VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta elektrotechniky a informatiky

Modelování objektů Petriho sítěmi

Disertační práce

Autor: Ing. Vladimír Janoušek
Ústav informatiky a výpočetní techniky FEI VUT v Brně

Školitel: Doc. RNDr. Milan Češka, CSc.
Ústav informatiky a výpočetní techniky FEI VUT v Brně

17. prosince 1998

Abstrakt

Petriho sítě poskytují výkonné prostředky pro modelování kauzality, nedeterminismu a paralelismu v diskrétních systémech. Protože jsou ve své podstatě matematickým modelem, nabízejí teorii, která může být úspěšně využita pro verifikaci modelů. Proveditelnost Petriho sítí je předurčuje pro simulaci a rychlé prototypování. Tyto pozitivní vlastnosti Petriho sítí se však v plné míře projeví pouze u nepříliš rozsáhlých a většinou velmi abstraktních modelů. Podrobnější realistické modely jsou díky absenci strukturovacích mechanismů v Petriho sítích komplikované a nepřehledné. Proto se pro detailní modelování obvykle používají obecné programovací jazyky. Vývoj v programovacích jazycích totiž odráží snahu pomocí vhodných jazykových konstrukcí vyjádřit (formalizovat) základní principy modelování složitých systémů – abstrakci a zapouzdření. Tento vývoj vedl k prosazení objektové orientace v programovacích jazycích, databázích i v metodách analýzy, návrhu a implementace programových systémů. Tyto velmi obecné a univerzální postupy však prozatím postrádají jasný matematický základ, který by umožnil formální analýzu.

Cílem této práce je ukázat, že diskrétní systémy lze modelovat v souladu s principy objektové orientace, aniž bychom se museli vzdát výhod, které přinášejí Petriho sítě. Na základě analýzy různých strukturovacích mechanismů, zavedených nad Petriho sítěmi, jsou vytypovány základní konstrukce, vhodné pro zavedení objektové orientace do Petriho sítí, která umožní jejich využití jako plnohodnotného programovacího nebo simulačního jazyka.

Práce přináší původní koncept spojení Petriho sítí a objektů. Je zde definována objektově orientovaná Petriho síť a odpovídající graficko-textový jazyk PNtalk. Tento formalismus je založen na aktivních objektech, zapouzdřujících shluky procesů a komunikujících mechanismem vzdáleného volání procedur. Pro podporu simulačního modelování objektově orientovanými Petriho sítěmi je navržen odpovídající počítačový nástroj. Teoretickým výsledkem práce je formální definice základních pojmů objektově orientovaného modelování, opírající se o teorii Petriho sítí. Tím je vytvořen rámec pro teoretické úvahy o aktivních objektech. Praktickým výsledkem je definice jazyka a návrh systému, který umožní objektově orientované modelování a simulaci paralelních systémů.

Obsah

Kapitola 1

Úvod

Modelování systémů je činnost, která člověku umožňuje uvažovat o reálném světě a na základě získaných poznatků ho cílevědomě ovlivňovat. Systémem obvykle rozumíme abstrakci reality, zaměřující se jen na ty skutečnosti, které jsou relevantní pro naše zkoumání. Systém má strukturu – skládá se ze složek (prvků systému), mezi nimiž existují jisté vazby (vztahy, relace). Obecně můžeme systémy považovat za dynamické, měnící se v čase. Pak nás zajímá i chování systému. V dynamickém systému se mohou měnit jak vazby, tak i množina prvků systému.

Modelem rozumíme uměle vytvořený systém, který je jistým zjednodušením originálu (modelovaného systému). Mezi originálem a jeho modelem existuje homomorfní zobrazení.[1] Lze hovořit o abstraktních (myšlenkových, teoretických) modelech, nad kterými můžeme vést logické úvahy, a o simulačních (konkrétních, fyzicky realizovaných, proveditelných) modelech, s kterými můžeme provádět simulační experimenty. Některé modely, realizované prostřednictvím počítačů, mohou patřit do obou těchto kategorií současně, protože poskytují teorii, umožňující logickým odvozováním dokazovat vlastnosti modelu, a zároveň je možné s tímtéž modelem provádět simulační experimenty. Zatímco logické odvozování vlastností umožňují formální modely, založené na nějakém jednoduchém, matematicky dobře zpracovatelném formalismu, počítačovou simulaci umožňují všechny proveditelné modely, zapsané v nějakém programovacím jazyku.

Složité dynamické systémy, jako jsou počítačové systémy, pružné výrobní systémy, workflow systémy, komunikační protokoly apod. jsou příklady systémů, které obvykle modelujeme jako diskrétní systémy, tj. systémy s diskrétní množinou stavů.

Tato práce přináší nový pohled na modelování složitých diskrétních dynamických systémů, založený na Petriho sítích a objektové orientaci.

1.1 Motivace

1.1.1 Petriho sítě

Petriho sítě představují populární formalismus pro modelování diskrétních (paralelních[2]) systémů, který spojuje výhody srozumitelného grafického zápisu a možnosti simulace s dobrou formální analyzovatelností. Srozumitelnost a analyzovatelnost Petriho sítí je dána jejich jednoduchostí. Model je popsán místy (places), která obsahují stavovou informaci ve formě značek (tokens), přechody (transitions), které vyjadřují možné změny stavu, a hranami (arcs), propojujícími místa a přechody navzájem. Existuje celá řada typů Petriho sítí, od C-E sítí (Condition-Event nets) [Pet62], přes P/T sítě a jejich speciální podtřídy, až po vysokoúrovňové sítě (High-

[1] Více prvků originálu může být reprezentováno jedním prvkem modelu.

[2] Sledujeme-li v diskrétním systému procesy jako sekvence událostí, spjaté s dílčími částmi systému, hovoříme o paralelním systému. Jde jen o specifický pohled na diskrétní systém.

Level Petri nets), jako jsou predikátové (Predicate-Transition Petri nets) [Gen87] a barvené sítě (Coloured Petri nets) [Jen92], umožňující podrobně modelovat kromě řídicí struktury systému i zpracování dat. Obecně jde vždy o precizně definovaný matematický stroj, jehož chování lze kvalitativně zkoumat (analyzovat, verifikovat) formálními metodami. Velmi významná (zvláště ve srovnání s řadou jiných specifikačních jazyků) je i přirozená možnost grafického vyjádření Petriho sítí a samozřejmě jejich proveditelnost (možnost simulovat dynamické chování modelu).

Existence různých variant Petriho sítí souvisí se snahou zvyšovat modelovací schopnosti a úroveň popisu modelu (jinak řečeno, přiblížit příslušný formalismus modelovaným skutečnostem) a přitom zachovat konceptuální jednoduchost, která je pro Petriho sítě příznačná. Obecně platí, že vyšší typy Petriho sítí jsou poněkud hůře analyzovatelné, ale poskytují vyšší komfort modelování. O vysokoúrovňových Petriho sítích už můžeme uvažovat jako o paralelním programovacím jazyku podobné úrovně jako například Prolog [CM81]. Přitom ovšem stále zachovávají možnost analýzy.

Praktické použitelnosti Petriho sítí v roli programovacího jazyka však brání výrazné omezení, kterým je statičnost struktury sítě a její plošnost. Jinak řečeno, Petriho sítě (ve své původní podobě) neposkytují strukturovací mechanismy, známé z jiných programovacích a specifikačních jazyků, jako jsou makra, procedury, funkce, objekty a moduly, umožňující skládat model ze submodelů, resp. program z podprogramů. To znamená, že pozitivní vlastnosti Petriho sítí se v plné míře projeví pouze u nepříliš rozsáhlých modelů, tedy většinou u modelů na vysokém stupni abstrakce. Potřebujeme-li modelovat více podrobností, model se nutně stává složitějším a méně přehledným a aplikace Petriho sítí je zde velmi problematická. Proto jsou Petriho sítě většinou chápány jen jako teoretický abstraktní model s velmi volnou vazbou na praktickou tvorbu rozsáhlých programových systémů. Tradiční obecný programovací jazyk, umožňující hierarchické modelování, zde vyhoví mnohem lépe, neposkytuje však možnost analýzy.

1.1.2 Programovací jazyky

K implemantaci rozsáhlých simulačních modelů se obvykle používají univerzální programovací jazyky. Ve vývoji programovacích jazyků je neustále patrná snaha pomocí vhodných jazykových konstrukcí usnadnit tvorbu rozsáhlých systémů. Významným mezníkem v tomto vývoji byl jazyk Simula 67 [Kin80], který poprvé zavedl pojmy *objekt*, *třída*, *dědičnost* a *instanciace*, čímž definoval *objektově orientovaný* přístup k tvorbě programů. O životaschopnosti objektově orientovaného přístupu svědčí fakt, že v poslední době došlo k masivnímu prosazení objektové orientace v programovacích jazycích, databázích i v metodách analýzy, návrhu a implementace programových systémů.

Vzhledem k poměrné komplikovanosti a absenci formálního základu a související teorie jsou však objektově orientované modely jen velmi obtížně formálně analyzovatelné. Objektově orientované jazyky sice formalizují objektovou orientaci, ale bohužel na příliš nízké (implementační) úrovni. Naproti tomu abstraktní objektově orientované modely, známé z metod objektově orientované analýzy a návrhu programových systémů, jsou neproveditelné (neumožňují simulaci). Je tedy zřejmě opodstatněná snaha vhodnou formalizaci objektově orientovaného přístupu provést na abstraktnější úrovni než je běžný programovací jazyk a přitom zachovat proveditelnost modelu [Mol95].

Kontroverzním aspektem objektově orientované technologie je souběžnost (concurrency). V kontextu objektově orientovaného programování se pojem souběžnost většinou omezuje jen na skutečnost, že objekty existují současně a nezávisle na sobě. Paralelní běh procesů není běžnými objektově orientovanými jazyky, které nejsou přímo určeny k simulačnímu modelování, podporován. Příkladem může být jazyk C++. Jazyk Simula 67 poskytuje kvaziparalelismus.[3]

[3] Kvaziparalelismus je však možné implementovat i v C++, příkladem může být [Per96].

Jinou formu paralelismu implementuje jazyk Smalltalk-80 [GR83], kde se o předávání řízení mezi procesy stará plánovač, ale jde zde, stejně jako v předchozích případech, o koncept, který je nezávislý na objektech.[4] Ve skutečnosti tedy v těchto jazycích koexistují buď jeden nebo více globálních toků řízení spolu s pasivními objekty, zapouzdřujícími funkce a data, nikoliv aktivitu. Tento dualismus objekt–proces komplikuje objektově orientované modelování nutností explicitně se zabývat implementací paralelismu v modelu.

Obecnější a z hlediska objektové orientace „čistší“ přístup předpokládá, že každý objekt může vyvíjet vlastní aktivitu, tj. každý objekt zapouzdřuje nejenom stav, ale i tok (svého vlastního) řízení. Hovoříme pak o aktivních objektech (agentech [Sho93]) a pasivní objekty považujeme za jejich speciálními případy. S každým objektem je zde spjat proces a procesy jednotlivých objektů běží paralelně, nezávisle na sobě, přičemž se mohou do jisté míry synchronizovat předáváním zpráv. Jde zde o skutečnou komunikaci aktivních objektů, nikoliv o předávání řízení mezi objekty. Modelování založené na aktivních objektech vede ke srozumitelnějším modelům, protože případná aktivita objektů je zde vyjádřena přirozeně, bez dodatečných konstrukcí pro vytváření procesů a jejich vzájemnou synchronizaci a komunikaci. Problematika paralelních objektově orientovaných jazyků, které implicitně považují všechny objekty za potenciálně aktivní, však dosud nepřekročila rámec výzkumných projektů [Nie93].

1.1.3 Programování Petriho sítěmi

Postavíme-li vedle sebe objektově orientované jazyky a Petriho sítě, lze říci, že vlastnosti obou přístupů (pozitiva a negativa v otázkách formálnosti, strukturovatelnosti a paralelismu) ze zdají být komplementární. Jejich vhodné spojení může potlačit nežádoucí vlastnosti a zachovat ty pozitivní.

Možným řešením, které si klade za cíl sblížit teorii s praxí na poli modelování a simulace paralelních systémů, je přizpůsobení Petriho sítí požadavkům, které bývají běžně kladeny na programovací jazyky, zavedením vhodného strukturování. V této oblasti bylo v minulosti učiněno několik více či méně úspěšných pokusů. Prvním významným pokusem o strukturování Petriho sítí bylo zavedení makro-uzlů (substitučích uzlů), které umožňují konstruovat rozsáhlejší síť za pomoci podsítí [HJS90]. Podobný přístup zavádějí modulární Petriho sítě [CP95]. Pro modelování systémů, jejichž struktura se za běhu nemění, toto řešení vyhovuje. Systémy, jejichž komponenty dynamicky vznikají a zanikají, se však modelují stejně obtížně, jako v případě klasických Petriho sítí. Tento problém přetrvával i v prvních pokusech kombinovat Petriho sítě s objekty: značky sice reprezentovaly dynamicky vznikající a zanikající objekty, ale síť představovala statickou globální strukturu (např. v [CT93]).

V diskusi kolem hierarchických barvených Petriho sítí (HCPN) [HJS90] byl prezentován i koncept invokace podsítí (dynamické vytváření instancí sítí prostřednictvím invokačních přechodů). Koncept dynamické instanciace sítí se (ať už přímo či nepřímo, v různých modifikacích) stal základem univerzálního zavedení objektové orientace do Petriho sítí, důsledně vycházejícímu z objektově orientovaných metodologií a programovacích jazyků, k jakému došlo v [LK94, SB94, Jan94a].[5] Posledně jmenovaná varianta se stala základem této práce.

1.2 Cíle a metody

Tato práce si klade za cíl ukázat, že diskrétní systémy lze modelovat v souladu s principy objektové orientace, aniž bychom se museli vzdát výhod, které přinášejí Petriho sítě. Aplikace

[4] Totéž lze říci i o jazyku Java [Fla96].

[5] Všechny uvedené přístupy k objektové orientaci v Petriho sítích vznikly takřka současně a zcela nezávisle na sobě. Jejich porovnání je uvedeno v kapitole 4.

Petriho sítí v objektově orientovaném modelování by měla na jedné straně kompenzovat absenci formálního aparátu a skutečného paralelismu v objektově orientovaných technologiích, a na druhé straně problematickou použitelnost Petriho sítí při modelování složitějších systémů. Celá práce tedy sleduje dva cíle, odrážející teoretickou a praktickou stránku věci:

1. Vytvoření teoretického rámce pro zkoumání objektů a objektově orientovaných modelů formálními metodami na bázi Petriho sítí.

2. Přímé použití Petriho sítí jako plnohodnotného programovacího jazyka, umožňujícího paralelní objektově orientované programování a simulační modelování.

Teoretického zázemí pro objektově orientované modelování paralelních systémů bude dosaženo formální definicí objektově orientované Petriho sítě (OOPN), přímo navazující na vysokoúrovňové a hierarchické Petriho sítě [Jen92, HJS90, ČJ93] a inspirovanou čistou objektovou orientací jazyka Smalltalk-80 [GR83], obohacenou o paralelismus, který respektuje princip zapouzdření. Přitom bude kladen důraz na konzistenci objektově orientovaného inskripčního jazyka (jazyka pro zápis výrazů ve vysokoúrovňových Petriho sítích) s objektově orientovaným strukturováním Petriho sítí.

Praktického cíle bude dosaženo návrhem jazyka a na něm postaveného programového systému, nazvaných shodně PNtalk. Jazyk PNtalk je konkrétní implementací OOPN, která pro popisy sítí používá jazyk Smalltalk. Systém PNtalk pak umožňuje vizuální objektově orientované simulační modelování a prototypování paralelních systémů, specifikovaných v jazyce PNtalk.

1.3 Struktura práce

Struktura práce sleduje myšlenkovou osu, která vede od vysokoúrovňových a hierarchických Petriho sítí k objektově orientované síti (OOPN) a posléze k jazyku a systému, který OOPN implementuje.

Úlohou první kapitoly (Úvod) bylo vysvětlit motivaci, která vedla k výzkumu objektové orientace v Petriho sítích, společně s cíli a metodami řešení. Kapitola 2 (Petriho sítě) uvádí do problematiky Petriho sítí a jejich strukturování. Kapitola 3 (Objektová orientace) uvádí do problematiky objektové orientace v souvislosti se zkoumáním diskrétních dynamických systémů. Kapitola 4 (Modelování objektů Petriho sítěmi) vysvětluje navrhovaný způsob zavedení objektové orientace do Petriho sítí a konfrontuje ho s jinými přístupy. V kapitole 5 (Objektově orientovaná Petriho síť) je formálně definována objektově orientovaná Petriho síť jako základ pro teoretické úvahy o systémech s aktivními objekty. Kapitola 6 (Jazyk a systém PNtalk) se zabývá praktickou stránkou OOPN – nástrojem pro modelování, simulaci a rychlé prototypování. Kapitola 7 (Aplikace a návaznosti) ukazuje některé aplikace, demonstrující programovací techniky pro OOPN, a diskutuje některé teoretické i praktické důsledky modelování prostřednictvím OOPN. V kapitole 8 (Závěr) jsou shrnuty dosažené výsledky spolu s možnostmi navazujícího výzkumu.

Kapitola 2

Petriho sítě

Tato kapitola uvádí do problematiky Petriho sítí. Petriho sítě budou představeny jako jasně definovaný matematický formalismus, vhodný pro modelování a teoretické zkoumání paralelních systémů. Budou též analyzovány metody strukturování Petriho sítí, vhodné pro modelování složitých reálných systémů.

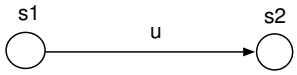
2.1 Modelování diskrétních systémů

Chování dynamického systému můžeme charakterizovat *stavovou proměnnou* a změnami její hodnoty v čase. Mapování času na stav (průběh hodnot stavu v čase) nazveme *procesem*. U diskrétních systémů pozorujeme jen skokové změny stavu, způsobené výskyty *událostí*.

Významnými příklady systémů, které obvykle zkoumáme jako diskrétní systémy, mohou být informační systémy, workflow systémy, pružné výrobní systémy a jejich řízení, komunikační protokoly, počítačové systémy, sociotechnické systémy atd.

Při zkoumání diskrétních systémů často vystačíme s časem abstrahovaným na sekvenci (nebo množinu sekvencí) výskytů událostí. Procesem pak rozumíme sekvenci hodnot stavu mezi výskyty událostí, popřípadě sekvenci výskytů událostí.

Systém, resp. jeho proces, specifikujeme obvykle implicitně, generativním způsobem, tj. konečným předpisem, na jehož základě lze příslušnou sekvenci výskytů událostí resp. hodnot stavu generovat. Použitelným modelem je zde stavový stroj (konečný automat), obecně nedeterministický, založený na myšlence, že stav systému je abstrakcí relevantní informace, nezbytné k popisu jeho možného budoucího stavu (vývoje) [Pet81]. Tento model je definován množinou stavů a množinou vzorů událostí (přechodů mezi stavy systému, obr. 2.1). Výhodou stavových strojů je možnost explicitně popsat souvislost událostí a stavů systému a skutečnost, že mají přesně definovanou operační sémantiku – jsou proveditelné. Díky tomu umožňují simulaci a mohou mít přímou návaznost na počítačovou implementaci modelu.



Obrázek 2.1: Přechod mezi stavy systému – vzor události.

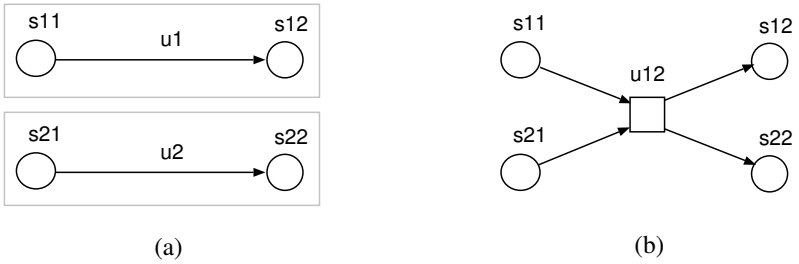
Netriviální systémy obvykle nelze snadno popsat jedním stavovým strojem. Raději je pak specifikujeme jako množiny komponent, přičemž s komponentou nakládáme jako se systémem. Lze potom rozlišovat kompozitní a elementární systémy. Každý elementární systém má svoji stavovou proměnnou a probíhá v něm proces (specifikovatelný dílčím stavovým strojem). Všechny

procesy v kompozitním systému probíhají nezávisle na sobě, paralelně. Pro zdůraznění této skutečnosti lze takový systém nazvat paralelním systémem. V paralelních systémech je významná otázka vzájemné synchronizace, komunikace a kooperace jednotlivých procesů. Vhodným modelem paralelního systému, který je schopen snadno a srozumitelně vyjádřit kauzalitu událostí, asynchronnost, paralelismus a synchronizaci, je Petriho síť [Pet62].

2.2 Petriho sítě

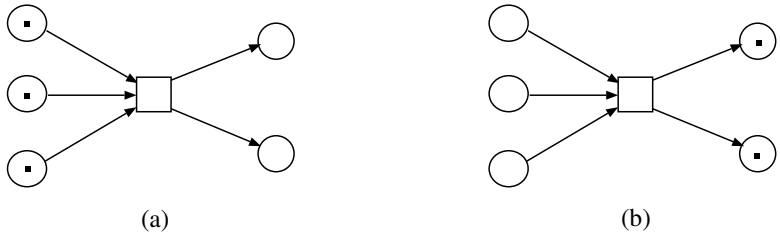
2.2.1 Základní koncepty

Stav systému, popsaného množinou stavových strojů, je určen množinou stavů jednotlivých strojů. Stav (stavová proměnná) systému je zde distribuován do množiny *parciálních stavů* systému. Provádění přechodů v jednotlivých strojích je však zapotřebí synchronizovat.



Obrázek 2.2: Synchronizace událostí ve dvou strojích – přechod v Petriho síti.

Petriho síť definuje přechod tak, aby mohl pracovat s více parciálními stavy současně, jak ukazuje obr. 2.2. Množina strojů je nahrazena jednou Petriho sítí, která modeluje systém jako celek.[1]



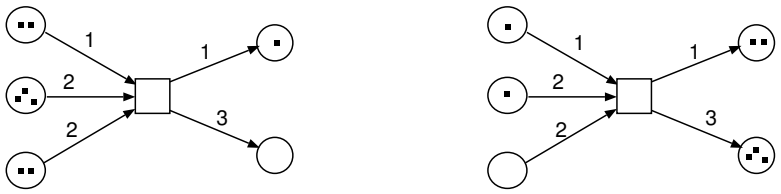
Obrázek 2.3: Provedení přechodu (výskyt události) v Petriho síti.

Parciální stavy systému jsou modelovány *místy* a vzory možných událostí jsou definovány *přechody*. Místo se v grafu Petriho sítě vyjadřuje jako ◯ a přechod jako □. Okamžitý stav systému je definován umístěním *značek* (tokens) v místech, což v grafu Petriho sítě vyjadřujeme tečkami v místech. Přítomnost značky v místě modeluje skutečnost, že daný aspekt stavu (parciální stav) je momentálně aktuální, resp. podmínka je splněna. Každý přechod má definována vstupní a výstupní místa, což je v grafu Petriho sítě vyjádřeno orientovanými hranami mezi místy a přechody: ◯→□ a □→◯. Tím je deklarováno, které aspekty stavu systému podmiňují výskyt odpovídající události (provedení přechodu), a které aspekty stavu jsou výskytem této

[1] Toto je poněkud kontroverzní aspekt Petriho sítí, který budeme analyzovat později.

události ovlivněny. Každý přechod má tedy definovány vstupní a výstupní podmínky. Přechod může být proveden v případě, že všechna jeho vstupní místa obsahují značky, tj. má splněny všechny vstupní podmínky. Provedením přechodu (viz obr. 2.3) se odstraní značky ze vstupních míst (vstupní podmínky přestanou platit) a umístí se nové značky do výstupních míst (uplatní se výstupní podmínky). Provedení přechodu je atomická operace, která odpovídá výskytu události.

Petriho sítě, s kterými budeme nadále pracovat, umožňují výskyt libovolného počtu značek v místech.[2] Vstupní a výstupní podmínky přechodů specifikují počty odebíraných/umisťovaných značek. V grafu Petriho sítě se to vyjádří ohodnocením orientovaných hran do/z přechodu (viz obr. 2.4). Přitom budeme dodržovat konvenci, že hrana, která není v grafu sítě explicitně ohodnocena, má implicitně váhu 1.



Obrázek 2.4: Provedení přechodu (výskyt události) v Petriho síti.

2.2.2 Petriho síť jako matematický stroj

Petriho síť je matematický stroj s přesně definovanou syntaxí a sémantikou. V literatuře se lze setkat s různými variantami definice Petriho sítě. Styl definice většinou odráží účel, ke kterému má být použita. Dále uvedeme definici, která odráží chápání Petriho sítí jako jazyka, který popisuje chování systémů pravidly, operujícími nad parciálními stavy, resp. jako automat, kde místa reprezentují jeho paměť a přechody reprezentují pravidla pro podmíněnou modifikaci obsahu paměti. Specifikace vstupních a výstupních podmínek jsou zde chápány jako součásti těchto pravidel.

Definice 2.2.1 Petriho síť je čtveřice $N = (P_N, T_N, PI_N, TI_N)$, kde

1. P_N je konečná množina **míst**
2. T_N je konečná množina **přechodů**, $P_N \cap T_N = \emptyset$
3. $PI_N : P_N \longrightarrow \mathbb{N}$ je **inicializační funkce** (place initialization function).
4. TI_N je **popis přechodů** (transition inscription function). Je to funkce, definovaná na T_N, taková, že $\forall t \in T_N$:

   $$TI_N(t) = (PRECOND_t^N, POSTCOND_t^N),$$

   kde

   (a) $PRECOND_t^N : P_N \longrightarrow \mathbb{N}$ jsou **vstupní podmínky (vstupy)** přechodu t.
   (b) $POSTCOND_t^N : P_N \longrightarrow \mathbb{N}$ jsou **výstupní podmínky (výstupy)** přechodu t.

[2] Jde o PT-sítě, které jsou zobecněním původního konceptu Petriho sítí.

Pro potřeby grafové reprezentace Petriho sítě definujeme množinu **hran**

$$A_N \subseteq (P_N \times T_N) \cup (T_N \times P_N),$$

tak, že

$$\forall (p,t) \in (P_N \times T_N) \; [\; (p,t) \in A_N \iff PRECOND_t^N(p) > 0 \;],$$

$$\forall (t,p) \in (T_N \times P_N) \; [\; (t,p) \in A_N \iff POSTCOND_t^N(p) > 0 \;].$$

Dále definujeme ohodnocení hran jako funkci $W_N : A_N \longrightarrow \mathbb{N}$, pro kterou platí:

$$\forall (p,t) \in A_N \cap (P_N \times T_N) \; [\; W_N(p,t) = PRECOND_t^N(p) \;],$$

$$\forall (t,p) \in A_N \cap (T_N \times P_N) \; [\; W_N(p,t) = POSTCOND_t^N(p) \;].$$

Je-li $(p,t) \in A_N \cap (P_N \times T_N)$, říkáme, že p je **vstupní místo** a (p,t) je **vstupní hrana** přechodu t. Je-li $(t,p) \in A_N \cap (T_N \times P_N)$, říkáme, že p je **výstupní místo** a (t,p) je **výstupní hrana** přechodu t.

Stav systému, popsaného Petriho sítí, je určen rozmístěním značek v místech sítě, jejím značením.

Definice 2.2.2 Značení sítě N je funkce $M : P_N \longrightarrow \mathbb{N}$. Funkce $M_0 = PI_N$ je počáteční značení sítě N.

Dynamika Petriho sítě spočívá v provádění přechodů. Vliv značení sítě na proveditelnost přechodů a vliv provedení přechodu na značení sítě jsou určeny tzv. evolučními pravidly, danými následující definicí.

Definice 2.2.3 Uvažujme síť N a značení M.

1. Přechod $t \in T_N$ je **proveditelný** ve značení M právě tehdy, když

   $$\forall p \in P_N \; [\; PRECOND_t^N(p) \leq M(p) \;].$$

2. Je-li přechod $t \in T_N$ proveditelný ve značení M, může být **proveden**, čímž se značení M změní na M', definované takto:

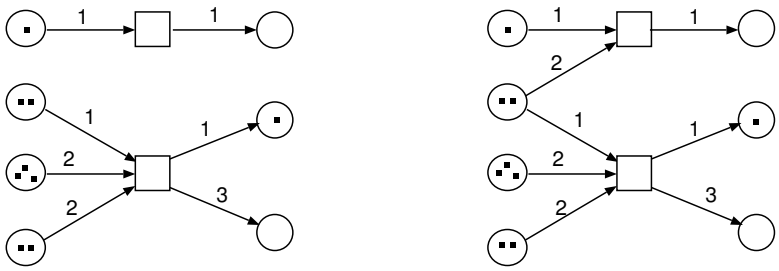
   $$\forall p \in P_N \; [\; M'(p) = M(p) - PRECOND_t^N(p) + POSTCOND_t^N(p) \;].$$

3. Může-li být přechod $t \in T_N$ ve značení M proveden, přičemž výsledným značením je M', říkáme, že M' je **přímo dostupné** z M **provedením** t, což zapisujeme

   $$M[t\rangle M'.$$

Definovali jsme Petriho síť jako matematický stroj, který prováděním přechodů postupně mění značení sítě. Pokud přechod rozpozná určitý vzor v reprezentaci stavu systému, který odpovídá jeho vstupním podmínkám, nahradí ho jiným, odpovídajícím jeho výstupním podmínkám. Přechod se projevuje pouze lokálně, zbytek reprezentace stavu systému zůstává nezměněn. Stav systému je tedy inkrementálně a lokálně modifikován.

Pokud přechody v daném značení nejsou konfliktní (viz dále), mohou být realizovány nezávisle na sobě, v libovolném pořadí (paralelně). Dva současně proveditelné přechody (v daném značení sítě) prohlásíme za konfliktní, když provedení jednoho z nich způsobí, že druhý přestane



Obrázek 2.5: Nezávislé (a) a konfliktní (b) přechody.

být proveditelný[3] (viz obr. 2.5). Konfliktní přechody modelují soupeření o zdroje a vzájemnou výlučnost přístupu ke zdrojům, nezávislé přechody modelují asynchronnost a paralelismus.

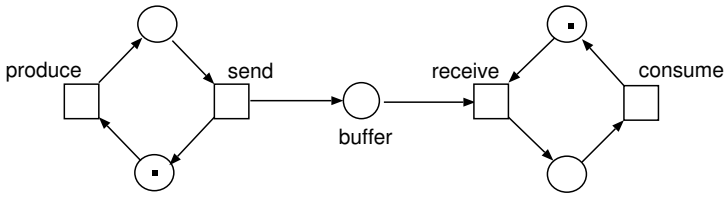
Obecně jde o nedeterministický stroj, neboť v daném značení může být proveditelných více přechodů současně a stroj se tedy může dostat potenciálně do několika dalších stavů. Množina všech možných posloupností změn stavu reprezentuje chování systému. Množina všech možných stavů, do kterých se stroj může dostat z počátečního stavu, je jeho stavový prostor.

Definice 2.2.4 Stavový prostor (množina dosažitelných značení) Petriho sítě N je nejmenší množina X značení sítě N, definovaná induktivně takto:

1. $PI_N \in X$.
2. Je-li $M \in X$ a $M[t\rangle M'$ pro nějaké $t \in T_N$, pak $M' \in X$.

2.2.3 Příklady

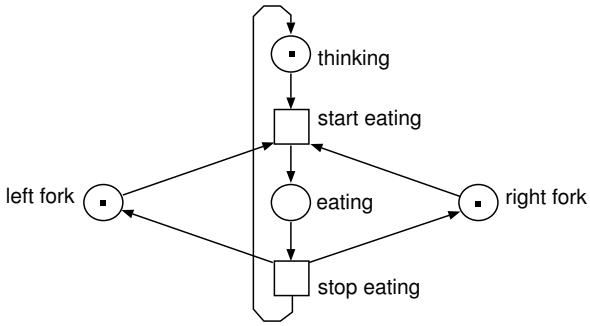
Obr. 2.6 ukazuje model systému producent-konzument. Jde o abstraktní systém, kde producent postupně posílá značky do místa `buffer`, odkud je konzument postupně přijímá. Místo `buffer` může v průběhu simulace obsahovat libovolný počet značek.



Obrázek 2.6: Producent a konzument.

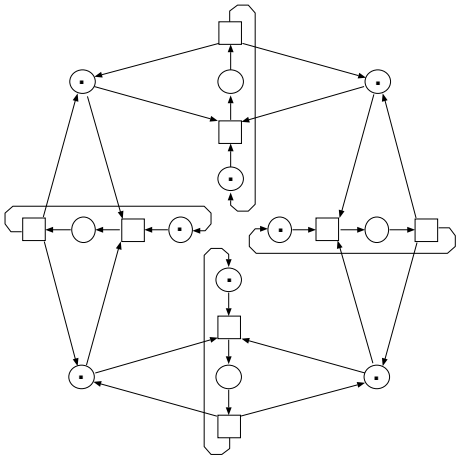
Druhý příklad je klasický model večeřících filosofů, demonstrující soupeření o sdílené zdroje. Systém je definován takto: Kolem kulatého stolu s jídlem a s n vidličkami sedí n filosofů, $n >= 2$. Každý filosof se může nacházet v jednom ze dvou stavů, „jezení“ a „přemýšlení“. K jezení filosof potřebuje dvě vidličky, které sdílí se svými sousedy.

[3]Přesněji (zajímá-li nás násobné provádění přechodů), sníží se stupeň jeho proveditelnosti, přičemž stupeň proveditelnosti přechodu říká, kolikrát (s jakou násobností) může být v daném značení proveden.



Obrázek 2.7: Večeřící filosof.

Model jednoho filosofa a jeho levé a pravé vidličky je na obr. 2.7. Model systému čtyř filosofů se čtyřmi vidličkami je na obr. 2.8.



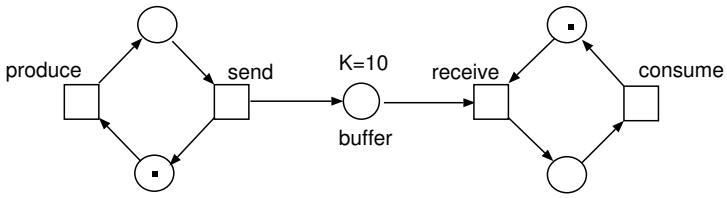
Obrázek 2.8: Čtyři večeřící filosofové, sdílející čtyři vidličky.

2.2.4 Varianty Petriho sítí

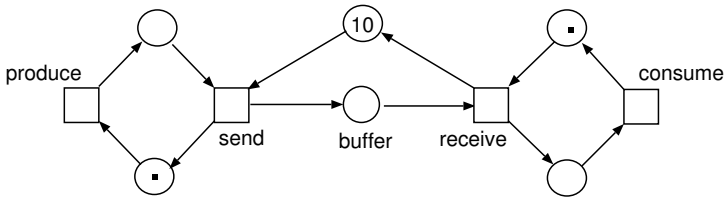
Síť, kterou jsme definovali, v literatuře nazývaná PT-síť [Pet81, Rei85], je zobecněním jednodušího modelu, a sice CE-sítě (Condition-Event Net). CE-síť, připouští maximálně jednu značku v místě (model večeřících filosofů na obr. 2.8 je CE-síť). Místa zde skutečně představují booleovské podmínky. Přechod CE-sítě je proveditelný, když všechny jeho vstupní podmínky jsou splněny a všechny jeho výstupní podmínky splněny nejsou. Pro tento druh Petriho sítí existuje bohatá teorie.

Síť se specifikovanými kapacitami míst. Petriho síť lze snadno doplnit o možnost specifikovat kapacitu míst (viz obr. 2.9), což se dobře uplatní při modelování reálných systémů.

Rozšíření Petriho sítě o možnost specifikace kapacity míst nepředstavuje pro teorii Petriho sítí problém, protože místa s kapacitami lze odstranit komplementací, tj. přidáním komplementárního místa, jak ukazuje obr. 2.10 (číslo 10 v místě znamená, že počáteční značení místa je 10 značek).



Obrázek 2.9: Producent a konzument s omezenou kapacitou bufferu.



Obrázek 2.10: Producent a konzument s omezeným bufferem, modelovaným komplementací.

Inhibitory. Závažnější rozšíření Petriho sítí představují inhibitory. Inhibitor je dodatečná speciální podmínka přechodu (dodatečná speciální hrana z místa do přechodu). Zjišťování proveditelnosti přechodu je oproti přechodu v síti bez inhibitorů doplněno o další podmínku, která vyžaduje, aby v každém místě, které je k přechodu připojeno inhibitorem, bylo méně značek, než je ohodnocení inhibitoru. Příklady inhibitorů jsou na obr. 2.11 (oba přechody jsou proveditelné).

Obrázek 2.11: Příklady inhibitorů (inhibitory jsou vyjádřeny hranami, zakončenými kolečkem).

Inhibitory umožňují testovat počet značek v místě a tím dávají Petriho sítím výpočetní sílu Turingova stroje a jsou tedy schopny počítat všechny vyčíslitelné funkce [Pet81, DA92]. Takovýmito sítěmi je možné specifikovat libovolný algorimus. Možnosti analýzy těchto sítí jsou však poněkud chudší. Jinou variantou Petriho sítě, která má výpočetní sílu Turingova stroje, je Petriho síť s prioritami přechodů [DA92].

Speciální podtřídy Petriho sítí. Je též třeba poznamenat, že kromě snahy zvyšovat modelovací sílu Petriho sítí a komfort modelování je zde také snaha omezit tyto vlastnosti s cílem získat formální aparát s vyšší rozhodovací silou, tedy s lepší možností model analyticky zpracovat. Kromě CE-sítí takto vznikly například značené grafy a free-choice nets [Pet81, Rei85].

Petriho sítě s časem. Pro potřeby simulace, řízení, real-time a stochastické analýzy je třeba zavést do Petriho sítí vhodné časování. Časová omezení je možné spojit jak s místy, tak s přechody, případně i s hranami Petriho sítí. Přiřazení časové informace místu znamená, že značka, umístěná do toho místa může být odstraněna teprve po uplynutí specifikovaného času. Přiřazení časové informace přechodu znamená, že přechod se může provést, je-li proveditelný po specifikovanou dobu. Existují různé varianty těchto principů [DA92, MBC$^+$95, Bow96, Wie96, Ess97b]. Dále se budeme zabývat jen sítěmi bez časových omezení, neboť časování bezprostředně nesouvisí s předmětem této práce a je ho možné do různých variant Petriho sítí dodatečně zavést.

2.3 Vysokoúrovňové Petriho sítě

Přestože PT sítě s inhibitory jsou z teoretického hlediska schopny modelovat vše, co je možné vyjádřit algoritmem, v situaci, kdy požadujeme podrobnější model a nelze se spokojit s hrubou abstrakcí, představují příliš nízkoúrovňový model (jazyk), který i jednoduché skutečnosti, jako je například vyhodnocení aritmetického výrazu, modeluje příliš složitě, podobně, jako bychom konstrukce programovacích jazyků programovali přímo instrukcemi procesoru. Z tohoto důvodu tyto sítě nepředstavují adekvátní prostředek pro detailní modelování reálných systémů.

Jedním z pokusů tento problém překonat bylo zavedení globálních proměnných, které mohou být testovány a modifikovány v rámci provádění přechodů. Petriho síť zde vyjadřuje řídicí strukturu modelu. Pro testování a manipulaci s globálními proměnnými jsou pak použity libovolné predikáty a funkce, definované mimo rámec Petriho sítí [DA92].[4]

Dále existují specializované varianty Petriho sítí, jejichž uzly mají pozměněnou sémantiku, aby přímočařeji modelovaly systémy z určité problémové domény (například pružné výrobní systémy). Přestože tato rozšíření Petriho sítí umožňují lépe vyjádřit jisté specifické skutečnosti, obvykle komplikují aplikaci metod analýzy, založených na původních Petriho sích, případně jsou analytické metody, vyvinuté pro jeden druh sítí, těžko transformovatelné do prostředí jiných sítí, které používají jiná aplikačně orientovaná rozšíření [Jen92].

Prvním pokusem obecně (tj. bez návaznosti na konkrétní aplikační oblast) vyřešit problematickou použitelnost Petriho sítí při detailním modelování reálných systémů byly predikátové Petriho sítě, PrT sítě (Predicate-Transition nets) [Gen87], ve kterých značky reprezentují konkrétní data, která mohou být přechody testována a modifikována (rozdíl oproti PT síti s globálními proměnnými je zřejmý – v síti nejsou žádné cizorodé prvky, vymykající se analytickému zpracování). Tím byly poprvé definovány vysokoúrovňové Petriho sítě, HLPN (High-Level Petri nets), které celý model popisují pouze prostředky těchto sítí, bez nutnosti používat inhibitory, priority a globální proměnné (všechna zmíněná vylepšení lze tranformovat do HLPN bez nich). Významným přínosem HLPN bylo kromě možnosti modelovat obecné výpočty a složité manipulace s daty a datovými strukturami také zachování možnosti algebraické analýzy (jde o výpočet, resp. ověření invariantů [Gen87, Jen92]). Je ovšem pravda, že tato analýza je mnohem obtížnější, než u CE a PT sítí, což je u formalismu této úrovně pochopitelné. Jiné varianty HLPN, jako je Barvená Petriho síť, CPN (Coloured Petri net) [Jen92],[5] byly vyvinuty pro usnadnění interpretace výsledků analýzy pomocí invariantů. Z hlediska modelování mezi PrT sítěmi a CPN nejsou žádné významné rozdíly, odlišnosti se však mohou projevit ve způsobu analýzy těchto sítí. Pro potřeby modelování vznikají i jiné varianty HLPN. Jensen [Jen92] je však označuje jen za různé dialekty CPN, přičemž teorii, která vzniká kolem CPN, lze pro tyto varianty HLPN relativně snadno přizpůsobit.

[4] Toto rozšíření Petriho sítí bývá součástí *interpretovaných* Petriho sítí, v nichž lze provádění přechodů synchronizovat vnějšími událostmi a provádění přechodů může provádět akce vně Petriho sítě (v okolí modelu). Tyto sítě mají zvláštní význam pro řízení.

[5] Barvami se v HLPN myslí hodnoty datových typů, svázané se značkami.

V našem případě budeme HLPN (tuto variantu HLPN nazveme HL-síť) definovat tak, aby stylově navazovala na definici 2.2.1 a umožnila modifikace, kterými se budeme zabývat v dalších částech této práce.

2.3.1 Multimnožiny

K tomu, abychom mohli vysvětlit základní koncepty HL-sítí, potřebujeme pomocný pojem *multimnožina* a operace s mutimnožinami.

Definice 2.3.1 Mějme libovolnou neprázdnou množinu E. **Multimnožina** nad množinou E je funkce $x : E \longrightarrow \mathbb{N}$. Hodnota $x(e)$ je **počet výskytů** (koeficient) prvku e v multimnožině x. Multimnožinu zapisujeme jako formální sumu[6]

$$\sum_{e \in E} x(e)`e.$$

Množinu všech multimnožin nad E označíme E^{MS}. Pro multimnožiny x, y nad E a přirozené číslo n definujeme:

1. sčítání: $x + y = \sum_{e \in E} (x(e) + y(e))`e$
2. skalární násobení: $n`x = \sum_{e \in E} (n\ x(e))`e$
3. porovnání:

$$x \neq y = \exists e \in E\ [\ x(e) \neq y(e)\ ]$$
$$x \leq y = \forall e \in E\ [\ x(e) \leq y(e)\ ]$$

4. odčítání (pro $x \geq y$): $x - y = \sum_{e \in E} (x(e) - y(e))`e$
5. velikost: $|x| = \sum_{e \in E} x(e)$.

2.3.2 Základní koncepty

Výrazy na hranách přechodu na obr. 2.12 specifikují značky, které přechod chce odebrat ze vstupních míst a které chce umístit do výstupních míst. Jelikož vstupní místa obsahují potřebné značky, přechod lze provést a jeho provedením se značení sítě změní podle obr. 2.12(b). Poznamenejme, že v multimnožinových výrazech na hranách vynecháváme koeficient 1.

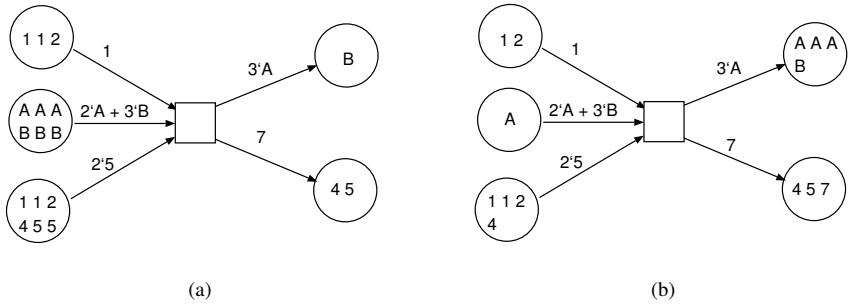
Přechod na obr. 2.13 obsahuje proměnné v hranových výrazech.[7] Je proveditelný pro dvě různá navázání proměnných $\{x = 1, y = A\}$ a $\{x = 1, y = B\}$. Jeho provedení je realizováno pro $\{x = 1, y = A\}$.

Hranové výrazy mohou obecně kromě konstant a proměnných obsahovat libovolné funkce (lambda-výrazy). Tuto skutečnost si ukážeme na konkrétním příkladě. Síť na obr. 2.14 modeluje čtyři večeřící filosofy. Každý filosof je modelován přirozeným číslem, každá vidlička také. Filosof x používá vidličky x a $(x + 1)$ mod 4, což je vyjádřeno hranovými výrazy v síti na obr. 2.14.

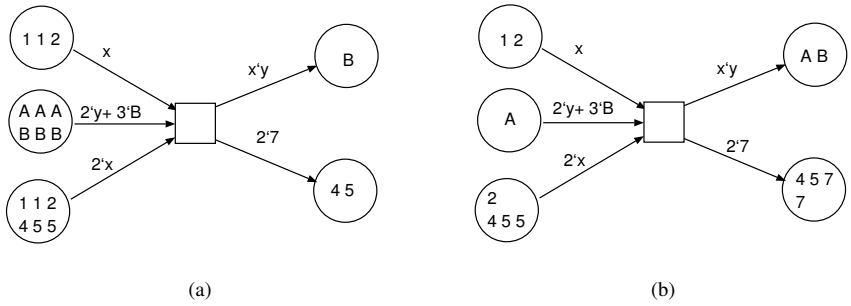
Přechody v HL-síti mohou obsahovat *stráž*, což je booleovský výraz (predikát), který vyjadřuje dodatečnou podmínku pro navázání proměnných, pro které je přechod proveditelný. Obr. 2.15 ukazuje HL-síť, modelující večeřící filosofy, která má jednodušší výrazy na hranách

[6] Například zápis $3`a + 4`b$ představuje multimnožinu se třemi výskyty prvku a a čtyřmi výskyty prvku b. Koeficient 1 obvykle vynecháváme, tj. například zápis c představuje tutéž multimnožinu jako zápis $1`c$.

[7] Proměnné budeme značit malými písmeny, případně identifikátory, začínajícími malým písmenem.



Obrázek 2.12: Provedení přechodu v HL-síti s konstantami na hranách. (a) – stav před provedením, (b) – stav po provedení přechodu.



Obrázek 2.13: Provedení přechodu v HL-síti s proměnnými v hranových výrazech pro navázání proměnných $\{x = 1, y = A\}$. (a) – stav před provedením, (b) – stav po provedení přechodu.

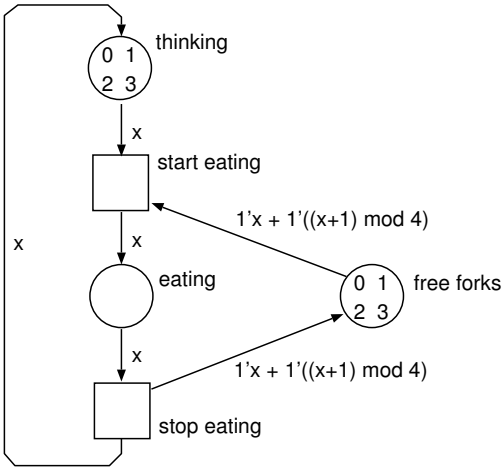
díky přesunutí výpočtů do stráží přechodů. Stráž vždy zapisujeme podtrženou. V zápisu multimnožiny dovolíme nadále místo symbolu + používat čárku, tj. hranový výraz „`x, y`" označuje multimnožinu, obsahující jeden výskyt `x` a jeden výskyt `y`. Inskripční jazyk, kterým zapisujeme hranové výrazy, počáteční značení míst a stráž přechodu, budeme specifikovat v rámci definice HL-sítě.

2.3.3 Definice HL-sítě

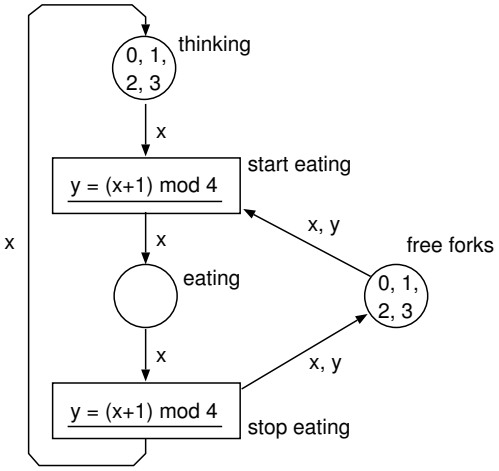
HL-síť lze definovat tak, že každé hraně HL-sítě je přiřazena multimnožina obecných výrazů a každému přechodu je přiřazen strážní výraz. Provedení přechodu je možné pro určité navázání proměnných ve výrazech na jeho vstupních hranách a ve stráži přechodu. Problémem je nalezení vhodného navázání proměnných. Testování všech možných navázání je v netriviálních případech prakticky nemožné a proto se z praktických důvodů na vstupních hranách přechodů používají jednodušší výrazy, které lze snadno unifikovat s obsahem míst a zajistit tak snadné nalezení vhodného navázání proměnných. V našem případě použijeme multimnožiny n-tic termů (termem rozumíme konstantu nebo proměnnou). Složitější vyhodnocení proveditelnosti přechodu je pak přesunuto do stráže přechodu, kde už je dovoleno použít obecný výraz (předpoládá se booleovský výsledek).[8]

Z důvodu lepší přizpůsobivosti dalším modifikacím stejně zjednodušíme i výrazy na výstupních hranách a veškeré složitější výpočty koncentrujeme do akce přechodu, kde je opět dovoleno

[8]Předpokládáme však možnost odvodit vhodné navázání proměnných z hodnot značek ve vstupních místech přechodu.



Obrázek 2.14: Čtyři večeřící filosofové, sdílející čtyři vidličky.



Obrázek 2.15: Čtyři večeřící filosofové – alternativní model.

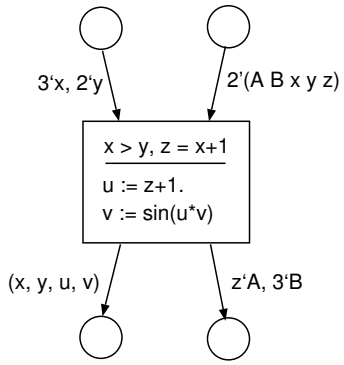
použít obecný výraz (zde se objeví přiřazení výsledku výrazu do proměnné). Příklad přechodu se stráží i akcí je na obr. 2.16.

Poznamenejme, že akce přechodu je ve skutečnosti zbytečná, protože veškeré výpočty lze realizovat ve stráži přechodu.[9] Akci zde definujeme z toho důvodu, že nám v některých případech umožní sugestivnější zápis, odlišující výpočty, které se realizují při provádění přechodu, od těch, které se realizují při zjišťování proveditelnosti přechodu. Také může usnadnit implementaci simulátoru HL-sítě. Později nám tento tvar přechodu také umožní modifikaci sémantiky přechodu, spočívající v možnosti jeho neatomického provedení.

Dříve, než uvedeme definici HL-sítě, je třeba definovat inskripční jazyk a jeho sémantiku. Začneme pomocným pojmem *n-tice*. Poté následuje definice *systému barev*, nad kterým bude inskripční jazyk definován.

Definice 2.3.2 Nechť E je libovolná množina. Prvek množiny $E^* = \bigcup_{n \in N} E^n$ je **n-tice** a zapisuje se $x = (e_1, ..., e_n)$. () je jedinečný prvek E^0. Jakékoli uspořádání $\leq$ na E definuje relaci

[9] Stráž přechodu je predikát, umožňující v rámci zjišťování proveditelnosti přechodu navázat volné proměnné.



Obrázek 2.16: Přechod se stráží a akcí.

částečného uspořádání $\leq$ na E^* tak, že $(x_1, ..., x_m) \leq (y_1, ..., y_n)$ právě tehdy, když $m = n$ a $x_i \leq y_i$ pro všechna $i \in \{1, ..., n\}$.

Definice 2.3.3 Systém barev C je struktura (U, O, I, V), kde

1. U je **universum**, definované takto:
   (a) $\{0, 1, ...\} \subset U$, $\{true, false\} \subset U$.
   (b) $n \in \mathbb{N} \wedge x_1 \in U \wedge ... \wedge x_n \in U \Longrightarrow (x_1, ..., x_n) \in U$.
2. O je konečná množina **operátorů** s aritou, $O \cap U = \emptyset$.
   Pro $o^{(m)} \in O$ je m arita operátoru o, $m \geq 1$.
3. I je interpretace operátorů. Je to funkce, definovaná na O tak, že:
   $I(o^{(m)}) = f$, kde f je funkce tvaru $f : U^m \longrightarrow U$.
4. V je konečná množina **proměnných**. $V \cap U = \emptyset, V \cap O = \emptyset$.
   Jakoukoliv funkci $b : V' \longrightarrow U$, $V' \subseteq V$, nazveme **navázání** množiny proměnných V'.

Definice 2.3.4 Nad systémem barev $C = (U, O, I, V)$ definujeme tyto druhy výrazů:

1. **Term** je prvek množiny $TERM(C)$, $TERM(C) = U \cup V$.
2. **Obecný výraz** je prvek množiny $EXPR(C)$, definované takto:
   - $TERM(C) \subseteq EXPR(C)$.
   - Každý výraz tvaru $o^{(m)}(e_1, e_2, ..., e_m)$, kde $e_i \in EXPR(C)$ a $o^{(m)} \in O$, je prvkem množiny $EXPR(C)$.
3. Definujeme pomocnou množinu $LISTEXPR(C)$ takto:
   - $TERM(C) \subseteq LISTEXPR(C)$.
   - Každý výraz tvaru $(e_1, e_2, ..., e_m)$, kde $e_i \in LISTEXPR(C)$, $m \in \mathbb{N}$, je prvkem množiny $LISTEXPR(C)$.
4. **Hranový výraz** je prvek množiny $ARCEXPR(C)$, definované takto:
   - Každý výraz tvaru e_1‘e_2, kde $e_1 \in TERM(C)$, $e_2 \in LISTEXPR(C)$, je prvkem množiny $ARCEXPR(C)$.

- Každý výraz tvaru $e_1 + e_2$, kde $e_1, e_2 \in ARCEXPR(C)$, je prvkem $ARCEXPR(C)$.[10]

Nechť $Var(e) \subseteq V$ je množina všech proměnných ve výrazu e.

Definice 2.3.5 Mějme systém barev $C = (U, O, I, V)$. Definujeme $e\langle b \rangle$ jako výsledek vyhodnocení výrazu e při libovolném navázaní $b : V' \longrightarrow U$, $Var(e) \subseteq V' \subseteq V$, takto:

1. (a) Je-li $e \in U$, pak $e\langle b \rangle = e$.
   (b) Je-li $e \in V$, pak $e\langle b \rangle = b(e)$.
2. (a) Je-li e výraz tvaru $o^{(m)}(e_1, e_2, ..., e_m)$, kde $e_i \in EXPR(C)$ a $o^{(m)} \in O$, pak $e\langle b \rangle = (I(o^{(m)}))(e_1\langle b \rangle, e_2\langle b \rangle, ..., e_m\langle b \rangle)$.
3. (a) Je-li $e = (e_1, ..., e_n)$, $e_i \in LISTEXPR(C)$, pak $e\langle b \rangle = (e_1\langle b \rangle, ..., e_n\langle b \rangle)$.
4. (a) Je-li e výraz tvaru e_1‘e_2, kde $e_1 \in TERM(C)$, $e_2 \in LISTEXPR(C)$, pak $e\langle b \rangle = e_1\langle b \rangle$‘$e_2\langle b \rangle$.
   (b) Je-li e výraz tvaru $e_1 + e_2$, kde $e_1, e_2 \in ARCEXPR(C)$, pak $e\langle b \rangle = e_1\langle b \rangle + e_2\langle b \rangle$.

Poznamenejme, že když se term v koeficientu zápisu multimnožiny nevyhodnotí jako přirozené číslo, výraz nelze vyhodnotit. Tato situace se nesmí v HL-síti vyskytnout.

Nyní, když jsme definovali inskripční jazyk i jeho sémantiku, jsme již schopni definovat HL-síť.

Definice 2.3.6 HL-síť je pětice $N = (C, P_N, T_N, PI_N, TI_N)$, kde

1. C je systém barev.
2. P_N je konečná množina **míst**
3. T_N je konečná množina **přechodů**, $P_N \cap T_N = \emptyset$
4. $PI_N : P_N \longrightarrow ARCEXPR(C)$ je **inicializační funkce míst**, taková, že $\forall p \in P_N \; [\; Var(PI_N(p)) = \emptyset \;]$.
5. TI_N je **popis přechodů**. Je to funkce, definovaná na T_N, taková, že $\forall t \in T_N$:

   $$TI_N(t) = (PRECOND_t^N, GUARD_t^N, ACTION_t^N, POSTCOND_t^N),$$

   kde

   (a) $PRECOND_t^N : P_N \longrightarrow ARCEXPR(C)$ jsou **vstupní podmínky** přechodu t.
   (b) $GUARD_t^N \in EXPR(C)$ je **stráž přechodu**.[11]
   Definujeme množinu $InVar_N(t) = Var(GUARD_t^N) \cup \bigcup_{p \in P_N} Var(PRECOND_t^N(p))$.
   (c) $ACTION_t^N$ je **akce přechodu** – konečná (může být i prázdná) sekvence výrazů tvaru

[10] Poznamenejme, že místo symbolu + v hranových výrazech můžeme použít čárku, jak již bylo dříve uvedeno.

[11] Stráž je funkce, vracející `true` nebo `false`, v závislosti na hodnotách argumentů (argumenty mohou být také vyhodnoceny funkcemi). Předpokládá se, že lze bez problémů najít všechna navázání proměnných, pro které výraz ve stráži vrací `true`, jinak řečeno, lze tato navázání najít lepším způsobem, než je zkoušení všech možností navázání proměnných (připomeňme, že univerzum je nekonečné).

$y_i := expr_i$

pro $i \in \{1, ..., n\}, n \in \mathbb{N}$, kde $y_i \in V - InVar_N(t) - \{y_j \mid j < i\}$, $expr_i \in EXPR(C)$, $Var(expr_i) \subseteq InVar_N(t) \cup \{y_j \mid j < i\}$.

Definujeme množinu $Var_N(t) = InVar_N(t) \cup \{y_1, ..., y_n\}$.

(d) $POSTCOND_t^N : P_N \longrightarrow ARCEXPR(C)$, $Var(POSTCOND_t^N) \subseteq Var_N(t)$, jsou **výstupní podmínky** přechodu t.

Poznamenejme, že v grafové reprezentaci HL-sítě se objeví hrana mezi p a t jen v případě, že pro hranový výraz $e = PRECOND_t^N(p)$ resp. $e = POSTCOND_t^N$ neplatí $Var(e) = \emptyset \wedge e\langle\emptyset\rangle = \emptyset$, tj. hranami jsou reprezentovány pouze neprázdné vstupní a výstupní podmínky přechodů.

Stav systému, popsaného HL-sítí, je určen, analogicky k „jednobarevné“ síti, značením HL-sítě.

Definice 2.3.7 Značení HL-sítě N je funkce $M : P_N \longrightarrow U^{MS}$.

Evoluční pravidla HL-sítě jsou opět definována analogicky k „jednobarevné“ síti, je zde však navíc vyhodnocení stráže a akce přechodu. Definici HL-sítě uzavírá pojem *stavový prostor*.

Definice 2.3.8 Uvažujme značení M HL-sítě N.

1. Přechod $t \in T_N$ je **proveditelný** ve značení M pro navázání $b : InVar_N(t) \longrightarrow U$ právě tehdy, když

   $$\forall p \in P_N \; [\; PRECOND_t^N(p)\langle b\rangle \leq M(p) \wedge GUARD_t^N\langle b\rangle = true \;].$$

2. Je-li přechod $t \in T_N$ proveditelný ve značení M pro navázání b, může být **proveden**, čímž se značení M změní na M', definované takto:

   $$\forall p \in P_N \; [\; M'(p) = M(p) - PRECOND_t^N(p)\langle b\rangle + POSTCOND_t^N(p)\langle b'\rangle \;],$$

   kde b' je definováno takto:

   Nechť $ACTION_t^N$ je sekvence výrazů $y_i := expr_i$, $i \in \{1, ...n\}$, $n \in \mathbb{N}$, $r_i = expr_i\langle b_i\rangle$ a $b_i = b \cup \{(y_j, r_j) \mid j < i\}$. Definujeme $b' = b_n$ pro $n > 0$, jinak $b' = b$.

3. Může-li být přechod $t \in T_N$ ve značení M pro navázání b proveden, přičemž výsledným značením je M', říkáme, že značení M' je **přímo dostupné** z M **provedením** t pro navázání b, což zapisujeme

   $$M[t, b\rangle M'.$$

Definice 2.3.9 Stavový prostor HL-sítě N je nejmenší množina X značení sítě N, induktivně definovaná takto:

1. $PI_N \in X$.
2. Je-li $M \in X$ a $M[t, b\rangle M'$ pro nějaké $t \in T_N$ a $b : InVar_N(t) \longrightarrow U$, pak $M' \in X$.

Na závěr tohoto odstavce poznamenejme, že možná rozšíření, jako jsou kapacity míst, inhibitory, časování atd., lze do HL-sítě zavést podobným způsobem, jako do „jednobarevných“ sítí. Pro další výklad to však není významné.

Z hlediska modelovací síly HL-sítí je významné, že umožňují neomezené Turingovské výpočty v rámci zjišťování proveditelnosti a provádění přechodů [Gen87, Jen92]. Jde o stroj s pamětí, reprezentovanou místy, počátečním obsahem paměti a pravidly, reprezentovanými přechody, která definují vývoj obsahu paměti, reprezentovaného značením sítě. Pravidla mohou být nedeterministicky, asynchronně a v případě bezkonfliktnosti i paralelně aplikována na stav systému.

HL-sítě tedy představují univerzální jazyk, založený na pravidlech, vyznačující se kromě jednoduchosti také skutečností, že nevyžaduje žádné komplikované synchronizační prostředky. Synchronizace je přirozeně popsána prostředky Petriho sítí.

2.4 Modelování Petriho sítěmi

Mapování prvků modelovaného systému do míst, přechodů a značek je výsledkem abstrakce. Toto mapování může být různé, závisí na účelu modelu. Přitom je třeba respektovat skutečnost, že zatímco místa a přechody, spolu s jejich vzájemnými vazbami, existují staticky, značky mohou dynamicky vznikat, zanikat a migrovat působením přechodů.

- Místo reprezentuje pasivní objekt (prvek systému), který je schopen obsahovat (pamatovat si) jiné objekty. Typicky modeluje sklad, komunikační kanál, databázi, paměť apod.
- Přechod reprezentuje aktivní prvek systému (nebo část aktivního prvku, s kterou jsou svázány výskyty událostí), jehož aktivita může být podmíněna složkami stavu systému a která může stav systému změnit. Typicky modeluje procesor (nebo jeho část, nebo vzor události, který je ním spjatý), funkční blok apod.
- Značka reprezentuje mobilní pasivní objekt, který je možno uložit do místa. Typicky modeluje data, příznak stavu subsystému apod.

Výhody Petriho sítí. Konceptuální jednoduchost Petriho sítí a jejich přesný matematický základ umožňují aplikovat formální metody při řešení problémů, spojených s validací modelu, resp. s verifikací kroků při zjemňování modelu [Rei85, Pet81, Jen92]. Sugestivní grafová reprezentace Petriho sítí také významnou měrou přispívá k popularitě tohoto formalismu, protože usnadňuje chápání systému jako celku na základě podmínek a událostí.

Nevýhody Petriho sítí. Petriho síť byla koncipována jako plošný (nestrukturovaný) model, umožňující sugestivně vyjádřit souvislost dílčích aspektů celkového stavu systému a událostí. Petriho síť modeluje systém jako celek, jehož stav je strukturován do parciálních stavů a v němž dochází k událostem. Hierarchický aspekt modelovaného systému zde není nijak vyjádřitelný, může se však projevit během modelování, při postupné tvorbě Petriho sítě, kterou budujeme skládáním dílčích částí výsledné sítě, popřípadě zjemňováním jistých částí sítě. Výsledná síť však zůstává plošným modelem.

2.5 Statické strukturování Petriho sítí

Plošnost modelů, založených na Petriho sítích představuje hlavní bariéru, bránící jejich použití pro podrobné modelování rozsáhlých systémů. Je tedy třeba nějakým způsobem se s tímto

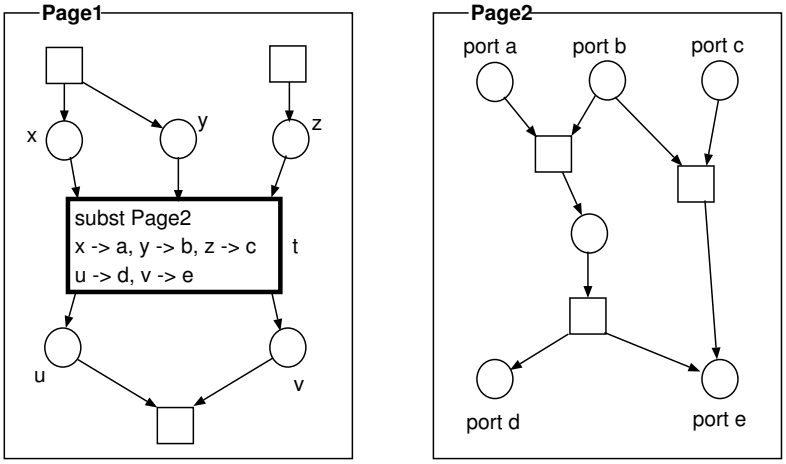
problémam vyrovnat. Zvládání rozsáhlých systémů je obecně možné díky dekompozici na subsystémy, které se modelují zvlášť, a potom se z těchto dílčích modelů komponuje celkový model. V našem případě tedy systém rozložíme vhodným způsobem na subsystémy, a tyto subsystémy, pokud již nemá smysl je dále rozkládat, popíšeme Petriho sítěmi. Přitom je ovšem třeba brát v úvahu otázku komunikace každého subsystému s okolím, tedy s ostatními subsystémy. Poznamenejme, že tyto úvahy jsou do značné míry nezávislé na tom, zda jde o sítě nízké, či vysoké úrovně.

Co se týče komunikace dílčích Petriho sítí, resp. procesů, řízených těmito sítěmi, přichází obecně v úvahu sdílení míst a sdílení přechodů. Modelovaný systém je pak určen množinou dílčích sítí a specifikací jejich propojení (např. nějakou formou grafické notace, jako např. v [Obe87]).

2.5.1 Hierarchická Petriho síť

Jsou-li dílčí sítě specifikovány odděleně a pojmenovány, je tím umožněna násobná použitelnost sítí statickým instanciováním. To je výhodné u systémů, které obsahují více strukturně shodných subsystémů, jejichž počet a propojení se za života systému nemění.

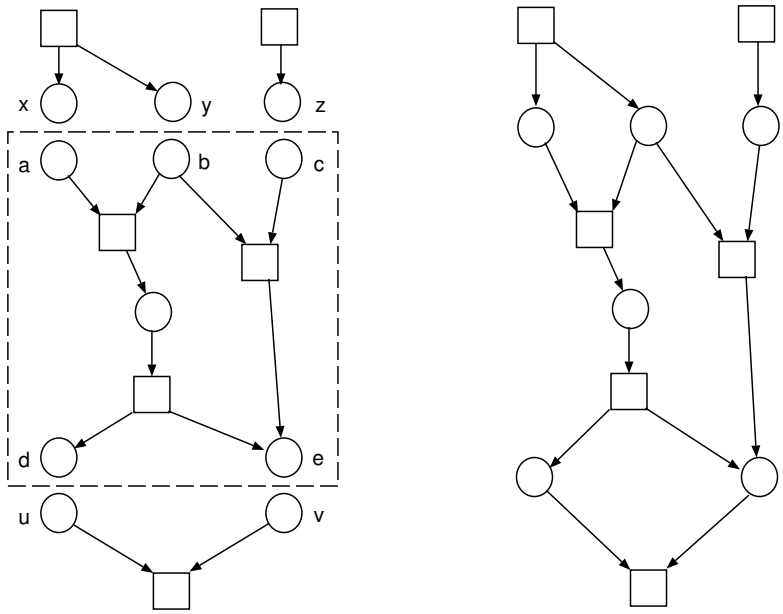
Tento přístup používají hierarchické Petriho sítě [HJS90]. Systém je zde popsán množinou pojmenovaných *stránek*. Každá stránka obsahuje Petriho síť a definuje *porty*, což jsou uzly sítě, určené k propojování s jinými sítěmi. Instance stránky může být specifikována v libovolné stránce formou *substitučního uzlu* (místa nebo přechodu). Pro instanciaci substitučním přechodem musí stránka definovat místa jako porty a naopak, pro instanciaci substitučním místem musí stránka definovat přechody jako porty. Propojení instance stránky s hierarchicky nadřazenou instancí stránky je realizováno sloučením portů s uzly, které bezprostředně sousedí se substitučním uzlem (tyto uzly se podle [HJS90] nazývají *sokety*).



Obrázek 2.17: Substituční přechod.

Substituční přechod. Na obr. 2.17 jsou dvě stránky. Ve stránce `Page1` je definován substituční přechod `t`, reprezentující instanci stránky `Page2` a definující propojení soketů a portů. V definici propojení se explicitně uvádí jen propojení různě pojmenovaných soketů a připojených portů.

Sémantika *substitučního přechodu* může být definována konstrukcí ekvivalentní nehierarchické sítě, což lze provést ve třech krocích (podle [HJS90]):



Obrázek 2.18: Sémantika substitučního přechodu (podle [HJS90]).

1. Odstranit substituční přechod (spolu s okolními hranami).
2. Vložit kopii podsítě (tuto kopii budeme nazývat substituční instancí podsítě).
3. Sloučit každý soket s přiřazeným portem.[12]

Nehierarchická síť, odpovídající hierarchické síti z obr. 2.17 je na obr. 2.18. Jak je vidět, substituční přechod není proveditelnou součástí modelu, protože je pro dobu běhu nahrazen příslušnou instancí podsítě.

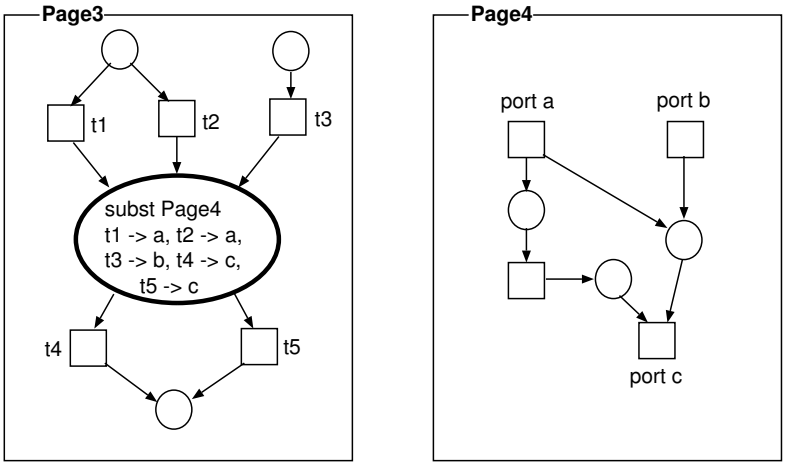
Substituční místo. Na obr. 2.19 jsou opět dvě stránky. Ve stránce `Page3` je definováno substituční místo `p`, reprezentující instanci stránky `Page4` a definující propojení soketů a portů.

Sémantika *substitučního místa* může být opět definována konstrukcí ekvivalentní nehierarchické sítě, což lze provést ve třech krocích:

1. Odstranit substituční místo (spolu s okolními hranami).
2. Vložit kopii podsítě a případně duplikovat porty tak, aby každému soketu příslušela separátní kopie přiřazeného portu.[13] Je-li port přiřazen jen jednomu soketu, duplikace není třeba. Nepřiřazené porty jsou ostraněny.
3. Sloučit každý soket s přiřazeným portem.

[12] Výsledkem sloučení dvou uzlů je jeden uzel, přičemž množina s ním incidujících hran je sjednocením množin hran, incidujících s jednotlivými uzly.

[13] Výsledkem duplikace portu (přechodu) je přechod se stejnými vstupními a výstupními podmínkami jako duplikovaný přechod.



Obrázek 2.19: Substituční místo.

Nehierarchická síť, odpovídající hierarchické síti z obr. 2.19 je na obr. 2.20. Jak je vidět, substituční místa, stejně jako substituční přechody, nejsou proveditelnými součástmi modelu, protože jsou během simulace nahrazeny příslušnými instancemi podsítí.

Příklad hierarchické sítě. Jednoduchým příkladem hierarchické Petriho sítě je model večeřících filosofů na obr. 2.21.

Statické strukturování Petriho sítí není tak výrazným konceptem, že by měl zásadní vliv na provádění Petriho sítí nebo jejich analýzu, jde jen o zohlednění hierarchického modelování. Z praktického hlediska jsou však hierarchické Petriho sítě velmi významné, protože modelování většiny průmyslových aplikací vystačí se statickou instanciací sítí [Jen92].[14]

2.6 Dynamická instanciace Petriho sítí

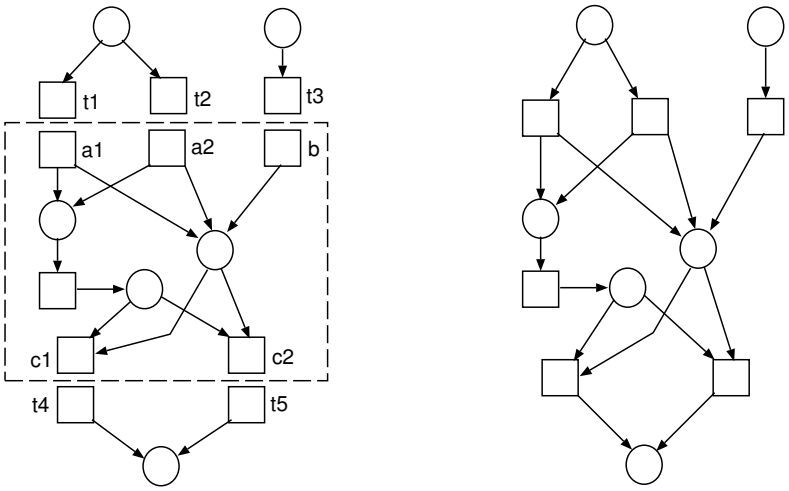
Petriho sítě, včetně hierarchických, umožňují přímo modelovat systémy, jejichž struktura se za běhu nemění. Statická instanciace sítí, kterou zavádějí hierarchické Petriho sítě, na této skutečnosti nic nemění, protože má charakter maker v programovacích jazycích, řeší se tedy v době kompilace. Za běhu je struktura sítě (v případě hierarchické sítě množina instancí stránek i stránky samotné) neměnná.

Místa a přechody v Petriho síti, případně struktury podsítí, mohou modelovat pouze statickou stránku systému. Vše, co se v systému mění za běhu, může být modelováno výhradně značkami v místech sítě. To je důvod, proč se systémy s proměnlivou strukturou Petriho sítěmi modelují obtížně.

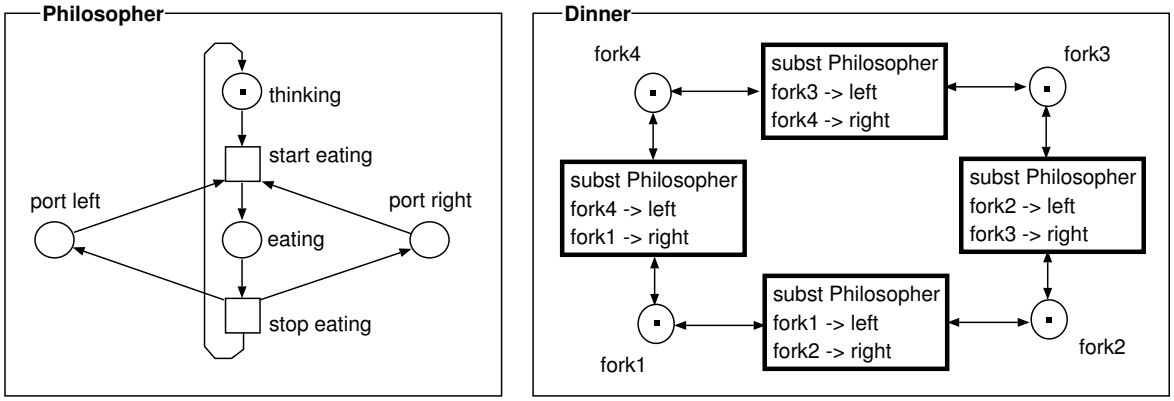
2.6.1 Invokační přechod

Jistý způsob řešení tohoto problému nabízí koncept invokace sítí. *Invokační přechod* je zde analogií volání procedury nebo funkce.

[14] Nejrozšířenější počítačový nástroj pro vysokoúrovňové Petriho sítě Design/CPN [Jen92] i formální definice hierarchické CPN [Jen92] pracují dokonce pouze se substitučními přechody a fúzemi míst (fúze míst bude vysvětlena později). Ostatní hierarchické konstrukty se zde nepoužívají.



Obrázek 2.20: Sémantika substitučního místa (podle [HJS90]).



Obrázek 2.21: Večeřící filosofové.

Koncept invokace sítí byl v [HJS90][15] zaveden tak, aby byl konzistentní s hierarchickými Petriho sítěmi. Syntakticky se invokační přechod podobá substitučnímu přechodu, včetně deklarace propojení soketů s porty. Jeho sémantika je však radikálně odlišná. Invokační přechod totiž je, na rozdíl od substitučního přechodu, proveditelnou součástí modelu, má však oproti normálnímu přechodu poněkud pozměněnou sémantiku, jak bude dále vysvětleno.

Síť, která se má dynamicky instanciovat invokačním přechodem, se také syntakticky podobá stránce, určené k substituci v hierarchické Petriho síti, má však navíc deklarovány *ukončující uzly* (místa nebo přechody označené klíčovým slovem `exit`).

Invokační přechod se provádí neatomicky. Jeho provádění je ohraničeno dvěma atomickými akcemi (událostmi) a spočívá v dynamickém rozšíření reprezentace modelu v době běhu o instanci invokované sítě (stránky):

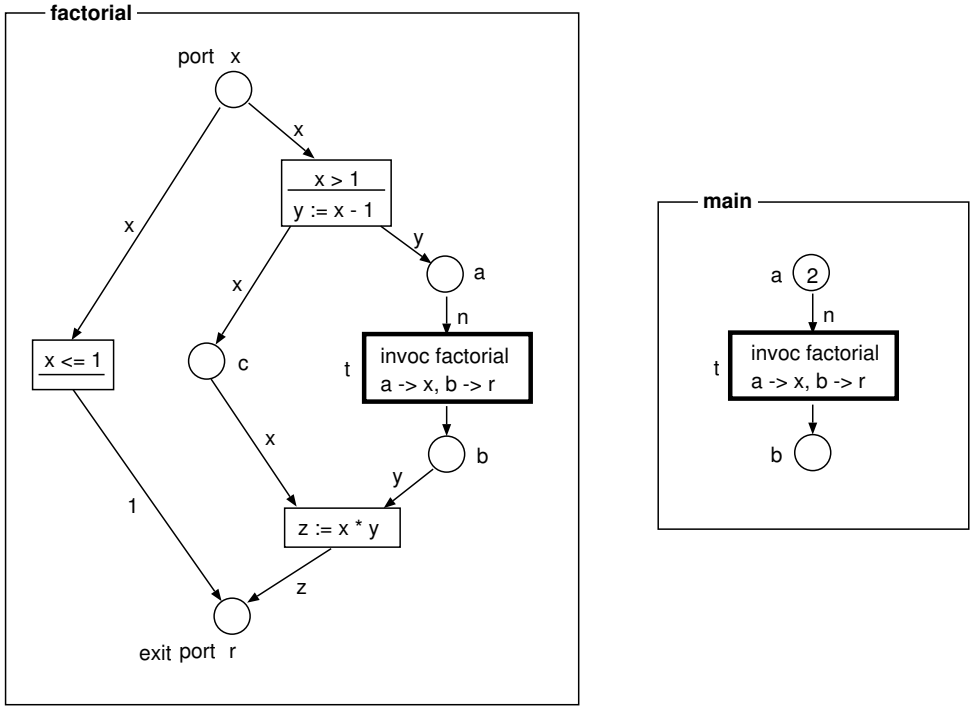
[15] Tento koncept byl představen v kontextu úvah kolem zavedení hierarchických barvených Petriho sítí, nebyl však implementován v rámci nástroje Design/CPN. Jiní autoři však podobný mechanismus používají [SSW95a, SSW95b].

1. Provede se vstupní část invokačního přechodu, což spočívá v odebrání značek ze vstupních míst invokačního přechodu a ve vytvoření nové instance specifikované stránky (sítě), do jejíchž vstupních míst jsou umístěny značky, odebrané ze vstupních míst invokačního přechodu.

2. Invokovaná síť posléze běží nezávisle na ostatních instancích sítí v systému, dokud v ní nedojde k nějaké ukončující události, jako je provedení ukončujícího přechodu nebo umístění značky do ukončujícího místa.

3. Dokončení invokace spočívá v provedení výstupní části invokačního přechodu. Značky z portů, přiřazených výstupním místům invokačního přechodu jsou do výstupních míst invokačního přechodu zkopírovány. Současně s provedením výstupní části invokačního přechodu dojde ke zrušení instance invokované sítě.

Jak je vidět, invokační přechod odpovídá volání procedury, přičemž procedura, popsaná Petriho sítí, je vyčíslena samostatným procesem, kterému se při vytvoření předají argumenty a po skončení je k dispozici výsledek. Tento mechanismus je znám v prostředí komunikujících procesů jako vzdálené volání procedur (RPC – Remote Procedure Call).



Obrázek 2.22: Demonstrace rekurzivní invokace.

Na obr. 2.22 je uveden poněkud netypický příklad, ale dostatečně reprezentativní pro demonstraci možností dynamicky instanciovaných sítí. Jde o rekurzivní výpočet faktoriálu. Později (v kapitole 4) se ještě k tomuto příkladu vrátíme.

2.7 Interakce mezi instancemi sítí

Jak v hierarchické Petriho síti, tak v případě invokace sítí, se objevují jistá omezení ve způsobu komunikace instancí sítí:

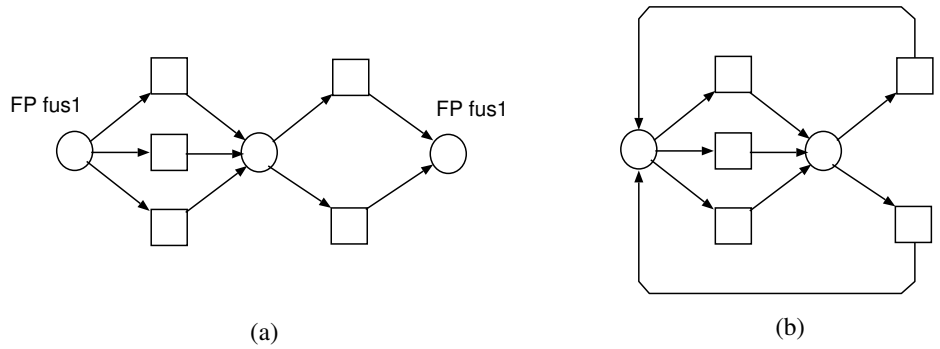
- Komunikace mezi instancemi stránek v hierarchické Petriho síti je omezena tak, že instance stránky s může komunikovat jen s hierarchicky nadřazenou instancí stránky (se stránkou, ve které nějaký substituční uzel stránku s instanciuje).
- Invokační přechod komunikuje s dynamicky instanciovanou sítí jen v okamžiku jejího vytvoření a v okamžiku jejího zrušení. Obecně však potřebujeme modelovat dynamicky vznikající a zanikající systémy, se kterými mohou za jejich života (existence, běhu) jiné systémy komunikovat.

2.7.1 Fúzní množiny

Výše uvedený problém řeší fúzní množiny uzlů sítí, zavedené také v [HJS90]. Každý uzel sítě může specifikovat jméno fúzní množiny, do které patří. Všechny uzly, patřící do jedné fúzní množiny jsou z hlediska běhu sítě sloučeny do jednoho uzlu.

Existují tři varianty fúzních množin uzlů:

1. *Instanční fúzní množina* má platnost jen uvnitř jedné instance sítě (stránky).
2. *Stránková fúzní množina* je sdílena všemi instancemi téže stránky.
3. *Globální fúzní množina* je dostupná všem instancím všech stránek.



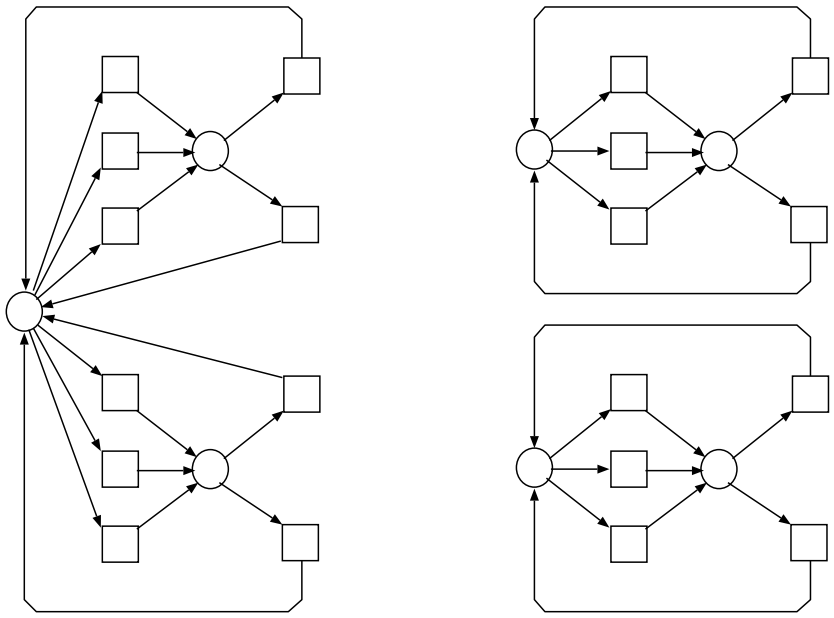
Obrázek 2.23: Fúzní množina.

Na obr. 2.23 je příklad fúzní množiny `fus1` (prefix `FP` říká, že jde o stránkovou fúzní množinu). V pravé části obrázku je odpovídající síť bez fúzní množiny. Obr. 2.24 ukazuje odpovídající síť bez fúzní množiny pro případ, že stránka z obr. 2.23 má dvě instance a fúzní množina byla v prvním případě (a) deklarována jako stránková a v druhém případě (b) jako instanční.

2.7.2 Synchronní kanály

Další variantou komunikace v rámci Petriho sítí jsou *synchronní kanály* [CH94]. Přechody mohou komunikovat prostřednictvím pojmenovaných kanálů. Vyjadřuje se to komunikačními výrazy typu $expr!?ch$ a $expr?!ch$ (tyto dvě varianty komunikačního výrazu nám umožní rozlišit stejně a opačně polarizované komunikační výrazy).[16] Přestože stejným kanálem může komunikovat více přechodů (více přechodů může specifikovat komunikaci tímtéž kanálem), jedno konkrétní provedení synchronní komunikace se vždy týká jedné dvojice přechodů s opačně polarizovanými komunikačními výrazy, specifikujícími stejný synchronní kanál.

[16] Synchronní kanály byly inspirovány mechanismy pro jednosměrnou komunikaci procesů, rozlišujícími odesilatele a příjemce synchronní zprávy. V případě synchronních kanálů jde o obousměrnou synchronní komunikaci, ale rozlišují se dva „konce“ synchronního kanálu.



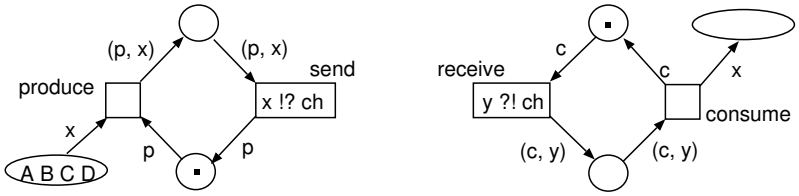
Obrázek 2.24: Sémantika stránkové (a) a instanční (b) fúzní množiny (podle [HJS90]).

Na obr. 2.25 je příklad producenta a konzumenta, komunikujících synchronním kanálem *ch* a na obr. 2.26 je odpovídající síť bez synchronních kanálů. Na obr. 2.27 je příklad dvou producentů a dvou konzumentů, komunikujících jedním synchronním kanálem *ch* a na obr. 2.28 je odpovídající síť bez synchronních kanálů. Jsou zde čtyři možné varianty komunikace přes synchronní kanál, modelované čtyřmi přechody.

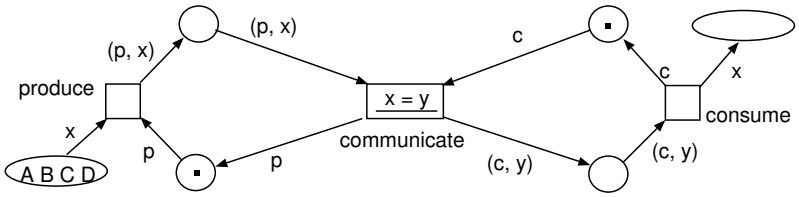
2.7.3 Identita instancí a komunikace ve víceúrovňových modelech

V případě dynamicky instanciovaných sítí je sice možné použít všechny dosud diskutované varianty komunikace, ale v některých případech by bylo vhodnější komunikovat s explicitně identifikovanou instancí sítě, dostupnou ve formě značky v Petriho síti. Takovýto způsob komunikace je výhodný v případě modelování interakcí ve víceúrovňových systémech, kde se jisté (aktivní) komponenty systému pohybují ve struktuře jiné komponenty systému. Jako příklad lze uvést křižovatku ([Lak94]), kde uvnitř vozidel, pohybujících se křižovatkou, probíhá vnitřní aktivita (spotřeba paliva, poruchy, chování řidiče atd.). Podobné situace nastávají při modelování některých workflow systémů a pružných výrobních systémů [Val98].

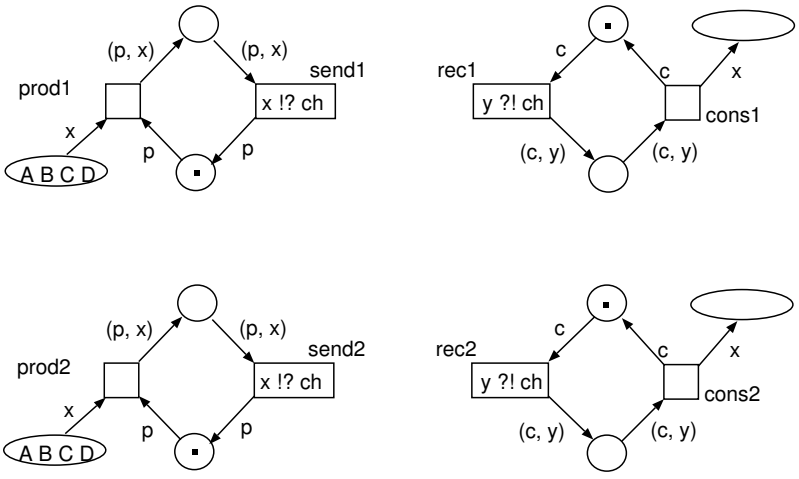
Interakci přechodu s instancí sítě, reprezentovanou značkou v Petriho síti lze realizovat umístěním značky do komunikačního místa sítě, reprezenované značkou, nebo synchronním provedením přechodu v síti, reprezentované značkou, a v síti, v níž se značka vyskytuje. První varianta se používá například v Cooperative Nets [SB94] a druhá varianta řeší koordinaci ve víceúrovňových sítích, které studuje R. Valk [Val95, Val98]. Tyto a podobné varianty komunikace dynamicky vznikajících a zanikajících instancí sítí již souvisí s některými způsoby zavedení objektové orientace do Petriho sítí a budou diskutovány v kapitole 4.



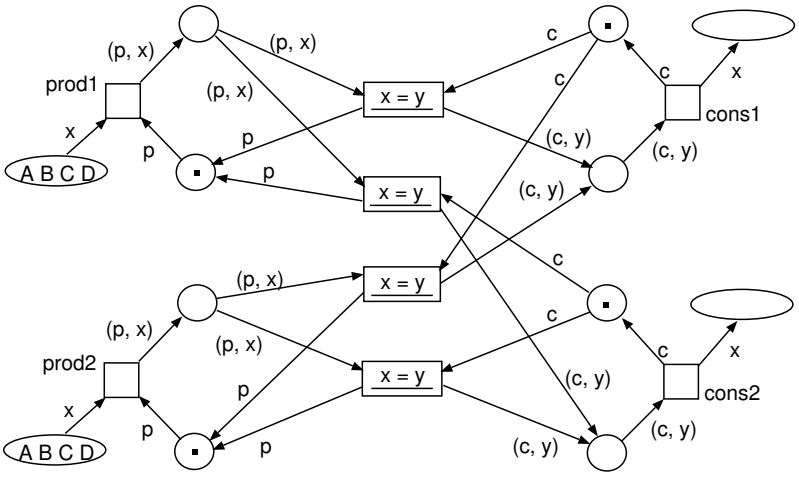
Obrázek 2.25: Komunikace s použitím synchronního kanálu.



Obrázek 2.26: Ekvivalentní síť k síti na obr. 2.25.



Obrázek 2.27: Komunikace různých přechodů tímtéž synchronním kanálem.



Obrázek 2.28: Ekvivalentní síť k síti na obr. 2.27.

Kapitola 3

Objektová orientace

Tato kapitola uvádí do problematiky objektové orientace. Zvláštní důraz je kladen na vztah objektů a procesů při modelování paralelních systémů. V závěru kapitoly je naznačena možnost spojení objektové orientace s Petriho sítěmi.

3.1 Základní principy objektové orientace

Prvním programovacím jazykem, který použil objektově orientované strukturování, byl jazyk Simula. Byl inspirován Algolem a zavedl objekty, třídy a dědičnost především jako mechanismy, určené k modelování skutečných objektů reálného světa pro potřeby počítačové simulace. Výhodou byla možnost přímo reprezentovat zkoumané objekty softwarovými objekty.

Smalltalk [GR83] byl prvním pokusem vyvinout plně (čistě) objektově orientovaný jazyk. Základním přínosem Smalltalku bylo rozpoznání objektové orientace nejen jako techniky pro simulační modelování, ale jako obecný přístup k tvorbě softwaru. Smalltalk je často označován za referenční objektově orientovaný jazyk, de facto definující pojmy objekt, třída a dědičnost.

Přesto však objektovou orientaci nelze považovat za uzavřené a přesně definované paradigma. Mnoho jazyků i metodologií si pojmy objektové orientace upravuje pro svoje potřeby. Dále se pokusíme představit základní principy objektové orientace, ve kterých se většina autorů shoduje [Mey88, Boo91, RBP$^+$91].

3.1.1 Objekty, operace a metody

Objekty představují specifický pohled na fyzikální a konceptuální jednotky reálného světa. Objekt je charakterizován třemi složkami:

1. *Stav.* Objekty nesou stavovou informaci. V daném čase se objekt nachází v určitém stavu. Stav je chápán jako vnitřní záležitost objektu (stavy různých objektů se navzájem neovlivňují). Například vhození dopisu do schránky změní stav poštovní schránky, stav dopisu se však nemění. Složité objekty mohou nést komplexní stavovou informaci. Díky abstrakci jsme však schopni omezit možné stavy objektu jen na stavy relevantní pro náš model.

2. *Chování.* Chování objektu definuje závislost stavu objektu na vnějších a vnitřních vlivech. Definuje, jak se změní stav při výskytu určitých podmínek, jako je například komunikace s jiným objektem. Některé objekty jsou pasivní (statické), jejich stav se mění jen v důsledku požadavků jiných objektů (například bankovní účet), jiné jsou aktivní (živé objekty, actors), samy si mění svůj stav (například hodiny). Objekty, které vůbec nemění stav, nazveme konstanty.

3. *Identita.* Každý objekt je jednoznačně identifikovatelný v rámci systému – má jedinečné jméno (identifikaci), kterým se liší od každého jiného objektu.

Klíčovým rysem objektové orientace je *zapouzdření* (encapsulation), tj. ukrytí vnitřní struktury objektu, přičemž viditelné je jen jeho rozhraní.

Veřejné rozhraní objektu je určeno množinou *operací*. Operace říká, *co* objekt umí. Například bankovní účet má operaci vklad. Předpis, který říká, *jak* operaci provést (algoritmus, sada pravidel), se nazývá *metoda*. Metoda, na rozdíl od operace, není součástí veřejného rozhraní objektu. Metody jsou skryty uvnitř objektu a definují jeho chování.

Pro zdůraznění nezávislosti objektů se komunikace objektů označuje jako *předávání zpráv*. Dává se tím najevo, že s daty objektu nelze manipulovat přímo, ale pouze zasláním zprávy objektu, po jejímž přijetí objekt sám rozhodne, jak na ni zareaguje – najde svou vlastní implementaci příslušné operace, tj. metodu, a provede ji.

Zapouzdření tedy odlišuje zprávu a reakci na ni, spolu s vlivem na stav objektu. Metody ani stav objektu nejsou vně objektu viditelné, jsou k dispozici jen nepřímo, prostřednictvím zpráv. Kromě toho různé objekty mohou tutéž zprávu zpracovávat různými metodami, což označujeme jako *polymorfismus*. Díky polymorfismu objektová orientace umožňuje tvorbu generického softwaru, který umí správně zacházet i s objekty, o jejichž existenci se ještě neví.

3.1.2 Instanciace, třídy a prototypy

Většina objektově orientovaných jazyků provádí klasifikaci objektů do *tříd*, přičemž třída reprezentuje množinu objektů se stejnou vnitřní strukturou. Lze rozlišit tři různé pohledy na definici třídy [Ber93]:

1. Třída je vzor nebo forma pro množinu strukturálně identických prvků. Prvky, vytvořené podle tohoto vzoru, nazýváme *instance* třídy.

2. Třída je objekt, obsahující vzor a mechanismus pro tvorbu instancí. Instance třídy vznikají aplikací zmíněného mechanismu.

3. Třída je množina všech prvků, které lze vytvořit podle určitého vzoru. Jinak řečeno, třída je množina všech instancí tohoto vzoru.

Ve většině objektově orientovaných jazyků třída objektu definuje množinu viditelných operací (nabízených služeb), množinu skrytých *instančních proměnných* (atributů objektu) a množinu skrytých metod, které implementují operace. Když je vytvořena nová instance třídy, tato má svou vlastní množinu instančních proměnných, zatímco metody operací sdílí s ostatními instancemi své třídy.

V některých jazycích je třída také považována za objekt. Je to zcela v souladu s principy objektově orientovaného modelování, protože třída reprezentuje množinu objektů, což je konceptuální entita modelované reality.

Je-li třída považována za objekt, je také instancí jisté třídy. *Metatřída* je třída, jejíž instance jsou třídy. Jelikož metatřída je také objekt, lze takto teoreticky pokračovat do nekonečna. Toto má smysl v případě, že je třeba modifikovat standardní chování tříd (způsob vytváření instancí). Většina jazyků však programování na úrovni metatříd nepodporuje a nabízí standardní sémantiku tříd.

Podobným konceptem je *parametrizovaná (generická) třída*. Je to vzor pro třídu, přičemž některé položky tohoto vzoru jsou považovány za parametry, jejichž hodnoty se musí doplnit. Nelze tedy přímo vytvářet instance parametrizové třídy, dokud nejsou definovány hodnoty parametrů. Teprve jejich doplněním vznikne třída, která umožňuje tvorbu svých instancí.

Alternativou k instanciaci tříd jsou *prototypové objekty* jako vzory pro instance. Instance prototypu jsou vytvářeny prostým kopírováním prototypu a případnou modifikací vzniklé kopie. Tento přístup je vhodný v případech, kdy objekty se rychle vyvíjejí a vykazují více odlišností než společných rysů. Typickým příkladem „beztřídního“ jazyka s prototypovými objekty je Self [US87].

Objekty mohou být instanciovány staticky a dynamicky. Staticky instanciovaný objekt je alokovaný v době kompilace a existuje po celou dobu provádění programu. Dynamicky instanciované objekty vznikají a zanikají v průběhu výpočtu.

3.1.3 Dědičnost

Velmi výrazným (i když ne nejdůležitějším) rysem objektové orientace je dědičnost. Dědičnost obecně umožňuje objektu sdílet jisté charakteristiky jiného (obecnějšího) objektu.

V objektově orientovaných jazycích se objevují různé formy dědičnosti [Nie89]. Může jít o statickou nebo dynamickou dědičnost (vysvětlíme dále), může se týkat tříd nebo instancí, může být děděna reprezentace nebo chování, dědičnost může být jen částečná (něco se zdědí a něco ne), popřípadě lze rozlišit násobnou a jednoduchou dědičnost podle počtu objektů nebo tříd, od kterých daný objekt nebo třída dědí.

Smalltalk [GR83] používá *jednoduchou dědičnost tříd*. Každá třída je definována jako podtřída jiné (obecnější) třídy. Dědí se *reprezentace* (množina instančních proměnných) a *chování* (množina metod). Zděděné proměnné a metody mohou být doplněny novými, zděděné metody mohou být nahrazeny novými verzemi. Jiné jazyky (C++) umožňují násobnou dědičnost tříd. Sémantika dědičnosti se tím však poněkud komplikuje.

Dědičnost tříd je většinou jazyků chápána jako statická forma dědičnosti – nová třída dědí v době definice, nikoli v době běhu programu. Jakmile je třída definována, vlastnosti všech jejích instancí jsou fixovány. Připustíme-li redefinici tříd za běhu, všechny instance nově definovaných tříd a jejich podtříd musí adekvátně změnit svoji strukturu a chování.

Dynamická dědičnost je mechanismus, umožňující objektům měnit své chování v rámci normálních interakcí objektů (na rozdíl od rekompilace tříd), uvnitř objektového modelu. Lze rozlišit (podle [Nie89]) dva druhy dynamické dědičnosti: part-inheritance a scope-inheritance. V prvním případě objekt explicitně změní své chování tím, že vymění svou část, v druhém případě objekt mění své chování implicitně (nepřímo) v důsledku změn v jeho okolí.

Dynamická dědičnost (obě varianty) je typická pro „beztřídní“ prototypové objektově orientované systémy, jako je například Self [US87]. Objekt zde má také instanční proměnné a metody,[1] ale může delegovat příchozí zprávy na jiný objekt. Part-inheritance se projeví, když si objekt změní objekt určený k delegaci, zatímco scope-inheritance se projeví, když objekt, na který deleguje, změní své chování. Poznamenejme, že dynamickou dědičnost lze emulovat i v třídním prostředí a naopak, třídy je možné bez problémů emulovat v beztřídním prostředí.

Abstraktní třídy

Obvykle si představujeme třídy jako úplné definice. Existují však situace, kdy je užitečná neúplná definice. Předpokládá se přitom, že tato neúplná definice bude doplněna třídou, která tuto neúplnou definici zdědí. Takovýto neúplný, abstraktní, avšak užitečný koncept vyjádříme třídou, která sice specifikuje rozhraní instancí, ale neimplementuje je. Takováto třída není přímo použitelná k tvorbě instancí, ale je určena ke konkretizaci (specializaci) prostřednictvím dědičnosti jinými třídami, proto se nazývá abstraktní třída.

[1] Zvláštností jazyka Self je skutečnost, že atributy i metody objektu lze v rámci dědičnosti vzájemně předefinovávat. Jak atributy, tak metody jsou považovány za vzájemně zaměnitelné *sloty* objektu.

3.2 Objektová orientace a paralelismus

Doposud jsme se zabývali objektovou orientací obecně. Má-li být objektová orientace (objektově orientovaný jazyk) použitelná pro modelování reálných systémů, musí být schopna adekvátně modelovat nezávislé paralelní aktivity (procesy) v systému a jejich vzájemnou komunikaci a synchronizaci. Nyní se zaměříme na to, jak se objektově orientované jazyky vyrovnávají s paralelismem.

3.2.1 Používané přístupy z hlediska autonomie objektů

Používané přístupy závisí na tom, do jaké míry je respektována nebo porušována autonomie objektů v paralelním prostředí. Použijeme dělení podle [Nie93]. Jemnější dělení lze nalézt v [Pap89].

Ortogonální přístup

Některé jazyky ponechávají koncepty paralelismu (vytváření a komunikaci procesů) a objektové orientace (vytváření a komunikace objektů) nezávislé. Zodpovědnost za korektní kombinaci těchto konceptů je na programátorovi. Tento přístup k paralelní objektové orientaci nazveme *ortogonální*. V případě ortogonálního přístupu musí programátor pomocí standardních synchronizačních prostředků (např. semaforů a monitorů) ošetřit situaci, kdy s objektem pracuje více procesů současně. K jazykům tohoto druhu patří Smalltalk-80 [GR83], Emerald [BHJL87], Trellis/Owl [EMK87], Java [Fla96].

Nevýhodou tohoto přístupu je skutečnost, že objekt je možno korektně použít jen se znalostí jeho implementace (jeho schopnosti vyrovnat se s paralelismem). Je zde tedy porušen princip zapouzdření a zkomplikována znovupoužitelnost objektů.

Heterogenní přístup

Jiné jazyky definují objekty tak, že již obsahují prostředky pro použití v paralelním prostředí a rozlišují se zde aktivní a pasivní objekty. Pasivní objekty mohou být použity jen uvnitř obvykle jednoprocesových aktivních objektů (nemohou tedy být použity více aktivními objekty současně). Aktivními objekty jsou procesy, které mohou vzájemně komunikovat. K jazykům tohoto druhu patří paralelní rozšíření jazyka Eiffel [Car89], ACT++ (paralelní rozšíření C++) [KL89].

Homogenní přístup

V případě homogenního přístupu jsou všechny objekty (potenciálně) aktivní a jsou schopny synchronizovat požadavky na provádění svých metod. K jazykům tohoto druhu patří ABCL/1 [YBS87], POOL-T [Ame87], Hybrid [Nie87].

Je zřejmé, že jedině tento přístup je plně v souladu s obecnými představami o objektové orientaci, protože nerozlišuje objekty na speciální podtřídy (aktivní a pasivní) a respektuje princip zapouzdření i v případě paralelních požadavků na invokace metod.

3.2.2 Interakce procesů a objektů

Dále se budeme zabývat komunikací procesů a souvislostí procesů s objekty.

Komunikační mechanismy

Existují dva způsoby, jak se programovací jazyky (bez ohledu na objektovou orientaci) vyrovnávají s paralelismem a komunikací [Nie89, AS83]:

1. Aktivní entity (procesy) komunikují nepřímo, prostřednictvím sdílených pasivních objektů (proměnných, databází apod.)

2. Aktivní entity komunikují přímo, předáváním zpráv.

V prvním případě rozlišujeme *aktivní* a *pasivní objekty*. Aktivní objekty musí synchronizovat přístup k pasivním objektům (semafory, zámky, monitory, transakce). Aktivní objekty nemohou přímo komunikovat. Lze tedy říci, že druhý přístup je v jistém smyslu obecnější. Každý objekt může libovolnému objektu poslat zprávu. Není nutná explicitní synchronizace, protože zprávy řeší jak komunikaci, tak synchronizaci.

Předávání zpráv může být synchronní nebo asynchronní (s vyrovnávací pamětí), jednosměrné nebo může používat protokol klient-server (dotaz/odpověď), případně může být povoleno přerušení běhu aktivních objektů naléhavou zprávou.

Všechny uvedené přístupy mají stejnou vyjadřovací sílu a jsou vzájemně převoditelné. Jazyky a systémy, podporující komunikaci procesů prostřednictvím sdílené paměti se však obtížněji realizují v distribuovaném prostředí. Komunikace předáváním zpráv je v tomto ohledu pružnější a umožňuje komunikujícím objektům větší nezávislost. Přitom jednosměrná komunikace je jednodušší (implementačně), ale klient-server protokol podporuje strukturovanější programování.

Máme-li tyto komunikační mechanismy porovnat s komunikací objektů, je zřejmé, že pro implementaci předávání zpráv objektům, chápaným tak, jak jsou implementovány většinou programovacích jazyků, je přirozené použít *klient-server protokol*, tedy schema, kde klient předá zprávu a čeká na odpověď (výsledek, potvrzení).[2]

Procesová struktura objektu

Procesová struktura objektu [Pap89] určuje počet procesů, které mohou být současně aktivní uvnitř objektu, a jak jsou tyto procesy vytvářeny a synchronizovány.

Počet procesů v objektu se *statickou procesovou strukturou* je pevný. Programátor musí zajistit přepínání (směrování) požadavků na operace mezi těmito procesy. V tomto případě je třeba vzhledem k omezenosti zdrojů počítat s podmínečným přijetím požadavku. Speciálním případem tohoto druhu jsou jednoprocesové objekty.

V případě *dynamické procesové struktury* dochází k vytváření a rušení procesů uvnitř objektu. Programátor zde musí zajistit synchronizaci těchto procesů. Vytváření procesů se zde (obvykle) děje automaticky při invokaci metod.

Procesy lze rozdělit na sekvenční, kvaziparalelní a paralelní. Se sekvenčním procesem je spjat jeden tok řízení, kvaziparalelní proces má více toků řízení, ale jen jeden je aktivní v jednom časovém okamžiku a paralelní proces má více paralelních toků řízení, které jsou současně aktivní.

Lze tedy rozlišit jednoprocesové a víceprocesové objekty, které mohou být realizovány sekvenčními, kvaziparaleními a paralelními procesy.[3]

[2] Nevylučujeme však možnost, že pro jiné chápání objektů, což je vzhledem k volnosti ve výkladu tohoto pojmu možné, může být výhodnější jiný typ komunikace.

[3] Jemnější dělení, které přesahuje rámec této práce, lze nalézt například v [Nie93].

3.2.3 Agenti

Úvahy o integraci paralelismu a objektové orientace se potkávají s konceptem agentů, známým z umělé inteligence. Není to náhodou, neboť programovací jazyky se snaží prostřednictvím objektové orientace tvorbu softwaru přiblížit pojmům reálného světa a umělá inteligence se snaží fungování reálného světa pochopit prostřednictvím adekvátních modelů.

Pojem *agent* se poprvé objevil v oblasti psychologie. Zavedl ho M. Minsky, zabývající se modely fungování lidské mysli [Min85]. V rámci umělé inteligence se zkoumají systémy agentů (multi-agent-systems) při řešení problémů.

Hlavní rysy agentů jsou podle [Sho93, Hol95a, MW97] tyto:

1. *Nezávislost.* Agent je schopen vykonávat určitou činnost a to paralelně s ostatními agenty i s ostatními činnostmi, které sám provádí. Je také schopen některé činnosti delegovat na jiné agenty. Dalším aspektem nezávislosti agenta je vlastnictví zdrojů (znalostí).

2. *Komunikace.* Agenti komunikují předáváním zpráv.

3. *Inteligence.* Agent má schopnost na základě znalostí dedukovat nové znalosti.

Nehledě na problematičnost pojmu inteligence je vidět, že jde o koncept, který plně odpovídá představě víceprocesového objektu.

Tato krátká informace o agentech je zde zmíněna z důvodu vyjasnění souvislosti tohoto v poslední době často citovaného pojmu s předmětem této práce.

3.3 Objektový model pro Petriho sítě

Nyní specifikujeme objektový model (konkrétní představu o objektech a jejich komunikaci), který bude později možné spojit s Petriho sítěmi s cílem definovat objektově orientovaný formalismus pro modelování paralelních systémů. Obecné pojmy, uvedené výše, budou nyní upřesněny.

3.3.1 Objekty, zprávy a metody

Systém budeme modelovat množinou objektů, které mohou během existence systému dynamicky vznikat a zanikat. Každý objekt je jednoznačně identifikovatelný jménem, lze ho kdykoliv (nezávisle na čase) odlišit od každého jiného objektu.

Každý objekt se v daném čase nachází v určitém stavu, tento stav se může s časem měnit. Každý objekt má svůj stav pod kontrolou, žádný jiný objekt ho nemůže přímo ovlivnit. Stav objektu vyjadřuje veškeré relevantní znalosti objektu v daném čase, například může obsahovat jména ostatních objektů. Každý objekt poskytuje (zveřejňuje) množinu služeb, které mohou být využívány jinými objekty. Realizace služeb (metody), stejně jako reprezentace stavu objektu a realizace jeho vlastní nezávislé aktivity (tzv. *implicitní metoda*) jsou privátními záležitostmi objektu.

Každý objekt může požádat jiný objekt, jehož identifikaci zná,[4] o provedení služby tím, že mu pošle zprávu. Zpráva specifikuje jméno objektu, který má službu provést, jméno požadované operace a případné parametry (což jsou opět jména objektů). Objekt reaguje na příchozí zprávu tím, že vybere vhodnou metodu a provede ji. Tím může dojít ke změně jeho stavu. Po dokončení metody vrátí výsledek (což je opět jméno nějakého objektu) jako odpověď na zprávu. Za výběr metody a její provedení je zodpovědný příjemce požadavku. Objekt může změnit svůj stav buď

[4] O existenci okolních objektů se objekt může dovědět při svém vytvoření, nebo dodatečně z informací, získaných od jiných objektů.

samostatně (z vnitřní příčiny, podle implicitní metody, popisující nezávislou aktivitu objektu), nebo v důsledku poskytnutí služby jinému objektu (podle metody pro obsluhu požadavku).

3.3.2 Třídy

Objekty jsou popsány třídami. Třídy definují společné vlastnosti příslušné množiny objektů, tj. reprezentaci a chování. Reprezentací rozumíme strukturu paměti a chováním rozumíme množinu metod, včetně implicitní metody. Každý objekt je instancí nějaké třídy a je charakterizován identifikací a stavem v daném okamžiku, resp. průběhem změn stavu v čase.

Třídy jsou speciální objekty,[5] které reprezentují množiny potenciálních objektů, které jsou realizovány stejným způsobem, tj. mají stejnou strukturu paměti (reprezentaci stavu) a stejné metody pro realizaci stejné množiny služeb. Prvek takové množiny nazýváme instancí příslušné třídy. Instance jedné třídy se v daném čase mohou lišit (kromě jména) jen stavem (a tím, zda právě existují, nebo ne). Třídy poskytují službu vytváření instancí. Každý objekt může třídu požádat o vytvoření její nové instance. Tímto způsobem mohou dynamicky vznikat nové objekty. Třídy mají charakter konstant. Zatímco třídy existují staticky, jejich instance se podílejí na dynamickém běhu systému.

Dědičnost. Třídy se definují inkrementální modifikací existujících tříd využitím dědičnosti. Každá třída specifikuje svoji nadtřídu, tedy třídu, od které dědí.[6] Dědí se reprezentace a chování. Dědičnost reprezentace objektu spočívá v tom, že lze přidat další atributy objektu (rozšíříme tím strukturu jeho paměti). Dědičnost chování spočívá v tom, že lze přidávat další metody a zděděné metody lze nahrazovat jinými definicemi.

3.3.3 Paralelismus

Objekty existují souběžně v čase. *Existence objektu* je množina všech okamžiků, kdy objekt existuje [Kin80]. V každém okamžiku své existence se objekt nachází v určitém stavu. Sekvenci změn stavu objektu v čase označíme za *proces objektu*.

Podobně i provádění metody je aktivita, probíhající v čase. Může se v čase překrývat s prováděními jiných metod nebo i s jinými prováděními téže metody. Proces objektu je výsledkem paralelního provádění *procesů metod* objektu během jeho existence. Jde tedy o víceprocesový objekt s dynamickou procesovou strukturou.

Zpracování požadavku neproběhne okamžitě a objekt (některá jeho metoda), který požaduje službu, musí čekat na její dokončení. Metody nemusí být specifikovány sekvenčně, je tedy možné během čekání na dokončení zpracování požadavku jiným (nebo i tímtéž) objektem paralelně provádět jinou činnost. Lze tedy hovořit o paralelismu uvnitř metody (kromě paralelismu mezi metodami).

Zapouzdření procesů. Objekt zapouzdřuje množinu procesů, komunikujících prostřednictvím sdílené paměti – procesy metod tak mohou ovlivňovat stav objektu. Objekty navzájem komunikují výhradně předáváním zpráv (protokolem klient-server). Přitom zpráva způsobí vytvoření procesu pro odpovídající metodu uvnitř příjemce zprávy. Po vyčíslení metody a předání výsledku je tento proces zrušen. Odesilatel zprávy (přesněji, dílčí proces uvnitř příslušného objektu) po předání zprávy vždy čeká na odpověď (výsledek). Mezitím ostatní procesy uvnitř objektu běží.

[5] Odlišují se od jiných objektů tím, že vznikají v době kompilace a za běhu modelu se nemění.

[6] Budeme používat jednoduchou dědičnost.

Objekty tedy chápeme jako vyšší celky nad procesy. Tyto celky komunikují jiným (volnějším) způsobem, než procesy uvnitř nich. Toto radikální odlišení způsobu komunikace mezi objekty a uvnitř objektů přímo odráží základní princip objektové orientace, a sice zapouzdření stavu a chování objektů.

3.3.4 Dynamika objektů

Systém je během své existence (běhu) realizován množinou jednoznačně identifikovaných objektů, zapouzdřujících stav a metody pro obsluhu požadavků, přicházejících formou zpráv.

Systém je (staticky) definován konečnou množinou tříd $\mathbb{C}$ a počáteční třídou $C_0 \in \mathbb{C}$. Předpokládá se implicitní existence množiny primitivních objektů, obsahující alespoň množinu celých čísel (spolu s celočíselnou aritmetikou). Třídy a primitivní objekty existují staticky a spolu se specifikací počáteční třídy implicitně vymezují chování systému. Dynamiku systému můžeme zkoumat na úrovních (1) systém, (2) objekt a (3) proces. Objekt je instancí třídy (je definován třídou), proces[7] je instancí metody (je definován metodou).

Reprezentace stavu systému je organizována hierarchicky. Okamžitý stav systému (konfigurace systému) je určen(a) stavem všech jeho objektů, stav objektu je určen stavem všech jeho procesů.

Vývoj systému začíná počáteční konfigurací, která obsahuje prvotní objekt, což je instance (počáteční) třídy C_0. Další vývoj systému je dán chováním prvotního objektu (měl by to být aktivní objekt).

Pro dynamické vytváření objektů je třeba zavést zdroj, který lze v průběhu výpočtu (simulace, běhu) alokovat. Na teoretické úrovni bude tento zdroj pro jednoduchost nekonečný. Na počátku tedy máme (nekonečnou) množinu objektů, které se nacházejí ve stavu neživých (neexistujících). Každá třída ma svoji doménu – množinu potenciálních instancí. Vytvoření objektu pak spočívá ve vyhledání vhodné neživé instance příslušné třídy a jejím oživení. Toto provede třída na žádost nějakého objektu. Během existence systému jsou objekty vytvářeny, existují a zanikají. Aby bylo možné objekt jednoznačně identifikovat bez ohledu na čas, je vhodné zavést axiom, že každý objekt může být vytvořen právě jednou (reinkarnace objektu tedy není možná).[8] Uvažujeme-li jeden určitý časový okamžik, každý objekt je buď neživý (ještě nevznikl, nebo už zanikl), nebo existuje a nachází se v některém ze svých stavů. Z praktického hlediska je důležité, že neživý objekt neobsazuje žádnou paměť. Ta se alokuje až při vytvoření. Po likvidaci objektu se paměť opět uvolní a je recyklovatelná (jméno objektu však nikoliv[9]).

Totéž, co bylo řečeno o dynamickém vytváření objektů, platí i pro vytváření procesů (instancí, invokací) metod. Při akceptování zprávy objekt vybere příslušnou metodu, a odstartuje nepoužitou instanci metody[10] (s inicializací a předáním parametrů a kontextu, který je dán stavem objektu). To je standardní chování všech objektů, které nejsou deklarovány jako konstanty.

Procesy jsou tedy vytvářeny objekty jako reakce na přicházející zprávy. Existence objektu a instance implicitní metody (definující nezávislou vnitřní aktivitu objektu) jsou totožné. Existence (doba života) instance obyčejné metody je ohraničena přijetím příslušné zprávy a odesláním odpovědi. Je totožná s dobou provádění komunikace, tj. s dobou čekání klienta na odpověď. Komunikace s konstantou (třídou a primitivním objektem) je vždy atomická – předání zprávy a odpověď proběhnou okamžitě, bez čekání; konstanta nikdy nevytváří procesy.

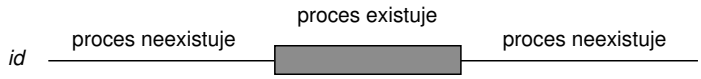
[7] Myslíme tím proces metody. Proces objektu ztotožňujeme s pojmem objekt a je paralelní kompozicí všech procesů metod daného objektu.

[8] Tento axiom má význam pro některé teoretické úvahy. Prakticky většinou vystačíme s recyklovatelnými jmény objektů.

[9] S respektováním poznámky 8.

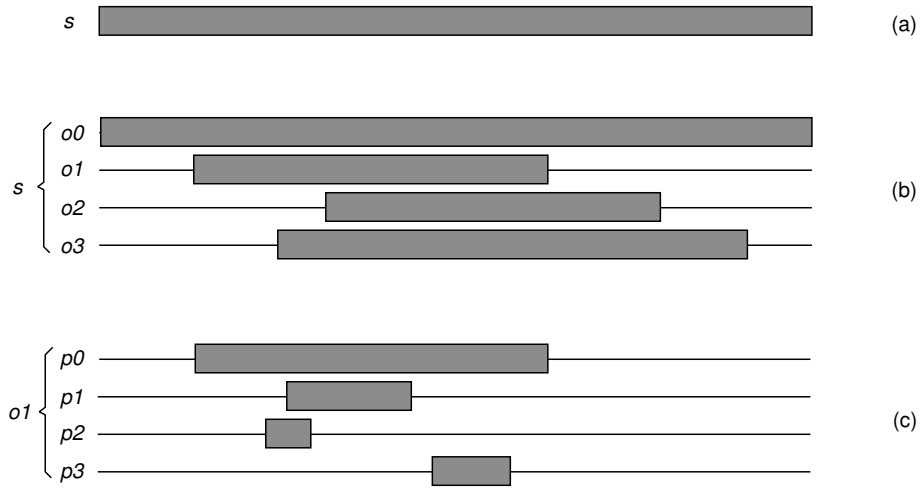
[10] Nebo (vzhledem k poznámce 8) inicializuje a odstartuje takovou instanci metody, která momentálně v systému neexistuje.

Proces, podobně jako objekt, má jednoznačnou identifikaci (jméno, číslo, adresu), která je nezávislá na čase i na jeho existenci a umožňuje ho odlišit od ostatních procesů v systému. To nám umožní podrobně sledovat vývoj systému v čase a znázornit ho časovým diagramem (viz obr. 3.1).



Obrázek 3.1: Časový diagram procesu (čas plyne zleva doprava).

Časový diagram systému (v případě nedeterministického systému sledujeme jen jednu cestu jeho stavovým prostorem) je na obr. 3.2a. Systém existuje z hlediska své časové množiny věčně. Skládá se z objektů. Lze tedy provést zjemnění časového diagramu (viz obr. 3.2b). Poznamenejme, že počáteční objekt *o0* existuje stejně dlouho, jako celý systém. Každý objekt se skládá z procesů. Například objekt *o1* může být zjemněn podle obr. 3.2c. Proces *p0* je proces implicitní metody, definující vlastní aktivitu objektu, ostatní procesy jsou procesy metod, vznikající a zanikající v důsledku komunikace objektů.



Obrázek 3.2: Časový diagram (a) systému s, (b) zjemnění systému s, (c) zjemnění objektu *o1*.

Procesy operují částečně nad svým lokálním stavem, částečně nad sdíleným stavem v rámci objektu. Změny stavu jsou *události*. Každou událost můžeme zařadit do jedné z těchto kategorií:

1. *Vnitřní událost objektu.*

   Vnitřní událost změní stav objektu, uvnitř kterého k ní došlo. Stav ostatních objektů zůstává nezměněn. Interakce s primitivními objekty považujeme také za vnitřní události.

2. *Předání zprávy (přijetí zprávy).*

   Předání požadavku na službu (zpráva) je událost, která proběhne současně s přijetím zprávy jiným objektem.[11] Předání zprávy způsobí změnu stavu jak u odesilatele, tak u příjemce zprávy (předání a přijetí zprávy jsou dva různé pohledy na tutéž událost).

[11] Tento model komunikace neuvažuje zpoždění mezi odesláním a přijetím zprávy.

3. *Odpověď na zprávu (přijetí odpovědi).*

   Totéž platí o odpovědi na zprávu. Při předání zprávy a odpovědi na ni tedy dochází k synchronní změně stavu obou zúčastněných objektů. Předání (přijetí) zprávy a odpověď (přijetí odpovědi) jsou kauzálně svázány a týká se vždy jedné dvojice objektů.

4. *Vytvoření nového objektu.*

   Vytvoření objektu změní stav objektu, ve kterém k této události došlo, a současně nový objekt přejde z neexistence do počátečního stavu (je pro něj alokována paměť).

Rušení objektů. Konec existence objektu je otázka, která dosud nebyla diskutována. Teoreticky mohou být objekty rušeny explicitně (zasláním zprávy) nebo implicitně (provede to garbage-collector, který může být aktivován buď asynchronně, nebo synchronně s jinými událostmi), na základě splnění určitých podmínek.

V našem případě půjde o implicitní zánik objektů, které nejsou prostřednictvím referencí tranzitivně dostupné z provotního objektu. Zároveň se zánikem objektu jsou rekurzivně zrušeny všechny procesy metod, volaných z tohoto objektu. K tomuto dochází synchronně se všemi výše uvedenými událostmi.

Konstanty. Připoměňme, že ne všechny objekty jsou složeny z dynamicky se vyvíjejících procesů. Konstanty, což jsou primitivní objekty a třídy, nemění stav a existují věčně (z hlediska existence systému). Veškeré požadavky zpracovávají okamžitě a atomicky, beze změny stavu.

3.3.5 Aplikace Petriho sítí v modelování objektů

Metoda může být specifikována množinou pravidel typu IF <*podmínka*> THEN <*akce*>, přičemž *podmínka* je predikát nad stavem systému a *akce* je přechod do nového stavu a případné odeslání požadavku na službu některého objektu s následným čekáním a přijetím výsledku. Pravidla mohou být aplikována paralelně, přístup k reprezentaci stavu objektu však musí být nějakým způsobem synchronizován. Úvahy o paralelní aplikaci pravidel a jejich synchronizaci mohou přímo vést k použití Petriho sítí jako jazyka pro specifikaci metody.[12]

Paměť pro reprezentaci stavu objektu je implementována množinou míst. Jednotlivé invokace (procesy) metod mohou mít své lokální stavy, tj. každá metoda může mít svoji lokální paměť. Lokální paměti metod jsou také implementovány množinami míst. Pravidla jsou implementována přechody.

Stav systému je dán stavy všech objektů. Stav objektu je určen stavem všech jeho právě prováděných procesů metod, stav procesu metody je určen značením míst jeho příslušné Petriho sítě. Změna stavu systému je způsobena buď provedením přechodu sítě v některém procesu některého objektu, nebo předáním zprávy, nebo přijetím odpovědi přechodem sítě. Místa mohou obsahovat multimnožiny značek, které nesou reference na objekty.

Toto je ovšem velmi hrubá představa o spojení Petriho sítí a objektů. V následující kapitole bude tato problematika popsána podrobněji.

[12] Přestože Petriho síť je schopna modelovat i strukturální stránku systému (toky entit v systému), my ji budeme primárně chápat jako specifikaci chování (souvislost parciálních stavů a událostí).

Kapitola 4

Modelování objektů Petriho sítěmi

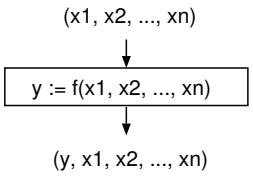
Úkolem této kapitoly je vysvětlit princip spojení Petriho sítí s objekty, který může být základem formalismu a nástroje pro modelování, prototypování a simulaci systémů.

Jako mezikrok na cestě k objektové orientaci v Petriho sítích představíme funkcionální strukturování Petriho sítí, kde se v omezené podobě objeví problémy, které musíme řešit při modelování objektů Petriho sítěmi. Objektově orientované strukturování pak představíme jako zobecnění a modifikaci strukturování funkcionálního. V závěru kapitoly provedeme stručné srovnání s alternativními přístupy k objektové orientaci v Petriho sítích.

4.1 Funkcionální strukturování Petriho sítí

4.1.1 Invokační přechod jako volání funkce

Invokační přechod, jak byl uveden v kapitole 2, je analogií volání funkce. Chceme-li do Petriho sítí zavést stejná strukturování, jaká zavádějí programovací jazyky, je třeba známé strukturovací mechanismy Petriho sítí poněkud modifikovat. Konkrétně je třeba syntax invokačního přechodu upravit tak, aby byl zcela kompatibilní s voláním funkce v rámci inskripčního jazyka Petriho sítě. Uvažujme přechod na obr. 4.1, který volá funkci `f(x1, x2, ..., xn)`.

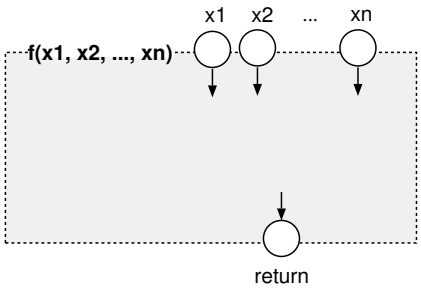


Obrázek 4.1: Přechod s akcí `y := f(x1, x2, ..., xn)`.

Je třeba zajistit, aby nebylo třeba syntax tohoto přechodu měnit pro případ, že by funkce `f(x1, x2, ..., xn)` byla implementována Petriho sítí. Má-li být funkce `f(x1, x2, ..., xn)` implementována Petriho sítí, tato síť musí mít odpovídající počet portů (vstupních, parametrových míst) pro předání parametrů a jeden výstupní port (výstupní, ukončující místo) pro výsledek,[1] který může současně hrát roli ukončujícího místa (viz obr. 4.2).

Každou invokovatelnou síť lze tedy považovat za implementaci funkce, kterou je možno volat uvnitř akce přechodu. Syntax volání funkce v akci přechodu je jednotná jak pro funkce,

[1] Připustíme-li zápis `(y1, y2, ..., ym) := f(x1, x2, ..., xn)`, síť, implementující funkci `f`, musí mít odpovídající počet výstupních míst, nebo to znamená, že ve výstupním místě se očekává m-tice. Těmito variantami se pro jednoduchost nebudeme zabývat.



Obrázek 4.2: Implementace funkce `f(x1, x2, ..., xn)` Petriho sítí.

definované v rámci inskripčního jazyka, tak pro funkce, definované Petriho sítěmi, a není třeba v případě invokace sítě specifikovat propojení portů.

Takto získaná kompatibilita inskripčního jazyka a strukturovacího mechanismu pro Petriho sítě umožňuje lepší znovupoužitelnost již vytvořených sítí díky nezávislosti implementace sítí na způsobu implementace volaných funkcí. Lze si představit případ, že vytvoříme síť, jejíž přechod volá funkci `f`, nejprve specifikovanou v rámci inskripčního jazyka, a při přechodu k jemnějšímu modelu tuto funkci specifikujeme Petriho sítí. Volající síť pak zůstává beze změny. Je zde respektován princip funkcionálního strukturování: implementace funkce je nezávislá na implementaci ostatních funkcí.

Co se týče balancování mezi modelováním Petriho sítěmi a inskripčním jazykem, teoreticky může být funkcionální inskripční jazyk nahrazen Petriho sítěmi až na celočíselnou aritmetiku, což je postačující pro inskripční jazyk HL-sítě. Které skutečnosti budou vyjádřeny strukturou sítě a které jejími inskripcemi prakticky záleží na účelu modelu. Tam, kde je důležité modelovat chování, tedy postup výpočtu, použijeme síť, kde jde jen o získání výsledku libovolným způsobem, použijeme inskripční jazyk.

4.1.2 Funkcionálně strukturovaná Petriho síť

Funkcionálně strukturovaná Petriho síť je meziprodukt na cestě k objektům. Je samostaně použitelná, její praktická použitelnost je však diskutabilní.[2] Zkoumání problematiky funkcionálního strukturování Petriho sítí nám však umožní zvládnout model s dynamickou instanciací Petriho sítí. Ve zjednodušené podobě se zde objeví problémy, které je třeba řešit v případě objektově orientovaného strukturování. Objektovou orientaci potom do Petriho sítí zavedeme na základě analogie s funkcionálním strukturováním.

Funkcionálně strukturovaná (funkcionální) Petriho síť (FPN) se skládá z množiny *primitivních funkcí* a množiny funkcí definovaných vysokoúrovňovými Petriho sítěmi, tzv. *neprimitivních funkcí*. Neprimitivní funkce `f(x1, x2, ..., xn)` je definována Petriho sítí, která má parametrová místa `x1, x2, ..., xn` a výstupní místo `return`. Značky v FPN reprezentují konstanty a v rámci provádění přechodů mohou být volány funkce, jako v případě HL-sítě.

Provedení přechodu s akcí `y := f(x1, x2, ..., xn)` závisí za implementaci funkce `f` takto:

[2] Problém je jednak v tom, že modely, založené na funkcionálním přístupu, nejsou příliš obvyklé, a také v tom, že funkce, definované Petriho sítěmi, jsou nedeterministické nejen ve způsobu vyčíslování, ale i ve výsledku.

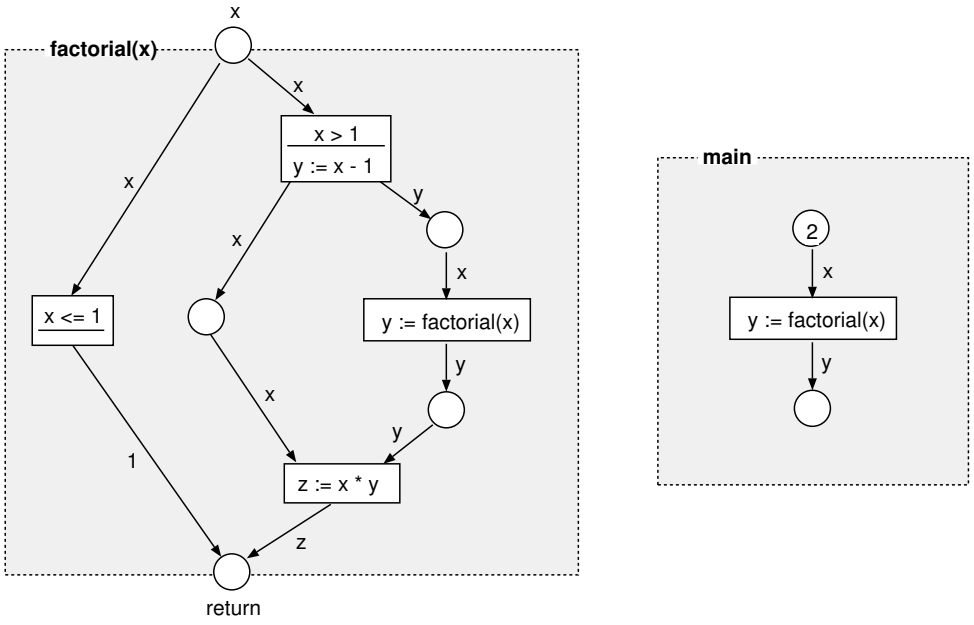
Vzhledem k tomu, že Petriho síť připouští množinu různých sekvencí provádění přechodů, reprezentuje množinu různých funkcí, přičemž je nedeterministicky vybrána jen jedna z nich.

Je třeba si uvědomit, že Petriho sítě zde vlastně nemodelují funkce, ale způsob vyčíslování, kde se už nedeterminismus může efektivně uplatnit.

- Je-li `f` primitivní funkce (definovaná inskripčním jazykem), přechod je proveden atomicky standardním způsobem, funkce je vyhodnocena jako součást provedení přechodu.
- Je-li `f` definována Petriho sítí, provedení přechodu proběhne ve třech krocích:

  1. Provede se vstupní část přechodu, tj. odeberou se příslušné značky ze vstupních míst, vytvoří se nová instance sítě, která funkci implementuje, a do jejích parametrových míst se umístí značky, reprezentující parametry funkce.
  2. Přechod čeká na dokončení provádění instance invokované sítě. Mezitím se mohou nezávisle provádět ostatní přechody, včetně těch, které jsou rozpracované (tj. i včetně tohoto právě popisovaného čekajícího přechodu).
  3. Přechod může ze stavu čekání vystoupit a dokončit se, pokud v instanci invokované sítě dojde provedením nějakého přechodu k umístění značky do místa `return`. Dokončení přechodu spočívá v tom, že se zruší instance invokované sítě, hodnota značky v místě `return` invokované sítě se přiřadí do proměnné `y` a provede se výstupní část přechodu, tj. do výstupních míst přechodu se umístí příslušné značky.

  Krok 1 i krok 3 proběhnou atomicky.



Obrázek 4.3: Příklad funkcionálně strukturované sítě – výpočet faktoriálu čísla 2.

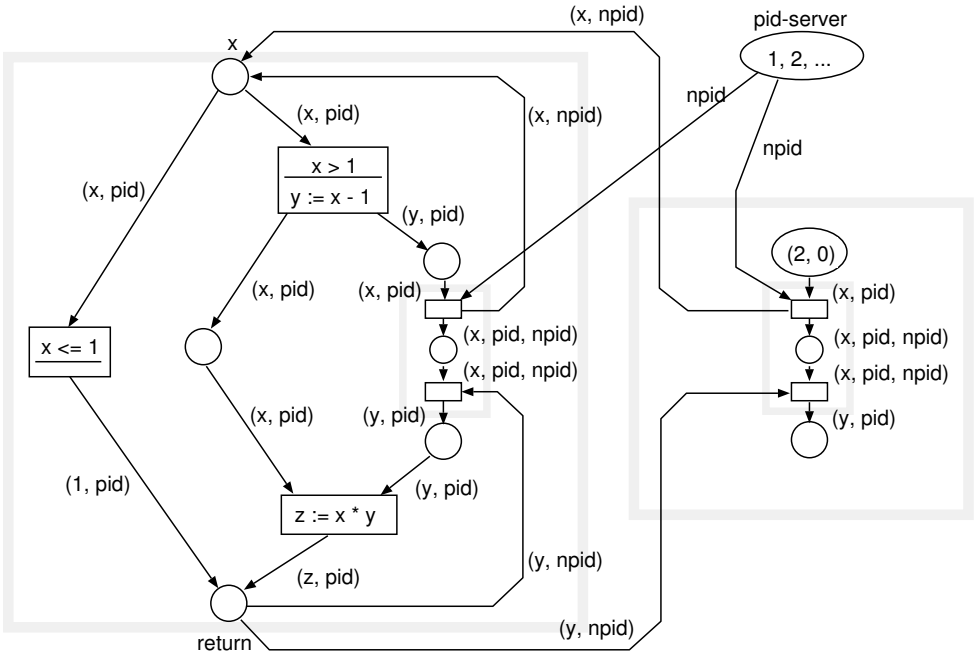
Příklad FPN na obr. 4.3 počítá rekurzivně 2!. Jsou zde dvě funkce, `factorial` a `main`, přičemž síť `main` je prohlášena za prvotní. Dynamika FPN začíná implicitní instanciací prvotní sítě.

4.1.3 Transformace do HL-sítě

Pro analytické zkoumání FPN může být zajímavá možnost transformace FPN do HL-sítě bez funkcionálního strukturování. Vzhledem k tomu, že HL-síť má výpočetní sílu Turingova stroje

[Jen92] a vzhledem k tomu, že chování FPN je specifiovatelné algoritmem, emulace FPN prostřednictvím HL-sítě je možná. Otázkou však zůstává, do jaké míry struktura příslušné HL-sítě odpovídá struktuře FPN, jinými slovy, jaká část FPN je vyjádřena inskripčním jazykem a jaká strukturou HL-sítě.

Za jistých omezujících podmínek lze pro FPN sestrojit HL-síť s ekvivalentním chováním, tj. takovou HL-síť, že provedení přechodu v FPN odpovídá provedení přechodu v HL-síti. Transformace je založena na rozšíření značek v invokovatelné síti o identifikace (čísla) procesů, které odpovídají jednotlivým invokacím sítě. Odpovídajícícm způsobem je třeba transformovat hranové výrazy.[3] Invokační přechod pak neprovádí dynamickou instanciaci invokované sítě, ale musí zajistit umístění takovýchto rozšířených značek do parametrových míst invokované sítě a do míst s neprázdným počátečním značením. Výstupní část invokačního přechodu je pak podmíněna výskytem příslušně identifikované značky ve výstupním místě invokované sítě. Každý přechod s neprimitivní akcí se tedy rozpadne na dva kauzálně svázané přechody. Navíc se ve výsledné HL-síti objeví místo s nepoužitými čísly procesů (`pid-server`). HL-síť na obr. 4.4 je výsledkem právě popsané transformace FPN z obr. 4.3.



Obrázek 4.4: HL-síť ekvivalentní FPN z obrázku 4.3.

V uvedeném případě byla transformace do HL-sítě jednoduchá, protože při výskytu značky v místě `return` je značení všech ostatních míst invokované sítě prázdné a není rozpracovaná žádná vnořená invokace. V případě, že by tomu tak nebylo, což musíme obecně připustit, lze ekvivalentní chování zajistit tím, že provádění každého přechodu podmíníme testem, zda číslo procesu, které je součástí zpracovávané značky, odpovídá momentálně existujícím procesům.[4] Je tedy třeba globálně evidovat čísla právě běžících (existujících) procesů. Problémem je však hromadění „mrtvých“ značek v místech. Počet takových značek není dopředu znám a lze je tedy odstraňovat jen postupně, dodatečnými přechody, připojenými ke každému místu v síti.

[3] Každé provedení přechodu se týká právě jednoho procesu. Přechod bez vstupních míst musíme před transformací doplnit o místo, obsahující jako počáteční značení libovolnou značku, a o testovací hranu, propojující ho s tímto místem, aby ve výsledné síti bylo možné zajistit jeho provádění pro určité číslo procesu.

[4] Neřešíme tím však problém vnořených invokací.

Provádění těchto přechodů je však asynchronní a nedeterministické, což komplikuje analýzu stavového prostoru modelu. Tomuto lze zabránit náhradou míst jinými strukturami[5] a úpravou přechodů tak, aby pracovaly s těmito strukturami, které umožní synchronní odstranění všech simulovaných značek při zániku procesu. Podobně by měl být řešitelný problém rušení vnořených invokací. Tím však zkomplikujeme výslednou HL-síť natolik, že analýza takové sítě již může být velmi problematická.

Toto jsou důvody, proč transformaci do HL-sítě uvádíme jen pro dokreslení neformálního výkladu uváděných strukturovacích mechanismů. Pro potřeby případné analýzy by bylo třeba definovat takovou transformaci do HL-sítě, která by byla pro analýzu použitelná, případně provádět analýzu bez nutnosti transformace do HL-sítě.

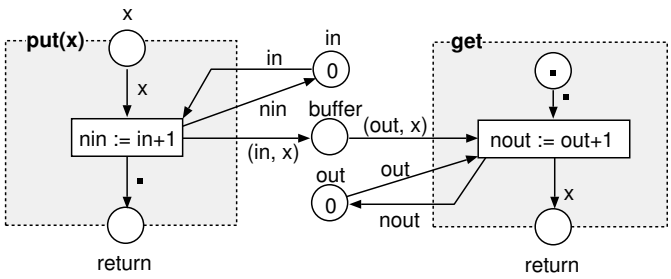
Závěrem lze k úvahám o FPN dodat, že podobně, jako některé jazyky umožňují předávat jména funkcí jako parametry, můžeme jména funkcí, resp. invokovatelných Petriho sítí, také používat jako hodnoty značek v Petriho síti. Transformace přechodu s akcí `eval(f, x1, ..., xn)` do HL-sítě s ekvivalentním chováním by byla podobná jako transformace polymorfního předání zprávy objektu, což bude diskutováno později.

4.2 Od funkcí k objektům

Funkcionální strukturování vneslo do Petriho sítí znovupoužitelnost a dynamiku. Samo o sobě je však obtížně použitelné, a proto je třeba ho doplnit o další prvky, které nyní popíšeme. Postupně se tak dostaneme k objektově orientovanému strukturování.

4.2.1 Funkce s vedlejším efektem

První modifikace FPN spočívá v tom, že připustíme vedlejší efekt funkcí. Přechody, patřící instanci funkce, mohou ovlivňovat obsah globálně dostupných míst, která mohou být sdílena instancemi všech funkcí. Na obr. 4.5 jsou dvě funkce, `put` a `get`, realizující vkládání a vybírání prvků z a do globálně existující fronty (bufferu) typu first-in/first-out.

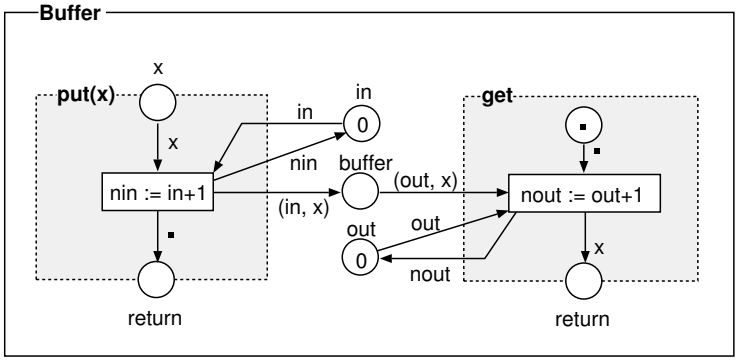


Obrázek 4.5: Příklad funkcí s vedlejším efektem.

4.2.2 Zapouzdření a instanciace objektů

Množinu funkcí a množinu míst, která jsou funkcemi sdílena, nyní nazveme *objektem* a umožníme dynamickou instanciaci takto vzniklého modelu. Tím jsme prakticky realizovali zapouzdření. Funce nazveme *metodami* a sdílená místa prohlásíme za *atributy* (datové složky) objektu. Struktura, obsahující množinu invokovatelných sítí a množinu sdílených míst, se tak stala *třídou* dynamicky instanciovatelných objektů. Na obr. 4.6 je příklad třídy `Buffer`.

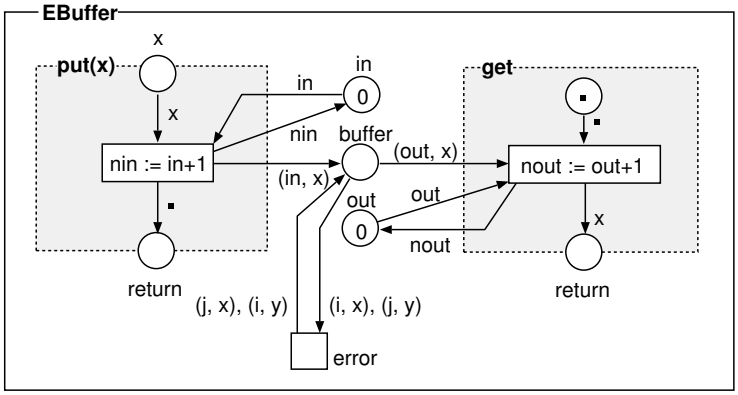
[5]Místo může například obsahovat jednu značku, reprezentující multimnožinu (simulovaných) značek.



Obrázek 4.6: Příklad třídy.

Třída umožňuje vytvářet svoje instance jako nezávislé objekty s vlastní identitou. Kdykoliv je objekt vytvořen, je mu dynamicky přidělena paměť pro atributy objektu. Kdykoliv objekt přijme zprávu, je invokována (dynamicky instanciována) síť odpovídající metody. Tato instance sítě metody má přístup k atributům objektu, v rámci kterého k invokaci došlo. Atributy jiných objektů (i jiných objektů téže třídy) jsou pro tuto instanci sítě metody nedostupné.

Vytvoření nového objektu se podobá invokaci sítě, ale nečeká se na ukončení existence nově vytvořené instance sítě. Toto je možné realizovat modifikovaným invokačním přechodem. Modifikace spočívá v tom, že tento přechod nečeká na ukončení procesu invokované sítě, ale spokojí se jen s jeho vytvořením. Novým prvkem oproti invokaci funkcí je skutečnost, že vytvořením nového objektu invokační přechod získá identifikaci (dynamicky přidělené jméno) tohoto objektu, kterou pak může uložit ve formě značky do svého výstupního místa. Je-li v síti k dispozici identifikace nějakého objektu, je možné s ním komunikovat zasíláním zpráv, jak bude dále ukázáno.



Obrázek 4.7: Buffer s poruchami pořadí.

4.2.3 Aktivní objekty

Doposud jsme uvažovali pasivní (reaktivní) objekty, které pouze reagují na přicházející zprávy. Zahrneme-li do reprezentace objektu (tj. k jeho atributům) i přechody a vytvoříme-li tak *síť objektu*, umožníme vnitřní samostatnou aktivitu objektů. Jde pak o *aktivní objekty*.

Kdykoliv je objekt vytvořen, vytvoří se instance sítě objektu. Obsahuje-li tato síť proveditelné přechody, může objekt sám od sebe vykonávat nějakou činnost. Když objekt přijme zprávu, je invokována síť odpovídající metody. Ta může přistupovat k místům sítě objektu, měnit jejich obsah a tím nepřímo ovlivňovat chování aktivního objektu.

Na obr. 4.7 je příklad třídy `EBuffer`, specifikující nezávislou aktivitu svých objektů, která simuluje poruchy v pořadí prvků ve frontě. Přechod `error` náhodně vybere dvě značky z místa `buffer`, zamění jim pozice ve frontě (první složky dvojic) a umístí je zpět do místa `buffer`.

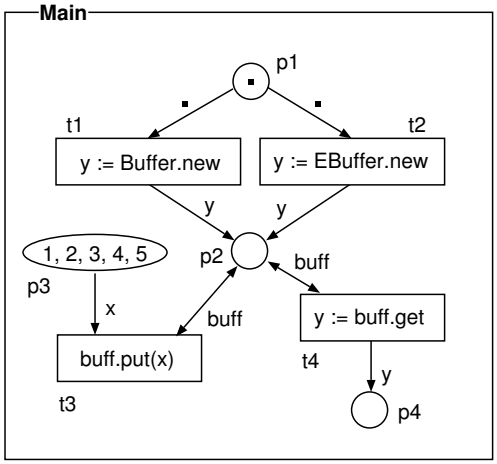
4.2.4 Předávání zpráv a polymorfismus

Objekty komunikují předáváním zpráv. Prakticky se předávání zpráv podobá volání funkcí, přičemž povinným argumentem takové funkce je identifikace adresáta zprávy. Toto se ve většině objektově orientovaných jazyků syntakticky vyjadřuje tečkovou notací, tj. povinný argument (adresát zprávy) předchází jménu operace (selektoru zprávy) a je od něho oddělen tečkou. Následují ostatní argumenty. Podobným způsobem tedy modifikujeme akci přechodu, která nyní bude mít tvar

$$y := o.msg(a_1, ..., a_n),$$

kde o je adresát zprávy msg s argumenty $a_1, ..., a_n$. Vytváření instancí tříd budeme specifikovat zasláním zprávy `new` příslušné třídě.

Zpráva téhož jména může být potenciálně poslána objektům různých tříd a v rámci těchto tříd mohou být pro tutéž zprávu definovány různé metody. To znamená, že zasílání zpráv je *polymorfní*. Síť, která se invokuje (síť, jejíž instance se vytváří) se určí dynamicky, za běhu programu, a závisí na třídě adresáta zprávy.

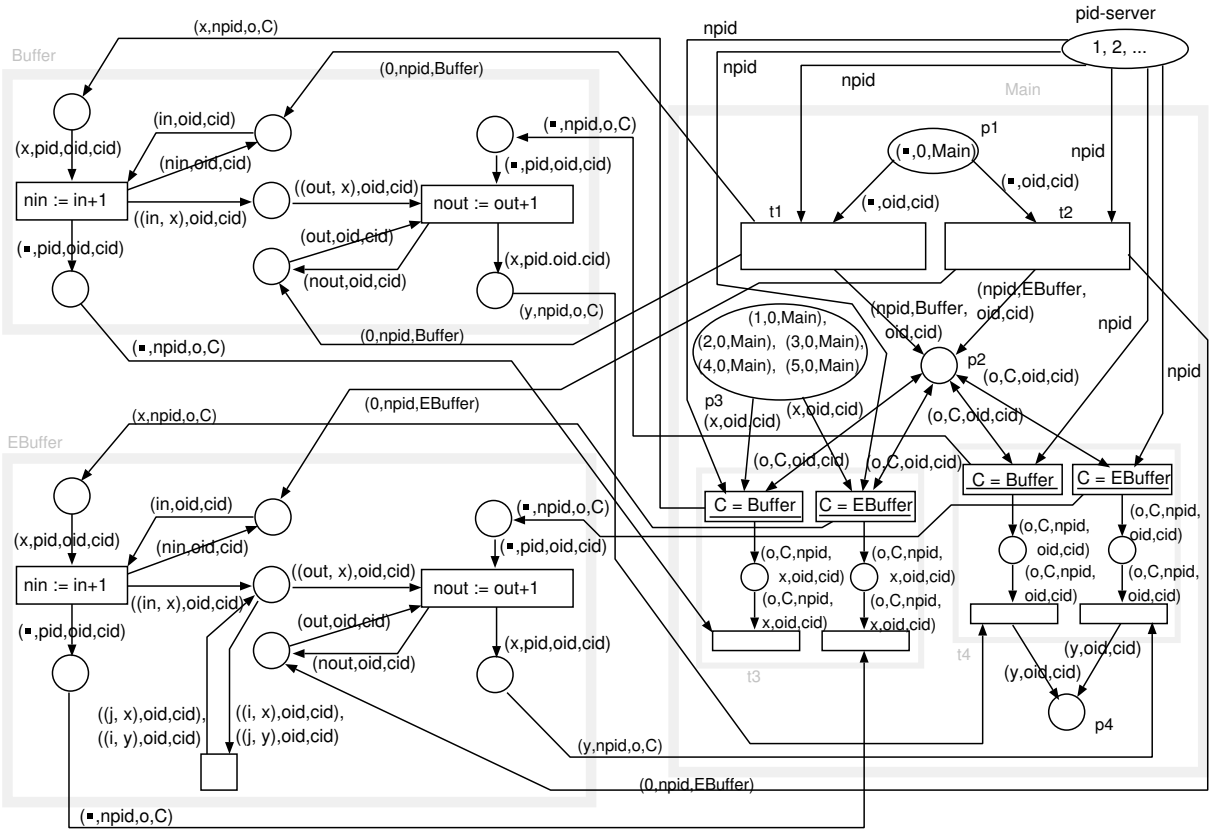


Obrázek 4.8: Demonstrace instanciace tříd a polymorfního zasílání zpráv.

Na obr. 4.8 je uvedena třída `Main`, která nedefinuje žádnou metodu, ale pouze nezávislou aktivitu, která nedeterministicky (viz konfliktní přechody `t1` a `t2`) vytvoří buď objekt třídy `Buffer`, nebo objekt třídy `EBuffer`. S vytvořeným objektem pak zbývající přechody (`t3` a `t4`) komunikují. Vzhledem k tomu, že adresát zprávy (a třída adresáta) se zjistí až za běhu, předávání zpráv `put` a `get` je polymorfní.

V třídě `Main` na obr. 4.8 jsou oboustrannými šipkami (mezi `t3` a `p2` a mezi `t4` a `p2`) specifikovány tzv. *testovací hrany*. Jejich sémantika je v případě atomického provedení přechodu ekvivalentní dvojici opačně orientovaných hran, ohodnocených stejným hranovým výrazem jako testovací hrana. V případě neatomického provedení přechodu, tj. když je přechodem invokována síť metody, vstupní hrana přechodu odebere ze vstupního místa potřebné značky a výstupní hrany se uplatní až po ukončení invokace. Testovací hrana však stále ponechává testované značky v příslušném místě. V případě neatomického provedení přechodu tedy není testovací hrana ekvivalentní dvojici opačně orientovaných hran.

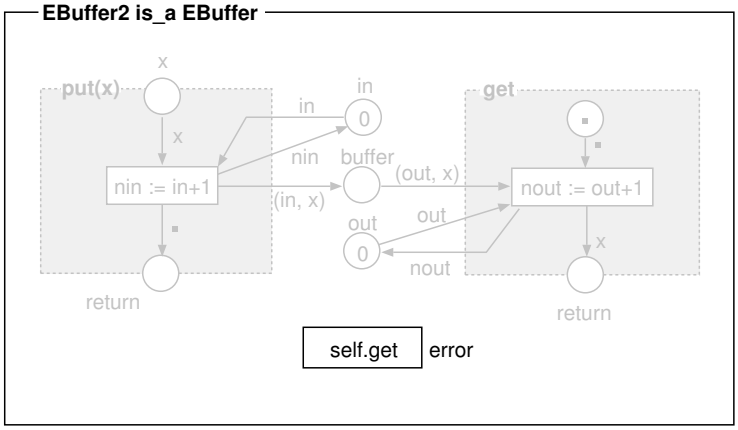
Transformace uvedených konstrukcí do ekvivalentní HL-sítě se provede analogicky k FPN, ale jsou zde některé odlišnosti. Značky v síti objektu jsou rozšířeny o číslo procesu a identifikaci třídy, značky v síti metody jsou rozšířeny o číslo procesu, číslo procesu sítě objektu a identifikaci třídy. Tomu odpovídají i hranové výrazy. Hodnota značky, reprezentující referenci na objekt, je dvojice (o, C), kde o je číslo procesu sítě objektu a C je jméno třídy. Vytvoření nového objektu spočívá v umístění rozšířených značek, odpovídajících počátečnímu značení sítě objektu. Polymorfní předání zprávy se realizuje podobně jako invokace sítě, ale je třeba invokační přechod transformovat pro všechny varianty, tj. pro všechny implementace metody pro příslušnou zprávu ve všech třídách. Každá varianta má ve stráži přechodu test na příslušnou třídu. HL-síť, ekvivalentní objektově orientované síti, sestávající z tříd `Buffer`, `EBuffer` a `Main` z obrázků 4.6, 4.7 a 4.8 je na obr. 4.9. Tato síť je poměrně jednoduchá (podobně, jako v případě FPN, obr. 4.4), protože nebylo třeba řešit problémy, spojené se zánikem objektů a ukončováním metod s nedokončeným vnořeným voláním.



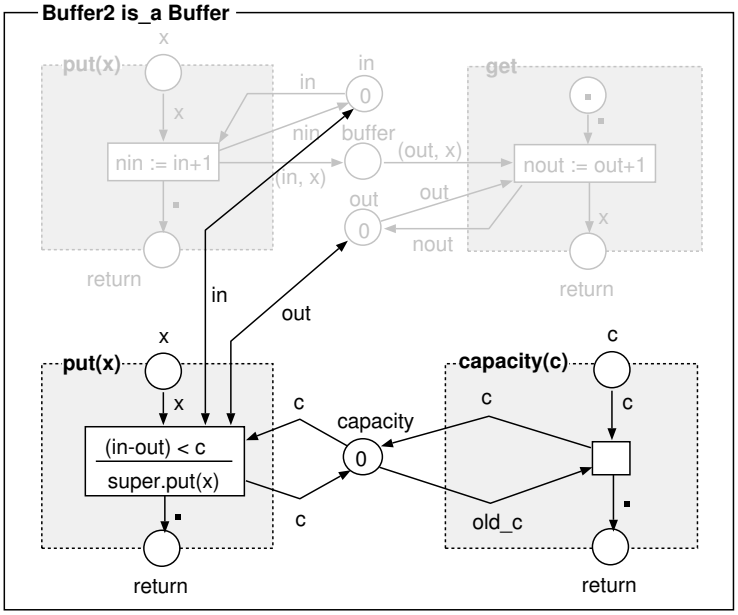
Obrázek 4.9: HL-síť, ekvivalentní třídám `Buffer`, `EBuffer` a `Main`.

4.2.5 Dědičnost

Dalším krokem k objektové orientaci v Petriho sítích je definování dědičnosti, která umožní inkrementální specifikaci tříd. Nová třída se pak definuje jako podtřída jiné třídy, od které dědí, a specifikují se pouze rozdíly.



Obrázek 4.10: Buffer s náhodně se ztrácejícími prvky.



Obrázek 4.11: Buffer s omezenou kapacitou.

Dědičnost metod může být realizována klasickým způsobem (tj. jako v jazyce Smalltalk-80), bez jakýchkoliv modifikací. Metoda pro danou zprávu může být buď přidána, nebo zděděná metoda může být zcela nahrazena novou definicí.

Dědičnost reprezentace objektu může být realizována mechanismem dědění struktury sítě objektu. Ke zděděné síti mohu být přidány nové uzly (místa a přechody) a zděděné uzly mohou

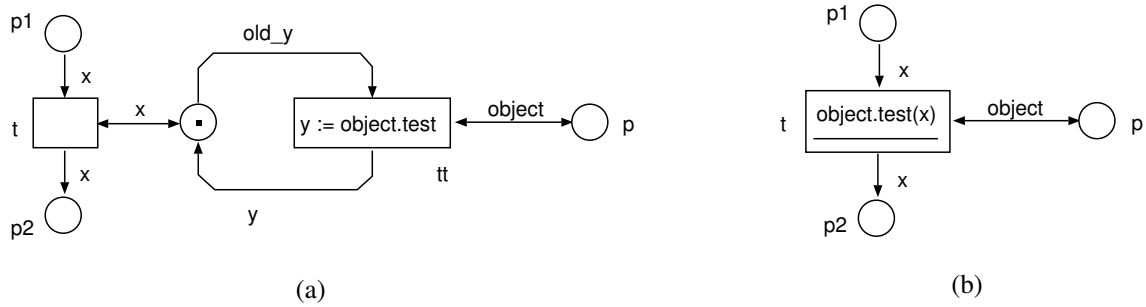
být předefinovávány. Vstupní a výstupní hrany přechodů považujeme za součást přechodů, takže novou definicí přechodu se nově definují všechny jeho okolní hrany. Uzly sítí jsou identifikovány jmény. Je-li v síti uveden uzel stejného jména jako uzel sítě, jejíž strukturu nově definovaná síť dědí, znamená to, že je tímto předefinován.

Třída `EBuffer2` na obr. 4.10 dědí od třídy `EBuffer` a předefinovává přechod `error` (původní verze je na obr. 4.7) tak, že nedeterministicky[6] odebírá prvky ze začátku fronty voláním `self.get`.[7] Poznamenejme, že předefinováním přechodu došlo i k předefinování jeho vstupů a výstupů (v našem případě přechod `error` nemá žádné vstupy ani výstupy). Zděděné prvky třídy jsou zobrazeny šedě, protože nejsou touto třídou definovány. V grafické reprezentaci třídy je (nepovinně) uvádíme jen kvůli přehlednosti.

Třída `Buffer2` na obr. 4.11 dědí od třídy `Buffer` (třída `Buffer` je na obr. 4.6), přidává místo `capacity` a metodu `capacity` a předefinovává metodu `put`. Původní verze metody `put` je dostupná, stejně jako v jazyce Smalltalk-80, pouze voláním `super.put(x)`.[8] Místo `capacity` slouží kromě uložení hodnoty kapacity také jako semafor, zajišťující, že v každém okamžiku nejvýše jedna invokace metody `put` může pracovat s obsahem fronty.

4.2.6 Atomická synchronní komunikace

Uvedené konstrukce již umožní plně realizovat objektově orientované strukturování Petriho sítí. Pro usnadnění modelování synchronních interakcí však lze doplnit některé další konstrukce.



Obrázek 4.12: Příklad testování stavu objektu v místě `p`: (a) – aktivní, (b) – voláním predikátu.

Predikáty

Prvním kritickým případem je podmíněné provedení přechodu v závislosti na stavu jistého objektu. Vzhledem k tomu, že předávání zpráv jsme prozatím umožnili jen v akci a nikoliv ve stráži[9] přechodu, testování stavu objektu a následné rozhodnutí o provedení přechodu se modeluje zbytečně komplikovaně. V příkladu na obr. 4.12a přechod `tt` aktivně testuje stav objektu v místě `p` opakovaným zasíláním zprávy `test` tomuto objektu. Vyhovuje-li stav objektu (zjištěný

[6] Může se provést kdykoliv, protože nemá vstupní hrany.

[7] Volámím `self` objekt posílá zprávu sám sobě.

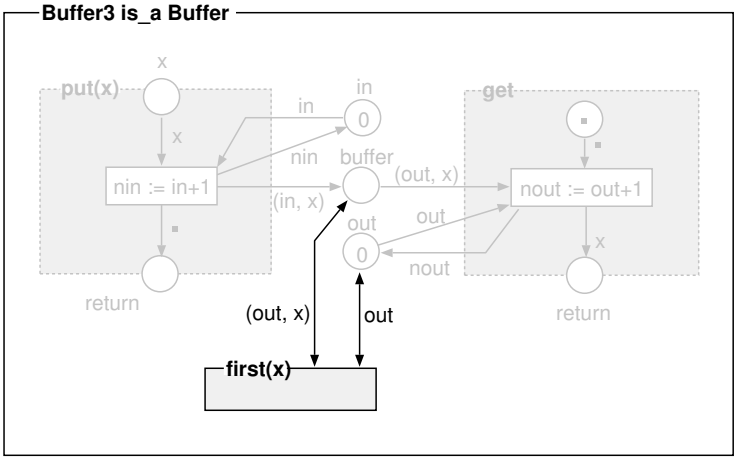
[8] Sémantika volání `super` je převzata z jazyka Smalltalk-80 [GR83]: Je-li adresátem zprávy `super`, je invokována metoda, která je definována nebo zděděna (bez předefinování) v nejbližší nadtřídě třídy, v níž se volání `super` vyskytuje. Volá se však v kontextu objektu, identifikovaného jako `self`. Volání `super` umožňuje využít původní implementaci metody v rámci nově definovaných metod.

[9] Testování stráže musí být bez vedlejšího efektu, protože z hlediska Petriho sítě je (musí být) lhostejné, kdy a jak (případně kolikrát) se testování stráže provede.

posledním vyhodnocením zprávy `test`) podmínkám přechodu `t`, může být tento přechod proveden. Aby se mezitím stav testovaného objektu nezměnil, bylo by nutné navíc implementovat a volat metody pro uzamknutí a odemknutí objektu, aby bylo zajištěno jeho korektní chování.

Připustíme-li existenci atomicky proveditelných metod, které pouze testují stav objektu a nemění ho, můžeme takovéto metody volat i ze stráží přechodů (viz obr. 4.12b). Tyto speciální metody nazveme *predikáty* (*predikátové metody*). Predikátová metoda má tvar virtuálního přechodu, obsahujícího pouze testovací hrany, který je určen k dynamické fúzi s volajícím přechodem při vyhodnocování predikátové zprávy. Sám o sobě nemůže být nikdy proveden. Predikát má přístup k místům sítě objektu, ovšem pouze nedestruktivní, testovacími hranami.[10] V případě, že predikát voláme s nenavázaným parametrem (s volnou proměnnou), predikát může provést navázání parametru na určitou hodnotu. Jelikož testování stavu objektu predikátem proběhne atomicky a současně s rozhodnutím o provedení přechodu, který predikátovou metodu volá, není třeba pro testovaný objekt implementovat metody pro uzamykání. Příklad predikátu je na na obr. 4.13. Třída `Buffer3` zde dědí od třídy `Buffer` a přidává predikátovou metodu `first`.

Poznamenejme, že objekty, komunikující voláním predikátových metod, jsou implementačně těsněji vázané, než objekty, komunikující invokací sítí metod. Díky komplikovanější praktické realizaci je proto synchronní komunikace určena především pro modelování na vyšší úrovni abstrakce, kde otázka konkrétní realizace modelovaného systému ustupuje do pozadí.



Obrázek 4.13: Predikát.

Synchronní porty

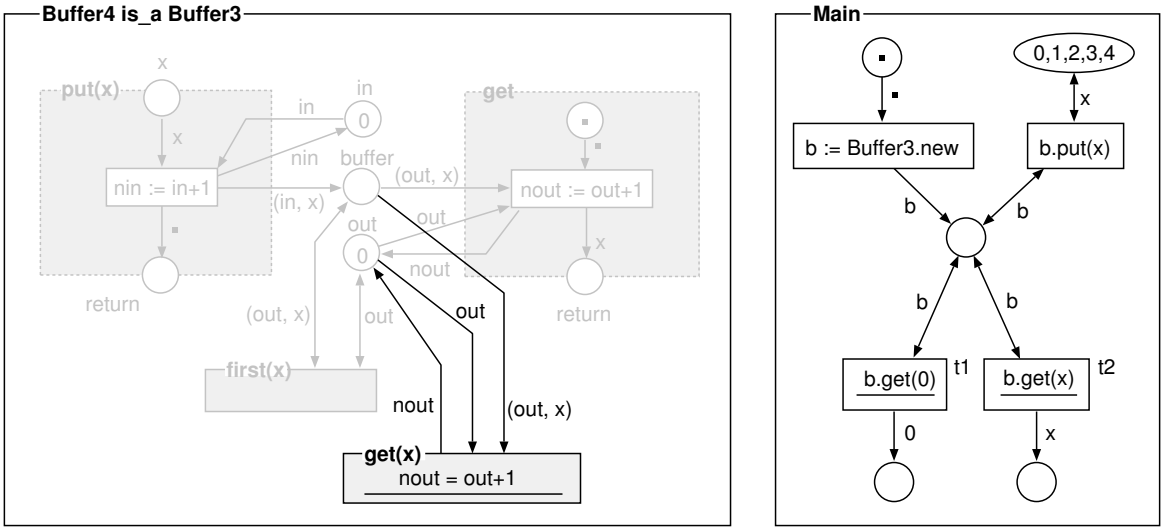
V některých případech je vhodné synchronizovat provádění přechodů v různých instancích sítí.[11] To lze ve staticky strukturovaných sítích řešit fúzí přechodů, resp. synchronními kanály, jak bylo uvedeno v kapitole 2. V případě objektově orientovaného strukturování Petriho sítí je však třeba zachovat princip komunikace objektů předáváním zpráv. Proto musíme uvedené mechanismy poněkud upravit.

Uvedený problém lze řešit modifikací predikátu, která připouští vedlejší efekt, tj. může ovlivňovat obsah míst sítě objektu. Pak nemluvíme o predikátu, ale o *synchronním portu*. Při testování stráže přechodu, odkud se synchronní port volá, se stav systému nemění a uplatní se

[10] Poznamenejme, že díky atomičnosti vyhodnocování predikátu je v tomto případě testovací hrana ekvivalentní dvojici hran, vstupní a výstupní, ohodnocených stejným hranovým výrazem.

[11] Jde o některé případy modelování víceúrovňové aktivity, zmíněné v kapitole 2.

jen testování proveditelnosti synchronního portu (jako by to byl přechod). Pokud se přechod, který ze své stráže volal synchronní port, provede, provede se i synchronní port (jako by to byl přechod).



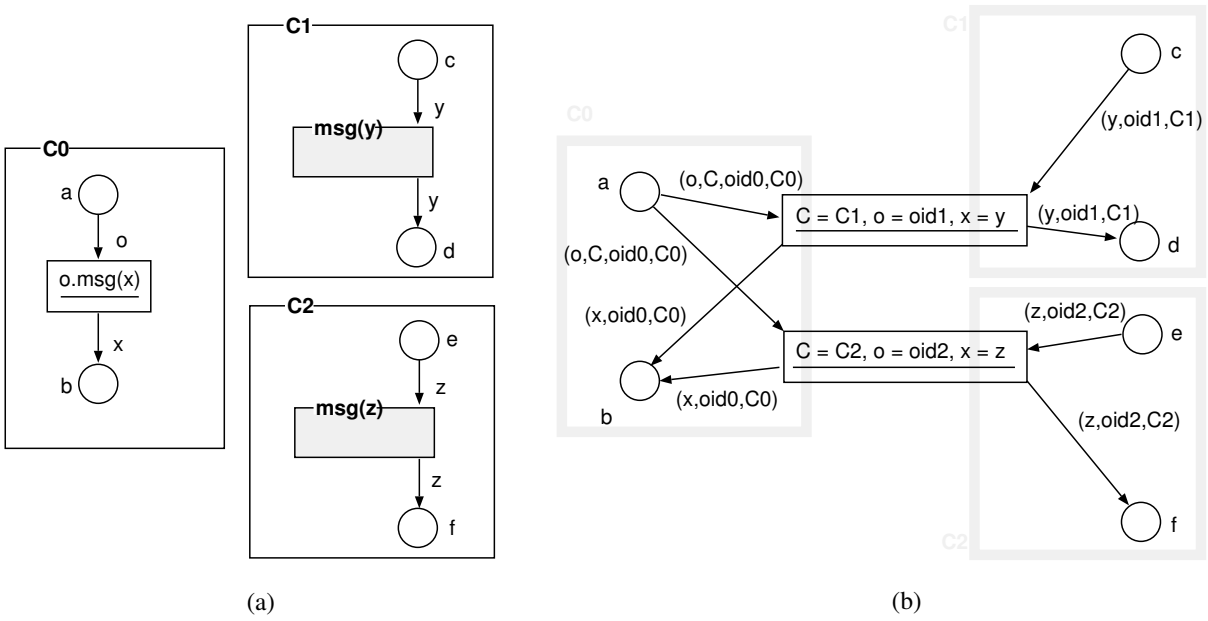
Obrázek 4.14: Synchronní port a jeho použití.

Příklad synchronního portu je na obr. 4.14. Třída `Buffer4` dědí od třídy `Buffer3` a přidává synchronní port `get(x)`. Synchronní port může obsahovat stráž (jako na obr. 4.14), na rozdíl od přechodu však nemůže mít definovanou akci.[12] V pravé části obrázku 4.14 je příklad volání synchronního portu. Přechod `t1` je proveditelný jen tehdy, je-li prvním prvkem ve frontě číslo `0`. Provedením přechodu `t1` se tento prvek z fronty vyjme. Přechod `t2` je proveditelný, lze-li z fronty cokoliv vyjmout. Jeho provedením se tak stane. Z hlediska aplikace synchronních portů je třeba zdůraznit, že jde, stejně jako v případě predikátů, o mechanismus, určený především k usnadnění tvorby abstraktních modelů, ve kterých otázka implementace modelovaného systému není významná.[13]

Volání synchronního portu může být opět polymorfní. Obr. 4.15 ukazuje objektově orientovanou síť, obsahující fragmenty tříd `C0`, `C1` a `C2`, a ekvivalentní HL-síť. Vzhledem k tomu, že synchronní port `msg` je jednou volán a je implementován ve dvou třídách, v ekvivalentní HL-síti se objeví dva přechody, modelující dvě varianty volání synchronního portu. Hranové výrazy jsou modifikovány tak, jak bylo uvedeno v ostavci 4.2.4. Proměnná `o`, reprezentující objekt, se kterým se komunikuje, je transformována na dvojici `(o, C)`, kde `o` je jméno objektu a `C` je jeho třída. Poslední dvě složky každé n-tice v každém hranovém výrazu tvoří proměnné, reprezentující jméno objektu a třídu objektu, který je instancí třídy, jejíž síť objektu transformujeme do HL-sítě (podrobněji viz 4.2.4). Přechodu, volajícímu synchronní port `msg` v ekvivalentní HL-síti odpovídají dva přechody, které vznikly fúzí volajícího přechodu a volaného synchronního portu pro každou variantu, která díky polymorfismu přichází v úvahu. Každý z těchto přechodů ve své stráži unifikuje odpovídající si proměnné, reprezentující třídu, jméno adresáta zprávy a parametry synchronního portu, používané ve třídě odesilatele a ve třídě adresáta synchronní zprávy. V obecnějším případě, když stráž přechodu specifikuje volání dvou a více, obecně n syn-

[12] Synchronní port musí být proveditelný atomicky, akce přechodu však připouští i neatomické provedení.

[13] Synchronní interakce se implementuje komplikovaněji než protokol klient-server, kde vystačíme s asynchronním zasláním požadavku a asynchronním předáním odpovědi.



Obrázek 4.15: Polymorfní volání synchronního portu (a) a ekvivalentní HL-síť (b).

chronních portů, je samozřejmě každá varianta provedení reprezentována v ekvivalentní HL-síti přechodem, který vznikne fúzí volajícího přechodu s n synchronními porty.

Závěrem poznamenejme, že jsme pro jednoduchost neuvažovali možnost volání synchronních portů ze stráží jiných synchronních portů. K této problematice se ještě vrátíme v kapitolách 5 a 7.

4.3 Modelování objektů Petriho sítěmi

Ukázali jsme, že objektová orientace může být do Petriho sítí zavedena prostřednictvím dvou mechanismů, umožňujících vzájemnou interakci Petriho sítí. Prvním mechanismem je modifikovaný koncept invokace. Původní koncept invokace, který umožňoval dynamicky vytvářet instance Petriho sítí, byl rozšířen o polymorfismus v tom smyslu, že síť, která se má invokovat, je určena až v době běhu a je dána třídou adresáta zprávy. Druhý použitý interakční mechanismus je komunikace instancí sítí prostřednictvím sdílených míst, patřících jedné z komunikujících instancí sítí. Každá instance sítě metody může sdílet místa, patřící odpovídající instanci sítě objektu. Tímto mechanismem je umožněna modifikace stavu objektu prováděním metod.

Tyto dva základní koncepty lze pro usnadnění specifikace víceúrovňových modelů obohatit o možnost synchronní komunikace, řešené také předáváním zpráv, tentokrát ale specifikovaným ve strážích přechodů.

Nyní představíme objektově orientovanou Petriho síť, vybudovanou na základě výše zmíněných konceptů, jako implementaci objektového modelu, který jsme definovali v kapitole 3. Následující text lze chápat jako neformální úvod do objektově orientovaných Petriho sítí, přičemž formální definice bude uvedena v další kapitole.

4.3.1 Struktura OOPN

Model systému, ve kterém mohou dynamicky vznikat a zanikat objekty, komunikující předáváním zpráv, je specifikován množinou tříd těchto objektů. *Objektově orientovaná Petriho síť* (OOPN), jako model takového systému, je množina tříd, hierarchicky organizovaná podle vztahu dědičnosti. Jedna z tříd je prohlášena za počáteční třídu (ta je implicitně instanciována při spuštění simulace). Každá třída obsahuje:

- síť objektu, definující reprezentaci instancí třídy (atributy) a jejich nezávislou aktivitu,
- množinu sítí metod, definujících reakce na zprávy, zasílané z akcí přechodů,
- množinu synchronních portů, definujících reakce na synchronní zprávy, zasílané ze stráží přechodů.

Sítě metod mohou v rámci třídy sdílet místa sítě objektu. Mají navíc oproti síti objektu parametrová místa, do kterých se při invokaci uloží parametry zprávy, a výstupní místo `return`, ve kterém se očekává návratová hodnota – výsledek vyhodnocení zprávy.

Každá třída je definována inkrementálně vůči své nadtřídě – specifikují se metody, kterými se liší od své nadtřídy, a uzly sítě objektu, kterými se liší od sítě objektu své nadtřídy.

Značky v sítích reprezentují reference na objekty (nikoliv objekty samotné). Přitom ne všechny objekty jsou popsány Petriho sítěmi. Rozlišujeme tři druhy objektů:

- Některé objekty jsou instancemi primitivních tříd. Jsou to konstanty jako čísla, znaky, řetězce, symboly a seznamy. Jejich metody jsou funkce bez vlivu na stav těchto objektů. Tyto objekty budeme považovat za *primitivní*.
- Jiné objekty jsou *třídy*. Tyto objekty se po dobu běhu modelu nemění. Rozumějí zprávě `new` a jsou schopny vytvářet své instance.
- Ostatní objekty jsou instancemi tříd, definovaných Petriho sítěmi. Takové objekty považujeme za *neprimitivní*.

4.3.2 Dynamika OOPN

Dynamika OOPN vychází z chování HL-sítí a spočívá v postupném vývoji stavu systému prováděním přechodů.

Proveditelnost přechodu

Proveditelnost přechodu v instanci sítě v OOPN je definována podobně jako pro HL-síť, ale je zde modifikováno vyhodnocování stráže přechodu.

Stráž přechodu obsahuje sekvenci výrazů, z nichž každý je zaslání zprávy. Tato sekvence má význam konjunkce, tj. stráž má hodnotu `true`, pokud každé dílčí zaslání zprávy vrací `true`. Dílčí podvýrazy (zasílání zpráv) ve stráži přechodu se vyhodnocují takto:

- V případě, že adresátem zprávy je primitivní objekt, výraz se vyhodnotí v rámci inskripčního jazyka.
- Jinak jde o synchronní komunikaci s neprimitivním objektem. V tomto případě má podvýraz stráže hodnotu `true`, je-li odpovídající synchronní port adresáta zprávy proveditelný (pro určité navázání proměnných) a není v konfliktu s volajícím přechodem ani s ostatními synchronními porty, zúčastněnými na synchronní komunikaci. Jinak má hodnotu `false`.

Provedení přechodu

Provedení přechodu v instanci sítě v OOPN je podobné provedení přechodu v HL-síti, ale realizuje se, podobně jako v případě FPN, buď atomicky, nebo neatomicky, přičemž rozhodnutí o způsobu provedení se realizuje za běhu.

Akce přechodu specifikuje zaslání zprávy. Bez ohledu na konkrétní syntax akci přechodu pro teoretické úvahy symbolicky označíme $y := x_0.msg(x_1, ..., x_n)$, kde y je proměnná, x_i je term (literál nebo proměnná) a msg je selektor zprávy. Objekt, identifikovaný jako x_0, je adresát zprávy a objekty x_i jsou její argumenty. Způsob provedení přechodu závisí na třídě adresáta zprávy, tj. objektu x_0:

- Je-li x_0 primitivní objekt, přechod se i s vyhodnocením akce provede atomicky, stejně jako v klasické Petriho síti. Která metoda (v tomto případě funkce) se skutečně provede, závisí na třídě objektu x_0.
- Je-li x_0 třída a zpráva je `new`, provedení přechodu proběhne atomicky s tím, že dojde k vyvoření nové instance třídy x_0. Proměnná y přitom obsahuje identifikaci nového objektu.
- Je-li x_0 neprimitivní objekt, provedení přechodu proběhne neatomicky, a sice takto:
  - Provede se vstupní část přechodu, tj. odeberou se příslušné značky ze vstupních míst. Podle třídy objektu x_0 se nalezne odpovídající metoda pro selektor zprávy msg, vytvoří se nová instance sítě metody a do jejích parametrových míst se umístí značky, reprezentující parametry zprávy.
  - Přechod čeká na dokončení provádění metody. Mezitím se mohou nezávisle provádět ostatní přechody, včetně těch, které jsou rozpracované (tj. i včetně tohoto právě popisovaného čekajícího přechodu).

    Vzhledem k tomu, že přechod ve stavu čekání nese informaci o invokované síti a navázání proměnných, zavádíme pojem *značení přechodu*.
  - Přechod může ze stavu čekání vystoupit a dokončit se, pokud v instanci invokované sítě dojde provedením nějakého přechodu k umístění značky do místa `return`. Dokončení přechodu spočívá v tom, že se zruší instance invokované sítě,[14] objekt, reprezentovaný značkou v místě `return` invokované sítě, se přiřadí do proměnné y a provede se výstupní část přechodu, tj. do výstupních míst přechodu se umístí příslušné značky.

Implicitní součástí provedení přechodu je provedení synchronní komunikace, byla-li specifikována ve stráži příslušného přechodu. Ta spočívá v provedení odpovídajícího synchronního portu (jako by to byl přechod) v síti objektu adresáta zprávy.

Evoluce stavu systému

Stav systému, popsaného objektově orientovanou Petriho sítí, je určen okamžitými stavy všech momentálně existujících objektů, přičemž stav každého objektu je určen stavem jeho sítě objektu a stavy instancí všech invokovaných (a ještě nedokončených) sítí metod (včetně násobných invokací téže sítě).

Evoluce systému začíná implicitním vytvořením prvotního objektu jako instance prvotní třídy. Je tedy vytvořena instance odpovídající sítě objektu. Provádění přechodů postupně modifikuje stav systému. Lze identifikovat čtyři typy změn stavu systému, tedy čtyři typy událostí:

[14] Včetně všech nedokončených vnořených invokací.

- klasické provedení přechodu uvnitř jedné sítě (nebo synchronně ve více sítích současně, jsou-li volány synchronní porty),
- vytvoření nového objektu (vytvořeni instance příslušné sítě objektu),
- invokace metody (vytvoření instance sítě metody),
- ukončení provádění metody s předáním výsledku (odstranění instance sítě sítě metody ze systému).

Jako součást událostí prvních tří typů může dojít k synchronní komunikaci, protože se zde uplatňuje stráž přechodu, která může volat synchronní porty.

Pokud výskyt události způsobí, že některý objekt už není tranzitivně referencován z počátečního objektu, tento objekt je automaticky odstraněn ze systému. Tato forma garbage-collectingu se provádí implicitně v rámci každé události.

V příloze B je demonstrována dynamika OOPN na konkrétním modelu, specifikovaném v jazyce PNtalk (jazyk PNtalk je definován v kap. 6).

4.3.3 Shrnutí

Výsledná OOPN má všechny atributy objektově orientovaného jazyka, tj. umožňuje zapouzdření, polymorfismus a dědičnost. Navíc poskytuje výkonné prostředky pro popis paralelismu. Všechny uvedené skutečnosti budou detailně specifikovány formální definicí OOPN v kapitole 5.

Je zřejmé, že funkcionální Petriho síť je speciálním případem OOPN: je to jeden objekt s prázdnou sítí objektu a s metodami bez vedlejšího efeku. Podobně také ostatní variany Petriho sítí můžeme považovat za speciální případy OOPN.

4.4 Jiné přístupy k objektové orientaci v Petriho sítích

Zkoumání souvislosti Petriho sítí a objektů se již v současné době věnuje řada vědeckých pracovišť ve světě. Tento výzkum v některých případech vyústil ve formalismy, které jsou srovnatelné s přístupem, prezentovaným v této práci. Na základě výše uvedeného neformálního úvodu do OOPN jsme již schopni tento přístup konfrontovat s ostatními. V následujících odstavcích stručně popíšeme a zhodnotíme alternativní přístupy k objektové orientaci v Petriho sítích.

4.4.1 DesignBeta

Výzkumníci z University of Aarhus, kteří vyvinuli CPN a nástroj Design/CPN, také experimentovali s integrací objektově orientovaného jazyka BETA do Design/CPN [CT93]. Jazyk BETA je zde použit k deklaraci typů značek a pro specifikaci kódových segmentů, připojených k přechodům. Objekty jsou zde tedy spojeny se značkami v CPN. CPN samotná však zůstává globální strukturou, vymykající se objektově orientovanému přístupu. Jde vlastně jen o drobnou modifikaci inskripčního jazyka CPN.

4.4.2 HOOD/PNO a HOOD Nets

Zajímavým projektem je HOOD/PNO a jeho varianta HOOD Nets [Gio91], který kombinuje Petriho sítě s návrhovou metodou HOOD (Hierarchical Object Oriented Design). Petriho síť je zde použita ke specifikaci chování objektu a pro meziobjektovou komunikaci. Pro hierarchický objekt je pak možné zkomponovat síť, definující jeho chování. Smyslem tohoto počínání je umožnit analýzu hierarchických objektů. Jistou modifikací HOOD/PNO jsou HOOD Nets [Gio91],

kde je každá metoda specifikována samostatnou sítí. V tomto ohledu se přístup HOOD Nets podobá OOPN. Metoda HOOD však byla vyvinuta primárně pro tvorbu paralelních programů v jazyce Ada a pod pojmem objekt rozumí abstraktní datový typ. Není zde Petriho sítěmi řešena otázka polymorfismu, dědičnosti, ani dynamické instanciace sítí objektů.

4.4.3 Petriho sítě jako součást algebraických specifikací

Existuje řada projektů, které obohacují algebraické specifikace abstraktních datových typů o paralelismus a synchronizaci pomocí Petriho sítí. Patří sem mimo jiné OBJSA nets [BCMR91], CLOWN [BCC95], CO-OPN [BG91] a revidovaná verze CO-OPN/2 [BB95], doplňující dědičnost. V těchto formalismech Petriho síť v rámci definice třídy specifikuje kauzalitu volání operací objektů. Kooperace objektů je modelována statickým propojením sítí zúčastněných objektů na základě fúze přechodů. Prostředí SANDS pro specifikaci tříd pomocí CO-OPN dovoluje generovat proveditelný kód v PROLOGu. Tyto formalismy se vesměs zabývají kooperací staticky propojených objektů a neřeší problematiku modelování víceúrovňové aktivity.

4.4.4 Elementary Object Nets

R. Valk [Val98] studuje sítě s dvěma úrovněni aktivity – po systémové síti se pohybují značky, spojené s objektovými sítěmi, což jsou klasické (low-level) Petriho sítě. Objektová síť se synchronizuje se systémovou sítí společným provedením přechodu, pojmenovaným stejně, jako přechod systémové sítě, který odpovídající značku zpracovává. V systémové síti provedením přechodu nedochází k rušení značek a generování nových, ale přemisťují se při zachování jejich identity. Zobecněná varianta, která připouští více úrovní objektů [Val95] má velmi blízko k objektové orientaci. Není to však plně objektově orientovaný formalismus, protože neřeší dynamickou instanciaci, polymorfismys a dědičnost, ale zabývá se problematikou, která je pro objektovou orientaci zajímavá, a sice synchronizací aktivit v různých úrovních modelu.

4.4.5 PN-TOX

PN-TOX [Hol95b] není jazyk ani formalismus, ale poměrně složité vývojové prostředí pro paralelní objektově orientované programování Petriho sítěmi. Objekt zde zapouzdřuje Petriho síť. PN-TOX podporuje tvorbu multimodelů tím že je schopen použít různé druhy sítí, od C/E sítí, přes P/T sítě k high-level sítím, jako jsou CPN a Pr/T sítě. Ke značkám v těchto sítích mohou být (dodatečnými prostředky) připojeny objekty, což dovoluje modelovat víceúrovňovou aktivitu. Podobně mohou být k přechodům připojeny bloky kódu, který se provádí v rámci provádění přechodů. Rozhraní objektu je k síti připojeno tak, že vyvolání operace objektu způsobí předání značky do určitého místa zapouzdřené sítě. PN-TOX umožňuje také statickou kompozici objektů, založenou na fúzi míst a/nebo přechodů jednotlivých sítí. Kromě toho je zde možné koordinovat objekty společným prováděním přechodů v tzv. koordinačním objektu a v objektu, který je připojen ke značce v síti koordinačního objektu, podobně jako je tomu v případě dynamických objektů R. Valka. V rámci dědičnosti je možné zjemňovat strukturu zděděné sítě (obdobný přístup používá i OOPN). PN-TOX však nedefinuje chování systému jako celku, ale místo toho doporučuje tvorbu komponent a jejich synchronizačních schemat, analyzovatelných autonomně. Tento přístup koresponduje s tvorbou otevřených systémů.

V poslední době se autor PN-TOXu T. Holvoet zabývá specifikací agentů Petriho sítěmi, komunikujícími generativním způsobem (prostřednictvím nástěnky, jako v případě jazyka Linda) [HV96].

4.4.6 Interaction Coordiantion Nets

Obdobný přístup, tedy zapouzdření sítí do objektů v jazycích Smalltalk a Java, vedl k vývoji Interaction Coordiantion Nets [Von95a, Von95b, Von97, Von98], vyvinutých pro modelování a řízení workflow. Interaction Coordiantion Nets jsou modifikované hierarchické sítě, kde značky reprezentují libovolné objekty. Síť, spolu s jejím interpretem je implementována jako objekt hostitelského jazyka (Smalltalk, Java). Sítě komunikují sdílenými místy a k vzájemnému dynamickému propojení mohou použít specializovaný objekt, ve kterém se jednotlivé sítě registrují. Celý systém je otevřený a může pracovat distribuovaně.

4.4.7 CodeSign

Modely v CodeSign [Ess97b, Ess97a] se hierarchicky konstruují z komponent. Komponenty jsou definovány Petriho sítěmi a mohou být staticky propojeny na principu fúze míst. Komponenty jsou definovány jako třídy, které mohou být specifikovány inkrementálně s využitím dědičnosti. Je zde však dovolena jen statická instanciace. Zajímavým rysem CodeSign je možnost tranformace jiných podobných formalismů do Petriho sítí na základě grafových gramatik.

4.4.8 Object Petri Nets

C. Lakos vyvinul formalismus, nazvaný Object Petri nets [Lak94, Lak95], a s ním spojený jazyk LOOPN++ [LK94]. LOOPN++ je následníkem jazyka LOOPN (Language for Object Oriented Petri Nets) [Lak91], od kterého se liší především tím, že již umožňuje používat dynamicky instanciované sítě jako značky.

Jde o jistou formu zobecnění hierarchické Petriho sítě. Objekt, s kterým je možno synchronně komunikovat předáváním značek, je zde chápán jako zobecněné místo. Funguje zde dědičnost i polymorfismus. Klient-server komunikace je řešitelná emulací, tj. dvojicí přechodů pro předání zprávy a převzetí odpovědi. Jde o univerzální a flexibilní řešení. Definice i samotný formalismus jsou však poměrně komplikované.

4.4.9 Cooperative Nets

C. Sibertin-Blanc definoval formalismus nazvaný Cooperative Nets [SB94]. Petriho sítě zde reprezentují komponenty systému, které komunikují protokolem klient-server. Je zde možná dynamická instanciace těchto sítí. Zaslání zprávy způsobí umístění značek v roli parametrů do speciálních míst v instanci sítě serveru. Klient potom čeká na výsledek. Každý objekt má jednoznačnou identifikaci, která je součástí předávaných značek.

Implementace Cooperative Nets v C++, SYROCO [SBHT95], určená k vývoji otevřeného softwaru, dovoluje navíc oproti definici také synchronní přístup k atributům objektu. Existuje několik dalších projektů, spjatých s Cooperative Nets. Jde o různé varianty implementace téhož principu [BP95, PB95].

Tento způsob zavedení objektové orientace do Petriho sítí má nejblíže k přístupu, který používá OOPN. Odlišnost spočívá v tom, že C. Sibertin-Blanc používá kompaktní definici bez explicitního vyjádření sítí metod. Celý objekt je reprezentován jednou sítí a metoda je zde reprezentována jen vstupními a výstupními místy. Další odlišností je skutečnost, že OOPN vychází z širšího chápání polymorfismu, což souvisí s inspirací Smalltalkem, který nepracuje s typy proměnných. OOPN navíc formálně definuje také synchronní komunikaci objektů.

4.4.10 Object Colored Petri Nets

Daniel Moldt se zabývá aplikovatelností Petriho sítí v objektově orientovaných metodách tvorby softwaru [Mol95]. Moldt a Maier [MM97, Mai97] navrhují začlenit do OMT [RBP+91] formalismus, který je zdánlivě totožný s Cooperative Nets, liší se však tím, že klient-server komunikace je implementována synchronními kanály [CH94]. Další významnou vlastností OCPN je silná vazba na CPN – definice OCPN se odvolává na definici CPN, fúzi míst a na definici synchronních kanálů. Díky této provázanosti s CPN by mělo být perspektivně možné snadněji přizpůsobit teorii CPN pro OCPN. Srovnání tohoto formalismu s OOPN je stejné jako v případě Cooperative Nets.

4.4.11 Shrnutí

Z výše uvedného vyplývá, že objektová orientace může být do Petriho sítí integrována různými způsoby, lišícími se definicí pojmu objekt, způsobem komunikace objektů a její implementací prostředky Petriho sítí. Lze též posuzovat úroveň „čistoty“ použitého objektového modelu a transformovatelnost objektových konstrukcí do HL-sítě.

Významné odlišnosti OOPN od jiných přístupů lze shrnout takto:

- Značky v OOPN jsou reference na objekty.
- Třída není definována jedinou sítí, ale množinou sítí (zde se uplatnila inspirace Smalltalkem, jehož browser přistupuje k jednotlivým metodám i k reprezentaci objektu zvlášť). OOPN tedy zavádí oproti alternativním přístupům ještě jednu úroveň strukturování navíc.
- Objekty, díky přímé inspiraci Smalltalkem i jinými objektově orientovanými jazyky, komunikují protokolem klient-server. Tento způsob komunikace je implemementován invokací sítí metod, kterým jsou předány parametry, a které komunikují sdílenými místy se sítí objektu. Invokační přechod čeká na výsledek v return-místě a pak se dokončí.[15]
- Polymorfní značené přechody. Nerozlišuje se invokační a atomický přechod, přechody se chovají polymorfně: až při provádění přechodu se rozhodne, zda se provede atomicky či nikoliv. Proto se zavádí značení přechodu, které obsahuje informaci o invokovaných metodách. Vyšší míra polymorfismu souvisí s absencí statické typové kontroly, což považujeme za výhodu – umožňuje lepší znovupoužitelnost a snazší programování.
- Synchronní interakce jako součást stráže přechodu. I stráž přechodu může obsahovat zaslání zprávy objektu, popsanému Petriho sítěmi. Vyhodnocení stráže je vždy výpočet bez vlivu na stav systému. Specifikace zaslání zprávy ve stráži přechodu ovlivňuje způsob zjištění proveditelnosti přechodu (bez vlivu na stav systému) a s respektováním polymorfismu může implikovat synchronní komunikaci v případě provedení přechodu.
- Jednoduchost a sugestivnost formalismu. Grafická reprezentace třídy s explicitně vyjádřenými metodami je sugestivnější než u alternativních přístupů.[16] Samotné Petriho sítě jsou minimálně modifikovány, takže výsledný formalismus je snadno pochopitelný. Programování je významně usnadněno tím, že není třeba deklarovat typy proměnných.

[15] Ostatní přístupy (Sibertin, Lakos) definují klient-server interakci jako nadstavbu nad jednosměrným modelem komunikace, ať už synchronním nebo asynchronním, a objekt je implementován jednou sítí, se speciálními uzly, určenými k předávání zpráv.

[16] Alternativní přístupy nemají metody explicitně vyjádřené separátními sítěmi.

Kapitola 5

Objektově orientovaná Petriho síť

V této kapitole bude formálně definována objektově orientovaná Petriho síť. Definice je formulována analogicky k definici HL-sítě, uvedené v kapitole 2. Čtenář může průběžně konfrontovat definované pojmy (především dynamiku OOPN) s komentovaným příkladem, uvedeným v příloze B (str. 125).

5.1 Úvod

Definice objektově orientované Petriho sítě (OOPN) má několik úrovní. Základní úroveň tvoří definice *systému jmen a primitivních objektů*. Tento systém poskytuje primitivní sémantiku výrazům, které jsou součástí Petriho sítí. Dále je definován *systém sítí* a *instance sítí*. Sítě jsou součástí tříd. OOPN je definována *systémem tříd* a specifikací *počáteční třídy*. Dále je definován *objekt* a *systém objektů*. Systém objektů modeluje okamžitý stav systému, popsaného OOPN. Dynamika OOPN je vymezena *počátečním systémem objektů* a pravidly pro generování bezprostředně následujících systémů objektů prováděním přechodů Petriho sítí.

Styl definice OOPN vychází z definice HL-sítě (viz kap. 2) a některé pojmy, jako je systém sítí a instance sítě, jsou inspirovány pojmy, zavedenými v [SB94].

5.2 Inskripční jazyk a primitivní objekty

5.2.1 Systém jmen a primitivních objektů

Rozlišujeme *primitivní* a *neprimitivní objekty*. Neprimitivní (uživatelem definované) objekty jsou specifikovány třídami (jsou to instance tříd) a obsahují stavovou informaci, která může být v průběhu evoluce systému měněna jednak vnitřní aktivitou objektu, jednak prováděním metod v důsledku akceptování příchozích zpráv. Naproti tomu, primitivní objekty jsou konstanty, které jsou implicitně dostupné prostřednictvím svých jmen. Jde například o čísla, booleovské hodnoty, symboly, textové řetězce atd. Jelikož jde o konstanty, je možné, na rozdíl od neprimitivních objektů, primitivní objekty ztotožnit s jejich jmény. Podobně i třídy mají charakter konstant.

Jako výchozí bod formální definice OOPN zavedeme *universum*, které zahrnuje množinu primitivních objektů, množinu jmen neprimitivních objektů a množinu jmen tříd. Prvky těchto množin nazveme *atomy*. Každému atomu je přiřazen *typ*. Každému typu přísluší *doména* – množina všech potenciálních instancí typu. V případě, že typem je třída, doménu tvoří množina jmen všech jejích potenciálních instancí. Kromě atomů universum obsahuje n-tice atomů (s možností vnořování).

Dále zavedeme množinu *selektorů zpráv* a množinu *selektorů speciálních zpráv*. Pro primitivní objekty definujeme sémantiku všech zpráv prostřednictvím funkcí. Pro neprimitivní objekty

je tímto zůsobem definována jen sémantika speciálních zpráv, které operují pouze nad jmény, nikoliv nad stavem neprimitivních objektů (jde například o testování rovnosti jmen objektů).

Universum a zprávy tvoří systém jmen a primitivních objektů, který umožní definovat tzv. primitivní sémantiku výrazů jazyka, který bude použit v rámci Petriho sítí. Petriho sítě posléze umožní definovat specifikaci struktury instancí tříd, čímž bude definice struktury OOPN uzavřena. Na ni bude dále navazovat definice dynamiky OOPN.

Definice 5.2.1 Systém jmen a primitivních objektů je struktura

$$\Pi = (CONST, NAME, CLASS, Type, MSG, Mth_T, MSG_S, Mth_S, V),$$

kde

1. $CONST$ je množina, jejíž prvky nazveme **primitivní objekty**.
   $\{0, 1, ...\} \subset CONST$, $\{\texttt{true}, \texttt{false}\} \subset CONST$.

2. $NAME$ je množina, jejíž prvky nazveme **jména neprimitivních objektů**.

3. $CLASS$ je konečná množina, jejíž prvky nazveme **jména tříd**.

   Množiny $CONST$, $NAME$ a $CLASS$ jsou po dvojicích disjunktní.

   Nechť $U_0 = CONST \cup NAME \cup CLASS$. Prvky množiny U_0 nazveme **atomy**.

   Definujeme **universum** U takto:[1]

   (a) $U_0 \subseteq U$,
   (b) $n \in \mathbb{N} \wedge x_1 \in U \wedge ... \wedge x_n \in U \Longrightarrow (x_1, ..., x_n) \in U$.

4. $Type$ je funkce, definovaná na U_0 tak, že

   - $\forall x \in CONST$ [$Type(x) \notin CLASS$, $Type(x)$ je **typ** primitivního objektu],
   - $\forall x \in NAME$ [$Type(x) \in CLASS$],
   - $\forall x \in CLASS$ [$Type(x) = \texttt{class}$].

   Definujeme množinu primitivních typů $TYPE = \{Type(x) \mid x \in CONST\}$.
   $\texttt{class}$ je speciální konstanta, $\texttt{class} \notin CLASS \cup TYPE$.

   Dále definujeme funkci $Dom : TYPE \cup CLASS \cup \{\texttt{class}\} \longrightarrow 2^{U_0}$ takto:[2]

   $Dom(t) = \{x \mid x \in U_0 \wedge Type(x) = t\}$.

5. MSG je konečná množina, jejíž prvky nazveme **selektory zpráv**.

   $MSG \cap U = \emptyset$, $\texttt{new}^{(0)} \in MSG$.

   Je-li $msg^{(n)} \in MSG$ selektor zprávy, n je jeho arita, $n \geq 0$.

6. Mth_T je funkce, definovaná na $MSG \times TYPE$ tak, že:[3]

   $Mth_T(msg^{(n)}, t) = f$, kde f je funkce tvaru $f : Dom(t) \times CONST^n \longrightarrow CONST$.

[1] Universum kromě atomů obsahuje vnořené n-tice atomů.

[2] Domény definujeme jako disjunktní, bez možnosti inkluze. Nevytváříme žádnou hierarchii typů. Pro n-tice není třeba Dom definovat.

[3] Definujeme zde pro každý primitivní objekt reakce na všechny zprávy se selektory z MSG a s argumenty, kterými jsou primitivní objekty. V případě, že by bylo třeba rozpoznat situaci, kdy objekt zprávě nerozumí, lze reakce na zprávy prohlásit za parciální funkce, avšak s tím, že jejich nedefinovanost musí být algoritmicky rozhodnutelná. Pak lze v definici sémantiky inskripčního jazyka (def. 5.2.4) rozlišovat vyhodnotitelné a nevyhodnotitelné výrazy.

7. MSG_S je množina, jejíž prvky nazveme **selektory speciálních zpráv**, $MSG_S \cap MSG = \emptyset$, $MSG_S \cap U = \emptyset$.

8. Mth_S je funkce, definovaná na $MSG_S \times (TYPE \cup CLASS \cup \{\texttt{class}\})$ tak, že: $Mth_S(msg^{(n)}, c) = g$, kde g je funkce tvaru $g : Dom(c) \times U_0^n \longrightarrow U_0$.

9. V je konečná množina **proměnných**.

   Symboly `self` a `super` jsou **pseudoproměnné**, $\{\texttt{self}, \texttt{super}\} \cap (U \cup V) = \emptyset$.

   Jakoukoliv funkci $b : V' \longrightarrow U$, $V' \subseteq V$, nazveme **navázání** množiny proměnných V'.

5.2.2 Multimnožiny a n-tice

Multimnožiny a n-tice, potřebné pro definici sémantiky hranových výrazů, byly již definovány v kapitole 2, def. 2.3.1 a def. 2.3.2. Doplníme definici funkce *supp*, která vrací množinu prvků multimnožiny nebo n-tice. Funkce pracuje korektně i pro vnořené n-tice.

Definice 5.2.2 Nechť E je libovolná množina a Π je systém jmen a primitivních objektů (def. 5.2.1).

1. - Pro libovolnou multimnožinu $x : E \longrightarrow \mathbb{N}$ definujeme funkci $supp^{MS}(x) = \{e \in E \mid x(e) \neq 0\}$.
   - Pro libovolnou n-tici $l = (a_1, ..., a_n)$, $l \in E^n$, $n \in \mathbb{N}$, definujeme funkci $supp^*(l) = \bigcup_{i \in \{1,...,n\}} \{a_i\}$.

2. Nad systémem jmen a primitivních objektů Π definujeme funkci *supp* takto:

   - $\forall x \in U_0$ definujeme $supp(x) = \{x\}$.
   - $\forall x \in U - U_0$ definujeme[4] $supp(x) = \bigcup_{e \in supp^*(x)} supp(e)$.

3. Funkci *supp* rozšíříme i na množiny a multimnožiny prvků univerza takto:

   - $\forall x \in 2^U$ definujeme $supp(x) = \bigcup_{e \in x} supp(e)$.
   - $\forall x \in U^{MS}$ definujeme $supp(x) = \bigcup_{e \in supp^{MS}(x)} supp(e)$.

5.2.3 Výrazy

Nyní definujeme syntax a tzv. primitivní sémantiku výrazů, které budeme používat pro popisy (inskripce) Petriho sítí. Jde o *zaslání zprávy* a *hranový výraz*. Zasílání zpráv se bude vyskytovat ve strážích a akcích přechodů, hranové výrazy na hranách Petriho sítí (jde o hrany bipartitního grafu, reprezentující Petriho síť, která bude definována v sekci 5.3).

Definice 5.2.3 Nad systémem jmen a primitivních objektů Π (def. 5.2.1) definujeme tyto druhy výrazů:

1. **Term** je prvek množiny $TERM(\Pi)$, $TERM(\Pi) = U_0 \cup V \cup \{\texttt{self}, \texttt{super}\}$.

2. **Zaslání zprávy** je prvek množiny $EXPR(\Pi)$, definované takto:

[4] Tímto definujeme *supp* pro vnořené n-tice.

- $TERM(\Pi) \subseteq EXPR(\Pi)$.
- Každý výraz tvaru $e_0.msg^{(m)}(e_1, e_2, ..., e_m)$, kde $\forall i \in \{0, ..., m\}$ [$e_i \in TERM(\Pi)$] a $msg^{(m)} \in (MSG \cup MSG_S)$, je prvkem množiny $EXPR(\Pi)$.[5]

3. Definujeme pomocnou množinu $LISTEXPR(\Pi)$ takto:

   - $TERM(\Pi) \subseteq LISTEXPR(\Pi)$.
   - Každý výraz tvaru $(e_1, e_2, ..., e_m)$, kde $\forall i \in \{1, ..., m\}$ [$e_i \in LISTEXPR(\Pi)$], $m \in \mathbb{N}$, je prvkem množiny $LISTEXPR(\Pi)$.

4. **Hranový výraz** je prvek množiny $ARCEXPR(\Pi)$, definované takto:[6]

   - Každý výraz tvaru e_1‘e_2, kde $e_1 \in TERM(\Pi)$, $e_2 \in LISTEXPR(\Pi)$, je prvkem množiny $ARCEXPR(\Pi)$.
   - Každý výraz tvaru $e_1 + e_2$, kde $e_1, e_2 \in ARCEXPR(\Pi)$, je prvkem $ARCEXPR(\Pi)$.

Nechť $Var(e) \subseteq V$ je množina všech proměnných ve výrazu e.

Poznámka. Uvedená syntax zaslání zprávy je definována s ohledem na potřeby definice OOPN. V konkrétních implementacích OOPN (např. v jazyce PNtalk, jak bude uvedeno v kapitole 6) může být syntax zaslání zprávy a hranového výrazu mírně odlišná.

Definice 5.2.4 Mějme systém jmen a primitivních objektů Π (def. 5.2.1). Definujeme $e\langle b \rangle$ jako výsledek vyhodnocení výrazu e při navázaní $b : V' \longrightarrow U$, $Var(e) \subseteq V' \subseteq V$, takto:

1. (a) Je-li $e \in (CONST \cup CLASS)$, pak $e\langle b \rangle = e$.
   (b) Je-li $e \in V$, pak $e\langle b \rangle = b(e)$.
   (c) Je-li $e \in \{\texttt{self}, \texttt{super}\}$, pak vyhodnocení výrazu e závisí na kontextu a jeho sémantika bude definována později (def. 5.5.1).

2. Je-li e výraz tvaru $e_0.msg(e_1, e_2, ..., e_m)$, kde $\forall i \in \{0, ..., m\}$ [$e_i \in TERM(\Pi)$], pak:

   (a) Je-li $msg^{(m)} \in MSG$ a když $\forall i \in \{0, ..., m\}$ [$e_i\langle b \rangle \in CONST$], pak[7] $e\langle b \rangle = (Mth_T(msg^{(m)}, Type(e_0\langle b \rangle)))(e_1\langle b \rangle, e_2\langle b \rangle, ..., e_m\langle b \rangle)$.
   (b) Je-li $msg^{(m)} \in MSG_S$ a když $\forall i \in \{0, ..., m\}$ [$e_i\langle b \rangle \in U_0$], pak $e\langle b \rangle = (Mth_S(msg^{(m)}, Type(e_0\langle b \rangle)))(e_1\langle b \rangle, e_2\langle b \rangle, ..., e_m\langle b \rangle)$.
   (c) Je-li $msg^{(m)} \in MSG$ a když $e_0\langle b \rangle \in (NAME \cup CLASS)$, jde o **neprimitivní zaslání zprávy**. Jeho sémantika bude vysvětlena v rámci definice dynamiky OOPN (sekce 5.5).
   (d) V případech, které nejsou pokryty body (a), (b), (c), výraz e nelze vyhodnotit.[8]

[5] Bylo by sice možné připustit, aby e_i bylo zaslání zprávy (nikoliv jen term) ale zkomplikovala by se tím definice sémantiky neprimitivního zaslání zprávy (viz dále, sekce 5.5). Proto ponecháme e_i jako termy. Není to na úkor obecnosti, protože vnořené zaslání zprávy lze vyjádřit jinými prostředky (buď strukturou sítě nebo vyhodnocením v rámci stráže přechodu).

[6] Na hranách Petriho sítí se mohou vyskytovat multimnožiny seznamů termů.

[7] Podle poznámky k bodu 6 v def. 5.2.1 lze připustit, že výraz e může být nevyhodnotitelný. Vzhledem k tomu, že vyhodnocování tohoto typu výrazů se bude vyskytovat ve strážích a akcích přechodů, nevyhodnotitelnost výrazu implikuje neproveditelnost přechodu, v jehož stráži nebo akci se výraz vyskytuje.

[8] Tento případ může nastat při zjišťování proveditelnosti přechodu, jak bude uvedeno v sekci 5.5. Přechod je pak neproveditelný.

V případech (a), (b), (d) jde o **primitivní zaslání zprávy**, v případech (a), (b) **vyhodnotitelné**, v případě (d) **nevyhodnotitelné**.[9]

3. (a) Je-li $e = (e_1, ..., e_n)$, kde $\forall i \in \{1, ..., m\}$ [$e_i \in LISTEXPR(\Pi)$],
   pak $e\langle b \rangle = (e_1\langle b \rangle, ..., e_n\langle b \rangle)$.

4. (a) Je-li e výraz tvaru e_1‘e_2, kde $e_1 \in TERM(\Pi)$, $e_2 \in LISTEXPR(\Pi)$,
   pak $e\langle b \rangle = e_1\langle b \rangle$‘$e_2\langle b \rangle$.

   (b) Je-li e výraz tvaru $e_1 + e_2$, kde $e_1, e_2 \in ARCEXPR(\Pi)$,
   pak $e\langle b \rangle = e_1\langle b \rangle + e_2\langle b \rangle$.

5.3 Struktura OOPN

Neprimitivní objekty jsou definovány třídami, které jsou specifikovány prostřednictvím Petriho sítí. Sítě jsou složeny z míst a přechodů, propojených hranami, které vyjadřují vstupní a výstupní podmínky přechodů. Místo sítě může obsahovat značky, přičemž značka může reprezentovat primitivní objekt (číslo, symbol, řetězec apod.), jméno objektu (zde máme na mysli neprimitivní, uživatelem definovaný objekt), jméno třídy, nebo n-tici z nich složenou (uvažujeme i vnořené n-tice). Hrany, reprezentující vstupní a výstupní podmínky přechodů, jsou ohodnoceny hranovými výrazy, které po navázání proměnných reprezentují multimnožiny prvků univerza U.

Přechod obsahuje stráž a akci. Stráž obsahuje výrazy (zasílání zpráv) a je splněna právě tehdy, když jsou všechny tyto výrazy vyhodnoceny jako `true`. Akce je zaslání zprávy s možností přiřadit výsledek do proměnné. Zaslání zprávy v akci přechodu se interpretuje (za běhu) podle typu adresáta a selektoru zprávy buď jako primitivní zaslání zprávy, nebo jako vytvoření nového objektu (neprimitivního), nebo jako žádost o provedení operace objektu s čekáním na výsledek (jde o invokaci metody). Provádění přechodů bude definováno v sekci 5.5 Dynamika OOPN.

Vzhledem k potenciální možnosti neatomického provedení přechodu jsou zavedeny tzv. *podmínky přechodu* (podmínky bez přívlastku, různé od vstupních a výstupních podmínek), graficky reprezentované oboustrannými šipkami. Jejich sémantika je v případě atomického provedení přechodu ekvivalentní dvojici vstupní a výstupní podmínky (dvě orientované hrany – jedna vedoucí z a druhá do přechodu) se stejným hranovým výrazem. Sémantika podmínky při neatomickém provedení přechodu je odlišná a bude popsána v sekci 5.5 Dynamika OOPN.

Specifikace třídy obsahuje síť objektu, sítě metod, synchronní porty a mapování zpráv na metody a synchronní porty. Synchronní porty jsou určeny k volání ze stráží přechodů. Slouží k atomickému otestování stavu neprimitivního objektu a v případě provedení přechodu i k jeho případné změně.

5.3.1 Sítě

Definice 5.3.1 Síť objektu je pětice

$$N = (\Pi, P_N, T_N, PI_N, TI_N),$$

kde

1. Π je systém primitivních jmen a objektů.
2. P_N je konečná množina **míst**.
3. T_N je konečná množina **přechodů**, $P_N \cap T_N = \emptyset$.

[9] Za nevyhodnotitelné primitivní zaslání zprávy budeme považovat i případ z poznámky k bodu 6 v def. 5.2.1.

4. $PI_N : P_N \longrightarrow ARCEXPR(\Pi)$ je **inicializační funkce míst**,
   $\forall p \in P_N \; [\; Var(PI_N(p)) = \emptyset \;]$.

5. TI_N je **popis přechodů**. Je to funkce, definovaná na T_N, taková, že $\forall t \in T_N$:

   $$TI_N(t) = (COND_t^N, PRECOND_t^N, GUARD_t^N, ACTION_t^N, POSTCOND_t^N),$$

   kde

   (a) $COND_t^N$, $PRECOND_t^N$, $POSTCOND_t^N : P_N \longrightarrow ARCEXPR(\Pi)$ jsou funkce **podmínek**, **vstupních podmínek** a **výstupních podmínek**.

   (b) $GUARD_t^N = \{G_1, ..., G_m\}$,
   kde $\forall i \in \{1, 2, ..., m\} \; [\; G_i \in EXPR(\Pi) \;]$, je **stráž přechodu**.[10] Definujeme množinu
   $InVar_N(t) = \bigcup_{i \in \{1,...,m\}} Var(G_i) \cup \bigcup_{p \in P_N} (Var(COND_t^N(p)) \cup Var(PRECOND_t^N(p)))$.

   (c) $ACTION_t^N$ je **akce přechodu** – výraz tvaru[11]

   $$y := expr,$$

   kde $y \in V - InVar_N(t)$ a $expr \in EXPR(\Pi)$, $Var(expr) \subseteq InVar_N(t)$.
   Definujeme množinu $Var_N(t) = InVar_N(t) \cup \{y\}$.

Definice 5.3.2 Mějme síť objektu N. **Synchronní port** nad sítí N je čtveřice

$$R^N = (COND_R^N, PRECOND_R^N, GUARD_R^N, POSTCOND_R^N),$$

kde $COND_R^N, PRECOND_R^N, GUARD_R^N, POSTCOND_R^N$ jsou definovány stejně jako v definici popisu přechodů sítě objektu (viz předchozí definice). Definujeme $Var(R)$ jako množinu proměnných, vyskytujících se ve výrazech $COND_R^N, PRECOND_R^N, GUARD_R^N, POSTCOND_R^N$. Prvky $Var(R)$ nazveme **parametry** synchronního portu.

Synchronní port se volá předáním zprávy, specifikovaným ve stráži přechodu. Vyhodnocení jeho proveditelnosti má vliv na proveditelnost volajícího přechodu. Při provedení přechodu se synchronně provede i synchronní port, jako by to byl přechod, pro příslušné navázání proměnných. Tyto skutečnosti budou definovány v sekci 5.5 Dynamika OOPN.

Definice 5.3.3 Síť metody je struktura

$$F = (\Pi, P_F, T_F, PI_F, TI_F, PP_F, RP_F),$$

kde

1. $\Pi, P_F, T_F, PI_F, TI_F$ mají stejný význam jako $\Pi, P_N, T_N, PI_N, TI_N$ v definici sítě objektu.

2. $PP_F \subset P_F$ je množina **parametrových míst**.

3. $RP_F \in P_F - PP_F$ je **výstupní (return) místo**.[12]

[10] Předpokládá se, že navázání proměnných, pro které je výraz G_i vyhodnotitelný jako `true`, lze vypočítat lépe než zkoušením všech možných navázání. Tento problém se vyskytuje v každé variantě HL-sítí [Jen92].

[11] Vzhledem k možnosti nepoužít y ve výrazech na výstupních hranách a k možnosti vyhodnocení jako primitivní zaslání zprávy uvedený tvar výrazu pokrývá i variantu, kdy je výraz prázdný. Sekvenci výrazů zde nepřipouštíme.

[12] Předpokládá se, že výstupní místo má prázdné počáteční značení, tj. inicializační výraz, přiřazený výstupnímu místu, se vyhodnotí jako prázdná multimnožina.

Jak síť objektu, tak síť metody budeme jednotně nazývat síť, pokud jejich odlišnosti nebudou významné. Pro potřeby definice systému tříd je dále třeba definovat systém sítí a některé významné relace mezi sítěmi.

Definice 5.3.4 Systém sítí NS je dvojice

$$NS = (NET, Inst),$$

kde

1. NET je množina sítí, splňující podmínku $(\bigcup_{N \in NET} P_N) \cap (\bigcup_{N \in NET} T_N) = \emptyset$.
2. $Inst$ je funkce, definovaná na NET taková, že
   - $\forall n \in NET$ [$Inst(n)$ je množina, jejíž prvky nazveme **jména instancí** sítě n],[13]
   - pro $n \neq n'$ je $Inst(n) \cap Inst(n') = \emptyset$.

Pro systém sítí NS definujeme množinu

$$INST_{NS} = \bigcup_{n \in NET} Inst(n)$$

a funkci

$Net : INST_{NS} \longrightarrow NET$, takovou, že

$\forall id \in INST_{NS}$ [$Net(id) = n \iff id \in Inst(n)$].

Definice 5.3.5 Mějme systém sítí $NS = (NET, Inst)$. NS je **polodisjunktní systém sítí**, když $\forall (N_1, N_2) \in NET \times NET$ [$N_1 \neq N_2 \Longrightarrow T_{N_1} \cap T_{N_2} = \emptyset$].

Systém sítí zaručuje, že přechod jedné sítě nemůže vystupovat v roli místa jiné sítě a naopak. Připouští však, aby se jeden uzel (místo nebo přechod) mohl vyskytovat ve více sítích současně. Tato skutečnost umožní definovat dědičnost Petriho sítí. Polodisjunktní systém sítí umožňuje pouze sdílení míst různými sítěmi. Množiny přechodů jednotlivých sítí jsou po dvojicích disjunktní. Sdílení míst využijeme v definici struktury instance třídy.

Definice 5.3.6 Mějme systém sítí $(\{N_1, N_2\}, Inst)$. Síť N_1 **dědí strukturu** sítě N_2, což označujeme $N_1 \leq N_2$, právě tehdy, když $P_{N_1} \supseteq P_{N_2} \wedge T_{N_1} \supseteq T_{N_2}$. Síť N_1 **sdílí místa** sítě N_2, což označujeme $N_1 \prec N_2$, právě tehdy, když $P_{N_1} \supseteq P_{N_2} \wedge T_{N_1} \cap T_{N_2} = \emptyset \wedge PI_{N_1}|_{P_{N_2}} = PI_{N_2}$.

Dědičnost sítí je definována jako sdílení uzlů různými sítěmi. Propojení uzlů zde není významné.

5.3.2 Třídy

Definice 5.3.7 Systém tříd je pětice

$$\Sigma = (\Pi, Spec_\Sigma, Inst_\Sigma, ROOT_\Sigma, Super_\Sigma),$$

kde

1. $\Pi = (CONST_\Sigma, NAME_\Sigma, CLASS_\Sigma, Type_\Sigma, MSG_\Sigma, Mth^T_\Sigma, MSG^S_\Sigma, Mth^S_\Sigma, V_\Sigma)$ je systém jmen a primitivních objektů (def. 5.2.1),

[13] Prozatím nespecifikujeme žádný vztah jmen instancí sítí ke jménům neprimitivních objektů. Do souvislosti je uvedeme v definici systému tříd.

2. $Spec_\Sigma$ je **specifikace struktury instancí třídy**, definovaná tak, že $\forall c \in CLASS_\Sigma$:

$$Spec_\Sigma(c) = (ONET_c, MNETS_c, SYNC_c, MSG_c, Method_c),$$

kde

(a) $ONET_c$ je síť objektu,

(b) $MNETS_c$ je konečná množina sítí metod,

(c) $SYNC_c$ je konečná množina synchronních portů nad sítí $ONET_c$,

(d) MSG_c je konečná množina **selektorů zpráv**, $MSG_c \subseteq MSG_\Sigma$,

(e) $Method_c$ je bijekce $Method_c : MSG_c \longleftrightarrow (MNETS_c \cup SYNC_c)$, taková, že:

- $\forall m^{(r)} \in MSG_c \; [\; Method_c(m^{(r)}) \in MNETS_c \Longrightarrow |PP_{Method_c(m^{(r)})}| = r \;]$,
- $\forall m^{(r)} \in MSG_c \; [\; Method_c(m^{(r)}) \in SYNC_c \Longrightarrow |Var(Method_c(m^{(r)}))| = r \;]$.

3. $(NET_\Sigma, Inst_\Sigma)$, kde $NET_\Sigma = \bigcup_{c \in CLASS_\Sigma}(\{ONET_c\} \cup MNETS_c)$, je systém sítí, takový, že $\forall c \in CLASS_\Sigma$ jsou současně splněny tyto podmínky:

- $(\{ONET_c\} \cup MNETS_c, Inst_\Sigma|_{\{ONET_c\} \cup MNETS_c})$ je polodisjunktní systém sítí, pro který platí:

  $$\forall n \in MNETS_c \; [\; n \prec ONET_c \wedge PP_n \cap P_{ONET_c} = \emptyset \wedge RP_n \notin P_{ONET_c} \;],$$

- $Inst_\Sigma(ONET_c) = \bigcup\limits_{c' \in H(c)} Dom(c')$, kde[14]

  $$H(c) = \{c' | c' \in CLASS_\Sigma \wedge ONET_{c'} = ONET_c\},$$

- $\forall n \in MNETS_c \; [\; Inst_\Sigma(n) \cap NAME_\Sigma = \emptyset \;]$,

4. $ROOT_\Sigma \in CLASS_\Sigma$ je **kořen hierarchie dědičnosti tříd**, pro který platí

$$\forall c \in CLASS_\Sigma \; [\; ONET_c \leq ONET_{ROOT_\Sigma} \wedge MSG_c \supseteq MSG_{ROOT_\Sigma} \;],$$

5. $Super_\Sigma : CLASS_\Sigma - \{ROOT_\Sigma\} \longrightarrow CLASS_\Sigma$ je funkce, splňující současně tyto podmínky:[15]

- $\forall c \in CLASS_\Sigma - \{ROOT_\Sigma\} \; [\; ONET_c \leq ONET_{Super_\Sigma(c)} \wedge MSG_c \supseteq MSG_{Super_\Sigma(c)} \;]$,
- $\forall (c_1, c_2) \in CLASS_\Sigma \times CLASS_\Sigma$
  $[\; MNETS_{c_1} \cap MNETS_{c_2} \neq \emptyset \vee SYNC_{c_1} \cap SYNC_{c_2} \neq \emptyset \vee ONET_{c_1} = ONET_{c_2} \Longrightarrow$
  $\exists c \in CLASS_\Sigma \; [\; c \in Super^*(c_1) \wedge c \in Super^*(c_2) \;] \;]$,

[14] $Dom(c)$ specifikuje množinu jmen všech potenciálních instancí třídy se jménem c se specifikací $Spec(c)$. Je to podmnožina množiny instancí sítě objektu $ONET_c$, neboť tato síť se může vyskytovat ve specifikacích více tříd a přitom je třeba zajistit, aby bylo možné objekt identifikovat stejným jménem, jako má instance sítě $ONET_c$.

[15] Použijeme zde tuto notaci: Je-li f funkce, f^* označuje její tranzitivně-reflexivní uzávěr (funkci interpretujeme jako relaci). Je-li R relace, $R(x)$ je množina druhých složek prvků relace R, jejichž první složkou je x.

Funkce $Super_\Sigma$ definuje hierarchii tříd. První podmínka definuje vztah dědičnosti. Druhá podmínka vymezuje dědění metod: Vyskytuje-li se síť metody nebo synchronní port ve dvou třídách, tyto mají v hierarchii dědičnosti společného předka. Ostatní tři podmínky zajišťují konzistenci dědění metod, synchronních portů a sítě objektu, aby mohla být definována funkce Def (viz def. 5.3.9), která pro síť nebo synchronní port vrací třídu, ve které byla tato síť nebo synchronní port definován(a), tj. třídu, která se nachází se v hierarchii dědičnosti tříd nejblíže kořenu (nejvýš) ze všech tříd, které danou síť objektu nebo synchronní port obsahují. Konkrétně, třetí podmínka zakazuje přiřadit tutéž metodu (síť metody nebo synchronní port) různým selektorům zpráv v různých třídách. Čtvrtá podmínka zakazuje vracet se při dědění po nové definici metody k původní metodě (tj. i když se k původní podobě metody vrátíme, není to tatáž metoda). Pátá podmínka zakazuje vracet se po nové definici sítě objektu k původní síti (tj. i když se k původní podobě sítě objektu vrátíme, není to tatáž síť).

- $\forall(c_1, c_2) \in CLASS_\Sigma \times CLASS_\Sigma$
  $\forall(N_1, N_2) \in (MNETS_{c_1} \cup SYNC_{c_1}) \times (MNETS_{c_2} \cup SYNC_{c_2})$
  $[\ N_1 = N_2 \Longrightarrow Method_{c_1}^{-1}(N_1) = Method_{c_2}^{-1}(N_2)\ ]$,
- $\forall(c_1, c_2, c_3) \in CLASS_\Sigma \times CLASS_\Sigma \times CLASS_\Sigma$
  $[\ c_1 \in Super^*(c_2) \wedge c_2 \in Super^*(c_3) \Longrightarrow Method_{c_1} \cap Method_{c_3} \subseteq Method_{c_2}\ ]$,
- $\forall(c_1, c_2, c_3) \in CLASS_\Sigma \times CLASS_\Sigma \times CLASS_\Sigma$
  $[\ c_1 \in Super^*(c_2) \wedge c_2 \in Super^*(c_3) \wedge ONET_{c_1} = ONET_{c_3} \Longrightarrow ONET_{c_1} = ONET_{c_2}\ ]$.

Definujeme množiny:

$T_\Sigma = \bigcup_{n \in NET_\Sigma} T_n$,
$P_\Sigma = \bigcup_{n \in NET_\Sigma} P_n$,
$INST_\Sigma = \bigcup_{n \in NET_\Sigma} Inst_\Sigma(n)$,
$SYNC_\Sigma = \bigcup_{c \in CLASS_\Sigma} SYNC_c$,
$MNET_\Sigma = \bigcup_{c \in CLASS_\Sigma} MNETS_c$,
$ONET_\Sigma = \bigcup_{c \in CLASS_\Sigma} \{ONET_c\}$.

Teorém 5.3.1 Je-li Σ systém tříd, pak

$$\forall(c_1, c_2) \in CLASS_\Sigma \times CLASS_\Sigma\ [\ c_1 \neq c_2 \Longrightarrow Dom(c_1) \cap Dom(c_2) = \emptyset\ ].$$

Důkaz. Důkaz plyne přímo z bodu 4 v definici 5.2.1 (funkce $Type$ vytváří na množině U_0 rozklad).

Definice 5.3.8 Mějme systém tříd Σ. Třída $c_1 \in CLASS_\Sigma$ **dědí** od třídy $c_2 \in CLASS_\Sigma$, což zapisujeme $c_1 \leq c_2$, právě tehdy, když $c_2 \in Super_\Sigma^*(c_1)$.

Dědičnost umožňuje vytvářet třídy inkrementálně, tak, že se v definici třídy c pouze doplní odlišnosti od třídy $Super_\Sigma(c)$. Funkce $Super_\Sigma$ vrací jméno nejbližší nadtřídy v hierarchii dědičnosti.

Dále definujeme funkci Def_Σ, vracející jméno třídy, ve které byla síť nebo synchronní port definován(a). Ostatní třídy, ve kterých se daný synchronní port nebo metoda vyskytuje, od této třídy dědí (viz vlastnosti systému tříd, def. 5.3.7).

Definice 5.3.9 Pro systém tříd Σ definujeme funkci

$$Def_\Sigma : NET_\Sigma \cup SYNC_\Sigma \longrightarrow CLASS_\Sigma$$

tak, že $Def_\Sigma(n) = c$ právě tehdy, když jsou současně splněny tyto podmínky:

1. $n \in (\{ONET_c\} \cup MNETS_c \cup SYNC_c)$,
2. $\forall c' \in CLASS_\Sigma\ [\ c \leq c' \wedge n \in (\{ONET_{c'}\} \cup MNETS_{c'} \cup SYNC_{c'}) \Longrightarrow c' = c\ ]$.

Nyní již můžeme definovat OOPN (její strukturu). Dynamika OOPN bude vysvětlena v následujícím textu.

Definice 5.3.10 Objektově orientovaná Petriho síť je trojice

$$OOPN = (\Sigma, c_0, oid_0),$$

kde Σ je systém tříd, $c_0 \in CLASS_\Sigma$ je **počáteční třída** a $oid_0 \in Dom(c_0)$ je **jméno prvotního (počátečního) objektu**.

5.4 Systém objektů

OOPN modeluje dynamický systém. Abychom mohli zkoumat dynamiku OOPN, musíme zavést instance prvků OOPN, zejména instance sítí, ze kterých se skládají instance tříd, tedy objekty. Instance chápeme jako pojmenované okamžité stavy příslušných entit. Zatímco v případě HL-sítě (viz kap. 2) je stav systému určen značením sítě, v případě OOPN je stav systému určen systémem (množinou) objektů.

5.4.1 Instance sítí

Objekt (v daném stavu) je množina instancí sítí, splňující určité podmínky. Jedna z těchto instancí sítí je instance sítě objektu, ostatní (pokud jsou nějaké) jsou instance sítí právě rozpracovaných volání metod. Jméno instance sítě objektu je totožné se jménem objektu (viz def. 5.3.7, bod 3). Instance sítě je pojmenovaný stav sítě. Stav sítě je určen jejím značením – rozmístěním značek v místech sítě a stavem přechodů. Značka reprezentuje prvek univerza.

Definice 5.4.1 Mějme systém tříd Σ, třídu $c \in CLASS_\Sigma$, její specifikaci $Spec_\Sigma(c)$, síť objektu $ONET_c$ a síť metody $N \in MNETS_c$.

1. **Značení míst sítě objektu** $ONET_c$ je funkce $PM : P_{ONET_c} \longrightarrow U^{MS}$.
2. **Značení míst sítě metody** N je funkce[16] $M : (P_N - P_{ONET_c}) \longrightarrow U^{MS}$.

Konvence: Nerozlišujeme-li síť objektu a síť metody, mluvíme jednoduše o síti. V tom případě o značení míst sítě objektu a o značení míst sítě metody mluvíme jako o značení míst sítě.

Přechody v OOPN mohou, na rozdíl od tradičních Petriho sítí, probíhat neatomicky a nést stavovou informaci. Tato stavová informace říká, které dosud neukončené instance sítí (invokace metod) byly přechodem vytvořeny. Stav přechodu je tedy určen množinou invokací sítí.

Definice 5.4.2 Mějme systém sítí $NS = (NET, Inst)$ a sítě $N_1, N_2 \in NET$.

1. **Invokace** sítě N_2 přechodem $t \in T_{N_1}$ je dvojice (id, b), kde $id \in Inst(N_2)$ je jméno instance sítě N_2 a b je navázání proměnných $InVar_{N_1}(t)$. Pro každou invokaci $i = (id, b)$ definujeme $Id(i) = id$. Tuto funkci rozšíříme také pro množiny invokací:
   $Id(\{(id_1, b_1), ..., (id_n, b_n)\}) = \{id_1, ..., id_n\}$.
   Množinu všech invokací označíme INV.
2. Jakoukoliv funkci $TM : T_{N_1} \longrightarrow INV$, nazveme **značení přechodů** sítě N_1.
3. Nechť PM je značení míst sítě N_1. Funkci $M = PM \cup TM$ nazveme **značení** sítě N_1.

Definice 5.4.3 Mějme systém sítí $NS = (NET, Inst)$.

1. **Instance sítě** $N \in NET$ je dvojice (id, m), kde $id \in Inst(N)$ je jméno instance a m je značení sítě N.

   Pro každou instanci $i = (id, m)$ sítě $N \in NET$ definujeme $Id(i) = id$. Tuto funkci rozšíříme také pro množiny instancí sítí: $Id(\{(id_1, m_1), ..., (id_n, m_n)\}) = \{id_1, ..., id_n\}$.
2. **Systém instancí sítí** je množina instancí sítí $Q = \{(id_1, m_1), ..., (id_n, m_n)\}$, splňující tyto podmínky:

[16] Síť metody sdílí místa sítě objektu (podle definice systému tříd). Aby se zabránilo nežádoucím duplicitám, definujeme značení sítě metody jen pro místa, která nejsou sdílena.

(a) $\forall i \in \{1,...,n\}$ [(id_i, m_i) je instance sítě $Net(id_i)$, $Net(id_i) \in NET$],

(b) $\forall i,j \in \{1,...,n\}$ [$i \neq j \Longrightarrow id_i \neq id_j$].

3. Pro systém instancí sítí Q definujeme $Marking_Q(id) = m \Longleftrightarrow (id, m) \in Q$.

4. Definujeme funkce $ref^P, ref^T : Q \longrightarrow 2^{INST_{NS}}$ takto:

   Nechť pro každou množinu dvojic $X = \{(a_1, b_1), ..., (a_n, b_n)\}$ je definováno $Im(X) = \bigcup_{j\in\{1,...,n\}}\{b_j\}$. Pak:

   $\forall (id, m) \in Q$ [$ref^P((id, m)) = (\bigcup_{p\in P_{Net(id)}} supp(m(p))) \cap INST_{NS} \wedge$
   $ref^T((id, m)) = (\bigcup_{t\in T_{Net(id)}} (Id(m(t)) \cup \bigcup_{b\in Im(m(t))} supp(Im(b)))) \cap INST_{NS}$].

5. **Uzavřený systém instancí sítí** je systém instancí sítí Q, pro který platí

   $\forall i \in Q$ [$(ref^P(i) \cup ref^T(i)) \subseteq Id(Q)$].

Funkce ref^P a ref^T vracejí množiny jmen instancí sítí, které jsou referencované z míst resp. přechodů dané instance sítě. Uzavřený systém instancí sítí zaručuje platnost všech referencí, tj. skutečnost, že značení míst a přechodů referencují jen instance systému sítí.

5.4.2 Objekty

Definice 5.4.4 Mějme systém tříd Σ.

1. **Instance (objekt) třídy** $c \in CLASS_\Sigma$ je systém instancí sítí o, splňující současně tyto podmínky:

   (a) $\forall (id, m) \in o$ [$Net(id) \in \{ONET_c\} \cup \bigcup_{c'\in Super^*(c)} MNETS_{c'}$],

   (b) $|Id(o) \cap NAME_\Sigma| = |Id(o) \cap Dom(c)| = 1$.

2. Množina $S = \{o_1, ..., o_n\}$ je **systém objektů** nad systémem tříd Σ, když splňuje současně tyto podmínky:

   (a) $\forall j \in \{1,...,n\}$ $\exists c_j \in CLASS_\Sigma$ [o_j je objekt třídy c_j],

   (b) $\forall i,j \in \{1,...,n\}$ [$i \neq j \Longrightarrow Id(o_i) \cap Id(o_j) = \emptyset$],

   (c) Nechť $FLAT(S) = \bigcup_{o\in S} o$. $FLAT(S)$ je uzavřený systém instancí sítí.

3. Pro systém objektů S definujeme funkci $Oid_S(id)$ tak, že[17] $Oid_S(id) = oid$ právě tehdy, když existují značení m_1, m_2 a objekt $o \in S$ takové, že $(oid, m_1) \in o$ a $(id, m_2) \in o$ a $oid \in NAME_\Sigma$.

4. Pro systém objektů S definujeme funkci $Marking_S = Marking_{FLAT(S)}$.

5. Pro systém objektů S definujeme funkci $CMarking_S$ takto:[18]

   - $\forall id \in INST_\Sigma$ [$id = Oid_S(id) \Longrightarrow CMarking_S(id) = Marking_S(id)$],
   - $\forall id \in INST_\Sigma$ [$id \neq Oid_S(id) \Longrightarrow CMarking_S(id) = Marking_S(id)$
     $\cup\, Marking_S(Oid_S(id))$].

[17] Ke jménu instance sítě vrací jméno objektu, který tuto instanci sítě obsahuje.

[18] Funkce $CMarking_S(id)$ vrací tzv. úplné značení instance sítě, identifikované jménem id. Úplné značení sítě metody zahrnuje i značení míst, sdílených s příslušnou sítí objektu.

6. Pro systém objektů S definujeme bijekci $Object_S : Id(FLAT(S)) \cap NAME_\Sigma \longleftrightarrow S$ takto: $Object_S(oid) = o$ právě tehdy, když existuje $(oid, m) \in o$.

7. Množinu všech systémů objektů nad Σ označíme $SO(\Sigma)$.

8. Na množině $FLAT(S)$ definujeme **relaci viditelnosti** V_S takto:

$$\forall (i_1, i_2) \in FLAT(S) \times FLAT(S) \; [\; (i_1, i_2) \in V_S \Longleftrightarrow Id(i_2) \in (ref^P(i_1) \cup ref^T(i_1)) \;],$$

Definice 5.4.5 Počáteční systém objektů pro $OOPN = (\Sigma, c_0, oid_0)$ je systém objektů $S_0 = \{o_0\}$, kde $o_0 = \{(oid_0, PI_{ONET_{c_0}}\langle\emptyset\rangle)\}$. Objekt o_0 nazveme **prvotní objekt**.

Následuje definice funkce GC, která reprezentuje garbage-collector, odstraňující ty instance sítí, které nejsou (ani nepřímo) referencované z počátečního objektu. Tato funkce se uplatní při každém provedení přechodu v OOPN, jak bude ukázáno dále. V definici využijeme relaci viditelnosti a relaci zapouzdření z def. 5.4.4.[19]

Definice 5.4.6 Mějme $OOPN = (\Sigma, c_0, oid_0)$. **Garbage-collector** je funkce

$$GC : SO(\Sigma) \longrightarrow SO(\Sigma),$$

definovaná $\forall s \in SO(\Sigma)$ takto:

1. Nechť $i_0 \in FLAT(S)$ je instance sítě počátečního objektu, nechť $Id(i_0) = oid_0$.

2. $\forall o \in S \; [\; o \in S \wedge (o \cap (V_S^*(i_0))) \neq \emptyset \Longleftrightarrow (o \cap (V_S^*(i_0))) \in GC(S) \;]$.

5.5 Dynamika OOPN

OOPN definuje počáteční systém objektů, který se dynamicky vyvíjí prováděním přechodů Petriho sítí; mluvíme o evoluci systému objektů. Provádění přechodů je polymorfní: v závislosti na navázání proměnných a stavu systému se přechod může provést několika rozdílnými způsoby. Rozlišujeme a-provedení, n-provedení, f-provedení a j-provedení přechodu.

5.5.1 Kontext

Výrazy, definované v sekci 5.2.3 (zasílání zpráv ve strážích a akcích přechodů a hranové výrazy), se vyhodnocují při zjišťování proveditelnosti, resp. při provádění přechodů uvnitř instancí sítí. Sémantika hranového výrazu a primitivního zaslání zprávy byla dána definicí 5.2.4. Způsob vyhodnocení těchto výrazů nezávisí na okamžitém stavu systému, protože je vyhodnocován primitivními objekty, což jsou konstanty. Naproti tomu, vyhodnocení neprimitivního zaslání zprávy a synchronní komunikace závisí na okamžitém stavu systému. Tyto výrazy se vyhodnocují při zjišťování proveditelnosti a při provádění přechodů. Aby bylo možné tyto výrazy, které mohou obsahovat i pseudoproměnné `self` a `super`, vyhodnotit, definujeme *kontext* provádění přechodu.

Definice 5.5.1 Mějme systém tříd Σ. **Kontext** je dvojice (S, id), kde $S \in SO(\Sigma)$ a $id \in Id(FLAT(S))$. V kontextu (S, id) pro libovolné navázání b platí: $\texttt{self}\langle b\rangle = \texttt{super}\langle b\rangle = Oid_S(id)$.

[19] Použijeme následující notaci: Je-li R relace, R^* označuje její tranzitivně reflexivní uzávěr a R^{-1} je inverzní relace k relaci R.

5.5.2 Vyhodnocování stráží a synchronní komunikace

Stráž přechodu t v síti N může podle definice 5.3.1 obsahovat množinu výrazů,[20] z nichž každý je zaslání zprávy. Je-li možno pro nějaké navázání proměnných $InVar_N(t)$ vyhodnotit výrazy ve stráži přechodu jako `true` a jsou-li splněny další podmínky proveditelnosti přechodu, je dovoleno přechod provést, jinak je přechod v daném stavu neproveditelný.

Další podmínky proveditelnosti přechodu souvisí s případnou synchronní komunikací. Nyní definujeme vyhodnocení stráže přechodu a synchronní komunikaci. Vyhodnocování stráže přechodu probíhá takto: Ta část stráže volajícího přechodu, kterou lze vyhodnotit jako primitivní zasílání zpráv, je takto vyhodnocena. Dále se na základě neprimitivního zasílání zpráv ve stráži přechodu zjistí všechny synchronní porty a objekty, účastnící se synchronní komunikace. Vstupní podmínky všech těchto synchronních portů, včetně vstupních podmínek volajícího přechodu, se sečtou (jako multimnožiny) a otestuje se jejich splnitelnost. Je-li každé primitivní zaslání zprávy ve stráží volajícího přechodu i ve volaných synchronních portech vyhodnotitelné a vrátí-li `true` a všechny synchronní porty i volající přechod jsou současně proveditelné, prohlásíme přechod za *s-proveditelný*. Testování s-proveditelnosti se objeví jako součást testů jiných typů proveditelnosti přechodů, jak bude uvedeno dále.

Následující definice zavádí pomocnou funkci $Class$, kterou využijeme v dalších definicích.

Definice 5.5.2 Mějme systém tříd Σ. Pro libovolnou síť nebo synchronní port $x \in NET_\Sigma \cup SYNC_\Sigma$, literál e a navázání proměnných $b : V' \longrightarrow U$, $Var(e) \subseteq V'$, definujeme $Class(x, e, b)$ takto: Je-li $e =$ `super`, pak $Class(x, e, b) = Super_\Sigma(Def_\Sigma(x))$, jinak $Class(x, e, b) = Type(e\langle b\rangle)$.

Volání funkce $Class(x, e_0, b)$ se bude používat pro zjištění třídy adresáta zprávy, specifikované výrazem $e_0.msg(e_1, ..., e_n)$ při navázání b, přičemž uvedený výraz se nachází buď ve stráži synchronního portu x, nebo je součástí stráže nebo akce přechodu v síti x.

Následující definice zavede pomocnou funkci $SATISFIED$ pro synchronní porty. Tato funkce bude využita v def. 5.5.4, která zavádí pojem s-proveditelnost přechodu.

Definice 5.5.3 Mějme systém tříd Σ, kontext (S, oid), $oid \in NAME_\Sigma$, synchronní port $R \in SYNC_{Type(oid)}$ a navázání proměnných $b : V' \longrightarrow U$, $Var(R) \subseteq V'$. Nechť $GUARD_R^{Net(oid)} = \{G_1, ..., G_m\}$.

Je-li G_i, kde $i \in \{1, 2, ..., m\}$, při navázání b vyhodnotitelné primitivní zaslání zprávy (podle def. 5.2.4), přičemž $\forall i \in \{1, 2, ..., m\}$ [$G_i\langle b\rangle =$ `true`], pak $SATISFIED(S, oid, R, b) =$ `true`, jinak je $SATISFIED(S, oid, R, b) =$ `false`.

Definice 5.5.4 (Synchronní komunikace, s-proveditelnost přechodu.) Mějme systém tříd Σ, kontext (S, id), přechod $t \in Net(id)$ a navázání proměnných b.[21] Zavedeme tyto pomocné pojmy:

1. Nechť $GUARD_t^{Net(id)} = \{G_1, ..., G_m\}$, kde každý výraz G_i, $i \in \{1, 2, ..., m\}$, má tvar $e_0^i.msg^i(e_1^i, e_2^i, ..., e_n^i)$, přičemž bez ztráty obecnosti[22] předpokládáme, že $\exists k \in \{0, ..., m\}$ takové, že $\forall j \in \{1, ..., k\}$, platí, že G_j je při navázání b neprimitivní zaslání zprávy (podle def. 5.2.4) a současně $\forall j \in \{k + 1, ..., m\}$ platí, že G_j je při navázání b primitivní zaslání zprávy (podle def. 5.2.4).

2. Označme $oid_i = e_0^i\langle b\rangle$ a $R_i = Method_{Class(Net(id), e_0^i, b)}(msg^i)$, $\forall i \in \{1, ..., k\}$.

[20] Jde vlastně o konjunkci booleovských výrazů.

[21] Jde o navázání množiny proměnných, obsahující $InVar_{Net(id)}(t)$ a proměnné všech synchronních portů, volaných přechodem t, což upřesníme v bodu 3 této definice.

[22] Množinu výrazů ve stráži lze indexovat libovolným způsobem.

3. Abychom mohli snadno definovat předávání parametrů synchronním portům, předpokládejme (aniž bychom tím způsobili implementační obtíže), že pro každé volání synchronního portu R_i ze stráže G_i, tedy pro každou dvojici (G_i, R_i), implicitně existuje mapování (přejmenování) proměnných tohoto synchronního portu na proměnné, použité při jeho volání, $r_i : Var(R_i) \longrightarrow V_{(R_i,G_i)}$. Přitom množina proměnných $V_{(R_i,G_i)}$ respektuje předání parametrů: Je-li parametrem e^i_j volání $e^i_0.msg^i(e^i_1, e^i_2, ..., e^i_n)$ synchronního portu R_i proměnná z $Var(t)$, je tato proměnná prvkem $V_{(R_i,G_i)}$. Je-li tímto parametrem konstanta $c \in CONST \cup CLASS$, příslušným prvkem $V_{(R_i,G_i)}$ je unikátní proměnná v_c, reprezentující tuto konstantu, která je v b vždy na tuto konstantu navázána, tj. $b(v_c) = c$.[23]

4. Nechť $IOIDS$ označuje množinu jmen objektů, účastnících se synchronní komunikace (připomeňme zavedení R_j a oid_j v bodě 2): $IOIDS = \bigcup_{j \in \{1,...,k\}} \{oid_j\}$.

5. Nechť $IPORTS$ je funkce, která každému jménu objektu přiřazuje množinu synchronních portů (spolu s přejmenováním proměnných), účastnících se synchronní komunikace a patřících objektu uvedeného jména (připomeňme zavedení R_j a oid_j v bodě 2): $IPORTS(oid) = \{(R_j, r_j) | oid_j = oid\}$.

Definujeme:

1. Přechod t je v kontextu (S, id) pro navázání b **s-proveditelný**, jsou-li současně splněny tyto podmínky:[24]

   (a) $\forall j \in \{1, ..., k\}$ [$msg^j \in MSG_{Class(Net(id),e^j_0,b)} \wedge$
   $Method_{Class(Net(id),e^j_0,b)}(msg^j) \in SYNC_{Class(Net(id),e^j_0,b)}$],

   (b) $\forall j \in \{1, ..., k\}$ [$SATISFIED(S, oid_j, R_j, r_j \circ b) =$ `true`],[25]

   (c) $\forall j \in \{k+1, ..., m\}$ [$G_j\langle b \rangle =$ `true`],[26]

   (d) $\forall oid \in IOIDS\ \forall p \in P_{Net(oid)}$
   $[\sum_{(R,r) \in IPORTS(oid)} (COND_R^{Net(oid)}(p)\langle r \circ b \rangle + PRECOND_R^{Net(oid)}(p)\langle r \circ b \rangle)$
   $\leq Marking_S(oid)(p)$],

   (e) $\forall p \in P_{Net(Oid_S(id))}$
   $[\sum_{(R,r) \in IPORTS(Oid_S(id))} (COND_R^{Net(Oid_S(id))}(p)\langle r \circ b \rangle +$
   $PRECOND_R^{Net(Oid_S(id))}(p)\langle r \circ b \rangle)$
   $+ (COND_t^{Net(Oid_S(id))}(p)\langle b \rangle + PRECOND_t^{Net(Oid_S(id))}(p)\langle b \rangle)$
   $\leq Marking_S(Oid_S(id))(p)$].

[23] b je tedy navázání proměnných $InVar(t)$ a dalších proměnných, které odpovídají (po přejmenování) dalším parametrům všech synchronních portů, volaných přechodem t. Takovýto předpoklad nám umožní snadno zapsat předání parametrů jako složení funkcí $r \circ b$, kde r je přejmenování proměnných a b je navázání proměnných. Výsledkem je navázání proměnných volaného synchronního portu.

[24] (a) Každému neprimitivnímu zaslání zprávy ve stráži přechodu t odpovídá volání synchronního portu. (b) Všechny volané synchronní porty mají splněnou stráž. (c) Každé primitivní zaslání zprávy ve stráži přechodu t se vyhodnotí jako `true`. (d) Všechny volané synchronní porty jsou současně a bezkonfliktně proveditelné. (e) Všechny volané synchronní porty, patřící objektu, ve kterém testujeme s-proveditelnost přechodu t, jsou současně s přechodem t bezkonfliktně proveditelné.

[25] Notace $f \circ g$ má tento význam: $(f \circ g)(x) = g(f(x))$.

[26] Tato podmínka implicitně zahrnuje požadavek, aby G_j bylo při navázání b vyhodnotitelné primitivní zaslání zprávy.

2. Definujeme $SYNC(S, id, t, b)$ jako účinek synchronní komunikace, iniciované přechodem t v kontextu (S, id) při navázání b takto:
   Nechť $IOBJS = \{Object_S(oid) | oid \in IOIDS\}$. Pak

   $$SYNC(S, id, t, b) = (S - IOBJS) \cup IOBJS',$$

   kde množina $IOBJS'$ je definována takto:

   $$o \in IOBJS \Longleftrightarrow o' \in IOBJS',$$

   kde objekt o' je definován na základě objektu o takto:

   Nechť $oid = Object_S^{-1}(o)$.
   $o' = (o - \{(oid, M)\}) \cup \{(oid, M')\}$,

   kde značení M' je definováno na základě M takto:

   $$\forall tt \in T_{Net(oid)} \ [\ M'(tt) = M(tt)\ ],$$
   $$\forall p \in P_{Net(oid)} \ [\ M'(p) = M(p) - \sum_{(R,r) \in IPORTS(oid)} PRECOND_R^{Net(oid)}(p)\langle r \circ b\rangle + \sum_{(R,r) \in IPORTS(oid)} POSTCOND_R^{Net(oid)}(p)\langle r \circ b\rangle\ ].$$

5.5.3 Událost typu A – vnitřní událost objektu

Událost typu A (atomic) je atomické provedení přechodu uvnitř jednoho objektu, s tím, že synchronní komunikace může ovlivnit stav jiných objektů. Odebrání značek ze vstupních míst, provedení akce a uložení značek do výstupních míst přechodu se provede současně, jako nedělitelná operace.

Definice 5.5.5 Mějme $OOPN = (\Sigma, c_0, oid_0)$ a kontext (S, id).
Nechť $N = Net(id)$, $C = Type(Oid_S(id))$.

1. Přechod $t \in T_N$ je **a-proveditelný** v kontextu (S, id) pro navázání[27] b právě tehdy, když jsou současně splněny tyto podmínky:

   - přechod t je v kontextu (S, id) pro navázání b s-proveditelný,[28]
   - $\forall p \in P_N \ [\ (COND_t^N(p)\langle b\rangle + PRECOND_t^N(p)\langle b\rangle) \leq CMarking_S(id)(p)\ ]$,
   - $ACTION_t^N$ má tvar $y := expr$, kde $expr$ je pro navázání b vyhodnotitelné primitivní zaslání zprávy (podle def. 5.2.4).

   Nechť $b' = b \cup \{(y, r)\}$, kde $r = expr\langle b\rangle$.
   Nechť $SY = SYNC(S, id, t, b)$, $oid = Oid_{SY}(id)$, $o = Object_{SY}(oid)$.
   Nechť $M = CMarking_{SY}(id)$, $MM = Marking_{SY}(id)$ a $OM = Marking_{SY}(Oid_{SY}(id))$.

2. Je-li přechod $t \in T_N$ a-proveditelný v kontextu (S, id) pro navázání b, může být **a-proveden**, čímž se systém S změní na systém S', definovaný takto:

[27] Předpokládáme, že b je navázání množiny proměnných, obsahující $InVar_N(t)$ a proměnné všech synchronních portů, volaných přechodem t, viz definice s-proveditelnosti přechodu. Totéž platí také pro události typů N a F, kde se uplatňuje testování s-proveditelnosti přechodu.

[28] Stráž přechodu je splněna a pokud specifikuje synchronní komunikaci, lze ji uskutečnit (viz odstavec 5.5.2).

$$S' = GC((SY - \{o\}) \cup \{o'\}),$$

kde objekt o' je definován takto:

(a) Je-li $id = oid$, pak[29]:
$o' = (o - \{(oid, M)\}) \cup \{(oid, M')\}$,
kde M' je definováno takto:
$\forall tt \in T_N$ [$M'(tt) = M(tt)$],
$\forall p \in P_N$ [$M'(p) = M(p) - PRECOND_t^N(p)\langle b\rangle + POSTCOND_t^N(p)\langle b'\rangle$].

(b) Je-li $id \neq oid$, pak[30]:
$o' = (o - \{(id, MM), (oid, OM)\}) \cup \{(id, MM'), (oid, OM')\}$,
kde $MM' = M'|_{(P_N - P_{ONET_C}) \cup T_N}$ a $OM' = M'|_{P_{ONET_C} \cup T_{ONET_C}}$,
kde M' je definováno jako v (a).

3. Může-li být přechod $t \in T_N$ v kontextu (S, id) pro navázání b a-proveden, přičemž výsledným stavem je S', říkáme, že S' je **přímo dostupný** z S **a-provedením** t v instanci id pro navázání b, což zapisujeme

$$S[\mathrm{A}, id, t, b\rangle S'.$$

5.5.4 Událost typu N – vytvoření nového objektu

Událost typu N (new) je vytvoření nového objektu provedením přechodu. Přechod se provede atomicky, podobně jako v případě události typu A, ale vznikne přitom nový objekt.

Definice 5.5.6 Mějme $OOPN = (\Sigma, c_0, oid_0)$ a kontext (S, id).
Nechť $N = Net(id)$, $C = Type(Oid_S(id))$.

1. Přechod $t \in T_N$ je **n-proveditelný** v kontextu (S, id) pro navázání b právě tehdy, když jsou současně splněny tyto podmínky:

   - přechod t je v kontextu (S, id) pro navázání b s-proveditelný,
   - $\forall p \in P_N$ [$(COND_t^N(p)\langle b\rangle + PRECOND_t^N(p)\langle b\rangle) \leq CMarking_S(id)(p)$],
   - $ACTION_t^N$ má tvar $y := e_0.\texttt{new}$,
     kde y je proměnná a e_0 je term takový, že $e_0\langle b\rangle \in CLASS_\Sigma$. Označme $C_{new} = e_0\langle b\rangle$.
   - $\exists oid_{new} \in Dom(C_{new}) - Id(FLAT(S))$.[31]

   Nechť $ON = ONET_{C_{new}}$ a $b' = b \cup \{(y, oid_{new})\}$.
   Nechť $SY = SYNC(S, id, t, b)$, $oid = Oid_{SY}(id)$, $o = Object_{SY}(oid)$.
   Nechť $M = CMarking_{SY}(id)$, $MM = Marking_{SY}(id)$ a $OM = Marking_{SY}(Oid_{SY}(id))$.

2. Je-li přechod $t \in T_N$ n-proveditelný v instanci id pro navázání b v systému S, může být **n-proveden**, čímž se systém S změní na systém S', definovaný takto:

$$S' = GC((SY - \{o\}) \cup \{o', o_{new}\}),$$

[29] Jde o síť objektu.
[30] Jde o síť metody.
[31] Připouštíme zde recyklaci jmen objektů.

kde objekty o' a o_{new} jsou definovány takto:

- Objekt o' je definován takto[32]:
  (a) Je-li $id = oid$, pak[33]:
  $o' = (o - \{(oid, M)\}) \cup \{(oid, M')\}$,
  kde M' je definováno takto:
  $\forall tt \in T_N$ [$M'(tt) = M(tt)$],
  $\forall p \in P_N$ [$M'(p) = M(p) - PRECOND_t^N(p)\langle b\rangle + POSTCOND_t^N(p)\langle b'\rangle$].
  (b) Je-li $id \neq oid$, pak[34]:
  $o' = (o - \{(id, MM), (oid, OM)\}) \cup \{(id, MM'), (oid, OM')\}$,
  kde $MM' = M'|_{(P_N - P_{ONET_C}) \cup T_N}$ a $OM' = M'|_{P_{ONET_C} \cup T_{ONET_C}}$,
  kde M' je definováno jako v (a).
- Objekt o_{new} je definován takto:
  $o_{new} = \{(oid_{new}, M_{new})\}$,
  kde M_{new} je definováno takto:
  $\forall p \in P_{ON}$ [$M_{new}(p) = PI_{ON}(p)\langle\emptyset\rangle$],
  $\forall t \in T_{ON}$ [$M_{new}(t) = \emptyset$].

3. Může-li být přechod $t \in T_N$ v kontextu (S, id) pro navázání b n-proveden, přičemž výsledným stavem je S', říkáme, že S' je **přímo dostupný** z S **n-provedením** t v instanci id pro navázání b, což zapisujeme

   $S[\mathrm{N}, id, t, b\rangle S'$.

5.5.5 Událost typu F – předání zprávy

Událost typu F (fork) je předání zprávy neprimitivnímu objektu. Jde vlastně o vytvoření procesu pro odpovídající metodu. Je to neúplné provedení přechodu. Odeberou se značky ze vstupních míst a vytvoří se nová instance sítě odpovídající metody – to vše jako atomická operace. Očekává se, že někdy později dojde k události typu J (join), která přechod dokončí (zruší instanci sítě metody a umístí značky do výstupních míst přechodu).

Definice 5.5.7 Mějme $OOPN = (\Sigma, c_0, oid_0)$ a kontext (S, id).
Nechť $N = Net(id)$, $C = Type(Oid_S(id))$.

1. Přechod $t \in T_N$ je **f-proveditelný** v kontextu (S, id) pro navázání b právě tehdy, když jsou současně splněny tyto podmínky:

   - přechod t je v kontextu (S, id) pro navázání b s-proveditelný.
   - $\forall p \in P_N$ [$(COND_t^N(p)\langle b\rangle + PRECOND_t^N(p)\langle b\rangle) \leq CMarkings_S(id)(p)$].
   - $ACTION_t^N$ má tvar $y := expr$, kde y je proměnná a $expr$ je pro navázání b neprimitivní zaslání zprávy[35] tvaru $e_0.msg(e_1, e_2, ..., e_m)$, kde e_i jsou termy. Nechť $oid_r = e_0\langle b\rangle$ a $C_r = Class(N, e_0, b)$.
   - $msg^{(m)} \in MSG_{C_r}$.[36] Nechť $MN = Method_{C_r}(msg)$.

[32] Stejně jako při a-provedení.
[33] Jde o síť objektu.
[34] Jde o síť metody.
[35] Viz def. 5.2.4.
[36] Když objekt zprávě nerozumí, přechod je neproveditelný.

- $MN \in MNETS_{C_r}$.
- $\exists id' \in Inst(MN) - Id(FLAT(S))$.[37]
- $PP_{MN} = \{pp_1, ..., pp_m\}$.[38]

Nechť $SY = SYNC(S, id, t, b)$, $oid = Oid_{SY}(id)$, $o = Object_{SY}(oid)$.
Nechť $M = CMarking_{SY}(id)$, $MM = Marking_{SY}(id)$, $OM = Marking_{SY}(Oid_{SY}(id))$ a $o_r = Object_{SY}(oid_r)$.

2. Je-li přechod $t \in T_N$ f-proveditelný v kontextu (S, id) pro navázání b, může být **f-proveden**, čímž se systém S změní na systém S', definovaný takto:

   (a) Je-li $o \neq o_r$, pak[39]
   $$S' = GC((SY - \{o, o_r\}) \cup \{o', o'_r\}),$$
   kde o' a o'_r jsou definovány takto:
   - Objekt o' je definován takto[40]:
     i. Je-li $id = oid$, pak[41]:
     $$o' = (o - \{(oid, M)\}) \cup \{(oid, M')\},$$
     kde M' je definováno takto:
     $$\forall p \in P_N \ [\ M'(p) = M(p) - PRECOND_t^N(p)\langle b\rangle\ ],$$
     $$M'(t) = M(t) \cup \{(id', b)\},\ \forall tt \in T_N - \{t\} \ [\ M'(tt) = M(tt)\ ].$$
     ii. Je-li $id \neq oid$, pak[42]:
     $$o' = (o - \{(id, MM), (oid, OM)\}) \cup \{(id, MM'), (oid, OM')\},$$
     kde $MM' = M'|_{(P_N - P_{ONET_C}) \cup T_N}$ a $OM' = M'|_{P_{ONET_C} \cup T_{ONET_C}}$,
     kde M' je definováno jako v (i).
   - Objekt o'_r je definován takto:
     $$o'_r = o_r \cup \{(id', M'')\},$$
     kde značení M'' je definováno takto:
     $$\forall p \in P_{MN} - P_{ONET_{C_r}} - PP_{MN} \ [\ M''(p) = PI_{MN}(p)\langle\emptyset\rangle\ ],$$
     $$\forall i \in \{1, ..., m\} \ [\ M''(pp_i) = PI_{MN}(pp_i)\langle\emptyset\rangle + e_i\langle b\rangle\ ].$$

   (b) Je-li $o = o_r$, pak[43]
   $$S' = GC((SY - \{o\}) \cup \{o'\}),$$
   kde o' je definován takto:
   i. Je-li $id = oid$, pak[44]:
   $$o' = (o - \{(oid, M)\}) \cup \{(oid, M'), (id', M'')\},$$
   kde M' a M'' jsou definovány jako v (a).
   ii. Je-li $id \neq oid$, pak[45]:
   $$o' = (o - \{(id, MM), (oid, OM)\}) \cup \{(id, MM'), (oid, OM'), (id', M'')\}\},$$
   kde MM', OM' a M'' jsou definovány jako v (a).

[37] Připouštíme zde recyklaci jmen instancí sítí metod.

[38] pp_i jsou formální parametry, které budou navázány na skutečné parametry $e_i\langle b\rangle$; jejich počty se shodují.

[39] Adresátem zprávy není `self` (ať už přímo, tj. adresát je identifikovaný pseudoproměnnou `self` nebo `super`, či nepřímo, tj. adresát je identifikovaný proměnnou, navázanou na stejnou hodnotu jako `self`).

[40] o' je definován podobně jako při a-provedení, ale neuplatňuje se $POSTCOND_t^N(p)\langle b'\rangle$ a modifikuje se značení přechodu t.

[41] Jde o síť objektu.

[42] Jde o síť metody.

[43] Adresátem zprávy je `self` (ať už přímo či nepřímo) nebo `super`.

[44] Jde o síť objektu.

[45] Jde o síť metody.

3. Může-li být přechod $t \in T_N$ v kontextu (S, id) pro navázání b f-proveden, přičemž výsledným stavem je S', říkáme, že S' je **přímo dostupný** z S **f-provedením** t v instanci id pro navázání b, což zapisujeme

$$S[\mathrm{F}, id, t, b\rangle S'.$$

5.5.6 Událost typu J – akceptování odpovědi na zprávu

Událost typu J (join) je předání výsledku dokončené metody volajícímu přechodu a jeho dokončení. Jde vlastně o ukončení procesu, dříve vytvořeného událostí typu F. Zruší se instance sítě odpovídající metody a uloží se značky do výstupních míst přechodu – to vše jako atomická operace.

Definice 5.5.8 Mějme $OOPN = (\Sigma, c_0, oid_0)$ a kontext (S, id).
Nechť $N = Net(id)$, $C = Type(Oid_S(id))$.

1. Přechod $t \in T_N$ je **j-proveditelný** v kontextu (S, id) pro navázání $b' : Var_N(t) \longrightarrow U$ právě tehdy, když

$$\exists (pid, b) \in M(t)\ \exists r \in U\ [\ Marking_S(pid)(RP_{Net(pid)})(r) > 0 \wedge b' = b \cup \{(y, r)\}\ ].^{46}$$

   Nechť $o_r = Object_S(Oid_S(pid))$, $oid = Oid_S(id)$, $o = Object_S(oid)$.
   Nechť $M = CMarking_S(id)$, $MM = Marking_S(id)$ a $OM = Marking_S(Oid_S(id))$.

2. Je-li přechod $t \in T_N$ j-proveditelný v kontextu (S, id) pro navázání b', může být **j-proveden**, čímž se systém S změní na systém S', definovaný takto:

   (a) Je-li $o \neq o_r$, pak[47]

   $$S' = GC((S - \{o, o_r\}) \cup \{o', o'_r\}),$$

   kde o' a o'_r jsou definovány takto:

   - Objekt o'_r je definován takto:
     $o'_r = o_r - \{(pid, Marking_S(pid))\}$.
   - Objekt o' je definován takto[48]:
     i. Je-li $id = oid$, pak[49]:
     $o' = (o - \{(oid, M)\}) \cup \{(oid, M')\}$,
     kde M' je definováno takto:
     $\forall p \in P_N\ [\ M'(p) = M(p) + POSTCOND_t^N(p)\langle b' \rangle\ ]$,
     $M'(t) = M(t) - \{(pid, b)\}$, $\forall tt \in T_N - \{t\}\ [\ M'(tt) = M(tt)\ ]$.
     ii. Je-li $id \neq oid$, pak[50]:
     $o' = (o - \{(id, MM), (oid, OM)\}) \cup \{(id, MM'), (oid, OM')\}$,
     kde $MM' = M'|_{(P_N - P_{ONET_C}) \cup T_N}$ a $OM' = M'|_{P_{ONET_C} \cup T_{ONET_C}}$,
     kde M' je definováno jako v (i).

   (b) Je-li $o = o_r$, pak[51]

[46] b je navázání proměnných $InVar_N(t)$.
[47] Adresátem zprávy nebyl `self` (ať už přímo či nepřímo) ani `super`.
[48] o' je definován podobně jako při a-provedení, ale neuplatňuje se $PRECOND_t^N(p)\langle b \rangle$ a modifikuje se značení přechodu t.
[49] Jde o síť objektu.
[50] Jde o síť metody.
[51] Adresátem zprávy byl `self` (ať už přímo či nepřímo) nebo `super`.

$$S' = GC((S - \{o\}) \cup \{o'\}),$$

kde o' je definován takto:

i. Je-li $id = oid$, pak[52]:
$o' = (o - \{(oid, M), (pid, Marking_S(pid))\}) \cup \{(oid, M')\}$,
kde M' je definováno jako v (a).

ii. Je-li $id \neq oid$, pak[53]:
$o' = (o - \{(id, MM), (oid, OM), (pid, Marking_S(pid))\})$
$\cup \{(id, MM'), (oid, OM')\}$,
kde MM' a OM' jsou definovány jako v (a).

3. Může-li být přechod $t \in T_N$ v kontextu (S, id) pro navázání b' j-proveden, přičemž výsledným stavem je S', říkáme, že S' je **přímo dostupný** z S **j-provedením** t v instanci id pro navázání b', což zapisujeme

$$S[\mathrm{J}, id, t, b'\rangle S'.$$

5.5.7 Stavový prostor OOPN

Systém objektů reprezentuje okamžitý stav systému, popsaného OOPN. Stavový prostor OOPN je množina všech stavů, které OOPN může potenciálně dosáhnout z počátečního stavu.

Definice 5.5.9 Mějme $OOPN = (\Sigma, c_0, oid_0)$, nechť $S_0 = \{\{(oid_0, PI_{ONET_{c_0}}\langle\emptyset\rangle)\}\}$ je její počáteční systém objektů (podle def. 5.4.5).

1. Množina všech událostí $EV(\Sigma)$ je možina čtveřic (e, id, t, b), takových, že
   (a) $e \in \{\mathrm{A}, \mathrm{N}, \mathrm{F}, \mathrm{J}\}$ je typ události,
   (b) $id \in INST_\Sigma$ je instance sítě, ve které k události došlo,
   (c) $t \in T_{Net(id)}$ je přechod, který událost vyvolal,
   (d) b je navázání proměnných přechodu t, pro které k události (e, id, t, b) došlo.
2. **Stavový prostor** X objektově orientované Petriho sítě $OOPN$ s počátečním systémem objektů S_0 je nejmenší množina systémů objektů, taková, že
   (a) $S_0 \in X$,
   (b) $S \in X \ \wedge \ (e, id, t, b) \in EV(\Sigma) \ \wedge \ S[e, id, t, b\rangle S' \implies S' \in X$.

5.6 Shrnutí

Definovali jsme objektově orientovanou Petriho síť, její syntax i sémantiku. Vytvořený formalismus umožňuje modelovat aktivní objekty, které reagují na příchozí zprávu vytvořením nové instance metody. Objekty jsou instancemi tříd. Třídy mohou být vytvářeny jako odvozené z jiných tříd pomocí dědičnosti, přičemž připouštíme pouze jednoduchou dědičnost (v souladu se Smalltalkem). Některé objekty (např. čísla, symboly, řetězce atd.) jsou považovány za primitivní – jsou konstantní, nemění stav a nevyvíjejí žádnou aktivitu. Jejich metody jsou funkce (bez vedlejších účinků, pro stejné vstupy dávají vždy stejné výsledky). Primitivní objekty jsou jazykově dostupné jako literály.

[52] Jde o síť objektu.
[53] Jde o síť metody.

Dědičnost v OOPN zahrnuje dědičnost reprezentace objektu (reprezentací objektu rozumíme síť objektu) a dědičnost metod. Tím je umožněna inkrementální specifikace tříd v hierarchii dědičnosti. Diskutabilní otázkou je potenciální možnost specifikovat i reprezentaci metod inkrementálně, s využitím dědičnosti sítí metod. Tato možnost nebyla v definici OOPN explicitně vyjádřena ani využita, i když ji lze snadno zavést. Další otázkou je násobná dědičnost. Definici OOPN by bylo možné modifikovat tak, aby umožnila násobnou dědičnost. V tomto případě jsme zůstali pouze u jednoduché dědičnosti, a to především z důvodu zachování jednoduchosti formalismu.

Dynamika OOPN je definována událostmi čtyř typů, A, N, F a J. Součástí událostí typů A, N a F může být synchronní komunikace. Součástí všech událostí je garbage-collecting. V příloze B (str. 125) je dynamika OOPN demonstrována na konkrétním příkladu.

Závěrem lze říci, že formální definice OOPN představuje rozumný kompromis mezi snahou definovat všechny potřebné výkonné strukturovací a synchronizační mechanismy a snahou udržet definici relativně jednoduchou a přehlednou.

Kapitola 6

Jazyk a systém PNtalk

V této kapitole bude popsán jazyk a systém, založený na objektově orientovaných Petriho sítích. Jeho jméno PNtalk je odvozeno z *"Petri Nets & Smalltalk"*.

Jazyk PNtalk je konkrétní implementací OOPN. Vychází přímo z definice OOPN, má však některé syntaktické odlišnosti, inspirované jazykem Smalltalk. PNtalk také konkretizuje některé skutečnosti, ve kterých definice OOPN ponechává určitou volnost. Jde především o primitivní objekty a hierarchii dědičnosti tříd. PNtalk také přináší oproti definici OOPN některá rozšíření, která umožňují snadněji zapisovat některé složitější konstrukce, jako jsou složené zprávy, seznamy a konstruktory. Vesměs jde jen o syntaktická vylepšení.

6.1 Jazyk PNtalk

Programování v PNtalku spočívá ve vytváření tříd neprimitivních objektů a jejich zařazování do hierarchie dědičnosti. Třídy jsou definovány množinami Petriho sítí, skládajících se z míst a přechodů, propojených hranami. Místa, přechody i hrany mohou mít přiřazeny popisy. Nejdříve uvedeme syntax popisů prvků sítí (inskripčního jazyka) a pak přejdeme k sítím a třídám.

6.1.1 Termy

Termy jsou nejjednodušší výrazy PNtalku. Jazykově reprezentují objekty PNtalku. Patří sem:

1. *Literály.* Literály identifikují významné[1] primitivní objekty PNtalku:

   (a) *Čísla.* Čísla jsou objekty, které reprezentují číselné hodnoty a reagují na zprávy, které požadují výsledky matematických operací. Literál, reprezentující číslo, je sekvence číslic, jíž může předcházet znaménko mínus a může za ní následovat desetinná tečka a další sekvence číslic. Lze též použít notaci s exponentem, který je uvozen znakem `e`. Příklady čísel jsou `5`, `456`, `-25`, `0.005`, `-12.0`, `1.345e5`, `0.33e-20`.

   (b) *Znaky.* Znaky jsou objekty, reprezentující jednotlivé symboly abecedy. Jejich literály jsou uvozeny symbolem dolar, například `$a`, `$B`, `$+`, `$$`, `$7`.

   (c) *Řetězce.* Řetězce jsou objekty, reprezentující sekvence znaků. Řětězce reagují na zprávy, požadující přístup k jednotlivým znakům a porovnání s jinými řetězci. Literál, reprezentující řetězec, je sekvence znaků, uzavřená v apostrofech. Apostrof uvnitř řetězce musí být zdvojen. Příklady řetězcových literálů jsou `'abcd'`, `'can''t'`.

[1] Ne všechny primitivní objekty jsou jazykově dostupné jako literály.

(d) *Symboly.* Symboly jsou objekty, typicky používané jako jména. Symbol je reprezentován sekvencí znaků s prefixem `#`, například `#abc`, `#B52`, `#'3x'`. Je zaručeno, že dva stejně zapsané symboly reprezentují tentýž objekt (na rozdíl od řetězců).

(e) *Booleovské konstanty.* Jsou reprezentovány vyhrazenými identifkátory `true` a `false`.

(f) *Nedefinovaný objekt.* Je reprezentován vyhrazeným identifikátorem `nil`. Například všechny neinicializované proměnné jsou automaticky nastaveny na hodnotu `nil`.

Implementace PNtalku uvnitř systému Smalltalk (PNtalk jako transparentní vrstva nad Smalltalkem) připouští i další literály, pole a blok (jejich syntax lze nalézt v [GR83]), pro zajištění plné konzistence s jazykem Smalltalk. Také se nepožaduje, aby primitivní objekty byly konstanty.[2] Tyto skutečnosti však nejsou pro další výklad významné.

2. *Proměnné.* Proměnné mohou v průběhu výpočtu reprezentovat různé objekty – jejich hodnotu lze programově ovlivňovat. Jsou to identifikátory (sekvence znaků, začínající písmenem) s malým počátečním písmenem, například `x`, `y`, `anInteger`. Některé identifikátory jsou rezervovány (`true`, `false`, `nil`).

3. *Pseudoproměnné.* Existují dvě pseudoproměnné, `self` a `super`. Jejich hodnota závisí na kontextu (kontextem rozumíme místo v programu a okamžitý stav výpočtu)[3] a není programově ovlivnitelná.

4. *Jména tříd.* Identifikátory s velkým počátečním písmenem jsou vyhrazeny pro jména tříd, například `Object, PN, C1`. Jde o konstanty, v průběhu výpočtu se nemění.

6.1.2 Zasílání zpráv

Zaslání zprávy je výraz, který se může objevit jako součást stráže i akce přechodu. Speciálním případem zaslání zprávy je term. Jinak má zaslání zprávy syntax

⟨*adresát*⟩ ⟨*zpráva*⟩.

Zpráva sestává ze selektoru a případných argumentů. Podle tvaru selektoru zprávy rozlišujeme tři druhy zpráv:

1. *Unární zpráva.* Selektor zprávy je identifikátor, který neobsahuje dvojtečku. Unární zpráva nemá žádné argumenty. Příklady zaslání unární zprávy jsou `1 factorial` a `C new`.

2. *Binární zpráva.* Selektor zprávy je jeden z těchto symbolů:

   `+ - / * = < > ~= <= >= & | // \\ , == ~==`

   Následuje jeden argument. Příklad zaslání binární zprávy je „`1 + 2`“, kde „`1`“ je adresát zprávy „`+ 2`“, přičemž „`+`“ je selektor zprávy a „2“ je argument.

3. *Zpráva s klíčovými slovy.* Zpráva obsahuje jeden nebo více klíčů, což jsou identifikátory, zakončené dvojtečkou, s jejich vlastními argumenty. Příklad zaslání zprávy s klíčovými slovy je „`anObject at: 1 put: #e`“. Zde je „`anObject`“ adresátem zprávy „`at: 1 put: #e`", kde „`at:put:`“ je selektor zprávy a „`1`“ a „`#e`“ jsou argumenty.

[2] Pokud primitivní objekt (objekt, definovaný jinak než Petriho sítěmi) není konstanta, je třeba tuto skutečnost zohlednit v teoretických úvahách o OOPN.

[3] Sémantika pseudoproměnných `self` a `super` je táž jako v jazyce Smalltalk a byla vysvětlena v kapitole 5.

Jsou-li adresát zprávy i argumenty termy, jde o *jednoduché zaslání zprávy*. PNtalk připouští i *složené zaslání zprávy*, kde jako příjemce nebo argument zprávy je uvedeno opět zaslání zprávy. Složené zaslání zprávy se vyhodnocuje v tomto pořadí:

1. Unární zprávy, zleva doprava.
2. Binární zprávy, zleva doprava.
3. Zprávy s klíčovými slovy, zleva doprava.

Pořadí vyhodnocování zpráv lze ovlivnit použitím závorek. Příkladem složeného zaslání zprávy je

```
1.5 tan rounded,
```

kde je zpráva `tan` zaslána číslu `1.5` a výsledku tohoto vyhodnocení je zaslána zpráva `rounded`. Jiným příkladem je

```
a + (b * 2).
```

Zde je nutné použití závorek, aby se násobení provedlo přednostně.

Detailní popis sémantiky všech zpráv, kterým rozumějí primitivní objekty, lze nalézt v popisu jazyka Smalltalk (např. [GR83, HH95]). Pro další výklad (především pro porozumění příkladům v kapitole 7) uvedeme jen nejdůležitější zprávy, kterým rozumí čísla a booleovské hodnoty. Každý objekt rozumí zprávám `==` (rovnost identity), `~==` (nerovnost identity), `=` (rovnost), `~=` (nerovnost). Číselné hodnoty rozumějí zprávám `>` (větší než), `>=` (větší nebo rovno), `<` (menší než), `<=` (menší nebo rovno), `+` (sčítání), `-` (odčítání), `*` (násobení), `/` (dělení), `//` (celočíselné dělení), `\\` (zbytek po celočíselném dělení), `abs` (absolutní hodnota), `negated` (opačná hodnota), `truncate` (odseknutí necelé části), `rounded` (zaokrouhlení). Booleovské hodnoty rozumějí zprávám `&` (logický součin), `|` (logický součet), `not` (negace).

6.1.3 Inskripce přechodů

Zasílání zpráv (pod pojmem zaslání zprávy budemem nadále rozumět obecně složené zaslání zprávy) se vyskytuje ve výrazech, které specifikují stráže a akce přechodů.

Stráž přechodu je tvořena sekvencí výrazů, oddělených tečkou, přičemž každý dílčí výraz je zaslání zprávy. Příklad stráže je

```
o state: x. x >= 50.
```

Sekvence výrazů ve stráži má význam konjunkce výsledků dílčích výrazů. Složené zaslání zprávy je ekvivalentní sekvenci výrazů (s pomocnými proměnnými pro mezivýsledky).

Akce přechodu (nikoliv stráž) může obsahovat výraz *přiřazení*. Tento výraz má tvar

⟨*proměnná*⟩ `:=` ⟨*zaslání zprávy*⟩,

například `z := x + y`. Rozšíření PNtalku oproti definici zde připouští sekvenci takovýchto výrazů, která má význam sekvenčního provádění, tj. je ekvivalentní množině kauzálně svázaných přechodů (výstupem prvního přechodu je místo, které je vstupem dalšího přechodu atd.), z nichž každý má jen jeden přiřazovací výraz.

Rozsah platnosti jmen proměnných zahrnuje stráž, akce a výrazy na okolních hranách přechodu. U míst zahrnuje počáteční značení a počáteční akci.

6.1.4 Hranové výrazy

Hranové výrazy se vyskytují na hranách grafů Petriho sítí a jako specifikace počátečního značení míst. Hranové výrazy reprezentují multimnožiny a mají tvar

$$n_1\text{`}c_1\texttt{ + }n_2\text{`}c_2\texttt{ + ... + }n_m\text{`}c_m,$$

kde n_i je term,[4] a c_i je term nebo seznam. Seznam má tvar

$$\texttt{(}e_1\texttt{, }e_2\texttt{, ..., }e_m\texttt{)},$$

kde e_i je term nebo seznam. PNtalk připouští i Prologovský zápis seznamu

$$\texttt{(}h_1\texttt{, }h_2\texttt{, ..., }h_m\texttt{| }t\texttt{)},$$

kde h_i reprezentují prvky na začátku seznamu a t reprezentuje jeho zbytek.

Na rozdíl od definice se v zápisu multimnožiny místo symbolu `+` používá čárka. Je-li v zápisu multimnožiny koeficient $n_i = 1$, může být vynechán. Příkladem hranového výrazu je

```
2`#e, x`y, 5`(x, y, $a), 6`7.
```

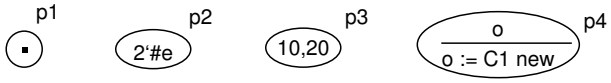
Tento výraz reprezentuje multimnožinu, obsahující dva výskyty symbolu `#e`, `x` výskytů (kde `x` je proměnná) objektu, referencovaného obsahem proměnné `y`, pět výskytů trojice `(x, y, $a)`, a šest výskytů čísla `7`.

Prázdnou multimnožinu můžeme zapsat například jako `0`#e`. Symbol `#e` většinou používáme v roli „anonymní, černé, resp. bezbarvé značky“. Jinou možností, vhodnou pro tento účel, je literál `nil`. V grafické reprezentaci sítí můžeme použít • (černou tečku) jako synonymum pro `nil`.

6.1.5 Místa, přechody a hrany

Sítě se v PNtalku specifikují graficky. Pro potřeby strojového zpracování existuje i možnost textové specifikace sítí, která je uvedena v příloze A.

Místo Petriho sítě je v PNtalku reprezentováno kružnicí nebo elipsou. Každé místo má jméno (pokud jméno není specifikováno programátorem, je při překladu přiděleno automaticky) a může mít specifikováno *počáteční značení*. Počáteční značení místa syntakticky odpovídá hranovému výrazu. PNtalk rozšiřuje definici OOPN o možnost použít v počátečním značení místa proměnné, přičemž vyčíslení hodnot proměnných je úkolem *počáteční akce* místa. Počáteční akce místa je syntakticky totožná s akcí přechodu. Příklady míst jsou na obr. 6.1. Místo `p4` má počáteční značení `o` (`o` je proměnná) a počáteční akci `o := C1 new` (vysvětlení sémantiky je na obr. 6.2).

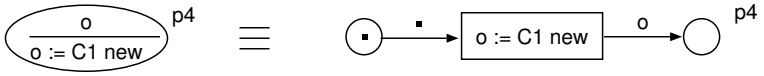


Obrázek 6.1: Příklady míst s počátečním značením.

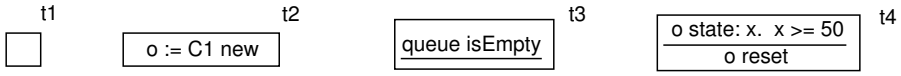
Přechod je v PNtalku reprezentován čtyřúhelníkem. Každý přechod má jméno (pokud jméno není specifikováno programátorem, je při překladu přiděleno automaticky) a může mít specifikovánu *stráž* a *akci*, jejichž syntax i sémantika byly vysvětleny v 6.1.3. Příklady přechodů jsou na obr. 6.3. Stráž přechodu je vždy podtržena.

Místa a přechody mohou být propojeny *hranami*. Ke každé hraně je připojen *hranový výraz* (viz 6.1.4). Z hlediska přechodu lze rozlišit tři typy hran:

[4] Předpokládá se, že se vyhodnotí jako nezáporné celé číslo. Jinak je jeho hodnota považována za nulovou.



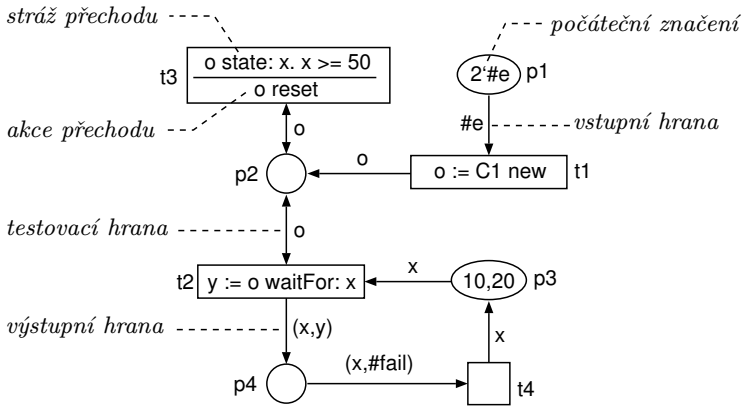
Obrázek 6.2: Sémantika počáteční akce místa p4 z obr. 6.1.



Obrázek 6.3: Příklady přechodů.

1. *Vstupní hrana*, →□, orientovaná z místa do přechodu, reprezentuje vstupní podmínku přechodu, specifikující značky, odebírané přechodem z příslušného místa.
2. *Výstupní hrana*, ←□, orientovaná z přechodu do místa, reprezentuje výstupní podmínku přechodu, specifikující značky, umisťované přechodem do příslušného místa.
3. *Testovací hrana*, ↔□, obousměrná hrana, reprezentuje podmínky přechodu (podmínky bez přívlastku), kterými přechod pouze testuje přítomnost značek v příslušném místě.

Rozsah platnosti jmen míst a přechodů je omezen sítí, v níž se vyskytují. Výjimkou je sdílení míst sítě objektu sítí metody. Všechna místa sítě objektu jsou viditelná i v síti objektu. Jména uzlů sítí se uplatňují také při inkrementální specifikaci sítí (využití dědičnosti), jak bude vysvětleno dále.



Obrázek 6.4: Demonstrace míst, přechodů a hran v PNtalku.

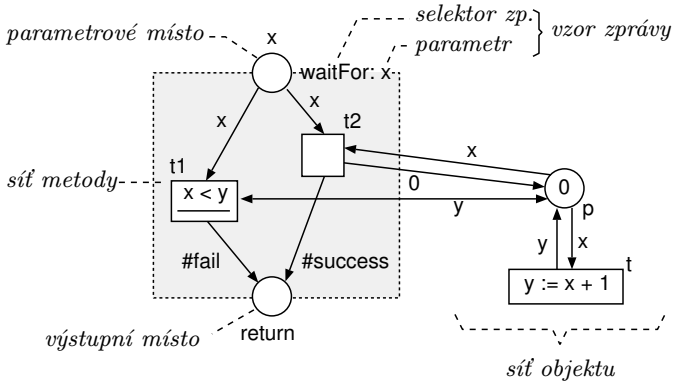
6.1.6 Sítě

Místa a přechody, propojené hranami, tvoří *síť*. Sítě jsou součástí specifikace tříd objektů, komunikujících předáváním zpráv. PNtalk rozlišuje dva druhy sítí:

- *Síť objektu* reprezentuje atributy objektu (v podobě míst) a jeho vlastní (implicitní) aktivitu. Příklad sítě objektu je na obr. 6.4.

- *Síť metody* specifikuje reakci objektu na příchozí zprávu. Definice třídy může obsahovat několik sítí metod. Síť metody sestává z míst a přechodů, propojených hranami, stejně jako síť objektu, ale ke každé síti metody je přiřazen *vzor zprávy*, jejíž přijetí objektem vyvolá dynamické vytvoření instance sítě metody. Vzor zprávy je složen ze *selektoru zprávy* a formálních parametrů. Jistá podmnožina (i prázdná podmnožina) míst sítě objektu jsou *parametrová místa*, která slouží k předání parametrů při vyvolání metody. Jejich jména musí odpovídat jménům formálních parametrů ve vzoru zprávy. Každá síť metody obsahuje jedno *výstupní místo* (pojmenované `return`), které slouží k předání výsledku vyhodnocení zprávy volajícímu objektu.

Sítě metod mohou být v rámci definice třídy propojeny se sítí objektu hranami mezi místy sítě objektu a přechody sítě metody. Tímto způsobem může metoda přistupovat k datům objektu a případně je měnit.



Obrázek 6.5: Příklad sítě metody.

Obr. 6.5 demonstruje prvky sítě metody a její propojení se sítí objektu. Poznamenejme že na rozdíl od formální definice OOPN jsou v PNtalku sdílená místa specifikována jen v síti objektu. Skutečnost, že se podle definice vyskytují i v sítích metod, je implicitní.

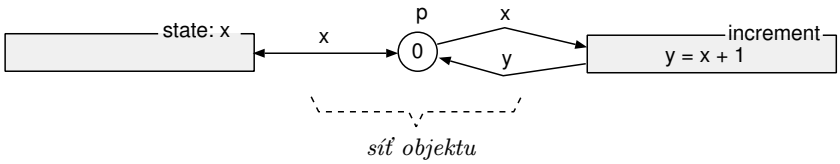
6.1.7 Synchronní porty

Kromě sítě objektu a sítí metod jsou součástí specifikace tříd i *synchronní porty*. Synchronní porty mají charakter přechodů i metod. Tomu odpovídá i jejich grafická podoba v PNtalku. Nejsou to sítě, takže neobsahují místa a přechody. Synchronní port, podobně jako přechod, může být propojen hranami s místy sítě objektu a může obsahovat stráž. K synchronnímu portu je, podobně jako k síti metody, připojen vzor zprávy, na kterou reaguje, a může být volán zasláním zprávy ze stráže libovolného přechodu.

Synchronní porty slouží k testování a případné změně stavu volaného objektu. Mohou být volány s předem vyhodnocenými parametry, ale je též možné volat je s volnými (nenavázanými) proměnnými.[5] Synchronní port může být v daném stavu buď proveditelný, nebo neproveditelný pro určité navázání proměnných. Hledání vhodného navázání proměnných probíhá (s ohledem na hodnoty formálních parametrů synchronního portu) stejně, jako v případě zjišťování proveditelnosti přechodů.

Speciálním případem synchronního portu je *predikát*, který neovlivňuje stav objektu (je propojen s místy sítě objektu pouze testovacími hranami). Příklady synchronních portů jsou na obr. 6.6. Stráž synchronního portu nemusí být podtržena.

[5] Připoměňme, že synchronní porty se volají ze stráží přechodů.



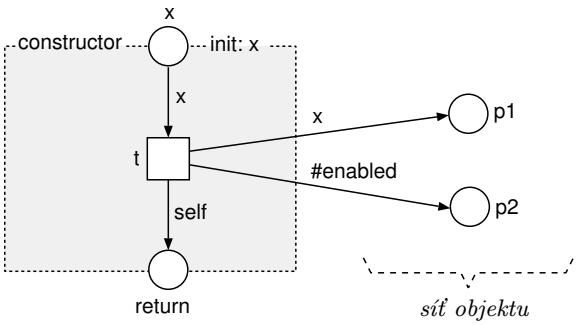
Obrázek 6.6: Synchronní porty `state:` a `increment` (`state:` je predikát).

6.1.8 Konstruktory

Vytvoření objektu se v PNtalku realizuje zasláním zprávy `new` příslušné třídě (třídy budou popsány později). Jména tříd jsou globálně dostupná. Implicitní konstruktor `new` je pevně zabudován do jazyka. Zprávě `new` rozumí každá třída a jako reakci na ni vytvoří instanci třídy, inicializovanou počátečním značením sítě objektu.

Pro dodatečnou a případně parametrizovanou inicializaci objektů slouží speciální metody nazvané (neimplicitní) *konstruktory*. Syntakticky jsou podobné ostatním metodám objektu, ale jako součást jejich specifikace se objevuje klíčové slovo `constructor`.

Je-li v rámci definice třídy specifikován konstruktor, příslušné zprávě pak rozumí jak třída, tak její instance. Třída na tuto zprávu reaguje tak, že nejprve vytvoří instanci implicitním konstruktorem `new` a poté zašle stejnou zprávu takto vzniklému objektu. Objekt na tuto zprávu reaguje standardním způsobem. Konstruktor vždy vrací hodnotu `self`, tj. jméno vytvořeného objektu. Příklad velmi jednoduchého konstruktoru, umisťujícího značky do míst sítě objektu, je na obr. 6.7.



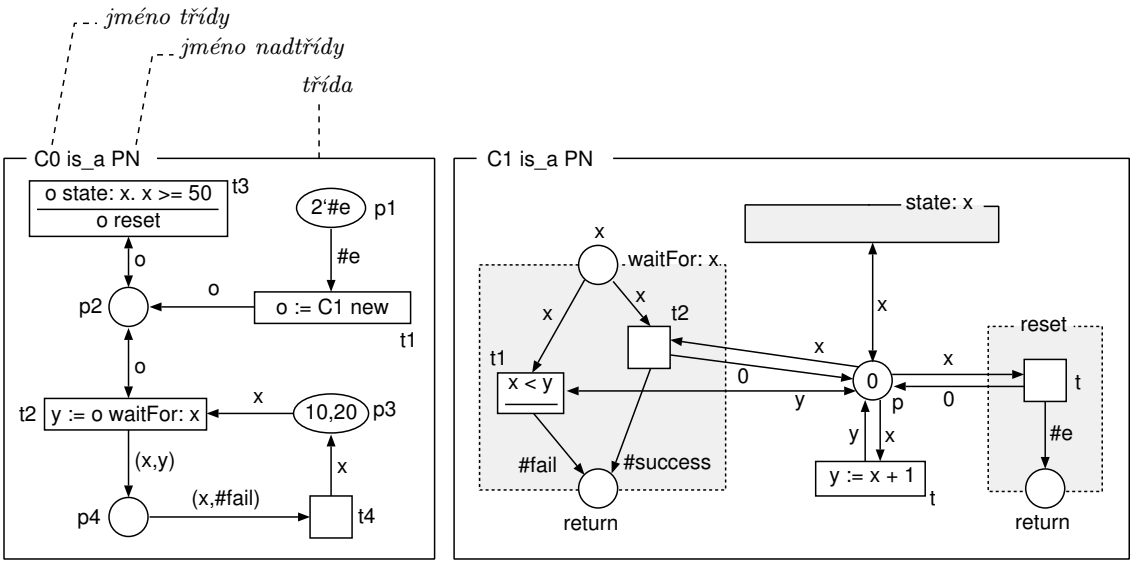
Obrázek 6.7: Příklad konstruktoru.

6.1.9 Třídy a dědičnost

Model v PNtalku sestává z množiny *tříd*. Jedna z nich je deklarována jako *počáteční třída*. Ta je implicitně instanciována při spuštění (zahájení simulace) modelu.

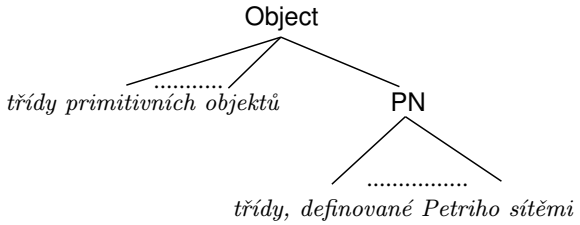
Třída je složena ze sítě objektu, množiny sítí metod, konstruktorů a synchronních portů. Příklad OOPN, skládající se ze dvou tříd, je na obr. 6.8.

Každá třída OOPN má právě jednoho předchůdce v hierarchii dědičnosti. Vrchol hierarchie tříd, definovaných Petriho sítěmi, je třída `PN`, která sama už není popsána Petriho sítěmi (její síť objektu je prázdná) a je přímým následníkem třídy `Object`, která tvoří vrchol hierarchie všech tříd PNtalku (viz obr. 6.9). Kromě (abstraktní) třídy `PN` jsou dalšími následníky (abstraktní) třídy `Object` též třídy primitivních objektů PNtalku, tj. konstant, z nichž nejběžněji používané jsou jazykově dostupné jako literály PNtalku. Abstraktní třída `Object` definuje reakce na zprávy,



Obrázek 6.8: Dvě třídy, tvořící OOPN

kterým musí rozumět všechny objekty. Jde především o porovnání identity objektů (zprávy `==` a `~==`).



Obrázek 6.9: Hierarchie dědičnosti tříd v PNtalku

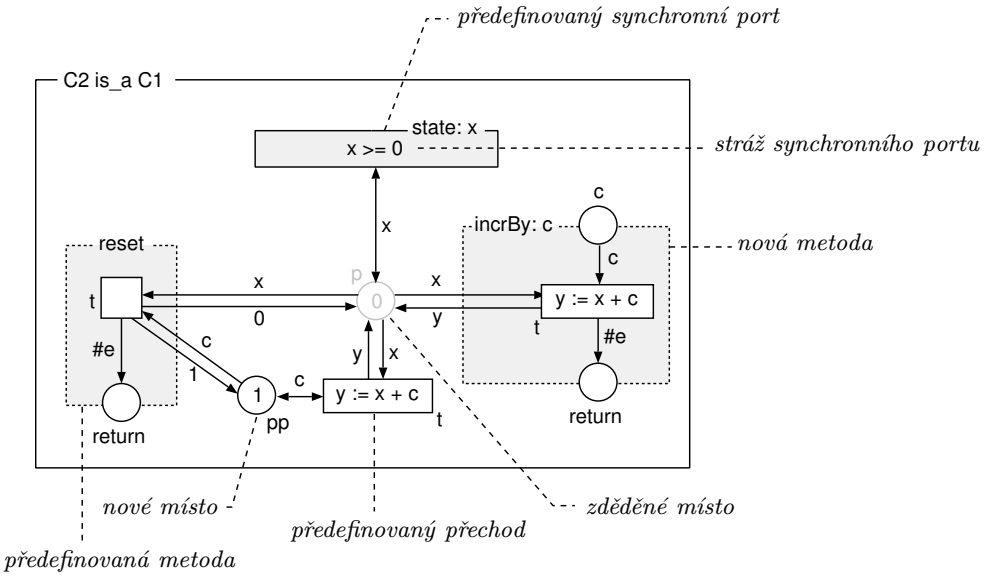
Každá třída dědí od své nadtřídy (předchůdce v hierarchii dědičnosti) strukturu sítě objektu a všechny metody, konstruktory a synchronní porty. Vše, co třída zdědí od třídy `PN` a jejích potomků, může předefinovat. Metody, zděděné od třídy `Object` však předefinovat nelze. Metody, konstruktory a synchronní porty mohou být předefinovány uvedením nových definic pro stejný selektor zprávy, jako u nadtřídy.

Síť objektu může být předefinována po částech redefinicí jednotlivých míst a přechodů, tj. uvedením míst a přechodů se stejným jménem jako v nadtřídě. V případě redefinice přechodu jsou současně redefinovány i okolní hrany. V případě redefinice místa zůstávají okolní hrany beze změny. To odpovídá již dříve uvedené skutečnosti, že hrany jsou chápány jako součást přechodů.

Kromě redefinice zděděných metod, konstruktorů, synchronních portů a uzlů sítě objektu mohou tyto prvky být také přidávány (a propojovány se zděděnými prvky sítě objektu). Příklad využití dědičnosti a redefinice prvků třídy `C1` z obr. 6.8 třídou `C2` je uveden na obr. 6.10.[6]

V grafické reprezentaci OOPN můžeme (a nemusíme) zobrazovat zděděné prvky nadtřídy, které nejsou přímo propojeny s nově definovanými. V případě třídy `C2` na obr. 6.10 nejsou kromě místa `p` žádné další prvky nadtřídy zobrazeny.

[6]Pro porozumění dědičnosti je třeba sledovat obě třídy současně. Poznamenejme, že uvedená třída nemodeluje žádný smysluplný systém, pouze demonstruje možnosti jazyka PNtalk.



Obrázek 6.10: Ilustrace dědičnosti a předefinování zděděných prvků třídy `C1` z obr. 6.8 v třídě `C2`. Zděděné metody a predikáty nejsou zobrazeny.

Příklady, demonstrující programovací techniky pro OOPN, uvedeme v kapitole 7. Dynamika OOPN byla formálně definována v kapitole 5. V příloze B je uveden konkrétní příklad OOPN, na kterém je demonstrována dynamika tohoto modelu.

6.2 Systém PNtalk

Praktické použití OOPN a jazyka PNtalk vyžaduje odpovídající počítačový nástroj, který umožní specifikovat model, testovat ho a provádět s ním simulační experimenty.

Tento nástroj nazveme *systém PNtalk*. Uvedeme zde obecné doporučení, jak by měl systém PNtalk vypadat. Jeho konkrétní implementace se mohou v detailech a v případných rozšířeních lišit. Prototypovou implementaci systému PNtalk, na kterou se budeme dále v některých případech odvolávat, lze nalézt v [JV97].[7] Obecně lze říci, že každá implementace systému PNtalk by měla umožnit

1. vytvářet a editovat OOPN,
2. realizovat běh (simulaci) OOPN, a to jak automaticky, tak interaktivně.

6.2.1 Browser

K vytváření a editaci OOPN slouží *browser*.[8] Tento nástroj umožňuje navigaci v hierarchii tříd a její editaci, tj. vytváření nových tříd, jejich umisťování do hierarchie dědičnosti, popřípadě rušení tříd. V rámci každé třídy umožňuje prohlížet a editovat její implementaci, tj. množinu sítí (síť objektu a sítě metod). V rámci každé sítě umožňuje vytvářet, editovat a rušit její uzly, tj. místa a přechody, propojené hranami a obsahujícící výrazy inskripčního jazyka. V síti

[7] Případné odlišnosti prototypové implementace jazyka a systému PNtalk od zde uvedených doporučení, lze nalézt v její dokumentaci [JV97].

[8] Použijeme anglický termín, protože neexistuje jeho český ekvivalent. Je zde použit ve stejném smyslu jako v rámci systému Smalltalk [GR83].

objektu lze vytvářet a editovat synchronní porty. Browser též umožňuje deklarovat počáteční třídu a umisťovat, resp. rušit, body zastavení (breakpoints) pro potřeby ladění modelu. Ukázka browseru v prototypové implementaci systému PNtalk je uvedena v příloze D.

6.2.2 Inkrementální překladač a simulátor

Každá síť, vytvořená nebo modifikovaná prostřednictvím browseru, je *inkrementálním překladačem* přeložena do vnitřní reprezentace, která umožňuje simulaci.

Simulátor umožňuje na základě zvoleného režimu simulace postupně provádět přechody a realizovat tak běh modelu. Simulace může probíhat v některém z těchto režimů:

- *Superautomatická simulace.* Model běží zcela nezávisle a zastaví se v případě, že není proveditelný žádný přechod, nebo vypršel předem zvolený simulační čas,[9] nebo byl běh násilně ukončen uživatelem.
- *Automatická simulace.* Model běží nezávisle, dokud nenarazí na bod zastavení. Poté přechází do režimu interaktivní simulace.
- *Interaktivní simulace.* Každá událost v modelu je podmíněna souhlasem uživatele. Lze též přejít do režimu automatické simulace.

6.2.3 Debugger

Interaktivní sledování a ovládání běhu systému, popsaného prostřednictvím OOPN, umožňuje v rámci systému PNtalk nástroj, zvaný *debugger*.[10] Debugger obsahuje prostředky pro navigaci v hierarchii běžících (rozpracovaných) procesů. Lze zvolit buď hierarchické uspořádání procesů jednotlivých sítí podle relace „kdo koho vytvořil“,[11] nebo hierarchii systém–objekt–proces. Ke každému procesu je debugger schopen zobrazit odpovídající síť, její aktuální značení (každému místu přísluší multimnožina objektů a každému přechodu přísluší množina rozpracovaných invokací) a pro každý přechod seznam těch navázání proměnných, pro které je proveditelný, spolu se způsobem proveditelnosti (A, N, F, J – viz definice OOPN v kap. 5). Uživatel může vybrat proveditelný přechod (spolu s navázáním proměnných) a provést ho. Debugger poté zobrazí nový stav a takto lze dále pokračovat nebo přejít do režimu automatické simulace. Ukázka debuggeru v prototypové implementaci systému PNtalk je uvedena v příloze D.

6.2.4 Možnosti implementace

Výše uvedená doporučení pro systém PNtalk vycházejí jednak přímo z vlastností OOPN a jazyka PNtalk, jednak ze zkušeností s prototypovou implementací.[12]

Třída PNtalku je v prototypové verzi systému PNtalk implementována jako třída Smalltalku, což umožňuje Smalltalkovským objektům transparentně komunikovat s objekty PNtalku. Obráceně, vzhledem k tomu, že jako primitivní objekty PNtalku mohou vystupovat libovolné objekty Smalltalku, může takováto implementace PNtalku využívat všech možností, které Smalltalk nabízí. Tyto skutečnosti a jejich využití budou diskutovány v kapitole 7.

[9] Práce se simulačním časem bude diskutována v kapitole 7.

[10] Opět kvůli výstižnosti a neexistenci českého ekvivalentu použijeme anglický termín.

[11] V tomto případě se v hierarchii objevují i již ukončené procesy, které mají dosud neukončené potomky.

[12] Prototypová verze systému PNtalk byla navržena v [Jan94b, Jan96], detailně rozpracována a implementována v [Voj96, Voj97b] a zveřejněna v [JV97]. Alternativní implementace byla provedena v [Š96]. Uživatelské rozhraní pro PNtalk bylo implementováno v [Vla96]. Některé implementační otázky byly diskutovány též v [Jan97c].

Kapitola 7

Aplikace a návaznosti

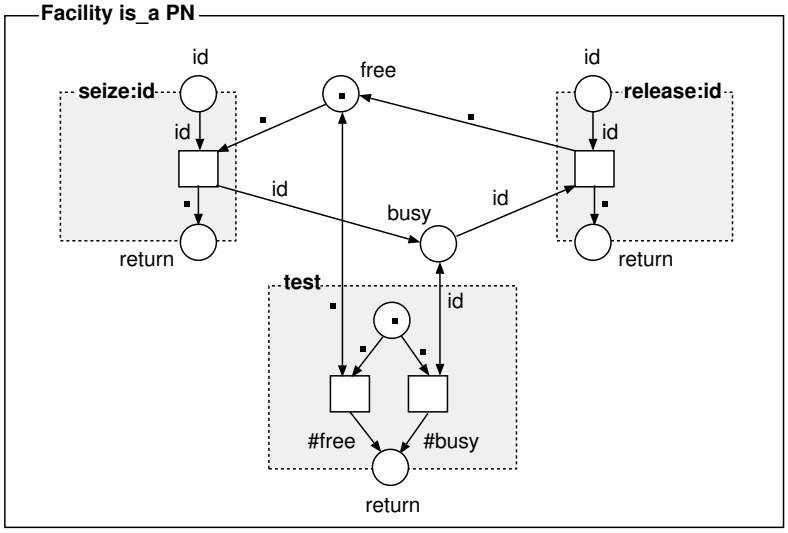
Tato kapitola se zabývá modelováním a programováním v jazyce PNtalk, diskutuje jeho vlastnosti a naznačuje možnosti navazujícího výzkumu.

7.1 Modelování a programování v jazyce PNtalk

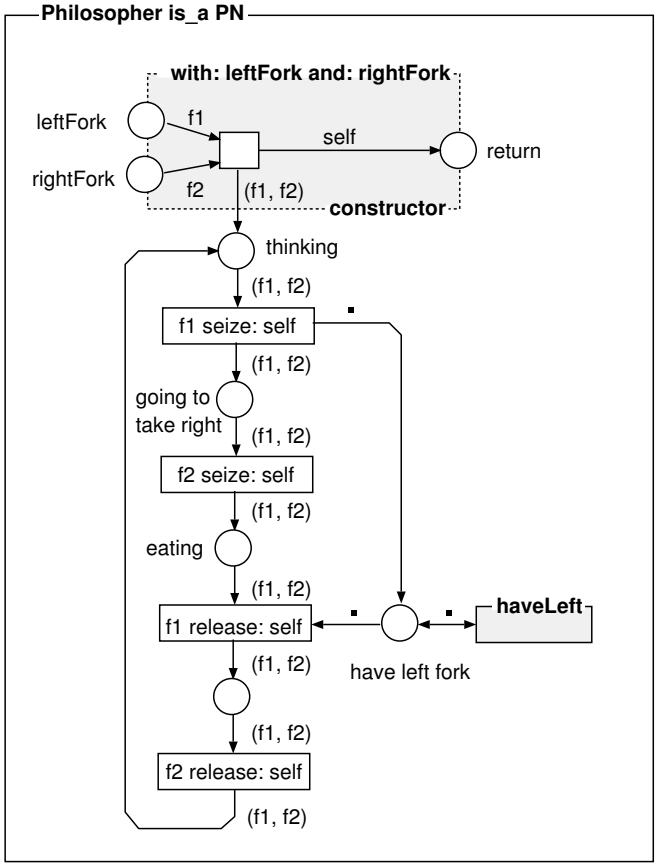
Pro modelování systémů je vždy nutné zvolit adekvátní prostředky. Přestože OOPN a PNtalk umožňují modelování klasickými variantami Petriho sítí (jde totiž o speciální podtřídy OOPN), zde se zaměříme na příklady takových modelů, ve kterých se projeví převážně ty rysy OOPN a jazyka PNtalk, kterými se od ostatních variant Petriho sítí liší, tj. objektově orientované strukturování, klient-server komunikace a synchronní interakce objektů.

Způsob modelování v jazyce PNtalk budeme demonstrovat na několika modifikacích systému večeřících filosofů, vytvořených tak, aby reprezentovaly významné rysy systémů, k jejichž modelování je vhodné PNtalk použít.

7.1.1 Večeřící filosofové



Obrázek 7.1: Zařízení.



Obrázek 7.2: Filosof.

Pro demonstraci objektově orientovaného modelování uvedeme příklad,[1] který navazuje na model večeřících filosofů, uvedený v kapitole 2. Navrhneme detailnější model, ve kterém filosofové i vidličky jsou objekty.

Vidlička je objekt třídy zařízení (facility). Zařízení může být libovolným jiným objektem, v našem případě filosofem, obsazeno a později uvolněno. V každém okamžiku může být zařízení obsazeno nejvýše jedním objektem (klientem). Jde o pasivní objekt, s kterým ostatní objekty mohou manipulovat zasíláním zpráv `seize:` a `release:` pro obsazení a uvolnění zařízení.

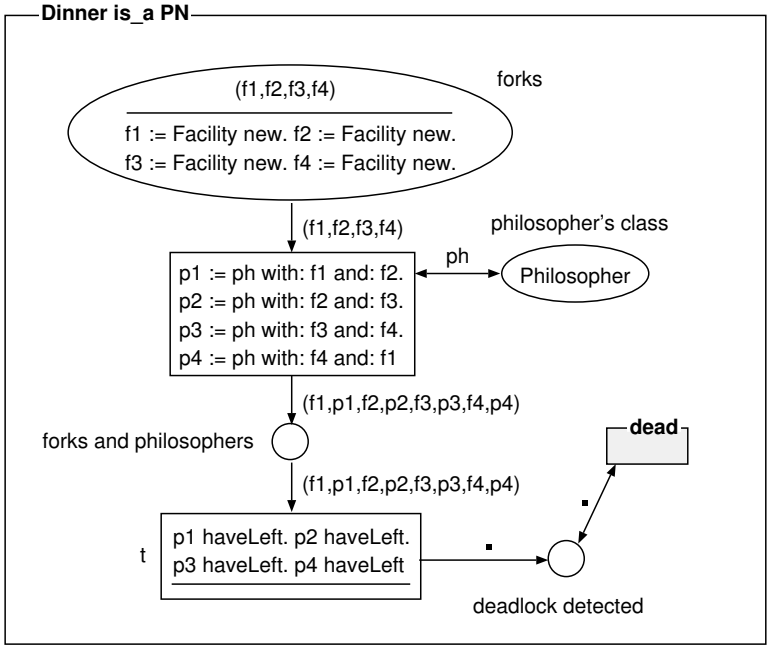
Zařízení je specifikováno třídou `Facility`, uvedenou na obr. 7.1. Metody `seize:` a `release:` pro obsazení a uvolnění zařízení akceptují jako parametr identifikaci klienta. Metoda `test` vrací stav zařízení.[2]

Filosof je aktivní objekt, který při svém vytvoření obdrží reference na dvě vidličky (instance třídy `Facility`). Jeho aktivita spočívá v postupném obsazení obou vidliček v pevném pořadí a poté v jejich postupném uvolnění. Tato činnost se neustále opakuje.

Filosof je specifikován třídou `Philosopher` (viz obr. 7.2). Konstruktor `with:and:` akceptuje jako parametry referenci na levou a pravou vidličku. Filosof, který se snaží postupně obsazovat vidličky v pevně daném pořadí (nejprve levou a potom pravou vidličku), může způsobit zablokování (uváznutí, deadlock). Stav, který vede k zablokování, lze v našem případě snadno detekovat – každý filosof má obsazenou levou vidličku a čeká na pravou. Pro detekci tohoto

[1] Jde o zjednodušenou verzi příkladu z [Jan94b].

[2] V tomto příkladu metodu `test` nevyužijeme. Je určena k použití v dalších příkladech.



Obrázek 7.3: Večeřící filosofové.

stavu definujeme ve třídě `Philosopher` synchronní port (predikátovou metodu) `haveLeft`.

Počáteční třídou modelu večeřících filosofů je třída `Dinner` (viz obr. 7.3), která definuje inicializaci modelu, tj. vytvoření vidliček a filosofů. Další úlohou objektu třídy `Dinner` je detekce zablokování (přechodem `t`). Pro další využití schopnosti tohoto objektu detekovat zablokování (v dalších příkladech) je zde definován synchronní port `dead`.

7.1.2 Živí filosofové

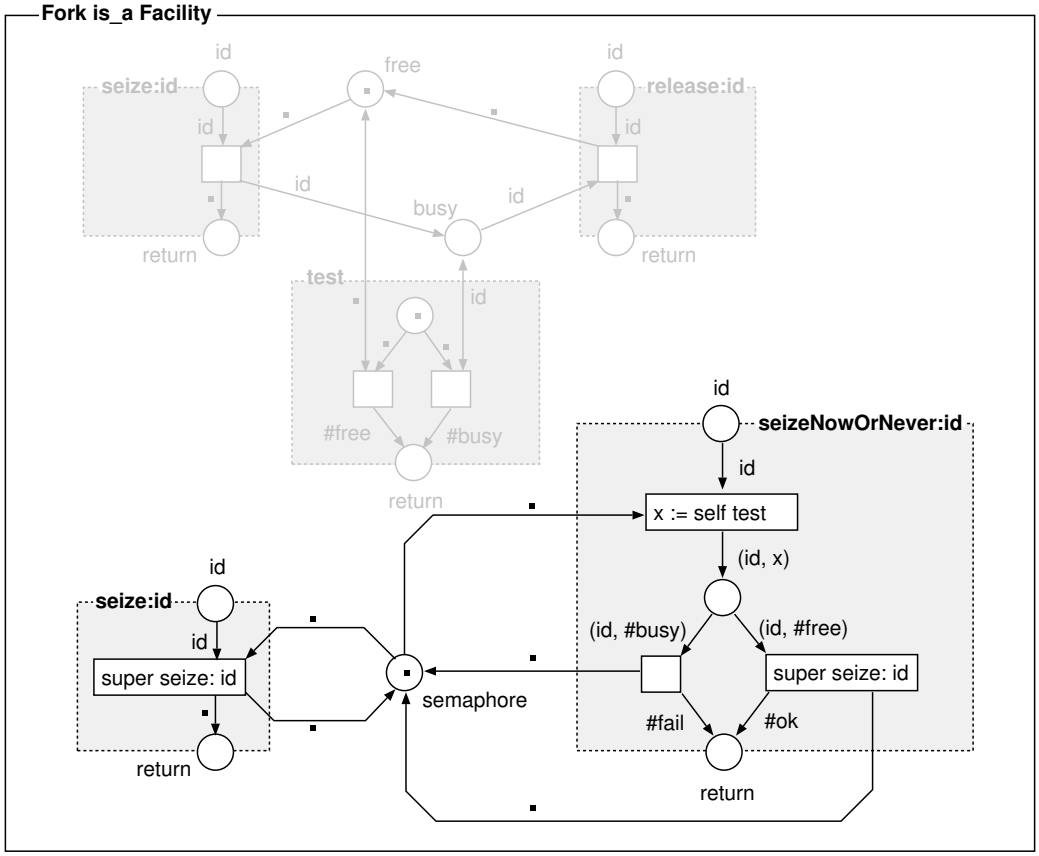
Na příkladu další varianty večeřících filosofů[3] (živí filosofové, tj. filosofové bez možnosti zablokování) budeme demonstrovat dědičnost. Vraťme se k třídě `Facility` z obr. 7.1. Poznamenejme, že testování stavu zařízení voláním metody `test` je poněkud problematické, protože ještě než klient může na informaci o stavu zařízení zareagovat, může dojít k jeho změně. Tento problém by bylo možné řešit synchronními porty, ale pro demonstraci jiných rysů jazyka PNtalk nyní uvedeme odlišné řešení.[4] Ve třídě `Fork` (viz obr. 7.4), která dědí od třídy `Facility`, definujeme metodu `seizeNowOrNever:`, která v případě, že zařízení (vidlička) je obsazené, nečeká na jeho uvolnění, ale skončí s informací o neúspěchu.[5] Pro zajištění korektního chování je metoda `seize:`, kterou budeme také používat, předefinována tak, že modifikuje stav zařízení (vidličky) ve vzájemném vyloučení s metodou `seizeNowOrNever`. Metoda `seize:` používá předchozí verzi téže metody voláním `super`.[6] Proto také šedě zobrazené zděděné prvky třídy `Faility` stále obsahují i původní verzi metody `seize:`. Ta je však v našem případě dostupná pouze z přechodů v sítích třídy `Facility` voláním `super`.

[3] Jde o příklad z [Jan94b].

[4] Demonstraci použití synchronních portů uvedeme později.

[5] Selektor zprávy pro tuto metodu nebudeme brát doslova. Další pokus o obsazení může již proběhnout úspěšně.

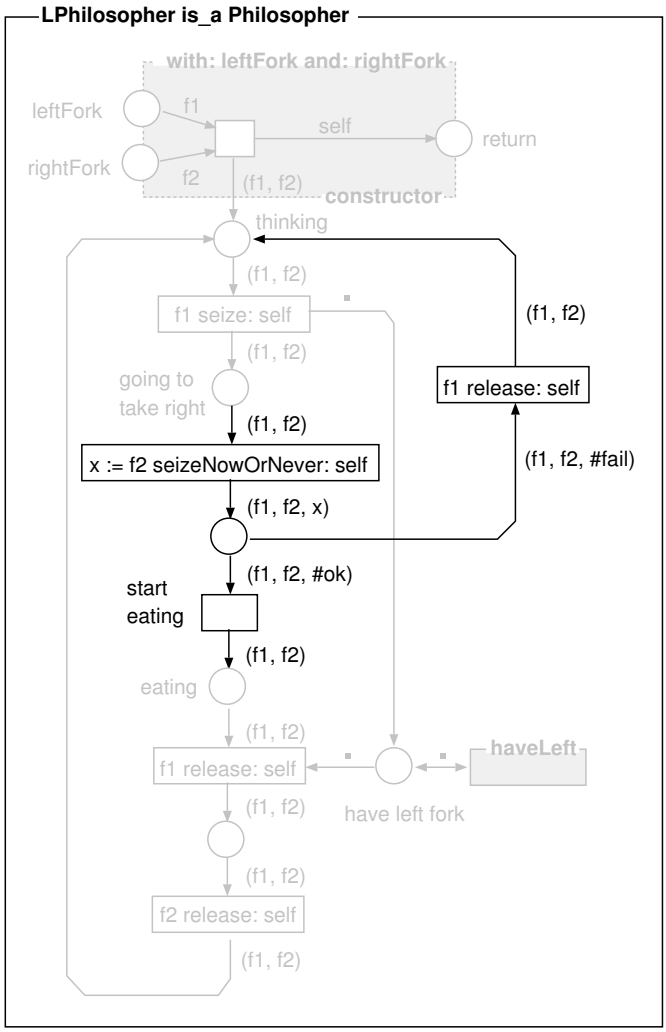
[6] Připomeňme, že sématika volání `super` je totožná se stejným voláním v jazyce Smalltalk – volá se metoda, která je definována v nejbližší nadtřídě třídy, ve které je definována síť, obsahující přechod, jehož akce volání `super` obsahuje.



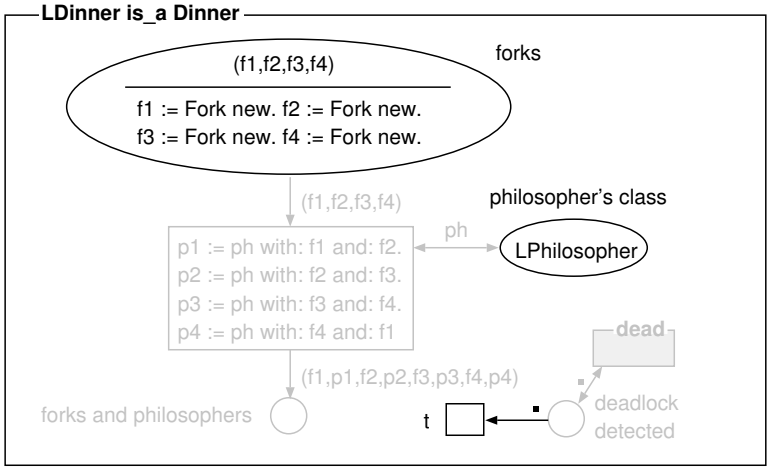
Obrázek 7.4: Vidlička.

Takto definovaná třída `Fork` nám umožní vytvořit model večeřících filosofů, kteří sice také obsazují vidličky postupně, ale v případě neúspěchu uvolní již obsazenou vidličku a pokouší se vidličky obsadit znovu, jak je vidět v definici třídy `LPhilosopher` (live philosopher) na obr. 7.5.

Třída `LDinner` (viz obr. 7.6) předefinovává přechod `t`, zděděný od třídy `Dinner`, tak, aby nebyl nikdy proveditelný, protože detekce zablokování je v tomto případě zbytečná. Jako jediné vstupní místo předhodu `t` je zde použito zděděné místo „`deadlock detected`“, které má stále prázdné značení a také ztratilo původní význam.

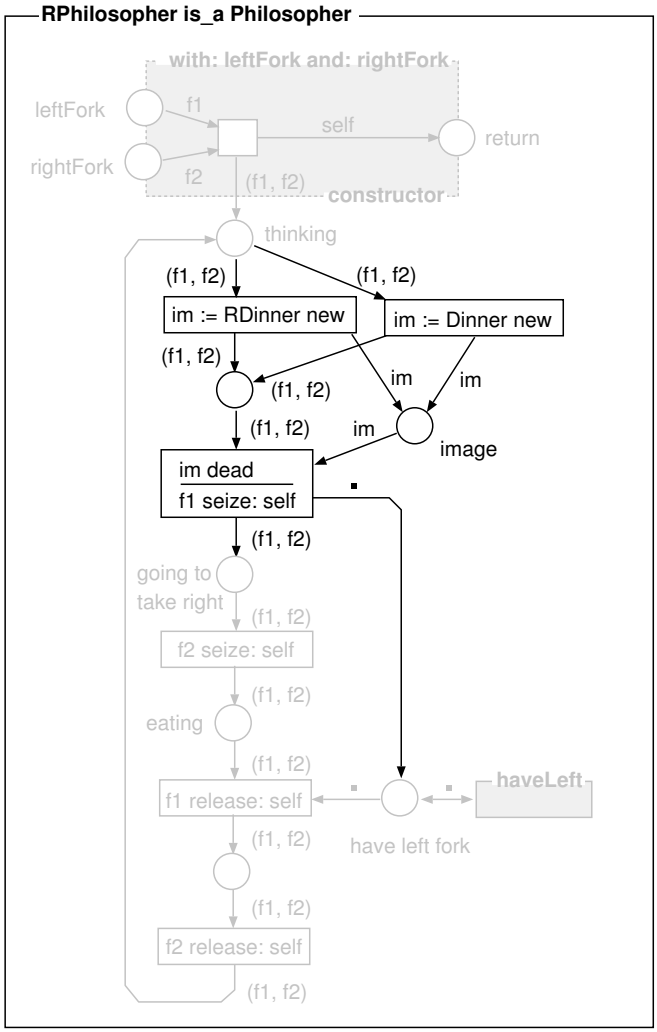


Obrázek 7.5: Živý filosof.



Obrázek 7.6: Živí večeřící filosofové.

7.1.3 Ruští filosofové

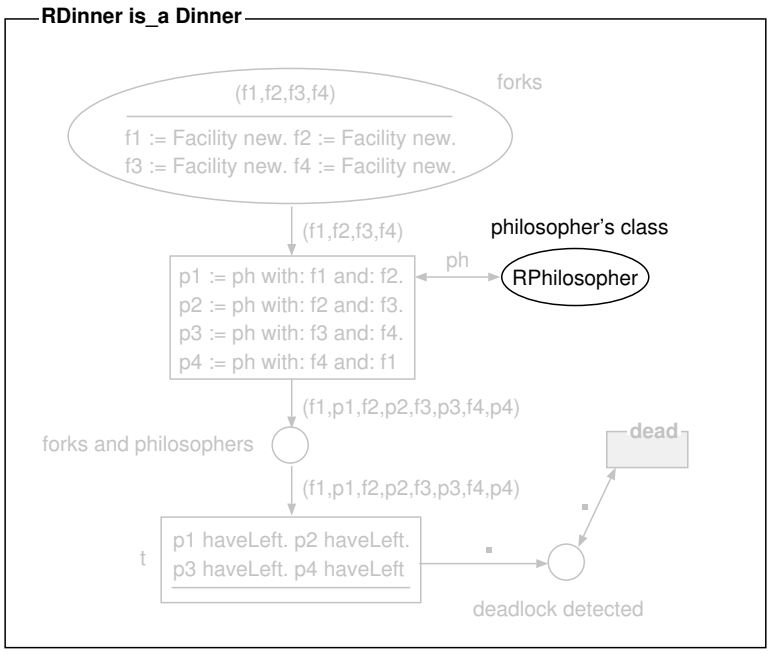


Obrázek 7.7: Ruský filosof.

Pro demonstraci modelování víceúrovňové aktivity C. Lakos v [Lak94] definoval systém ruských filosofů.[7] Ruský filosof se od filosofa z obr. 7.2 liší pouze tím, že ve stavu „přemýšlení“ si ve své představě simuluje systém ruských filosofů, a to tak dlouho, dokud v této představě nedojde k zablokování, tj. dokud všichni filosofové, které si ruský filosof představuje, nedrží levou vidličku. Pak se sám snaží získat levou vidličku a pokračovat standardním způsobem. Aby vnořování představ systémů ruských filosofů mohlo být konečné, dovolíme ruskému filosofovi alternativně vytvořit představu standardních (nikoliv jen ruských) večeřících filosofů.

Ruského filosofa definujeme inkrementálně vůči standardnímu filosofovi na obr. 7.7. Systém ruských filosofů je specifikován třídou `RDinner` na obr. 7.8.

[7] Tento model je inspirován známou ruskou figurkou, obsahující uvnitř opět figurku, která uvnitř opět obsahuje figurku atd.



Obrázek 7.8: Večeře ruských filosofů.

7.1.4 Distribuovaní filosofové

V předchozích příkladech šlo o komunikaci typu klient-server mezi aktivním a pasivním objektem. Klientem byl aktivní objekt (filosof) a serverem byl pasivní objekt (vidlička). Výjimku tvořila pouze synchronní komunikace (detekce zablokování filosofů ve třídách `Dinner`, `RDinner` a `RPhilosopher`. Nyní na další variantě večeřících filosofů ukážeme komunikaci typu klient-server mezi rovnocennými aktivními objekty, což je situace, typická pro obecné distribuované systémy. Kromě toho ukážeme i jinou možnost inicializace modelu, než jaká byla realizovaná v předchozích příkladech.

Systém je tvořen množinou filosofů. Každý filosof může v daném okamžiku vlastnit nejvýše jednu levou vidličku a nejvýše jednu pravou vidličku. Každý filosof zná svého levého a pravého souseda. Svou levou vidličku sdílí se svým levým sousedem (z jeho hlediska jde o pravou vidličku) a naopak, svou pravou vidličku sdílí se svým pravým sousedem. V systému se nachází právě tolik vidliček, kolik je filosofů. Aktivita filosofa spočívá v paralelním provádění několika činností. V případě, že právě nevlastní levou vidličku, může požádat levého souseda, aby mu ji předal. Totéž platí pro pravou vidličku. Kromě těchto dvou činností filosof přemýšlí. Vyhladoví-li a má-li obě vidličky, může vidličky použít k jídlu. Poté vidličky uvolní, ale má je stále u sebe. Je-li filosof požádán některým (levým nebo pravým) sousedem o některou (levou nebo pravou) vidličku, předá mu ji. Toto však může nějakou dobu trvat (když právě jí, musí žádající filosof čekat).

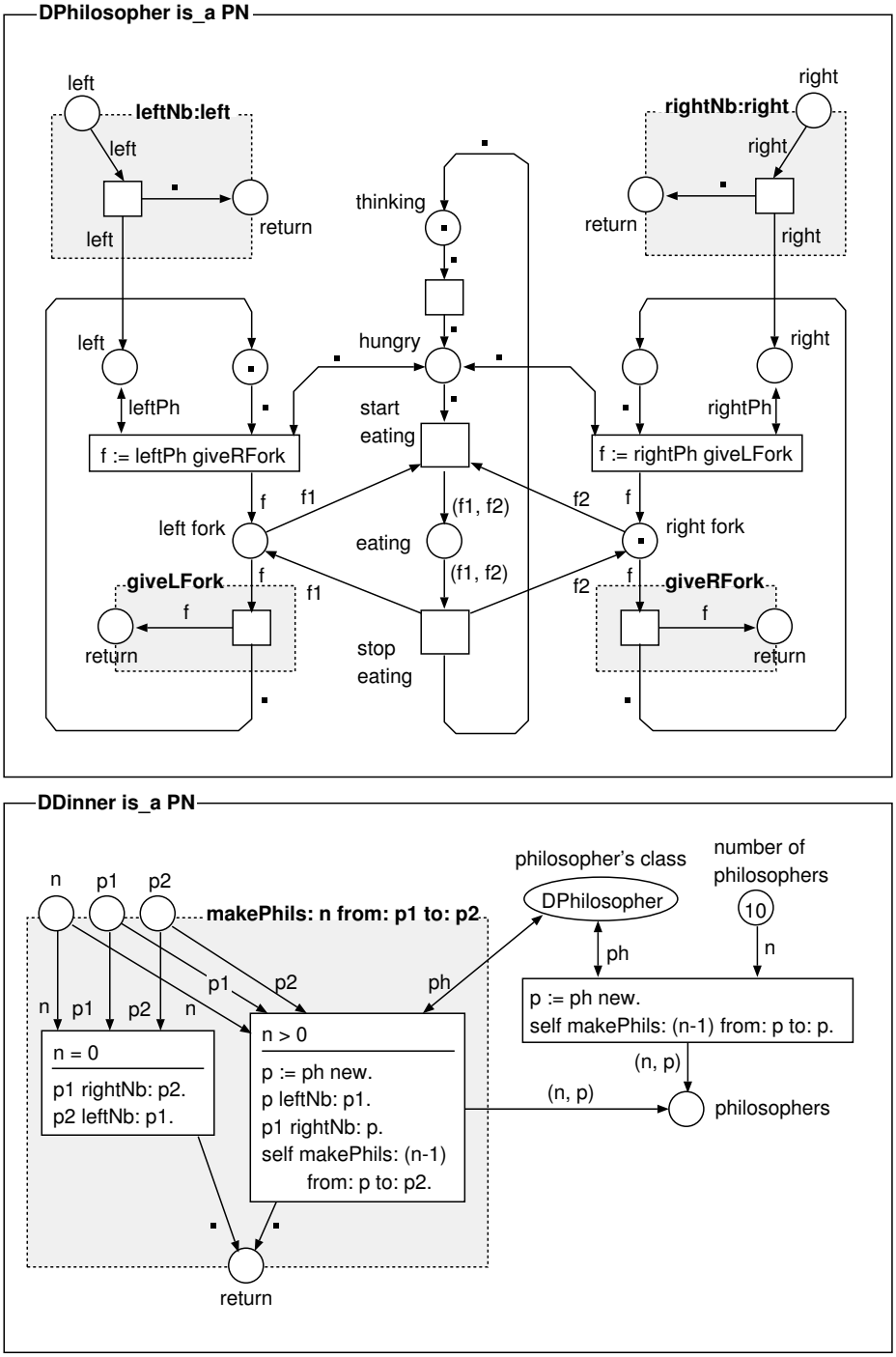
Model distribuovaných filosofů, obsahující třídy `DPhilosopher` a `DDinner` je na obr. 7.9. Třída `DDinner` je prvotní třídou. Vytváření filosofů ve třídě `DDinner` je specifikováno rekurzivně. Nejprve se vytvoří první filosof a poté se voláním rekurzivní metody `makePhils:from:to:`, jejímiž parametry jsou: počet filosofů, nejlevější filosof (z hlediska řetězu filosofů, vytvářených touto metodou) a nejpravější filosof. Zasláním zprávy `self makePhils: (n-1) from: p to: p` dojde k vytvoření a zřetězení `(n-1)` filosofů, přičemž předem vytvořený filosof `p` řetěz uzavírá zleva i zprava, takže je vytvořen obousměrně zřetězený cyklický seznam `n` filosofů. Každý filosof při svém vytvoření vlastní pravou vidličku a přemýšlí.

Poznamenejme, že všechny vidličky jsou modelovány značkou „•“, ale pracuje se s nimi tak, že jako vidličky lze použít (v případné modifikaci tohoto modelu) jakékoliv, i navzájem rozlišitelné objekty.

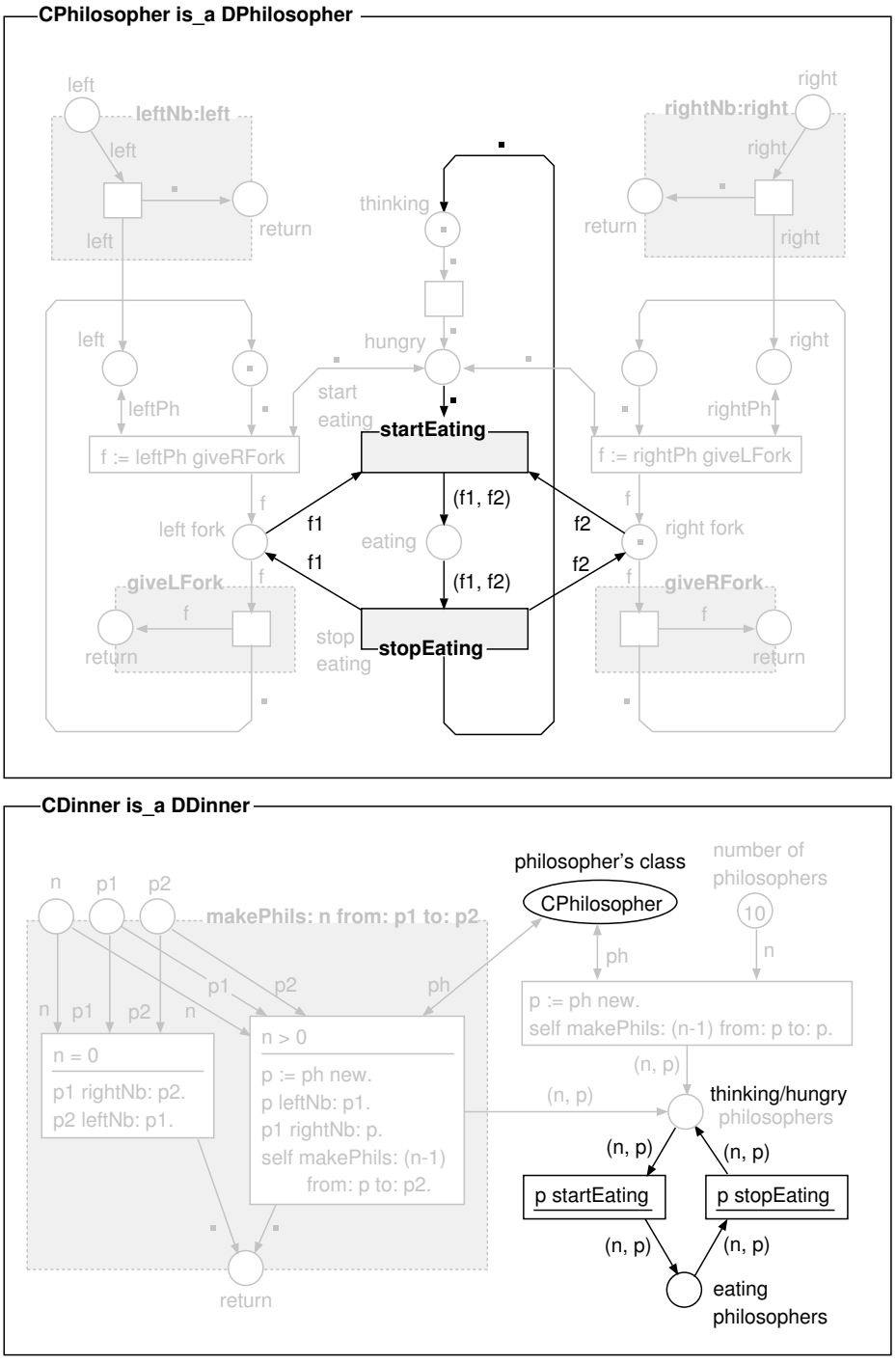
7.1.5 Řízení filosofové

Tento příklad demonstruje jednoduchou aplikaci synchronních portů. Nahradíme-li přechody „`start eating`“ a „`stop eating`“ ve třídě `DPhilosopher` synchronními porty `startEating` a `stopEating` ve třídě `CPhilosopher` na obr. 7.10 a definujeme-li počáteční třídu `CDinner` podle obr. 7.10, docílíme toho, že objekt třídy `CDinner` může sledovat (a případně ovlivňovat) výskyty význačných událostí v objektech třídy `CPhilosopher`. Konkrétně, objekt třídy `CDinner` sleduje, kteří filosofové právě jedí a kteří ne.

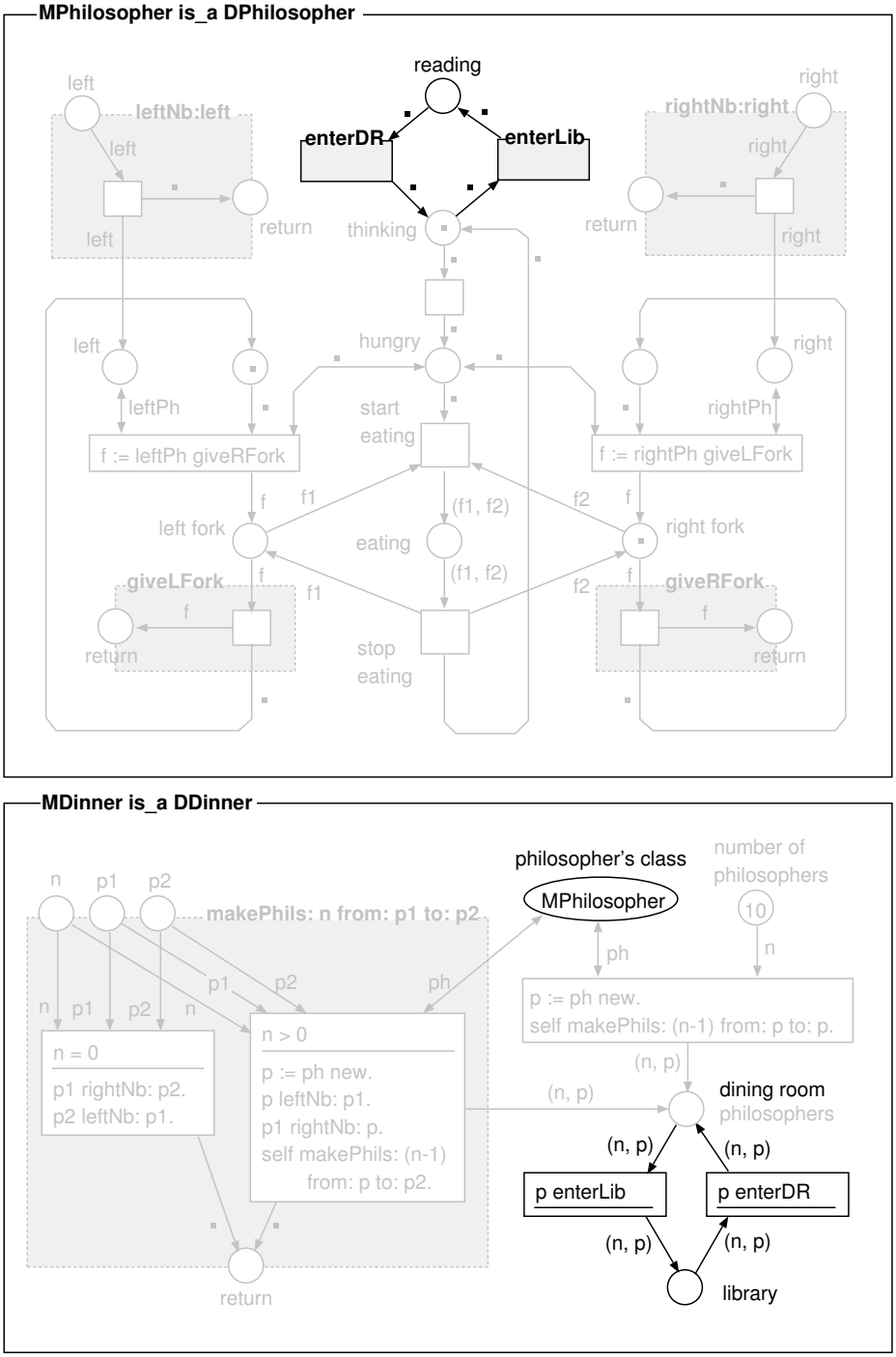
V rámci specifikace tříd `CPhilosopher` a `CDinner` jsme použili dva z dosud nediskutovaných rysů jazyka PNtalk, a sice předefinování přechodu synchronním portem (předchody „`start eating`“ a „`stop eating`“ byly předefinovány synchronními porty `startEating` a `stopEating`) a přejmenování uzlu sítě objektu (místo `philosophers`, zděděné z třídy `DDinner`, bylo přejmenováno na „`thinking/hungry`“). V obou případech je původní jméno uzlu šedě zobrazeno u nové definice. V obou případech jde jen o syntaktické ulehčení programování v jazyce PNtalk, nijak nevybočující z rámce, daného definicí OOPN.



Obrázek 7.9: Deset distribuovaných večeřících filosofů.



Obrázek 7.10: Deset řízených distribuovaných večeřících filosofů.



Obrázek 7.11: Deset migrujících distribuovaných filosofů.

7.1.6 Migrující filosofové

Výše uvedený způsob synchronizace aktivit v různých objektech, realizovaný synchronními porty, lze s výhodou využít při modelování pohybu aktivních objektů uvnitř struktury jiného objektu. R.Valk tuto problematiku v [Val98] demonstruje na příkladu migrujících filosofů, pohybujících se mezi jídelnou (dining room) a knihovnou (library). Je-li filosof ve stavu „přemýšlení“, může (i s vidličkami, pokud je právě vlastní) odejít do knihovny a kdykoliv se vrátit zpět do jídelny. Model takovéto varianty distribuovaných večeřících filosofů, specifikovaný v PNtalku, je uveden na obr. 7.11. Objekt třídy `MDinner` jednak řeší inicializaci modelu, jednak reprezentuje fyzickou strukturu, ve které se filosofové pohybují.

7.1.7 Přicházející a odcházející filosofové

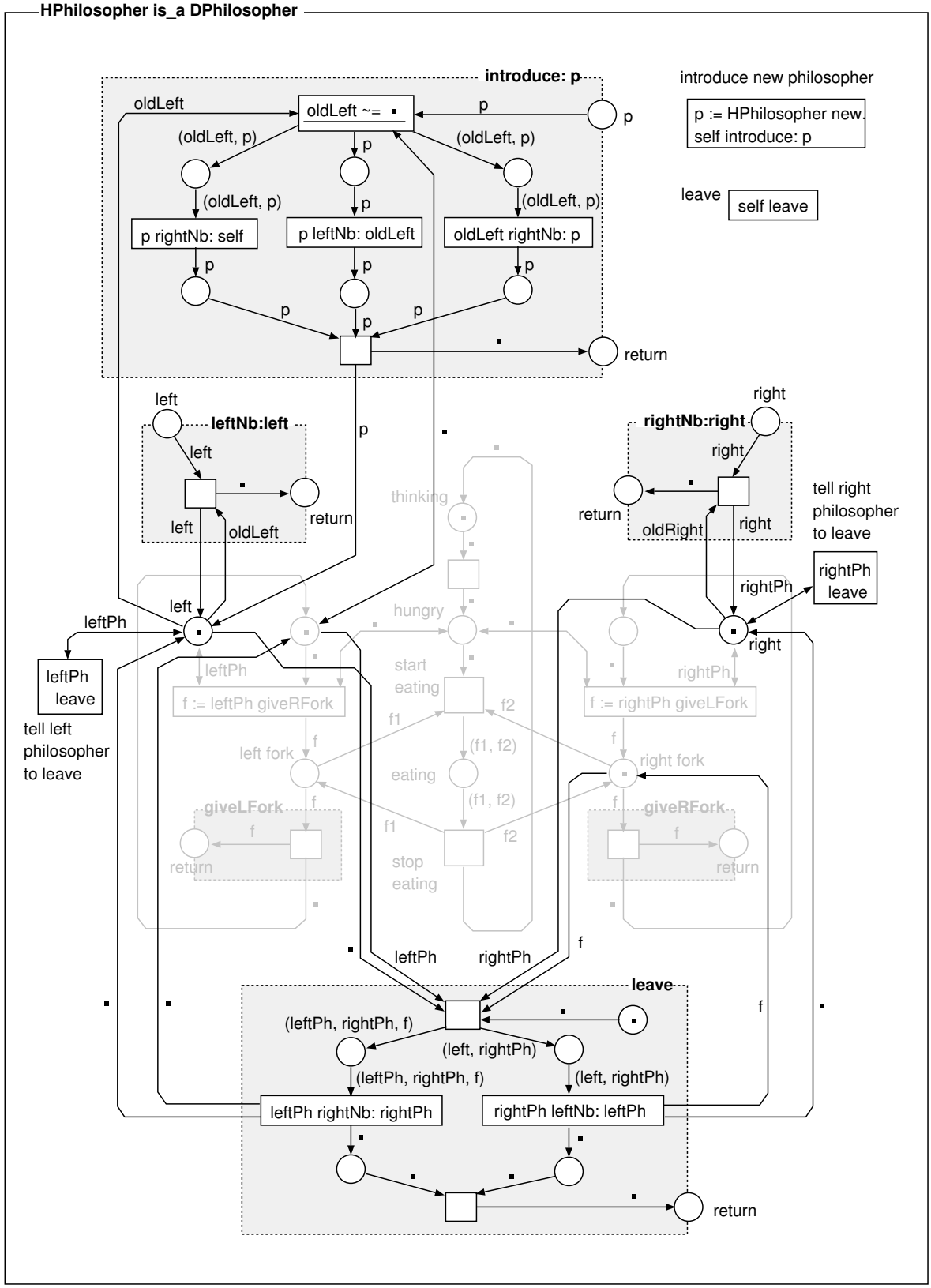
Distribuovanou variantu večeřících filosofů z obr. 7.9 lze snadno modifikovat tak, že umožníme filosofovi pozvat nového filosofa (vlastnícího pravou vidličku) a zapojit ho do cyklického seznamu, kde sdílí levou vidličku s levým sousedem a pravou vidličku s pravým sousedem. Také umožníme každému filosofovi, momentálně vlastnícímu pouze pravou vidličku, odejít ze systému. Tuto variantu večeřících filosofů (hurried philosophers) definoval C.Sibertin-Blanc v [SB94] jako příklad dynamicky rekonfigurovatelného distribuovaného systému.

Třída `HPhilosopher` na obr. 7.12 definuje metodu `introduce:`, která zařadí filosofa, který je uveden jako parametr, do cyklického seznamu filosofů tak, že se nový filosof stane levým sousedem filosofa, který metodu `introduce:` provádí. Metoda `leave` realizuje odchod filosofa z cyklického seznamu tím, že svému levému i pravému sousedu oznámí jejich nového pravého a levého souseda. Pozvání nového filosofa a rozhodnutí odejít realizují přechody „`introduce new philosopher`“ a „`leave`“. Filosof také může provedením přechodu „`tell right philosopher to leave`“ a „`tell left philosopher to leave`“ požádat pravého a levého souseda, aby odešel.

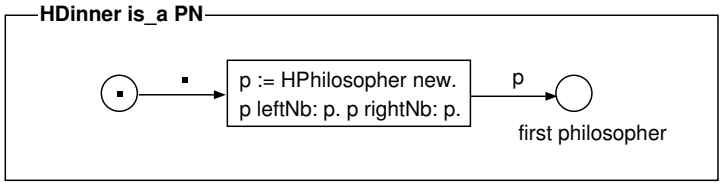
Metody `leftNb:` a `rightNb:` jsou modifikovány tak, aby mohly kdykoliv korektně modifikovat informaci o sousedech filosofa. Místa `left` a `right` po vytvoření filosofa obsahují značku, reprezentující nedefinovaný objekt. Jelikož tento objekt nerozumí zprávám `giveRFork` a `giveLFork`, přechody, žádající vidličky od sousedů jsou neproveditelné, dokud není informace o sousedech k dispozici.

Počáteční třídou této varianty večeřících filosofů je třída `HDinner`. Inicializace modelu spočívá ve vytvoření prvního filosofa, který je současně svým levým i pravým sousedem. Další filosofové se v systému objevují v důsledku aktivity již existujících filosofů. Jelikož z objektu počáteční třídy `HDinner` je referencován pouze první filosof, jeho odchodem z cyklického seznamu filosofů přestanou být ostatní filosofové referencováni[8] z počátečního objektu a jsou garbage-collectorem odstraněni. První filosof se pak už jen marně snaží nové filosofy zapojovat do cyklického seznamu prováděním přechodu „`introduce new philosopher`“, protože metoda `introduce:` nemůže skončit. Lepšího chování modelu by bylo možné docílit registrací nově vytvářených filosofů v počátečním objektu a odstraňováním těcho registrací při odchodu filosofů ze systému. V dalším příkladu uvedeme toto řešení v kombinaci s další technikou programování v jazyce PNtalk, která mimo jiné zamezí nevhodnému překrývání aktivit filosofů, které by mohlo vést k zablokování.

[8] První filosof ztratí reference na levého a pravého souseda.



Obrázek 7.12: Třída přicházejících a odcházejících filosofů.



Obrázek 7.13: Přicházející a odcházející filosofové.

7.1.8 Přicházející, migrující a odcházející filosofové

Tento příklad, který může poslužit jako vzor pro modelování workflow nebo výrobních systémů, demonstruje použití synchronních portů ke koordinaci aktivit dynamicky vznikajících a zanikajících objektů a k modelování jejich pohybu v systému.

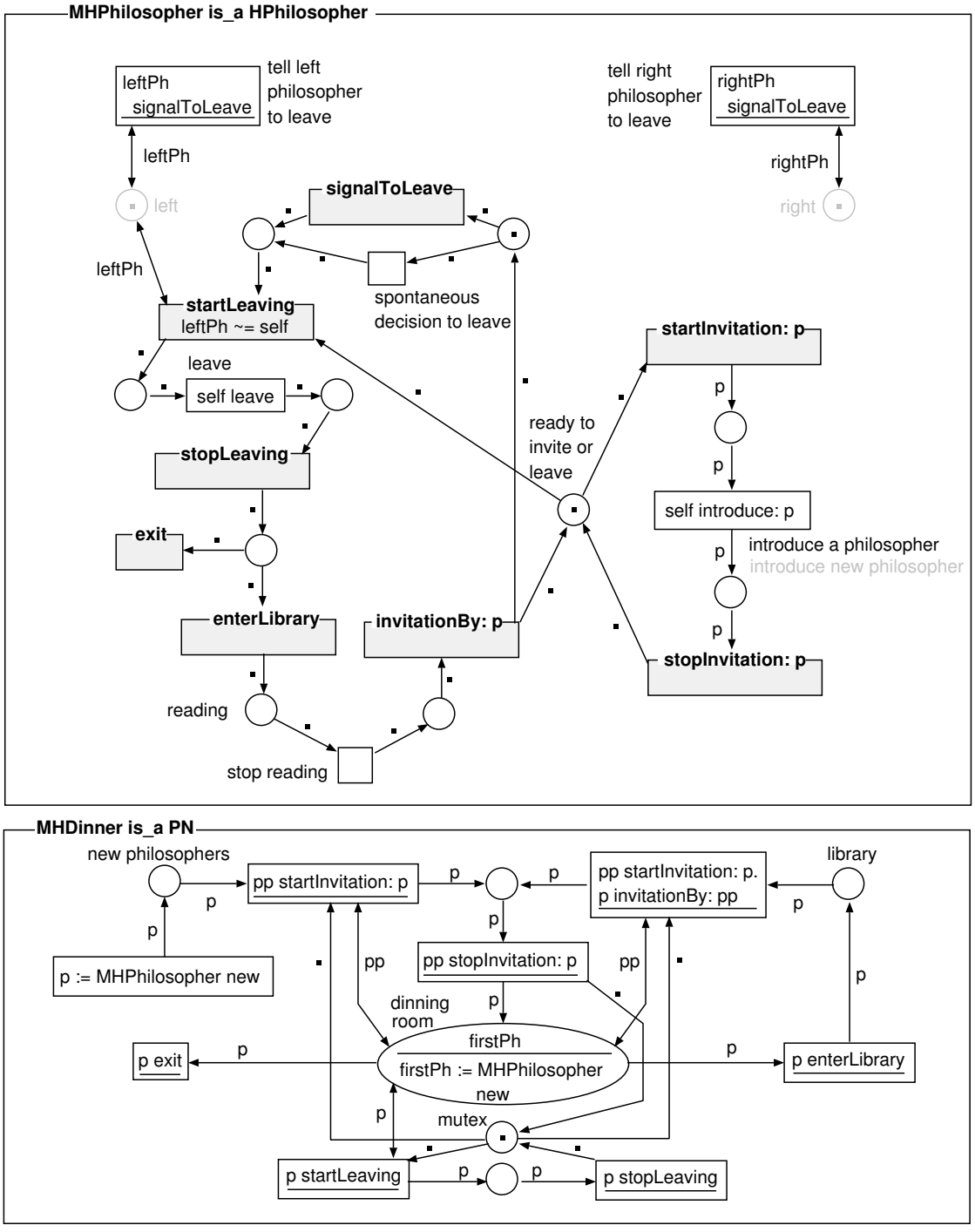
Jde o variantu distribuovaných večeřících filosofů, kteří mohou přicházet do systému, pohybovat se mezi jídelnou a knihovnou a odcházet ze systému. Nově příchozí filosof může být pozván do jídelny některým filosofem, který už v jídelně je. Filosof se může kdykoliv rozhodnout opustit jídelnu, a to buď spontánně, nebo na základě signálu k odchodu, který dostal od některého ze svých sousedů. Odejít však může pouze tehdy, když v jídelně není sám. Při odchodu z jídelny filosof korektně upraví reference mezi svými sousedy a poté může buď odejít do knihovny, nebo navždy opustit systém. Je-li filosof v knihovně a přestane-li číst, může být pozván zpět do jídelny některým z filosofů, kteří již v jídelně jsou.

Třída `MHPhilosopher` na obr. 7.14 je inkrementální modifikací přicházejících a odcházejících distribuovaných večeřících filosofů z obr. 7.12. Poznamenejme, že v tomto případě nezobrazujeme zděděnou třídu, ale pouze ty zděděné uzly sítě objektu, které souvisejí s nově definovanými nebo předefinovanými uzly a synchronními porty. Třída `MHPhilosopher` definuje několik synchronních portů. Synchronní port `signalToLeave` je volán jiným filosofem, ostatní synchronní porty jsou volány počátečním objektem třídy `MHDinner`.

Povšimněme si blíže úlohy, kterou plní třída `MHDinner` (viz obr. 7.14). Tato třída modeluje strukturu, ve které se filosofové pohybují a která jim umožňuje nejen přesuny v této struktuře, ale i jejich vzájemnou interakci a koordinaci jejich vlastních aktivit. Podívejme se na tuto problematiku podrobněji. Aby nedošlo k zablování, je třeba řešit vzájemnou výlučnost akcí, spojených s odchodem filosofů a se zvaním filosofů do jídelny. Odchod filosofů je v třídě `MHDinner` řešen voláním synchronních portů `startLeaving` a `stopLeaving` ve vzájemném vyloučení s voláním synchronních portů `startInvitation` a `stopInvitation`.

Aby bylo učiněno zadost požadavku, aby v jídelně vždy zůstal alespoň jeden filosof, každý filosof volání `startLeaving` akceptuje jen když není sám svým levým (i pravým) sousedem, tj. když nevečeří sám.[9] Filosof může vstoupit do knihovny, je-li k tomu sám připraven (tj. poté, co oznámil sousedům, že odchází), a pokud mu to struktura, ve které se nachází, dovolí. Z knihovny zpět ho může zavolat libovolný filosof, který je právě v jídelně a je připraven někoho pozvat. Vzhledem k tomu, že filosof v jídelně zná pouze svého pravého a levého souseda, o existenci filosofa v knihovně se může dovědět pouze prostřednictvím struktury, v níž se nachází. Tato struktura, modelovaná třídou `MHDinner` tedy umožňuje seznamování, komunikaci, migraci a koordinaci aktivit filosofů.

[9] Večeří-li filosof sám, ve skutečnosti se nedostane k jídlu, neboť neustále sám sebe (je totiž sám svým sousedem) žádá o levou nebo pravou vidličku a když ji dostane, má opět jen jednu a proto opět sám sebe žádá o druhou atd.



Obrázek 7.14: Přicházející, migrující a odcházející filosofové.

Poznamenejme, že třída `MHDinner` modeluje statickou strukturu, obsahující knihovnu a jídelnu, a okolí systému, kde vznikají noví filosofové. Až na přechod, který vytváří nové filosofy, mají všechny přechody stráže, které specifikují volání synchronních portů filosofů. Jejich proveditelnost závisí na vlastní aktivitě (rozhodování) filosofů. Stojí za povšimnutí, že v případě synchronní komunikace (na rozdíl od klient-server komunikace) je lhostejné, kdo z účastníků synchronní komunikace je považován za aktivního a kdo za pasivního.[10] Při implementaci modelu je však nutné určit iniciátora komunikace. Iniciátorem je vždy ten objekt, který zná (má reference na) všechny účastníky synchronní komunikace, v našem případě jde o objekt třídy `MHDinner`.

Co se týče použití synchronních portů k zajištění vzájemného vyloučení invokací metod filosofů, které realizují odchod filosofů a zvaní filosofů do jídelny, je třeba zdůraznit, že jde o programování na vyšším stupni abstrakce, než jaké poskytuje komunikace objektů výhradně protokolem klient-server. Pokud způsob realizace prostředků pro zajištění synchronních interakcí modelovaných objektů není při tvorbě modelu významný, aplikace synchronních portů výrazně zjednoduší modelování. V opačném případě je třeba k modelování použít prostředky, které mají blíže k cílové implementaci, tj. (obvykle) protokol klient-server, za cenu komplikovanějšího modelu.

7.2 Diskuse vlastností OOPN a možností navazujícího výzkumu

7.2.1 K modelování a prototypování v jazyce PNtalk

PNtalk a Smalltalk

Různé implementace jazyka PNtalk se mohou lišit množinou primitivních objektů a schopností uživatelského programování tříd primitivních objektů. Prototypová implementace PNtalku [JŠV96, ČJV97a] je řešena jako transparentní vrstva nad Smalltalkem. Jsou zde možné dvě úrovně programování:

1. Programování neprimitivních tříd v PNtalku umožňuje specifikaci aktivních objektů a modelování na vyšší úrovni abstrakce. Předpokládá se, že v PNtalku se programují hlavní aplikační třídy.

2. Smalltalk poskytuje své knihovní třídy a umožňuje tvorbu dalších tříd, implementujících podpůrné datové typy (jde o třídy primitivních objektů). Umožňuje též programovat vstupně/výstupní operace, popřípadě tvorbu grafických uživatelských rozhraní.

Takováto implementace systému PNtalk umožňuje adekvátními prostředky vyjádřit jednotlivé části modelu nebo prototypu. Je však třeba respektovat skutečnost, že většina objektů Smalltalku není implicitně připravena pro použití v paralelních programech a přístup k nim je často nutné synchronizovat například použitím semaforů.

Komunikace s okolím

Aplikace jazyka a systému PNtalk pro simulaci v reálném čase, prototypování, monitorování technologických procesů a jejich řízení vyžaduje komunikaci objektů, programovaných v jazyce PNtalk, s okolním světem. Obecně existují dvě možnosti komunikace s okolím:

1. Stráž i akce přechodu mohou zasíláním zpráv komunikovat s objekty, realizovanými jako uživatelem definované primitivní objekty PNtalku, realizující styk s okolním světem.

[10] V našem případě jsou oba objekty aktivní, i když třída `MHDinner` modeluje ve skutečnosti pasivní struktutu (až na generování nových filosofů).

2. Uživatelem definované primitivní objekty mohou zasíláním zpráv komunikovat s neprimitivními objekty PNtalku, i s jejich třídami.

Stochastické simulační modelování v jazyce PNtalk

Procesor jazyka PNtalk (jeho současná prototypová implementace [JŠV96, ČJV97a]) umožňuje práci se simulačním časem. Jsou zde implementovány prostředky pro specifikaci zpožďujících přechodů, konformní s principem komunikace zasíláním zpráv. Například, přechod s akcí

```
self wait: (Exp mv: x)
```

čeká po dobu, určenou hodnotou z exponenciálního pravděpodobnostního rozložení se střední hodnotou `x`. Zprávě `wait:` rozumí každý neprimitivní objekt (je definována v rámci třídy `PN`). Typicky se tato zpráva zasílá adresátu `self`. Jsou zde implementovány i jednoduché prostředky pro sběr statistik. Potenciální možnost obohacení jazyka a systému PNtalk o časované přechody, pracující s dobou proveditelnosti (na rozdíl od doby provádění), je diskutována v [Voj97a]. V tomto případě se potřebná doba proveditelnosti přechodu specifikuje v jeho stráži například výrazem

```
self delay: (Exp mv: x).
```

7.2.2 Možná rozšíření OOPN a jazyka PNtalku

Pro ulehčení modelování v jazyce PNtalk je perspektivně možné zavést další konstrukce, jako je hierarchické strukturování jednotlivých sítí v rámci OOPN, rozšíření možností inkrementální specifikace sítí děděním a rozšíření možností synchronní komunikace.

Hierarchické strukturování sítí v jazyce PNtalk

Definice objektově orientované Petriho sítě neřeší otázku statické hierarchie sítí s odůvodněním, že tento strukturovací mechanismus je možné zavést zcela nezávisle na objektově orientovaném strukturování a nepřináší z teoretického hlediska žádné významné problémy. Z hlediska praktické použitelnosti jazyka PNtalk je však výhodné tento jazyk o možnost specifikace statické hierarchie objektů obohatit. Jednotlivé sítě, definující třídy, lze hierarchicky strukturovat tak, jak bylo uvedeno v kapitole 2. Tento způsob zavedení hierarchického strukturování do jazyka PNtalk je diskutován v [Voj97a].

Síťové morfismy realizované dědičností

Inkrementální specifikace sítě objektu v OOPN je založena na přidávání nových uzlů a předefinovávání uzlů zděděné sítě. Lze však zavést i jiné varianty dědičnosti sítí. Jako příklad lze uvést možnosti, které nabízí zavedení pojmenovaných podsítí:

- Předefinování zděděného uzlu (stejně pojmenovanou) podsítí,
- Předefinování pojmenované podsítě zděděné sítě jinou (stejně pojmenovanou) podsítí,
- Předefinování podsítě (stejně pojmenovaným) uzlem.

Synchronní komunikace

V definici OOPN byla synchronní komunikace definována v poněkud omezenější podobě, než jak je možné ji potenciálně zavést. Synchronní port zde nemůže volat jiný synchronní port. Toto omezení je možné (za cenu komplikovanější formální definice) překonat a definovat i rekurzivní volání synchronních portů. Tím se může otevřít prostor pro nové techniky programování v jazyce PNtalk.

Poznamenejme, že současná experimentální verze systému PNtalku [JV97] dosud neimplementuje synchronní komunikaci v plném rozsahu. Umožňuje však používat predikáty, tj. synchronní porty, určené pouze ke zjištění stavu objektu. Predikáty zde umožňují specifikovat i vnořené volání dalších predikátů. Pro potřeby modelování workflow a pružných výrobních systémů je však třeba doplnit synchronní komunikaci alespoň podle formální definice z kapitoly 5.

7.2.3 K teorii OOPN

Formální definice OOPN (uvedená v kapitole 5) položila základ pro teoretickou práci s objekty, popsanými Petriho sítěmi. Teorie OOPN by měla perspektivně umožnit analytické zkoumání vlastností modelů, specifikovaných v jazyce PNtalk.

Jednou z uvažovaných možností analýzy je transformace OOPN do některého dobře analyzovatelného formalismu.[11] V tomto případě bude třeba zajistit možnost interpretace výsledků analýzy na úrovni OOPN. Vzhledem k tomu, že tato transformace není triviální, bude se také zkoumat možnost analýzy přímo na úrovni OOPN.

Publikace [JV98b] zahájila výzkum v oblasti analýzy stavového prostoru OOPN. Přímo se zde navazuje na definici OOPN, uvedené v kapitole 5. Je zde diskutována možnost redukce stavového prostoru na principu ekvivalence jmen objektů.

Při analýze OOPN může také pomoci znalost typů objektů, vyskytujících se v místech sítí. V [Voj98] je naznačena možnost dedukce typů. V úvahu přichází také možnost (nepovinných) deklarací typů míst a proměnných.

Předpokládá se, že v rámci dalšího výzkumu podlehne změnám i samotná definice OOPN. Je například třeba doplnit obecnější variantu synchronní komunikace[12] a pokusit se nalézt jiný styl definování některých pojmů, který by celou definici OOPN zjednodušil.

[11] V tomto směru byl učiněn jistý krok v rámci ročníkového projektu T.Vojnara [Voj95] pod vedením autora této práce, který vyústil v algoritmus převodu OOPN, specifikované v jazyce PNtalk do HL-sítě bez objektově orientovaného strukturování, specifikované opět v jazyce PNtalk, a to na základě myšlenek, uvedených v kapitole 4 této práce. Vzhledem k nutnosti řešit množství jiných akutních problémů byl další výzkum v tomto směru pozastaven.

[12] Jde pouze o novou definici pojmu s-proveditelnost a funkce $SYNC$ (viz def. 5.5.3 v kapitole 5).

Kapitola 8

Závěr

Cílem této práce bylo překonat nevýhody Petriho sítí při detailním modelování realistických systémů zavedením objektově orientovaného strukturování. Zamýšlený výsledný formalismus (objektově orientovaná Petriho síť – OOPN) měl umožnit používat Petriho sítě v roli univerzálního jazyka, srovnatelného s Prologem a Smalltalkem, vhodného pro modelování a rychlé prototypování. Shrňme nyní použitý postup a dosažené výsledky, spolu se směry navazujícícho výzkumu.

Postup. Východiskem práce na OOPN byly hierarchické Petriho sítě [HJS90], které zavedly instanciaci sítí a jejich hierarchickou kompozici. Na základě vlastních zkušeností s implementací hierarchických Petriho sítí [ČJ93] a inspirace projektem, aplikujícím Petriho sítě v rámci metody HOOD (hierarchical object oriented design[1]) [Gio91], se autor této práce rozhodl definovat objektově orientovanou Petriho síť tak, aby byla plně kompatibilní se smalltalkovským pojetím objektové orientace, které lze považovat za standardní.

Jako klíčový mechanismus byla zvolena dynamická instanciace Petriho sítí invokačním přechodem [HJS90]. Koncept invokačního přechodu byl poněkud modifikován (byly zavedeny polymorfní přechody) [Jan94a, Jan94b, Jan95a, Jan95b, Jan97a], aby byla zajištěna konzistence s objektově orientovaným inskripčním jazykem (viz kap. 4).

Inspirace Smalltalkem [GR83], kde proměnné obsahují reference na objekty, vedla k tomu, že značky v OOPN byly definovány jako reference na objekty. Koncept programování ve Smalltalku pomocí class-browseru, který přistupuje k metodám jednotlivě, ovlivnil skutečnost, že metody se v OOPN specifikují separátními sítěmi. Specifikací třídy množinou dynamicky instanciovatelných sítí se OOPN výrazně odlišuje od srovnatelných řešení[2] objektové orientace v Petriho sítích, která nezávisle nabízejí C. Lakos [LK94] a C. Sibertin-Blanc [SB94], u nichž je celá třída definována jedinou sítí.

Výsledky. Teoretickým výsledkem této práce je původní formální definice OOPN (viz kap. 5) [Jan97b, ČJ97]. Tento formalismus je založen na aktivních víceprocesových objektech (viz kap. 3), komunikujících protokolem klient-server. Pro potřebu modelování synchronizace aktivit ve víceúrovňových modelech byla zavedena také atomická synchronní komunikace (viz kap. 4 a 5). Oba druhy komunikace jsou syntakticky k dispozici jako zasílání zpráv. OOPN definuje základní pojmy objektové orientace v kontextu Petriho sítí. Tím je vytvořen základ pro teoretické úvahy o systémech s aktivními objekty. Formální definice OOPN umožňuje takové systémy zkoumat

[1] Jak již bylo uvedeno v kap. 4, nelze zde hovořit o objektové orientaci tak, jak ji definuje Smalltalk.

[2] Srovnatelným řešením se zde rozumí objektově orientovaný formalismus, založený na Petriho sítích, kde značky v Petriho síti reprezentují objekty, tyto objekty jsou specifikovány Petriho sítěmi a existuje formální definice tohoto modelu.

jako přechodové systémy (pro OOPN je definována množina stavů a přechody mezi stavy systému) a vytváří prostor pro aplikaci teorie Petriho sítí (objektově orientované strukturování je v kap. 4 a 5 zavedeno s ohledem na možnou transformaci do Petriho sítě bez objektů).

Praktickým výsledkem je definice jazyka PNtalk a doporučení pro implementaci systému, který umožní objektově orientované modelování a simulaci paralelních systémů (viz kap. 6) [Jan94a, Jan94b, Jan95b, ČDJ95, Jan96, JŠV96, ČJ96, Voj97b, ČJV97a, ČJV97b, ČJV98]. Systém PNtalk zahrnuje jazyk (PNtalk), simulátor a vývojové prostředí (browser a debugger). Jazyk PNtalk přímo vychází z definice OOPN. Jde o graficko-textový jazyk. Grafická část tohoto jazyka se opírá o vysokoúrovňové Petriho sítě, textová část (inskripční jazyk) vychází z jazyka Smalltalk. Programátor má k dispozici browser, umožňující editovat jednotlivé sítě uvnitř tříd, a debugger, umožňující interaktivně řídit simulaci. Simulátor je též schopen automatické simulace, a to jak v simulačním, tak v reálném čase. Experimentální prototypová verze systému PNtalk byla implementována jako součást prostředí VisualWorks 2.0 (prostředí pro vývoj aplikací ve Smalltalku). Významným rysem této implementace jazyka a systému PNtalk je možnost interakce s objekty, které jsou implemenovány jinak než Petriho sítěmi (tj. přímo ve Smalltalku). Díky tomu je možná i tvorba modelů, komunikujících s okolím, čehož lze využít v rychlém prototypování, případně v řídicích aplikacích.

Aplikace a možnosti dalšího výzkumu. Programovací techniky pro OOPN a PNtalk byly prezentovány v kapitole 7 na řadě příkladů. Šlo vesměs o různé modifikace systému večeřících filosofů. Příklady však byly vytvořeny tak, aby demonstrovaly modelování typických aspektů reálných systémů, kvůli kterým byly OOPN a PNtalk vyvinuty, což je klient-server komunikace aktivních objektů a synchronizace aktivit ve víceúrovňových modelech.

Modely reálných systémů však budou obvykle poněkud rozsáhlejší než uvedené příklady. V systému PNtalk byl pokusně vytvořen model pružného výrobního systému [JV98a], čímž byla ověřena aplikovatelnost PNtalku při modelování rozsáhlejších systémů. Součástí navazujícího výzkumu budou i další aplikace podobného druhu. Pro tyto účely je však třeba systém PNtalk reimplementovat, protože současná verze je jen prototypová implementace, určená k demonstraci základních principů objektově orientovaného modelování Petriho sítěmi. Budou zkoumány různé varianty implementace OOPN (například distribuovaná implementace, propojení s jinými systémy pro tvorbu heterogenních modelů atd.).

Zajímavým se jeví použití PNtalku ve výuce.[3] OOPN/PNtalk přirozeně podporuje objektově orientované modelování, založené na třídách (class-based), s dynamicky vznikajícími a zanikajícími objekty, přičemž jde o aktivní víceprocesové objekty. Neobsahuje deklarace a je tedy snadno použitelný. Je vhodný pro první seznámení jak s Petriho sítěmi, tak s objektovou orientací (v uvedeném pořadí). Poněkud nezvyklá syntax inskripčního jazyka (Smalltalku) ve srovnání s běžnými programovacími jazyky by neměla být vážnou překážkou použití PNtalku pro výukové účely. PNtalk již byl pokusně aplikován ve výuce předmětů „Modelování a simulace systémů“ a „Objektově orientované modelování a prototypování“. Pro zkvalitnění výuky bude perspektivně vhodné do systému PNtalk doplnit podporu stochastické simulace (sběry statistik, histogramy atd.) a modul analýzy „černobílých“ Petriho sítí.[4]

Z hlediska současného stavu vědy velmi aktuálním a ambiciózním tématem výzkumu, navazujícího na tuto práci, bude rozvoj teorie OOPN. Definice OOPN, prezentovaná v kapitole 5, položila základy pro budování této teorie. Poznamenejme, že formální definice OOPN prošla složitým vývojem, kdy byl hledán takový styl definování struktury i dynamiky OOPN, který by byl únosný z hlediska čitelnosti i schopnosti dostatečně podrobně popsat nejdůležitější sku-

[3] Některé části této práce mohou být využity jako učební text pro výuku jazyka PNtalk a principů OOPN.

[4] Analýza jiných druhů Petriho sítí je samozřejmě také součástí budoucích výzkumných aktivit.

tečnosti. Je velmi pravděpodobné, že v rámci dalšího výzkumu se objeví i alternativní formy definice OOPN.

Z vlastních i zprostředkovaných[5] zkušeností autora této práce jednoznačně vyplývá, že OOPN lze mnohem lépe implementovat než formálně definovat a budovat na nich teorii. Přesto je již možné zahájit výzkum i v tomto směru. Zárodek teorie, spjaté s OOPN je v [JV98b].

Vzhledem k tomu, že alternativní přístupy k objektové orientaci v Petriho sítích (uvedené v kap. 4) jsou se zde prezentovaným formalismem srovnatelné, lze očekávat, že jejich předpokládané výsledky v oblasti formální analýzy budou použitelné i v kontextu OOPN. Lze též očekávat, že teorie OOPN přispěje k budování obecných teoretických základů objektové orientace.

[5] Viz přehled alternativních přístupů k objektové orientaci v Petriho sítích v kap. 4.

Literatura

[Ame87] P. America. POOL-T: A Parallel Object-Oriented Language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, Cambridge, Massachusets, 1987.

[AS83] G.R. Andrews and F.B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 5(1):3–43, 1983.

[BB95] O. Biberstein and D. Buchs. Structured Algebraic Nets with Object-Orientation. In G. Agha and F. De Cindio, editors, *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency, the 16th International Conference on Application and Theory of Petri Nets*, pages 131–145, Turin, Italy, June 1995.

[BCC95] E. Battison, A. Cizzoni, and F. De Cindio. Inheritance and Concurrency in CLOWN. In G. Agha and F. De Cindio, editors, *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency, the 16th International Conference on Application and Theory of Petri Nets*, Turin, Italy, June 1995.

[BCMR91] E. Battison, F. De Cindio, G. Mauri, and L. Rapanotti. Morphisms and Minimal Models for OBJSA Nets. In *Application and Theory of Petri nets, 12th International Conference*, pages 455–476, Gjern, Denmark, 1991. University of Aarhus, IBM Deutschland.

[Ber93] E.V. Berard. *Essays on Object-Oriented Software Engineering, Volume 1.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[BG91] D. Buchs and N. Guelfi. CO-OPN: A Concurrent Object Oriented Petri Net Approach. In *Application and Theory of Petri nets, 12th International Conference*, pages 432–454, Gjern, Denmark, 1991. University of Aarhus, IBM Deutschland.

[BHJL87] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *Proceedings of OOPSLA '86*, pages 78–86. ACM Sigplan Notices, 1987.

[Boo91] G. Booch. *Object Oriented Design.* The Benjamin/Cummings Publishing Company, 1991.

[Bow96] F.D.J. Bowden. Modelling Time in Petri Nets. In *Proceedings of the second Australia-Japan Workshop on Stochastic Models in Engineering, Technology and Management*, Gold Coast, Australia, July 1996.

[BP95] R. Bastide and Ph. Palanque. Petri Net Objects for the Design, Validation and Prototyping of User-Driven Interfaces. In G. Agha and F. De Cindio, editors, *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency, the 16th International Conference on Application and Theory of Petri Nets*, Turin, Italy, June 1995.

[Car89] Denis Caromel. A General Model for Concurrent and Distributed Object-Oriented Programming. In *Proceedings of OOPSLA '88*, pages 102–104. ACM Sigplan Notices, 1989.

[ČDJ95] M. Češka, R. Drabant, and V. Janoušek. An Integrated Environment for System Specification and Prototyping. In *Proceedings of the 12th International Conference on Systems Science*, pages 53–60. Oficina Wydawnicza Politechniki wroclawkiej, Wroclav, 1995.

[CH94] S. Christensen and N.D. Hansen. Coloured Petri Nets Extended With Channels for Synchronnous Communication. In G. Rosenberg, editor, *Application and Theory of Petri nets, International Conference*, volume 15 of *Lecture Notes in Computer Science 815*. Springer-Verlag, 1994.

[ČJ93] M. Češka and V. Janoušek. On the Hierarchical Petri Net Implementation. In J. Štefan, editor, *Proceedings of MOSIS '93*, Olomouc, Czech Republic, 1993. MARQ Ostrava.

[ČJ96] M. Češka and V. Janoušek. Object Orientation in Petri Nets. In *Object Oriented Modelling and Simulation, Proceedings of the 22nd Conference of the ASU*, pages 69–80, Clermont-Ferrand, France, September 1996. University Blaise Pascal.

[ČJ97] M. Češka and V. Janoušek. A Formal Model for Object-Oriented Petri Nets Modeling. *Advances in Systems Science and Applications, An Official Journal of the International Institute for General Systems Studies*, Special Issue:119–124, 1997.

[ČJV97a] M. Češka, V. Janoušek, and T. Vojnar. PNtalk – A Computerized Tool for Object Oriented Petri Nets Modelling. In F. Pichler and R. Moreno-Díaz, editors, *Computer Aided Systems Theory and Technology – EUROCAST'97. A Selection of Papers from the 6th International Workshop on Computer Aided Systems Theory*, volume 1333 of *Lecture Notes in Computer Science*, pages 591–610, Las Palmas de Gran Canaria, Spain, February 1997. Springer-Verlag.

[ČJV97b] M. Češka, V. Janoušek, and T. Vojnar. PNtalk – A Computerized Tool for Object Oriented Petri Nets Modelling. In F. Pichler and R. Moreno-Díaz, editors, *Proceedings of the 6th International Conference on Computer Aided Systems Theory and Technology – EUROCAST'97*, pages 229–231, Las Palmas de Gran Canaria, Spain, February 1997. C.I.I.C.C Universidad de Las Palmas de Gran Canaria.

[ČJV98] M. Češka, V. Janoušek, and T. Vojnar. Object-Oriented Petri Nets, Their Simulation, and Analysis. In *Proceedings of 1998 IEEE International Conference on Systems, Man, and Cybernetics, SMC'98*, IEEE Catalog Number 98CH36218, pages 256–261, Hyatt La Jolla, San Diego, California, USA, October 1998. Omnipress.

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[CP95] S. Christensen and L. Petrucci. Modular State Space Analysis of Coloured Petri Nets. In G. De Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 201–217, Turin, Italy, June 1995. Springer-Verlag.

[CT93] S. Christensen and J. Toksvig. DesignBeta V2.0.1 - BETA Code Segments in CP-nets. In *Lecture Notes OO&CPN - nr 5*, Computer Science Department, Aarhus University, 1993.

[DA92] R. David and H. Alla. *Petri Nets and Grafcet*. Prentice Hall, New York, 1992.

[EMK87] J. Eliot, B. Moss, and W.H. Kohler. Concurrency Features for the Trellis/Owl Language. In *Proceedings of ECOOP '87*, pages 223–232. BIGRE, 1987.

[Ess97a] R. Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, Institute TIK, Electrical Engineering Department, ETH Zurich, 1997. Available as TIK-Schriftenreihe Nr. 16, vdf Verlag 8092 Zurich.

[Ess97b] R. Esser. An Object Oriented Petri Net Language for Embedded System Design. In *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice*, London, UK, July 1997.

[Fla96] D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., 1996.

[Gen87] H.J. Genrich. Predicate/Transition Nets. In W. Brauer and W. Reisig, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, 1987.

[Gio91] R. Di Giovanni. HOOD Nets. In G. Rosenberg, editor, *Advances in Petri nets 1991*, volume 483 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[GR83] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison-Wesley, 1983.

[HH95] T. Hopkins and B. Horan. *Smalltalk: an introduction to application development using VisualWorks*. Prentice Hall, 1995.

[HJS90] P. Huber, K. Jensen, and R.M. Shapiro. Hierarchies in Coloured Petri Nets. In G. Rosenberg, editor, *Advances in Petra nets 1990*, volume 483 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Hol95a] T. Holvoet. Agents and Petri Nets. *The Petri Net Newsletter*, (49):3–8, 1995.

[Hol95b] T. Holvoet. PN-TOX: a Paradigm and Development Environment for Object Concurrency Specification. In G. Agha and F. De Cindio, editors, *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency, the 16th International Conference on Application and Theory of Petri Nets*, Turin, Italy, June 1995.

[HV96] T. Holvoet and P. Verbaeten. Using Petri Nets for Specifying Active Objects and Generative Communication. In *Proceedings of Workshop on Object-Oriented Programming and Models of Concurrency*, Osaka, Japan, 1996.

[Jan94a] V. Janoušek. Merging Petri Nets and Objects. In *Proceedings of International Winter School SOFSEM '94*, pages 45–50, 1994.

[Jan94b] V. Janoušek. OOPN: A High-Level Language for Modeling and Simulation. In J. Štefan, editor, *Proceedings of MOSIS '94*. MARQ Ostrava, 1994.

[Jan95a] V. Janoušek. Functional and Object Oriented Structuring of Petri Nets. In *Proceedings of International Conference on Computer Science*, pages 44–47, Ostrava, Czech Republic, September 1995. Technical University of Ostrava.

[Jan95b] V. Janoušek. PNtalk: Object Orientation in Petri Nets. In *Proceedings of European Simulation Multicoference ESM '95*, pages 196–200, Prague, Czech Republic, 1995. Czech Technical University.

[Jan96] V. Janoušek. The PNtalk Language and System. In J. Štefan, editor, *Proceedings of MOSIS '96*, pages 233–238. MARQ Ostrava, 1996.

[Jan97a] V. Janoušek. Objektově orientované modelování paralelních systémů. In J. Štefan, editor, *Sborník ASIS '97*, pages 365–371. MARQ Ostrava, 1997.

[Jan97b] V. Janoušek. Objektově orientované Petriho sítě – formální základ jazyka PNtalk. Technical report, Department of Computer Science and Engineering, Faculty of Electrical Engineering and Computer Science, Technical University of Brno, Czech Republic, January 1997. (In Czech).

[Jan97c] V. Janoušek. Reflective Approach to Petri Net Simulation. In J. Štefan, editor, *Proceedings of MOSIS '97*, pages 209–304. MARQ Ostrava, 1997.

[Jen92] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 1: Basic Concepts.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[JŠV96] V. Janoušek, L. Šmíd, and T. Vojnar. PNtalk - systém pro objektově orientované modelování Petriho sítěmi. In J. Štefan, editor, *Proceedings of 18th International Workshop on Advanced Simulation of Systems ASS '96*, pages 247–252, Zábřeh na Moravě, Czech Republic, September 1996. MARQ Ostrava. (In Czech).

[JV97] V. Janoušek and T. Vojnar. PNtalk Project, 1997.
`http://www.fee.vutbr.cz/~janousek/pntalk/pntalk.html`
`http://www.daimi.au.dk/~petrinet/tools/db/pntalk.html`.

[JV98a] V. Janoušek and T. Vojnar. Modelling a Flexible Manufacturing System. In J. Štefan, editor, *Proceedings of MOSIS '98*, pages 195–200. MARQ Ostrava, 1998.

[JV98b] V. Janoušek and T. Vojnar. State Spaces of Object-Oriented Petri Nets. In *Proc. of MFCS '98 Workshop on Concurrency*, pages 87–96. FI MU, Brno, 1998.

[Kin80] E. Kindler. *Simulační programovací jazyky.* SNTL, Praha, 1980.

[KL89] D.G. Kafura and K.H. Lee. Inheritance in Actor Based Concurrent Object-Oriented Languages. In S. Cook, editor, *Proceedings of ECOOP '89.* Cambridge University Press, 1989.

[Lak91] C.A. Lakos. LOOPN – Language for Object-Oriented Petri Nets. Technical report, Computer Science Department, University of Tasmania, 1991.

[Lak94] C.A. Lakos. Object Petri Nets – Definition and Relationship to Coloured Petri nets. Technical report, Computer Science Department, University of Tasmania, 1994.

[Lak95] C.A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *Application and Theory of Petri nets, 16th International Conference*, pages 278–297. Springer-Verlag, 1995.

[LK94] C.A. Lakos and C.D. Keen. LOOPN++: A New Language for Object-Oriented Petri Nets. Technical Report R94-4, Department of Computer Science, University of Tasmania, Hobart, Tasmania 7001, April 1994.

[Mai97] C. Maier. Objektorientierte Modellierung mit Petrinetzen. Technical report, Universitat Hamburg, 1997. `http://www.pst.informatik.uni-muenchen.de/~cmaier`.

[MBC+95] M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley and sons, 1995.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1988.

[Min85] M. Minsky. *The Society of Mind*. Syman and Shuster, New York, 1985.

[MM97] C. Maier and D. Moldt. Object Coloured Petri Nets - a Formal Technique for Object Oriented Modelling. In M.O. Stehr B. Farwer, D. Moldt, editor, *Petri Nets in System Engineering, Modelling, Verification and Validation PNSE '97*, pages 11–19. Univ. Hamburg, 1997.

[Mol95] D. Moldt. OOA and Petri Nets for System Specification. In G. Agha and F. De Cindio, editors, *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency, the 16th International Conference on Application and Theory of Petri Nets*, Turin, Italy, June 1995.

[MW97] D. Moldt and Frank Weinberg. Multi-Agent-Systems Based on Coloured Petri Nets. In G. Balbo P. Azéma, editor, *Application and Theory of Petri nets, 18th International Conference, ICATPN '97*, pages 83–101. Springer-Verlag, 1997.

[Nie87] O.M. Nierstrasz. Active Objects in Hybrid. In *Proceedings of OOPSLA '87*, pages 243–253. ACM Sigplan Notices, 1987.

[Nie89] O. Nierstrasz. A Survey of Object-Oriented Concepts. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 3–21. ACM Press and Addison-Wesley, 1989.

[Nie93] O. Nierstrasz. Composing Active Objects. In A. Yonezawa G. Agha, P. Wegner, editor, *Research Directions in Object-Based Concurrency*, pages 151–171. MIT Press, 1993.

[Obe87] H. Oberquelle. Human-machine Interaction and Role/Function/Action Nets. In W. Reisig W. Brauer and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, 1987.

[Pap89] M. Papathomas. Concurrency Issues in Object-Oriented Languages. In D. Tsichritzis, editor, *Object Oriented Development*, pages 207–245. Centre Universitaire d'Informatique, University of Geneva, 1989.

[PB95] Ph. Palanque and R. Bastide. A Formalism for Reliable User Interfaces. In G. Agha and F. De Cindio, editors, *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency, the 16th International Conference on Application and Theory of Petri Nets*, Turin, Italy, June 1995.

[Per96] P. Peringer. *Hierarchické modelování na bázi komunikujících objektů.* PhD thesis, FEI VUT Brno, 1996.

[Pet62] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für Instrumentelle Mathematik, University Bonn, Germany, 1962. Available as Schriften des IIM Nr. 2. (In German).

[Pet81] J. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice Hall, Engelwood Cliffs, New Jersey, 1981.

[RBP+91] J. Rumbaugh, M. Blaha, M. Premeralani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Engelwood Cliffs, New Jersey 07632, 1991.

[Rei85] W. Reisig. *Petri Nets – an Introduction.* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1985.

[SB94] C. Sibertin-Blanc. Cooperative Nets. In R. Valette, editor, *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 471–490, Zaragoza, Spain, June 1994. Springer-Verlag.

[SBHT95] C. Sibertin-Blanc, N. Hameurlain, and P. Touzeau. A C++ Implementation of CoOperative Objects. In G. Agha and F. De Cindio, editors, *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency, the 16th International Conference on Application and Theory of Petri Nets*, Turin, Italy, June 1995.

[Sho93] Y. Shoam. Agent-Oriented Programming. *Artificial Intelligence*, (60):51–92, 1993.

[SSW95a] S. Schöf, M. Sonnenschein, and R. Wieting. Efficient Simulation of THOR Nets. In G. De Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 103–120, Turin, Italy, June 1995. Springer-Verlag.

[SSW95b] S. Schöf, M. Sonnenschein, and R. Wieting. High-level Modeling with THORNs. In *Proceedings of the 14th International Congress on Cybernetics*, pages 453–458, Namur, Belgium, August 1995.

[US87] D. Ungar and R.B. Smith. Self: The Power od Simplicity. *SIGPLAN Notices*, 22(12), December 1987.

[Val95] R. Valk. Petri Nets as Dynamical Objects. In *Proceedings of Workshop on Object-Oriented Programming and Models of Concurrency*, Torino, Italy, 1995.

[Val98] R. Valk. Petri Nets as Token Objects. In M. Silva J. Desel, editor, *Application and Theory of Petri nets, 18th International Conference, ICATPN'98*, pages 1–25. Springer-Verlag, 1998.

[Vla96] B. Vlašínová. Grafický editor a animátor OOPN. Diplomová práce. ÚIVT FEI VUT Brno, 1996. (In Czech).

[Voj95] T. Vojnar. Analýza OOPN. Ročníkový projekt. ÚIVT FEI VUT Brno, 1995. (In Czech).

[Voj96] T. Vojnar. Simulátor OOPN ve Smalltalku. Diplomová práce. ÚIVT FEI VUT Brno, 1996. (In Czech).

[Voj97a] T. Vojnar. Hierarchical and Time Extensions of Pure Object Oriented Petri Nets. In J. Štefan, editor, *Proceedings of 19th International Workshop on Advanced Simulation of Systems ASIS '97*, pages 321–326, Krnov, Czech Republic, September 1997. MARQ Ostrava.

[Voj97b] T. Vojnar. Systém PNtalk. Technical report, Department of Computer Science and Engineering, Faculty of Electrical Engineering and Computer Science, Technical University of Brno, Czech Republic, January 1997. (In Czech).

[Voj98] T. Vojnar. PhD Thesis Outline. DCSE FEI VUT Brno, 1998. (not published).

[Von95a] I. Vondrák. Interaction Coordination Nets. In *Proceedings of International Conference on Computer Science*, Ostrava, Czech Republic, September 1995. Technical University of Ostrava.

[Von95b] I. Vondrák. System Simulation by Interaction Coordination Nets. In *Proceedings of European Simulation Multicoference ESM '95*, pages 206–210, Prague, Czech Republic, 1995. Czech Technical University.

[Von97] I. Vondrák. Coordination of Object Interactions for Process Modeling, Simulation and Enactment. In J. Štefan, editor, *Proceedings of MOSIS '97*, pages 3–7. MARQ Ostrava, 1997.

[Von98] I. Vondrák. Cellular Process. In J. Štefan, editor, *Proceedings of MOSIS '98*, pages 67–74. MARQ Ostrava, 1998.

[Š96] L. Šmíd. Simulátor OOPN v Prologu. Diplomová práce. ÚIVT FEI VUT Brno, 1996. (In Czech).

[Wie96] R. Wieting. Modeling and Simulation of Hybrid Systems Using Hybrid High-level Nets. In A.G. Bruzzone and E.J.H. Kerckhoffs, editors, *Proceedings of the 8th European Simulation Symposium (ESS'96)*, pages 158–162, Genoa, Italy, October 1996.

[YBS87] A. Yonezawa, J.P. Briot, and E. Shibayama. Object-Oriented Concurrent programming in ABCL/1. In *Proceedings of OOPSLA '86*, pages 258–268. ACM Sigplan Notices, 1987.

Příloha A

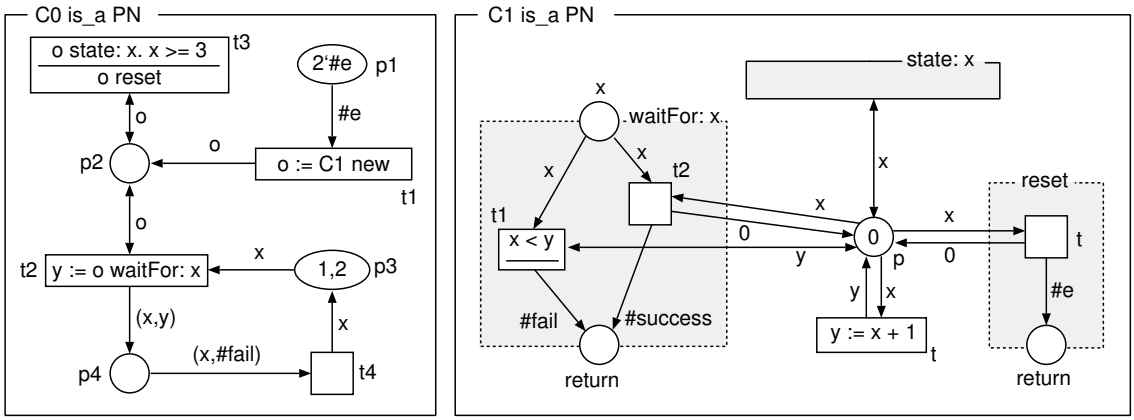
Syntax jazyka PNtalk

Textová verze jazyka PNtalk definuje kromě syntaxe inskripčního jazyka též syntax popisu struktury objektově orientované Petriho sítě.[1] 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]*
```

[1] Textový popis struktury objektově orientované Petriho sítě bývá obvykle automaticky generován nástrojem pro vizuální programování v jazyce PNtalk.

```
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 OOPN. Počáteční třída je C0.

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:

```
main C0

class C0 is_a PN
   object
      trans t4
         precond p4((x, #fail))
         postcond p3(x)
      trans t2
         cond p2(o)
         precond p3(x)
         action {y := o waitFor: x}
         postcond p4((x, y))
      trans t1
         precond p1(#e)
         action {o := C1 new.}
         postcond p2(o)
      trans t3
         cond p2(o)
         guard {o state: x. x >= 3}
         action {o reset.}
      place p1(2'#e)
      place p2()
      place p4()
      place p3(1, 2)
```

```
class C1 is_a PN
   object
      place p(0)
      trans t
         precond p(x)
         action {y := x + 1.}
         postcond p(y)
   method waitFor: x
      place return()
      place x()
      trans t1
         cond p(y)
         precond x(x)
         guard {x < y}
         postcond return(#fail)
      trans t2
         precond x(x), p(x)
         postcond return(#success), p(0)
   method reset
      place return()
      trans t
         precond p(x)
         postcond return(#e), p(0)
   sync state: x
      cond p(x)
```

Příloha B

Dynamika OOPN

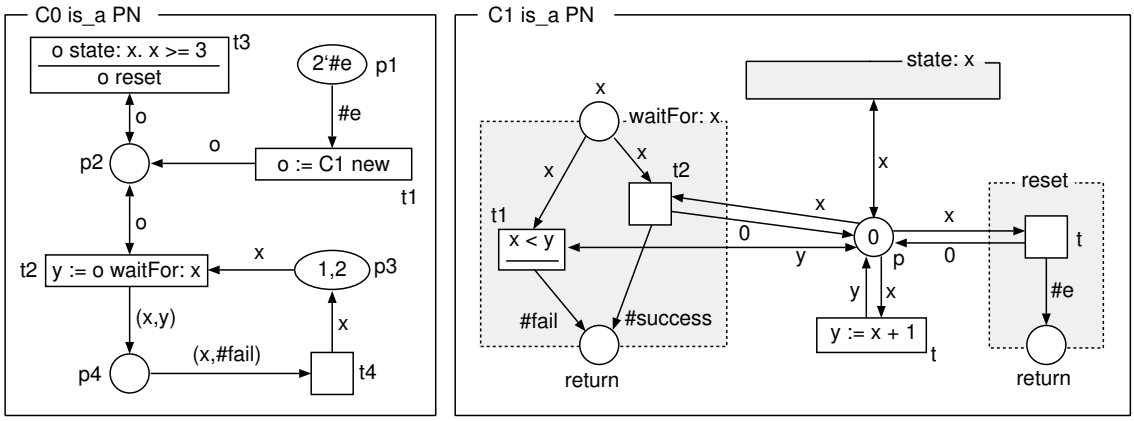
Příklad OOPN, specifikované v jazyce PNtalk, je na obr. B.1. Příklad byl vytvořen tak, aby demonstroval většinu rysů OOPN a aby bylo možné snadno objasnit dynamiku tohoto modelu. Nemodeluje žádný smysluplný systém.

Poznamenejme, že syntaktická rozšíření jazyka PNtalk oproti definici OOPN, tj. složené zprávy, sekvence výrazů v akci přechodu a konstruktory, je třeba pro potřeby sledování běhu modelu transformovat do podoby, která odpovídá definici OOPN. Prakticky to znamená, že například sekvence zasílání zpráv odpovídá množině skrytých přechodů a skrytých míst, jejichž značení je třeba také brát v úvahu. Zde uvedený příklad však zcela odpovídá formální definici.

Jsou zde třídy `C0` a `C1`. Nechť `C0` je počáteční třída. Třída `C0` obsahuje jen síť objektu. Třída `C1` obsahuje síť objektu, obsahující místo `p` a přechod `t`, a dále obsahuje predikát `state:` a metody `waitFor:` a `reset`. Šedé oblasti vymezují ty části sítí metod, které nejsou sdíleny se sítí objektu. Parametrová místa jsou v souladu s požadavky jazyka PNtalk pojmenována stejně jako formální parametry metody. Výstupní místo metody má, opět v souladu s požadavky jazyka PNtalk, jméno `return`.

Evoluce (běh) OOPN začíná implicitním vytvořením prvotního objektu, který je instancí prvotní třídy. Ten obsahuje instanci sítě objektu, která je inicializována počátečním značením. Veškerá dynamika OOPN spočívá v provádění přechodů uvnitř běžících instancí sítí, čímž systém přechází od jednoho stavu k dalšímu. Veškeré změny stavu probíhají atomicky ve formě *událostí*. Události souvisejí s prováděním přechodů podle formální definice OOPN, která rozlišuje čtyři typy událostí:

- A-událost (událost typu A), což je klasické provedení přechodu uvnitř jedné instance sítě nebo synchronně ve více instancích sítí současně (je-li ve stráži specifikováno volání synchronního portu). A-událost může nastat tehdy, když adresátem zprávy v akci proveditelného přechodu je primitivní objekt nebo je akce prázdná.



Obrázek B.1: Příklad OOPN. Počáteční třída je C0.

- N-událost, tedy vytvoření nového objektu v rámci provedení přechodu. N-událost může nastat tehdy, když adresátem zprávy v akci proveditelného přechodu je třída a selektoru zprávy ve třídě adresáta přísluší konstruktor.
- F-událost – předání zprávy, tedy vytvoření instance sítě metody při provedení vstupní části přechodu. F-událost může nastat tehdy, když neprimitivní objekt, který je adresátem zprávy, specifikované v akci proveditelného přechodu, má implementovanou metodu pro tuto zprávu.
- J-událost – akceptování výsledku zprávy při současném zrušení instance sítě metody a provedení výstupní části přechodu.

Uvedené události jsou vázány k provádění přechodů. Mluvíme proto (podle formální definice OOPN) o a-proveditelnosti a a-provedení, n-proveditelnosti a n-provedení, f-proveditelosti a f-provedení, j-proveditelnosti a j-provedení příslušného přechodu.

Dvojice událostí typu F a J je navzájem kauzálně svázána a souvisí s neatomickým provedením přechodu při předání zprávy neprimitivnímu objektu. Fáze čekání mezi událostmi typu F a J se ve stavu systému projeví příslušným značením přechodu.

Implicitní součástí provedení každé události je tzv. garbage-collecting, neboli odstranění všech instancí sítí (potažmo objektů), které nejsou (ani nepřímo) referencovány z prvotního objektu.

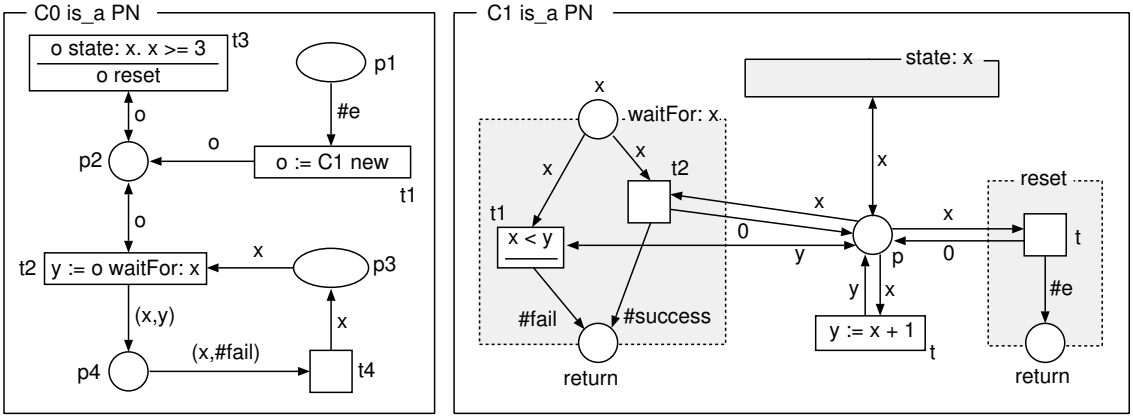
Události postupně modifikují stav systému. V jednom okamžiku (v daném stavu) může potenciálně nastat několik událostí (je zde několik různě proveditelných přechodů), ze kterých se nedetrministicky vybere a uskuteční jen jedna.[1] Stav tohoto nedeterministického stroje je určen množinou objektů, skládajících se z rozpracovaných instancí sítí, jejichž stav je dán aktuálním značením příslušných sítí. Instance sítě je dvojice (id, m), kde id je identifikace (číslo, jméno) instance sítě a m je její značení, tj. informace o stavu (obsahu) uzlů sítě. Každému jménu instance sítě je jednoznačně přiřazena síť, každé síti přísluší množina jmen jejích (potenciálních) instancí. Objekt (instance) třídy C je množina instancí sítí. Jednou z nich je instance sítě objektu třídy C, ostatní jsou právě rozpracované instance sítí metod třídy C. Objekt je identifikován tímtéž jménem jako instance jeho sítě objektu. Každé místo v instanci sítě představuje multimnožinu značek, identifikujících objekty (primitivní, neprimitivní i třídy) a každému přechodu přísluší množina invokací (rozpracovaných metod).

Stav systému budeme specifikovat formou tabulky podle tohoto vzoru:

	Horní záhlaví tabulky. V prvním řádku jsou jména objektů, ve druhém řádku jsou jména instancí sítí, patřící jednotlivým objektům.
Levé záhlaví tabulky. V prvním sloupci jsou jména tříd. Ve druhém sloupci jsou sítě, z nichž se jednotlivé třídy skládají. Sítě metod jsou identifikovány příslušnými selektory zpráv. Síť objektu je identifikována jako „*object net*“. Jsou zde i zděděné metody (v případě OOPN na obr. B.1 však žádné nejsou). Je-li zděděná metoda předefinována, je zde uvedena s prefixem *super*. Ve třetím sloupci jsou uzly jednotlivých sítí (v sítích metod jsou uvedeny jen ty uzly, které nejsou sdíleny se sítí objektu).	Tělo tabulky. Obsahuje aktuální značení míst a přechodů v jednotlivých instancích sítí.

Počáteční stav programu (modelu) je určen stavem prvotního objektu, který je instancí prvotní třídy. Je tedy vytvořen na základě počátečního značení sítě objektu třídy `C0`. Počátečnímu objektu, stejně tak jako instanci odpovídající sítě objektu, přidělíme identifikaci (jméno) id_0. Počáteční stav celého systému, popsaného výše uvedenou OOPN, označíme S_0 a specifikujeme ho následující tabulkou (kvůli lepší čitelnosti budeme spolu s tabulkou zobrazovat i samotnou OOPN, ale bez počátečního značení):

[1] Je též možné simulaci provádět tak, že se provedou všechny bezkonfliktní události současně (v jednom kroku). Tento pohled na dynamiku OOPN nebyl zahrnut do definice, ale jeho zavedení nic nebrání – lze to učinit analogicky k (například) CPN [Jen92].



S_0			id_0
			id_0
C0	*object net*	p1	2‘#e
		p2	empty
		p3	1, 2
		p4	empty
		t1	empty
		t2	empty
		t3	empty
C1	*object net*	p	
		t	
	waitFor:	x	
		return	
		t1	
		t2	
	reset	return	
		t	

Vzhledem k tomu, že rozsah platnosti jmen míst a přechodů je omezen sítí v níž se vyskytují, budeme uzly sítě objektu globálně identifikovat formou *"jméno třídy"*::*"jméno uzlu"* a uzly sítě metody formou *"jméno třídy"*::*"selektor zprávy"*::*"jméno uzlu"*.

Ve stavu S_0 je proveditelný jediný přechod, a to `C0::t1` v instanci sítě, identifikované jménem id_0. Je takzvaně n-proveditelný, tj. je schopen vytvořit nový objekt, protože obsahuje akci „`o := C1 new`“. Přechod neobsahuje žádné vstupní proměnné, navázání proměnných je tedy prázdné. Událost, odpovídající n-provedení tohoto přechodu, je určena čtveřicí

(N, id_0, `C0::t1`, {}).

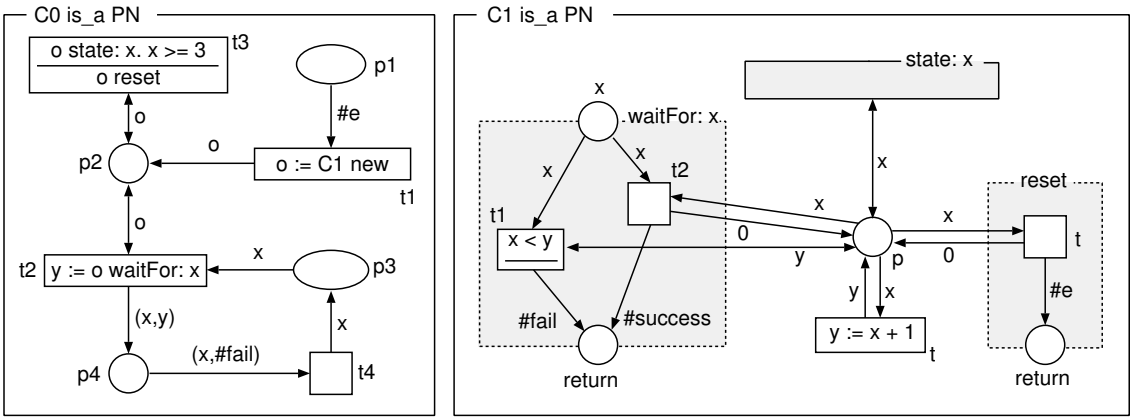
První složkou čtveřice je typ události, druhou složkou je jméno instance sítě, třetí složkou je jméno přechodu a poslední složkou je navázání proměnných přechodu. Dojde-li k uvedené události, stav systému se změní takto:

- V důsledku provedení akce přechodu vznikne nový objekt, kterému přidělíme jméno id_1. Stejné jméno bude mít i odpovídající instance sítě objektu. Dílčí stav, odpovídající instanci této sítě je dán počátečním značením, jak je specifikováno ve výše uvedené OOPN (místo `C1::p` má počáteční značení `0`).
- Dojde ke změně stavu instance sítě id_0 (změní se značení míst `p1` a `p2`; místo `p2` bude obsahovat identifikaci nově vzniklého objektu id_1).

Označíme-li výsledný stav S_1, realizaci uvedené události zapíšeme takto:

S_0 [N, id_0, `C0::t1`, {} ⟩ S_1.

Stav S_1 je popsán následující tabulkou:



S_1			id_0	id_1
			id_0	id_1
C0	*object net*	p1	#e	
		p2	id_1	
		p3	1, 2	
		p4	empty	
		t1	empty	
		t2	empty	
		t3	empty	
C1	*object net*	p		0
		t		empty
	waitFor:	x		
		return		
		t1		
		t2		
	reset	return		
		t		

V tomto stavu mohou nastat následující události:

- (N, id_0, C0::t1, {}) – přechod C0::t1 v instanci sítě, pojmenované id_0 je n-proveditelný,
- (F, id_0, C0::t2, {(x, 1), (o, id_1)}) – přechod C0::t2 v instanci sítě id_0 je f-proveditelný (protože je schopen invokovat metodu waitFor:) pro navázání proměnných {(x, 1), (o, id_1)},
- (F, id_0, C0::t2, {(x, 2), (o, id_1)}) – přechod C0::t2 v instanci sítě id_0 je f-proveditelný pro navázání proměnných {(x, 2), (o, id_1)},
- (A, id_1, C1::t, {(x, 0)}) – přechod C1::t v instanci sítě, identifikované jménem id_1, je a-proveditelný pro navázání proměnných {(x, 0)}.

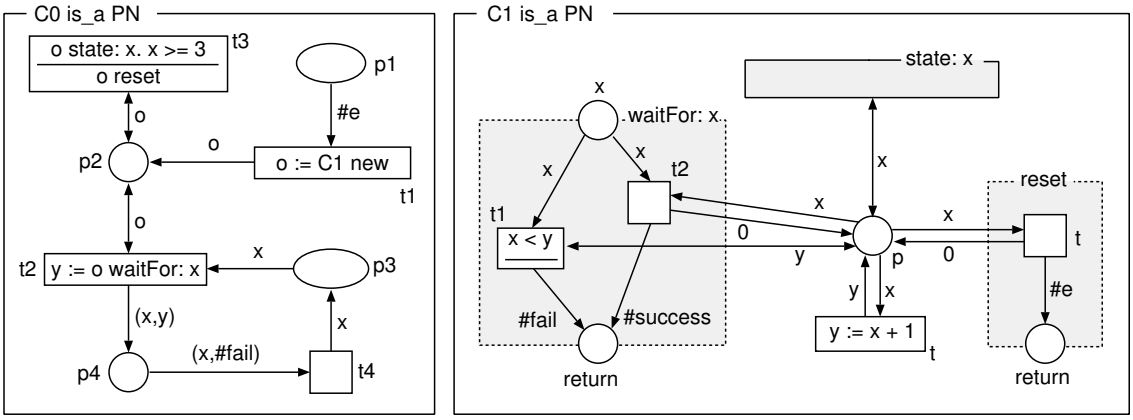
Dojde-li k f-události

$$S_1 \text{ [F, } id_0\text{, C0::t2, \{(x, 1), (o, } id_1\text{)\} } \rangle \text{ } S_2,$$

dojde ke změně stavu systému, a sice takto:

- Přechod C0::t2 v instanci sítě id_0 odebere potřebné značky ze vstupních míst (ale jen pokud se nejedná o testovací hrany, znázorněné obousměrnými šipkami).
- Na základě akce „y := o waitFor: x“ dojde k vytvoření nové instance sítě metody C1::waitFor: v objektu, identifikovaném jménem id_1. Této instanci přidělíme identifikaci id_2. Přechod C0::t2 si zapamatuje vytvořenou instanci sítě a navázání svých proměnných – značení tohoto přechodu bude (id_2, {(x, 1), (o, id_1)}). Neprázdné značení přechodu indikuje, že je rozpracovaný (čeká na ukončení invokované metody). Očekává se, že někdy v budoucnu dojde k dokončení tohoto přechodu (půjde o tzv. j-událost, která bude vysvětlena později).

Výsledný stav S_2 je popsán takto:



S_2			id_0	id_1	
			id_0	id_1	id_2
C0	*object net*	p1	#e		
		p2	id_1		
		p3	2		
		p4	empty		
		t1	empty		
		t2	(id_2, {(x, 1), (o, id_1)})		
		t3	empty		
C1	*object net*	p		0	
		t		empty	
	waitFor:	x			1
		return			empty
		t1			empty
		t2			empty
	reset	return			
		t			

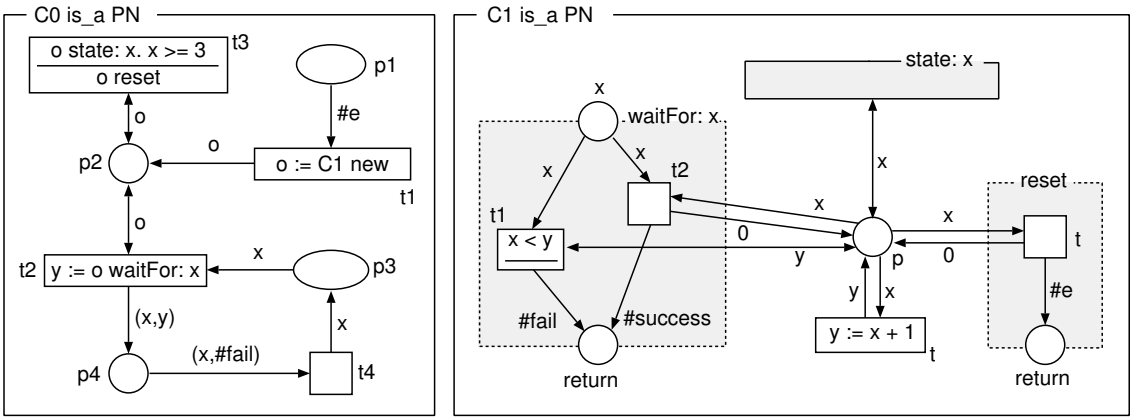
V tomto stavu mohou nastat následující události:

- (N, id_0, C0::t1, {}) – přechod C0::t1 v instanci sítě, pojmenované id_0 je n-proveditelný,
- (F, id_0, C0::t2, {(x, 2), (o, id_1)}) – přechod C0::t2 v instanci sítě id_0 je f-proveditelný pro navázání proměnných {(x, 2), (o, id_1)},
- (A, id_1, C1::t, {(x, 0)}) – přechod C1::t v instanci sítě, identifikované jménem id_1, je a-proveditelný pro navázání proměnných {(x, 0)}.

Abychom ukázali možnost násobného vyvolání téže metody uvnitř jednoho objektu, necháme provést ještě jednou f-událost. Realizací události

$$S_2 \ [\ \mathrm{F},\ id_0,\ \texttt{C0::t2},\ \{(\texttt{x},\ 2),\ (\texttt{o},\ id_1)\}\ \rangle\ S_3$$

dojde k vytvoření další instance sítě metody C1::waitFor: v objektu, identifikovaném jménem id_1. Této instanci sítě přidělíme jméno id_3. Následující stav systému ukazuje tabulka:



S_3			id_0	id_1		
			id_0	id_1	id_2	id_3
C0	*object net*	p1	#e			
		p2	id_1			
		p3	empty			
		p4	empty			
		t1	empty			
		t2	$(id_2, \{(x, 1), (o, id_1)\}), (id_3, \{(x, 2), (o, id_1)\})$			
		t3	empty			
C1	*object net*	p		0		
		t		empty		
	waitFor:	x			1	2
		return			empty	empty
		t1			empty	empty
		t2			empty	empty
	reset	return				
		t				

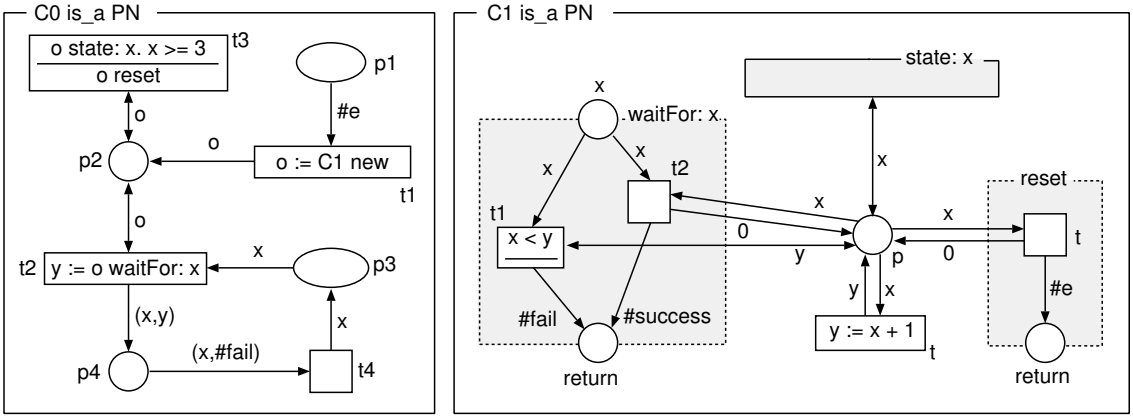
V tomto stavu mohou nastat následující události:

- (N, id_0, C0::t1, {}) – přechod C0::t1 v instanci sítě, pojmenované id_0 je n-proveditelný,
- (A, id_1, C1::t, {(x, 0)}) – přechod C1::t v instanci sítě, identifikované jménem id_1, je a-proveditelný pro navázání proměnných {(x, 0)}.

Provedením události

$$S_3 \ [\ \mathrm{A}, id_1, \texttt{C1::t}, \{(\texttt{x}, \texttt{0})\}\ \rangle\ S_4$$

dojde ke klasickému atomickému provedení přechodu C1::t v instanci sítě, identifikované jménem id_1, což způsobí změnu značení sítě objektu třídy C1 v této instanci. Výsledný stav ukazuje tabulka:



S_4			id_0	id_1		
			id_0	id_1	id_2	id_3
C0	*object net*	p1	#e			
		p2	id_1			
		p3	empty			
		p4	empty			
		t1	empty			
		t2	$(id_2, \{(x, 1), (o, id_1)\}), (id_3, \{(x, 2), (o, id_1)\})$			
		t3	empty			
C1	*object net*	p		1		
		t		empty		
	waitFor:	x			1	2
		return			empty	empty
		t1			empty	empty
		t2			empty	empty
	reset	return				
		t				

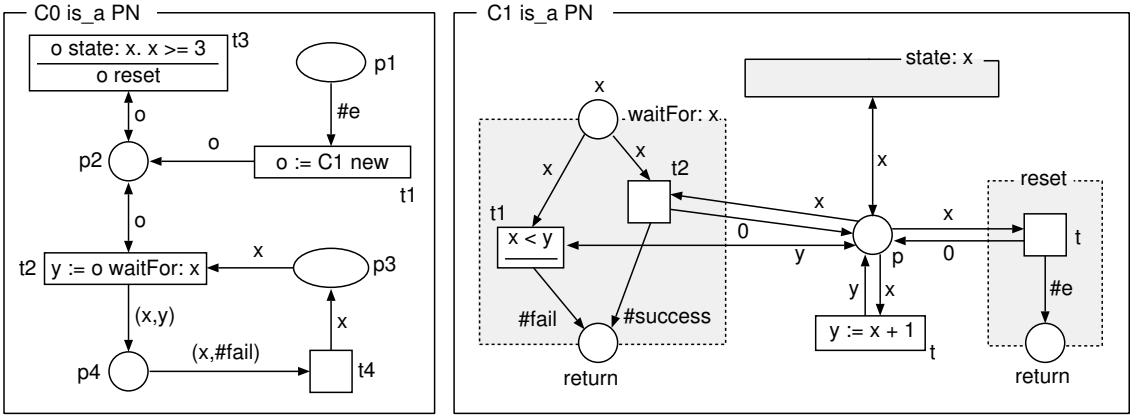
V tomto stavu mohou nastat následující události:

- (N, id_0, C0::t1, {}) – přechod C0::t1 v instanci sítě, pojmenované id_0 je n-proveditelný,
- (A, id_1, C1::t, {(x, 1)}) – přechod C1::t v instanci sítě, identifikované jménem id_1, je a-proveditelný pro navázání proměnných {(x, 1)}.
- (A, id_2, C1:waitFor:::t2, {(x, 1)}) – přechod C1:waitFor::t v instanci sítě, identifikované jménem id_2, je a-proveditelný pro navázání proměnných {(x, 1)}.

Provedením a-události

$$S_4 \; [\; \text{A}, id_2, \texttt{C1:waitFor:::t2}, \{(\texttt{x}, 1)\} \; \rangle \; S_5$$

systém přejde do následujícího stavu:



S_5			id_0	id_1		
			id_0	id_1	id_2	id_3
C0	*object net*	p1	#e			
		p2	id_1			
		p3	empty			
		p4	empty			
		t1	empty			
		t2	$(id_2, \{(x, 1), (o, id_1)\}), (id_3, \{(x, 2), (o, id_1)\})$			
		t3	empty			
C1	*object net*	p		0		
		t		empty		
	waitFor:	x			empty	2
		return			#success	empty
		t1			empty	empty
		t2			empty	empty
	reset	return				
		t				

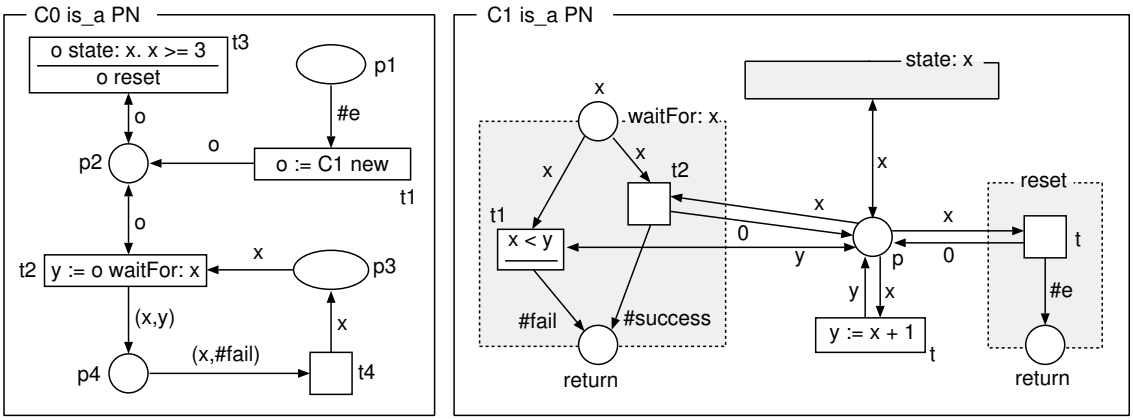
V tomto stavu mohou nastat následující události:

- (N, id_0, C0::t1, {}) – přechod C0::t1 v instanci sítě, pojmenované id_0 je n-proveditelný,
- (A, id_1, C1::t, {(x, 0)}) – přechod C1::t v instanci sítě, identifikované jménem id_1, je a-proveditelný pro navázání proměnných {(x, 0)}.
- (J, id_0, C1::t2, {(x, 1), (o, id_1), (y, #success)}) – přechod C0::t2 v instanci sítě, identifikované jménem id_0, je j-proveditelný pro navázání proměnných {(x, 1), (o, id_1), (y, #success)}.

V instanci id_2 sítě C1::waitFor: bylo v předchozím kroku označeno místo return (obsahuje symbol #success), což znamená, že příslušná metoda skončila a může vrátit výsledek přechodu, který ji v minulosti zavolal. Jde o přechod t2 v instanci id_2 sítě C0. Tento přechod je j-proveditelný pro navázání proměnných {(x, 1), (o, id_1), (y, #success)}, protože jeho značení obsahuje položku (id_2, {(x, 1), (o, id_1)}) a místo return má v instanci id_2 sítě C1::waitFor: neprázdné značení (hodnota značky #success v místě return je pak navázána na proměnnou y přechodu C0::t2). Provedením j-události

$$S_5 \; [\; \text{J}, id_0, \texttt{C0::t2}, \{(\texttt{x}, 1), (\texttt{o}, id_1), (\texttt{y}, \texttt{\#success})\} \; \rangle \; S_6$$

dojde ke zrušení instance id_2 sítě C1::waitFor: a k dokončení (provedení výstupní části) přechodu t2 v instanci id_0 sítě C0. Systém tedy přejde do stavu:



S_6			id_0	id_1	
			id_0	id_1	id_3
C0	*object net*	p1	#e		
		p2	id_1		
		p3	empty		
		p4	(1, #success)		
		t1	empty		
		t2	$(id_3, \{(\mathtt{x}, 2), (\mathtt{o}, id_1)\})$		
		t3	empty		
C1	*object net*	p		0	
		t		empty	
	waitFor:	x			2
		return			empty
		t1			empty
		t2			empty
	reset	return			
		t			

Dále již evoluci OOPN nebudeme uvádět. Poznamenejme jen, že až dosáhne místo `p` v instanci id_1 sítě `C1` značení `3` a více, bude proveditelný i přechod `t3` v instanci id_0 sítě `C0`, protože predikát `state:` v objektu id_1, který je testován ve stráži přechodu `C0::t3`, bude splněn současně s booleovským výrazem `x >= 3`. Pak bude možné přechod `t3` v instanci id_0 sítě `C1` provést. Půjde zřejmě o f-událost, která vyvolá metodu `reset` v kontextu objektu id_1.

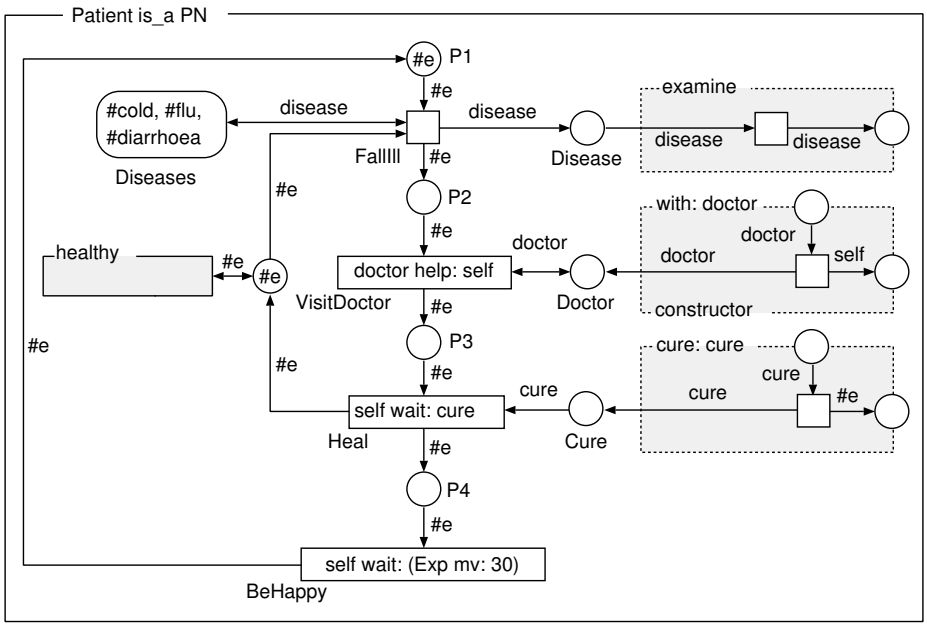
Již na tomto jednoduchém příkladu, demonstrujícím dynamiku OOPN, je vidět, že pro sledování běhu OOPN je potřebný vhodný počítačový nástroj, který je schopen stav systému sugestivně a přehledně prezentovat. Použití tabulek pro papírovou prezentaci lze považovat jen za nouzové řešení.

Příloha C

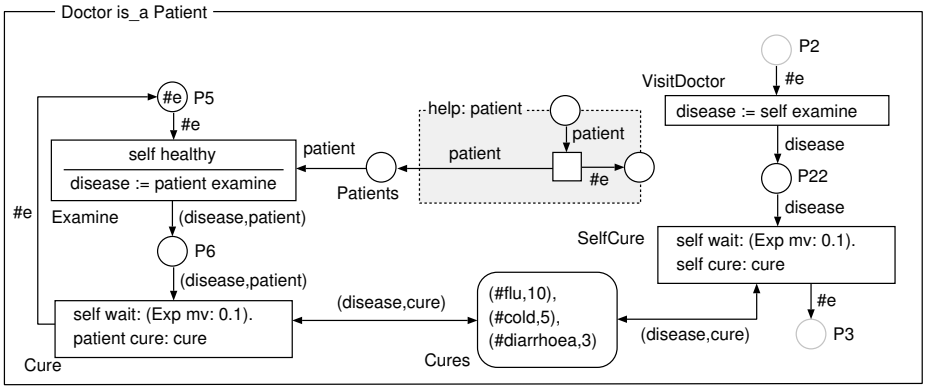
Příklad simulačního modelu

Na tomto místě uvedeme příklad simulačního modelu, používajícího zpožďující přechody, který byl uveden jako součást publikace [ČJV97a]. Komentář k příkladu je ponechán v původní podobě.

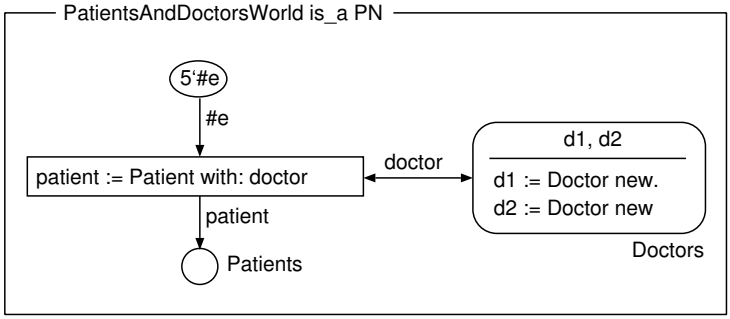
> There is a little bit more complex example given in this section. The ideas of inheritance, active objects and interobject communication are all included in this example. It is a simulation model of a society of patients and doctors. Because of lack of space, the model was rather simplified, but it is no problem to extend it so that it will become more realistic.
>
> In our model, patients are represented by objects of the class `Patient` (Fig. C.1). New patients can be created by the constructor `with: doctor` which also specifies for every patient his doctor. The life cycle of patients is described by their object net. We can see that a patient is initially healthy. Then he falls ill and asks his doctor to help him. This is done by sending the doctor a message with the selector `help:`. After having asked the doctor to help him, our patient has to wait until the doctor examines him and tells him what to do to become healthy again. In our case, doctors only tell the patients how long they must wait before they become healthy. Subsequently, the patients are healthy for some time and then the lifecycle is repeated.
>
> A doctor is a special case of a patient – he can fall ill and recover from the illness but, what is more, he can cure himself and also other patients. Therefore the class `Doctor` (Fig. C.2) is a descendant of the class `Patient`. Doctors inherit the own behaviour of patients and modify it in the sense that they do not require any help from another doctors because they can examine and cure themselves. This modification of patients behaviour concerns the redefined transition `VisitDoctor` and the new transition `SelfCure`. Apart from the inherited lifecycle, the class `Doctor` contains another cycle which is connected to the work of doctors. If a doctor is healthy and some patient announced him some illness, the doctor examines the patient, i.e. sends him the message `examine`, then he finds out how long the patient will have to wait to become healthy again and announces it to the patient. Then this cycle is repeated. The class `PatientsAndDoctorsWorld` (Fig. C.3) represents the top layer of our model. This class is used as the primary class of the OOPN. It states that there will be five patients and those will be treated by two doctors.
>
> It is obvious that our model is rather unrealistic – patients and doctors enjoy ethernal life, patients wait for their doctors without any limits of waiting times, doctors cannot fall ill in the middle of their work and so on. It is possible to improve the model, however this is beyond the scope of this article.



Obrázek C.1: An OOPN class describing patients



Obrázek C.2: An OOPN class describing doctors as special cases of patients



Obrázek C.3: An OOPN class describing a world of doctors and patients

Příloha D

Nástroje programátora

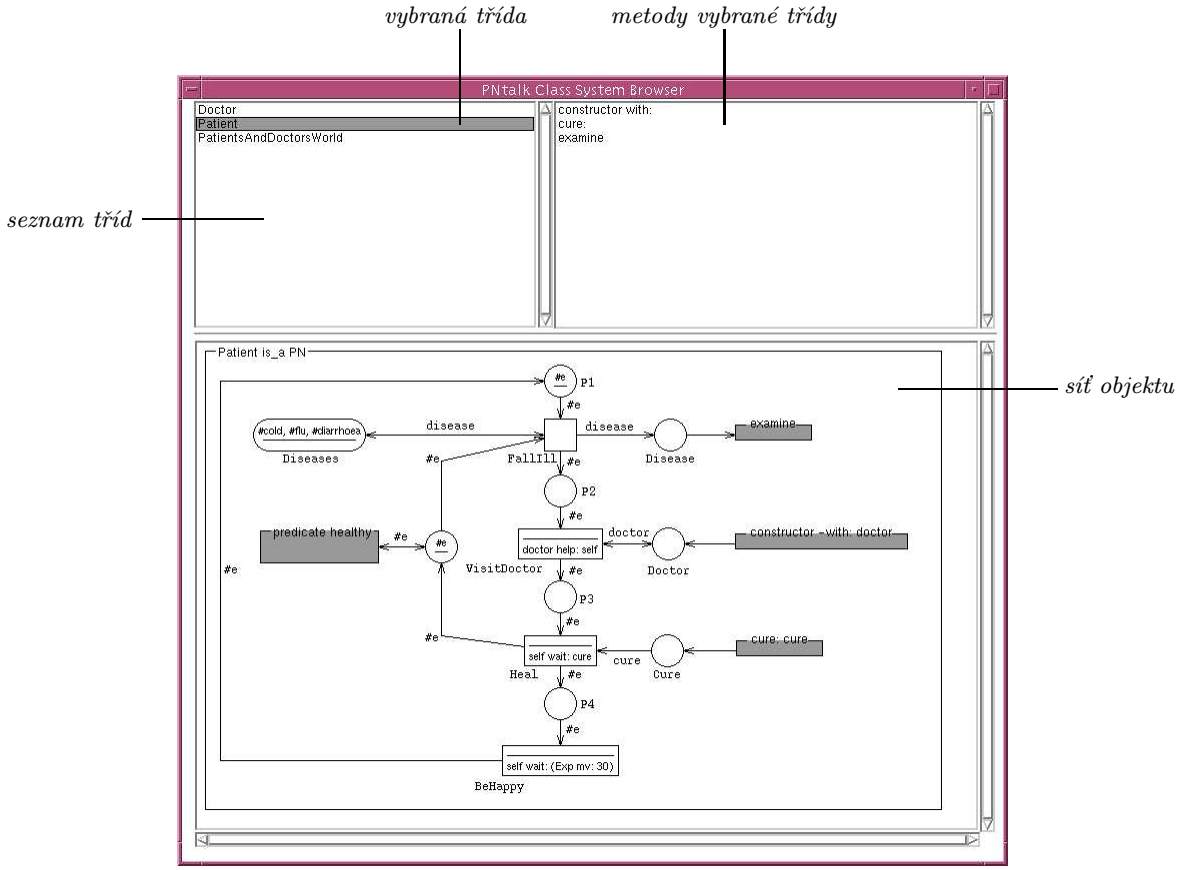
Na tomto místě stručně představíme dva hlavní nástroje programátora v prototypové implementaci systému PNtalk, a sice browser a debugger, zmíněné v kapitole 7.

PNtalk Browser

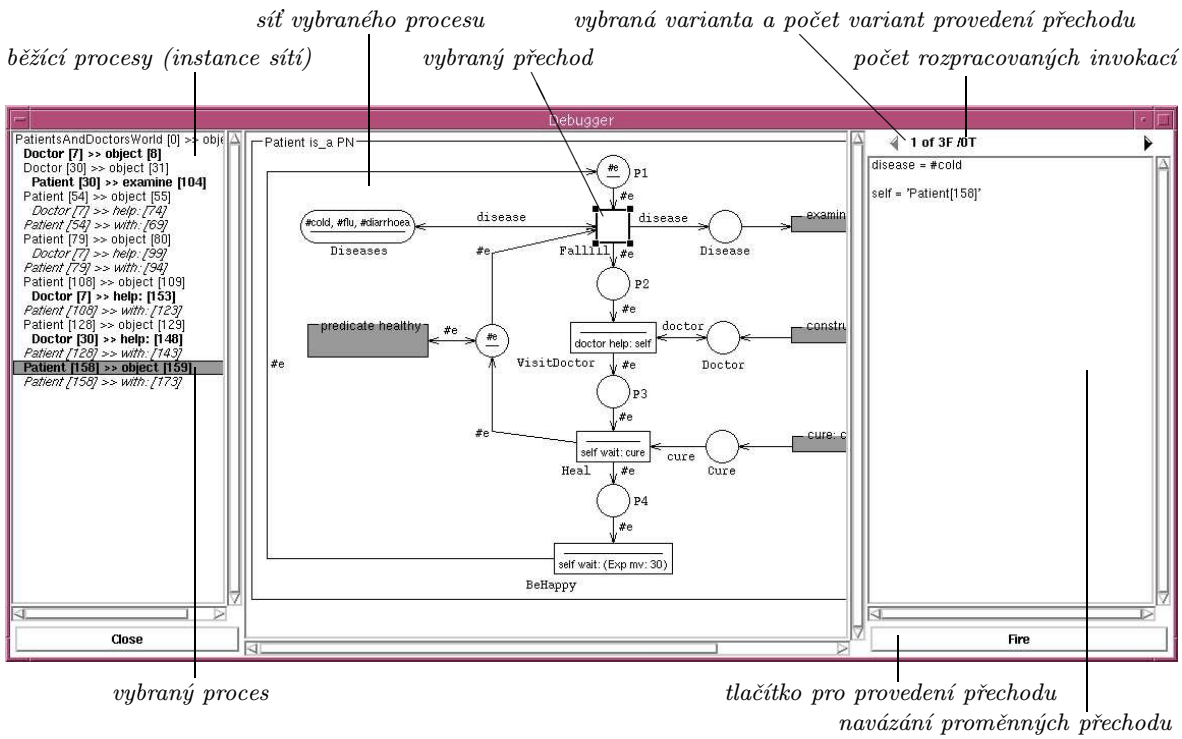
Obecně byl browser popsán v kapitole 7. Browser v prototypové verzi systému PNtalk (viz obr. D.1) je inspirován browserem Smalltalku [GR83]. Je-li v seznamu tříd (vlevo nahoře) vybrána třída, zobrazí se seznam metod (vpravo nahoře) a síť objektu (dole). V síti objektu jsou zobrazeny i fantómy metod (hlavičky metod, propojené se sítí objektu), aby bylo zřejmé, která místa sítě objektu jsou jednotlivými metodami ovlivňována. Po výběru metody se zobrazí síť metody a ta místa sítě objektu, s kterými je metoda propojena. Každou takto zobrazenou síť lze v browseru editovat.

PNtalk Debugger

Debugger byl také obecně popsán v kapitole 7. Na obr. D.2 je uveden debugger, který je součástí prototypové implementace systému PNtalk. Vlevo je zobrazena hierarchie procesů (instancí) sítí (hierarchie odpovídá relaci „kdo koho vytvořil“). Každý proces, ve kterém je alespoň jeden proveditelný přechod, je zvýrazněn. Kurzívou jsou vyznačeny procesy, které již neběží. Tyto procesy lze skrýt. Je-li vybrán některý proces, zobrazí se jeho síť a v ní jsou zvýrazněny proveditelné přechody. Je-li vybráno místo, vpravo se zobrazí jeho značení. Je-li vybrán přechod, vpravo se zobrazí seznam všech navázání proměnných, pro která je proveditelný, a seznam všeh rozpracovaných invokací (procesů, vytvořených vybraným přechodem). Je-li vybráno některé navázání proměnných, tlačítkem `FIRE` může byt přechod proveden. Poté debugger zobrazí nový stav.



Obrázek D.1: PNtalk Browser – nástroj pro vytváření, prohlížení a modifikaci OOPN.



Obrázek D.2: PNtalk Debugger – nástroj pro interaktivní simulaci OOPN.