Simulace a návrh vyvíjejících se systémů

Vladimír Janoušek

Fakulta informačních technologií, Vysoké učení technické v Brně

Brno, 2008

Simulace a návrh vyvíjejících se systémů

Vladimír Janoušek[1]

FIT, Brno University of Technology, Czech Rep., email: janousek@fit.vutbr.cz

Simulace a návrh vyvíjejících se systémů

Vladimír Janoušek

Brno 2008

Předmluva

Aplikací simulace a formálních modelů, zvláště pak modelů na bázi DEVS (Discrete Event Systems Specification) v návrhu systémů se ve světě dlouhodobě zabývá především skupina prof. Zeiglera (Univ. of Arizona). Přístup této skupiny v poslední době staví především na zavedených programovacích jazycích, jako je C++ a Java a na sladění zavedených metod softwarového inženýrství s využitím modelování a simulace v návrhu systémů. Tento přístup je s úspěchem používán v průmyslu, zvláště pak ve vývoji vojenských a kosmických technologií (výzkum v této oblasti podporuje zejména NASA a Ministerstvo obrany USA).

Skupina modelování a simulace na FIT VUT v Brně není ani zdaleka tak silná, ale také se snaží aplikovat formální modely a simulaci ve vývoji systémů. Vychází však z poněkud jiných kořenů, které nejsou tak těsně svázány s průmyslem a jsou vhodné spíše pro jiný typ projektů a s tím související jiný přístup k vývoji systémů. Klade důraz především na rychlé prototypování, experimentální programování, dynamické jazyky a objektovou orientaci založenou na prototypových objektech. Tento přístup je obecně vhodný pro návrh a vývoj systémů s nejasnou specifikací, kterou je třeba dopracovat až v rámci vývoje a cílového nasazení. Kromě možnosti zkoumat a interaktivně vyvíjet systém za běhu je též předmětem zkoumání možnost automatického vývoje modelů. Oba zmíněné případy vyžadují zabývat se otevřeností a reflektivitou v souvislosti s formálními modely, simulací a objektovou orientací.

Otevřenost, reflektivitu, experimentální programování a objektovou orientaci se členové skupiny snaží aplikovat přibližně od r. 2000. V té době existovala experimentální implementace interpretu Objektově orientovaných Petriho sítí (OOPN), které jsem navrhl a formálně definoval v rámci své disertační práce. Pro OOPN jsem v té době navrhl koncept otevřené architektury, umožnující reflektivní zásahy do modelu a simulátoru v době běhu. Toto téma posléze podrobně zpracoval ve své disertační práci Radek Kočí. Implementována byla tenkrát ale jen část potřebných reflektivních vlastností, které byly nezbytné pro jednoducé experimenty např. s vnořenou simulací. V rámci zkoumání možností multiparadigmatického přístupu k tvorbě modelů s jednoduchým jednotícím formalismem bylo posléze rozhodnuto paralelně aplikovat tytéž principy pro podstatně jednodušší formalismus, a sice DEVS (Discrete EVent Systems Specification). Díky konceptuální jednoduchosti a hierarchické struktuře s volně vázanými komponentami je DEVS použitelný jako společný základ pro aplikaci různých formalismů v rámci jednoho simulačního modelu. Experimentální implementace, kterou jsem vytvořil, splňovala všechny klíčové požadavky – byl to systém s plnou reflektivitou, umožňující neomezené zásahy do simulace v době běhu. Experimentálně byly též implementovány vizuální nástroje pro inspekci a editaci modelů (implementaci těchto nástrojů provedl v rámci své diplomové práce Elod Kironský). Vytvořené prostředí nyní poskytuje rámec, do kterého je možné potenciálně začlenit interprety jiných formalismů. V současné době je do tohoto prostředí vnořen a v tomto pro-

středí úspěšně používán interpret OOPN (díky Radkovi Kočímu). Jednou z větších aplikací OOPN je prostředí pro vývoj multiagentních systémů, které jako součást své disertační práce navrhl a implementoval Zdeněk Mazal.

V uvedeném kontextu se nyní jeví jako vhodné popsat klíčové aspekty současného stavu poznání v oblasti aplikace reflektivity pro potřeby modelování a simulace vyvíjejících se systémů. A o tom je tato práce.

Autor

Abstrakt

Tato práce se zabývá problematikou modelování, simulace a návrhu systémů s diskrétními událostmi s důrazem na nepřetržitě běžící a vyvíjející se systémy. Konkrétně se zabývá těmito tématy:

- Vymezení třídy vyvíjejících se systémů, zahrnující jak interaktivní, tak automatický vývoj.
- Systémy s diskrétními událostmi, formalismus DEVS, jeho varianty, včetně existujících modifikací, zavádějících strukturní dynamiku.
- Reflektivní DEVS a otevřená abstrakní architektura pro modelání, simulaci a návrh systémů.
- Prostředky pro interaktivní evoluční návrh systémů.
- Případové studie, demonstrující reflektivitu a dynamický vývoj struktury systému.

Hlavním přínosem práce je (1) zmapování problematiky vyvíjejících se systémů, zahrnující především dynamickou manipulaci s modely a simulacemi v kontextu návrhu systémů s využitím simulace, (2) koncept abstraktní architektury pro simlaci vyvíjejících se systémů a (3) a ověření navrženého konceptu abstraktní achitektury pro vyvíjející se systémy příkladem praktické realizace, aplikované v několika případových studiích. To vše v návaznosti na teorii modelování a simulace systémů s diskrétními událostmi.

Obsah

Kapitola 1

Úvod – vyvíjející se systémy

Tato práce se zabývá využitím simulace v návrhu a vývoji počítačových (i když nikoliv nutně jen počítačových) systémů, které jsou používány v reálném prostředí, mají tedy definované vstupy a výstupy a pracují v reálném čase. Zvláštní pozornost je přitom věnována vývoji struktury systémů nejen v průběhu návrhu a testování, ale i během reálného používání a údržby. K vývoji struktury systému může docházet jednak inkremetálními zásahy zvenčí, jednak automatickou adaptací systému na měnící se vnější podmínky.

Modelovací formalismy Problematika vyvíjejících se systémů bude popsána obecně, na úrovni abstraktních modelů s rigorózně definovanými pojmy čas, stav, vstup a výstup. Konkrétní formalismus pro nás není nijak zavazující, důraz bude kladen na v přímou návaznost na pojmy obecné teorie systémů. Jako společný základ pro úvahy o použití různých formalismů bude použit formalismus pro specifikaci systému s diskrétními událostmi – DEVS (Discrete EVent systems Specification). Tento formalismus je významný svou bezprostřední vazbou na obecnou teorii systémů a právě takovou úrovní abstrakce, která je ideální pro simulaci a návrh systémů, které jsou předmětem našeho zájmu (t.j. především počítačových systémů libovolného druhu). Ostatní v úvahu připadající formalismy, jako jsou např. konečné automaty, stavové diagramy (statecharts) a Petriho sítě, jsou do DEVS relativně snadno mapovatelné. I systémy jiných typů, jako jsou spojité systémy a systémy s diskrétním časem, lze také do DEVS snadno transformovat. DEVS jako společný formální základ pro různé přístupy k modelování a simulaci usnadňuje transformaci modelů, portabilitu, migraci a klonování modelů, obecný vývoj nových simulačních architektur bez ohledu na konkrétní formalismus nebo specifikační jazyk. Využití formalismu, jako je DEVS, umožní bezprostřední provázání teorie modelování a simulace s praxí – modely a simulátory postavené na formálním základě a v souladu s teorií lze analyzovat jako matematické objekty s využitím formálních metod a současně jsou k dispozici k přímému praktickému použití, t.j. jak k simulačním experimentům, tak k běhu v cílovém prostředí.

Vývoj systémů v simulovaném a v reálném prostředí Zvyšující se nároky na kvalitu a schopnosti vyvíjených systémů, které se v důsledku narůstajících požadavků stávají stále složitějšími, vede k vývoji nových metod návrhu, vývoje, implementace a údržby systémů. Vývoj systémů v simulaci (Simulation-Based Development) je metoda tvorby počítačových systémů, využívající důkladné testování a verifikaci navrhovaného systému ve všech fázích vývoje, aniž by bylo nutné se vázat na reálné okolí systému, případně na reálný čas. Mimo to je simulace ve fázi návrhu užitečná tehdy, když pro navrhované systémy

neexistuje předem jasná specifikace. Zde je nezbytné použít evoluční návrh, který spočívá v rychlém a bezprostředním vytváření, modifikaci, testování, vyhodnocování a porovnávání mnoha kandidátních řešení. Podle charakteru hodnotící funkce pak může evoluce systémů probíhat interaktivně, automaticky, nebo kombinovaně. V případě interaktivního testování simulovaného produktu jde o tzv. virtuální prototypování. Podpůrná architektura musí příslušnou variantu evolučného návrhu systémů efektivně umožnit.

Adaptace systému na měnící se okolní podmínky Má-li být vyvíjený systém v kontaktu s měnícím se prostředím a má-li i v předem neznámých podmínkách plnit definované cíle, je třeba již při návrhu systému zohlednit možnost adaptace, vývoje, resp. učení. Adaptace systému na měnící se prostředí může být prováděna buď inkrementálními vnějšími zásahy z prostředí, tedy působením jiného systému (typicky člověka-vývojáře), nebo samostatně, na základě autonomího zkoumání prostředí a učení se. Připustíme-li autonomní chování a adaptaci systému v neznámém prostředí, má smysl o systémech, které jsou předmětem našeho zkoumání, uvažovat jako o agentních systémech, resp. jako o inteligentních agentech. Pak je vhodné zabývat se možnými agentními architekturami – systém může být realizován jako jednoduchý reaktivní agent, který na základě vstupu a aktuálního stavu jednoduše definovanou funkcí generuje výstup, nebo může být chápán jako agent deliberativní (uvažující), jehož stav je již komplikovaněji strukturován a funkce, modifikující stav na základě aktuálního stavu a vstupu, případně generující výstup na základě stavu, jsou poměrně komplikované funkce, specifikované např. prostředky formální logiky, případně neuronovými sítěmi, s využitím fuzzy logiky, genetických algoritmů apod.

Motivační příklady Smyslem tohoto textu je zmapovat esenciální skutečnosti, které se týkají návrhu a udržování vyvíjejích se systémů. Přitom je kladen důraz na použití simulace, formálních modelů a jejich zachování v průběhu a vývoje i cílového nasazení. Jako motivační příklady a případové studie uvedeme několik různorodých systémů, které spojuje právě kontakt s okolní realitou a důraz na dynamickou manipulaci s modely a simulacemi:

(1) Simulující systém. V obecném pojetí jde o systém, jehož součástí je simulace. Simuluje-li systém sám sebe a používá-li výsledky vlastní simulace pro modifikaci svého vlastního chování v budoucnu, mluvíme o reflektivní simulaci. Jde o případ anticipujícícho systému, optimalizujícího své předpokládané budoucí chování podle zvolených kriterií na základě násobných simulací sebe sama. Příkladem může být optimalizace počtu otevřených přepážek v bance v závislosti na změnách pravděpodobnostního rozložení příchozích klientů v průběhu dne. Jde o analyticky neřešitelný problém, proto je nutné použít simulaci. Simulace simulujících systémů klade specifické nároky na simulační systém – je třeba pracovat s nezávislými časovými osami a zajistit přenos informací mezi jednotlivými simulacemi. To lze řešit buď s podporou operačního systému (jednotlivé simulace spouštět jako procesy OS), nebo v rámci simulačního systému samotného. První možnost je snadno dostupná, ale z programátorského hlediska těžkopádná, druhá možnost je pohodlnější, ale simulační prostředí ji musí samo o sobě umožnit.

(2) Inkrementální vývoj řídicího systému autonomího mobilního robota v simulovaném prostředí. Jde o vývoj systému, jehož specifikace není na počátku vývoje zcela jasná a je ji třeba dodatečně upřesňovat na základě výsledků testů, prováděných na pokusných realizacích v různých prostředích. Řídicí systém autonomního mobilního robota je vhodné vyvíjet a testovat v simulovaném prostředí dříve, než přistoupíme k testování v prostředí

reálném. Simulovat přitom lze i tvrdší podmínky, než jaké očekáváme v reálném prostředí, případně podmínky, které sice očekáváme, ale jsou reálně těžko vytvořitelné. V případě, že není jasný ani okamžik, kdy je cílový produkt prohlášen za hotový, musíme připustit i dodatečný vývoj při běžném provozu v cílovém prostředí. Aby bylo možné řídicí systém inkrementálně vyvíjet jak v simulovaném, tak v reálném prostředí, je nezbytné zachovat model řízení v průběhu celého vývoje, včetně cílového nasazení. Součástí cílové realizace pak musí být interpret formalismu, v kterém byl model vytvořen. Současně musí být tento interpret zpřístupněn pro monitorování stavu a pro inkrementální zásahy vývojáře – typicky vzdáleně, např. s využitím webových služeb.

(3) Řízení projektového portfolia, jeho optimalizace a monitorování. Projektové portfolio je množina aktuálně běžících a plánovaných projektů. Modely (rámcové plány) projektů, specifikované např. sít'ovými grafy nebo časovanými Petriho sítěmi, se používají k optimalizaci mechanismu přidělování zdrojů (rozvrhování) a k průběžnému monitorování průběhu projektu. Jak v rámci optimalizace, tak v průběhu monitorování se využívá simulace. V rámci optimalizace je třeba simulovat mnoho kandidatních modelů a na základě vyhodnocení výsledků vybrat model, který nejlépe vyhovuje zadaným kriteriím. V rámci monitorování se pak porovnává simulace plánovaného průběhu projektu s realitou. Postupem času se mění jak projektové portfolio (tj. množina aktuálních projektů), tak struktura dostupných zdrojů. Model celého systému se proto musí adekvátním způsobem průběžně přizpůsobovat takto se měnícím vnějším podmínkám. Obdobné (v mnohém tytéž) problémy se řeší v souvislosti se systémy plánování a řízení výroby.

Reflektivní a metaúrovňové architektury Uvedené příklady demonstrují významný aspekt celé třídy podobných systémů, kterým je nutnost zabývat se kromě základní (aplikační) úrovně také metaúrovní, tedy systémem popisujícím vývoj těchto systémů. Na této úrovni sledujeme přechody mezi dočasně existujícími dílčími systémy. Z praktického hlediska je třeba v se v uvedeném kontextu zabývat kromě vzájenného provázání reality a simulace (reality-in-the-loop simulation) také metaúrovňovou architekturou a reflektivním rozhraním smulátoru, aby bylo možné s modely a simulacemi dynamicky (to jest za běhu) manipulovat v souladu s teorií a bez ohledu na to, zda aktorem této manipulace je vnější systém nebo manipulovaný systém sám.

Souvislosti Metaúrovňové a reflektivní architektury patří k základům počítačové vědy a jsou neodmyslitelnou součástí informačních technologií. V oblasti teoretických základů stojí za zmínku především Goedelův důkaz neúplnosti formálního systému a Turingův univerzální stroj, použitý v důkazu nerozhodnutelnosti problému zastavení. V oblasti vědy o systémech se např. Klir zabývá existencí metasystémů, popisujících vývoj systémů. Z informačních technologií stojí za zmínku obecné operační systémy což jsou jsou vesměs systémy s metaúrovňovou a reflektivní architekturou – umožňují vytváření a manipulaci s programy a procesy, přičemž aktory takové manipulace jsou běžící procesy řízené existujícími programy. Co do ideové čistoty jsou zajímavé systémy, založené na dynamických jazycích, jako jsou LISP a Smalltalk, případně na jazycích, které jsou LISPem a Smalltalkem inspirovány. Zmíněné systémy jsou schopné nepřetržitě běžet a svými vlastními prostředky samy sebe vyvíjet, a to jak autonomně, tak na základě interakce s uživatelem. Vývoj systému za běhu na základě interakce s vývojářem je klíčovým předpokladem pro

techniku vývoje, zvanou experimentální programování (exploratory programming).[1] V oblasti umělé inteligence je sebemodifikace základním předpokladem schopnosti systému učit se.

Hlavním cílem tohoto textu je popsat fenomén vyvíjejích se systémů v kontextu simulačního modelování s využitím formálních modelů. Motivací k tomuto počínání je snaha aplikovat formální modely při vytváření i v rámci údržby a adaptace počítačových systémů na měnící se podmínky během jejich života. Takové zprůhlednění vyvíjených systémů i metod jejich vývoje by mělo být účinným prostředkem ke zvýšení jejich užitné hodnoty.

Obsah Následující text je organizován takto: Po obecném úvodu do modelování a simulace systémů s důrazem na systémy s diskrétními událostmi je popsána problematika systémů s dynamicky se měnící strukturou. Poté je definována abstraktní architektura pro simulaci vyvíjejících se systémů. Následuje diskuse praktických aspektů definované architektury, spolu s metodickými poznámkami. Nakonec jsou uvedeny jednoduché případové studie, demonstrující aplikaci simulace v návrhu a realizaci vyvíjejích se systémů.

[1] Český překlad není přesný, ale v daném kontextu je použitelný.

Kapitola 2

Systémy s diskrétními událostmi

Kapitola prezentuje systémy s diskrétními událostmi, které jsou vhodnou abstrakcí pro modelování počítačových (ale nejen počítačových) systémů v rámci jejich návrhu a vývoje. Adekvátnost tohoto způsobu modelování je opodstatněna tím, že jako systémy s diskrétními událostmi lze modelovat všechny typy dynamických systémů (t.j. systémů, u nichž má smysl zkoumat chování) a tedy všechny způsoby aplikací vyvíjených systémů.

V návaznosti na základní pojmy teorie systémů a teorie modelování a simulace je v následujících podkapitolách představen formalismus DEVS a jeho abstraktní simulátor. Pro ilustraci přímé vazby teorie a praxe modelování a simulace je prezentována i ukázka praktického použití formalismu DEVS v konkrétním programovacím jazyce.

2.1 Systémy, znalosti, problémy

Klir [Kli85] definuje rámec pro studium systémů, ve kterém rozlišuje 4 epistemologické úrovně, reflektující dostupné znalosti o systému. Každá úroveň zahrnuje znalosti dostupné v úrovních nižších.

0. *Zdrojová úroveň* (source level, source system) je nejnižší úroveň, kde jde jen o množinu proměnných, které nás zajímají.
1. *Úroveň dat* (data level, data system) zahrnuje temporální vývoj proměnných v podobě časových řad.
2. *Úroveň chování* (behavior level, behavior system), obsahuje znalosti o vztazích mezi historiemi proměnných. Behaviorální systémy jsou schopny generovat časové řady (time series), proto se také nazývají generativní systémy.
3. *Úroveň struktury* (structure level, staructural system) zahrnují znalost o subsystémech systému a o struktuře jejích vzájemních vztahů (propojení).

Tuto hierarchii završuje pátá úroveň - *úroveň metasystémů*, které obsahují informaci o tom, jak se datové, generativní a strukturní systémy vyvíjejí v čase. Na této úrovni nás zajímá dynamika samotného popisu systému.

Ve výše uvedeném kontextu existují 3 typy problémů k řešení [Kli85]:

1. *Analýza systému.* Systém buď existuje, nebo je plánována jeho existence a pokoušíme se pochopit jeho chování. Za použití generativního popisu získáváme a zkoumáme data.

2. *Inference systému.* Systém existuje a snažíme se přejít na vyšší epistemologickou úroveň, typicky od dat ke generativnímu popisu.

3. *Návrh systému.* Systém neexistuje a snažíme se ho navrhnout. Přecházíme na vyšší úroveň znalostí, od existujících generativních komponent ke složenému systému s vhodnou strukturou.

Zatímco Klir [Kli85] uvedenou hierarchii znalostí definuje především jako rámec pro zkoumání možnosti induktivní identifikace systémů (odvozování vyšších znalostních úrovní z nižších), Zeigler [Zei84, Zei90, ZKP00] a jiní se zaměřují ve větší míře na metody specifikace systémů, které se dají uplatnit v modelovacích a simulačních jazycích. Zabývají se především úrovní struktury a chování s cílem generovat data. Diskutovanou otázkou i v této oblasti je úroveň metasystémů, tedy možnost specifikovat a simulovat vyvíjející se generativní systémy.

Účinným prostředkem pro analýzu chování systémů je simulace. Modelování a simulaci systémů, zvláště pak systémů s diskrétními událostmi, se věnují následující části kapitoly.

2.2 Simulační modelování

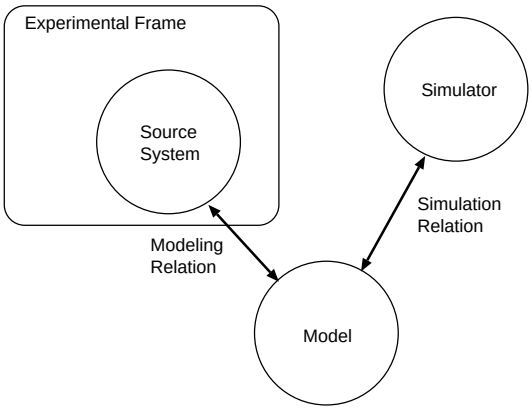
Základní rámec pro modelování a simulaci je podle [ZKP00] definován čtyřmi entitami (viz obr. 2.1):

1. *Zdrojový systém (source system)* představuje zdroj dat (chování).

2. *Model* představuje instrukce pro generování dat srovnatelných s reálným systémem.

3. *Simulátor* provádí instrukce modelu a skutečně generuje chování.

4. *Experimentální rámec (experimental frame)* specifikuje podmínky, za kterých systém pozorujeme a exeperimentujeme se s ním.



Obrázek 2.1: Základní entity a jejich vztahy

Základní vztahy mezi těmito entitami jsou *modelování* a *simulace*. Relace modelování přitom specifikuje, jak model reprezentuje odpovídající reálný systém. Mezi systémem a modelem musí být homomorfní vztah, tj. je přípustné zjednodušení. Relace simulace specifikuje, jak přesně simulátor realizuje instrukce modelu.

Přechod od zdrojového systému k modelu může být proveden na základě pozorování dat získaných z experimentů se systémem. Model je pak použit k vytvoření simulátoru, který slouží ke generování dat. Chování, generované simulátorem, musí být validní, tj. simulátor musí generovat stejná nebo podobná data jako zdrojový systém za stejných podmínek.

2.3 Čas, trajektorie a segmenty

Základním pojmem dynamického systému je čas, který je nezávislou veličinou. Čas je definovován strukturou

$$time = (T, <),$$

kde

T je množina,
$<$ je tranzitivní, irreflexivní a antisymetrická relace (uspořádání) nad T.

Jestliže pro každou dvojici (t, t') platí buď $t < t'$, nebo $t > t'$, nebo $t = t'$, pak jde o *lineární (úplné) uspořádání*. Relaci $<$ definujeme typicky jako lineární uspořádání, ale v některých případech je vhodné pracovat i s částečným uspořádáním, které může modelovat neurčitost v popisu systému a násobnost stavových trajektorií. *Časovou základnu* T typicky definujeme tak, že $T = R_0^+$ (pak jde o *spojitou časovou základnu*), nebo $T = N$ (pak jde o *diskrétní časovou základnu*). Nad T definujeme intervaly: $(t_1, t_2) = \{\tau | t_1 < \tau < t_2, \tau \in T\}$, $[t_1, t_2] = \{\tau | t_1 \leq \tau \leq t_2, \tau \in T\}$, $(t_1, t_2] = \{\tau | t_1 < \tau \leq t_2, \tau \in T\}$. Zápis $< t_1, t_2 >$ označuje libovolný z intervalů. Intervaly nad T někdy značíme $T_{<t_1,t_2>}$.
Je-li T časová základna a A libovolná množina, funkce

$$f : T \longrightarrow A$$

se nazývá *časová funkce* nebo *signál*. Restrikce f na intervalu $< t_1, t_2 >$ se nazývá *segment* nebo *trajektorie*

$$\omega :< t_1, t_2 > \longrightarrow A$$

a zapisuje se $\omega_{<t_1,t_2>}$. Dva segmenty nazveme *sousedící*, když jejich domény na sebe navazují. *Spojitý segment* nad spojitou časovou základnou je spojitý ve všech bodech. *Po částech spojitý* segment je spojitý ve všech bodech kromě konečného počtu bodů. *Po částech konstantní segment* je speciální případ po částech spojitého segmentu. *Událostní segment* $\omega_{<t_0,t_n>} :< t_0, t_n > \longrightarrow A \cup \{\emptyset\}$ je segment nad spojitou časovou základnou a množinou událostí $A \cup \{\emptyset\}$, kde $\emptyset$ značí ne-událost. $\emptyset$ je hodnotou $\omega_{<t_0,t_n>}$ mezi jednotlivými událostmi z množiny A, které se vyskytly v časech $t_1, t_2, ..., t_{n-1}$. Segment nad diskrétní časovou základnou se nazývá *sekvence*.

2.4 Systém

Systém je abstraktní koncept, popisující chování entit v čase.[1] Popisuje výstupní chování na základě vstupu a stavové informace. Formálně je systém popsán strukturou [ZKP00]

$$S = (T, X, \Omega, Q, q_0, Y, \delta, \lambda),$$

[1] Uvažujeme dynamický systém na úrovni 2 Klirovy hierarchie.

kde

T je časová základna,
X je množina vstupních hodnot,
Y je množina výstupních hodnot,
Ω je množina vstupních segmentů,
Q je množina stavů,
q_0 je počáteční stav,
$\delta : Q \times \Omega \longrightarrow Q$ je přechodová funkce,
$\lambda : Q \times X \longrightarrow Y$ je výstupní funkce.

Systém přitom splňuje následující omezení:

(1) Ω je uzavřená vzhledem ke kompozici;
(2) pro každý pár sousedících segmentů $\omega, \omega' \in \Omega$ a pro všechny stavy $q \in Q$ platí: $\delta(q, \omega\omega') = \delta(\delta(q, \omega), \omega)$.

Omezení (2) zaručuje, že systém může být přerušen v libovolném čase a jeho stav je registrován. Pokračování z tohoto stavu s pokračováním vstupního segmentu povede do stejného koncového stavu, jako kdyby k přerušení nedošlo.

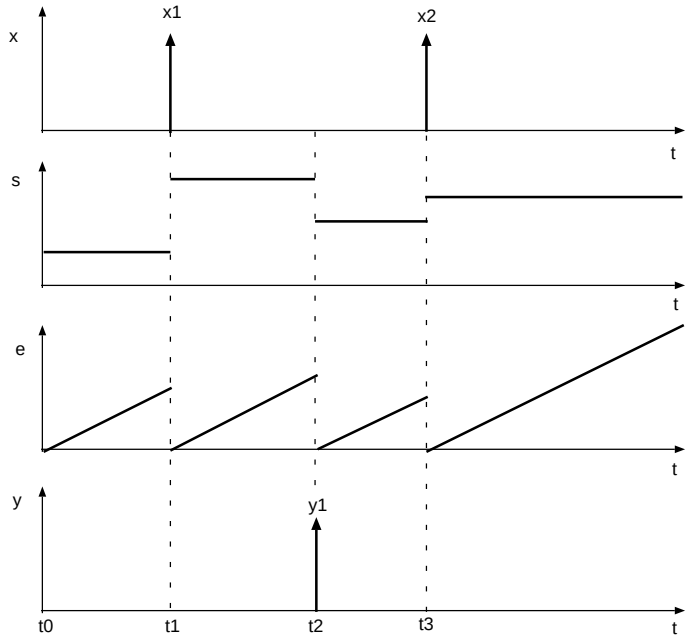
Typy systémů Zeigler [ZKP00] definuje 3 základní modelovací formalismy, resp. typy systémů: *Systém popsaný diferenciálními rovnicemi*, *systém s diskrétním časem* a *systém s diskrétními událostmi*. Posledně jmenovaný formalismus (DEVS, který bude popsán v následujícím textu) může být modifikován nebo přímo použit pro modelování ostatních typů systémů. Systém popsaný diferenciálními rovnicemi může být s využitím vhodného vzorkování a metod numerické integrace převeden na systém s diskrétním časem. Systém s diskrétním časem může být chápán jako zvláštní případ systému s diskrétními událostmi. DEVS je tedy univerzální, a proto tento formalismus považujeme za ideální nízkoúrovňovou abstrakci, vhodnou pro všechny typy systémů. Jiné (speciálnější, případně vysokoúrovňovější) formalismy je možné do DEVS relativně snadno transformovat.

2.5 Formalismus DEVS

DEVS je zkratka pro *Discrete EVent System Specification*, tedy specifikaci systému s diskrétními událostmi [ZKP00]. Pro tento formalismus je definován *abstraktním simulátor*, který lze kromě specifikace sémantiky chápat také jako vzor pro reálnou implementaci.

2.5.1 Systém s diskrétními událostmi

Neformálně lze systém s diskrétními událostmi popsat následujícím způsobem. Systém má vstupy a výstupy pozorovatelné jako události. Na některé vstupy systém reaguje výstupem (okamžitě nebo se zpožděním), na některé viditelně nereaguje (jen přechází mezi vnitřními stavy). Někdy generuje výstup bez přímé vnější příčiny. Uvnitř systém přechází mezi vnitřními stavy, a to buď samovolně (poté, co v daném stavu strávil jistý čas), nebo na základě vnější události (události na vstupu). Výstup vždy závisí jen na aktuálním stavu a je pozorovatlený jako událost při samovolném přechodu systému do stavu následujícího. Na obr. 2.2 je zobrazen vstupní událostní segment (x), stavová trajektorie (s), trajektorie uplynulého času (e) pro aktuální stav a výstupní událostní segment (y).



Obrázek 2.2: Časové segmenty systému s diskrétními událostmi

Souvislost s obecnou definicí systému Časová základna systému s diskrétními událostmi je

$$T = R_0^+.$$

Vstupní a výstupní segmenty jsou událostní segmenty. Je-li S je množina stavů, stavová trajektorie je po částech konstantní segment nad S a T. Podle obecné definice systému musí stav registrovat i dobu setrvávání ve stavu $s \in S$, proto definujeme Q jako množinu úplných stavů

$$Q = \{(s, e) | e \in S, e \in R_0^+\}.$$

Prvky Q jsou dvojice, tvořené stavem a časem setrvávání systému ve stavu s. Totální stavová trajektorie je po částech spojitý segment nad Q a T.

2.5.2 Základní model

Základní model je specifikován algebraickou strukturou

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

kde

X je *množina vstupních událostí*,
S je *množina stavů*,
Y je *množina výstupních událostí*,
$\delta_{int} : S \longrightarrow S$ je *interní přechodová funkce*,
$\delta_{ext} : Q \times X \longrightarrow S$ je *externí přechodová funkce*, kde
$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ je *množina úplných stavů*,
e je *čas uplynulý od poslední události*,
$\lambda : S \longrightarrow Y$ je *výstupní funkce*,

$ta : S \longrightarrow R_0^+ \cup \{\infty\}$ je *funkce posuvu času.*

Interpretace formalismu Systém je v daném okamžiku ve stavu $s \in S$. Nevyskytne-li se žádná externí událost, systém setrvá ve stavu s p o dobu $ta(s)$ time. Zvláštní hodnotou $ta(s)$ je symbol ∞ – v takovém případě je systém pasivní a plánuje setrvat ve stavu s navždy. Dosáhne-li uplynulý čas e hodnoty $ta(s)$, výstup $\lambda(s)$ se objeví na výstupu a systém změní stav na $\delta_{int}(s)$. Vyskytne-li se externí událost $x \in X$ v čase $e \leq ta(s)$, systém změní stav na $\delta_{ext}(s, e, x)$. Výstup se generuje pouze v okamžiku, kdy $e = ta(s)$. Při konfliktu interního a externího přechodu (mají-li se provést ve stejném časovém okamžiku) se provede pouze externí přechod.

Atomický model Uvedený formalismus umožňuje specifikovat jakýkoli systém s diskrétními událostmi. Chápeme-li ale systém jako hierarchickou strukturu, výše uvedená definice odpovídá specifikaci atomickému modelu, základní nedělitelné jednotce systému. Příklady mohou být generátor, procesor, fronta, binární čítač apod. Systémy složené ze subsystémů pak specifikujeme jako složené modely.

2.5.3 Složený model

Dalším konceptem formalismu DEVS je hierarchie – systém může být dekomponován do subsystémů. Subsystémy jsou propojeny a mohou na sebe vzájemně působit prostřednictvím svých externích (vstupních a výstupních) událostí. Stav složeného systému je pak určen stavy všech jeho subsystémů. Složením a propojením systémů vznikne opět systém.

Složený model je definován strukturou

$$N_{self} = (X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, select)$$

kde

X je *množina vstupních událostí*,
Y je *množina výstupních událostí*,
D je *množina jmen submodelů*,
$\{M_d \mid d \in D\}$ je *množina submodelů*,
$\{I_d \mid d \in D \cup \{self\}\}$ je *specifikace propojení*,
$\quad \forall d \in D \cup self : I_d \subseteq D \cup \{self\}, d \notin I_d$,
$\{Z_{i,d} \mid i \in I_d, d \in D \cup \{self\}\}$ je *specifikace překladu událostí*,
$\quad Z_{i,d} : X \longrightarrow X_d$ pro $i = self$,
$\quad Z_{i,d} : Y_i \longrightarrow Y$ pro $d = self$,
$\quad Z_{i,d} : Y_i \longrightarrow X_d$ pro $i \neq self$ a $d \neq self$,
$select : 2^D - \{\} \longrightarrow D$ je *preferenční funkce.*

Interpretace formalismu I_d pro každý model d je možina jmen modelů, jejichž výstupy jsou připojeny na vstup modelu d. $Z_{i,d}$ pro každý model i, který je připojen na vstup modelu d, specifikuje překlad výstupní události modelu i na vstupní událost modelu d. $self$ identifikuje složený model N. Funkce *select* se používá k řešení konfliktu současných událostí – vybírá jeden submodel z množiny submodelů, ve kterých je aktuálně proveditelný interní přechod.

Uzavřenost vzhledem ke skládání Množina systémů je uzavřená vzhledem k operaci skládání. Tomu odpovídá skutečnost, že i složený model je model se stavy, vstupy a výstupy a může být popsán jako základní DEVS. Tato vlastnost byla formálně dokázána a důkaz lze najít např. v [ZKP00].

2.5.4 Použití portů

Stavové proměnné a porty Množiny interních stavů a množiny vstupních a výstupních události je možné specifikovat jako *strukturované množiny*, což nám umožní použít libovolný počet vstupních a výstupních *portů* a libovolný počet *stavových proměnných*. Strukturovaná množina

$$S = (V, S_1 \times S_2 \times, ... \times S_n)$$

je definována množinou proměnných V, $|V| = n$, a kartézským součinem množin hodnot jedotlivých proměnných. Takže v definici DEVS s n stavovými proměnnými $\{v_1, v_2, ...v_n\}$, p vstupními porty $\{ip_1, ip_2, ..., ip_p\}$ a q výstupními porty $\{op_1, op_2, ..., op_p\}$ můžeme množiny X, S a Y zapsat takto:

$$\begin{aligned} X &= (\{ip_1, ..., ip_p\}, \{(x_{ip_1}, ..., x_{ip_p}) | x_{ip_1} \in X_{ip_1}, ..., x_{ip_1}, \in X_{ip_p}\}), \\ S &= (\{v_1, ...v_n\}, \{(s_{v_1}, ..., s_{v_n}) | s_{v_1} \in S_{v_1}, ..., s_{v_n}, \in S_{v_n}\}), \\ Y &= (\{op_1, ..., op_p\}, \{(y_{op_1}, ..., y_{op_q}) | y_{op_1} \in Y_{op_1}, ..., y_{op_q}, \in S_{op_q}\}). \end{aligned}$$

Strukturovaná množina může být specifikována i jinak, jako množina dvojic $(jmeno, hodnota)$, v našem případě:

$$\begin{aligned} X &= \{(p, v) | p \in InPorts, v \in X_p\}, \\ S &= \{(v, s) | v \in StVariables, s \in S_v\}, \\ Y &= \{(p, v) | p \in OutPorts, v \in Y_p\}, \end{aligned}$$

přičemž

$$\begin{aligned} InPorts &= \{p | (p, v) \in X\}, \\ StVariables &= \{v | (v, s) \in S\}, \\ OutPorts &= \{p | (p, v) \in Y\}. \end{aligned}$$

Alternativní definice složeného modelu Jsou-li X a Y strukturované množiny, specifikaci propojení lze definovat mezi jednotlivými porty jednotlivých modelů a funkci překladu událostí definovat jako identitu. Toto zjednodušení specifikace složeného modelu je snadno prakticky použitelné a je skutečně použito ve všech reálných implementacích formalismu DEVS. Následuje formální definice toho typu specifikace složeného modelu.

$$N_{self} = (X_{self}, Y_{self}, D, \{M_d\}, EIC, IC, EOC, select),$$

kde

$X_{self} = \{(p, v) | p \in InPorts_{self}, v \in X_{p,self}\}$ je *množina vstupních událostí*,
$Y_{self} = \{(p, v) | p \in OutPorts_{self}, v \in Y_{p,self}\}$ je *množina výstupních událostí*,
D je *množina jmen submodelů*,
$\{M_d \, | d \in D\}$ je *množina submodelů* se vstupy X_d a výstupy Y_d,

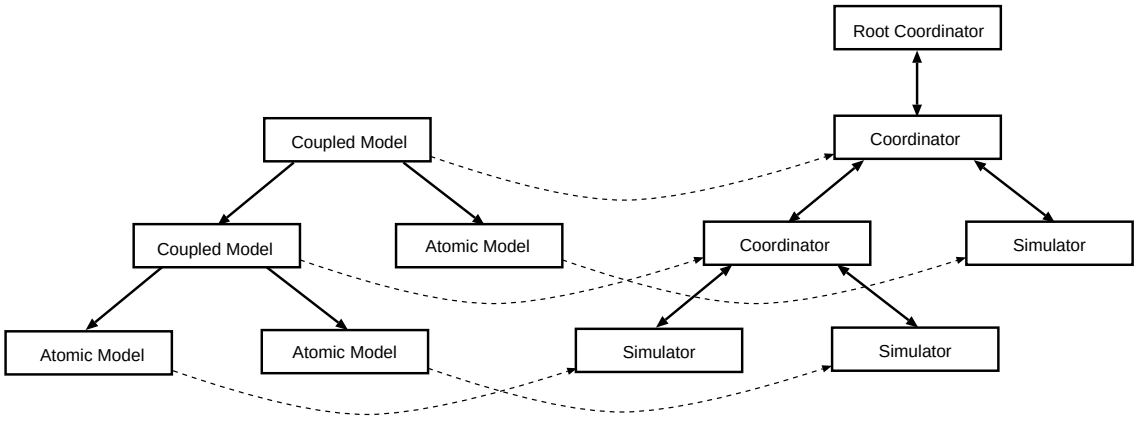
$X_d = \{(p, v) | p \in InPorts_d, v \in X_{p,d}\}$
$Y_d = \{(p, v) | p \in OutPorts_d, v \in Y_{p,d}\}$,
$EIC = \{((self, ip_{self}), (d, ip_d)) | ip_{self} \in InPorts_{self}, d \in D, ip_d \in InPorts_d\}$
je *množina externích vstupních propojení*,
$IC = \{((a, op_a), (b, ip_b)) | a, b \in D, op_a \in OutPorts_a, ip_b \in InPorts_b\}$
je *množina interních propojení*,
$EOC = \{((d, op_d), (self, op_{self})) | op_{self} \in OutPorts_{self}, d \in D, op_d \in OutPorts_d\}$
je *množina externích výstupních propojení*,
$select : 2^D - \{\} \longrightarrow D$ je *preferenční funkce*
a jsou splněna následující omezení:
$\forall((a, p), (b, q)) \in IC \Longrightarrow a \neq b$,
$\forall((a, p), (b, q)) \in EIC : X_{p,a} \subseteq X_{q,b}$,
$\forall((a, p), (b, q)) \in IC : Y_{p,a} \subseteq X_{q,b}$,
$\forall((a, p), (b, q)) \in EOC : Y_{p,a} \subseteq Y_{q,b}$.

2.5.5 Varianty a rozšíření základní definice DEVS

Existují různé varianty a modifikace DEVS, jako je paralelní DEVS, celulární DEVS, symbolický DEVS, fuzzy DEVS, nebo DEVS s dynamickou strukturou. Existuje také možnost mapování jiných formalismů do DEVS. Takto lze mapovat např. konečné automaty, Petriho sítě, stavové diagramy, ale i diferenciální rovnice (prostřednictvím numerické integrace). DEVS lze tedy chápat jako společný základ pro multiparadigmatické modelování a simulaci [ZKP00].

2.6 Hierarchická simulace DEVS (abstraktní simulátor)

Simulace DEVS je organizována hierarchicky (viz obr. 2.3). Hierarchie simulátorů odpovídá hierarchii modelů. Každý simulátor je zodpovědný za simulaci svého modelu, je podřízen nadřazenému simulátoru a může mít podřízené simulátory. Ke komunikaci simulátorů při

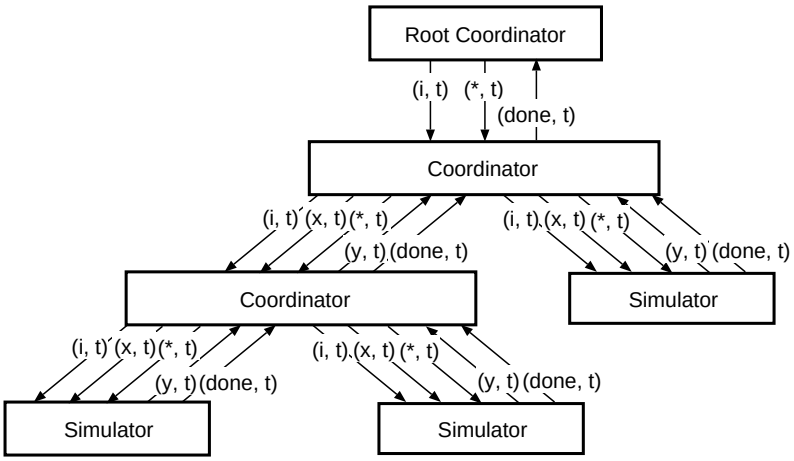


Obrázek 2.3: Mapování mezi modely a simulátory

simulaci slouží zprávy (viz obr. 2.4):

- (i, t) - inicializační zprávy posílá nadřazený simulátor podřízeným při odstartování simulace,

- $(*, t)$ - pokyn nadřazeného simulátoru podřízenému k vygenerování výstupu a provedení interního přechodu plánovaného na čas t,
- (y, t) - výstupní zprávy posílá podřízený simulátor nadřazenému (nadřazený simluátor je zopdpovědný za jejich případnou distribuci),
- (x, t) - vstupní zprávy posílá nadřazený simulátor podřízeným a vynucuje tak případnou distribuci a provedení externího přechodu,
- $(done, t_{next})$ - podřízený signalizuje nadřazenému dokončení zpracování zprávy (i, t), $(*, t)$, nebo (x, t) a oznamuje čas provedení následujícího interního přechodu.



Obrázek 2.4: Zprávy používané při simulaci DEVS

Každý simulátor udržuje informaci o čase poslední události t_{last} a čase následující plánované události t_{next}, je schopen provést plánovanou událost a reagovat na vstupní událost. Simulátor složeného modelu (Coordinator) udržuje t_{next} pro všechny submodely, deleguje na ně provádění událostí a distribuuje události mezi submodely podle jejich propojení. Na nejvyšší úrovni simulaci řídí kořenový (hlavní) simulátor (Root Coordinator), který iniciuje provádění jednotlivých kroků simulace. Dále uvedené algoritmy vycházejí z [ZKP00] a [Van00].

Abstraktní kořenový simulátor Kořenový simulátor je zodpovědný za postupné provádění kroků simulace. Je spuštěn s parametry podřízený simulátor *child*, čas začátku a konce simulace t_{BEGIN}, t_{END}. Používá proměnné t a t_{next}. Algoritmus:

$t := t_{BEGIN}$
pošli (i, t) podřízenému simulátoru *child*
počkej na $(done, t_{next})$ od podřízeného simulátoru *child*
$t := t_{next}$
dokud $t < t_{END}$ opakuj:
    pošli $(*, t)$ podřízenému simulátoru *child*
    počkej na $(done, t_{next})$ od podřízeného simulátoru *child*
    $t := t_{next}$

Abstraktní simulátor atomického modelu

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

používá proměnné t_{last}, t_{next}, y, úplný stav (s, e) a *parent*. Algoritmus:

Na zprávu (i, t) od nadřazeného simulátoru *parent* reaguj takto:
$t_{last} := t - e$
$t_{next} := t_{last} + ta(s)$
pošli $(done, t_{next})$ nadřazenému simulátoru *parent*

Na zprávu $(*, t)$ od nadřazeného simulátoru *parent* reaguj takto:
$y := \lambda(s)$
pošli (y, t) nadřazenému simulátoru *parent*
$s := \delta_{int}(s)$
$t_{last} := t$
$t_{next} := t_{last} + ta(s)$
pošli $(done, t_{next})$ nadřazenému simulátoru *parent*

Na zprávu (x, t) od nadřazeného simulátoru *parent* reaguj takto:
$e := t - t_{last}$
$s := \delta_{ext}(s, e, x)$
$t_{last} := t$
$t_{next} := t_{last} + ta(s)$
pošli $(done, t_{next})$ nadřazenému simulátoru *parent*

Abstraktní simulátor (koordinátor) složeného modelu

$$N_{self} = (X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, select)$$

používá proměnné t_{last}, t_{next}, y, $eventList$, $imminent$, d, $receivers$, $activeChildren$ a *parent*. Algoritmus:

Na zprávu (i, t) od nadřazeného simulátoru *parent* reaguj takto:
$\forall d \in D$:
    pošli (i, t) simulátoru d
$activeChildren := D$

Na zprávu $(done, t_{next})$ od podřízeného simulátoru d reaguj takto:
$eventList := (eventList - \{(d, \_)\}) \cup \{(d, t^d_{next})\}$
$activeChildren := activeChildren - \{d\}$
Je-li $activeChildren = \emptyset$, pak:
    $t_{last} := t$
    $t_{next} := min\{t^d_{next} | d \in D\}$
    pošli $(done, t_{next})$ nadřazenému simulátoru

Na zprávu $(*, t)$ od nadřazeného simulátoru *parent* reaguj takto:
$imminent := \{d | (d, t^d_{next}) \in eventList, t^d_{next} = t\}$
$d := select(imminent)$
pošli $(*, t)$ simulátoru d
$activeChildren := \{d\}$

Na zprávu (y, t) od podřízeného simulátoru d reaguj takto:
Je-li $d \in I_{self}$ a $Z_{d,self}(y) \neq \emptyset$, pak:
$y_{self} = Z_{d,self}(y)$
pošli (y_{self}, t) nadřazenému simulátoru *parent*
$receivers := \{d | d \in D, self \in I_d, Z_{self,d}(x) \neq \emptyset\}$
$\forall d \in receivers$:
$x_d := Z_{self,d}(x)$
pošli (x_d, t) simulátoru d
$activeChildren := activeChildren \cup receivers$

Na zprávu (x, t) od nadřazeného simulátoru reaguj takto:
$receivers := \{d | d \in D, self \in I_d, Z_{self,d}(x) \neq \emptyset\}$
$\forall d \in receivers$:
$x_d := Z_{self,d}(x)$
pošli (x_d, t) simulátoru d
$activeChildren := receivers$

Korektní implementace V korektní implementaci při přijetí zprávy $(*, t)$ vždy platí $t = t_{next}$ a při přijetí zprávy (x, t) vždy platí $t_{last} \leqq t \leqq t_{next}$. Důkaz korektní synchronizace abstraktního simulátoru je podán v [ZKP00].

Přerušování a klonování simulací Uvedený abstraktní simulátor připouští možnost přerušení a následného pokračování simulace (včetně možnosti klonování a migrace) za předpokladu, ze mezitím nedojde k manipulaci se strukturou. Pro potřeby dynamické editace modelu je třeba simulátor upravit. Tato problematika bude podrobně popsána v kapitole 4.

RT simulace a HIL Uvedený abstraktní simulátor není připraven na možnost propojení s reálným okolím. Tato problematika bude diskutována v kapitole 4.

Paralelní a distribuovaná simulace Simulaci lze paralelizovat a distribuovat. V tomto případě je vhodné použít paralelní variantu DEVSu, která se nepatrně liší od klasické varianty. Tato varianta počítá s možností současných událostí na vstupech modelů a je popsána v [ZKP00]. Pro další výklad není podstatné, o kterou variantu jde. Proto bude pro jednoduchost použita varianta klasická.

2.7 Praktické použití formalismu DEVS

DEVS byl v době svého vzniku určen k tomu, aby bylo možné formálně popsat a pracovat modely, které se pak obvykle implementovaly běžným způsobem, typicky v událostně orientovaném simulačním jazyce Simscript. Lepší možností je ale použít DEVS přímo pro implementaci a odstranit tak nutnost explicitního mapování formalismu do programovacího jazyka. K tomuto účelu byly vytvořeny podpůrné knihovny pro běžné programovací jazyky, jako je LISP, C, C++, Java, Smalltalk apod.

Existuje řada implementací DEVS, jako příklady lze uvést ADEVS (University of Arizona), CD++ (Carleton University), DEVS/HLA (ACIMS), DEVSJAVA (ACIMS), GALATEA (USB Venezuela), GDEVS (Aix-Marseille III, France), JDEVS (Université de

Corse - France), PyDEVS (McGill), PowerDEVS (University of Rosario, Argentina), SimBeans (University of Linz, Austria), VLE (Université du Litoral -France), SmallDEVS (Brno University of Technology, Czech Republic), James (University of Rostock, Germany).[2] V současné době v průmyslu nejpoužívanější implementací je DEVSJAVA, kterou používá např. NASA a DoD (USA).

2.7.1 DEVS a objektová orientace

Nejčastější implementace v současné době existují pro objektově orientované jazyky C++ a Java. Implementace DEVS v objektově orientovaném jazyce založeném na třídách vede k typickému způsobu modelování, kde modely jsou vytvářeny jako podtřídy existujících tříd (modelů). Podtřída definuje instanční proměnné pro reprezentaci stavu a redefinuje metody, odpovídající funkcím δ_{ext}, λ, δ_{int}, ta atomického modelu, případně specifikuje kolekci submodelů (tj. instancí příslušných tříd), spolu s relací propojení pro složené modely. V obou případech je inicializační metoda zodpovědná za vytvoření vstupních a výstupních portů. Využití dědičnosti pro specifikaci modelů je významným přínosem, stejně tak i instanciace modelů. Díky tomu lze udržovat knihovny znovupoužitelných modelů.

2.7.2 Příklady modelů a jejich implementace

Pro ilustraci praktického použití formalismu DEVS pro tvorbu simulačních modelů bude uveden jednoduchý příklad, demonstrující přímé mapování matematického modelu do programovacího jazyka. Pro implementaci bude použit Smalltalk [GR89] s modelovacím prostředím SmallDEVS [Jan06]. V jiných objektově orientovaných jazycích a prostředích, jako je JAVADEVS apod., je princip modelování totožný, liší se jen v použitém programovacím jazyku a v názvech některých metod.[3] Pro potřeby výkladu v tomto textu je Smalltalk vhodnější jednak kvůli stručnosti a přehlednosti, jednak proto, že pro výklad v kapitole 5 bude použití Smalltalku kvůli jeho specifikým vlastnostem prakticky nezbytné.

Příklad Specifikujme hráče pingpongu:[4]

$$PingPongPlayer = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta, s_0),$$

kde

$X = \{(receive, ball)\}$,
$Y = \{(send, ball)\}$,
$S = \{send, receive\}$,
$\delta_{int}(send) = receive$, $\delta_{int}(receive) = receive$,
$\delta_{ext}(receive, e, (receive, ball)) = send$, $\delta_{ext}(send, e, x) = send$,
$\lambda(send) = (send, ball)$, $\lambda(receive) = \emptyset$,
$ta(send) = 0.5$, $ta(receive) = \infty$.

[2] Seznam je převzat z prezentace B.P. Zeiglera na konferenci Mosim08: Modélisation, Optimisation et Simulation des Systmes, March 31-April 2 2008, Paris, France.

[3] SmallDEVS umoňuje pro modelování použít kromě kasického přístupu k objektové orientaci, založeného na třídách, také jiný, flexibilnější přístup, který bude popsán v kapitole 5. Na tomto místě ale zůstaneme u klasického přístupu, který je zcela konformní s běžně používanými nástroji pro modelování a simulaci na bázi formalismu DEVS. Příklady zde uvedené lze tedy jednoduše adaptovat pro libovolný z alternativních nástrojů.

[4] Specifikaci systému jsme rozšířili o počáteční stav $s_0 \in S$.

$s_0 = send.$

V praktických implementacích formalismu DEVS atomický model definuje množinu vstupních a výstupních portů, množinu stavových proměnných, funkci posuvu času, přechodové funkce (interní a externí), výstupní funkci a počáteční stav. Následuje implementace modelu *PingPongPlayer* v prostředí SmallDEVS.

```
AtomicDEVS subclass: #PingPongPlayer
  instanceVariableNames: 'phase'

initialize
  super initialize.
  phase ← #send.
  self addInputPortNamed: #receive.
  self addOutputPortNamed: #send.

extTransition
  (phase = #receive) & ((self peekFrom: #receive) = #ball) ifTrue: [ phase ← #send ]

intTransition
  phase = #send ifTrue: [ phase ← #receive ].

timeAdvance
  phase = #send ifTrue: [ ↑ 0.5 ].
  phase = #receive ifTrue: [ ↑ Float infinity ].

outputFnc
  phase = #send ifTrue: [ self poke: #ball to: #send ].
```

Druhý hráč by měl být v počátečním stavu pasivní, od prvního se liší jen počátečním stavem:

$$InitiallyPassivePingPongPlayer = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta, s_0),$$

kde

$X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta$ jsou definovány stejně jako v modelu *PingPongPlayer*,
$s_0 = receive.$

V objektově orientované implementaci formalismu DEVS můžeme využít dědičnosti – v modelu *InitiallyPassivePingPongPlayer*, dědícího difinici modelu *PingPongPlayer*, nově definujeme pouze inicializační metodu, která nstavuje počáteční stav.

```
PingPongPlayer subclass: #InitiallyPassivePingPongPlayer

initialize
  super initialize.
  phase ← #receive.
```

Systém dvou hráčů je specifikován jako složený DEVS

$$PingPong = (X, Y, D, \{M_d\}, EIC, IC, EOC, select),$$

kde

$X = Y = \{\}$,
$D = \{PlayerA, PlayerB\}$,
$\{M_d\} = \{PingPongPlayer_{PlayerA}, InitiallyPassivePingPongPlayer_{PlayerB}\}$,
$EIC = EOC = \{\}$,
$IC = \{(PlayerA.send, PlayerB.receive), (PlayerB.send, PlayerA.receive)\}$,
$\forall m \in 2^D - \{\} : \ select(m) = PlayerA$.

Specifikace složeného modelu v praktických iplementacích DEVS obsahuje množinu submodelů, množinu vlastních vstupních a výstupních portů, specifikaci propojení mezi porty jednotlivých modelů. Následuje implementace modelu hry Ping-Pong v prostředí SmallDEVS.

```
CoupledDEVS subclass: #PingPong

initialize
  super initialize.
  self addComponents: {
    #'Player A' -> PingPongPlayer new.
    #'Player B' -> InitiallyPassivePingPongPlayer new.
  }.
  self addCouplings: {
    {#'Player A'. #send} -> {#'Player B'. #receive}.
    {#'Player B'. #send} -> {#'Player A'. #receive}.
  }.
```

Hierarchie simulátorů zastřešená kořenovým simulátorem je vytvořena následujícím kódem – ten současně odstartuje simulaci.

```
PingPong new getSimulator simulate: 1.5.
```

Průběh simulace je zaznamenán jako sekvence událostí a změn stavu. Na obrázku 2.5 je orientační textový záznam, generovaný během simulace, existuje ale možnost generovat záznam v XML pro případné další zpracování.

Uvedený příklad demonstruje použití formalismu DEVS pro vytvoření modelu a následné simulační ověření jeho správné funkce (analýzou logu lze získat potřebné informace o chování systému). Existuje samozřejmě možnost použití modelů na bázi DEVS k jiným účelům, např. pro stochastickou simulaci. V takovém případě je třeba průběžně sbírat data o využití zdrojů, délkách front apod. Jinou možností je simulace v reálném čase, propojená s reálným okolím. K tomu je třeba zajistit komunikaci modelu s okolím a synchronizovat simulaci s reálným časem. Model pak chápeme jako program s reálnými vstupy a výstupy. Výhodou tohoto způsobu programování je konformita programu s použitým formalismem, což vytváří prostor pro aplikaci matematických metod pro analýzu a verifikaci. Možnost simulace systému v simulovaném prostředí pak umožňuje důkladné testování vyvíjeného systému v průběhu vývoje. Tomuto způsobu vývoje systémů říkáme vývoj založený na simulaci (simulation-based development). Jeden z možných přístupů k takovému využití simulačního modelování bude popsán v kapitole 5.

```
****************** time: 0.5
* Internal Transition: aPingPong/Player A
    * New State:
        phase:  #receive
    * Output Port Configuration:
        send:  #event
    * Next scheduled internal transition at time Infinity
* External Transition: aPingPong/Player B
    * Input Port Configuration:
        receive: #event
    * New State:
        phase:  #send
* Root DEVS Output Port Configuration:
****************** time: 1.0
* Internal Transition: aPingPong/Player B
    * New State:
        phase:  #receive
    * Output Port Configuration:
        send:  #event
    * Next scheduled internal transition at time Infinity
* External Transition: aPingPong/Player A
    * Input Port Configuration:
        receive: #event
    * New State:
        phase:  #send
* Root DEVS Output Port Configuration:
****************** time: 1.5
* Internal Transition: aPingPong/Player A
    * New State:
        phase:  #receive
    * Output Port Configuration:
        send:  #event
    * Next scheduled internal transition at time Infinity
* External Transition: aPingPong/Player B
    * Input Port Configuration:
        receive: #event
    * New State:
        phase:  #send
* Root DEVS Output Port Configuration:
```

Obrázek 2.5: Záznam průběhu simulace

Kapitola 3

Systémy s dynamicky se měnící strukturou

Systémy, u kterých se předpokládá adaptace a vývoj, chápeme obecně jako systémy s dynamickou strukturou. Modelování a simulace takových systémů musí nutně zahrnout fenomény jako:

- měnící se vazby mezi komponentami,
- dynamicky vznikající a zanikající komponenty,
- dynamicky se měnící definice atomů.

Modifikace struktury modelu lze obecně docílit rozšířením simulačního modelování o možnost vyjádřit reflektivní vlastnosti systému. Princip systémové reflexe tkví v tom, že systém vidí svoji vlastní specifikaci a je schopen do ní dynamicky zasahovat. Tyto zásahy jsou systémem bezprostředně reflektovány, tj. mají okamžitý vliv na jeho strukturu a chování.

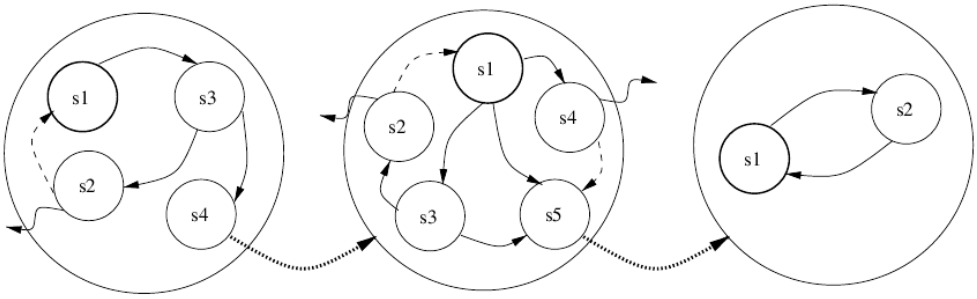
Problematikou reflexe v softwarových systémech se zabývá řada autorů, např. [Mae87, MMWY92, Paw98, TUN06, LRS01]. V oblasti formálních modelů systémů s diskrétními událostmi lze uvést vyvíjející se Petriho sítě [ABPD96] a různé varianty zavedení strukturní dynamiky do formalismu DEVS, jako je DSDEVS [Bar97] a Dynamic DEVS [Uhr01]. Zatímco DSDEVS umoňuje dynamiku na úrovni složených modelů a je konkrétně implementován např. v ADEVS, Dynamic DEVS (který bude uveden v následujcícím textu) je definován obecněji – dynamika je možná i na úrovni atomických modelů. Definice Dynamic DEVS zavádí strukturní přechody, které mění nikoliv stav, ale model (princip je na obr. 3.1, strukturní přechody jsou znázorněny tečkovaně).

Pro všechny přístupy, které zavádějí strukturní dynamiku, platí, že systém s měnící se strukturou je vždy specifikovatený jako systém statický. To znamená, že je simulovatelný ekvivalentním systémem se statickou strukturou, jehož stavový prostor je rozšířen tak, aby zahrnoval všechny varianty a přechodové funkce zahrnují specifikaci přechodových funkcí všech variant.

3.1 Dynamický DEVS

Dynamický DEVS Dynamický model je struktura [Uhr01]

$$DynM = (X, Y, m_0, M(m_0)),$$



Obrázek 3.1: Strukturní přechody [Uhr01]

kde

X je množina vstupních událostí,
Y je množina výstupních událostí,
$m_0 \in M(m_0)$ je počáteční model,
$M(m_0)$ je nejmenší množina se strukturou $\{(S, s_0, \delta_{int}, \delta_{ext}, \rho, \lambda, ta)|$
S je množina stavů,
$s_0 \in S$ je počáteční stav,
$\delta_{int} : S \longrightarrow S$ je interní přechodová funkce,
$\delta_{ext} : Q \times X \longrightarrow S$ je externí přechodová funkce, kde
$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ je množina úplných stavů,
$\rho : S \longrightarrow M(m_0)$ je funkce přechodu mezi modely,
$\lambda : S \longrightarrow Y$ je výstupní funkce,
$ta : S \longrightarrow R_0^+ \cup \{\infty\}$ je funkce posuvu času},
splňující podmínku
$\forall n \in M(m_0) : (\exists m \in M(m_0) : n = \rho(s^m), s^m \in S^m) \vee n = m_0$.

Dynamický složený DEVS Dynamický složený model je struktura [Uhr01]

$$DynN = (X, Y, n_0, N(n_0)),$$

kde

X je množina vstupních událostí,
Y je množina výstupních událostí,
$m_0 \in M(m_0)$ je počáteční model (počáteční konfigurace),
$N(n_0)$ je nejmenší množina se strukturou $\{(D, \rho_N, \{DynM_d\}, \{I_d\}, \{Z_{i,d}\}, select)|$
D je množina jmen submodelů,
$\rho_N : SSS \longrightarrow M(m_0)$ je funkce přechodu mezi modely, kde
$SSS = \times_{d \in D, m \in DynM_d} S^m$,
$\{DynM_d \mid d \in D\}$ je množina submodelů,
$\{I_d \mid d \in D \cup \{self\}\}$ je specifikace propojení,
$\forall d \in D \cup self : I_d \subseteq D \cup \{self\}, d \notin I_d$,
$\{Z_{i,d} \mid i \in I_d, d \in D \cup \{self\}\}$ je specifikace překladu událostí,
$Z_{i,d} : X \longrightarrow X_d$ pro $i = self$,
$Z_{i,d} : Y_i \longrightarrow Y$ pro $d = self$,
$Z_{i,d} : Y_i \longrightarrow X_d$ pro $i \neq self$ a $d \neq self$,

$select : 2^D - \{\} \longrightarrow D$ je preferenční funkce},
splňující podmínku
$\forall n \in N(m_0) : (\exists m \in N(m_0) : n = \rho_N(sss^m), sss^m \in SSS^m) \vee n = n_0$.

3.2 Abstraktní simulátor pro dynamický DEVS

Abstraktní kořenový simulátor Proměnné: Podřízený simulátor *child*, čas začátku a konce simulace t_{BEGIN}, t_{END}, aktuální čas t a čas následující plánované události t_{next}. Algoritmus:

$t := t_{BEGIN}$
pošli (i, t) podřízenému simulátoru *child*
počkej na $(done, s, t_{next})$ od podřízeného simulátoru *child*
$t := t_{next}$
dokud $t < t_{END}$ opakuj:
    pošli $(*, t)$ podřízenému simulátoru *child*
    počkej na $(done, s, t_{next})$ od podřízeného simulátoru *child*
    $t := t_{next}$

Abstraktní simulátor atomického modelu Promnné: Aktuální model m, čas poslední události t_{last}, čas následující události t_{next}, výstup y, úplný stav (s^m, e) a nadřazený simulátor *parent*. Algoritmus:

Na zprávu (i, t) od nadřazeného simulátoru *parent* reaguj takto:
    $t_{last} := t - e$
    $t_{next} := t_{last} + ta(s^m)$
    pošli $(done, s^m, t_{next})$ nadřazenému simulátoru *parent*

Na zprávu $(*, t)$ od nadřazeného simulátoru *parent* reaguj takto:
    $y := \lambda(s^m)$
    pošli (y, t) nadřazenému simulátoru *parent*
    $s^m := \delta_{int}(s^m)$
    $m := \rho(s^m)$
    $t_{last} := t$
    $t_{next} := t_{last} + ta(s^m)$
    pošli $(done, s^m, t_{next})$ nadřazenému simulátoru *parent*

Na zprávu (x, t) od nadřazeného simulátoru *parent* reaguj takto:
    $e := t - t_{last}$
    $s^m := \delta_{ext}(s^m, e, x)$
    $m := \rho(s^m)$
    $t_{last} := t$
    $t_{next} := t_{last} + ta(s^m)$
    pošli $(done, s^m, t_{next})$ nadřazenému simulátoru *parent*

Abstraktní simulátor (koordinátor) složeného modelu Proměnné: Aktuální model n, stav s^n, čas poslední události t_{last}, čas následující události t_{next}, výstup y, seznam

následujících událostí v submodelech *eventList*, množina submodelů s událostmi plánovaými na aktuální čas *imminent*, vybraný submodel d, seznam submodelů pro předání vstupu *receivers*, nadřazený simulátor *parent*. Algoritmus:

Na zprávu (i, t) od nadřazeného simulátoru *parent* reaguj takto:
$\forall d \in D$:
pošli (i, t) simulátoru d
počkej na $(done, s, t_{next})$ od všech *receivers*
aktualizuj s^n
aktualizuj *eventList*
$t_{last} := t$
$t_{next} := min\{t^d_{next} | d \in D\}$
pošli $(done, s^n, t_{next})$ nadřazenému simulátoru

Na zprávu $(*, t)$ od nadřazeného simulátoru *parent* reaguj takto:
$imminent := \{d | (d, t^d_{next}) \in eventList, t^d_{next} = t\}$
$d := select(imminent)$
pošli $(*, t)$ simulátoru d
počkej na (y, t) od simulátoru d
je-li $d \in I_{self}$ a $Z_{d,self}(y) \neq \emptyset$, pak:
$y_{self} = Z_{d,self}(y)$
pošli (y_{self}, t) nadřazenému simulátoru *parent*
$receivers := \{d | d \in D, self \in I_d, Z_{self,d}(x) \neq \emptyset\}$
$\forall d \in receivers$:
$x_d := Z_{self,d}(x)$
pošli (x_d, t) simulátoru d
počkej na $(done, s, t_{next})$ od všech *receivers* aktualizuj s^n
$D_{old} := D$
$n := \rho_n(s^n)$
$\forall d \in D - D_{old}$: pošli $(init, t)$ simulátoru d
počkej na $(done, s, t_{next})$ od všech $d \in D - D_{old}$
aktualizuj s^n
aktualizuj *eventList*
$t_{last} := t$
$t_{next} := min\{t^d_{next} | d \in D\}$
pošli $(done, s^n, t_{next})$ nadřazenému simulátoru

Na zprávu (x, t) od nadřazeného simulátoru reaguj takto:
$receivers := \{d | d \in D, self \in I_d, Z_{self,d}(x) \neq \emptyset\}$
$\forall d \in receivers$:
$x_d := Z_{self,d}(x)$
pošli (x_d, t) simulátoru d
počkej na $(done, s, t_{next})$ od všech *receivers* aktualizuj s^n
$D_{old} := D$
$n := \rho_n(s^n)$
$\forall d \in D - D_{old}$: pošli $(init, t)$ simulátoru d
počkej na $(done, s, t_{next})$ od všech $d \in D - D_{old}$
aktualizuj s^n
aktualizuj *eventList*

$t_{last} := t$
$t_{next} := min\{t^d_{next} | d \in D\}$
pošli $(done, s^n, t_{next})$ nadřazenému simulátoru

Korektní implementace V korektní implementaci při přijetí zprávy $(*, t)$ vždy platí $t = t_{next}$ a při přijetí zprávy (x, t) vždy platí $t_{last} \leqq t \leqq t_{next}$.

3.3 Aplikace

Dynamický DEVS (DynDEVS) byl použit jako formální základ systému James [Uhr01] pro modelování a simulaci agentních systémů. Formální model DynDEVS je koncipován jako uzavřený, neřeší problematiku klonování a editaci modelů asynchronními zásahy z vnějšku. Umožňuje však kromě vzniku a zániku modelů také jejich migraci rámci systému, což je pro multiagentní systémy typické. Vzhledem k tomu, že migraci provádí model, který se chce přesunout, je přesun součásti jeho vlastního interního nebo externího přechodu. V okamžiku migrace je tedy v definovaném stavu, přičemž $e = 0$. V definici dynamického DEVS tak není třeba pracovat s úplným stavem (s, e) a lze vystačit jen s tím, že inicializace nově přidávaných modelů správně nastaví čas poslední a plánované události. Ale v případě, že celou simulaci chápeme jako otevřený systém se vstupy a výstupy (případně také s vlastním, nezávislým časem),[1] musíme se zabývat asynchronními zásahy z vnějšího světa. Pak již nevystačíme se samotnou reflexí, jako v případě DynDEVS, ale musíme zpřístupnit reflektivní rozhraní vnějšímu systému. Přitom je třeba zajistit korektní chování při klonování modelů a editačních operacích v libovolném okamžiku. Touto problematikou se zabývá kapitola 4.

[1] Otevřené chápání simulace s možností neomezeně klonovat a editovat modely je nezbytné pro interaktivní evoluční vývoj a údržbu nepřetržitě běžících systémů.

Kapitola 4

Otevřená abstraktní architektura pro simulaci vyvíjejících se systémů

Kapitola definuje abstraktní architekturu univerzálního systému pro simulaci vyvíjejících se systémů. Nejprve jsou vymezeny požadované vlastnosti, poté je na základě jejich analýzy vytvořen návrh vhodné architektury a diskutovány možnosti použití.

4.1 Vymezení požadovaných vlastností

Systémy, kterými se hodláme nadále zabývat a pro které máme v úmyslu definovat vhodnou simulační architekturu, lze charakterizovat jako

- optimalizující, učící se, adaptivní systémy,
- otevřené, dynamicky zkoumatelné a dynamicky editovatelné systémy,
- reflektivní systémy.

Vzhledem k předpokládanému způsobu použití simulace v rámci návrhu, vývoje a údržby systémů mezi hlavní požadavky na podpůrné nástroje patří

- simulace v reálném čase, propojení simulace s okolní realitou,
- klonování a editace modelů v průběhu simulace,
- multisimulace – systémy simulací, simulace simulací,

V následujícím textu budou jednotlivé požadavky podrobně analyzovány a bude z nich odvozen návrh otevřené architektury pro modelování a simulaci vyvíjejících se systémů.

4.2 Simulace v reálném čase, propojení s reálným okolím

Reálný čas Při simulaci modelu v reálném čase plyne čas modelu stejně rychle jako čas reálný. Otázkou zůstává konkrétní hodnota časového údaje, resp. mapování mezi hodnotou světového času a času modelu. Lze rozlišit dva případy:

Lokální reálný čas. Tok času při simulaci v reálném čase může být chápán jako reálný čas běhu modelu. V takovém případě hodnota času v daném okamžiku udává reálnou dobu, po jakou model od svého startu běžel. Je-li simulace přerušena, reálný čas běhu modelu zůstává konstantní, je také pozastaven. Po opětovném spuštění hodnota času narůstá plynule od hodnoty času v okamžiku přerušení. Toto chápání reálného času modelu je souladu s definicí systému – jde o čas, v kterém systém existuje. Zastavení simulace znamená, že systém v době, kdy je simulace zastavena, neexistuje a jeho existence pokračuje až po opětovném spuštění simulace. Zastavování a spouštění simulace jsou operace, které model nevnímá a pokud mu metasystém (okolní realita) tuto informaci neposkytne, žije model ve svém čase, aniž by viděl nějaké přerušení.

Globální reálný čas. Tok tohoto časového údaje nelze žádným způsobem ovlivnit, natož zastavit. Má-li model pracovat s globálním reálným časem, pozastavení simulace způsobí problém – simulace se opozdí oproti reálnému času. Po opětovném spuštění se se zpožděním provedou akce, plánované na dobu, kdy byla simulace přerušena. Hodnota zpoždění (latence) může být v některých aplikacích kritická. Nedodržení časového rámce (timeoutu) pro provedneí akce je chápáno jako selhání a musí být systémem detekováno. To vyžaduje dodat časová omezení do modelu a tato omezení v průběhu simulace hlídat.

V některých případech je výhodné použít *proporcionální čas*. Rychlost toku modelového času vzhlem k rálnému času je pak dána koeficientem, tzv. *RT-fakorem*. Simulace je tedy na reálném čase závislá, ale běží buď zrychleně, nebo zpomaleně.

Simulace v reálném čase Simulaci je možné synchronizovat s reálným časem a propojit simulované a reálné entity do jednoho celku.[1] Základním předpokladem je dostatečně rychlé provádění jednotlivých kroků simulace. Pak stačí po každém kroku počkat na hodnotu reálného času rovnou času následující události v simulačním čase a pak provést další krok. Problémem, který je třeba řešit v případě distribuované RT simulace, je synchronizace hodin na všech uzlech. Možným řešením je jeden ze simulačních uzlů je prohlásit za referenční, ostatní se pak podle něho synchronizují.

Vstupně-výstupní aktivity Interakce simulátoru s vnějším světem je realizována s využitím prostředků operačního systému. Z hlediska modelu se interakce s reálným okolím řeší speciálními komponentami (modely), doplněnými o možnost posílat data do vnějšího prostředí a přijímat asynchronně přicházející data z vnějšího prostředí (viz např. [Hu04]). V atomickém modelu, reprezentujícím rozhraní na vnější svět, běží tzv. vstupně-výstupní aktivita. Jde o proces (nebo množinu procesů) operačního systému,[2] realizující výstupy a čekající na možné vstupy. Z procesu vstupně-výstupní aktivity atomického modelu může být kdykoli asynchronně (tj. nezávisle na procesu simulace) aktualizován stav atomického modelu. Při každé takové změně musí proces vstupně-výstupní aktivity okamžitě aktualizovat $t_{next} := clock$ hodnotou aktuálního reálného času a signalizovat tzv. *stavovou událost*

[1] Toto dává smysl např. v některých fázích vývoje řídicích systémů – po otestování modelu řídicího systému (přesněji řídicího systému realizovaného jako model) v simulovaném prostředí (obsahujícím simulovaný řízený systém) se simulované prostředí nahradí rozhraním na reálné okolí (a reálný řízený systém).

[2] Pojem operační systém zde zahrnuje jakékoliv výpočetní prostředí, ve kterém simulátor běží, tj. například JVM, Smalltalk, stejně tak, jako např. UNIX apod.

zasláním zprávy (se, t_{next}) nadřazenému simulátoru. Simulátor složeného modelu si v reakci na tuto zprávu aktualizuje t_{next} pro podřízenou komponentu v seznamu *eventList*, aktualizuje si vlastní t_{next} a pošle zprávu (se, t_{next}) svému nadřazenému simulátoru. Kořenový simulátor si v reakci na (se, t_{next}) aktualizuje t_{next}, takže v následujícím kroku může příslušný atomický model na stavovou událost zareagovat interním přechodem. Reakce simulátoru na zprávy přicházející z různých procesů (z procesu kořenového simulátoru a z procesů aktivit v atomech) musí být kvůli přístupu ke sdáleným proměnným simulátoru synchronizovány.

Abstraktní simulátor pro běh v reálném čase Předpokládejme, že *clock* je pseudoproměnná,[3] obsahující v každém okamžiku aktuální hodnotu reálného času. Na začátku simulace je třeba synchronizovat čas modelu a hodiny reálného času (*clock*) – lze nastavit buď $t_{BEGIN} := clock$, nebo $clock := t_{BEGIN}$. V prvním případě pracujeme s globálním časem, v druhém případě pracujeme časem reálného běhu modelu (hodnota *clock* udává reálnou dobu běhu od spuštění). Algoritmus kořenového simulátoru pro simulaci v reálném čase vypadá takto:

synchronizuj *clock* a t_{BEGIN}
$t := t_{BEGIN}$
pošli (i, t) podřízenému simulátoru *child*
počkej na $(done, t_{next})$ od podřízeného simulátoru *child*
$t := t_{next}$
dokud $clock < t_{END}$ opakuj:
    čekej na $clock = t$ nebo na přerušení zprávou (se, t_{next})
    proveď synchronizovaně:[4]
        $t := min\{clock, t_{next}, t\}$
        pošli $(*, t)$ podřízenému simulátoru *child*
        počkej na $(done, t_{next})$ od podřízeného simulátoru *child*
        $t := t_{next}$

Na zprávu (se, t_{next}) reaguj synchronizovaně takto:
    $t := t_{next}$

Simulátor atomického modelu z kap. 2 doplníme takto:

Na zprávu (se, t_{next}) reaguj synchronizovaně takto:
    $t := t_{next}$
    Pošli zprávu (se, t_{next}) nadřazenému simulátoru

Na ostatní zprávy reaguj stejně jako v kap. 2, ale synchronizovaně.

Simulátor složeného modelu z kap. 2 doplníme takto:

Na zprávu (se, t_{next}) od podřízeného simulátoru d reaguj synchronizovaně takto:
    $eventList := (eventList - \{(d, \_)\}) \cup \{(d, t^d_{next})\}$
    $t := t_{next}$

[3] Pseudoproměnná je proměnná, do které v doménové úrovni nelze přiřadit hodnotu, tj. je pro model k dispozici pouze pro čtení.

[4] Pod pojmem *synchronizované provedení* máme na mysli kritickou sekci hlídanou semaforem pro přístup k proměnným simulátoru.

Pošli zprávu (se, t_{next}) nadřazenému simulátoru
Na ostatní zprávy reaguj stejně jako v kap. 2, ale synchronizovaně.

Aktivace a deaktivace vstupně-výstupních aktivit Aktivity atomických modelů se startují v okamžiku spuštění simulace. Při zastavení simulace by měly být ukončeny. Toto je významný požadavek zejména v situaci, kdy dovolíme editaci modelu – pro takové zásahy se simulace přechodně zastaví a po provedení operace se znovu spustí. Kopírování, serializace apod. jsou bezpečné jen tehdy, když model neobsahuje žádné běžící procesy.

Toto lze realizovat tak, že ke spuštění vstupně-výstupních aktivit atomického modelu dojde v reakci na zprávu (i, t) a zastavení aktivit se provede v reakci na zprávu $(sync, t)$,[5] kterou zavedeme v následující sekci pro potřeby klonování modelů. Kořenový simulátor je pak třeba upravit tak, aby při každém startu a zastavení posílal podřízenému simulátoru zprávy (i, t) a $(sync, t)$.

Rozšířená definice atomického modelu Pro úplnost uvedeme ještě rozšířenou definici atomického modelu, který je určem pro simulaci v reálném čase a umožní dynamické klonování a editaci (jak bude ukázáno dále). *Atomický model se stavem a vstupně-výstupní aktivitou* je specifikován algebraickou strukturou

$$M = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta, s_0, (s, e), \alpha),$$

kde
$(X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta)$ je DEVS,
$s_0 \in S$ je počáteční stav,
$(s, e) \in Q$ je aktuální totální stav,
α je vstupně výstupní aktivita.

Vstupně-výstupní aktivita α běží v reálném čase a nezávisle na simulaci ovlivňuje stav $s \in S$. Při každé modifikaci stavu $s \in S$ oznamuje simulátor nadřazenému simulátoru stavovou událost (se, t) s hodnotou aktuálního reálného času. Aktivita α rozumí zprávám $(suspendActivity)$ a $(resumeActivity)$.

Rozšířená definice složeného modelu *Složený model se stavem* je definován libovolnou ze struktur[6]

$$N = (X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, select),$$
$$N = (X, Y, D, \{M_d\}, EIC, IC, EOC, select),$$

kde
X, Y, D, $\{I_d\}$, $\{Z_{i,d}\}$, $select$, EIC, EOC, IC
jsou definovány stejně jako v klasické definici (viz kap. 2),
$\{M_d \mid d \in D\}$ je množina submodelů, přičemž submodelem je
buď atomický model se stavem a vstupně-výstupní aktivitou,
nebo složený model se stavem.

[5] V simulačním prostředí SmallDEVS (viz kap. 5) je zastavování a spouštní vstupně-výstupních aktivit implementováno metodami atomického modelu `prepareToStop` a `prepareToStart`. Tyto metody jsou zděděné jako prázdné a v každém modelu, který implementuje rozhraní na realitu musí být adekvátně implementovány.

[6] Jejich souvislost byla vysvětlena v kap. 2.

4.3 Klonování a migrace systémů a simulací

DynDEVS (viz kap. 3, [Uhr01]) řeší problematiku sebemodifikace modelu, kdy ke změně modelu dochází v rámci plánovaných událostí v průběhu simulace. Neřeší ani problematiku klonování, ani vazbu na vnější prostředí, a tedy ani asynchronní editační zásahy do modelu z vnějšího prostředí, včetně ovládání simulátoru. Právě tuto problematiku nyní popíšeme a rozšíříme abstraktní simulátor DEVS tak, aby takové zásahy umožnil.

Kontinuace a DEVS Kontinuace je objekt, reprezentující pokračování výpočtu. Je to snímek, vytvořený v libovolném okamžiku z běžícího procesu. S kontinuací se pracuje jako s daty – lze ji uložit, přenést jinam, kopírovat apod. Kontinuace umožňují implementovat multitasking, klonování a migraci procesů, návraty v čase, prohledávání stavového prostoru.

Definice systému i definice DEVS odpovídají tomuto konceptu – systém s aktuálním stavem obsahuje veškerou potřebnou informaci k provedení simulace, tj. ke generování stavové trajektorie, tj. ke zjištění budoucího chování.

Dynamické klonování a migrace V případě systému s diskrétními událostmi je stav modelu v libovolném okamžiku určen úplným stavem (s, e). Díky uzavřenosti množiny systémů vzhledem ke skládání pro každý složený systém vždy existuje ekvivalentní základní systém s úplným stavem (s, e). Přitom s každým podsystémem lze nakládat stejně jako se složeným systémem. K tomu, aby bylo možné kdykoliv pořídit snímek (klon) systému nebo jeho části, je třeba vhodně doplnit abstraktní simulátor. Před každým klonováním a jinými operacemi nad běžící simulací je třeba formálně aktualizovat úplný stav modelu a od tohoto stavu pak po provedení operace dál pokračovat.

Vzhledem k tomu, že systém je definován nezávisle na kontextu[7], je možné bez problémů nechat klon systému dál existovat v jiném kontextu, než v jakém běžel originál. Je tak možné realizovat migraci komponent mezi subsystémy v rámci jedné simulace i mezi různými simulacemi.[8]

Úprava abstraktního simulátoru pro potřeby dynamické editace modelu – synchronizace simulátorů Abstraktní simulátor DEVS z kapitoly 2 implicitně připouští možnost zastavování a klonování celé simulace – simulátory si potřebnou informaci pamatují a po okopírování simulace plynule pokračuje. Problém nastává v případě strukturní editace modelů, t.j. při manipulaci s jednotlivými komponentami ve smyslu klonování, odstraňování a vkládání. Po takových operacích je třeba provést korektní inicializaci všech komponent dříve, než necháme simulaci pokračovat.

Inicializace se na začátku pokračování simulace musí vždy provést, aby byla aktualizována hodnota t_{last} a t_{next} ve všech komponentách (po případné editaci struktury se mohou tyto hodnoty stát neaktuálními) a aby nadřazený simulátor získal korektní informaci o čase následující události.

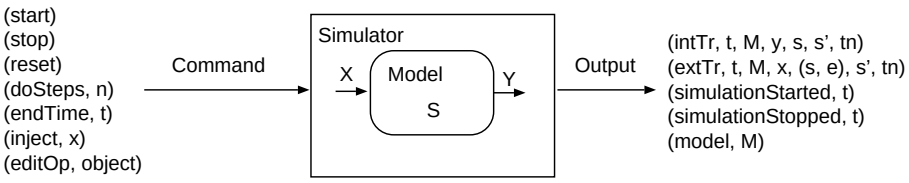
[7] Systém neví o hierarchicky nadřazeném systému ani o systémech, s nimiž je propojen. Na druhé straně, složený systém zná všechny své komponenty i jejich propojení.

[8] Toto je v případě jiného paradigmatu, jako je např. objektová orientace, problém – kontext objektu je dán všemi dostupnými objekty. Vyměnit kontext v průběhu výpočtu pak není tak jednoduché, jako je tomu v případě formalismu DEVS. Obě paradigmata však lze vhodně kombinovat – objekty lze bez problémů použít uvnitř komponent DEVS a migraci omezit na komponenty DEVS.

Simulátor dynamického DEVS [Uhr01] vždy inicializuje přidávané subsystémy. V našem případě ale počítáme se vznikem nových subsystémů klonováním s následnou editací, přičemž k naklonování systému může dojít asynchronně, v libovolném okamžiku. Proto musíme řešit korektní vytvoření klonu tak, aby jeho následná inicializace v novém kontextu proběhla správně a aby klon mohl plynule pokračovat v existenci.

Simulátor atomického modelu hodnotu e času stráveného ve stavu s standardně aktualizuje při zpracování zprávy (x, t) a tato hodnota je použita pouze pro výpočet $\delta_{ext}(s, e, x)$. Pro nepřetržitý běh simulace je to v pořádku, ale chceme-li simulaci přerušovat a manipulovat s jejími kontinuacemi (ve smyslu klonování, migrace, zpětného navracení apod.), musíme tuto hodnotu aktualizovat v okamžiku přerušení. Zavedeme proto novou zprávu $(sync, t)$.[9] Simulátor na $(sync, t)$ reaguje aktualizací hodnoty uplynulého času e. Příští zpráva (i, t) po opětovném odstartování simulace pak správně synchronizuje pokračování simulace.

Vazba simulátoru na vnější prostředí, simulátor jako systém Simulátor daného modelu chápeme opět jako systém se vstupy a výstupy. Vstupy jsou události $(start)$, $(stop)$, $(reset)$, $(endTime, t)$ a události představující požadavky na editační operace. Ty budou popsány v sekci 4.4. Sekvence výstupních událostí nese informaci o stavových trajektoriích a událostních segmentech všech submodelů, včeně změn stavu samotného simulátoru (spuštěn, zastaven apod.). Jde o události $(intTr, t, M, y, s, s')$, $(extTr, t, M, x, e, s, s')$, $(simulationStarted, t)$ a $(simulationStopped, t)$. Na editační a in(tro)spekční operace systém reaguje informací o struktuře a aktuálním stavu modelu, kterého se operace týká, tj. $(model, M)$ kde M je model se stavem (viz sekce 4.2).



Obrázek 4.1: Simulátor jako systém

Rozšíření simulátoru atomického modelu Abstraktní simulátor atomického modelu doplníme o reakci na zprávu $(sync, t)$. Kvůli přímé souvislosti uvedeme i reakci na zprávu (i, t), reakce na ostatní zprávy zůstávají stejné jako v kap. 2, jsou jen doplněny o generování událostí odpovídající změnám stavu modelu $(intTr, t, M, y, s, s')$ a $(extTr, t, M, x, e, s, s')$ do vnějšího prostředí.

Na zprávu $(reset)$ od nadřazeného simulátoru *parent* reaguj takto:
$s := s_0$
$e := 0$
$t_{last} := 0$
$t_{next} := \infty$
pošli $(done, t_{next})$ nadřazenému simulátoru *parent*

[9] Stejnou zprávu jsme již použili v předch. sekci k jinému účelu, a sice k zastavení vstupně-výstupní aktivity. Oba důvody k jejímu zavedení jsou nezávislé a může proto plnit oba účely současně.

Na zprávu (i, t) od nadřazeného simulátoru *parent* reaguj takto:
$t_{last} := t - e$
$t_{next} := t_{last} + ta(s)$
pošli $(resumeActivity)$ vstupně-výstupní aktivitě α
pošli $(done, t_{next})$ nadřazenému simulátoru *parent*

Na zprávu $(sync, t)$ od nadřazeného simulátoru *parent* reaguj takto:
pošli $(suspendActivity)$ vstupně-výstupní aktivitě α
$e := t - t_{last}$
pošli $(done, t_{next})$ nadřazenému simulátoru *parent*

Rozšíření simulátoru složeného modelu Abstraktní simulátor složeného modelu doplníme o reakci na zprávu $(sync, t)$. Kvůli přímé souvislosti uvedeme i reakce na zprávy (i, t) a $(done, t)$, reakce na ostatní zprávy zůstávají stejné jako v kap. 2.

Na zprávu $(reset)$ od nadřazeného simulátoru *parent* reaguj takto:
$\forall d \in D$:
pošli $(reset)$ simulátoru d
$activeChildren := D$

Na zprávu (i, t) od nadřazeného simulátoru *parent* reaguj takto (jako v kap. 2):
$\forall d \in D$:
pošli (i, t) simulátoru d
$activeChildren := D$

Na zprávu $(sync, t)$ od nadřazeného simulátoru *parent* reaguj takto:
$\forall d \in D$:
pošli $(sync, t)$ simulátoru d
$activeChildren := D$

Na zprávu $(done, t_{next})$ od podřízeného simulátoru d reaguj takto (jako v kap. 2):
$eventList := (eventList - \{(d, \_)\}) \cup \{(d, t^d_{next})\}$
$activeChildren := activeChildren - \{d\}$
Je-li $activeChildren = \emptyset$, pak:
Je-li poslední zpráva od nadřazeného simulátoru (i, t),
pak $t_{last} := max\{t^d_{last} | d \in D\}$
Je-li poslední zpráva od nadřazeného simulátoru $(*, t)$, nebo (x, t),
pak $t_{last} := t$
$t_{next} := min\{t^d_{next} | d \in D\}$
pošli $(done, t_{next})$ nadřazenému simulátoru

Rozšíření kořenového simulátoru (ovládání simulace) Kořenový simulátor reaguje na 4 zprávy z vnějšího prostředí, sloužící k ovládání simulace: $(reset)$, $(start)$, $(stop)$, $(endTime, te)$. Do vnějšího prostředí posílá metaudálosti $(simulationStarted, t)$, $(simulationStopped, t)$. Algoritmus:

$t := 0$
$t_{END} := \infty$

Na zprávu $(start)$ reaguj takto:

Jestliže $p = nil$, pak
$p :=$ nový proces:
pošli (i, t) podřízenému simulátoru *child*
počkej na $(done, t_{next})$ od podřízeného simulátoru *child*
$t := t_{next}$
$t_e := t_{END}$
generuj metaudálost $(simulationStarted, t)$
dokud $t < t_e$ opakuj:
pošli $(*, t)$ podřízenému simulátoru *child*
počkej na $(done, t_{next})$ od podřízeného simulátoru *child*
$t := t_{next}$
pošli $(sync, t)$ podřzenému simulátoru *child*
počkej na $(done, t_{next})$ od podřízeného simulátoru *child*
generuj metaudálost $(simulationStopped, t)$
$p := nil$

Na zprávu $(stop)$ reaguj takto:
Jestliže $t >= t_e$ a $p = nil$, generuj metaudálost $(simulationStopped, t)$
$t_e := t$

Na zprávu $(reset)$ reaguj takto:
pošli $(reset)$ podřízenému simulátoru *child*
počkej na $(done, \_)$ od podřízeného simulátoru *child*
$t := 0$
$t_e := 0$

Na zprávu $(endTime, te)$ reaguj takto:
$t_{END} := t_e := te$

Simulace v reálném čase Úpravu výše uvedeného simulátoru pro běh v reálném čase si na základě již uvedených informací laskavý čtenář provede sám.

4.4 Čtení a modifikace modelu, dynamické aplikační rozhraní simulátoru

V předchozí sekci byl popsán simulátor, reagující na operace pro ovládání simulace a připouštějící při pozastavené simulaci provádět klonování, migraci a editaci modelů. Nyní doplníme návrh dynamického aplikačního rozhraní simulátoru, které umožní

- dynamickou inspekci modelu a stavu simulace,
- dynamickou změnu vazeb mezi modely,modelsestavem
- dynamické vytváření a rušení modelů,
- dynamickou editaci definice atomů,
- dynamické vytváření, editace a rušení pomocných objektů.

Objekty Objekty, nad kterými jsou dynamicky (za běhu) k dispozici editační operace, jsou prvky množin

- $SIMULATIONS$ – množina simulací,
- $MODELS$ – množina modelů,
- $INPORTS$ – množina vstupních portů modelu,
- $OUTPORTS$ – množina výstupních portů modelu,
- $SLOTS$ – množina stavových proměnných modelu s asociovanými složkami aktuálního stavu modelu,[10]
- $METHODS$ – množina funkcí, specifikujících atomický model,
- $COUPLINGS$ – množina propojení, specifikujících složený model.

Tyto množiny jsou hierarchicky organizovány. Na obr. 4.2 je ukázka hierarchie objektů existujících na počátku simulace modelu PingPong z kapitoly 2.[11] Nedochází-li k zásahům do simulace z vnějšku, v průběhu simulace se mění pouze obsah slotů atomických modelů.

Operace Nad objektem o, resp. nad množinou objektů O zavedeme tyto operace:[12]

- NEW – vytvoří nový objekt o (parametrem je jméno a specifikace objektu) a přidá ho do množiny O,
- $COPY$ – zkopíruje objekt o do proměnné $PASTEBUFFER$,
- $PASTE$ – vloží kopii objektu uloženého v proměnné $PASTEBUFFER$ do množiny O,
- $DELETE$ – odstraní objekt o z množiny O,
- CUT – odstraní objekt o z množiny O a umístí ho do proměnné $PASTEBUFFER$,
- $RENAME$ – přejmenuje objekt o (parametrem je nové jméno) v množině O,
- $SAVE$ – uloží objekt o do vstupně výstupní proměnné IO,[13]
- $LOAD$ – vloží objekt ze vstupně výstupní proměnné IO do množiny O,[14]
- $EDIT$ – modifikuje specifikaci objektu o (parametrem je nová specifikace),
- makrooperace složené z uvedených operací.

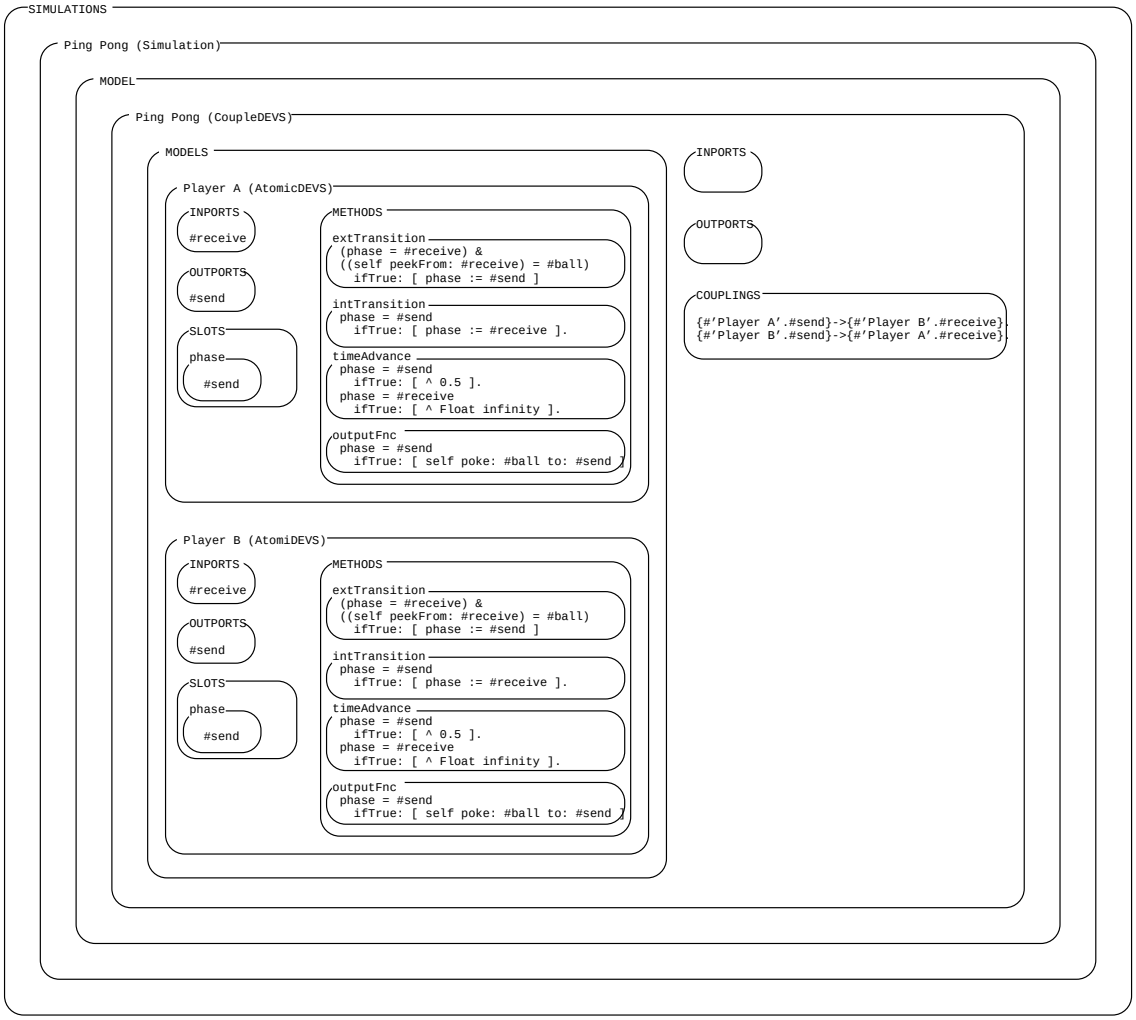
[10] V objektově orientovaném prostředí založeném na prototypových objektech (viz kap. 5) pracujeme ještě se speciálními sloty $DELEGATES$.

[11] Původní model v kap. 2 je vytvořen z instancí předem připravených tříd, a jeho implementace obshuje i inicializační metody, které v případě, že se zaměříme jen na objekty simulace bez ohledu na způsob vytvoření, nemají pro specifikaci modelu žádný význam, známe-li aktuální stav. Třídy a inicializace instancí jsou implementační detaily vynucené konkrétním prostředím a nemají pro simulaci význam. Ani definice DEVS s ničím takovým nepočítá.

[12] Lze potenciálně definovat i jinou sadu operací, kterou lze docílit téhož efektu.

[13] Prakticky je objekt uložen v textové podobě.

[14] Prakticky proběhne rekonstrukce objektu z textové reprezentace.



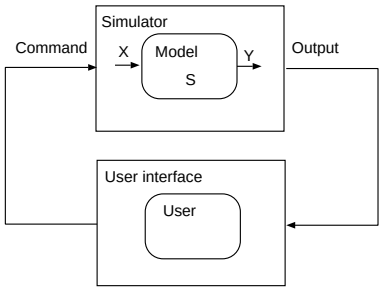
Obrázek 4.2: Hierarchie objektů – snímek stavu simulace PingPong

Uvedené operace lze bez omezení aplikovat v libovolném okamžiku na libovolné objekty. Jako příklad lze uvést všechny objekty na obr. 4.2. Pokud výsledkem operace je funkční model, simulace může pokračovat. Jinak je simulace ukončena kvůli chybě.

Jména objektů Operace jsou parametrizovány jmény objektů a modifikují tyto objekty nebo množiny tyto objekty obsahující. Vzhledem k hierarchické struktuře modelů lze pro identifikaci objektu použít pathname, tj. sekvenci jmen všech objektů, které jsou hierarchicky vnořeny a požadovaný objekt obsahují, včetně jména tohoto objektu. Např. v systému na obr. 4.2 můžeme objekt, odpovídající výstupní funkci hráče B identifikovat jménem `"SIMULATIONS/Ping Pong/MODEL/Ping Pong/Player B/METHODS/outputFnc"`.[15]

Editační zásahy do simulace z vnějšího prostředí Uvažujme situaci, kdy požadavky na editační operace přicházejí od vnějšího systému, např. od uživatele (viz obr. 4.3). Pokud

[15] Vzhledem k tomu, že některé složky jména lze eliminovat, aniž by došlo k nejednoznačnosti, můžeme tentýž objekt jednodušeji identifikovat jménem `"SIMULATIONS/Ping Pong/Player B/METHODS/outputFnc"`.



Obrázek 4.3: Architektura pro interaktivní evoluční vývoj

simulace běží, pro každou editační operaci nad modelem je nutné pozastavit simulaci, provést požadovanou operaci a poté simulaci znovu spustit:

pošli (*stop*) kořenovému simulátoru
počkej na (*simulationStopped*),
proveď editační operaci,
pošli (*start*) kořenovému simulátoru
počkej na (*simulationStarted*)

Pokud simulace neběží, provede se přímo editační operace. Na každý realizovatelný požadavek na editační operaci simulátor zareaguje provedením příslušné operace. Po provedení editační operace simulátor generuje výstupní událost $(model, M)$, kde M je model, kterého se editační operace týkala.[16]

Reflektivní editační zásahy – sebemodifikace modelu Tento typ modifikace modelu umožňuje definice DynDEVS. Editace modelu je v případě formalismu DynDEVS podle [Uhr01] dovolena v rámci interního nebo externího přechodu. Vzhledem k tomu, že algorimus simulace je synchronní, je provedení jakéhokoli v zásahu do modelu v tomto okamžiku bezpečné. Model může číst a editovat sám sebe, ale nepřímo také kontext, v kterém se nachází. Nadřazený model má k dispozici stav všech svých subsystémů a součástí těchto dílčích stavů mohou být požadavky na editační zásahy v nadřazeném modelu. Vzhledem k tomu, že složený systém je konceptuálně systém paralelní, musí nadřazený systém řešit konflikty editačních zásahů, požadovaných v tomtéž okamžiku různými subsystémy současně.[17] V případě, že operaci nelze provést, neprovede se. Informaci o kontextu, v kterém se model nachází, může získat jedině prostřednictvím svých vstupů – to lze zajistit vhodnou strukturou hierarchicky nadřazeného nadřazeného složeného modelu.

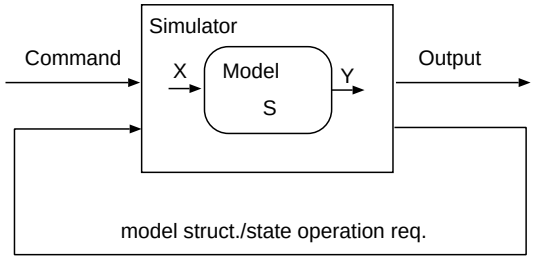
Náš přístup je založen na metaúrovňové architektuře simulátoru. Model chápeme jako systém doménové úrovně, simulátor chápeme jako metasystém,[18] který má na vstupu frontu požadavků na inspekci a editaci, včetně případných maker (skriptů). Fronta se vyprazdňuje po každém kroku simulace, přičemž všechny proveditelné požadavky na editaci

[16] Včetně aktuálního stavu a včetně případných submodelů, viz definice v sekci4.2.

[17] Například při provedení interního přechodu je generován výstup, který invokuje externí přechody v navazujících komponentách. Ve všech těchto přechodech mohou být odpovídající změnou stavu požadovány změny v nadřazeném modelu.

[18] V obou případech jde o systémy specifikovatelné formalismem DEVS.

modelu jsou zpracovány.[19] Reflektivity je dosaženo propojením výstupů simulátoru na jeho vstup (viz obr. 4.4). Vzhledem k tomu, že na výstupu simulátoru je k dispozici stav modelu, model může takto komunikovat s vlastním simulátorem a docílit tak vykonání požadovaných instrospekčních i reflektivních operací.



Obrázek 4.4: Simulace reflektivního systému

4.5 Systémy simulací (multisimulace)

Pozvedněme se nyní mimo čas a prostor jedné simulace. Na této úrovni pracujeme s mnoha nezávislými simulacemi a jejich časoprostory a zabýváme se dynamickým vytvářením, rušením a propojováním simulací. Toto dává smysl pro optimalizaci, vnořenou a reflektivní simulaci a pro multiparadigmatickou simulaci.

Optimalizace založená na simulaci spočívá v prohledávání prostoru všech simulací (a všech modelů) s cílem najít nejlepší model podle daných kriterií, přičemž míra, do jaké jsou kritéria splněna, je odvozena od výsledků simulačních experimentů.

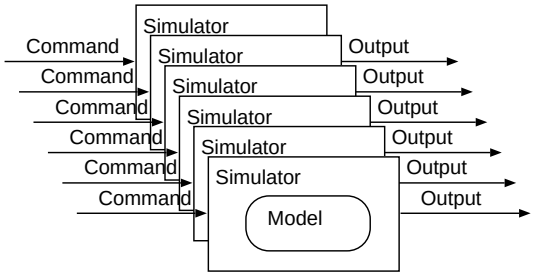
Vnořená a reflektivní simulace je v podstatě simulace simulujících systémů – součástí modelu M je simulátor jiného modelu M'. Pokud M' je modelem M, jde o reflektivní simulaci – simulujeme systém, který obsahuje model sebe sama a provádí s ním simulační experimenty.

Multiparadigmatická simulace je realizována různými simulátory pro různá modelovací paradigmata. Přitom simulátory dílčích modelů (specifikovaných různými formalismy spjatými s použitými paradigmaty) jsou synchronizovány a propojeny v rámci jedné multisimulace.

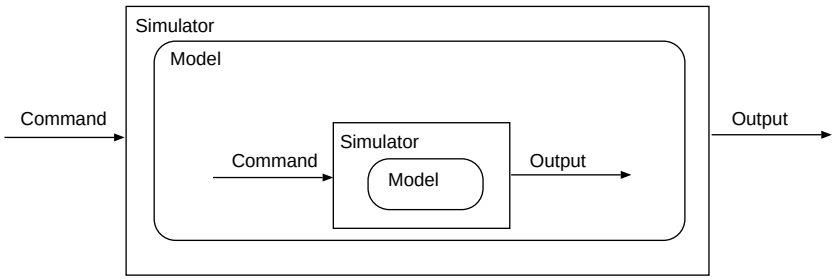
Kompozice simulátorů Uvedené případy se liší především tím, zda simulace běží z konceptuálního pohledu paralelně (na stejné úrovni), nebo zda jsou simulace vnořeny. V prvním případě (paralelní simulace) je simulátor zapouzdřen tak, že běží se synchronizovaným časem a zpřístupňuje rozhraní vnořeného modelu. V druhém případě (vnořená simulace) se simulátor jeví jako komponenta, jejíž rozhraní (vstupy a výstupy) zpřístupňuje operace vnořeného simulátoru, což jsou z pohledu vnořeného modelu metaoperace a metainformace (na této úrovni můžeme s modelem a jeho simulací zacházet zcela libovolně bez jeho "vědomí"). Vnořený model pak běží ve vlastním, na vnějším modelu nezávislém,

[19] Z hlediska praktické implementace lze frontu v některých případech vynechat a editační operace provádět přímo. Pro jednoduché zásahy to nevadí, obecně je to ale nutné, kvůli serializaci požadavků při násobnému přístupu k refl. rozhraní simulátoru z mnoha submodelů současně.

čase. Je samozřejmě možné paralelní simulaci vnořit a vnořené simulace nechat běžet paralelně. Vzhledem k tomu, že simulátor se jeví jako komponenta DEVS, je možné ho přirozeně použít jako submodel složeného modelu a pracovat s ním stejně jako s ostatními komponentami.



Obrázek 4.5: Paralelní simulace



Obrázek 4.6: Vnořená simulace

Multisimulační systém Multisimulační systém je systém, jehož subsystémy jsou simulace, resp. simulátory.[20] Simulátor můžeme jednak modelovat (implementovat) přímo jako DEVS (viz kap. 7), jednak ho můžeme (bez ohledu na realizaci) zapouzdřit do komponenty DEVS (tj. dát mu kompatibilní rozhraní) a použít ho jako subsystém multisimulačního systému, který specifikujeme jako DEVS a simulujeme simulátorem DEVS (příkladem může být simulátor pro jiné paradigma, uvedený v kapitole 6). Takto zapouzdřený simulátor může pracovat buď s nezávislým časem, nebo se časem synchronizovaným s nadřazeným systémem, podle toho, o jaký typ aplikace jde (příklad vnořené simulace s nezávislým časem je uveden v kapitole 8).

Plánování simulací V multisimulačním prostředí je třeba řešit otázku přidělování procesoru jednotlivým simulacím. Procesy jednotlivých simulací jsou obvykle implementovány jako procesy hostitelského operačního systému. Pak se o přidělování procesoru stará plánovač operačního systému. V případě multisimulačního prostředí SmallDEVS, které bude

[20] Pojem simulace, resp. simulátor, používáme analogicky k pojmu proces, resp. procesor v operačních systémech. Proces (v terminologii operačních systémů) je objekt, obsahující program a aktuální stav jeho provádění. Totéž platí obecně pro procesor. Stejně chápeme i simulaci, resp. simulátor - zahrnuje model a jeho aktuální stav. Z jiného (nadčasového) úhlu pohledu jsou jak procesy, tak simulace sekvence událostí měnící stav. V našem pojetí jak proces, tak simulace reprezentují systémy, schopné generovat budoucí události.

popsáno v kapitole 5, se o přidělování procesoru stará standardní plánovač Smalltalku. Obdobně by se situace řešila i v jiných podobných prostředích, jako je např. Java.

4.6 Shrnutí

V návaznosti na existující přístupy, zabývajícími se modely s vyvíjející se strukturou (zmíněnými v kapitole 3) byla definována abstraktní architektura univerzálního simulačního systému s dynamickým aplikačním rozhraním, umožňující jak interaktivní vývoj modelu, tak reflektivitu.

Ke klíčovým atributům této architektury patří simulace v reálném čase, propojení simulace s okolní realitou, klonování, migrace a editace submodelů za běhu, metaúrovňová multisimulační architektura (simulátory jsou systémy, s kterými se zachází stejně jako s modely).

Konkrétní implementací této abstraktní architektury je systém SmallDEVS, kterému se věnuje kapitola 5.

Kapitola 5

Vývoj systémů v simulaci, nástroj SmallDEVS

V této kapitole bude představen konkrétní nástroj (SmallDEVS [Jan06]), implementující reflektivitu a podporující interaktivní inkrementální vývoj systému za běhu. Implementačním prostředím je Smalltalk. Popsané principy lze uplatnit i v jiných prostředích. Existující implementace ovšem využívá toho, že Smalltalk sám o sobě je vysoce portabilním a do jiných systémů vnořitelným operačním systémem, podporujícím perzistenci, multitasking, interaktivitu a experimentální programovní (exploratory programming). Smalltalk a jeho principy jsou také jedním z hlavních zdrojů inspirace pro vybudování systému SmallDEVS.

5.1 Motivace a zvolený přístup

Jako motivační příklad lze uvést inkrementální vývoj řídicího systému autonomího mobilního robota v simulovaném prostředí. Jde o vývoj systému, jehož specifikace není na počátku vývoje zcela jasná a je ji třeba dodatečně upřesňovat na základě výsledků testů, prováděných na pokusných realizacích v různých prostředích, a to jak simulovaných, tak reálných. Vzhledem k tomu, že je mnohdy nutné dovyvíjet realizovaný systém při běžném provozu v cílovém prostředí, je nezbytné zachovat model řízení v průběhu celého vývoje, včetně cílového nasazení, s možností vrátit se kdykoli zpět do prostředí simulovaného. Cílová realizace řídicího systému pak musí obsahovat simulátor modelu řízení, běžící v reálném čase a propojený s reálným okolím. Přitom je vhodné zajistit vzdálené monitorování stavu a inkrementální zásahy vývojáře do modelu řízení za běžného provozu. Uvedený styl vývoje systémů klade specifické nároky na podpůrné prostředky. Zvolený přístup je založen na kombinaci čtyř paradigmat:

Experimentální modelování (exploratory modeling) Jednoduše řečeno, netvoříme model, abychom s ním experimentovali, ale experimentujeme, abychom našli (vytvořili) správný model. Jde v podstatě o formu evolučního programování s interaktivní fitness funkcí a s interaktivním generováním různých mutací modelu.

Otevřené a flexibilní podpůrné prostředky Experimentální modelování je třeba podpořit odpovídajícím nástrojem. K jeho hlavním rysům musí patřit reflektivní aplikační rozhraní a interaktivní vizuální nástroje pro inspekci a editaci modelů a manipulaci se simulacemi. Otevřenost a flexibilitu potřebnou pro inkrementální zásahy

(programové i interaktivní) v době běhu může zajisit objektově orientovaný přístup smalltalkovského typu, ale vyšší míru flexibility a bezpečnosti (díky jednodušší realizaci) může zajistit přístup založený na prototypových objektech.

Propojení reality se simulací (reality-in-the-loop simulation) V průběhu vývoje a hlavně v cílové realizaci se předpokládá propojení modelu (který ve vhodném okamžiku formálně prohlásíme za cílovou realizaci) s reálným okolím. K tomu je nutné okolní realitu zpřístupnit se stejným rozhraním, jaké mají komponenty modelu. Prakticky to znamená prvky reálného okolí zpřístupnit formou speciálních atomických komponent.

Vývoj v simulaci a zachování modelu (model continuity) Vývoj založený na simulaci je vhodné kromě postupu od simulace k realitě umožnit i reverzně, tj. umožnit návraty z reálného nasazení zpět do simulace. Výhodou je pak možnost pracovat i v simulaci s daty získanými z běhu v reálném prostředí. Prakticky to znamená umožnit dynamické klonování komponent modelu (resp. cílového programu) a přenos těchto klonů do simulovaného prostředí.

5.2 Experimentální programování (exploratory programming) a Smalltalk

Smalltalk je všeobecně chápán jako vzorový jazyk pro experimentální programování (exploratory programming), neboť poskytuje účinné prostředky pro interaktivní zásahy do běžícího systému. V této části popíšeme podrobněji některé související skutečnosti.

Experimentální programování (exploratory programming) Pojem Exploratory programming je těžko přeložitelný. Použitelný je termín zkoumavé, zkoumající, pokusné, či *experimentální programování (EP)*. Poslední z uvedených termínů budeme preferovat.

Tento styl programování umožňuje rychlou tvorbu prototypů, tedy programů, o kterých předpokladáme, že možná řeší zadaný problém. Je vhodný v situacích, kdy není k dispozici dostatečná specifikace budoucího díla a vhodný způsob realizace je třeba teprve objevit na základě experimentů s pokusnými realizacemi. Jde o interaktivní prohlednávání (exploration) prostoru všech možných programů s cílem najít ten, který nejlépe vyhovuje obvykle velmi nejasně zadaným požadavkům. Taková situace je typická v oblasti umělé inteligence, ale i v řadě jiných případů. LISP, Smalltalk, Prolog a Self jsou typické jazyky umožňující právě tento styl programování. Smalltalk (spolu s jím přímo inspirovanými jazyky, jako je Self) je navíc zajímavý tím, že tento způsob programování podporuje i vizuálními nástroji.

Inkrementální kompilace Technicky je pro podporu experimentálního programování nezbytná možnost okamžitého přechodu od vytvoření programu k jeho bezprostřednímu interaktivnímu (ale i automatickému) otestování. Kompilace tedy musí trvat zlomky, maximalně jednotky vteřin, nikoliv minuty, jak je tomu u staticky typovaných mainstreamových jazyků. Také musí probíhat z pohledu uživatele skrytě, bez nutnosti explicitně spouštět kompilátor. Toho lze prakticky docílit inkrementálním kompilátorem, který zpracovává jen dílčí části vytvářeného programového díla právě v okamžiku jejich modifikace a přeložený kód inkrementálně zakomponovává do výsledného celku. V případě Smalltalku je největší

a jedinou jednotkou překladu jedna metoda. Výsledkem je objekt (konkrétně zkompilovaná metoda), který je zaregistrován ve struktuře příslušné třídy, což je také objekt, který je součástí objektové paměti (object memory) Smalltalku.

Programování za běhu Podstatou EP je téměř nepřetržitě využívaná možnost zkoumat stav a průběh výpočtu, přičemž program, který řídí výpočet, je okamžitě modifikovatelný inkrementálními zásahy. Stav výpočtu je také v libovolném okamžiku v široké míře editovatelný. Smalltalk poskytuje vizuální nástroje pro zkoumání a editaci programu (t.j. systému tříd a jejich metod) i stavu výpočtu (t.j. obsahu instančních proměnných jednotlivých objektů). Jak obsah instančních proměnných živých objektů, tak i jejich třídy a metody lze v libovolném okamžiku modifikovat a bezprostředně zkoumat důsledky takových zásahů. Takže jak zkoumání běhu programu, tak samotné programování probíhá současně. Takovýmto spojením programování s okamžitým ověřováním chování lze obecně dosáhnout mnohem vyšší produktivity programátorské práce i vyšší kvality kódu než zdlouhavým opakovaným programováním, následnou explicitní kompilací a spouštěním výsledné aplikace s omezenými možnostmi sledovat chování programu, jak je tomu v případě běžných programovacích jazyků, použitých pro prototypování a evoluční vývoj. Běžné programovací jazyky totiž podporují klasický přístup softwarového inženýrství, založený na důkladné analýze požadavků a z ní odvozené přesné specifikace programového díla. To ovšem v případě nejasných požadavků nelze aplikovat.

Objektová paměť' Zajímavým důsledkem přístupu EP je fakt, že tak zvaný zdrojový kód ztrácí původní význam, protože to, čeho se programátor snaží inkrementálními zásahy docílit, není primárně zdrojový text, ale uspokojivě běžící programové dílo. To se totiž vývojář snaží neustále testovat, analyzovat a upravovat. V případě Smalltalku tedy jde programátorovi primárně o vytvoření fungujícícho systému živých objektů v perzistentní objektové paměti, nikoliv o vytvoření souborů se zdrojovými texty, které po zkompilování a spuštění takový systém objektů při odstartování vytvoří. Programátor ve Smalltalku needitуje žádné textové soubory, ale inkrementálně modifikuje objekty přímo v objektové paměti za pomoci inkrementálního kompilátoru spojeného s nástroji pro navigaci v systému objektů. Kompilátor je automaticky aktivován jako reakce na zásahy programátora, který edituje jen jednotlivé textové reprezentace metod, které jsou mu zpřístupňeny v nástroji na prohlížení tříd. Textovou podobu metod Smalltalk spravuje ve vlastní režii a umožňuje přístup i k historickým verzím metod. Tyto informace Smalltalk ukládá do logu mimo objektovou paměť. V případě, že by tento log z jakéhokoliv důvodu nebyl k dispozici, nabízí k prohlížení a editaci dekompilovanou podobu metod. Tento přístup jenom podtrhuje nezávislost Smalltalku na externích souborech a textech. Předmětem zkoumání a modifikace jsou výlučně jen živé objekty v objektové paměti Smalltalku.

Export a import objektů Soubory s textovou podobou tříd a jejich metod lze v případě potřeby kdykoliv ze Smalltalku vygenerovat, ale rozhodně je nelze považovat za primární. Přestože tak mohou být viděny, nejsou to zdrojové texty v pravém smyslu slova. Needitují se, slouží typicky jen pro případný přenos vytvořeného díla do objektové paměti jiného Smalltalku. Takto vygenerované texty nejsou určeny k tomu, aby je četl, analyzoval a editoval člověk. Jak prohlížení, tak i analýza a editace systému tříd se přirozeně realizuje v jejich primární, živé podobě, t.j. v podobě objektů v objektové paměti. Proto také mnohé Smalltalky preferují export nikoliv v podobě, která by zdrojový text připomínala,

ale aději v XML, což je dnes považováno za standard pro textovou reprezentaci tříd a objektů, určenou ke strojovému zpracování.

Perzistence, obraz paměti Jelikož nepracujeme se zdrojovými texty v souborech, ale s objektovou pamětí, musí systém pro EP podporovat možnost jejího kompletního uložení na externí médium (obvykle do souboru) v podobě tzv. obrazu (image). Tento obraz objektové paměti je při startu systému (Smalltalku) načten a objektová paměť je rekonstruována přesně v té podobě, v jaké byla předtím uložena. Některé Smalltalky jsou schopny nepřetržitě obraz objektové paměti udržovat na disku a také řešit odkládání nepoužívaných částí na disk v případě nedostatku volné operační paměti. Takovým Smalltalkem je např. systém GemStone [**?**], deklarovaný jako objektově orientovaný databázový systém pro klient-server aplikace.

Reflektivita a dynamičnost Smalltalk je poměrně rozsáhlý programový systém vytvořený ve Smalltalku samotném. Jde o typickou ukázku systému, který je sám sebe schopen za běhu vyvíjet. K tomu je třeba, aby objekty mohly zkoumat a modifikovat objekty (včetně sebe). Objekt může zkoumat i modifikovat jak třídy (tj. množinu metod a specifikaci reprezentace stavu), tak jejich instance (tj. obsah instančních proměnných). Modifikace třídy ve Smalltalku okamžitě ovlivní strukturu a chování všech jejích instací.

Smalltalk je dynamický jazyk. To znamená, že programové struktury mohou vznikat a zanikat za běhu. Smalltalk je systém, vytvořený a běžící ve Smalltalku. Programování ve Smalltalku znamená postupnou modifikaci Smalltalku směrem ke kýžené aplikaci. Smalltalk, resp. aplikace ve Smalltalku vytvářená, tedy ve skutečnosti provádí sebemodifikaci a tím se toto programové dílo vyvíjí.

Použitelnost EP EP je obecně vhodné v těchto situacích:

- malé projekty,
- malé týmy (v ideálním případě aplikující zásady extremního programování),
- nejasné zadání,
- omezené prostředky (čas, prostor, peníze).

EP je naopak nevhodné pro velké projekty a velké týmy. Tam se počítá právě se zavedenými běžnými technikami softwarového inženýrství s důkladnou předběžnou analýzou a s existujícími rozsáhlými a otestovanými knihovnami pro běžné, obvyle staticky typované jazyky. Také veškeré podpůrné nástroje, umožňující týmový vývoj, jsou založeny na správě souborů se zdrojovými texty, které se považují za primární a jedinou udržovanou podobu programového díla.

Bezpečnost Jazyky pro EP jsou vesměs jazyky s dynamickou typovou kontrolou. Jazyky s dynamickou typovou kontrolou nevyžadují deklarace typů promenných, protože nejsou k překladu zapotřebí. Některé jazyky (např. Strongtalk [**?**]) umožňují nepovinně deklarovat typy proměnných tam, kde to má vysvětlující, dokumentační význam. Smalltalk, LISP, Self, Prolog ani Python s deklaracemi typů nepočítají. Je však třeba zdůraznit, že ve všech těchto případech jde o jazyky typově bezpečné, tj. nemůže za žádných okolností dojít k chybě, způsobené aplikací operace nad nesprávným typem dat.

Potenciální problém dynamicky typovaných jazyků ale tkví v tom, že na případné typové nekompatibility se přijde až za běhu. Ale vzhledem k tomu, že EP je založeno na velmi důkladném testování, pravděpodobnost neodhalení takové chyby je obvykle mnohem přijatelnější než cena vývoje s použitím klasického staticky typovaného jazyka (nehledě na fakt, že statická typová kontrola neodhalí jiné vážné chyby, které nejsou způsobeny typovou nekompatibilitu).

5.3 Objektová orientace založená na prototypových objektech

Dalším zdrojem inspirace pro SmallDEVS je objektová orientace založená na prototypových objektech [Lie86]. Vzorovým jazykem, založeným na prototypových objektech je Self [US87]. Přímé použití tohoto jazyka by pro naše účely bylo problematické z hlediska interoperability a portability, proto výhodně využujeme toho, že tento styl programování je k dispozici přímo ve Smalltalku díky rozšíření, které je implementováno v balíku Prototypes [MA04]. Jde o jednoduchý framework pro Smalltalk, nikoliv o vnořený jazyk. Jazykem pro specifikaci metod zůstává Smalltalk. Na rozdíl od klasického beztřídního OO jazyka Selfu [US87] jsou zde proto některé mírné odlišnosti:

- Reprezentace objektů zůstává zachována, takže se rozlišují datové sloty a metody.
- Jazyk pro specifikaci metod je Smalltalk, takže pro přístup ke slotům je třeba zasílat zprávy explicitně (přes self).
- Lze libovolně využívat existujících objektů Smalltalku a kombinovat prototypové objekty s objekty definovanými třídami.
- Všechny prototypové objekty jsou instancemi třídy PrototypeObject, která je podtřídou Behavior, stejně jako metatřídy.

Základní princip prototypových objektů lze charakterizovat takto:

- Prototypový objekt má sloty, metody a delegáty (viz dále), umí reagovat na zprávy. Reakce na zprávu spočívá v provedení odpovídající metody nebo vrácení obsahu odpovídajícícho slotu.
- Dědičnost je realizována delegací. Nerozumí-li objekt zprávě, hledá se odpovídající metoda (nebo slot) v delegátech, po nalezení se metoda provede v kontextu původního příjemce zprávy.
- Prototypový objekt rozumí metaoperacím pro editaci objektu (manipulaci se sloty, metodami a delegáty).
- Nad libovolným prototypovým objektem lze v libovolném okamžiku otevřít Inspektor, který umožní interaktivní zkoumání a editaci objektu.

Typické použití a současně princip programování bez tříd si ukážeme na příkladech.

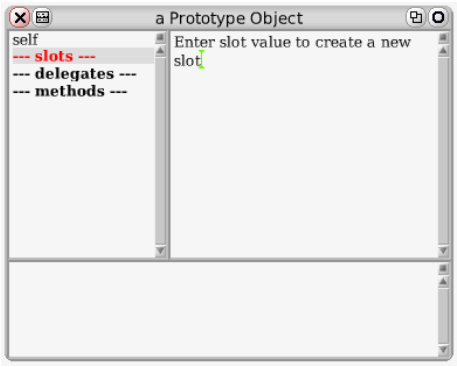
Vytvoření prázdného objektu

```
anObject ← PrototypeObject new.
```

Inspektor

```
anObject inspect.
```

Výsledkem je otevření inspektoru (obr. 5.1).



Obrázek 5.1: Inspektor prototypového objektu

Inspektor je kombinací browseru a klasického smalltalkovského inspektoru, protože prototypový objekt je v podstatě kombinací třídy a její instance. Pomocí tohoto nástroje je možné zkoumat a modifikovat stav objektu, ale také jeho metody a delegáty (delegáti jsou objekty, na které objekt deleguje zprávy, které není sám schopen obsloužit – takto je v beztřídních systémech realizováno sdílené chování množiny objektů, tj. dědičnost).

Klonování

```
anotherObject ← anObject clone.
```

Klon je mělkou kopií originálu. Pro jiné než mělké kopie je třeba vytvořit vlastní metody.

Operace nad sloty

```
anObject addSlot: 'x'.
anObject addSlot: 'x' withValue: 0.
anObject removeSlot: 'x'.
```

Téhož efektu lze docílit pomocí inspektoru. Stačí vybrat –slots– a sledovat instrukce, případně vybrat konkrétní slot a vyvolat kontextovou nabídku.

Operace nad metodami

```
anObject addMethod: 'printOnTranscript
                     Transcript show: self x printString'.
anObject methodSourceAt: #printOnTranscript.
anObject removeMethod: #printOnTranscript.
```

Téhož efektu lze docílit pomocí inspektoru. Stačí vybrat –methods– a sledovat instrukce, případně vybrat konkrétní metodu a vyvolat kontextovou nabídku.

Operace nad delegáty

```
anObject addDelegate: 'parent' withValue: (PrototypeObject new).
anObject removeDelegate: 'parent'.
```

Téhož efektu lze docílit pomocí inspektoru – stačí kliknout na –delegates– a sledovat instrukce, případně kliknout na konkrétního delegáta a vyvolat kontextovou nabídku.

Zaslání zprávy

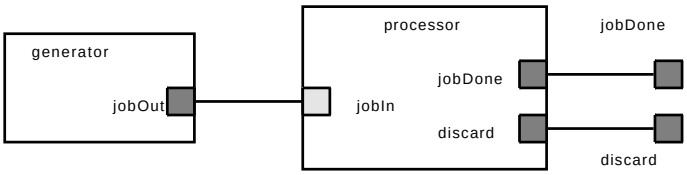
```
anObject printOnTranscript
```

S prototypovými objekty lze komunikovat stejně jako s ostatními objekty Smallalku. Prototypové objekty a ostatní objekty Smalltalku jsou z vnějšího pohledu zcela zaměnitelné a lze je libovolně kombinovat.

5.4 Modelování systémů prototypovými objekty

Vyjasněme nyní důvody, proč je k modelování vyvíjejících se systémů vhodné použít prototypové objekty. Třídní přístup k modelování je omezující tím, že buď vyžaduje znát všechny třídy předem, v době kompilace (v případě staticky kompilovaných jazyků), nebo vyžaduje tvořit (a odstraňovat) třídy za běhu (v případě dynamických jazyků). První možnost skutečně omezuje modelovací sílu jazyka – dynamicky lze měnit jen strukturu modelu, složeného z předem známých komponent. Druhá možnost je zbytečně komplikovaná, i když ji v principu dynamické jazyky připouštějí. Důvodem je principiální statičnost systému tříd. Jakékoli dynamické modifikace toho v principu statického systému jsou sice možné, ale jejich realizace je komplikovaná a ne každý jazyk ji umožňuje. Je totiž třeba při jakémkoli dynamickém zásahu zachovat konzistenci systému. Např. Smalltalk umožňuje modifikovat definici třídy i v případě, že existují její instance (instance jsou vyměněny za nové se zachováním původní identity), nebývá to však běžným zvykem ve většině dynamických jazyků.

Naproti tomu, jakákoliv dynamická manipulace s prototypovými objekty je triviální, jak jsme viděli v předchozí podkapitole. Omezuje se na klonování a editaci objektů. Sdílené chování, kvůli kterému dávají třídy smysl, je jednoduše realizovatelné prostřednictvím delegace, kterou lze také dynamicky měnit triviálním způsobem. Kromě toho, manipulace s prototypovými objekty připomíná manipulaci s libovolnými objekty, blízkými člověku. Např. interaktivní (ale i skriptovaná) práce s jakýmikoli dokumenty také spočívá v kopírování a editaci. Chceme-li umožnit interaktivní dynamické zásahy do simulace, je tento způsob manipulace s objekty velmi vhodný díky své jednoduchosti a přímočarosti.



Obrázek 5.2: Generátor a procesor.

Příklad Nyní ukážeme příklad modelu, vytořeného inkrementálně z prototypových objektů. Následující sekvence výrazů je program (skript), který postupně vytvoří model generátrou a procesoru. Tento skript může být proveden naráz, nebo mohou být jednotlivé výrazy vyhodnocovány postupně, krok za krokem v interaktivním prostředí Smalltalku (ve workspace). Nejprve vytvoříme generátor úloh.

```
| system generator processor jobPrototype |
jobPrototype ← PrototypeObject new.
jobPrototype addSlots: { 'n' -> 0.  'size' -> 0.  'name' -> 'aJob'. }.
jobPrototype addMethod: 'setSizeBetween: sl and: sh
                self size: (sl to: sh) atRandom'.

generator ← AtomicDEVSPrototype new.
generator addSlots: {
  'jobPrototype' -> jobPrototype.
  'ia' -> 2. "interval min"   'ib' -> 7. "interval max"
  'sa' -> 5. "job size min"   'sb' -> 10. "job size max"
  'first' -> true.    'n' -> 0.    "number of jobs generated"}.
generator intTransition: 'self first: false'.
generator outputFnc:  '
  self n: self n +1.
  self
    poke:
      ((self jobPrototype setSizeBetween: self sa and: self sb) clone
        n: self n;
        yourself)
    to: #out'.
generator timeAdvance: '
  ↑ self first
      ifTrue: [ 0 ]
      ifFalse: [ (self ia to: self ib) atRandom ]'.
generator addOutputPorts: {#out}.
```

Následující sekvence výrazů vytvoří model procesoru.

```
processor ← AtomicDEVSPrototype new.
processor addSlots: {
  'queue' -> OrderedCollection new.
  'queueSize' -> 5.
  'processorStatus' -> #idle.
  'currentJob' -> nil.
  'timeSpent' -> 0 }.
processor addInputPorts: {#in}.
processor addOutputPorts: {#out. #discard}.
processor intTransition: '
  self processorStatus caseOf: {
  [ #busy ] -> [
    self queue size > 0
      ifTrue: [
        self currentJob: (self queue removeFirst) ]
      ifFalse: [
        self processorStatus: #idle.
        self currentJob: nil ].
    self timeSpent: 0 ].
```

```
    [ #discard ] -> [
      self queue removeFirst.
      self queue size <= self queueSize ifTrue: [
        self processorStatus: #busy ]].
    [ #idle ] -> [ "nothing" ] } '.
  processor extTransition: '
    self queue add: (self peekFrom: #in).
      self processorStatus caseOf: {
    [ #idle ] -> [
      self processorStatus: #busy.
      self currentJob: (self queue removeFirst) ].
    [ #busy ] -> [
      self timeSpent: self timeSpent + self elapsed.
      self queue size > self queueSize
        ifTrue: [ self processorStatus: #discard ]].
    [ #discard ] -> [ "nothing" ] } '.
  processor outputFnc:  '
    self processorStatus caseOf: {
    [ #busy ] -> [ self poke: self currentJob to: #out ].
    [ #discard ] -> [ self poke: (self queue last) to: #discard ].
    [ #idle ] -> [ "nothing" ] } '.
  processor timeAdvance: '
    self processorStatus caseOf: {
    [ #busy ]   -> [ ↑ self currentJob size - self timeSpent ].
    [ #discard ]  -> [ ↑ 0 ].
    [ #idle ]     -> [ ↑ Float infinity ] } '.
```

Komponenty propojíme do jednohoho celku vyhodnocením následující sekvence výrazů.

```
system ← CoupledDEVSPrototype new.
system addOutputPorts: {
  #out.
  #discard }.
system addComponents: {
  #generator -> generator.
  #processor -> processor }.
system addCouplings: {
  #(generator out)  -> #(processor in).
  #(processor out)  -> #(self out).
  #(processor discard) -> #(self discard) }.
```

Takto vytvořený model může být simulován vyhodnocením výrazu

```
system getSimulator simulate: 500.
```

Výše uvedený kód je skript, který zasíláním zpráv vhodným objektům inkrementálně vytvoří model. Kódy metod jsou předány jako parametry odpovídajících zpráv a odpovídající objekty si je v reakci na tyto zprávy automaticky zakompilují do svých struktur.

Pro jakoukoliv manipulaci s modelem je podstatný model sám, nikoliv skript, který ho vytvořil. Jakoukoliv potřebnou informaci je možné získat komunikací s objekty modelu. Např. kód metody, kterou jsme dříve zakompilovali do objektu, lze zíkat takto:

```
anObject methodSourceAt: aMethodName.
```

Pro takto inkrementálně vytvořený programový systém není způsob jeho vytvoření podstatný, takže skript, který ho vytvořil, nechápeme jako zdrojový kód v pravém slova smyslu. Takový skript lze kdykoli pro existující systém automaticky vygenerovat.

Skript, který zrekonstruuje objekt (včetně včech z něho odkazovaných objektů) může být získán vyhodnocením výrazu

```
script ← anObject storeString.
```

Jinou možností je získat definici objektu v XML s využitím knihovny SIXX:

```
sixxString ← anObject sixxString.
```

Rekonstrukce objektu se pak provede vyhodnocením některého z výrazů:

```
anObject ← Object readFrom: StoreString.
anObject ← Object readSixxFrom: sixxString.
```

Takto lze manipulovat se všemi objekty, včetně modelů a běžících simulací.

5.5 Sdílené chování, znovupoužitelnost

Ve výše uvedném modelu můžeme jednoduše přidat další procesory naklonováním extujícího procesoru. Ale pozdější modifikace procesoru se pak dotknou jen jednoho exempláře. Chceme-li zajistit, aby všechny procesory sdílely jednu definici, je třeba ji přesunout z procesoru do separátního objektu, který se nazývá trait. Samotný procesor je pak popsán modelem, definujícím jen instančně specifické skutečnosti a delegujícím zpracování všech ostatních zpráv na příslušný trait. Klony takto definovaného procesoru pak sdílí jednu definici a její případná modifikace pak ovlivní chování všech klonů. Trait procesoru může být vytvořen takto:

```
processorTrait ← AtomicDEVSTrait new.
processorTrait addMethod: 'intTransition ........'.
processorTrait addMethod: 'extTransition ........'.
processorTrait addMethod: 'outputFnc ........'.
processorTrait addMethod: 'timeAdvance ........'.
```

Metody *intTransition*, *extTransition*, *outputFnc*, *timeAdvance* traitu jsou definovány stejně jako u procesor ve výše uvedeném modelu. Společně s trait objektem je třeba specifikovat prototyp procesoru:

```
processorPrototype ← AtomicDEVSPrototype new.
processorPrototype addSlots: {
        'queue' -> OrderedCollection new.
        'queueSize' -> 5.
        'processorStatus' -> #idle.
        'currentJob' -> nil.
        'timeSpent' -> 0 }.
processorPrototype addInputPorts: {#in}.
processorPrototype addOutputPorts: {#out. #discard}.
processorPrototype addDelegate: 'defaultTrait' withValue: processorTrait.
```

Specifikováním delegace na trait zajistíme sdílení definice chování všemi klony prototypu procesoru. Nyní můžeme specifikovat systém s několika procesory stejného typu:

```
system ← CoupledDEVSPrototype new.
system addOutputPorts: { #out. #discard }.
system addComponents: { #generator -> generator }.
previousPort ← {#generator. #out}.
1 to: 3 do: [ :i |
  newProcessor ← processorPrototype copy.
  newProcessorName ← (#processor, i printString) asSymbol.
  system addComponents: { newProcessorName -> newProcessor }.
  system addCouplings: {
    previousPort -> {newProcessorName. #in}.
    {newProcessorName. #out} -> {#self. #out} }.
  previousPort ← {newProcessorName. #discard} ].
system addCouplings: {
  {newProcessorName. #discard} -> {#self. #discard} }.
```

Poznamenejme, že trait objekty mohou také delegovat zpracování zpráv na další trait objekty. Tímto způsobem trait objekty hrají roli tříd a delegace modeluje dědičnost v klasickém přístupu k objektové orientaci. Je možná i násobná delegace i dynamicé změny této relace.

5.6 In(tro)spekce a reflektivita

Výše uvedený model, obsahující generátor a 3 procesory, byl vytvořen skriptem, který komponenty dynamicky propojil. Výsledné propojení lze zkontrolovat. Propojení získáme vyhodnocením výrazu

```
system couplings
```

nebo

```
system couplingStoreString.
```

Výsledkem je množina asociací dvojic <copmponentName portName>, případně kód, jehož vyhodnocením zíkáme příslušnou struktutu, která vypadá takto:

```
{
 {#generator. #out} -> {#processor1. #in}.
 {#processor1. #out} -> {#self. #out}.
 {#processor1. #discard} -> {#processor2. #in}.
 {#processor2. #out} -> {#self. #out}.
 {#processor2. #discard} -> {#processor3. #in}.
 {#processor3. #out} -> {#self. #out}.
 {#processor3. #discard} -> {#self. #discard}.
}
```

Tato struktura může být (po případných modifikacích) použita později jako argument zpráv

```
system removeCouplings: couplings.
system addCouplings: couplings.
```

Podobně můžeme získat další informace o modelu a jeho komponentách, jako jsou jména slotů, objekty, které jsou referencovány ze slotů, metody, delegáti, komponenty složeného

modelu apod. Také můžeme klonovat a editovat celý model nebo libovolnou jeho část, a to i v průběhu simulace. Můžeme přidávat a odstraňovat komponenty, sloty, delegáty, metody apod. Tyto operace jsou přístupné uživateli, resp. nadřazenému systému, ale také (reflektivně) modelu, kterého se tyto operace týkají.

5.7 Operační systém

Pod pojmem operační systém rozumíme veškeré programové prostředky pro podporu vytváření a běhu aplikačně specifického softwaru, včetně možnosti libovolné manipulace s ním. Jde o software, vytvářející potřebnou abstrakci nad tzv. holým počítačem a poskutující operace pro manipulaci s programy a běžícími procesy jak uživateli, tak těmto programům a procesům.

5.7.1 Smalltalk a SmallDEVS

Odmyslíme-li hostitelský operační systém, základní vrstvu operačního systému SmallDEVS tvoří Smalltalk. Ten zahrnuje virtuální stroj, kompilátor, víceprocesové prostředí, knihovnu tříd, vstupy a výstupy a interaktivní vývojové nástroje, z nichž pro SmallDEVS jsou podstatné nástroje Workspace a Inspector.[1] Workspace umožňuje editaci textů a interaktivní vyhodnocování fragmentů kódu. Výsledné objekty mohou být zkoumány v textové podobě nebo prostřednictvím inspektorů. Inspektor umožňuje zkoumat stav objektu, komunikovat s ním a editovat jeho aktuální stav.

Model, který byl prezentován vpředchozích sekcích, může byt vytvořen vyhodnocením příslušného kódu ve Workspace. Stejným způsobem lze model zkoumat i editovat, jak již bylo ukázáno. Lze též odstartovat simulaci a sledovat její průběh (log). Simulaci lze spustit jako proces na pozadí a asynchronně s ní komunikovat.

```
[ s ← system getSimulatorRT. s simulate: 500. ]
        forkAt: Processor userBackgroudPriority.
```

```
s ← system getSimulatorRT.
s stopTime: Float infinity.
s RTFactor: 1.
s start.
```

```
s stop.
```

Komunikací se simulátorem *s* můžeme získat simulovaný model a zkoumat stav jeho komponent. Následující příklad otevře inspektor na obsahem slotu 'queue' prvního procesoru:

```
(s rootDEVS componentNamed: #processor1) model queue inspect
```

Model můžeme v průběhu simulace editovat způsobem, který byl uveden v předchozích sekcích. Můžeme též celý model v libovolném okamžiku klonovat:

```
savedSystem ← system copy.
```

[1] Ostatní nástroje Smalltalku, jako je Browser, Debugger apod., jsou ovšem také k dispozici.

5.7.2 Perzistence

V předchozích sekcích byla perzistence modelu jednoduše řešena přiřazením do proměnných v rámci Workspace. Systematičtější přístup je založen na využití hierarchického uložiště *MyRepository*:

```
MyRepository at: '/Simulations/GeneratorAnd3Processors' put: system.
```

```
aComponent ← MyRepository at: '/Simulations/GeneratorAnd3Processors'.
```

```
aComponent ← (MyRepository componentNamed: 'Simulations')
                    componentNamed: 'GeneratorAnd3Processors'.
```

```
aComponent addComponents: { name1 -> object1. name2 -> object2 }.
```

Toto uložiště (včetně všech v něm aktuálně běžících simulací) i libovolnou jeho část lze v libovolném okamžiku klonovat, případně v textové formě uložit na externí médium nebo přenést do jiného Smalltalku.

5.7.3 Vizuální nástroje pro manipulaci s modely a simulacemi

Workspace a Inspector, s kterými jsme vystačili v předchozích sekcích, jsou standardními nástroji, které jsou součástí operačního systému Smalltalku. Prostředí SmallDEVS ale navíc poskytuje specializované vizuální nástroje pro sugestivnější a přehlednější manipulaci s modely a simulacemi.

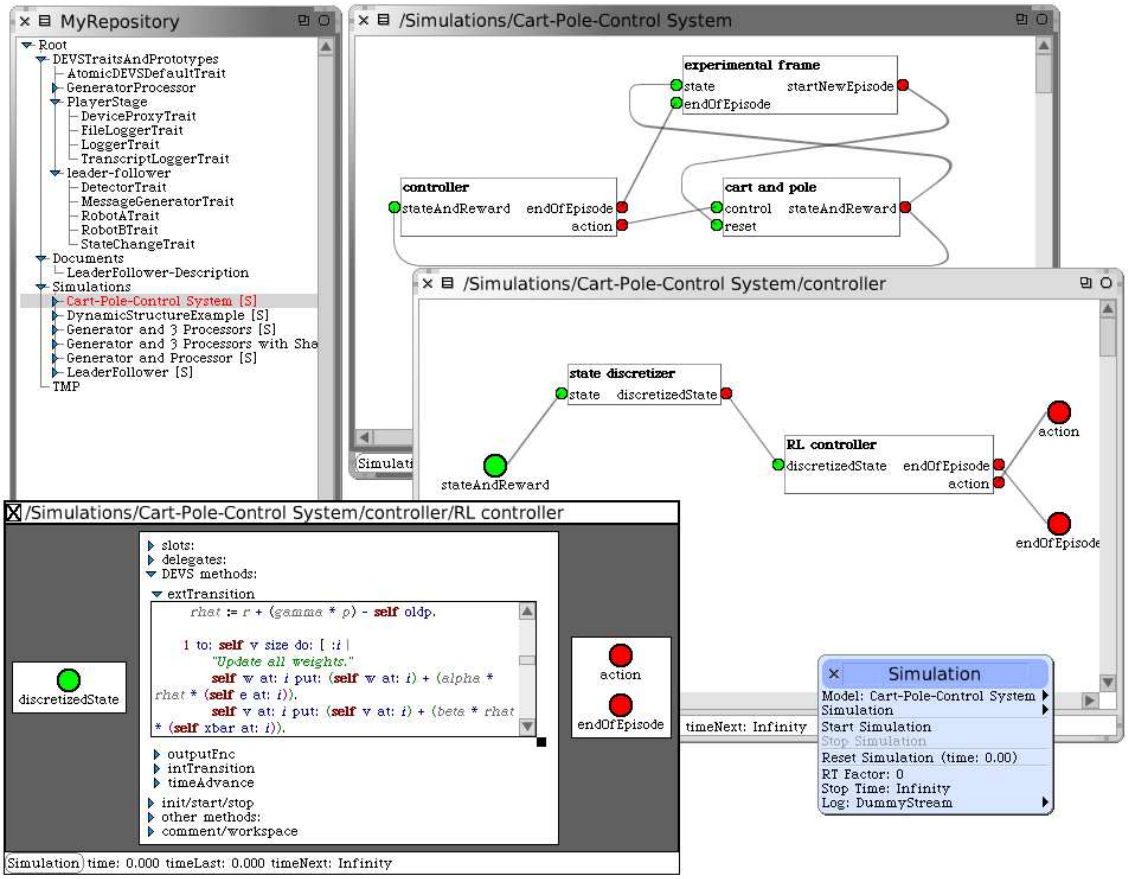
Tyto nástroje byly inspirovány nástroji pro manipulaci s prototypovými objekty v systému Self [US87]. Self je jazyk a systém, založený na prototypových objektech. Základním vizuálním nástrojem programátora je tak zvaný Outliner. Jde o nástroj, umožňující zkoumat a modifikovat strukturu objektu. Srovnáme-li tento nástroj se nástroji Smalltalku, Outliner spojuje funkčnost Browseru a Inspectoru, protože v systému Self pojem třída a instance splývají v jeden koncept – objekt obsahuje data i metody. V prostředí systému Self jsou vizualizovatelné vztahy (reference) mezi objekty a je možné s obsahy slotů (referencemi na objekty) manipulovat kopírováním a vkládáním (copy-paste), resp. přesouváním (drag-and-drop). Refaktoring objektů (přemist'ování slotů mezi různými objekty) je tak možné provádět intuitivním, konkrétním a bezprostředním způsobem.

Velmi podobně se manipuluje s dokumenty a složkami pomocí vizuálních interaktivních nástrojů soudobých operačních systémů – ty umožňují vytvářet, vyjímat, kopírovat, vkládat, přejmenovávat a otvírat (new, cut, copy, paste, rename, open) dokumenty a složky. V našem případě takto pracujeme s objeky, umístěnými v hierarchické struktuře *MyRepository*. Rozdíl oproti manipulaci se soubory je ten, že jde o živé objekty, případně běžící simulace (nikoliv pojmenované řetězce zvané soubory) a vše se odehrává v operační paměti, nikoliv na externím médiu (případná externalizace objektů v textové podobě je ovšem možná, jak bylo uvedeno výše).

Podstatnou vlastností vizuálních nástrojů SmallDEVS je jejich transparentnost – umožňují zkoumat a manipulovat s objekty bez ohledu na to, zda byly interaktivně vytvořeny těmito nástroji nebo zda vznikly jako výsledek běhu programů. Této transparentnosti je dosaženo tím, že se nepracuje se zdrojovými texty objektů, ale pracuje se s objekty přímo (zdrojové texty se neuchovávají, protože jsou zbytečné, jak již bylo uvedeno).

Každý objekt je možné zkoumat a editovat. Podle typu objektu se otevře odpovídající nástroj. Ke zkoumání a editaci obecných objektů slouží PrototypeObjectInspector (obr. 5.1), ke zkoumání a modifikaci MyRepository slouží MyRepositoryBrowser (obr. 5.3 vlevo nahoře), ke zkoumání atomických modelů slouží AtomicModelExplorer (obr. 5.3 vlevo dole), ke zkoumání a editaci složených modelů slouží CoupledModelExplorer (obr. 5.3 vpravo nahoře), k ovládání simulací slouží SimulationControlPanel (obr. 5.3 vpravo dole). Odpovídající nástroj se otevře automaticky v důsledku vyhodnocení výrazu

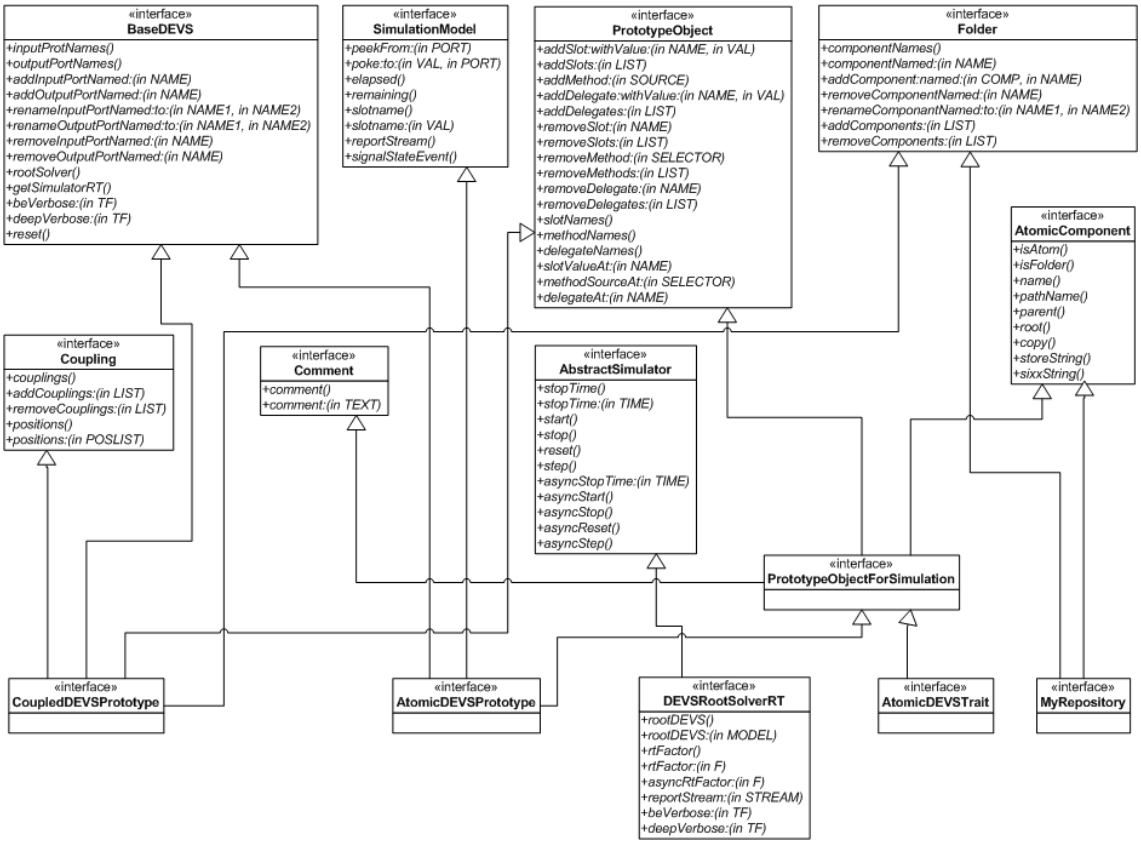
```
anObject open.
```



Obrázek 5.3: SmallDEVS – vizuální nástroje pro manipulaci s modely a simulacemi

5.8 Aplikační rozhraní jádra SmallDEVS

Obrázek 5.4 ukazuje kompozici protokolů *AtomicDEVSPrototype, CoupledDEVSPrototype, PrototypeObjectForSimulation, AtomicDEVSTrait, MyRepository* a *DEVSRootSolverRT*. Jde o kompletní aplikační rozhraní jádra SmallDEVS, dostupné jak vizuálním vývojovým nástrojům, tak programům a skriptům pro tvorbu modelů a manipulaci s nimi.



Obrázek 5.4: Aplikační rozhraní jádra SmallDEVS.

5.9 Poznámka k implementaci

SmallDEVS je implementován ve Smalltalku za použití rozšíření Prototypes. Může být chápán jako vzor, podle kterého lze prezentovaný koncept realizovat i v jiném prostředí. V případě, že by šlo o statický jazyk, jako je Java, C/C++ apod., nevyhneme se použití vnořeného interpretu vhodného dynamického jazyka s inkrementálním kompilátorem.

Kapitola 6

Specifikace systémů s diskrétními událostmi objektově orientovanými Petriho sítěmi

Přímé použitítí formalismu DEVS pro specifikaci systému nemusí být vždy výhodné. Problematické jsou zejména případy, kde se struktura systémů mění příliš rychle a komplikovaně. Rekonfigurovat strukturu složeného modelu by je pak bylo příliš obtížné. Lepší možností je v takových případech použít objektové paradigma s adresovatelnými objekty a zasíláním zpráv. Jako optimální se jeví kombinace, ve které je DEVS použit pro strukturování systému do víceméně statických komponent, zatímco dynamicky vznikající a zanikající objekty jsou použity uvnitř atomických komponent DEVS.

V následujícím textu je popsána možnost definovat systém s diskrétními událostni objektově orientovaými Petriho sítěmi (OOPN). Je stručně představen formalismus OOPN a jeho simulátor, včetně zapouzdření do atomické komponenty DEVS. Přitom je zohledněna možnost dynamických zásahů do běžící simulace v souladu s konceptem otevřeného systému, definovaného v kapitole 4.

6.1 Petriho sítě a objekty

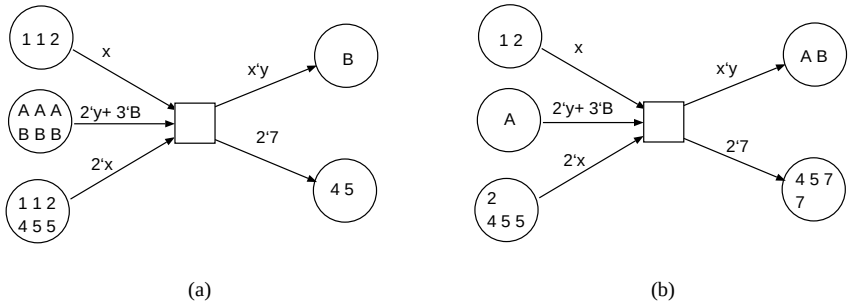
Základní koncept Petriho sítí definoval v šedesátých letech C. A. Petri. Jde o matematický stroj se sugestivní vizuální reprezentací, který srozumitelně modeluje synchronizaci a komunkaci procesů v paralelních a distribuovaných systémech. Existuje celá řada variant Petriho sítí. Na tomto místě se soustředíme především na vysokoúrovňové Petriho sítě, konkrétně pak na Objektově orientované Petriho sítě (OOPN) [Jan95, Jan98, MVT97, MVT02, JK07]. Ostatní běžně používané typy Petriho sítí jsou subformalismy objektově orientových Petriho sítí.[1] Konkrétní implementací OOPN je interpret jazyka PNtalk [PNt06, JK03, JK05a, JK05b, MVR+06].

[1] Objektově orentované Petriho sítě [Jan98] byly navrženy s cílem pokrýt schopnosti ostatních typů Petriho sítí a umožnit použít Petriho sítě způsobem, který je srovnatelný s programováním v univerzálním objektově orientovaném jazyce.

6.1.1 Princip vysokoúrovňové Petriho sítě

Petriho síť je specifikována formou bipartitního orientovaného grafu, obsahující dva typy uzlů – místa a přechody – propojené hranami. Místa, přechody i hrany mohou být opatřeny anotacemi, specifikovanými vhodným inskripčním jazykem. Místa mohou obsahovat multimnožiny značek (tokens). Stav systému je modelován rozložením značek v místech. Ke změnám stavu dochází v důsledku provádění přechodů. Přechod má definována vstupní a výstupní místa (ta jsou s přechodem propojena vstupními a výstupními hranami). Přechod je proveditelný, jsou-li v jeho vstupních místech značky, které jsou specifikovánu na vstupních hranách. Provedením přechodu se tyto značky ze vstupních míst odstraní a do výstupích míst se vygenerují značky pecifikované výstupními hranami.

Ve vysokoúrovňové Petriho sítí mohou značky reprezentovat libovolná data, která mohou být prováděním přechodů libovolně zpracovávána. Výrazy na hranách specifikují multimnožiny značek. Mohou obsahovat proměnné. Ty musí být v rámci provedejí přechodu navázány na konkrétní hodnoty. Přechod může navíc obsahovat strážní podmínku, jejíž splnění je vyžadováno pro provedení přechodu. Kromě toho může obsahovat akci, která provede libovolný výpočet nad vstupními daty. Výsledky pak mohou být uloženy do výstupních míst. Na obr. 6.1 je příklad přechodu ve vysokoúrovňové Petriho síti (bez strážní podmínky a akce). Obsahuje proměnné x a y. Je proveditelný pro dvě různá navázání proměnných $\{x = 1, y = A\}$ a $\{x = 1, y = B\}$. Jeho provedení je realizováno pro $\{x = 1, y = A\}$.



Obrázek 6.1: Provedení přechodu ve vysokoúrovňové Petriho síti pro navázání proměnných $\{x = 1, y = A\}$. (a) – stav před provedením, (b) – stav po provedení přechodu.

6.1.2 Paralelní objektově orientovaný systém

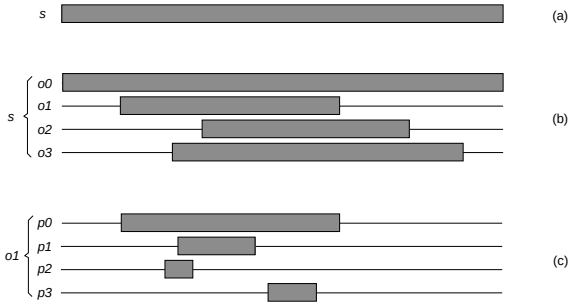
OOPN podle [Jan98] specifikuje paralelní objektově orientovaný systém. Takový systém lze popsat jako abstraktní matematický stroj, který má strukturu a chování. Struktura objektově orientovaného systému je určena množinou tříd, specifikujících reprezentaci instancí (instanční proměnné) a množinu metod, včetně tzv. *implicitní metody*, popisující implicitní aktivitu objektu. Dynamika systému spočívá v postupných změnách stavu (přičemž jeden stav je definován jako výchozí). Stav objektově orientovaného systému je dán množinou momentálně existujících objektů. Každý z nich se nachází ve stavu, který je dán hodnotami instančních proměnných a stavy všech momentálně existujících (rozpracovaných) metod. Evoluci (postupné změny stavu) tohoto systému lze pak definovat pravidly,

specifikujícími podmínky výskytu jistých událostí a změny stavu při výskytu těchto událostí. V paralelním objektově orientovaném systému existují čtyři typy událostí:

1. Událost typu A – změna stavu objektu bez interakce s ostatními objekty.
2. Událost typu N – vytvoření nového objektu při zaslání zprávy `new` příslušné třídě.
3. Událost typu F – vytvoření nové instance metody při zaslání odpovídající zprávy.
4. Událost typu J – ukončení existence instance metody s předáním výsledku volajícímu objektu.

Implicitní součástí každé události je *garbage collecting*, tedy odstranění nedostupných objektů.

Události mění stav systému. Objekty v rámci systému mohou dynamicky vznikat a zanikat, stejně tak i instance metod. Obrázek 6.2 ukazuje dekompozici procesu systému do objektů, skládajících se z procesů metod (v časovém diagramu jsou množiny okamžiků, kdy proces existuje, vyznačeny šedými oblastmi).



Obrázek 6.2: Proces systému (a) a jeho dekompozice do objektů (b). Objekt $o0$ je hlavní (prvotní) objekt, existující po dobu existence systému. Objekt $o1$ je dále dekomponován do procesů metod (c), $p1$ je proces implicitní metody, existující po dobu existence objektu.

6.1.3 Modelování objektů Petriho sítěmi

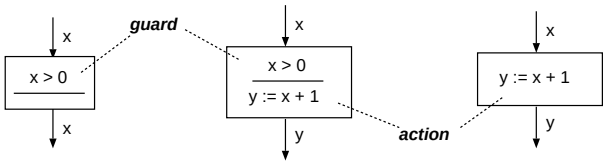
Objektově orientovaný paralelní systém lze podle [Jan98] popsat Petriho sítěmi takto:

1. Každá metoda je definována samostatnou vysokoúrovňovou Petriho sítí, kterou nazveme *síť metody*. Síť definující vlastní aktivitu objektu (jde o tzv. implicitní metodu, která je implicitně vyvolána vznikem objektu) nazveme *síť objektu*. *Třída* je specifikována jednou sítí objektu a množinou sítí metod.
2. Sítě metod sdílí místa sítě objektu (přechody sítě metody mají přístup k místům sítě objektu). Tato místa hrají podobnou úlohu, jako instanční proměnné v běžných objektově orientovaných programovacích jazycích.
3. Značky (tokens) v sítích reprezentují objekty (jsou to reference na objekty).
4. Třídy lze definovat jako podtřídy existujících tříd, tedy děděním. Metody jsou děděny klasickým způsobem [GR89], dědičnost reprezentace objektu spočívá v dědění uzlů sítě objektu (hrany sítě jsou považovány za součást přechodů).

5. Metody jsou invokovány zasíláním zpráv. Zaslání zprávy může být specifikováno anotací přechodu, patřícího některé síti metody nebo síti objektu.

6. Některé přechody sítě objektu jsou prohlášeny za speciální metody (tzv. *synchronní porty*), určené pro pro atomickou synchronní komunikaci, realizovanou zasíláním zpráv ze stráží přechodů.

6.1.4 Interakce objektů – zasílání zpráv



Obrázek 6.3: Anotace přechodu – stráž a akce.

V objektově orientovaném systému se veškeré výpočty realizují zasíláním zpráv. V OOPN je zasílání zpráv možné specifikovat v rámci akcí a stráží přechodů (viz obr. 6.3). Adresátem zprávy může být buď primitivní objekt (např. číslo, řetězec apod.), nebo objekt, definovaný Petriho sítěmi. Adresát, a tedy odpovídající metoda, se zjistí v rámci zjišťování proveditelnosti přechodu. Může jít o metodu primitivního objektu, nebo o metodu, definovanou Petriho sítí, nebo o synchronní port.

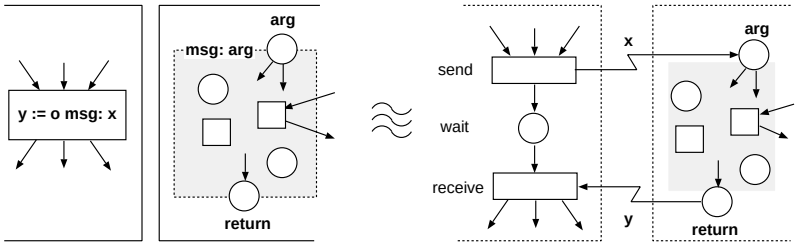
Zaslání zprávy, specifikované ve stráži přechodu, je interpretováno jako *atomická synchronní interakce*, která spočívá v *invokaci synchronního portu* (viz dále). Zaslání zprávy, specifikované v akci přechodu, je interpretováno jako *invokace sítě metody*.

6.1.5 Invokace metod objektů zasíláním zpráv

Sítě metod, specifikované jako součást tříd, jsou určeny k dynamické instanciaci v reakci na příchozí zprávy. Přijetí zprávy objektem odpovídá události typy F (fork, vytvoření nového procesu metody). Výsledkem je vytvoření nové instance (kopie) odpovídající sítě metody (a případné umístění parametrů do parametrových míst). Tato instance sítě existuje, dokud nedojde k umístění značky do místa *return*. Pak může dojít k události typu J (join, ukončení procesu metody). Tím dojde k ukončení existence instance metody, předání výsledku a provedení výstupní části přechodu, který zaslaním zprávy metodu invokoval. Na obrázku je znázorněn princip invokace metody. Poznamenejme, že přechody sítě metody mohou přistupovat k místům sítě objektu a ovlivňovat tak stav objektu.

6.1.6 Atomická synchronní interakce objektů

Zaslání zprávy neprimitivnímu objektu ve stráži přechodu je interpretováno jako *atomická synchronní komunikace*. Jde o komunikační koncept, umožňující podmínit proveditelnost přechodu současnou proveditelností jiného (volaného) přechodu, nazvaného *synchronní port*. Jsou-li oba přechody proveditelné, mohou být provedeny, a to synchronně, jako jedna atomická operace. Synchronní port lze považovat za speciální druh metody, kterou lze volat ze stráže jiného přechodu *zasláním zprávy*. Tento druh komunikace je vhodný pro



Obrázek 6.4: Invokace metody.

modelování synchronizace aktivit objektů ve víceúrovňových modelech [**?**]. Princip atomické synchronní interakce, respektující polymorfismus a navázání proměnných přechodu a parametrů synchronního portu je znázorněn na obrázku 6.5.

Obrázek 6.5: Atomická synchronní interakce objektů.

Synchronní port, který jen testuje, ale nemodifikuje obsah míst odpovídající sítě objektu, nazýváme *predikát*. V [Maz08] byl pro potřeby čitelnějšího modelování do OOPN zaveden také *negativní predikát*. Analogicky k *not* v Prologu, negativní predikát selže, pokud stejně specifikovaný pozitivní predikát uspěje a naopak. Pomocí negativního predikátu lze testovat nepřítomnost značky v místě. Jde o zobecnění konceptu inhibiční hrany v Petriho sítích.

6.2 Formalismus OOPN

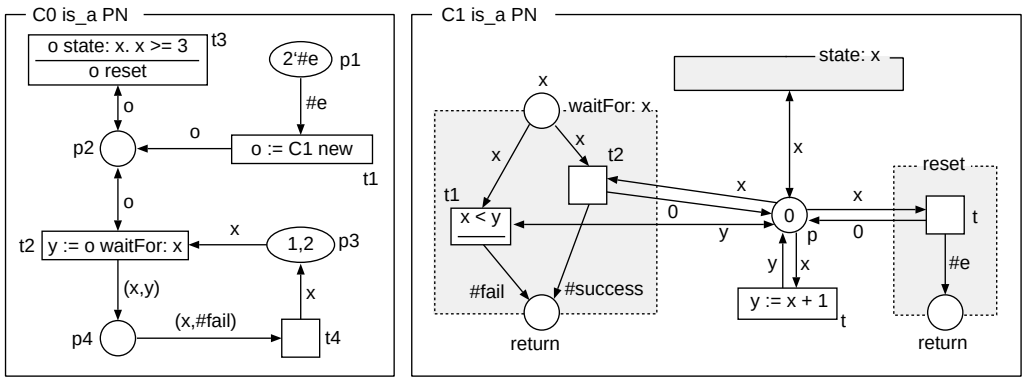
Formalismus OOPN je podrobně popsán a formálně definován v [Jan98]. Tamtéž je popsána i jeho konkrétní implementace, jazyk a systém PNtalk. Vzhledem k poměrné rozsáhlosti originální definice jsou dále uvedeny jen nejdůležitější pojmy a jejich souvislosti, které jsou nutné pro pochopení některých aplikací v navazujících kapitolách. Zaměříme se přitom na strukturu OOPN, na reprezentaci stavu a na dynamiku OOPN.

6.2.1 Struktura OOPN

Objektově orientovaná Petriho síť [Jan98] je definována jako trojice $OOPN = (\Sigma, c_0, oid_0)$, kde Σ je systém tříd, c_0 je počáteční třída a oid_0 je *jméno prvotního objektu*, který je instancí c_0. *Systém tříd* zahrnuje *množinu tříd*, hierarchicky uspořádanou podle *relace dědičnosti*. Každé třídě přísluší množina jmen instancí (doména) a je pro ni definována struktura instancí. *Specifikace struktury instancí třídy* zahrnuje síť objektu, množinu sítí metod, množinu synchronních portů, množinu selektorů zpráv a bijekci množiny selektorů zpráv na množinu, obsahující metody a synchronní porty. Systém tříd musí splňovat jisté

podmínky umožňující dědičnost a zaručující, že pro každou instanci sítě lze určit odpovídající síť a třídu, ve které byla tato síť definována. Anotace Petriho sítí jsou specifikovány inskripčním jazykem umožňujícím pracovat také s *primitivními objekty*, jako jsou čísla, řetězce atd.

Příklad Příklad OOPN je na obrázku 6.6. Obsahuje třídy `C0` a `C1`. Třída `C0` obsahuje jen síť objektu. Třída `C1` obsahuje síť objektu, obsahující místo `p` a přechod `t`, a dále obsahuje synchronní port `state:` a metody `waitFor:` a `reset`. Poznamenejme, že tento příklad nemodeluje nic smysluplného, pouze demonstruje formalismus OOPN.



Obrázek 6.6: Příklad OOPN.

6.2.2 Reprezentace stavu OOPN

Stav OOPN je definován jako *systém objektů*. *Objekt* je definován jako *systém instancí sítí*, obsahující v daném stavu právě jednu instanci sítě objektu a množinu instancí sítí metod, které jsou v daném stavu rozpracovány. Každá *instance sítě* je dvojice (id, m), kde id je jméno instance sítě a m je *značení* této sítě (obsahuje značení míst i přechodů). Značení m každému místu v instanci sítě přiřazuje multimnožinu značek, identifikujících objekty (primitivní, neprimitivní i třídy) a každému přechodu množinu invokací (rozpracovaných metod). Systém objektů je *uzavřený*, tj. zaručuje, že značení všech instancí sítí ve všech objektech referencují pouze instance sítí obsažené v systému objektů. Také je zaručeno, že neobsahuje objekty, které by nebyly tranzitivně dostupné z prvotního objektu.

Počáteční systém objektů objektově orientované Petriho sítě $OOPN = (\Sigma, c_0, id_0)$ obsahuje jediný objekt, identifikovaný jménem id_0, který je instancí počáteční třídy c_0. Značení jeho sítě objektu odpovídá počátečnímu značení této sítě, specifikované ve třídě c_0.

Příklad Uvažujme OOPN specifikovanou třídami na obrázku 6.6. Počáteční stav této OOPN je definován stavem prvotního objektu (instance hlavní třídy). Obsahuej poráteční značení sítě objektu třídy C0. Objekt je identifikován tímtéž jménem jako instance jeho sítě objektu, a sice id_0. Počáteční stav označíme s_0 a můžeme ho specifikovat tabulkou 6.1.

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

Tabulka 6.1: Příklad systému objektů – počáteční stav s_0.

6.2.3 Dynamika OOPN

Dynamika OOPN je definována počátečním systémem objektů a pravidly specifikujícími proveditelnost a provádění přechodů v instancích sítí. Provedení přechodu je událost, formalizovaná jako čtveřice (e, id, t, b), kde e specifikuje typ události (A, N, F nebo J, viz 6.1.2, str. 59), jméno id identifikuje instanci sítě, ve které k události došlo, t identifikuje přechod jehož provedení událost způsobilo a b je navázání proměnných přechodu t. Výskyt události (e, id, t, b) ve stavu S způsobí změnu tohoto stavu na S', což označujeme jako *krok* a zapisujeme $S[e, id, t, b\rangle S'$.

Přechod může potenciálně být proveden čtyřmi způsoby, které odpovídají čtyřem typům událostí z 6.1.2. Přechod může být pro jisté navázání proměnných *A-proveden*, pokud zaslání zprávy vede na primitivní výpočet, může být *N-proveden*, pokud jde o zaslání zprávy `new` třídě, může být *F-proveden*, pokud adresátem zprávy je objekt, který má příslušnou metodu definovánu Petriho sítí, a může být *J-proveden*, pokud metoda invokovaná předchozím F-provedením tohoto přechodu právě skončila. A-provedení přechodu odpovídá provedení přechodu ve standardní Petriho síti, N-provedení přechodu má za následek vytvoření nového objektu, tj, invokaci příslušné sítě objektu, F-provedení má za následek invokaci příslušné sítě metody a uplatní se jen vstupní hrany přechodu, J-provedení má za následek odstranění příslušné instance sítě metody a uplatní se jen výstupní hrany přechodu. Vzhledem k tomu, že mezi F- a J-provedením si přechod musí pamatovat příslušnou invokaci, byl zaveden pojem *značení přechodů*. Je to funkce, přiřazující každému přechodu množinu invokací, tj. dvojic (id, b), kde id je identifikace (jméno) vytvořené instance sítě a b je navázání proměnných přechodu při jeho F-provedení.

Události tvaru (e, id, t, b) postupně modifikují stav systému. V jednom okamžiku (v daném stavu) může potenciálně nastat několik událostí (je zde potenciálně několik různě proveditelných přechodů), ze kterých se nedetrministicky vybere a uskuteční jen jedna.

Příklad Uvažujme OOPN specifikovanou třídami na obrázku 6.6 a počáteční stav z 6.2.2. Proveďme sekvenci kroků

$$
\begin{aligned}
&s_0\ [(N, id_0, C0.t1, ())\rangle \\
&s_1\ [(F, id_0, C0.t2, (x = 1, o = id_1))\rangle \\
&s_2\ [(F, id_0, C0.t2, (x = 2, o = id_1))\rangle \\
&s_3\ [(A, id_1, C1.t, (x = 0))\rangle \\
&s_4\ [(A, id_2, C1.waitFor : .t2, (x = 1))\rangle\ s_5.
\end{aligned}
$$

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

Tabulka 6.2: Příklad systému objektů – stav s_5.

Výsledný stav s_5 je specifikován tabulkou 6.2. Stav s_5 obsahuje dva objekty identifikované jmény $id0$ a $id1$. Značení některých míst a přechodů objektu $id0$ se změnilo. Nový objekt $id1$ obsahuje instance sítí identifikované jako $id1$, $id2$ a $id3$. Instance sítí s jmény $id2$ a $id3$ jsou invokacemi metody `waitFor:`, vytvořené provedením přechodu $t2$ v objektu $id1$.

6.3 Simulátor OOPN

Simulace OOPN spočívá v opakovaném provádění kroku simulace. V rámci simulačního kroku se provede testování proveditelnosti přechodů v jednotlivých instancích sítí, tvořících systém objektů. Z proveditelných přechodů je pak vybrán jeden a proveden.

Pro práci se simulačním časem je však třeba tento koncept upravit a zavést možnost specifikovat čekání. Stejně jako ostatní aktivity, tok času se v OOPN specifikuje zasíláním zpráv v rámci akcí přechodů. Čekání v simulačním čase je specifikováno akcí tvaru `self hold: t`. Provedení přechodu je v takovém případě pozdrženo (v simulačním čase) po definovanou dobu a teprve po jejím uplynutí je provedena výstupní část přechodu.

Plánování událostí je v simulátoru OOPN realizováno kalendářem. Průběh simulace je následující:

1. Dokud existuje proveditelný přechod, provede se. Při provádění přechodů se do kalendáře umisťují plánované události ve tvaru (čas, událost). Jde o události typu J (dokončení rozpracovaného přechodu), které se plánují v důsledku čekání, specifikovaného akcí `self hold: t`.

2. Není-li žádný přechod proveditelný, dojde k posuvu času podle nejbližší plánované události a událost (t.j. dokončení čekajícího přechodu) se provede. Pokračuje se bodem 1.

Simulátor OOPN připouští také zaslání zprávy `self hold: t` ve stráži přechodu. V takovém případě je stráž splněna (a přechod se může provést), pokud je přechod nepřetržitě proveditelný po specifikovanou dobu.

6.4 Zapouzdření OOPN do DEVS

Simulátor OOPN můžeme popsat, resp. zapouzdřit jako systém s diskrétními událostmi v souladu s definicí DEVS. To nám umožní specifikovat komponenty DEVS formalismem OOPN. Princip zapouzdření OOPN do DEVS tkví v tom, že dvě disjunktní podmnožiny množiny míst prohlásíme za vstupní, resp. výstupní, asynchronní porty a definujeme reprezentaci stavu zapouzdřujícího DEVS a funkce zapouzdřujícího DEVS takto:

- Množina stavů je množina systémů objektů, dosažitelných z počátečního systému objektů.
- Vstupní porty odpovídají vstupním asynchronním portům hlavního objektu. Výstupní porty odpovídají výstupním asynchronním portům hlavního objektu. Hodnoty vstupů a výstupů patří do množiny primitivních objektů OOPN.
- Interní přechodová funkce δ_{int} realizuje provedení jedné události v OOPN, plánované na aktuální čas. V případě konfliktu se vybere jedna událost pomocí preferenční funkce *select*, definované podobně, jako v případě složených modelů DEVS (vybere jeden z libovolné podmnožiny množiny všech módů přechodů[2]).
- Funkce posuvu času ta vrací dobu, za kterou má dojít k následující plánované události v OOPN, tj. buď 0, existuje-li alespoň jeden proveditelný přechod v aktuálním čase, nebo $(t_c - t)$, kde t_c je čas nejbližší plánované události v kalendáři a t je čas aktuální, je-li v kalenáři alespoň jedna položka. Je-li kalendář prázdný, ta vrací hodnotu ∞ (systém je pasivován).
- Externí přechodová funkce δ_{ext} modifikuje stav vstupních asynchronních portů hlavního objektu přidáním objektů ze vstupních portů DEVS.
- Výstupní funkce λ mapuje (z implementačního hlediska přesune) obsah výstupních asynchronních portů hlavního objektu do výstupních portů DEVS.

6.5 Dynamické aplikační rozhraní simulátoru

Simulátor OOPN může být realizován souladu s principy otevřené abstraktní architektury pro simulaci vyvíjejících se systémů, popsané v kapitole 4. Dynamické aplikační rozhraní simulátoru OOPN pak umožňuje manipulovat se strukturou tříd i jejich instancí, s metodami i s jejich invokacemi. Manipulace spočívá ve čtení struktury a stavu a v modifikaci struktury a stavu jednotlivých Petriho sítí. Je-li výsledkem dynamické manipulace konzistentní systém objektů dle definice [Jan98], simulace po tomto zásahu korektně pokračuje, v opačném případě je simulace ukončena v důsledku chyby.

Třídy OOPN jsou v PNtalku globálně dostupné jménem. Přístup k metodám uvnitř třídy je možný prostřednitvím jména (selektoru odpovídající zprávy). Síť objektu je dostupná pod jménem `#object`. Uzly Petriho sítí jsou také přístupné prostřednictvím svých jmen. Třídy, metody, místa a přechody rozumění zprávám, které odpovídají metaoperacím pro inspekci a editaci.

[2]Mód přechodu je varianta provedení, zahrnující navázání proměnných, pro které je přechod proveditelný.

Podobně lze přistupovat k reprezentaci aktuálního stavu. Jde o systém objektů, zapouzdřujících množiny procesů. Hlavní objekt je v PNtalku globálně přístupný pod jménem `#main`. Objekty jsou dostupné prostřednictvím referencí, které se vyskytují v podobě značek v místech Petriho sítí. Procesy metod jsou provázány relací invokace do hierarchické struktury, která má kořen v procesu sítě hlavního objektu. Tato hierarchie procesů je obvykle východiskem pro navigaci ve struktuře stavu OOPN. Jednotlivé složky stavu lze číst i editovat použitím adekvátních metaoperací.

Integrace OOPN do prostředí SmallDEVS Interpret OOPN (PNtalk) s dynamickým aplikačním rozhraním je intergrován do prostředí SmallDEVS (popsaného v kapitole 5), kde umožňuje specifikovat atomické komponenty DEVS pomocí OOPN. Přístup k prvkům OOPN je možný prostřednictvím prohlížeče MyRepository (viz kap. 5). Třídy OOPN jsou v MyRepository přístupné ve složce PNtalk. Na další úrovni hierarchie jsou metody. Jak třídy, tak metody lze editovat odpovídajícími vizuálními nástroji. Procesy OOPN jsou v MyRepository přístupné jako hierarchicky nižší položky pod komponentou DEVS, specfikovanou formalismem OOPN. Hierarchicky nižší úroveň trvoří procesy vytvořené procesy na úrovni vyšší. Stav procesu je k dispozici jako značení odpovídající Petriho sítě.

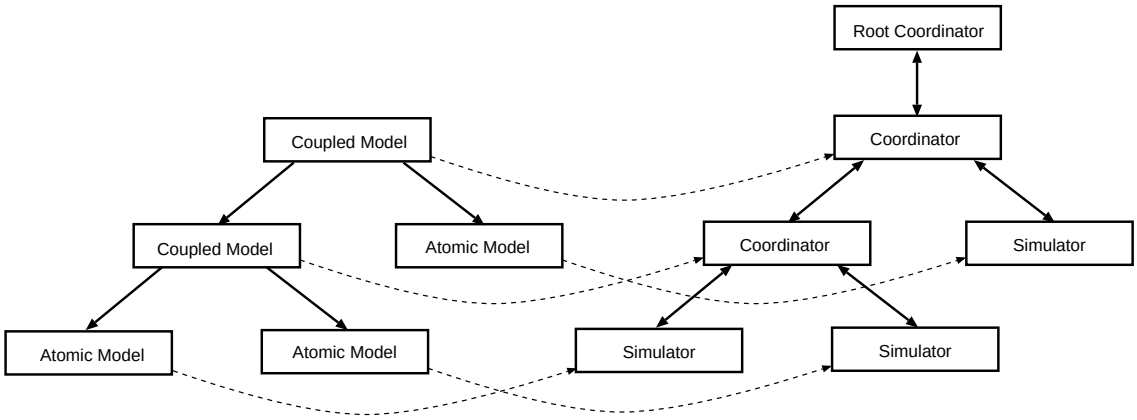
6.6 Shrnutí

OOPN lze použít pro specifikaci systému s diskrétními událostmi v souladu s definicí DEVS díky kompatibilnímu zapouzdření. V návaznosti na definici otevřené abstraktní architektury pro simulaci vyvíjejících se systémů, která byla uvedena za použití formalismu DEVS, lze definovat dynamické aplikační rozhraní i pro simulátor OOPN. Praktickou implementací je systém PNtalk/SmallDEVS, který umožňuje atomické komponenty DEVS specifikovat formalismem OOPN. Specifikace atomických komponent složeného modelu pomocí OOPN přináší vyšší úroveň popisu a sugestivní vizualizovatelnost. Aplikovatelnost OOPN a DEVS v simulačním modelování a návrhu systémů je demonstrována následujícími případovými studiemi.

Kapitola 7

Případová studie: Reflektivní simulátor DEVS

Užitečnou vlastností modelovacího formalismu je schopnost definovat vlastní sémantiku. Příkladem může být reflektivní simulátor DEVS. Je to simulátor DEVS specifikovaný jako DEVS.



Obrázek 7.1: Mapování mezi modely a simulátory

Připomeňme si nejprve princip hierarchické simulace DEVS. Každému modelu odpovídá příslušný simulátor a na vrcholu hierarchie je kořenový simulátor (vz obr. 7.1). Při realizaci reflektivního simulátoru lze vyjít ze specifikace abstraktního simulátoru DEVS. Tu je však třeba přizpůsobit charakteru cílového implementačního formalismu. Originální abstraktní simulátor DEVS podle [ZKP00] je specifikován s ohledem na implementaci v běžném prog. jazyce a proto obsahuje referenci na nadřazený simulátor a komunikuje přímým předáváním zpráv adresátovi (jde o synchronní komunikaci). Formalismus DEVS nepoužívá reference, komponenty o sobě vzájemně neví, jen nadřazená komponenta zná jejich propojení (komponenty jsou pojmenované a propojení je definováno s využitím jmen komponent a portů). Algoritmy a struktura simulátoru a koordinátoru proto musí být upraveny pro takto volně vázané hierarchicky organizované komponenty komunikující asynchroně předávanými zprávami. Lze přitom vyjít např. z [Van00] (podobně jsou specifikovány algoritmy simulátorů v kap. 2). V našem případě použijeme zprávy tvaru $(msg, from/to, t)$. Obsahují oproti originálu navíc jména odesilatele, resp. příjemce. Toto

adresování umožní připojit více komponent na jeden port, což zjednoduší propojování (přidávání dedikovaných portů pro každou komponentu kompozitu by model poněkud zkomplikovalo).

Simulátor atomického modelu Simulátor atomického modelu specifikujeme jako systém

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

Množina X obsahuje všechny zprávy tvaru (i, to, t), (x, to, t), $(*, to, t)$. Množina Y obsahuje všechny zprávy tvaru $(y, from, tn)$, $(done, form, tn)$, $(log, from, tn)$. Množina S je strukturována do proměnných t_{last}, t_{next}, y, (s, e). Funkce $\delta_{int}, \delta_{ext}, \lambda, ta$ jsou definovány tak, aby systém reagoval na vstupní události podle specifikace uvedené v kap. 2 v závislosti na specifikaci atomického modelu $M = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta)$, definovaného taktéž v kap. 2.

Simulátor složeného modelu Simulátor složeného modelu (koordinátor) specifikujeme jako systém

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

Množina X obsahuje všechny zprávy tvaru $(i, to, t), (x, to, t), (*, to, t), (y, from, tn), (done, from, tn)$. Množina Y obsahuje všechny zprávy tvaru $(y, from, tn)$, $(done, form, tn)$, (i, to, t), (x, to, t), $(*, to, t)$, $(log, from, tn)$. Množina S je strukturována do proměnných t_{last}, t_{next}, y, $eventList\ receivers$, $activeChildren$ (viz algoritmus simulátoru v kap. 2). Funkce $\delta_{int}, \delta_{ext}, \lambda, ta$ jsou definovány tak, aby systém reagoval na vstupní události podle specifikace uvedené v kap. 2. v závislosti na specifikaci složeného modelu $N = (X, Y, D, \{M_d\}, EIC, EOC, IC, select)$, definovaného taktéž v kap. 2.

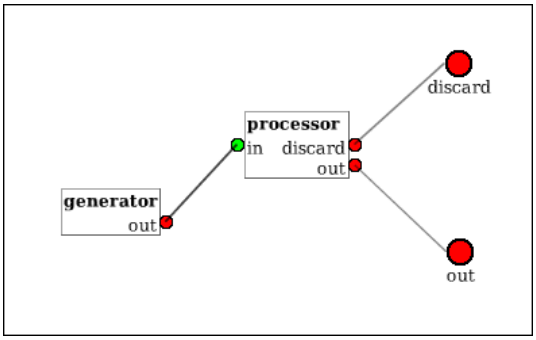
Kořenový simulátor Kořenový simulátor specifikujeme jako systém

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

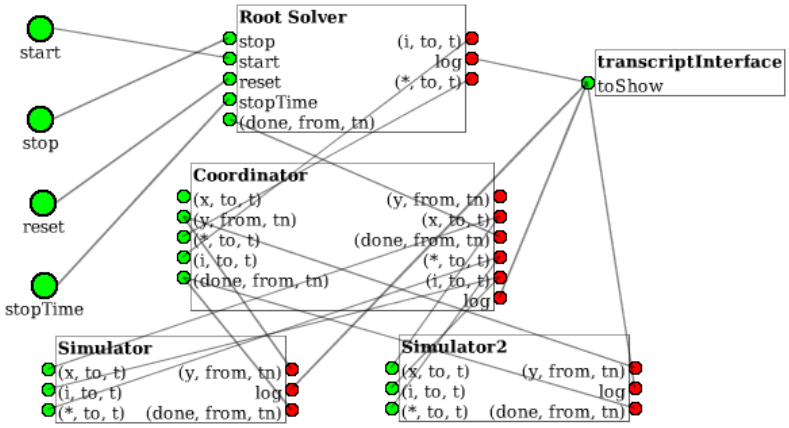
Množina X obsahuje všechny zprávy tvaru $(start), (stop), (reset), (stopTime, t)$, $(done, from, tn)$. Množina Y obsahuje všechny zprávy tvaru (i, to, t), $(*, to, t)$, $(log, from, tn)$. Množina S je strukturována do proměnných, reprezentujících stav simulace, aktuální čas t a čas následujícíc událostí t_{next}. Funkce $\delta_{int}, \delta_{ext}, \lambda, ta$ jsou definovány tak, aby systém reagoval na vstupní události provádením jednotlivých kroků simulace, případně změnou stavu (zastavení spuštění) a změnou paramertů simulace.

Reflektivní simulátor Mějme složený model, obsahující generátor a procesor, viz obr. 7.2. Kompletní reflektivní simulátor tohoto modelu je pak specifikován jako složený model, zobrazený na obr. 7.3. `Simulator` simuluje generator, Simulator2 simuluje `processor`, `Coordinator` simuluje celý model a `RootSolver` řídí celou simulaci.

Realizace Uvedený reflektivní simulátor lze pro daný model vygenerovat staticky: Na základě modelu se jednou naráz zkonstruuje celá struktura simulátoru pomocí kompilátoru DEVS do DEVS. Takto lze pracovat s DEVS podle původní (statické) definice. Jinou možností je v reflektivním simulátoru zpřístupnit kompletní aplikační rozhraní pro dynamické vytváření, inspekci a editaci modelů (a jejich simulátorů). Pak máme na začátku k



Obrázek 7.2: Model generátoru a procesoru

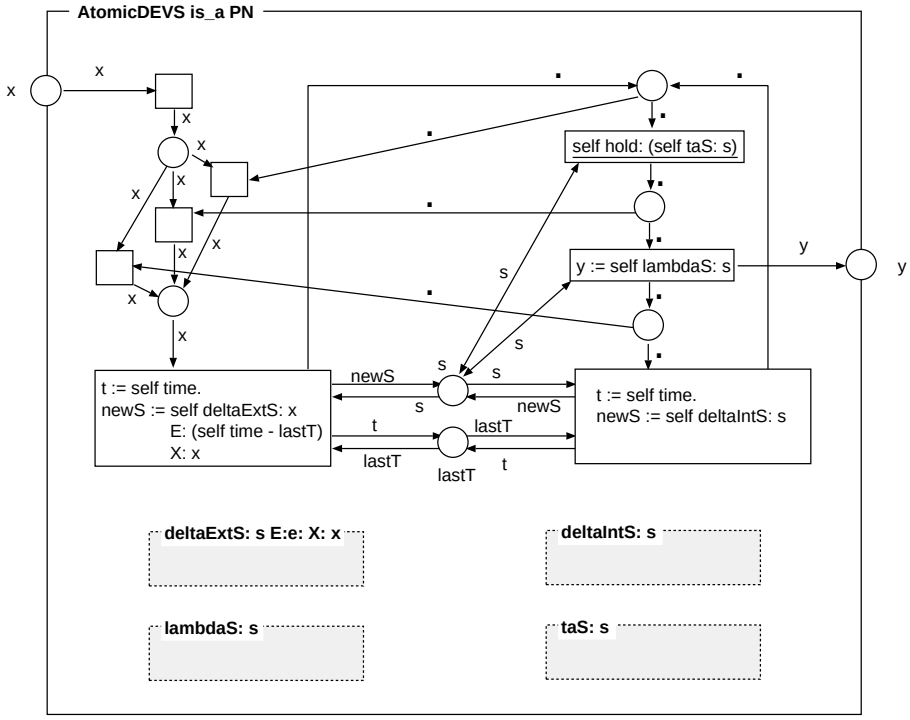


Obrázek 7.3: Model simulátoru modelu generátoru a procesoru

dispozici pouze kořenový simulátor s prázdným modelem. Díky zpřístupněným operacím lze pak dynamicky vytvářet, editovat a manipulovat s modely a s celou simulací.

Praktický význam reflektivní simulace DEVS Lze ověřit funkčnost a analyzovat výkonnostní aspekty různých modifikací simulátoru DEVS.

Použití jiných formalismů Uvedený reflektivní simulátor DEVS používá tzv. metacirkulární architekturu. To znamená, že pro doménovou úroveň i pro metaúroveň je použit tentýž formalismus. Potenciálně je možné pro různé úrovně použít různé formalismy, resp. jazyky. Nyní zvážíme možnost použití jiného formalismu, konkrétně OOPN. Vzhledem k tomu, že jde o velmi vysokoúrovňový formalismus s širokými vyjadřovacími schopnostmi (jako univerzální programovací jazyk) s možností komponentího modelování konformního s konceptem složených modelů DEVS, nebudeme implementovat hierarchický simulátor DEVS pomocí OOPN (přestože bychom to mohli udělat analogicky k uvedené implementaci simulátoru DEVS pomocí DEVS), ale použijeme přímé mapování formalismu DEVS do formalismu OOPN. Mapování atomického modelu DEVS do OOPN je ukázáno na obrázku 7.4. Mechanismus zpracování vstupů, modifikace stavu a generování výstupů je specifikován vysokoúrovňovou Petriho sítí. Každý konkrétní model, implementovaný jako podtřída *AtomicDEVS*, specifikuje (1) funkce DEVS, které jsou realizovány odpovídají-



Obrázek 7.4: DEVS v OOPN

cími metodami v OOPN, a (2) počáteční značení místa s, které obsahuje počáteční stav modelu.

Tento přístup je možné téměř beze změny použít pro mapování DEVS do CPN (Coloured Petri nets), implementovaných nástrojem CPNTools a využít jeho konceptu hierarchických míst a přechodů pro realizaci složených modelů. Takto lze simulovat a analyzovat modely na bázi DEVS s využitím standardního nástroje pro vysokoúrovňové Petriho sítě.

Kapitola 8

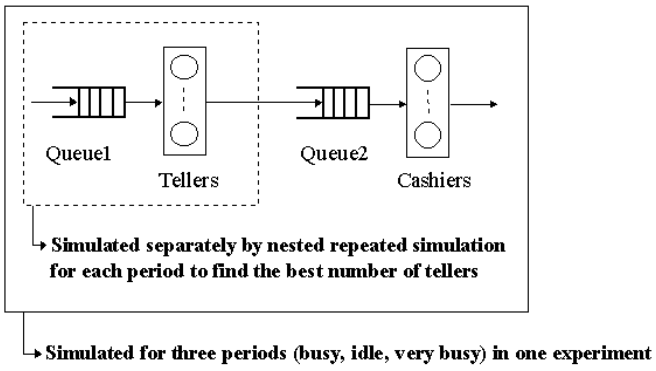
Případová studie: Simulace simulujícího systému

Simulující systém je systém, jehož součástí je simulace. Simuluje-li systém sám sebe a používá-li výsledky vlastní simulace pro modifikaci svého vlastního chování v budoucnu, jde o reflektivní simulaci.

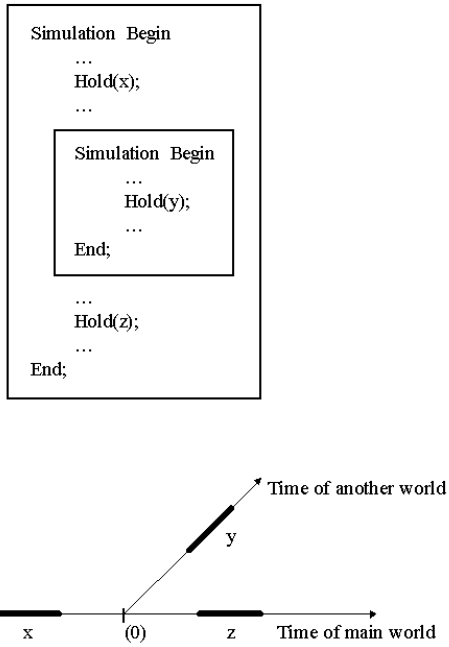
Do třídy simulujících systémů typicky patří systémy, jejichž součástí je rozhodovací proces, obsahující simulaci. Takové systémy se v praxi vyskytuji poměrně často. Stačí si představit situaci [Skl97], kdy skupina expertů navrhne množinu možných řešení daného problému v rámci složitého systému. Přitom důsledky jednotlivých řešení není možné analyticky zjistit ani jednoduše otestovat. Jediným řešením je pro všechny návrhy expertů provést simulační experimenty, jejich výsledky porovnat a poté provést rozhodnutí. Simulace simulujících systémů pak dává smysl jako součást procesu rozhodování problémů, týkajících se simulujících systémů. Jako příklad uvedeme optimalizaci počtu otevřených přepážek v bance v závislosti na změnách pravděpodobnostního rozložení příchozích klientů v průběhu dne.

8.1 Příklad vnořené simulace: Systém hromadné obsluhy, jehož část je optimalizována opakovanou vnořenou simulací

Obrázek 8.1 ukazuje systém, který chceme simulovat. Jde velmi zjednodušenou abstrakci hypotetické banky, kde klienti nejprve čekají ve frontě, by byli obslouženi úředníkem (teller) zpracovávajícím jejich požadavky. Poté je (po čekání v další frontě) obsloužen úředníkem na pokladně (cashier). Pravděpodobnostní rozložení doby mezi příchody klientů a doby trvání oběma typy úředníků jsou známa. Předpokládejme, že banka má k dispozici několk úředníků, kteří jsou schopni pracovat na libovolné z obou pozic. Řízení banky chce optimalizovat přidělení úředníků k oběma typům přepážek v různých částech dne, které se liší četností příchodu klientů do banky. Rozlišují se tři období, která nazveme ”busy”, ”idle” a ”very busy”. Pro nalezení nejlepší strategie přřazení úředníků k přepážkám je použita simulace (je-li systém simulován, jde o vnořenou simulaci): Na začátku každé periody se opakovaně provede simulace první části systému (tj. fronty s množinou přepážek prvního typu – teller) pro různé počty úředníků (tellers) v rozsahu od 1 do celkového počtu dostupných úředníků (10). Na základě výsledků jednotlivých simulací vedení banky určí počty



Obrázek 8.1: Systém hromadné obsluhy se dvěma frontami a obslužnými linkami (převzato z [Skl97])



Obrázek 8.2: Základní idea vnořené simulace (převzato z [Skl97])

úředníků pro přepážky prvního typu (tellers) a druhého typu (cashiers).

Pro jednoduchost v našem simulačním modelu ponecháme výběr nejlepší varianty na interaktivním zásahu uživatele – program nabídne možnosti přerozdělení úředníků (spolu s odpovídajícími simulačními výsledky), z kterých uživatel jednu vybere. Rozhodovacím kritériem je průměrný čas strávený klientem v bance (resp. v její první části, tj. ve frontě a u přepážky prvního typu).

Pro možnost přímého porovnání uvedeme dvě implementace modelu téhož systému, realizované v různých simulačních jazycích. Jedním z nich je klasický jazyk pro simulační modelování (SIMULA), druhý je experimentální vizuální jazyk pro paralelní objektově orientované modelování, založený na Petriho sítích (PNtalk). Nejprve uvedeme kód simulujícího modelu v jazyce SIMULA, převzatý z [Skl97].

8.2 Model s vnořenou simulací v jazyce SIMULA [Skl97]

Model v jazyce SIMULA využívá toho, že třída `Simulation` definuje simulační čas. Každá instance této třídy (nebo její podtřídy) tedy definuje samostatný svět s vlastním časem. Metoda `Hold()` způsobí čekání v simulačním čase příslušné instance třídy `Simulation`. SIMULA umožňuje definovat bloky (`Begin...End`) dědící od existujících tříd. Například konstrukce `Simulation Begin...End` definuje blok, dědící vlastnosti definované třídou Simulation. Příslušný anonymní objekt (instance nepojenované třídy, odpovídající bloku), se vytvoří (inicializuje) v okamžiku, kdy má dojít k zahájení vyhodnocování bloku. Vnořování simulací v jazyce SIMULA je tedy možné velmi přímočaře definovat vnořením bloků dědících od třídy Simulation.[1] Aby si čtenář učinil dokonalou představu, následuje kompletní kód simulačního modelu s vnořenou simulací, předvzatý z [Skl97].

```
! NESTED Simulation using the Simula's class SIMULATION    ;
!                                                          ;
! The example is a model of a bank. Customers are first    ;
! served by tellers, then by cashiers.                     ;
! The input rate changes in three periods: there is a busy ;
! period, then an idle period and again a busy one.        ;
! For each period the repeated inner simulation experiment ;
! simulates the first queue for the particular input rate  ;
! and for various numbers of servers. Then it shows the    ;
! results (average time spent at the first server) and     ;
! prompts the user for the number of tellers and the number;
! of cashiers. Tellers always finish a service that has    ;
! already started. The simulation should find the          ;
! time customers spend in the bank (average and maximum)   ;
! for various numbers of clerks in the three periods.      ;
!                                                          ;

Simulation Begin
                                        ! Global variables: ;
```

[1] V ostatních třídně založených objektově orientovaných jazycích, které neumožňují blokům dědit (což je drtivá většina programovacích jazyků), je třeba pro vnořené simulace explicitně příslušnou instanci třídy Simulation (nebo jejího ekvivalentu, pokud je v daném prostředí k dispozici) vytvořit. Princip ovšem zůstává zachován.

```
Integer Period,Trial;                ! Period, Trial number;
Real Array MinInt,MaxInt(1:3);       ! Min and Max intervals;
Real Array Duration(1:3);            ! Duration of periods [min];
Ref(Head) Queue1,Queue2;             ! The two queues;
Integer MaxClerks, Tellers, Cashiers;  ! Total numbers;
Integer BusyTellers, BusyCashiers;   ! Numbers of working clerks;
Real S1Mean, S1Std, S2Mean, S2Std;   ! Random normal servers;
Integer SeedG, SeedS1, SeedS2;       ! Seeds of the random generators;
Long Real TotalTime, MaxTime;        ! Variables for statistics;
Integer CustomersOut;                ! Number of served customers;

Process Class Generator;
Begin
   While true do begin
         ! Interval between arrivals: ;
      Hold(Uniform(MinInt(Period),MaxInt(Period),SeedG));
      Activate New Customer(Time);
   End While;
End of Generator;

Process Class Customer(Arrival); Real Arrival;
Begin
   Ref(Customer) Next;
   Real Spent;

   If (not Queue1.Empty) or (BusyTellers >= Tellers) then
      Wait(Queue1);    ! Has to wait in Queue1;
                       ! Service can start;
   BusyTellers ← BusyTellers + 1;
   Hold(Normal(S1Mean, S1Std, SeedS1));  ! This is the teller service;
   BusyTellers ← BusyTellers − 1;

   If (not Queue1.Empty) and (BusyTellers < Tellers) then begin
      Next :− Queue1.First;
      Next.Out;                        ! First from Queue1 served;
      Activate Next after Current;
   End If;

   If (not Queue2.Empty) or (BusyCashiers >= Cashiers) then
      Wait(Queue2);    ! Has to wait in Queue2;
                       ! Service can start;
   BusyCashiers ← BusyCashiers + 1;
   Hold(Normal(S2Mean, S2Std, SeedS2));  ! This is the cashier service;
   BusyCashiers ← BusyCashiers − 1;

   If (not Queue2.Empty) and (BusyCashiers < Cashiers) then begin
      Next :− Queue2.First;
      Next.Out;                        ! First from Queue2 served;
      Activate Next after Current;
   End If;

   CustomersOut ← CustomersOut + 1;
   Spent ← Time − Arrival;
```

```
      TotalTime ← TotalTime + Spent;
      If Spent > MaxTime then MaxTime ← Spent;
   End of Customer;

   Procedure Report;            ! Experiment evaluation;
   Begin
      OutText("  *** Report on external simulation ***"); OutImage;
      OutInt(CustomersOut,6); OutText(" customers ready at time ");
      OutFix(Time,2,10); OutImage;
      OutText("Average time in system: ");
      OutFix(TotalTime/CustomersOut,2,10); OutImage;
      OutText("Maximum time in system: ");
      OutFix(MaxTime,2,10); OutImage;
   End of Report;

! MAIN program body;

    SeedG  ← 11;                    ! Seeds of random variables;
    SeedS1 ← 13;
    SeedS2 ← 17;
    MinInt(1) ← 1; MaxInt(1) ← 4;   ! Min and Max intervals;
    MinInt(2) ← 2; MaxInt(2) ← 9;
    MinInt(3) ← 1; MaxInt(3) ← 3;
    Duration(1) ← 120;              ! Duration of periods;
    Duration(2) ← 240;
    Duration(3) ← 120;
    MaxClerks  ← 6;
    S1Mean ← 6;                     ! Random normal servers;
    S1Std  ← 1;
    S2Mean ← 8;
    S2Std  ← 2;
    Queue1 :- New Head;
    Queue2 :- New Head;
    Period ← 1;
    Activate New Generator;

For Period←1 step 1 until 3 do begin

 Real Array TimeSpent(1:MaxClerks);

 OutText("  *** Results of internal simulation *** Period ");
 OutInt(Period,1); OutImage;
 OutText("       Tellers      Average time spent"); OutImage;

 For Trial←1 step 1 until MaxClerks do
 ! ********************************************************* ;
  Simulation Begin
                                  ! Internal Global variables: ;
   Real TrialDuration;            ! Internal experiment [min];
   Ref(Head) Queue;               ! The queue;
   Integer Servers;               ! Total number;
   Integer BusyServers;           ! Numbers of working clerks;
   Integer TrialSeedG,TrialSeedS; ! Seeds of the random generators;
```

```
  Long Real TotTime;                    ! Variables for statistics;
  Integer CustOut;                      ! Number of served customers;

  Process Class IGenerator;
  Begin
     While true do begin
        Hold(Uniform(MinInt(Period),MaxInt(Period),TrialSeedG));
        Activate New ICustomer(Time);    ! Interval between arrivals: ;
     End While;
  End of IGenerator;

  Process Class ICustomer(Arrival); Real Arrival;
  Begin
     Ref(ICustomer) Next;

     If not Queue.Empty or (BusyServers >= Servers) then
        Wait(Queue);      ! Has to wait in Queue;
                          ! Service can start;
     BusyServers ← BusyServers + 1;
     Hold(Normal(S1Mean, S1Std, TrialSeedS));  ! Teller's service;
     BusyServers ← BusyServers - 1;

     If not Queue.Empty then begin
        Next :- Queue.First;
        Next.Out;                       ! First from Queue served;
        Activate Next after Current;
     End If;

     CustOut ← CustOut + 1;
     TotTime ← TotTime + Time - Arrival;
  End of ICustomer;

  ! Internal MAIN program body;

    TrialSeedG ← 7;                     ! Seeds for random variables;
    TrialSeedS ← 23;
    Servers ← Trial;
    TrialDuration ← 600;
    Queue :- New Head;
    Activate New IGenerator;
    Hold(TrialDuration);                ! Internal experiment duration;
    TimeSpent(Trial) ← TotTime/CustOut;
    OutInt(Trial,13);
    OutFix(TimeSpent(Trial),3,23); OutImage;

  End of internal simulation;
! ********************************************************* ;

 OutText("Enter the number of tellers : "); OutImage;
 Tellers ← InInt;
 OutText("Enter the number of cashiers : "); OutImage;
 Cashiers ← InInt;
 Hold(Duration(Period));
```

```
  Report;
  OutText("Press Enter to Continue."); OutImage; InImage;
 End For;
End of program;
```

8.3 Model s vnořenou simulací specifikovaný objektově orientovnými Petriho sítěmi v jazyce PNtalk

Nyní uvedeme model téhož systému, specifikovaný objektově orientovanými Petriho sítěmi (OOPN) v jazyce PNtalk. Vzhledem k převážně vizuálnímu charakteru jazyka PNtalk je styl programování výrazně odlišný od běžného textového, avšak princip, použitý v předchozí části, zůstává zachován. Odlišnosti oproti modelu v jazyce SIMULA jsou dané použitým jazykem a tím, že model pro vnořenou simulaci je smysluplné použít jako nadtřídu vnějšího modelu. Model je specifikován dvěma třídami, *BankModel* a *Bank*, přičemž *BankModel* specifikuje zjednodušený model banky pro vnořenou simulaci (tj. frontu a úředníky jednoho typu a *Bank* specifikuje model celého systému se dvěna frontami a dvěma typy úředníků. Model *BankModel* je zjednodušením modelu *Bank*, proto je výhodné třídu *Bank* specifikovat jako podtřídu třídy *BankModel*. Využije se zde výhodně možnosti dědění struktury Petriho sítí.

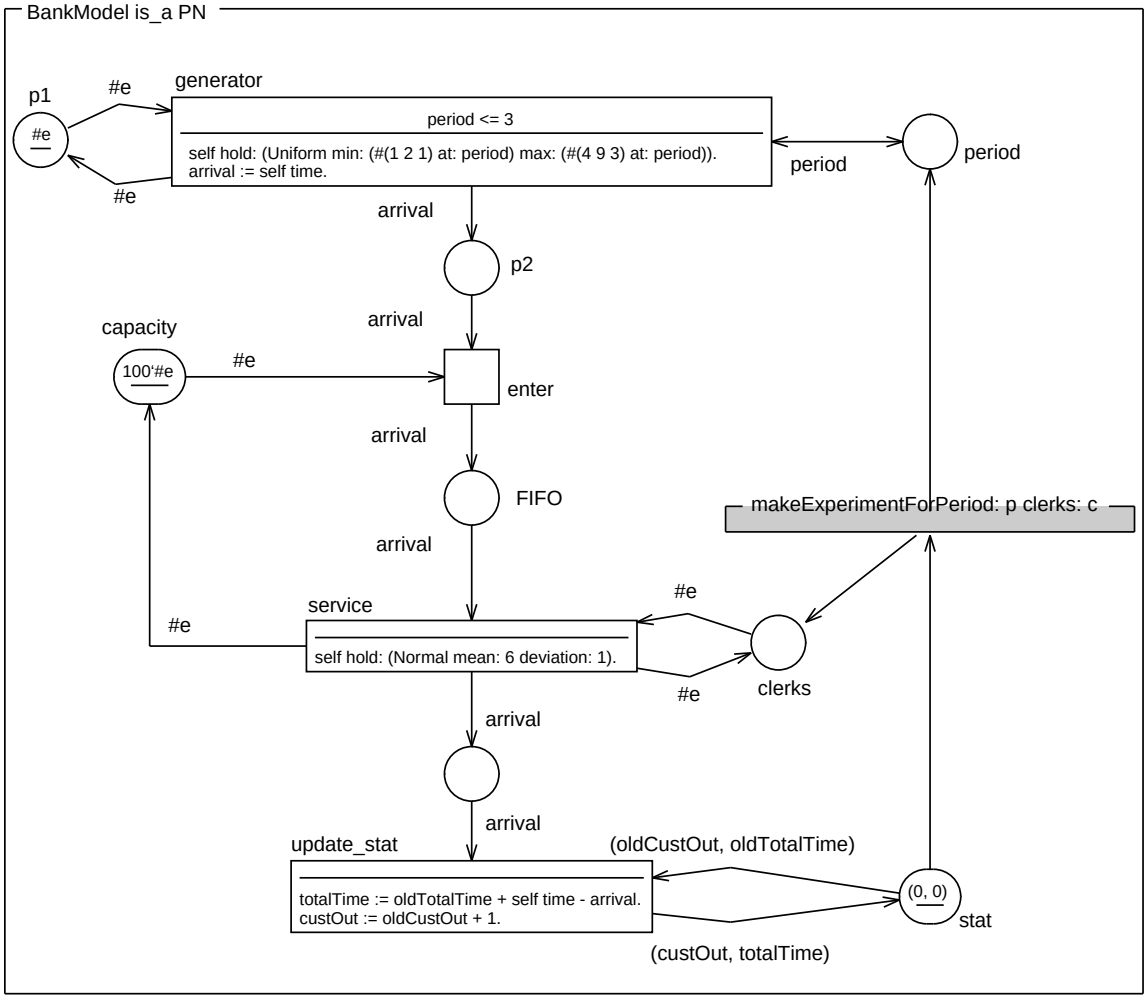
8.3.1 Třída BankModel

Třída *BankModel* (viz obr. 8.3) modeluje banku jako systém hromadné obsluhy. Klienti jsou generováni stochasticky časovaným přechodem *generator*. Banka má kapacitu 100 klientů. Zákazníci jsou paralelně obsluhováni úředníky, kteří jsou modelováni značkami v místě *clerks* (přechod *service* může být prováděn násobně, což modeluje paralelní obsluhu klientů množinou úředníků). Místo *stat* slouží ke sběru statictických dat. Model je navržen pro simulaci ve třech režimech: *busy*, *idle* a *very busy*. Číslo 1, 2 nebo 3 vmístě *period* reprezentuje příslušný režim, který v praxi odpovídá určité části pracovního dne. Ten je do příslušného místa umístěn metodou *makeExperimentForPeriod:clerks:* (viz obr. 8.4). Tato metoda inicializuje model na základě parametrů (period a clerks). Pak nechá běžet simulaci po dobu 600 časových jednotek, což je doba trvání experimentu v čase modelu (*self hold: 600*). Poté z místa *stat* vyzvedne, zpracuje a odevzdá (v místě *return*) výsledky experimentu (průměrnou dobu obsluhy jednoho klienta). Předpokládá se, že nad jednou instancí modelu se provede jen jeden experiment (model, tak, jak ke specifikován, pro jednoduchost nepočítá s možností restartů). Pro každý experiment se počítá s vytvořením nové instance třídy *BankModel*.

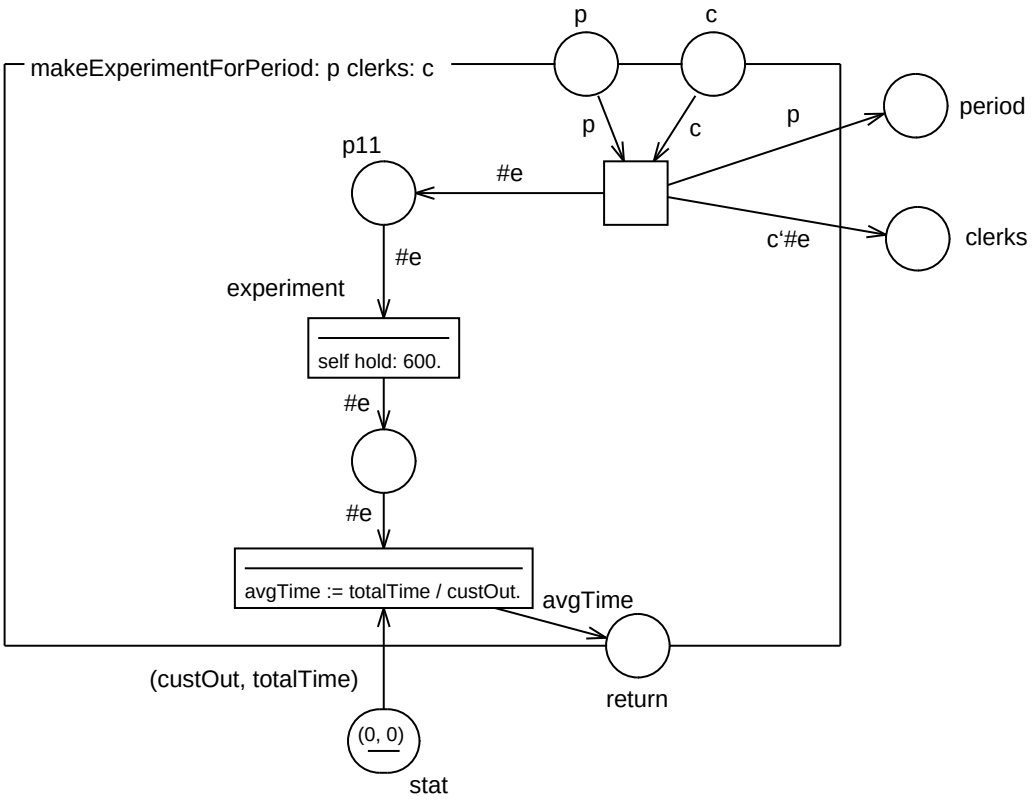
S takto vytvořeným modelem, který bude dále použit soufistikovanějším způsobem, můžeme experimentovat interaktivním vyhodnocením výrazu

BankModel new makeExperimentForPeriod: 1 clerks: 10

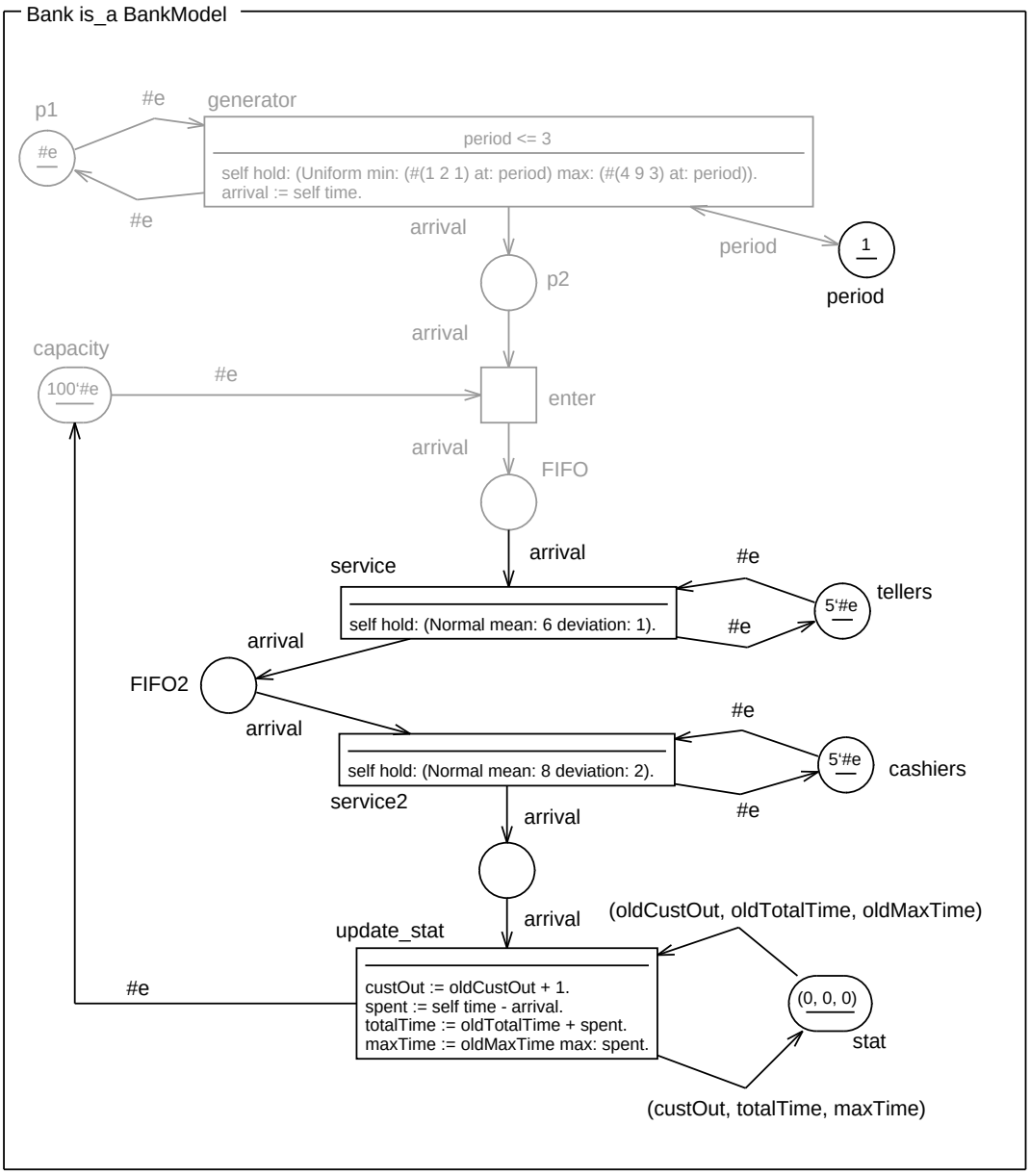
v rámci systému Smalltalk (s knihovnou PNtalk). Výsledkem vyhodnocení tohoto výrazu je průměrný čas strávený klientem v bance (pro zadané parametry).



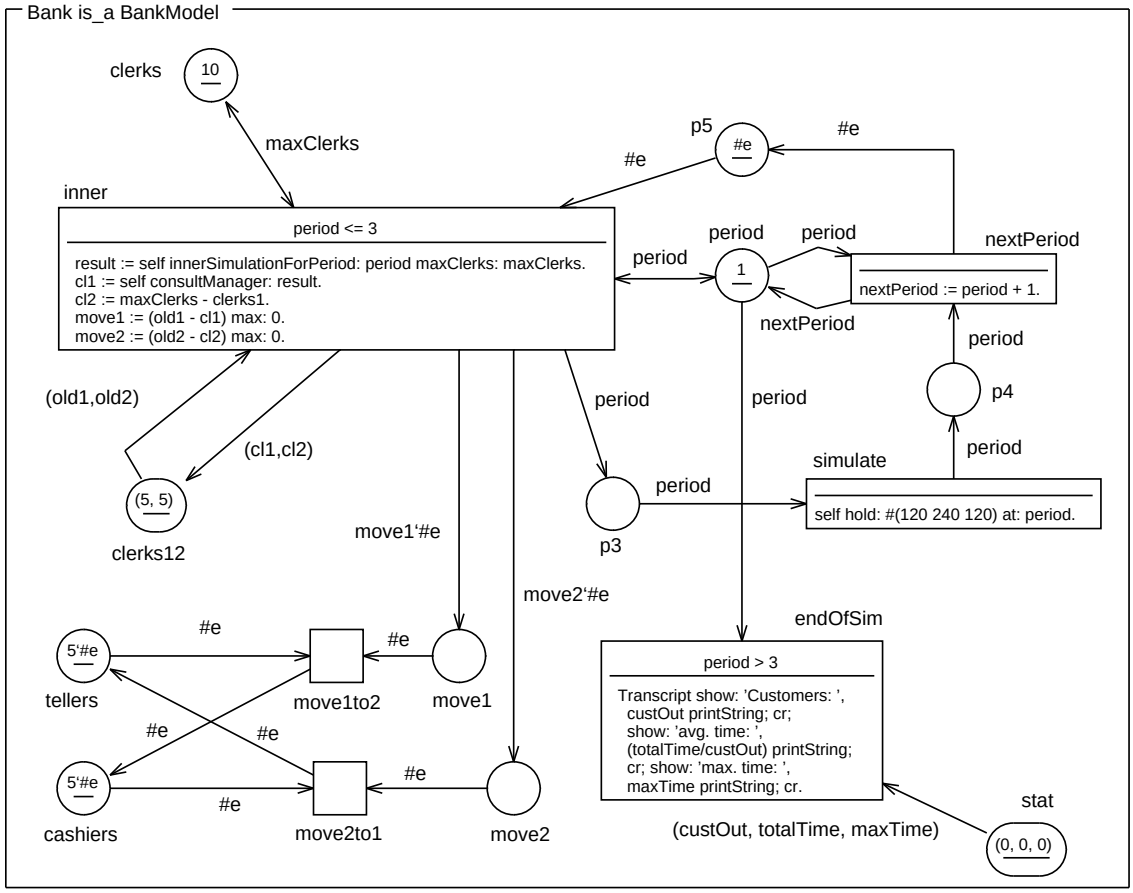
Obrázek 8.3: Třída *BankModel*



Obrázek 8.4: Metoda třídy *BankModel*, určená k provedení simulačního experimentu.



Obrázek 8.5: Třída *Bank* – první část sítě objektu



Obrázek 8.6: Třída *Bank* – druhá část sítě objektu, na první část navazuje místy *period tellers*, *cashiers* a *stat*

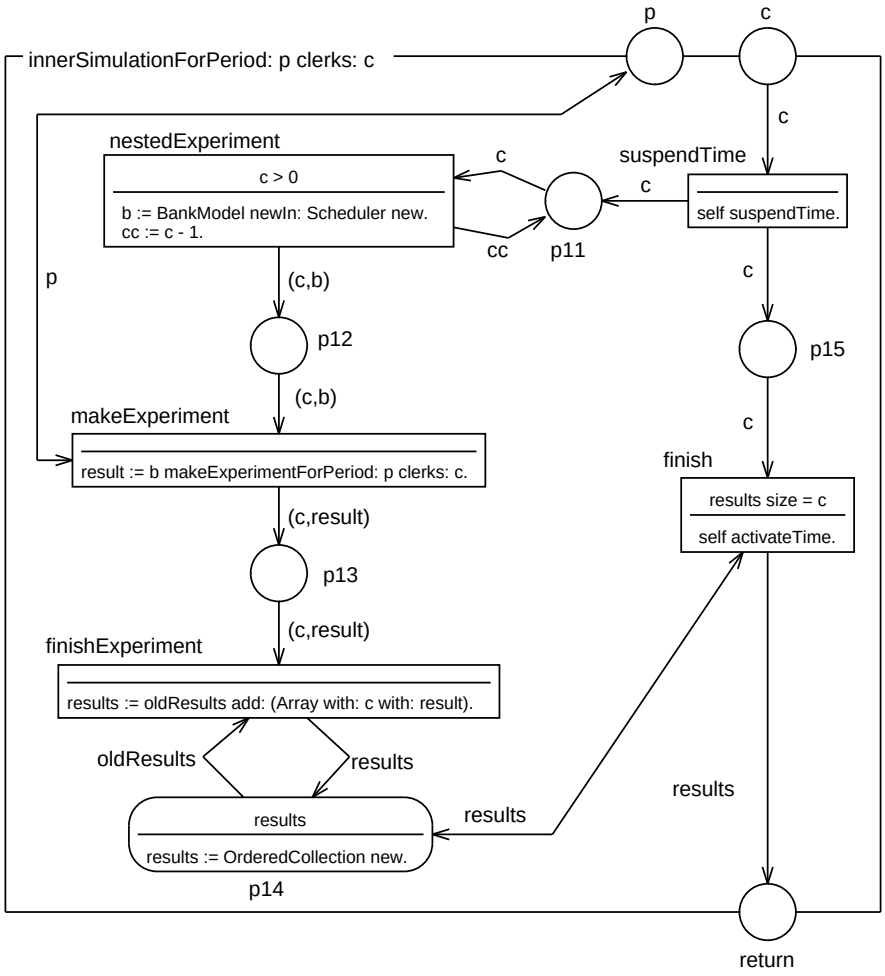
8.3.2 Class Bank

Již vytvořená třída *BankModel* bude nyní použita jako základ pro další specializaci v rámci sofistikovanějšího modelu banky. Navíc ukážeme, jak instance *BankModel* může být simulována uvnitř jiné simulace.

Sofistikovanější model banky se specifikován třídou *Bank*, dědící od třídy *BankModel*. Třída *Bank* přdává druhou obslužnou linku (druhou sadu přepážek) a model řízení banky, který provádí rozhodování o přiřazováných úředníků k jednotlivým typům přepážek na základě výsledků simulace modelu *BankModel*.

Síť objektu, definovaná třídou *Bank* je zobrazena na dvou obrázcích. Obrázek 8.5 zobrazuje průchod klientů bankou. Zděděné části sítě objektu, které zůstávají beze změny, jsou zobrazeny šedě. Obrázek 8.6 zobrazuje mechanismus rozhodování a přemisťování úředníků mezi přepážkami prvního a druhého typu na základě výsledků simulace banky pro tři fáze dne (místo period).

Vnořený model je simulován přechodem *inner* pro aktuálná fázi. Tato činnost je prováděna opakovaně (pro fázi 1, 2 a 3), pak simulace končí – viz cyklus *<p5, inner, p3, simulate, p4, nextPeriod, p5>*.



Obrázek 8.7: Metoda třídy *Bank*, určená k provedení vnořené simulace

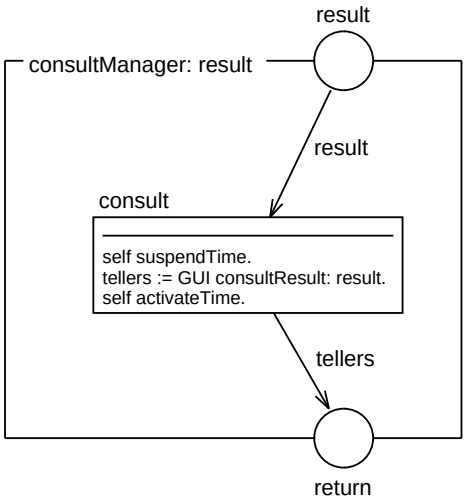
Po každé vnořené simulaci proběhne rozhodování. Rozhodntí o přemístění úředníků je realizovano dialogem s uživatelem (jsou mu přitom předloženy výsledky simulace). To je specifikováno metodou *consultManager*. Doporučená relokace úředníků je specifikována obsahem míst *move1* a *move2*. Přechody *move2to1* a *move1to2* realizují relokaci.

8.3.3 Vnořená simulace, interakce časových os

Vnořená simulace (viz obr. 8.6, přechod *inner*) je vždy startována se dvěma parametry, prvním je fáze dne (period p), druhým je počet dostupných úředníků (clerks c). Obrázek 8.7 ukazuje metodu třídy *Bank*, která provádí vnořené simulace simulaci a sbírá data. Vnořená simulace je prováděna pro všechny možné počty úředníků, začíná se počtem c a v každém kroku se c dekrementuje (viz obr. 8.6.

Vnořená simulace i rozhodnutí o počtu úředníků u obou typů přepážek proběhne z hlediska času modelu okamžitě. V obou případech jde o interakci různých časových os. V prvním případě jde o čas vnořené simulace, v druhém případě jde o reálný čas, ve kterém proběhne dialog. Z hlediska času hlavního modelu trvají obě operace nulovou dobu. Sku-

tečné trvání vnořené simulace v modelovém čase hlavního modelu je realizováno zpožděním v cyklu, prováděnícím jednotlivé simulace (viz obr. 8.6, přechod *simulate*). Vzhledem k tomu, že OOPN/PNtalk je paralelní jazyk, spuštění vnořené simulace i dialog s uživatelem jsou realizovány jako procesy, které běží paralelně s ostatními aktivitami v modelu. V obou případech ale není známa doba ukončení, takže nelze událost ukončení naplánovat. Jelikož chceme, aby vše proběhlo s nulovým zpožděním, řešením je pozastavení toku času hlavního modelu do okamžiku ukončení vnořené simulace nebo dialogu s uživatelem. Pro tyto účely existují operace *suspendTime* a *activateTime*. Je li čas pozastaven, plánovač dovolí provádět pouze události plánované na aktuální čas modelu. Po opětovné aktivaci se mohou uplatnit i další plánované události a čas modelu se může posunout, není-li na aktuální čas již nic naplánováno. Tyto operace jsou použity v metodách *consultManager* (viz obr. 8.8) a *innerSimulationForPeriod:clerks:* (viz obr. 8.7)



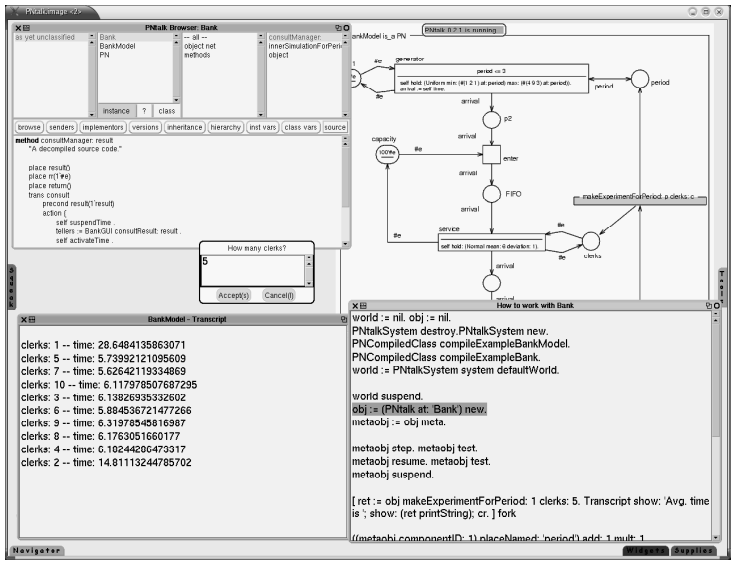
Obrázek 8.8: Metoda třídy *Bank*, určená k dialogu s uživatelem

Přechod *nestedExperiment* (viz obr. 8.7) vytvoří *c* vnořených simulací modelu *BankModel*. Výraz *BankModel newIn: Scheduler new* zajistí, že každá vnořená simulace je provedena v novém plánovači, tj. s nezávislým časem modelu. Reference na simulaci (viz proměnná *b* v obr. 8.7) je pak použita k provedení simulačního experimentu přechodem *makeExperiment*. Přechod *finishExperimet* posbárá výsledky všech simulací. Celý tento proces je odstartován přechodem *suspendTime* a ukončen přechodem *finish*.

8.3.4 Simulace

Simulace se spustí vyhodnocením výrazu *Bank new*. Postupně proběhnou tři fáze, přičemž v každé z nich jsou provedeny vnořené simulace pro různé rozdělení úředníků a jejich výsledky jsou předloženy uživateli i interaktivnímu výběry nejlepší varianty (obr. 8.9).

Tabulka 8.1 zobrazuje simulační výsledky vnořených simulací pro fáze *busy*, *idle* a *very busy* a pro počty úředníků od 1 do 10. Tučně jsou vyznačeny vybrané varianty. Celkové výsledky jsou v tabulce 8.2.



Obrázek 8.9: Simulace.

	1	2	3	4	5	6	7	8	9	10
busy	28.65	14.81	6.14	6.18	**5.74**	5.88	5.63	6.18	6.32	6.12
idle	15.65	**5.62**	5.85	5.85	5.51	5.85	6.10	6.02	6.32	5.56
very busy	22.59	15.01	7.22	**5.89**	6.13	6.10	5.90	5.65	6.18	6.12

Tabulka 8.1: Výsledky vnořených simulací.

	number of customers	avg. time	max. time
main simulation loop	135	14.74	32.43

Tabulka 8.2: Výsledky simulace.

8.4 Závěr

Prezentovaný příklad ve zjednodušené formě naznačil probematiku simulace simulujících systémů. V reálných situacích je třeba oproti uvedenému příkladu řešit komplikovanější parametrizaci, případně vytvoření nového modelu pro (vnořenou) simulaci na základě aktuálně dostupných dat.

Zatímco řešení v jazyce SIMULA umožuje pouze parametrizaci předem vytvořených modelů, PNtalk (případně SmallDEVS) umožňuje dynamickou konstrukci vnořených modelů na základě výsledků introspekce hlavního modelu.

Kapitola 9

Případová studie: Prostředí pro vývoj multiagentních systémů

V této kapitole bude představeno prostředí pro tvorbu multiagentních systémů na bázi objektově orientovaných Petriho sítí. Jde o demonstraci možnosti programovat agenty formálními modely s možností jejich dynamické adaptace za života systému. Tento přístup je podrobněji popsán v publikacích [**?**] a v disertační práci [Maz08].

9.1 Multiagentní systémy

Multiagentním systém je systém, ve kterém vzájemně interagují agenti. Agenti [Woo02] jsou autonomní entity, schopné dlouhodobě plnit cíle zadané uživatelem bez jeho dohledu v daném prostředí. Interakce agentů představuje komplexní sociální aktivity, jako kooperace, soupeření, vyjednávání, apod.

Jako příklady aplikací multiagentích systémů lze uvést komplexní softwarové systémy, kde jednotlivé komponenty jsou natolik nezávislé a složité, že z hlediska návrhu je efektivní považovat je za zcela samostatné jednotky, které spolu komunikují ve velmi malé míře pomocí vysokoúrovňového jazyka, například jen v případě nestandardních situací [SBD+00]. Dalšími oblastmi jsou týmová spolupráce robotů (např. robotický fotbal [Kit98]), telekomunikační aplikace (mobilní agenti, migrující je zdrojům dat, což vede k minimalizaci zatížení komunikační infrastruktury) [AM04], rekonfigurovatelné a adaptivní softwarové systémy, zahrnující zakázkovou výrobu [Kou01], řídící systémy letiště [GR96]), distribuované aplikace pro získávaví znalostí [KHS97], senzorové sítě [LTO03], sémantický web a vyhledávání informací [E. 03], simulace socioekonomických procesů [Mos01] apod. Velká pozornost je věnována nasazování multiagentních aplikací v průmyslu, s čímž úzce souvisí standardizace (o tu se stará organizace FIPA [FIP01]). Současně však stále pokračuje výzkum vhodných vnitřních modelů agenta – a to jak v teoretické, tak v praktické (implementační) rovině.

Využití dynamických modelů Existují tři hlavní důvody pro aplikaci dynamických modelů v oblasti inteligentních (agentních) systémů:

- Během návrhu a vývoje inteligentního systému je třeba se systémem nebo jeho prototypem průběžně experimentovat a dovyvíjet ho za běhu. Struktura modelu (např.

fragmenty plánů deliberativního agenta) jsou interaktivně a inkrementálně vyvíjeny během existence systému.

- Systém se vyvíjí vlastními prostředky sám. Na základě měnících se vnějších podmínek provádí průběžně optimalizaci vlastní struktury s ohledem na aktuální cíle.
- V případě multiagentních systémů je třeba počítat s dynamickým vytvářením, migrací a rušením agentů. Iniciátorem manipulace může být jak agent, tak uživatel nebo vývojář. Agentní platforma (operační systém) musí takovouto manipulaci s agenty umožnit.

V následujícím textu bude stručně naznačena realizace agentního systému prostředky Objektově orientovaných Petriho sítí (OOPN). Více detailů lze najít v [Maz08].

9.2 Aplikace v OOPN a DEVS v multiagentních systémech

Předmětem našeho zájmu jsou deliberativní agenti, založení na Beliefs-Desires-Intentions (BDI) architektuře [Bra87]. Takoví agenti pracují s vnitřnim modelem světa (beliefs), cíli (desires) a záměry (intentions). Záměry jsou vytvářeny v podobě plánů, které dočasně řídí činnost agenta a jsou dynamicky vytvářeny v souladu s aktuálními cíli na základě některých vstupních událostí.

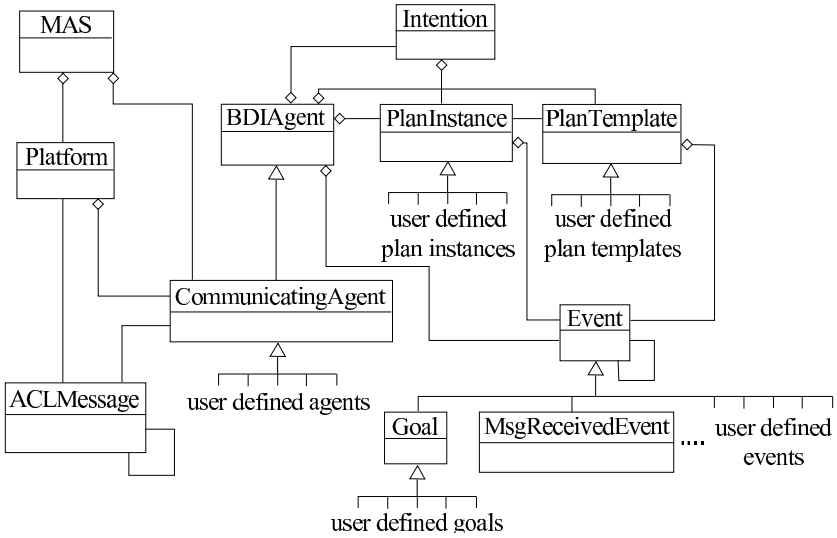
Koncept multiagentího systému v prostředí PNtalk/SmallDEVS Vzhledem k tomu, že agentní systémy jsou ve své podstatě systémy paralelní a distribuované, jeví se smysluplným použít pro jejich specifikaci Petriho sítě. Aplikací Petriho sítí v návrhu agentních systémů různých typů se zabývalo několik autorů, jako jsou [MW97], [WH02], a [YC06]. Tyto přístupy vesmě modelují agenty barvenými Petriho sítěmi, resp. sítěmi s objektovým strukturováním, ale bez dynamické instanciace.

V rámci námi zvolené architektury agenta (BDI) jsou Petriho sítě použity především pro reprezentaci plánů. Vzhledem k tomu, že se zde na konceptuální úrovni pracuje s dynamicky instanciovatelnými Petriho sítěmi, jeví se smysluplným použít pro specifikaci agenta formalismus OOPN, resp. jazyk PNtalk [Jan95] [MVT97] [MVR+06], který je na formalismu OOPN založen. Kromě samotných agentů je formalismem OOPN (v jazyce PNtalk) možné specifikovat i agentní platformu (agentní operační systém), vytvářející potřebnou abstrakci okolní reality a poskytující prostředí pro existenci agentů. Takto navržený agentní systém využívá infrastrukturu, poskytovanou prostředím SmallDEVS [JK06]. Jde zejména o mechanismus řízení simulací a vazbu na externí realitu. V našem případě předpokládáme aplikaci v oblasti malé mobilní robotiky. Vazba na reálné okolí agentů je v tomto případě implementována jednotným rozhraním, zpřístupňujícím jak reálné, tak simulované robotické platformy. Pro simulaci robotů v simulovaném prostředí je použit externí robotický simulátor.

Agentní architektura Architektura systému vychází z existujících architektur BDI agentů, jako je PRS [GL87] a jeho následníci dMars [dLG+04], 3APL [DdBDM03], Jadex [PBL03], AgentSpeak [Rao96] and [BWH07]. Nejvíce je inspirována systémy dMars [dLG+04] a Jason [BWH07]. Originálním přínosem je (1) formální specifikace celého agentního systému (jeho obecných i aplikačně-specifických součástí) objektově orientovanými

Petriho sítěmi a (2) otevřenost, umožňující experimentovat jak s variantními implementacemi agentů, tak i s modifikacemi celé agentní architektury.

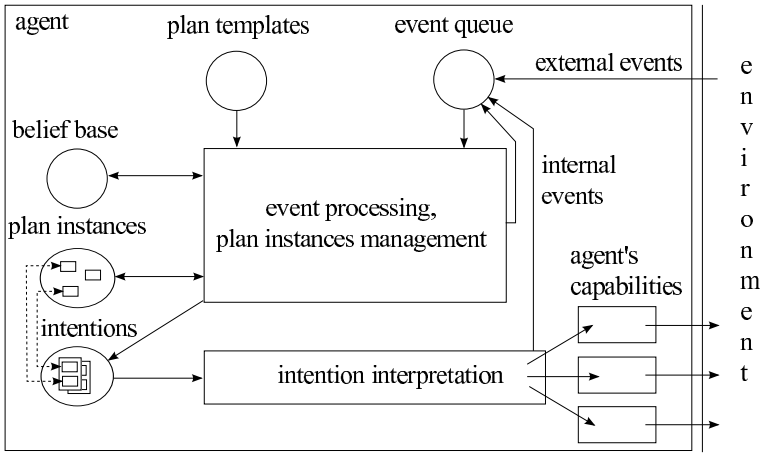
Na obr. 9.1 jsou znázorněny nejdůležitější třídy, kterými je implementováno prostředí PNagent. Z diagramu tříd je patrné, že multiagentní systém systém (MAS) obsahuje komunikující agenty (CommunicatingAgent), kteří si mohou prostřednitvím agentní platformy (Platform) posílat zprávy (ACLMessage). Každý agent obsahuje množinu plánů (PlanTemplate). Veškeré změny stavu systému jsou důsledkem událostí (Event). Události pro agenta generuje platforma. Každý plán má přiřazeny vzory spouštěcích událostí, případně vzory událostí, které plán suspendují, obnoví provádění, způsobí selhání, nebo úspěšné dokončení. Zvláštním typem události je cíl (Goal). Cíle jsou, na rozdíl od běžných událostí, perzistentní (zůstavají v platnosti, dokud nejsou splněny, nebo dokud se nestanou neaktuálními). Instance plánů (PlanInstance) jsou součástí záměrů (Intention). Na základě události nebo vzniku cíle se instanciuje odpovídající plán a vytvoří se záměr. Záměr také obsahuje případné instance podplánů. Obrázek 9.2 obsahuje schematické znázornění jádra agenta, které interpretuje události. V rámci výše uvedeného prostředí musí aplikační programátor specifikovat (1) bázi představ (model světa, beliefs), (2) metody třídy agenta, reprezentující agentovy schopnosti, (3) externí události, reprezentující změny v prostředí, (4) aplikačně specifické plány.



Obrázek 9.1: Zjednodušený diagram tříd systému PNagent, zobrazující nejdůležitější třídy a jejich vztahy.

Model světa (beliefs, báze představ) Třída agenta musí vždy specifikovat model světa. Způsob modelování lze demonstrovat na typickém ukázkovém příkladu pro multiagentní systémy, nazvaném Cleaner World. Jde 2D prostředí maticového typu, obsahující odpad, agenty a popelnice. Pozice odpadu, popelnic a agentů, stejně jako aktuální stav agenta, může být vyjádřen např. v Prologu takto:

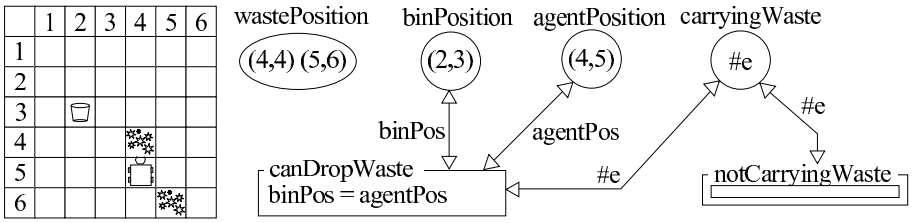
```
wastePosition(5,6).
wastePosition(4,4).
binPosition(2,3).
```



Obrázek 9.2: Schematické znázornění struktury interpretu systému PNagent.

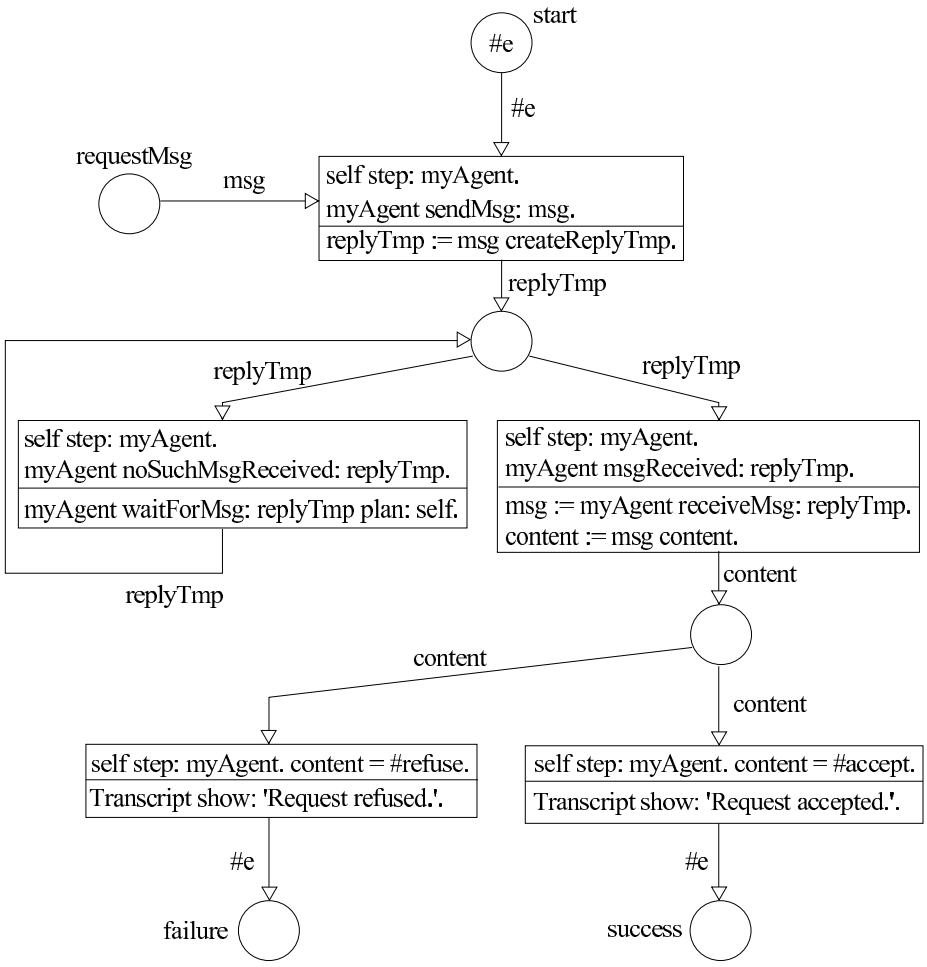
```
agentPosition(4,5).
carryingWaste.
```

Obrázek 9.3 ukazuje, jak lze tentýž model vyjádřit prostředky OOPN. Fakta jsou modelována značkami v místech. Synchronní port `canDropWaste` a negativní predikát `notCarryingWaste` slouží k testování stavu světa, resp. ke zjištění, zda stav světa dovoluje provést nějakou operaci.



Obrázek 9.3: Agentní prostředí Cleaner World a jeho reprezentace v jazyce PNtalk.

Záměry, plány Záměry obsahují instance plánů. Plány jsou předem připraveny aplikačním programátorem. Tato agentní architektura nepočítá s vytvářením plánů od počátčních principů, jako např. STRIPS, ale aktivuje předem připravené dílčí plány podle aktuální situace. Plány ale mohou být specifikovány jen částečně a k jejich dospecifikování může dojít až za běhu, při instaniaci. Plán je specifikován Petriho sítí, která vždy obsahuje místa start, success a failure. Další místa a přechody specifikují kauzalitu jednotlivých operací a stav provádění plánu. Na obrázku 9.4 je příklad instance plánu. Provádění přechodů, specifikujících jednotlivé kroky provádění plánu, je omezeno stráží *self step: myAgent*. Odpovídající synchronní port je proveditelný, je-li plán aktivní a interpret plánu je připraven k provedení kroku. Další stavy provádění plánu (kromě `#active`) jsou `#ready` (připraven k aktivaci), `#suspended` (pozastaven), `#succeeded` (ukončen úspěšně), `#failed` (ukončen neúspěšně). Plány mají přiděleny priority, které se uplatní při řešení konfliktů.

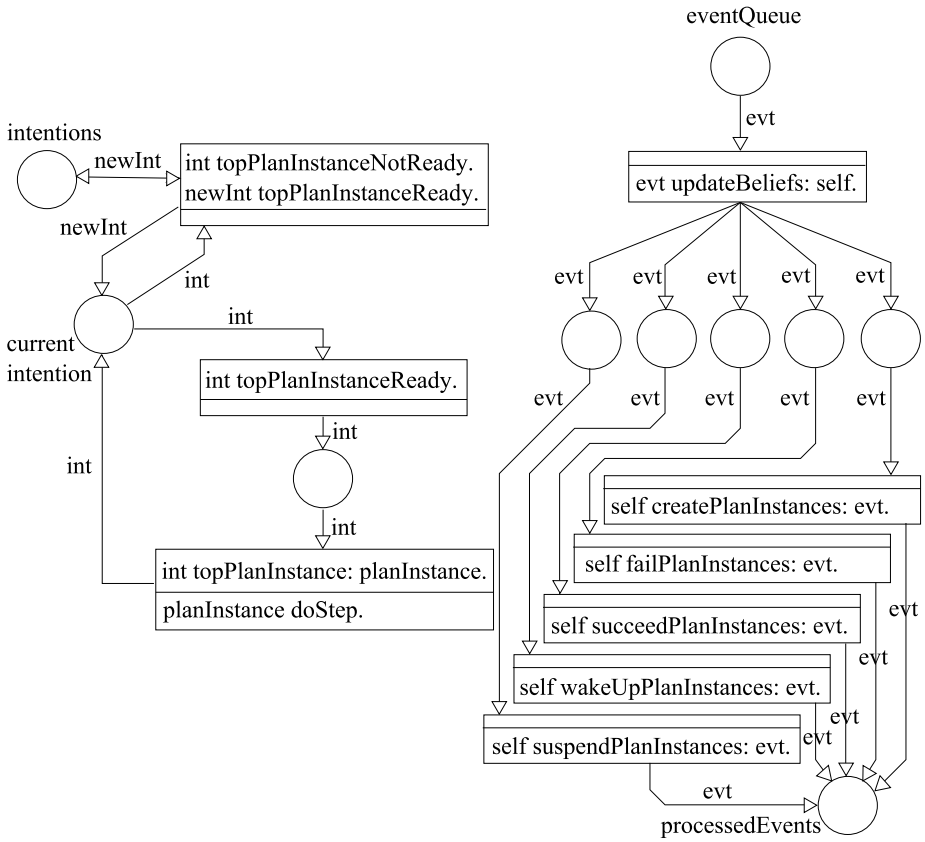


Obrázek 9.4: Příklad plánu specifikovaného Petriho sítí v prostředí PNagent.

Interpret Interpret je implementován třídou BDIAgent. Jádro interpretu je na obrázku 9.5. Tradiční cyklus interpretu BDI agenta – získej nové události, vytvoř odpovídající plány, uprav strukturu záměrů, vyber jeden záměr a proveď jeden krok jeho aktuálního plánu – je zde rozdělen do dvou nezávislých částí. Část zobrazená na obrázku 9.5 vpravo je zodpovědná za zpracování událostí – události jsou postupně odebírány z fronty událostí, mají možnost upravit agentovu bázi znalostí a poté jsou paralelně předány metodám, jež mají za úkol vytváření nových instancí plánů, resp. správu stavu existujících instancí (tedy uspávání, probouzení atd.). Druhá část, zachycená na obrázku vlevo, se stará o interpretaci struktury záměrů.

Interpret vykonává jeden záměr krok po kroku tak dlouho, dokud je tento záměr proveditelný. Poté vybere náhodně jiný proveditelný záměr. Interpretace záměrů představuje vlastní funkčnost agenta – zde se v jednotlivých krocích realizují agentovy plány, tedy provádějí akce, které ovlivňují prostředí.

Agentní platforma Třída Platform definuje prostředí, v kterém agenti existují. Poskytuje agentům tři základní služby: (1) Správu životního cyklu – zejména prostředky pro

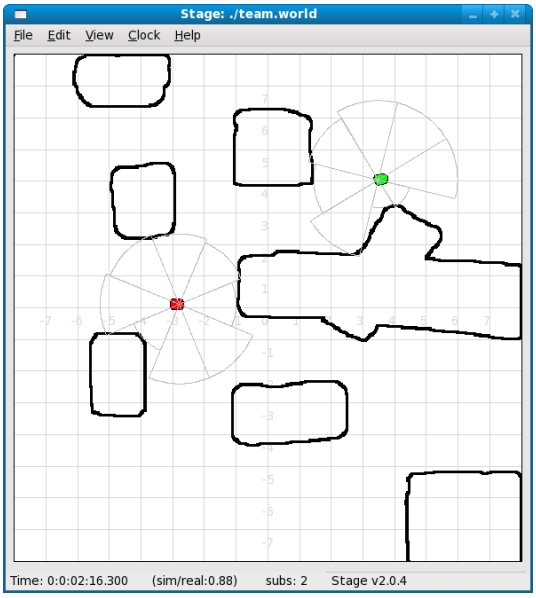


Obrázek 9.5: Zpracování událostí a interpretace záměrů agenta.

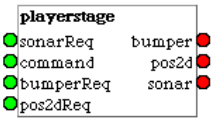
vytváření a odstraňování agentů, přidělování jednoznačných identifikátorů (AID) apod., (2) službu bílých stránek (vyhledávání agentů podle jejich AID), a (3) přenos zpráv mezi agenty. Jazyk zpráv odpovídá podmnožině jazyka FIPA ACL [FIP01]. Nižší vrstvu platformy, včetně vazby na reálné okolí, obstarává prostředí SmallDEVS.

Aplikace prostředí PNagent v oblasti robotiky Výše popsané multiagentní prostředí bylo experimentálně ověřeno v oblasti malé mobilní robotiky. Agent je použit pro řízení robota. To znamená, že má za úkol zpracovávat informace ze senzorů robota a ovládat jeho akční prvky. Vývoj řídicího softwaru (agenta) probíhá v prostředí PNtalk/SmallDEVS. Senzory a akční prvky robota jsou dostupné prostřednictvím speciálního rozhraní na externí realitu, které je v prostředí SmallDEVS realizováno jako komponenta s DEVS rozhraním (viz obr. 9.7). Tato komponenta zpřístupňuje rozhraní na Player/Stage (viz obr. 9.6). Player [GVH03] je middleware pro robotiku (vyvinutý v Univ. Of Southern California),[1] poskytující jednotné rozhraní na senzory a aktuátory fyzických i simulovaných robotů. Z hlediska aplikace jde o server, zpřístupňující senzory a aktuátory v abstrahované formě klientským programům, přičemž klientské programy implementují řízení robotů. Player obsahuje ovladače jak pro simulované, tak pro reálné senzory a aktuátory robotů. K simulaci chování robotů v 2D a 3D prostředí slouží simulátory Stage a Gazebo [GVH03], s kterými Player spolupracuje. Při experimentech byl použit jeden ze základních

[1] `http://playerstage.sourceforge.net/`



Obrázek 9.6: Simulátor Stage



Obrázek 9.7: Rozhraní na Player

modelů robota implementovaných simulátorem, ActiveMedia Pioneer P3-DX 3, vybavený osmi sonary o rozsahu 45° s dosahem 2,5 m, rozmístěnými radiálně kolem robota, osmi nárazníky, kompasem a rádiem, umožňujícím zasílání zpráv ostatním robotům. Příkazy, které je schopen robot vykonat (odpovídají agentovým schopnostem) jsou čtyři: robot se může pohybovat vpřed, zastavit, otočit se o daný úhel a zaslat zprávu pomocí rádia. V takto vytvořeném prostředí lze experimantovat s různými multirobotickými úlohami. Ověření praktické použitelnosti bylo provedeno na úloze formování týmu robotů, jak byla definována např. v [HZ04]. Více podrobností lze najít v [Maz08].

Od simulace k realitě Pro realizaci uvedené multirobotické úlohy byl použit externí software, sestávající ze serveru Player a simulátoru Stage. Přechod od simulovaných robotů v simulovaném prostředí k reálným robotům v reálném prostředí může být proveden několika způsoby, případně v několika fázích:

1. PNagent, PNtalk, SmallDEVS a Player běží na PC, Player komunikuje se senzory a akčními prvky robotů rádiovým spojením.

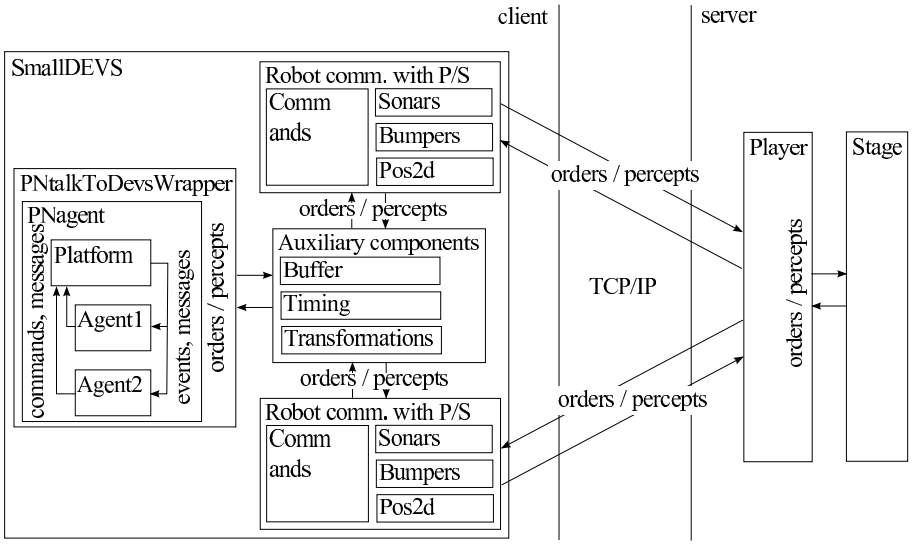
2. PNagent, PNtalk a SmallDEVS běží na PC, Player běží na internetově dostupném palubním počítači každého robota a přímo přistupuje k jeho senzorům a akčním prvkům.

3. PNagent, PNtalk a SmallDEVS běží na několika PC (na každém běží řízení jednoho

robota), Player běží na palubním počítači každého robota a přímo přistupuje k jeho senzorům a akčním prvkům.

4. PNagent, PNtalk, SmallDEVS a Player běží na palubním počítači každého robota.

Ve všech případech lze do prostředí PNagent/PNtalk/SmallDEVS za běhu přistupovat podle konfigurace buď lokálně, nebo vzdáleně. Je tak možné během experimentů detailně sledovat a případně i interaktivně modifikovat stav a strukturu agentů, řídících roboty.



Obrázek 9.8: Konfigurace systému pro multirobotickou úlohu.

9.3 Shrnutí

Kombinace formalismů OOPN a DEVS je pro tuto aplikaci užitečná z několika důvodů. Formalismus DEVS, resp. prostředí SmallDEVS, přináší hierarchicky organizované volně vázané komponenty, události, snadnou manipulaci (plug'n'play), přímou vazbu na teorii modelování a simulace, operační systém. Formalismus OOPN, resp. jazyk PNtalk, přináší paralelismus, synchronizaci, objekty, vizuální jazyk, přirozenou reprezentaci plánů. Jak aplikačně specifická část agenta, tj. reprezentace světa, dílčí plány atd., tak i obecná architektura agenta jsou popsány pomocí OOPN v jazyce PNtalk a interpretovány v rámci prostředí SmalDEVS. Tyto prostředky umožňují interaktivní (i automatický) evoluční vývoj modelů. Je tedy možný inkrementální vývoj jak agentní aplikace, tak i obecné agentní architektury, a to za použití stejných prostředků. Možnost snadné adaptace agentní architektury podle aktuálních požadavků dané aplikace může urychlit a zkvalitnit vývoj agentního systému. Kromě toho potenciálně umožňuje i automatickou evoluci agentů i agentní platformy.

Kapitola 10

Případová studie: Dynamické modely v řízení projektů

V této kapitole bude představen koncept využití dynamických OOPN v řízení projektů, konkrétně v modelování projektového portfolia pro potřeby rozvrhování a monitorování.

10.1 Základní pojmy z oblasti řízení projektů

Projekt Projekt [Anb05] je časově ohraničené úsilí směřující k vytvoření unikátního produktu nebo služby.

Řízení projektů Řízení projektů [Sch05] je proces plánování, koordinování a řízení úloh a zdrojů pro dosažení definovaného cíle v daném čase, s definovanými zdroji a s omezenou cenou. Problematika řízení projektu zahrnuje (1) plánování projektu, (2) organizaci projektu, (3) rozpočet projektu, (4) odhad a kontrolu spotřeby zdrojů a času, (5) řízení rizik projektu, (6) řízení změn, (7) projektovou dokumentaci.

Řízení projektového portfolia Jde o řízení nikoliv jednoho projektu, ale skupiny aktuálních i plánovaných projektů, aby se dosáhlo konkrétních cílů. Smyslem řízení projektového portfolia je optimalizace času, nákladů a zdrojů, přesahující rámec jednotlivých projektů [GK03, Lee04]. Množina aktuálních a plánovaných projektů, tvořící portfolio, se v průběhu času vyvíjí.

Proces Proces je sekvence aktivit vedoucí k očekávanému výsledku. K provádění aktivit jsou využívány zdroje, aktivity transfromují vstupy na výstupy. Aktivity a zdroje v rámci projektu jsou obvykle chápány jako procesy [Anb05].

Síťový diagram projektu Síťový diagram je grafová reprezentace uspořádání jednotlivých aktivit, které vedou k cíli projektu. Síťový diagaram umožňuje specifikovat sekvence akcí, paralelní větve i kolize. Aktivity mají přiřazenu dobu trvání a je tedy možné analyzovat časové charakteristiky projektu (doba trvání, kritická cesta). Často používanou alternativou síťového siagramu je časovaná Petriho síť.

Plánování Plánování je proces výběru a uspořádání aktivit, vedoucích k dosažení cíle [Spi92].

Rozvrhování Rozvrhování [Spi92, Pin02, Pin05] je alokace zdrojů umístěných v časoprostoru za daných podmínek na aktivity tak, aby se splnila zadaná kritéria, například minimalizace celkové ceny daných zdrojů. Výstupem procesu rozvrhování je rozvrh. Jedná se o datovou strukturu obsahující informace o přiřazení aktivit zdrojům v čase.

Statické a dynamické plánování a rozvrhování *Statické (off-line) plánování a rozvrhování* [Pin05, Koc02] předpokládá, že všechny požadavky (cíle) a všechny parametry dostupných zdrojů jsou dopředu známy a v budoucnu se nemění. V praxi jsou takové situace vzácné. V případě projektového portfolia se počítá s vývojem cílů i zdrojů v čase. Během řešení (za běhu systému) se objevují nové cíle a tedy vznikají nové projekty, jiné projekty jsou v důsledku změny cílů rušeny nebo jsou jim měněny parametry. Struktura a parametry zdrojů se také s časem mění. *Dynamické (on-line) plánování a rozvrhování* [Koc02] představuje proces tvorby rozvrhu za běhu systému. Rozvrh je tedy tvořen inkrementálně při změně aktuálně dostupných informací. Důležitým požadavkem je dostatečná rychlost on-line plánování a rozvrhování.

Monitorování Monitorování projektu je sledování významných událostí v průběhu řešení projektu a porovnávání se simulací projektu. Typicky jde o sledování začátků a konců jednotlivých aktivit, případně o události, jako jsou neplánované změny ve struktuře a parametrech zdrojů. V případě jakékoli odchylky reality od simulace je třeba upravit model a okamžitě provést dynamické plánování a rozvrhování, aby model (včetně rozvrhu zdrojů) odpovídal realitě a aby byly splněny všechny aktuální požadavky na systém.

10.2 Aplikace dynamických OOPN v modelování projektového portfolia pro potřeby rozvrhování a monitorování

V této studii ukážeme, že pro adekvátní modelování projektového portfolia na bázi Petriho sítí je třeba použít (at' už přímo, nebo jen na konceptuální úrovni) dynamickou variantu Objektově orientovaných Petriho sítí (OOPN), aby bylo možné provádět jak klonování a editaci pro potřeby optimalizace a rozvrhování, tak dynamickou modifikaci v průběhu monitorování podle měnících se vnějších podmínek (a reflektovat tak změny ve struktuře zdrojů, přidávání nových plánů, změny plánů apod).

Pro potřeby řízení projektového portfolia se používá řada přístupů, metod a modelovacích formalismů, mnohé z nich využívají modely na bázi Petriho sítí [GK03, AG04, HY03]. Vesměs jde o Petriho sítě sice vysokoúrovňové, ale nestrukturované. Využití možnosti strukturování, at' už statického, jako v případě hierarchických Petriho sítí, tak dynamického, jako v případě objektově orientovaných Petriho sítí, se jeví jako možné a přínosné. Očekávanou výhodou objektově orientovaných Petriho sítí je adekvátní strukturování modelu projektového portfolia.

Existuje řada potenciálně použitelných přístupů, kombinujících Petriho sítě a objekty, k nejpropracovanějším (z hlediska související teorie a možností analýzy a verifikace) patří Reference nets [Val98], [Ren06]. V této studii se ale zaměříme na Objektově orientované

Petriho sítě (OOPN) [Jan95, Jan98], vycházející důsledně z klasického objektového paradigmatu [GR89]. Hlavním důvodem tohoto výběru je koncept dynamicky instanciovatelných sítí metod, který v konkurenčních přístupech chybí a který je pro zamýšlenou aplikaci klíčový. Dalším důvodem je otevřená implementace simulátoru OOPN [JK04], která je také pro zamýšlenou aplikaci nepostradatelná.

Koncept modelování projektového protfolia Projektové portfolio lze modelovat pomocí OOPN takto:

- Projektové portfolio je modelováno jedním objektem OOPN. Zdroje jsou modelovány objekty, dostupnými jako značky v místech – značky nesou reference na objekty zdrojů. Tyto odkazy na zdroje jsou distribuovány do míst sítě objektu podle jejich rolí.

- Sítě metod modelují jednotlivé vzory plánů projektů. Jejich instance jsou dynamicky vytvářeny a rušeny v okamžiku startu, resp. ukončení projektu.Tyto plány sdílí přístup k místům sítě objektu, kde jsou k dispozici zdroje. Start projektu je technicky realizován zasláním odpovídající zprávy objektu portfolia (s případnými parametry, upřesňujícími plán). V tom okamžiku je vytvořena nová instance sítě metody (t.j. plánu) a začíná běžet.

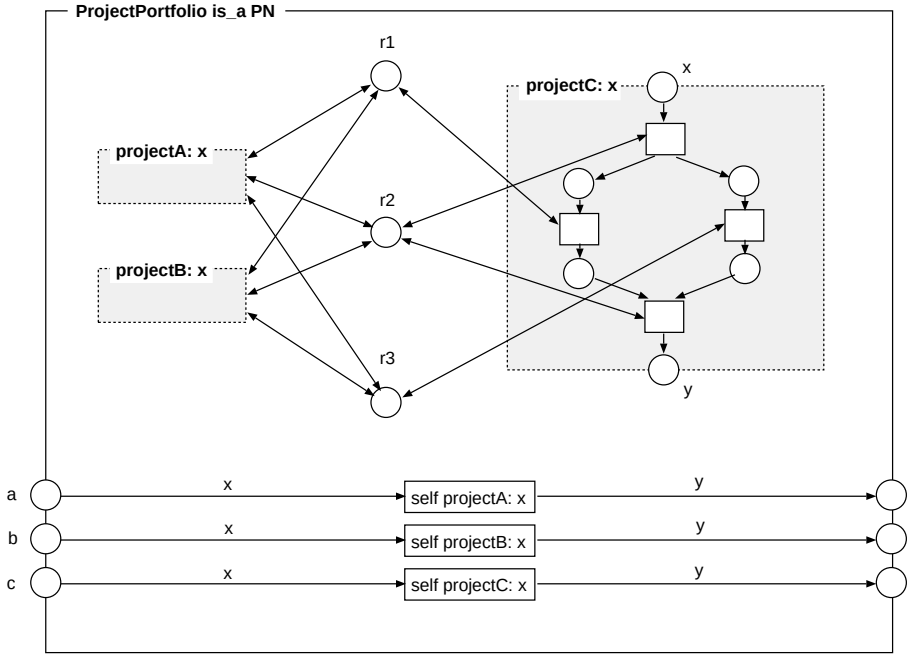
Model projektového portfolia Na obr. 10.1 je příklad modelu projektového portfolia. Obsahuje tři různé typy projektů, resp. tři typy plánů, modelované metodami OOPN. Tyto plány sdílí tři typy (resp. role) zdrojů. Typy zdrojů (jejich role) jsou modelovány místy sítě objektu portfolia.

Plány A a B jsou na obr. 10.1 znázorněny bez zobrazení jejich sktruktury, plán C je ukázán i s jeho strukturou. Přechody sítě objektu portfolia obsahují výrazy tvaru *self projectA: x*. Tyto přechody jsou určeny k invokaci odpovídajících metod zasláním příslušných zpráv, tedy pro vytvoření a nastartování jednotlivých konkrétních projektů. Poznamenejme, že jsou přípustné násobné invokace metod OOPN, které se pak mohou překrývat v čase.

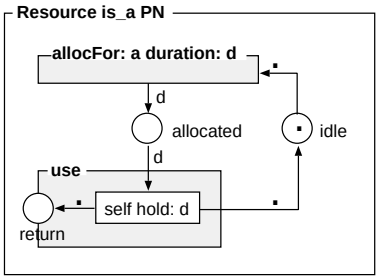
Hierarchická kompozice plánů Plány projektů mohou být složeny z podplánů. K tomu lze použít koncept invokačního přechodu, který zasláním zprávy *self subplan* vytvoří novou instanci subplánu a v okamžiku ukončení jejího provádění pokračuje umístěním značek do výstupních míst.

Zdroje a jejich role Obrázek 10.1 neukazuje všechny detaily – nejsou v něm prozatím uvažovány jednotlivé zdroje. Předpokládá se, že zdroje jsou modelovány značkami v místech, která odpovídají jejich rolím. Zdroje jsou instance třídy *Resource* (viz obr. 10.2). Vzhledem k tomu, že značky v OOPN jsou reference na objekty, každý zdroj může mít v takto koncipovaném modelu potenciálně více než jednu roli. Obrázek 10.3 ukazuje situaci, kde je jeden zdroj (objekt třídy *Resource*), mající dvě různé role, sdílen dvěma projekty.

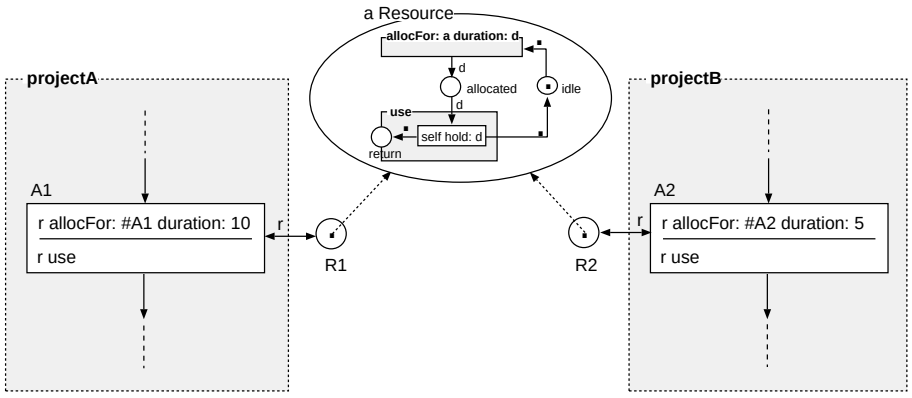
Každá aktivita požaduje zdroj určitého typu (zdroj schopný plnit určitou roli) na určitou dobu. Aktivita $A1$ projektu A požaduje zdroj s rolí $R1$ na 10 časových jednotek, aktivita $A2$ projektu B požaduje zdroj s rolí $R2$ na 5 časových jednotek. Požadovabá doba trvání aktivity je obvykle odhadnuta na základě předchozích zkušeností s podobnými



Obrázek 10.1: Model projektového portfolia

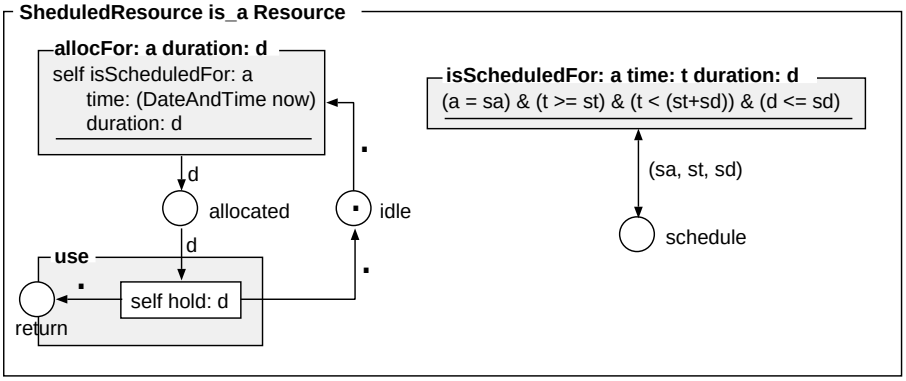


Obrázek 10.2: Zdroj



Obrázek 10.3: Konflikt dvou aktivit požadujících jeden zdroj se dvěmi rolemi

projekty.[1] Místa $R1$ a $R2$ obsahují značku odkazující na tentýž objekt, modelující sdílený zdroj. Každá aktivita se nejprve snaží zdroj alokovat. Není-li zdroj dostupný, čeká se na jeho uvolnění. To je specifikováno stráží každého přechodu, modelujícího aktivitu. Podaří-li se zdroj alokovat, je následně použit. Použití zdroje je modelováno časovaným přechodem (viz metoda *use* v třídě *Resource* – zpoždění je realizováno zprávou *self hold: d*).



Obrázek 10.4: Zdroj s rozvrhem.

Rozvrhy zdrojů Obrázek 10.4 ukazuje zdroj s rozvrhem. Stráž synchronního portu pro alokaci zdroje kontroluje, zda je požadavek v souladu s rozvrhem. Rozvrh je uložen v místě *schedule* jako množina trojic (*aktivita, čas, trvání*) a je k dispozici na dotaz (voláním synchronního portu). Rozvrh pro každý zdroj je vytvořen vhodným optimalizačním procesem mimo model a poté umístěn do místa *schedule*. Model s doplněnými rozvrhy pro jednotlivé zdroje je pak použitelný pro analýzu, ověření správnosti modelu a pro monitorovnání.

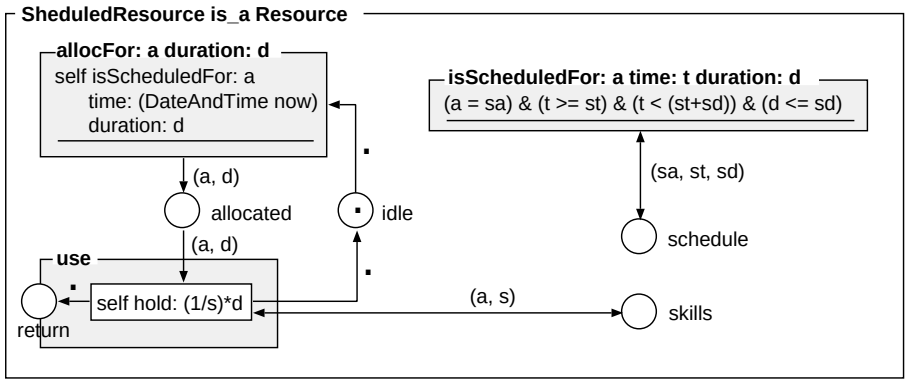
Schopnosti zdrojů Pro každý zdroj lze specifikovat jeho schopnost vykonávat určité aktivity. Jde o míru vhodnosti zdroje pro danou aktivitu. Tento údaj lze získat analýzou dat z předchozích projektů. Doba vykonávání aktivity je pak nepřímo uměrná schopnosti zdroje. V případě zdroje se schopností $s = 1.0$ je skutečná doba trvání (tj. doba použití zdroje) rovna typickým požadavkům aktivity, v jiných případech může být tato doba delší (i kratší). Časovaný přechod metody *use* na základě informací z místa *skills*, které obsahuje dvojice $(aktivita, schopnost)$, provede čekání po dobu $(1/s)d$. Obrázek 10.4 ukazuje doplněnou verzi třídy *ScheduledResource*.

Alokace více zdrojů pro jednu aktivitu Vyžaduje-li aktivita více zdrojů, je to třeba odpovídajícím způsobem specifikovat. Obrázek 10.6 ukazuje model aktivity, požadující tři zdroje se dvěmi rolemi. Synchronní použití zdrojů (srovnání začátků a dob trvání použití zdrojů) je modelováno zprávou (a odpovídající metodou zdroje) *r useWith: s with: t*.[2]

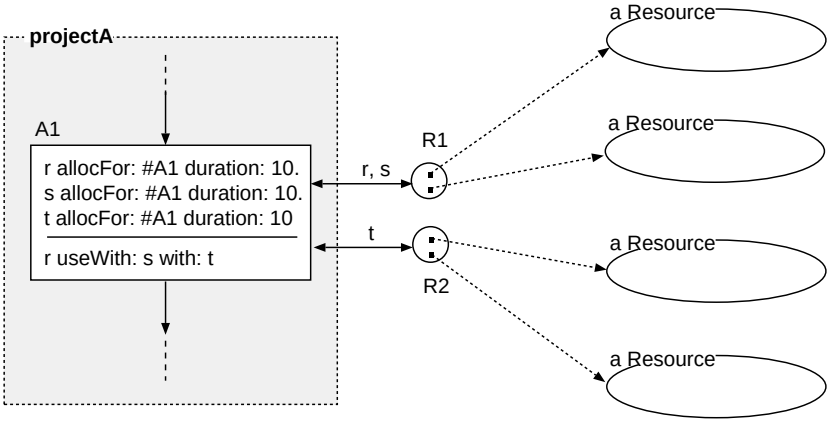
V případě, že je pro danou aktivitu nutné použít více zdrojů, lze zohlednit vhodnost dané kombinace zdrojů pro týmovou práci. To lze specifikovat maticí, která každé dvojici

[1] Například metodami získávání znalostí z databází (datamining).

[2] Existují varianty této zprávy pro různý počet parametrů. Odpovídající metoda třídy *ScheduledResource* zjistí maximální dobu trvání, nastaví ji všem zdrojům (voláním k tomu určených synchronních portů), poté zašle původní zprávu *use* paralelně všem zůčastněným zdrojům včetně sebe a počká na její dokončení všemi zůčastněnými zdroji.



Obrázek 10.5: Zdroj s rozvrhem a schopnostmi.



Obrázek 10.6: Aktivita vyžadující více zdrojů

zdrojů přiřazuje míru schopnosti spolupracovat. Podobně jako v případě typické doby trvání aktivit a schopnosti vykonávat konkrétní aktivity, schopnost týmové spolupráce zdrojů lze získat analýzou dat z předchozích projektů. Tato schopnost se opět projeví v délce skutečného trvání aktivity. Konkrétní implementace je v metodách typu *useWith: r* ve třídě *ScheduledResource*.

Koncept použití modelu pro rozvrhování a monitorování Praktické použití OOPN v modelování projektového portfolia pro potřeby rozvrhování a monitorování můžeme nyní podrobněji specifikovat takto:

- Simulace se používá pro ověření správnosti návrhu projektu. Zde se uplatní hlavně interaktivní simulace. Model je též použitelný pro analýzu, vedoucí např. ke zjištění kritické cesty v projektu. Start projektu se simuluje invokací odpovídající Petriho sítě (to je technicky řešeno zaslánám zprávy s případnými parametry, výsledkem je invokace odpovídající metody objektu projektového portfolia).
- Opakovaná automatická simulace je použita také jako součást fitness funkce při optimalizaci způsobu přidělování zdrojů aktivitám. Výsledkem tohto optimalizačního procesu je rozvrh, který vzájemně mapuje zdroje, časové intervaly a aktivity.

- Simulace s definovanými rozvrhy zdrojů je může být použita k další analýze. Model je již deterministický[3] a simulací lze např. vygenerovat Ganttův diagram pro potřeby dokumentace projektu. Lze potenciálně zohlednit i neurčitost a pracovat s fuzzy rozvrhem.
- Model s definovaným rozvrhem zdrojů může být použit pro monitorování projektu. Pro monitoring předpokládáme simulaci synchronizovanou reálným časem a událostmi z reálného prostředí. Mapování reálného času a času modelu přitom respektuje fakt, že čas modelu pokrývá pouze pracovní dobu.[4]
- Při změně ve struktuře zdrojů se musí okamžitě automaticky aktualizovat místa, značky a objekty, které tyto zdroje modelují, a provede se nové rozvrhování. Po vytvoření nového rozvrhu se aktualizují objekty zdrojů (protože obsahují vlastní rozvrh pro jednotlivé činnosti).
- Pro potřeby rozvrhování se vytvoří klon okamžitého stavu (jde o stav modelu v průběhu monitorování) a ten je použit jako počáteční stav pro zkoumání všech možných budoucností portfolia s různými způsoby přidělování zdrojů v rámci zvolené optimalizační strategie a zvolených kriterií.
- Vytvoření nového typu projektu (nového plánu) znamená přidání nové metody objektu portfolia za běhu. Síť, modelující plán projektu, může být podle měnících se vnějších požadavků modifikována v průběhu řešení, stejně tak i jednotlivé její instance.

Zachování modelu Díky předpokládané možnosti zasahovat do běžící simulace v průběhu monitorování a díky možnosti stav simulace klonovat a dále použít pro potřeby analýzy a optimalizace je snadno realizovatelný fenomén zachování modelu (*model continuity*) v projektovém řízení. Základní myšlenkou je považovat model projektového portfolia specifikovaný formalismem OOPN za základní a všechny ostatní pohledy na projektové portfolio automaticky generovat, případně z jiných pohledů automaticky generovat a realizovat automatické editační zásahy do modelu na bázi OOPN.

K praktické realizaci Uvedený způsob použití OOPN vyžaduje realizovat simulátor otevřeným způsobem. Prakticky to znamená zpřístupnit metaobjektový protokol (MOP) simulátoru. MOP (jinak též reflektivní rozhraní) simulátoru umožní zasahovat do struktury modelu za běhu.

Takové rozšíření OOPN bylo již navrženo a experimentálně implementováno v prostředí jazyka a systému PNtalk, což je implementace OOPN ve Smalltalku [JK04]. OOPN/PNtalk MOP umožňuje vytváření, rušení, inspekci a editaci jednotlivých Petriho sítí definujících třídy a jednotlivých instancí sítí implementujících jednotlivé živé objekty. Umožňuje také vytvořit klon běžícího systému a uložit, resp. načíst ho do/z databáze.

Alternativou k přímému použití simulátoru OOPN je možnost OOPN použít jen na konceptuální úrovni s tím, že samotná realizace může být jiná. OOPN je možné transformovat do jiných formalismů a prostředí. Cílovou realizací může být např. databázová

[3] Zdroje jsou modelovány objekty, dostupnými jako značky v místech sítě objektu. Stráže přechodů modelujících jednotlivé aktivity v rámci jednotlivých projektů kontrolují kromě okamžité fyzické dostupnosti zdroje také to, zda rozvrh zdroje dovoluje zdroj aktivitě přidělit (to je implementováno metodami zdroje).

[4] Čas modelu by mohl být zaveden i jinak. Model by pak byl obecnější, ale také komplikovanější.

aplikace. Pro optimalizaci a rozvrhování pak mohou být použity existující obecné nástroje, např knihovny pro genetické algoritmy apod.

10.3 Shrnutí

Studii o modelování projektového portfolia Objektově orientovanými Petriho sítěmi můžeme shrnout takto:

- Obvyklým řešením s využitím Petriho sítí je udržovat jednu globální vysokoúrovňovou Petriho síť pro celé projektové portfolio a tuto Petriho síť postupně v rámci monitorování aktualizovat a po aktualizaci použít jako východisko pro rozvrhování.
- Modely na bázi OOPN nabízejí možnost použít dynamicky instanciovatelné Petriho sítě pro modelování jednotlivých plánů projektů. Předem definované plány lze dynamicky instanciovat, instance plánů mohou být parametrizovány. Jednotlivé plány jsou ale staticky fixovány – odpovídající Petriho sítě jsou v případě klasické varianty OOPN pevně dány v době startu systému.
- Doporučené řešení: Instanciovatelné Petriho sítě s možností editovat jejich strukturu za běhu. Plány projektů pak mohou být libovolně modifikovány, zatímco strukturování modelu do jednotlivých plánů zůstává zachováno. Pracuje se s předem definovanými typy plánů projektů a s jejich instancemi. Dynamicky lze editovat jak typy plánů, tak jejich jednotlivé instance.

Výhodou přímého použití OOPN pro modelování projektového portfolia je adekvátní strukturování modelu, takže nevzniká jedna nestrukturovaná síť, kde jsou dílčí sítě jednotlivých projektů spojeny do jednoho celku, ale jednotlivé sítě existují a jsou pod kontrolou samostatně (např. zánik, resp. ukončení projektu jsou modelovány adekvátním způsobem). Pracuje se tím pádem na vyšší úrovni než v případě editace jedné nestrukturované sítě – pojmy problémové domény jsou v modelu zachovány, vytvořením modelu se neztrácí informace o jednotlivých projektech portfolia. Dalším přínosem tohoto řešení je zavedení typů (vzorů) plánů, a to jak předem připravených, tak i dynamicky vznikajících a dynamicky aktualizovatelných v průběhu řešení.

Kapitola 11

Závěr

V této práci byly popsány vybrané aspekty problematiky modelování a simulace vyvíjejících se a nepřetržitě běžících systémů, zahrnující jak teoretická východiska, tak možnosti praktické realizace. Byly uvažovány různé aplikační oblasti, zahrnující jak automatickou, tak interaktivní evoluci systémů.

Význam DEVS pro modelování, simulaci a návrh systémů Hlavním formalismem, použitým v tomto textu, je DEVS. Formalizace simulačních modelů pomocí DEVS přináší platformně nezávislé simulační modely a možnost transformace modelů do různých simulačních prostředí. Sofistikovanější formalismy, jako jsou například Petriho sítě a stavové diagramy (statecharts) je možné do DEVS relativně snadno transformovat. S využitím DEVS je možné srozumitelně a na formálním základě popsat problematiku distribuované simulace, viz např. [ZKP00]. DEVS také hraje významnou úlohu v současných aktivitách v oblasti vývoje systémů na bázi simulace (Simulation-Based Design and Development), kde koncept zachování modelu (Model Continuity) významně přispívá k efektivnosti a bezpečnosti vývoje [HZ04]. Hlavní témata aktuálního výzkumu souvisejícího s formalismem DEVS lze shrnout takto:

- standardizace modelovaích formalismů,
- transformace modelů mezi různými formalismy a implementačními jazyky,
- metodologie modelování a návaznost na metodiky softwarového inženýrství,
- interoperabilita nástrojů a přenositelnost modelů,
- standardní knihovny znovupoužitelných modelů,
- webové služby pro simulaci.

Dynamické prostředí pro modelování a simulaci modelů na bázi DEVS, o kterém byla v tomto textu řeč, je v souladu s obecným trendem ve stále větší míře aplikovat dynamické programovací jazyky v realizaci programových systémů. Výhodou je rychlejší vývoj systémů a možnost sledovat a modifikovat jejich dynamiku, což vede k lepšímu vhledu do chování systémů a tedy ke kvalitněji navženým systémům. Jednoduchý a srozumitelný formální základ modelů na bázi DEVS umožňuje kromě simulace a testování přímo aplikovat i matematické metody pro analýzu systémů, jejich struktury i chování.

Jiné formalismy DEVS modeluje systém jako stavový stroj (i když množina stavů může být i nekonečná) s explicitně vyjádřeným stavem a s explicitně vyjádřenými přechody, modifikujícími stav. To nemusí být na první pohled pro praktické programování příliš výhodné. DEVS má ovšem smysl používat jako komponentní prostředí, kde s výhodou využijeme toho, že jde o volně vázané komponenty, s kterými je možné snadno dynamicky manipulovat právě díky explicitně vyjádřenému stavu a volné vazbě na ostatní komponenty. Obvyklé přístupy k programování je výhodné použít uvnitř komponent DEVS. Mapování vybraného formalismu (jazyka) do komponenty DEVS znamená vyjádřit explicitně stavy a přechody.

V kapitole 6 byla popsána možnost specifikovat komponentu DEVS formalismem OOPN. Výhodou tohoto přístupu je čitelnější, vysokoúrovňovější, vizuální reprezentovatace modelu. Zapouzdřením interpretu OOPN bylo dosaženo spojení výhod paralelního objektově orientovaného jazyka a komponentního přístupu se snadno dynamicky manipulovatelnými komponentami. Interpret OOPN zpřístupňuje reflektivní rozhraní, umožňující s objekty interpretu manipulovat podobně jako s komponentami DEVS. Vzhledem k charakteru formalismu jsou zde ale jistá omezení plynoucí z toho, že objekty (na rozdíl od komponent) jsou provázány referencemi a zachovat konzistenci modelu po editačních zásazích není zcela jednoduché.[1]

CAST CAST je zkratka pro Computer-Aided Systems Theory [Pic89]. DEVS, jako modelovací formalismus, který bezprostředně z teorie systémů vychází, dokonale odpovídá konceptu spojení počítačových nástrojů a teorie systémů. Každá konkrétní implementace abstraktního simulátoru DEVS je vlastně přímočarou počítačovou implementací teorie systémů (s diskrétními událostmi). Vezmeme-li v úvahu koncept zachování modelu v cílové realizaci, jde v případě DEVS a jiných kompatibilních formalismů (DEVS-like systems) o příklad ideálního sepětí teorie a praxe tvorby počítačových systémů, kde konkrétní, prakticky realizované systémy a jejich komponenty jsou současně matematickými objekty, manipulovatenými v souladu s teorií a současně přímo použitelnými v reálném prostředí.

Počítačové nástroje V tomto textu popisované obecné principy simulačního modelování vyvíjejících se systémů (kapitola 4) jsou ilustrovány konkrétními implementacemi abtraktní simulační architektury, navržené právě pro tento typ systémů. Implementačním prostředím je Smalltalk. V něm vytvořené nástroje SmallDEVS a PNtalk (kapitoly 5 a 6) jsou použitelné jako pomůcky pro studium problematiky modelování a simulace a jako demonstrace některých ne zcela běžných přístupů, jejímž cílem je podpořit tvorbu nových a další vývoj existujících simulačních nástrojů. Propojení experimentálních nástrojů SmallDEVS a PNtalk se soudobými průmyslově použitelnými a používanými nástroji je potenciálně možné, protože jako součást dalšího výzkumu se počítá s možností zajištění interoperability za použití platformně neutrálního jazyka DEVSML, na jehož vývoji se v komunitě výzkumníků z oblasti modelování a simulace v současné době pracuje [MRMZ07]. Ve speciálních případech, kdy není třeba interoperabilitu s jinými nástroji řešit, lze tyto experimentální nástroje použít přímo. Situaci usnadňuje fakt, že jde o volně šiřitelné nástroje, s jejichž zdrojovým kódem lze díky liberální licenci (MIT) nakládat prakticky bez omezení.

[1] Tato problematika je stále předmětem výzkumu.

Originální výsledky Hlavním přínosem této práce jsou první kroky vedoucí k formalizaci problematiky vyvíjejících se systémů, zahrnující především dynamickou manipulaci s modely a simulacemi v kontextu návrhu systémů s využitím simulace. Přitom byl kladen důraz na důsledné provázání teorie modelování a simulace (reprezentované formalismem DEVS) s praxí v podobě nástrojů SmallDEVS a PNtalk, které byly demonstrovány v několika případových studiích.

Literatura

[ABPD96] Andrea Asperti, Nadia Busi, Piazza Porta, and S. Donato. Mobile petri nets. Technical report, 1996.

[AG04] Norman P. Archer and Fereidoun Ghasemzadeh. Project portfolio selection and management. In J.K. Pinto P.W.G. Morris, editor, *TheWiley Guide to Managing Projects*. Wiley, New York, 2004.

[AM04] K. A. Amin and A. R. Mikler. Agent-based Distance Vector Routing: A Resource Efficient and Scalable approach to Routing in Large Communication Networks. *Journal of Systems and Software*, 71(3):215–227, 2004.

[Anb05] F. Anbari. *Q & As for the PMBOK Guide*. ISBN 1930699395. Project Management Institute, 2005.

[Bar97] F. J. Barros. Modeling formalisms for dynamic structure systems. In *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 1997.

[Bra87] M. E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[BWH07] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[DdBDM03] M. Dastani, F. de Boer, F. Dignum, and J. Meyer. Programming Agent Deliberation: an Approach Illustrated Using the 3APL Language. In *Proceedings of AAMAS '03*, pages 97–104, New York, NY, USA, 2003. ACM.

[dLG+04] M. d'Inverno, M. Luck, M. P. Georgeff, D. Kinny, and M. Wooldridge. The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53, 2004.

[E. 03] E. I. Hsu and D. L. Mcguinness. Wine Agent: Semantic Web Testbed Application. In *Proceedings of 2003 Description Logics (DL2003)*, page 5–7. CEUR-WS.org, 2003.

[FIP01] FIPA. *FIPA ACL Message Structure Specification*. FIPA, 2001.

[GK03] S. Rollins G. Kendall. *Advanced Project Portfolio Management and the PMO*. J. Ross Publishing, Boca Raton, FL, 2003.

[GL87] M. P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, USA, July 1987. Morgan Kaufmann.

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

[GR96] Michael Peter Georgeff and Anand S. Rao. A profile of the australian artificial intelligence institute. *IEEE Expert: Intelligent Systems and Their Applications*, 11(6):89–92, 1996.

[GVH03] B. P. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[Hu04] X. Hu. *A Simulation-Based Software Development Methodology for Distributed Real-Time Systems*. PhD thesis, The University of Arizona, USA, 2004.

[HY03] X.C. Jiao H.S. Yan, Z. Wang. Modeling, scheduling and simulation of product development process by extended stochastic high-level evaluation Petri nets. *Robotics and Computer-Integrated Manufacturing*, 19(4):329–342, 2003.

[HZ04] X. Hu and B. P. Zeigler. Model Continuity to Support Software Development for Distributed Robotic Systems: a Team Formation Example. In *Journal of Intelligent & Robotic Systems, Theory & Application, Special Issue: Multiple and Distributed Cooperating Robots*, pages 71–87, 2004.

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

[Jan98] V. Janoušek. *Modelování objektů Petriho sítěmi*. PhD thesis, FEI VUT, Brno, CZ, 1998.

[Jan06] V. Janoušek. SmallDEVS. http://www.fit.vutbr.cz/~janousek/smalldevs, 2006.

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

[JK04] V. Janoušek and R. Kočí. Towards an Open Implementation of the PNtalk System. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation*. EUROSIM-FRANCOSIM-ARGESIM, Paris, FR, 2004.

[JK05a] V. Janoušek and R. Kočí. PNtalk Project: Current Research Direction. In *Simulation Almanac 2005*. Faculty of Electrical Engineering, Praha, CZ, 2005.

[JK05b] V. Janoušek and R. Kočí. Towards Model-Based Design with PNtalk. In *Proceedings of the International Workshop MOSMIC'2005*. Faculty of management science and Informatics of Zilina University, SK, 2005.

[JK06] V. Janoušek and E. Kironský. Exploratory Modeling with SmallDEVS. In *Proc. of ESM 2006*, pages 122–126. EUROSIS, 2006.

[JK07] V. Janoušek and R. Kočí. Embedding Object-Oriented Petri Nets into a DEVS-based Simulation Framework. In *Proceedings of the 16th International Conference on System Science*, volume 1, pages 386–395. Wroclaw University of Technology, 2007.

[KHS97] H. Kargupta, I. Hamzaoglu, and B. Stafford. Scalable distributed data mining - an agent architecture. In *Proceedings the Third International Conference on the Knowledge Discovery and Data Mining*, page 211–214, Menlo Park, California, 1997. AAAI Press.

[Kit98] H. Kitano, editor. Volume 1395 of Lecture Notes in Computer Science. Springer, 1998.

[Kli85] G. Klir. *Architecture of Systems Problem Solving*. Plenum Press, New Yourk, NY, 1985.

[Koc02] Waldemar Kocjan. Dynamic scheduling state of the art report. Technical Report T2002:28, Dynamic scheduling state of the art report, 2002.

[Kou01] Adamantios Koumpis. The european ist explantech project. *Ubiquity*, 2(36):1, 2001.

[Lee04] B. Lee. Multi-project management in software engineering using simulation modeling. *Software Quality Journal*, 12(1):59–82, 2004.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings. Also Sigplan Notices, volume 21, issue 11*, pages 214–223, 1986.

[LRS01] R. Laddaga, P. Robertson, and H. Shrobe, editors. *Self-Adaptive Software: Applications*. Springer, 2001.

[LTO03] Victor Lesser, Milind Tambe, and Charles L. Ortiz, editors. *Distributed Sensor Networks: A Multiagent Perspective*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[MA04] H. M. Mosner and R. Allen. Prototypes. http://map.squeak.org, 2004.

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

[Maz08] Z. Mazal. *Modelování uvažujících systémů Petriho sítěmi*. PhD thesis, FIT VUT, Brno, CZ, 2008.

[MMWY92] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, SIGPLAN Notices, volume 27, number 10, pages 127–144. ACM Press, New York, NY, 1992.

[Mos01] Multi-agent based simulation: second international workshop. volume 1979 of *Lecture notes in computer science: Lecture notes in artificial intelligence*, Berlin, 2001. Springer.

[MRMZ07] Saurabh Mittal, José Luis Risco-Martín, and Bernard P. Zeigler. Devsml: automating devs execution over soa towards transparent simulators. In *SpringSim '07: Proceedings of the 2007 spring simulation multiconference*, pages 287–295, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[MVR+06] M. Češka, V. Janoušek, R. Kočí, B. Křena, and T. Vojnar. PNtalk: State of the Art. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components, and Agents*, pages 301–307. Hamburg, DE, 2006.

[MVT97] M. Češka, V. Janoušek, and T. Vojnar. PNtalk – A Computerized Tool for Object Oriented Petri Nets Modelling. In *Proceedings of EUROCAST'97*, pages 229–231. Las Palmas de Gran Canaria, ES, 1997.

[MVT02] M. Češka, V. Janoušek, and T. Vojnar. Modelling, Prototyping, and Verifying Concurrent and Distributed Applications Using Object-Oriented Petri Nets. *Kybernetes: The International Journal of Systems and Cybernetics*, 2002(9), 2002.

[MW97] D. Moldt and F. Wienberg. Multi-Agent-Systems Based on Coloured Petri Nets. In *Lecture Notes in Computer Science: 18th International Conference on Application and Theory of Petri Nets*, pages 82–101. Springer-Verlag, 1997.

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

[PBL03] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *EXP – in search of innovation*, 3(3):76–85, 2003.

[Pic89] Franz Pichler. From systems theory to cast. In Franz Pichler and Roberto Moreno-Díaz, editors, *EUROCAST*, volume 410 of *Lecture Notes in Computer Science*, pages 2–6. Springer, 1989.

[Pin02] Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems.* second edition. Prentice Hall, 2002.

[Pin05] Michael Pinedo. *Planning and Scheduling in Manufacturing and Services.* Springer, 2005.

[PNt06] The PNtalk System. http://www.fit.vutbr.cz/~janousek/pntalk, 2006.

[Rao96] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Rudy van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[Ren06] Renew – the Reference Net Workshop. http://www.renew.de, 2006.

[SBD+00] S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press/AAAI Press, Cambridge, MA, USA, 2000.

[Sch05] K. Schwalbe. *Information Technology Project Management, Fourth Edition*. ISBN 1930699395. Course Technology, 2005.

[Skl97] J. Sklenar. Intorduction to OOPN in SIMULA. http://staff.um.edu.mt/jskl1/talk.html, 1997.

[Spi92] M. Spinner. *Elements of Project Management: Plan, Schedule, and Control, 2nd Ed.* Englewood Cliff, NJ, Prentice Hall, 1992.

[TUN06] The TUNES Project for a Free Reflective Computing System. http://tunes.org, 2006.

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

[US87] D. Ungar and R. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227–241, 1987.

[Val98] R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, volume 120. Springer-Verlag, 1998.

[Van00] H. Vangheluwe. *Multi-formalism Modelling and Simulation*. PhD thesis, Faculty of Science, Ghent University (UGent), December 2000.

[WH02] D. Weyns and T. Holvoet. A Colored Petri Net for a Multi-Agent Application. In *Components, Objects and Agents, MOCA02*. Computer Science Department, Aarhus University, 2002.

[Woo02] M. Wooldridge. *Introduction to MultiAgent Systems*. John Wiley & Sons, 2002.

[YC06] Z. Yu and Y. Cai. Object-Oriented Petri nets Based Architecture Description Language for Multi-agent Systems. *International Journal of Computer Science and Network Security*, 6(1B):123–131, 2006.

[Zei84] B. P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press Prof., Inc., San Diego, CA, 1984.

[Zei90] B. P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomporphic Systems*. Academic Press Prof., Inc., San Diego, CA, 1990.

[ZKP00] B. Zeigler, T. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., London, UK, 2000.