[Voj01] T. Vojnar. *Towards Formal Analysis and Verification over State Spaces of Object-Oriented Petri Nets*. PhD thesis, Brno University of Technology, 2001.

[Yok92a] Y. Yokote. The apertos reflective operating system: The concept and its implementation. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 414–434, New York, NY, 1992. ACM Press.

[Yok92b] Y. Yokote. The Apertos Reflective Operating System: The Concept and its Implementation. In *Proceedings of the 7th Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'92)*, volume 27, pages 414–434. ACM Press, 1992.

[ZH] M. Zapf and A. Heinzl. Techniques for Integrating Petri Nets and Object-Oriented Concepts.

[Zim96] C. Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.

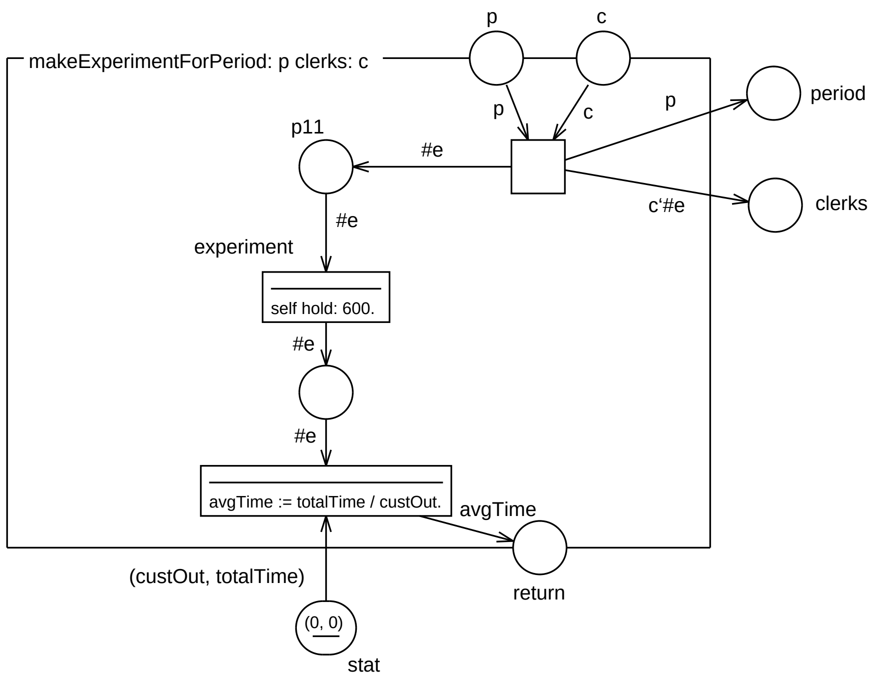
Příloha A

Syntax jazyka PNtalk

Textová verze jazyka PNtalk definuje kromě syntaxe inskripčního jazyka též syntax popisu struktury Objektově orientované Petriho sítě. Syntax popisů sítí, syntax struktury sítí a celého systému tříd formálně definujeme rozšířenou Backus-Naurovou formou. Zápis `[ ... ]` znamená, že "..." se může vyskytnout nepovinně, `[ ... ]*` znamená libovolně násobný nepovinný výskyt "...", `[ ... ]+` znamená výskyt "..." jednou nebo víckrát, `... | ... [ | ... ]*` znamená výběr jedné z variant. Řetězce v uvozovkách odpovídají lexikálním symbolům. Výchozí symbol je `classes`.

```
classes: [class]* "main" id [class]*
class: "class" classhead [objectnet] [methodnet|constructor|sync]*
classhead: id "is_a" id

objectnet: "object" net
methodnet: "method" message net
constructor: "constructor" message net
sync: "sync" message [cond] [precond] [guard] [postcond]
message: id | binsel id | [keysel id]+
net: [place|transition]*

place: "place" id"(" [initmarking] ")" [init]
init: "init" "{" initaction "}"
initmarking: multiset
initaction: [temps] exprs

transition: "trans" id [cond] [precond] [guard] [action] [postcond]
cond: "cond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
precond: "precond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
postcond: "postcond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
guard: "guard" "{" expr3 "}"
action: "action" "{" [temps] exprs "}"
arcexpr: multiset

multiset: [n "`"] c ["," [n "`"] c]*
n: [dig]+ | id
c: literal | id | list
list: "(" [c ["," c]* ["|" [id | list] ]] ")"
```

```
temps: ″|″ [id]* ″|″
unit: id | literal | ″(″ expr ″)″
unaryexpr: unit [ id ]+
primary: unit | unaryexpr
exprs: [expr ″.″]* [expr]
expr: [id ″:=″]* expr2
expr2: primary | msgexpr [ ″;″ cascade ]*
expr3: primary | msgexpr
msgexpr: unaryexpr | binexpr | keyexpr
cascade: id | binmsg | keymsg
binexpr: primary binmsg [ binmsg ]*
binmsg: binsel primary
binsel: selchar[selchar]
keyexpr: keyexpr2 keymsg
keyexpr2: binexpr | primary
keymsg: [keysel expr2]+
keysel: id″:″
literal: number | string | charconst | symconst | arrayconst
arrayconst: ″#″ array
array: ″(″ [number | string | symbol | array | charconst]* ″)″
number: [-][[dig]+ ″r″] [hexDig]+ [″.″ [hexDig]+] [″e″[″-″][dig]+].
string: ″'″[char]*″'″
charconst: ″$″char
symconst: ″#″symbol
symbol: id | binsel | keysel[keysel]* | string
id: letter[letter|dig]*
selchar: ″+″ | ″-″ | ″*″ | ″/″ | ″~″ | ″|″ | ″,″ | ″<″ | ″>″ | selchar2
selchar2: ″=″ | ″&″ | ″\″ | ″@″ | ″%″ | ″?″ | ″!″
hexDig: ″0″..″9″ | ″A″..″F″
dig: ″0″..″9″
letter: ″A″..″Z″ | ″a″..″z″
char: letter | dig | selchar | ″[″ | ″]″ | ″{″ | ″}″ | ″(″ | ″)″ | char2
char2: ″_″ | ″^″ | ″;″ | ″$″ | ″#″ | ″:″ | ″.″ | ″-″ | ″`″
```



Obrázek A.1: Příklad metody OOPN

Nyní pro ilustraci uvedeme příklad OOPN, specifikované v jazyce PNtalk, a to jak graficky, tak textově. Grafická podoba OOPN je na obr. A.1. Textová podoba tohoto příkladu je zde. Všiměte si místa se jménem *nonename87* – toto místo nemá v síti jméno, a proto mu bylo přiřazeno systémem:

```
method makeExperimentForPeriod: p clerks: c
    place p ()
    place nonename97()
    place c()
    place p11()
    place return()
    trans experiment
        precond p11( #e )
        action { self hold: 600. }
        postcond nonename97( #e )
    trans nonename95
        precond p ( p ), c( c ), semaphore( #e )
        postcond p11( #e ), period( p ), clerks( c`#e )
    trans update_stat
        precond nonename97( #e ), stat( (custOut, totalTime) )
        action { avgTime := totalTime / custOut. }
        postcond return( avgTime ), semaphore( #e )
```

Příloha B

Základní syntaxe a sémantika jazyka Smalltalk

Tato příloha slouží pro základní seznámení s konstrukcemi jazyka Smalltalk, které se vyskytují v popisech algoritmů. Hlubší seznámení s jazykem a systémem Smalltalk může čtenář najít v [GR89]. Ve Smalltalku existuje jediný výpočetní mechanismus – *zasílání zpráv*. Všechny operace od vyvolání metody objektu až po řídicí struktury jsou realizovány zasíláním zpráv objektům. Objekt je jediný datový typ, který lze ve Smalltalku vytvořit. Objekty se vždy vytváří dynamicky, přičemž reference na vytvořené objekty jsou uschovány v proměnných. Smalltalk je slabě typovaný jazyk v tom smyslu, že se explicitně nedeklarují typy proměnných. Typ vytvořeného objektu lze chápat jako třídu, ze které byl odvozen.

Základní objekty Smalltalku

- **Čísla.** Čísla lze ve Smalltalku zapisovat běžným způsobem (1, -1, 1.5e-6, ...) a lze na ně uplatnit běžné operátory a funkce ($<$, $>$, +, -, sin, ...).
- **Řetězce**. Zapisují se v apostrofech ('Text'), lze nad nimi provádět operace porovnání ($<$, $>$, =, ...) a konkatenace (,).
- **Symboly**. Symbol je definovám svým jménem, uvozuje se znakem # před jménem (#blue). Nad symboly lze provádět operace porovnání.
- **Znaky**. Uvozují se znakem $ (např. $a představuje znak *a*).
- **Boolovské konstanty**. Jedná se o instance tříd *True* a *False*, zapisované pomocí literálů *true* a *false*.
- **Nedefinovaný objekt**. Jeho identifikace je *nil*. Všechny deklarované a dosud nepoužité proměnné mají implicitní hodnotu *nil* (referují objekt nil). Nad každým objektem se dají použít operace *isNil* a *notNil*.
- **Konstantní inicializace pole**. Vytváří se konstrukcí #(seznam_prvků), kde jednotlivé prvky seznamu prvků pole jsou odděleny mezerou (např. #(1 2 #red)). Pole ve Smalltalku není homogenní. N-tý prvek lze zpřístupnit konstrukcí: pole *at:* n.

Používané symboly

- $\uparrow$ reprezentuje ukončení metody a definuje návratovou hodnotu. Před ukončením se provede vyhodnocení výrazu za znakem $\uparrow$ a tento výsledek se použije jako návratová hodnota metody (např. $\uparrow$ *nil*).

- ← reprezentuje přiřazení objektu (výrazu) do proměnné (např. *variable ← a + 1*).
- *»* odděluje název metodu od názvu třídy (např. *PNtalkThread » id*).
- . odděluje jednotlivé příkazy.
- *;* odděluje jednotlivé zprávy zasílané stejnému příjemci (např. zápis *methodContext block; activate.* má stejný význam jako zápis *methodContext block. methodContext activate.*)

Provádění operací Vyvolání operací probíhá ve Smalltalku pomocí zasílání zpráv objektu. Zaslání zprávy sestává z příjemce zprávy a ze zprávy samotné. Zpráva je složena ze *selektoru* (identifikace) a *argumentů*. Příjemce i argumenty zprávy jsou objekty, mohou to být i výsledky jiných zaslání zpráv. Rozeznáváme tři typy zpráv, které uvedeme v pořadí vzrůstající priority ve výrazech:

1. *Unární zprávy*. Nemají žádný argument (např. *2 sin.*).
2. *Binární zprávy*. Mají jeden argument (např. *4 + 2.*).
3. *Zprávy s klíčovými slovy*. Mají alespoň jeden argument (např. *pole at: 1 put: objekt.*).

Pokud tedy chceme např. vytvořit instanci třídy, zašleme objektu typu třída zprávu *new* (např. *object ← Class new.*).

Super a self Klíčová slova *super* a *self* jsou zvláštní proměnné Smalltalku, které nelze modifikovat a jsou vždy definované:

- *self* ukazuje na aktuální objekt, ve kterém se nachází,
- *super* referuje také aktuální objekt, ale při jeho použití se začíná prohledávat hierarchie dědičnosti při hledání implementace metody až od přímého předka objektu.

Deklarace proměnných Deklarace proměnných se provádí mezi dvě svislé čáry, např. | *var tmp* | je deklarací dvou proměnných *var* a *tmp*.

Kódové bloky Kódové bloky jsou objekty, které zapouzdřují množinu příkazů. Mohou mít lokální proměnné, vstupní parametry a dají se volat rekurzivně. Návratovou hodnotou bloku je výsledek posledního provedeného příkazu. Syntaxe a použití bloků je následující (součet 2*x+x):

*block ← [:in | | tmp | tmp ← 2 * in. tmp + in.].*
result ← block value: 2.

Podmíněné výrazy Všechny řídicí struktury (a tedy i podmíněné výrazy) se provádí zasíláním zpráv s využitím bloků. V podmíněných strukturách se využívá instancí tříd *True* a *False*, kterým lze zasílat zprávy *ifTrue:*, *ifFalse:* a *ifTrue:ifFalse:*. Parametry zpráv jsou bloky, které jsou vyhodnoceny při splnění či nesplnění podmínky. Hodnota vyhodnoceného bloku se pak stává hodnotou celého výrazu. Následující příklady simulují funkci *y:=abs(x)*:

*x > 0 ifTrue: [y ← x] ifFalse: [y ← -1 * x].*
*y ← x > 0 ifTrue: [x] ifFalse: [-1 * x].*

Podmínky se dají spojovat pomocí operátorů logické AND (&) a logické OR (|). Negace podmínky – zaslání zprávy *not*.

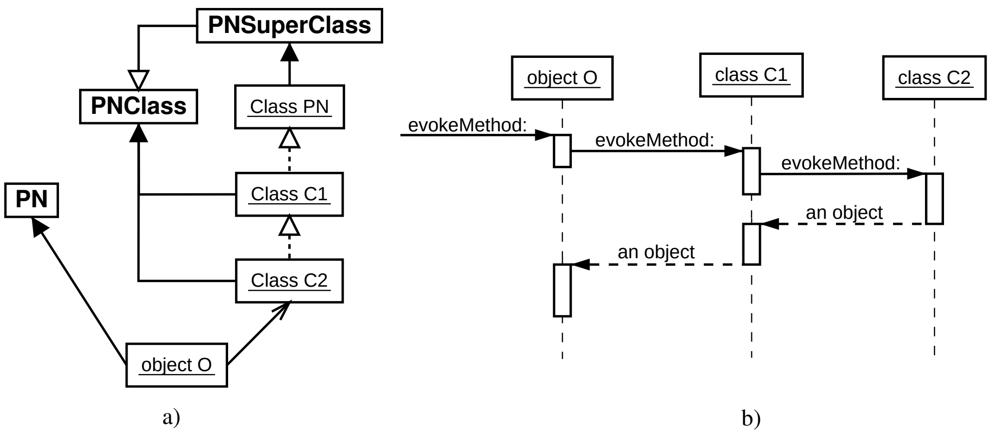
Příloha C

Objektově orientované modelování s OOPN

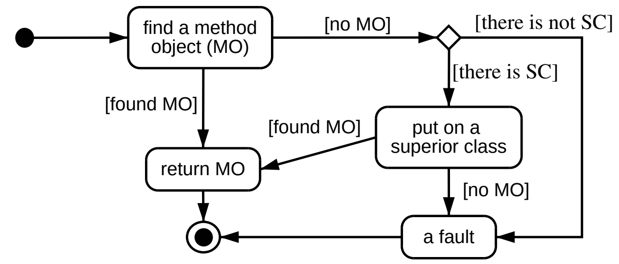
Možnosti objektově orientovaného modelování s použitím OOPN si přiblížíme na modelu prezentovaném v článku [Koč01c]. Jedná se o modelování vybrané části systému PNtalk, a to nejprve s použitím diagramu tříd a diagramu aktivit z prostředků nabízených UML a poté s použitím OOPN. Pro jednoduchost se vybraný model věnuje vyhledávání metody v hierarchii tříd podle dědičnosti. Objekt je instancí třídy. V okamžiku, kdy objekt přijímá zprávu, pokusí se nalézt odpovídající metody v této třídě. Pokud je pokus neúspěšný, pokračuje ve vyhledávání v nadřazené třídě. Tento postup se opakuje až po kořenovou třídu hierarchie (*PN*), která neimplementuje žádnou metodu. V případě úspěchu vrací příslušná třída metaobjekt, reprezentující hledanou metodu. V případě neúspěchu vrací objekt *nil*. Uvedený princip včetně vztahu k třídám Smalltalku, které tyto principy implementují, je znázorněn na obrázku C.1.

Na obrázku C.2 můžeme vidět diagram aktivit popisující chování námi modelovaného problému. Zkratka *SC* na obrázku znamená *Superior Class* (nadřazená třída); zkratka *MO* znamená *Method Object* (objekt metody). Pokud je metoda nalezena, je její objekt vrácen, pokud metoda není nalezena, pokračuje se ve vyhledávání v nadřazené třídě. Pokud nadřazená třída neexistuje, proces vyhledávání končí s chybou.

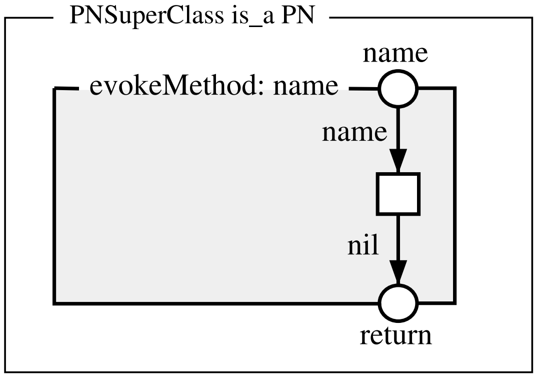
Nyní si ukážeme model problému s využitím OOPN. Třída *PNClass* modeluje doménovou třídu OOPN. Na obrázku C.4 je znázorněna její metoda *evokeMethod:*, která modeluje vyhledávání metody. Jméno hledané metody dostává jako argument (*name*). Současně objekt, ve kterém je metoda umístěna, zná referenci na modelovanou nadřazenou třídu (místo *superior*). Pro vlastní vyhledání metody slouží



Obrázek C.1: Schéma dědičnosti v OOPN

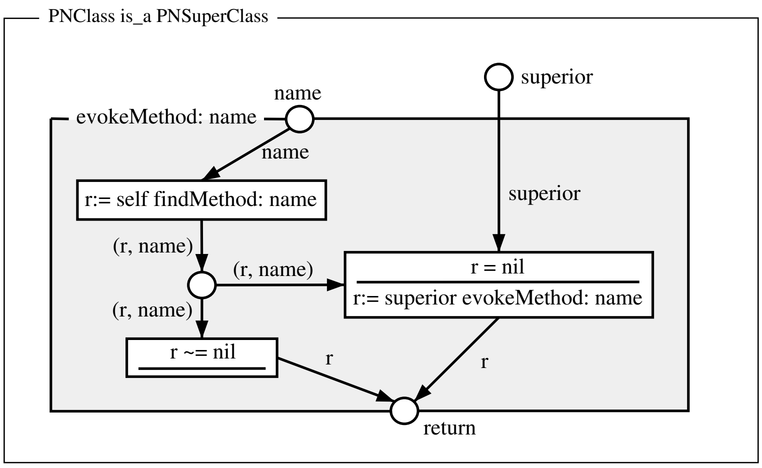


Obrázek C.2: Diagram aktivit pro vyhledávání metod v OOPN



b)

Obrázek C.3: Model vyhledávání metod v kořenové třídě PN



Obrázek C.4: Model vyhledávání metod v třídě OOPN operace *self findMethod: name*. Pokud je hledání úspěšné, vrátí se nalezená metoda (proměnná *r*). V opačném případě se vyvolá metoda *evokeMethod:* modelu nadřazené třídy.

Jak jsme si již řekli, na vrcholu hierarchie dědičnosti OOPN je třída *PN*. Pokud se algoritmus vyhledávání metod OOPN dostane až sem (viz obrázek C.3), vrátí algoritmus indikaci neúspěchu (objekt *nil*). Uvedený příklad plně zastupuje funkci diagramu aktivit. Pokud je model úplný (např. v našem příkladu chybí model metody *PN*), může být navíc správnost modelu ověřena simulací

Příloha D

Metaprotokol

Součástí systému je i definovaný základ metaprotokolu pro jednotlivé metaobjekty. V následující části si tento protokol popíšeme. Pro zjednodušení budeme všechny metaobjekty nazývat *komponenta*. Komponentu, která je součástí jiné komponenty, budeme nazývat *subkomponenta*. Takže např. subkomponentou procesoru je svět apod.

D.1 Společný metaprotokol

Následující metaprotokol je společný pro všechny metaobjekty, které jsou uvedeny v této příloze. Představuje základní operace pro manipulaci se subkomponentami příslušného metaobjektu. Některé metaobjekty mohou navíc implementovat speciální operace pro speciální manipulaci se subkomponentami.

Společný metaprotokol	
id	vrací lokální identifikátor komponenty
gid	vrací globální identifikátor komponenty
listComponents	výpis (kolekce identifikací) subkomponent
componentID:	vrací subkomponentu podle její identifikace
addComponent:	vloží subkomponentu
removeComponentNamed:	zruší subkomponentu podle její identifikace
clone	vytvoří identický klon komponenty
state	uložení stavu v kolekci
state:	obnovení stavu z kolekce
xmlState	uložení stavu v xml formátu
xmlState:	obnovení stavu z xml formátu

D.2 Metaprotokol systémových metaobjektů

Za systémové metaobjekty jsou považovány ty, které reprezentují ucelenou část systému, tj. doménová třída (*PNClass*), doménový objekt (*PNObject*) a svět (*World*). Následující popis představuje metaprotokol *metatřídy* třídy *PNClass*, umožňující vytvářet či kompilovat nové doménové třídy:

PNClass class metaprotokol	
newNamed:superClass:	vytvoří novou OOPN třídu (s daným jménem a nadtřídou)
compile:	zkompiluje a vytvoří novou OOPN třídu ze zdrojového textu
named:	vrací OOPN třídu podle jména

Metaprotokol třídy *PNClass* umožňuje vytvářet instance doménové třídy, kompilovat metody a volat metody doménové třídy.

PNClass metaprotokol	
name	vrací jméno třídy
compileMethod:	zkompiluje a vytvoří novou metodu ze zdrojového textu
new	vytvoří instanci OOPN třídy a vloží ji do aktuálního světa
newIn:	vytvoří instanci OOPN třídy a vloží ji do zadaného světa
performMessage:forObject:	vyhledá metodu odpovídající zprávě, vytvoří její instanci a zařadí ji do příslušného objektu
performPort:forObject:	vyhledá port odpovídající zprávě, provede jeho test (speciální vlastnost OOPN)

Následující metaprotokoly tříd *World* a *PNObject* umožňují řídit evoluci jednotlivých objektů a celého světa.

World metaprotokol	
test	provede test všech komponent (objektů) světa
step	provede jeden krok nad všemi komponentami (objekty) světa a poté operaci test
suspend	suspenduje všechny komponenty (objekty) světa
resume	aktivuje všechny komponenty (objekty) světa
cloneAndSuspend	vytvoří identický klon světa, okamžitě po jeho vytvoření jej suspenduje

PNObject metaprotokol	
test	provede test proveditelnosti přechodů
step	provede jeden krok nad objektem
suspend	suspenduje objekt
resume	aktivuje objekt
yourClass	vrací metaobjekt doménové třídy příslušející objektu
world	vrací metaobjekt světa, do kterého je *PNObject* zařazen
compile	zkompiluje metodu (síť) a vytvoří její instanci v objektu
cloneAndSuspend	vytvoří identický klon světa, okamžitě po jeho vytvoření jej suspenduje
listDependencies	vrací kolekci procesních vazeb vláken
listInvocatedMethods	vrací kolekci procesních vazeb vyvolaných metod

D.3 Metaprotokol komponent metaobjektu PNObject

Následující protokoly metaobjektů, které tvoří jádro doménového metaobjektu *PNObject*, umožňují přístup k interním aspektům metaobjektů, jejich modifikaci a také umožňují řízení evoluce příslušných komponent, tj. aktivace událostí.

PNtalkProcess metaprotokol	
name	vrací jméno metody spojené s procesem
object	vrací metaobjekt doménového objektu, který je nadřazenou komponentou
compile:	zakompiluje příslušnou část sítě do běžícího procesu, jako parametr je zdrojový kód sítě
listPlaces	vrací kolekci identifikací míst v procesu
listTransitions	vrací kolekci identifikací přechodů v procesu
listFirableTransitions	vrací kolekci událostí, tj. identifikací přechodů, které lze spustit
listThreads	vrací kolekci identifikací vláken v procesu
listPlacesNames	vrací kolekci jmen míst v procesu
listTransitionsNames	vrací kolekci jmen přechodů v procesu
placeNamed:	vrací místo nalezené podle jména
transitionNamed:	vrací přechod nalezené podle jména

PNtalkTransition metaprotokol	
name	vrací jméno přechodu
process	vrací metaobjekt procesu, který je nadřazenou komponentou
listBindings	vrací kolekci možných navázání (potenciálních událostí) přechodu
fire	spustí přechod (vytvoří vlákno) pro náhodně vybrané navázání
fire:	spustí přechod (vytvoří vlákno) pro zadané navázání

PNtalkThread metaprotokol	
process	vrací metaobjekt procesu, který je nadřazenou komponentou
listVariables	vrací kolekci proměnných svázaných s vláknem
fire	provede další krok v evoluci vlákna (následující příkaz)
variableNamed:put:	změní hodnotu proměnné daného jména

Komponenta *PNtalkPlace* reprezentuje místo v sítí. Její protokol je podobný protokolu objektu *Bag* ve Smalltalku.

PNtalkPlace metaprotokol	
name	vrací jméno přechodu
process	vrací metaobjekt procesu, který je nadřazenou komponentou
content	vrací kolekci s obsahem místa, prvkem kolekce je speciální dvojice – $pair = (multiplicity, object)$
addObject: mult:	přidá objekt do místa s příslušnou násobností
removeObject: mult:	odebere objekt do místa s příslušnou násobností
at:	vrátí prvek (*pair*) z místa na příslušné pozici

D.4 Metaprotokol proxy metaobjektů

V systému se rozlišují tři typy proxy metaobjektů podle toho, který typ objektu zastupují. Proxy *PNtalkObjectProxy* zastupuje doménový objekt, proxy *PNtalkMetaProxy* zastupuje doménový metaobjekt a proxy *PNtalkSmalltalkProxy* zastupuje objekt Smalltalku.

Společný metaprotokol	
meta	Vrací meta-proxy objekt (PNtalkMetaProxy)
domain	Vrací domain-proxy objekt (PNtalkObjectProxy)
reify	Vrací zastupovaný objekt (PNObject, objekt Smalltalku)
newFor:	Vytvoří nový proxyobjekt pro zadaný objekt
perform:withArguments:	Příjem zprávy identifikované podle jména (1. parametr) s argumenty (2. parametr)
isConstant	Test typu zastupovaného objektu (konstantní typ)
isSmalltalk	Test typu zastupovaného objektu (objekt Smalltalku)
isOOPN	Test typu zastupovaného objektu (objekt OOPN)
isPrimitive	Test typu zastupovaného objektu (speciální primitivní typ)

PNtalkObjectProxy metaprotokol	
doesNotUnderstand:	speciální metoda nabízená systémem Smalltalk je vyvolaná, pokud objekt nerozumí zasílané zprávě vytvoří strukturu *PNtalkMessage* a zavolá metodu performMessage:
performMessage:	předá zprávu zastupovanému objektu prostřednictvím řídicí zprávy *performDomainMessage:*
thisMessageFinished:with:	zprostředkovává přenos návratového objektu z ukončeného zpracování zprávy

PNtalkMetaProxy metaprotokol	
doesNotUnderstand:	speciální metoda nabízená systémem Smalltalk je vyvolaná, pokud objekt nerozumí zasílané zprávě vytvoří strukturu *PNtalkMessage* a zavolá metodu performMessage:
performMessage:	předá zprávu zastupovanému objektu prostřednictvím řídicí zprávy *performMessage:*

PNtalkSmalltalkProxy metaprotokol	
performMessage:	předá zprávu zastupovanému objektu prostřednictvím standardních prostředků Smalltalku *perform:withArguments:* zpráva je neblokující se zajištěním vyzvednutí návratového objektu po skončení volané metody