VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DISERTAČNÍ PRÁCE

k získání akademického titulu Doktor (Ph.D.)

ve studijním programu (oboru)

INFORMAČNÍ TECHNOLOGIE

Ing. Martin Hrubý

Prostředí pro modelování heterogenních systémů

Školitel: Doc. Ing. Zdena Rábová, CSc.
Datum státní doktorské zkoušky: 24. červen 2002
Datum odevzdání práce: březen 2004
Práce je k dispozici: Vědecké oddělení FIT VUT, Božetechova 2, BRNO

Co se vůbec dá říci, dá se říci jasně; a o čem se nedá mluvit, o tom se musí mlčet.[1] Věta jazyka je nedokonalý obraz původní myšlenky – jazyk klade *odpor* při pokusech se v něm vyjádřit. Markantní je to zejména u jazyků *programovacích.*

Rád bych touto cestou poděkoval své školitelce doc. Rábové za vedení mé práce, předané zkušenosti a velkou podporu. Dále chci vyjádřit svůj dík kolegům dr. Janouškovi, dr. Peringerovi a dr. Vojnarovi za konzultace v době dokončování textu práce.

V Brně, březen 2004 Martin Hrubý

[1] Wittgenstein, Ludwig: *Tractatus logico-philosophicus*

Abstrakt

Předložená disertační práce je zaměřena do oblasti metod a nástrojů pro výstavbu modelů složitých, rozsáhlých a heterogenních systémů. Výstavba modelů těchto systémů je spojena s hledáním vhodných metod a nástrojů umožňujících návrh a realizaci modelů v různém aplikačním prostředí. Vzhledem k tomu, že metody a nástroje modelování systémů na počítačích již tradičně odrážejí vývoj programování a programovacích jazyků, inspiroval jsem se filozofií a různými modifikacemi objektově orientovaného programování, které jsou obsaženy v několika zajímavých objektově orientovaných jazycích.
Výchozím bodem předložené práce je analýza heterogenních systémů. Modelovaný heterogenní systém rozdělíme na několik subsystémů, z nichž každý popíšeme a dále řešíme způsobem, jenž je pro daný subsystém nejvhodnější. Vznikne tak množina submodelů, které pak spojíme do celkového heterogenního modelu. Způsob realizace spojení submodelů a zajištění jejich kooperace není triviálním problémem. Můj návrh jeho řešení patří k původním přínosům práce. Jde o způsob propojování submodelů založený na komunikační síti AIN (**A**utomatic **I**nformation **N**et), na kterou úzce navazuje simulační jazyk HELEF (**He**terogeneous **L**anguage **E**nvironment, **F**inal version), jehož návrh je rovněž původním přínosem mé práce.

Obsah

Kapitola 1

Úvod

Předložená disertační práce je zaměřena do oblasti metod a nástrojů pro výstavbu modelů složitých, rozsáhlých a heterogenních systémů. Výstavba modelů těchto systémů je spojena s hledáním vhodných metod a nástrojů umožňujících návrh a realizaci modelů v různém aplikačním prostředí. Vzhledem k tomu, že metody a nástroje modelování systémů na počítačích již tradičně odrážejí vývoj programování a programovacích jazyků, inspiroval jsem se filozofií a různými modifikacemi objektově orientovaného programování, které jsou obsaženy v několika zajímavých objektově orientovaných jazycích.

Výchozím bodem předložené práce je analýza heterogenních systémů. Modelovaný heterogenní systém rozdělíme na několik subsystémů, z nichž každý popíšeme a dále řešíme způsobem, jenž je pro daný subsystém nejvhodnější. Vznikne tak množina submodelů, které pak spojíme do celkového heterogenního modelu. Způsob realizace spojení submodelů a zajištění jejich kooperace není triviálním problémem. Můj návrh jeho řešení patří k původním přínosům práce. Jde o způsob propojování submodelů založený na komunikační síti AIN (**A**utomatic **I**nformation **N**et), na kterou úzce navazuje simulační jazyk HELEF (**He**terogeneous **L**anguage **E**nvironment, **F**inal version), jehož návrh je rovněž původním přínosem mé práce.

Text práce je členěn do osmi kapitol a dvou příloh.

Druhá kapitola je rozšíředím úvodu o popis vybraných objektově orientovaných jazyků. Jazyky Simula 67, Smalltalk, Self, Beta a PNtalk měly velký vliv na vznik jazyka HELEF prezentovaný v této práci. Více než na podrobný popis těchto jazyků je druhá kapitola orientovaná na zvýraznění jejich zvláštních vlastností a přístupů, které tyto jazyky zavedly. Jsou to například pojmy perzistence, interaktivnost a beztřídní objektová orientace.

Třetí kapitola je analýzou heterogenních systémů a modelů. Ukazuje přínos zavedení heterogenity na *produktivitu práce* při modelování a programování. Řešení heterogenity souvisí s hledáním prvků společných pro všechny subsystémy, které mohou sloužit jako jejich komunikační rozhraní. V závěru kapitoly naznačíme koncepci architektury heterogenního modelu.

Čtvrtá kapitola je jádrem této disertační práce a obsahuje i její hlavní původní přínos. Definuje rozdělení libovolného výpočtu na jeho dvě základní složky – jeho data a jeho procedury, které jsou v terminologii čtvrté kapitoly nazývány simulační objekty typu data a proces. Ukáže se, že simulační objekty lze sestavit do sítě (AIN) propojující objekty podle jejich datových a logických závislostí. Vzniklá síť AIN představuje standardizovanou podobu každého subsystému heterogenního systému – je to paradigma společné všem submodelům zahrnutým do heterogenního modelu, a proto jsou tyto submodely slučitelné do jednoho funkčního celku. Podobnou vlastnost vykazuje i formalismus DEVS a jeho srovnáním se sítí AIN kapitola končí.

Metoda založená na síti AIN řeší metodologické a funkční propojení subsystémů (jejich technické propojení může být realizováno například platformou CORBA). *Pátá kapitola* je zamyšlením nad způsobem abstrahování reality do podoby modelů. Ukazuje na pojmu *znovupoužitelnosti* modelů jeden ze způsobů strukturování modelu do tak zvaných *prostředí* obsahujících *vlastnosti* a *entity*. Cílem kapitoly je návrh metody objektově orientované formalizace objektů reálného světa.

Závěry čtvrté a páté kapitoly tvoří koncept nového heterogenního simulačního jazyka HELEF. HELEF je objektově orientovaný simulační jazyk se sémantikou sítě AIN popisující systémy hierarchií objektů typu prostředí, entita a vlastnost. Jeho popis je náplní *šesté kapitoly*. Vzhledem k celkovému rozsahu práce je definován pouze rámcově, protože vychází ze znalostí o běžných programovacích jazycích.

V závěrečné části práce (*sedmá kapitola* a přílohy) je uvedena sada komentovaných příkladů typických modelů řešených v jazyce HELEF a rozbor jednoho rozsáhlejšího modelu.

1.1 Vymezení cílů práce

1. *Vytvořit koncept a metodologii budování heterogenních modelů.* Heterogenní modely (multi-formalism models) jsou složené ze submodelů, z nichž každý je založen na obecně odlišném paradigma jazyka, formalismu nebo nástroje. Kromě návrhu jednotlivých submodelů je hlavním problémem způsob *jejich propojení*. Předložená práce přináší návrh propojení na základě univerzální jednotné komunikační platformy.

2. *Vytvořit novou metodologii modelování objektů reálného světa.* Práce přináší analýzu struktury systémů do hierarchie prostředí, entit a jejich vlastností. Výhodou předkládané metody je vyšší forma zobecnění popisovaných objektů, která je základním předpokladem *znovupoužitelnosti* již vytvořených modelů.

3. *Navrhnout nový simulační jazyk založený na výsledcích prvního a druhého cíle.* Jejich závěry vedly k návrhu nového simulačního jazyka, který slouží k popisu heterogenních modelů.

1.2 Poznámky k terminologii

- Užívání slova *programování* a *modelování* v různých částech textu – tyto dva pojmy jsou velmi příbuzné a mnohdy zaměnitelné. V případě, že bude použito slovo programování, pak je kladen důraz na obecnější pojetí tohoto pojmu.

- Pro zápis syntaxe jazyků bude používána BNF forma, kde:

  - *Identifikátor* znamená nonterminální symbol gramatiky.
  - *"řetězec"* je terminální symbol gramatiky.
  - *[item]* je nepovinná deklarace řetězce *item*.
  - *{item}* je libovolné opakování řetězce *item*.

- *Simulátor* – počítačový program provádějící simulaci modelu – může být jednoúčelový nebo programovatelný.

- *Heterogenní systém* – systém natolik různorodý, že jej nelze popsat jediným formalismem, nebo systém popsatelný jediným formalismem, jehož výsledný model by byl příliš složitý.

- *Heterogenní model* – model heterogenního systému složený z několika submodelů, kde každý submodel je založen na jiném formalismu, metodě nebo nástroji modelování. V anglické literatuře se používá pro heterogenitu několik termínů – multi-model, multi-paradigm model, multi-formalism model.

- Proces vytváření modelu:

  – *Abstraktní model* – model systému obsahující pouze prvky relevantní pro zkoumání zadaného problému.
  – *Simulační model* – abstraktní model zapsaný formou programu v simulačním jazyce.
  – *Simulace* – experimentování se simulačním modelem, vykonávání programu napsaného v simulačním jazyce.

Kapitola 2

Objektově orientované jazyky

Vzhledem k tomu, že simulační jazyky již historicky odrážejí vyspělost programování a programovacích jazyků, zaměřím se na stručnou charakteristiku jejich vlastností, které ovlivnily můj návrh nového simulačního jazyka HELEF. Bude diskutována především linie jazyků Simula 67, Beta, Smalltalk a Self, které přispěly do vývoje programovacích jazyků zajímavými prvky.

Objektová orientace byla inspirována pozorováním reality a jejím modelováním. Proto také prvním objektově orientovaným jazykem byl simulační jazyk Simula 67. Objektovost rozvedla myšlenku spojit datové atributy a podprogramy (metody) do jednoho celku – objektu. Objektová orientace s sebou přinesla i několik pojmů a pravidel. Například, objektovost povoluje přístup k datům objektu pouze přes volání jeho metod (zapouzdření objektu). Volání metod objektu se v terminologii objektové orientace nazývá *zasílání zpráv* objektu. Uvidíme, že objektová orientace není přesně vymezený pojem a že různé objektově orientované jazyky přispěly k vývoji objektové technologie zajímavým rozšířením, nebo naopak zjednodušením sémantiky pojmu *objektová orientace* (OO).

2.1 Základní dělení OO jazyků

OO jazyky vznikly z neobjektových jazyků přidáním určitého rozšíření – objektovosti. Objektovost je tedy původně *abstrakce* položená *nad* určitým neobjektovým programem. Dalším vývojem OO jazyků se dospělo k jazykům tak zvané čisté objektovosti. Proto rozlišujeme dvě rodiny OO jazyků:

Hybridní OO jazyky

Klasickými představiteli hybridních OO jazyků jsou jazyky C++, Java a Simula 67. Hybridní OO jazyk kombinuje objekty a primitivní datové typy. V těchto jazycích lze napsat i zcela neobjektový program – například v C++.

Zvláštní postavení mezi hybridními jazyky zaujímá již zmíněná Java. Zcela určitě se totiž nejedná o čistý objektově orientovaný jazyk, protože specifikace Javy zavádí primitivní datové typy (čísla, znaky,...). Současně však knihovny Javy poskytují kompletní objektovou alternativu pro všechny primitivní datové typy. Na druhou stranu lze v Javě napsat neobjektový program, protože deklarování třídy aplikace může být považováno za pouhý nutný syntaktický prefix a postfix aplikace.

Čisté OO jazyky

Čisté objektově orientované jazyky můžeme definovat jako jazyky, kde:

1. Kód (procedury) lze psát pouze jako metody tříd nebo objektů.
2. Kterýkoliv element jazyka (včetně konstant) je objektem (nebo instancí třídy).
3. Jediná povolená operace je zaslání zprávy objektu.

Nyní by bylo vhodné rozlišit termíny *objekt* a *instance třídy*. V *třídních* OO jazycích zavádíme koncept tříd a jejich instancí (objektů). Definice těchto pojmů je zřejmá a všeobecně známá [Rumbauch 91, Liu 96]. Existují však i *beztřídní* OO jazyky definující existenci *pouze objektů* – objekt pak zastupuje a sjednocuje úlohu třídy i její instance. Diskuze obou přístupů bude uvedena v následující kapitole.

2.1.1 Třídní a beztřídní OO modelování

V článku [Hrubý ASU02] byla publikována určitá úvaha o vztahu modelování a třídního, resp. beztřídního objektově orientovaného modelování.

Při modelování objektů reálného světa se setkáváme s reálnými objekty, které mohou být zobecněny do podoby třídy a konkretizovány zadáním několika parametrů. Současně však existují reálné objekty, které *nemá smysl* zobecňovat a navíc by to bylo příliš komplikované. Příkladem mohou být složité objekty, které se v realitě vyskytují pouze v jedné kopii.

V podstatě nelze rozhodnout o *kvalitativním* rozdílu v modelování třídním nebo beztřídním. Mluvíme-li o aplikacích objektové orientace v modelování a simulaci, pak můžeme tyto rozdílné styly porovnávat v rámci *aplikovatelnosti* na daný druh simulačního problému. Více o beztřídním modelování najdeme například v [Ungar 91] nebo na stránkách firmy SUN, která podporovala vývoj beztřídního OO jazyka Self [SELF HP].

V beztřídních OO jazycích je objekt prvkem, který nevzniká vytvořením instance třídy – jeho vytvoření nepředchází žádná obslužná akce (konstruktor, operace new). V těchto jazycích nejsou tak ostře ohraničeny pojmy *aplikace* a *běh aplikace*. Pro simulace se nabízí aplikování, kde nesimulujeme dávkovým způsobem, ale více chceme zasahovat do běhu simulace (dále v kapitole 2.8.1).

2.2 Jazyk Simula 67

Simula 67[1] [Benda 78, Kindler 80] je na první pohled zastaralý jazyk; respektive, při pohledu na program v Simule to takto začátečníkovi může připadat. Z dnešního pohledu je však Simula vzorem objektově orientovaného simulačního jazyka a jeho knihoven.

Pro popis jazyka nejsou nyní podstatné konstrukce cyklů a větvení – ty jsou převzaty z jazyka Algol 60. To zajímavé je pojetí struktury simulačního modelu a objektové orientace jazyka (vytváření instancí tříd, vnořené třídy). Třídu deklarujeme podle vzoru, který se pak projevil i v návrhu jazyků Beta a Self:

```
class Name [Arg]
-- deklarace procedur, vnořených tříd a proměnných
```

[1] Existují dva jazyky: Simula I. a Simula 67. Ta hlavní, přelomová je Simula 67. Dnes se obvykle nazývá zkráceně “Simula”. Pokud bude tedy v textu uvedeno pouze “Simula”, pak je míněna právě Simula 67.

```
begin
-- operační část třídy
end
```

Standardní knihovny Simuly obsahují pouze základní prostředky pro kvaziparalelní vykonávání procesů. Další simulační konstrukce si musí uživatel vyrobit sám. Příkladem rozšíření simulační knihovny Simuly je knihovna DEMOS [DEMOS], kterou se inspirovala i standardní simulační knihovna Smalltalku 80 [Goldberg 89].

2.3 Jazyk Smalltalk

Smalltalk [Smalltalk HP] vznikl v 70-tých letech ve výzkumných laboratořích firmy Xerox. Během vývoje Smalltalku vzniklo i několik (dnes důležitých) součástí uživatelského rozhraní – počítačová myš a grafické uživatelské rozhraní s okny, tlačítky a podobně. Ve své době se však příliš nerozšířil pro své relativně vysoké nároky na výkon počítače a jeho grafického modulu. Smalltalk je totiž *interpretovaný jazyk* – zdrojový kód se přeloží do vnitřního jazyka tak zvaného virtuálního stroje[2] a tím je posléze vykonáván.

Jazyk Smalltalk je představitelem tak zvaného čistého OO programování. Každý element jazyka představuje objekt, včetně čísel a znaků. Překladač Smalltalku je úzce propojen se standardními knihovnami Smalltalku a z například číselných konstant přímo generuje instance tříd *Number*.

Smalltalk byl dlouhou dobu pouze výzkumným projektem nepřesahujícím hranice laboratoře (podobně jako například Unix v Bell Labs.). Až v roce 1980 dosáhl standardu ve verzi Smalltalk 80 [Goldberg 84, Goldberg 89]. Dnešní vývoj komerčního Smalltalku vede firma CINCOM se svým špičkovým produktem Cincom Smalltalk (dříve Visual Works firmy ParcPlace) [VisualWorks HP]. V posledních letech také probíhá vývoj volně šířitelného Smalltalku pod názvem Squeak Smalltalk [Squeak HP].

Smalltalk definuje třídy, proměnné, metody a objekty. Z hlediska Smalltalku je třída *metaobjekt* (objekt popisující jiný objekt). Smalltalk proto zná pouze objekty. Některé z nich slouží pro popis jiných objektů (metaobjekty ⇒ třídy).

Jako ukázku uvedeme příklad jedné ze základních programových konstrukcí:

```
fce: n “hlavička funkce”
n isNil “isNil vrací True, je-li n objekt Nil”
    ifTrue: [^false] “objekty True a False
                  obsahují metodu ifTrue:ifFalse: “
    ifFalse: []
```

Při tomto přístupu k výpočtu je samozřejmě zcela nemožná typová kontrola v době překladu a je dokonce pro styl programování ve Smalltalku naprosto nevhodná. Absence typové kontroly se považuje za základní vlastnost systémů určených pro rychlý návrh aplikací – případně k *prototypování aplikací*.

Polymorfismus Smalltalku je dotažen do zajímavého stavu, který oceníme zejména při implementaci matematických operací a čísel. Ve Smalltalku lze počítat bez ztráty přesnosti – každá číselná operace vrací objekt obecně jiné třídy, například operace podílu nad celými

[2] V dnešní době je interpretovaný jazyk již běžným jevem. Bariéru předsudků o interpretovaných jazycích překonal před několika lety programovací jazyk Java při masivním nasazení v internetových aplikacích. Logické a funkcionální jazyky jsou pochopitelně také interpretované, ale jejich interpretovanost a situaci s jazyky typu C a Pascal poněkud odlišujeme.

čísly může vrátit objekt `Fraction` (zlomek), který se po jiné operaci může stát celým číslem, případně reálným číslem s omezenou přesností (`Float`). To, co se v jazycích typu C++ komplikovaně definuje jako *přetížení operátoru*, je ve Smalltalku běžné.

Interaktivnost

Různé jazyky se liší různým přístupem uživatele k vytváření aplikací. Kromě následujícího sledu kroků existuje i tak zvaný přístup *interaktivní*. Uvažujme běžný přístup k programování, daný kroky:

1. Zápis programu (editování textu).
2. Přeložení programu, odstranění syntaktických chyb hlášených překladačem.
3. Běh aplikace, projevují se logické chyby návrhu programu.
4. Analýza výsledků (může být znehodnocena logickou chybou programu).

Běžné návrhové programátorské prostředí (Borland C++) by mělo umožňovat ukázat na místo kritické chyby při běhu programu, v programátorské praxi to však nemusí vždy nutně platit. To značně komplikuje ladění aplikace. Interaktivní prostředí (Smalltalk, Prolog,...) dovolují bez běhu aplikace ladit pouze její určitý segment kódu.

Každý objekt systému Smalltalk (publikace o aplikacích ve Smalltalku: [Goldberg 84, Liu 96]) lze zobrazit do okna zvaného *Object Inspector*. Inspector ukazuje seznam instančních proměnných s jejich hodnotami a obsahuje i příkazovou řádku pro sestavení zprávy pro objekt. Tím, že v interaktivním prostředí Smalltalku není tak ostrý rozdíl mezi běžící a neběžící aplikací, má i objekt při ladění své aplikační okolí. Aplikaci je možné za chodu naprosto *libovolně modifikovat* – včetně strukturálních zásahů do tříd. Prostředí Smalltalku obsahuje pro účely ladění i silné ladící nástroje (debugger) umožňující modifikovat kód právě probíhajících metod.

Reflektivita

Za velkou výhodu Smalltalku se považuje kompaktnost jazyka Smalltalk s jeho standardními knihovnami. Ty byly navrženy tak, že obsahují metody pro základní operace nad primitivními objekty jako jsou například čísla. Uživatel smí kdykoliv modifikovat libovolnou metodu třídy v jádru knihovny, která má na běh prostředí Smalltalku klíčový vliv[3]. Dokonce i překladač Smalltalku je tvořen sadou tříd v knihovně. Uživatel je tak schopen kdykoliv a okamžitě modifikovat syntaxi a sémantiku Smalltalku. Tato dynamičnost programovacího jazyka je typickým projevem *reflektivnosti*.

V současné době existuje několik projektů rozšiřování prostředí Smalltalku jeho obohacováním o jiné jazyky. Pak je možné psát metody jedné třídy v různých jazycích, což vede na tak zvané multi-paradigm jazyky (nebo heterogenní systémy).

Perzistence ve Smalltalku (a Selfu)

Při vývoji Smalltalku byl kladen důraz na otevřenost systému, znovupoužitelnost vytvořeného kódu a vlastnost zvanou *perzistenci* – objekty v systému jsou ukládány na trvalém paměťovém médiu (disku), a proto je předchozí stav systému při spuštění prostředí Smalltalku

[3] V kontextu jazyků Smalltalk a Self se nehovoří o bezpečnosti jako například v C++ (metody s atributem *protected*). Filozofie práce ve Smalltalku předpokládá u programátora jiný přístup.

znovu obnoven (systém má charakter databáze[4]). V systému Smalltalk jsou i procesy implementovány formou objektů (Smalltalk je prostředí s preemptivním multitaskingem) a objektem systému je i řízení přepínání procesů. Proto po znovuobnovení běhu Smalltalku začnou pracovat i (perzistentní) procesy. Vypnutí počítače tedy nezpůsobí žádnou zásadní změnu systému.

2.4 Jazyk Self

Jazyk Self byl velkou inspirací jazyka HELEF, který je prezentován v této práci. Self [SELF HP, Chambers 92] vznikl dalším vývojem systému Smalltalk, kdy se hledala forma Smalltalku bez zavádění tříd. Nakonec vývojáři Smalltalku z firmy SUN oddělili tento jazyk od vývoje Smalltalku a pojmenovali ho *Self*. Self je bohužel šířen pouze na platformách počítačů SUN (a omezeně na počítačích Apple MacIntosh) a byl používán firmou SUN k prototypování distribuovaných aplikací a pro týmový vývoj aplikací.

Objekt v Selfu je dvojice (S, E), kde

- S je seznam *pojmenovaných* referencí na jiné objekty (pojmenovaná reference na objekt se v Selfu nazývá *slot*).
- E je nepovinně uvedený výraz. Je-li výraz E v objektu definován, pak objekt má funkci metody a vrací nějakou hodnotu.

Stejně jako u Smalltalku, tak i v Selfu se pracuje s jedním blokem objektové paměti – tak zvaným *image*. S tím souvisí i perzistence všech objektů vytvořených při práci se systémem Self. Navíc tím, že objekt je současně i nositelem kódu, lze jej kdykoliv exportovat do textové podoby v jazyce Self a přenést na libovolný jiný systém Self.

Self je dobré studovat od nejjednodušších akcí – vytvoření objektu, zasílání zpráv a podobně. Kromě toho, že Self je textový programovací jazyk, existuje i grafické vývojářské perzistentní prostředí Selfu, které usnadňuje programování aplikací.

Syntaxí i sémantikou je Self velmi podobný Smalltalku. Self redukuje operace na jedinou a tou je *zaslání zprávy objektu*. Příjemcem zprávy je implicitně objekt sám – tedy *self*. Mějme objekt *obj*:

```
obj = ( | x <- 10 | x + 1 )
```

Vznikne objekt *obj* obsahující slot x s referencí na objekt 10. Vlastní kód objektu *obj* tvoří výraz $x + 1$. Hodnota objektu *obj* se získá posloupností zasílání zpráv (zjednodušeně):

1. Zašli zprávu x objektu $self$.
2. Objekt *obj* vyhledá slot x a vrátí jeho hodnotu 10.
3. Objektu 10 se zašle zpráva + s parametrem objekt 1.
4. Objekt 10 zpracuje zprávu (podobným postupem) a vrátí objekt s hodnotou výsledku. Ten je předán jako výsledek vyčíslení hodnoty objektu *obj*.

Instanční proměnné i metody jsou v jazyce Self dostupné v podobě tak zvaných slotů. Existence slotu `x` znamená, že objekt rozumí zprávě `x`. Pro objekt je to jednoduchá zpráva, jejímž vyhodnocením je objekt `10`. V případě deklarování operátoru `<-` u slotu, objekt rozumí i klíčové zprávě `x: value`, která do slotu zapíše referenci na objekt `value`.

[4]Dokonce existuje varianta Smalltalku zvaná Gemstone, která je speciálně upravená pro možnou správu obrovského množství objektů tak, že se Gemstone používá jako objektově orientovaná databáze.

Delegace

Aby bylo v Selfu možné prakticky programovat, zavádí Self mechanismus tak zvané *delegace*. Je to beztřídní obdoba dědičnosti [Ungar 91]. Určitá podmnožina slotů objektu je označena jako delegující odkaz na jiné objekty. Obdrží-li objekt $o = (S, E)$ zprávu msg, a $D \subseteq S$ jsou sloty delegující na jiné objekty, pak objekt o prochází jména slotů v tomto pořadí:

1. Hledej msg v $o(S)$.

2. $\forall i \in D$: hledej msg v $i(S)$.

3. Dál pokračuj *hledáním do šířky* ve stromu delegací.

Tato primitivní technika, společně se sémantikou zasílání zpráv, umožňuje beztřídnímu objektovému jazyku emulovat třídy, respektive vytvářet objekty dodávající jiné skupině objektů množinu společných atributů (metod). V terminologii Selfu se emulace tříd nazývají *traits* a jsou to objekty obsahující pouze metody a konstanty. V praxi to znamená, že uživatel prototypuje objekt na ploše vývojového prostředí – objekt obsahuje sloty proměnných a metod. V určitém okamžiku je prototyp hotov a uživatel jej může transformovat na úroveň instancovatelné třídy. Veškeré sloty metod přemístí do nového objektu T, v původním objektu O zbývají proměnné. Do O vloží delegační slot na T a do T vloží delegační slot na `traits clonable`. Objekt O pak rozumí zprávě `copy` a po jejím obdržení vytvoří svůj přesný klon. Emulovaná třída v Selfu je pak dvojice objektů – objekt metod a objekt instančních proměnných. Instancí je kopie objektu instančních proměnných s odkazem na objekt metod.

Sémantika Selfu sice představuje naprosté minimum možné sémantiky programovacího jazyka, ale výpočet virtuálním strojem se děje v neuvěřitelném množství kroků, kde se musí neustále prohledávat prostor jmen objektů. Interpretace kódu by byla takto velmi pomalá. Řeší se to zaváděním různých optimalizací do virtuálního stroje [Chambers 92]. Cenou za to je překomplikovaný kód programu virtuálního stroje a tím i omezená přenositelnost jazyka Self mezi různými platformami počítačů. Proto se Self příliš nerozšířil mezi programátory i přesto, že se jedná pravděpodobně o stále platný vrchol vývoje objektově orientovaného programování. Nadějný je vývoj projektu Squeak Smalltalk, který směřuje k Selfu a pak projektu *Self4Linux* [Self4Linux HP] – volně šiřitelného otevřeného prostředí jazyka Self v operačním systému Linux.

Pro mou práci představuje Self velkou inspiraci. Ukazuje OO programování zcela neobvyklého stylu. Pokud bychom chtěli ohodnotit kvalitu Selfu podle počtu jeho příznivců, pak je Self mimořádně neúspěšný. Problematická je dostupnost jeho implementace.

2.5 Jazyk Beta

Jazyk Beta [BetaBook, Beta Mjolner, Beta HP] je následovníkem úspěšného jazyka Simula 67. Beta vznikla na univerzitě Aarhus v Dánsku a jejím spoluautorem byl Kristen Nyygard (jeden z tvůrců Simuly 67).

Hlavními rysy Bety jsou silná typová kontrola a redukce veškerých jazykových konceptů do jediného, tak zvaného *vzoru* (pattern). Vzorem lze vyjádřit třídu, objekt i metodu. Výpočetní výraz reprezentuje aplikování jednoho vzoru na druhý. Beta je protipólem dvojice Smalltalk-Self. Její mimořádnost je ve vyšší úrovni abstrahování algoritmů na vzory algoritmů – podobné vlastnosti nacházíme ve funkcionálních jazycích (LISP, Haskell).

Beta je založena na pojmu *vzor*. Kromě elementárních vzorů dodávaných jazykem (integer, string) může uživatel deklarovat své vlastní vzory:

- Statické a dynamické instance vzorů.
- Vzory tvořící třídy (strukturovaný vzor).
- Vzory tvořící metody (obyčejné, virtuální) – vzor má definovaný vstup, výstup a tělo.

Ukázka demonstruje převzatý princip dědění *inner* od jazyka Simula:

```
twice: (# do inner; inner; #);
...
do
   twice (# do 'hello'->print #)
```

Na rozdíl od Simuly, jazyk Beta povoluje vícenásobný výskyt slova `inner` v těle metody (v Simule nemůže být `inner` ani v těle cyklu). Je zadán vzor `twice` a v těle nadřazeného bloku je odvozen nový nepojmenovaný dočasný vzor odvozením z `twice`, kde je dodáno tělo `do` vzoru.

Jazyk Beta velmi vybočuje ze stylu programování podobného jazykům C a Pascal. Snad proto nemá tolik uživatelů. Schopnost jazyka Beta zobecňovat algoritmy [Madsen 00] je však velmi inspirující, stejně jako vytváření dočasných vzorů a redukovaná syntaxe.

2.6 Jazyk PNtalk

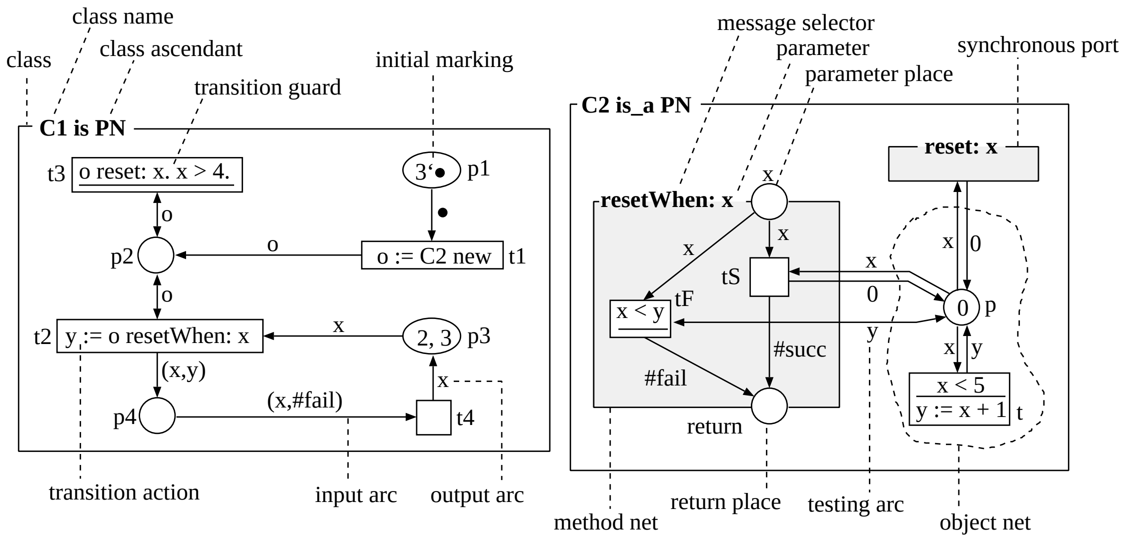
Jazyk PNtalk [PNtalk] je tak zvaný jazyk objektově orientovaných Petriho sítí (OOPN – Object Oriented Petri Nets). OOPN patří do třídy Petriho sítí vyšší abstrakce, podobně jako například barvené sítě (Colored Petri Nets). Základem je myšlenka skloubit modelování systémů s programováním – Petriho síť může být využita při běžném programování jako kostra programu zavádějící paralelismus.

Omezujícím faktorem v Petriho sítích byla jejich relativní *statičnost* – síť se nemění, mění se pouze její stavy (kombinace značení v místech sítě). OOPN umožňují definovat Petriho síť jako *třídu*, ze které lze známým způsobem generovat instance. Značky putující v síti jsou obecné objekty, včetně objektů instancovaných z tříd OOPN. Obrázek 2.1 ukazuje příklad třídy v OOPN.

Jsou-li Petriho sítě základním formalismem pro popis diskrétních systémů a speciálně paralelních procesů v diskrétních systémech, pak je jazyk PNtalk realizací Petriho sítí v počítačové simulaci založené na Petriho sítích. Nebudu zabíhat do podrobností definice PNtalku, ta je uvedena v [PNtalk]. Mým cílem bylo ukázat plnohodnotnou alternativu k simulačním modelům ve formě textu.

PNtalk má dvě funkční implementace. Kompletní implementace simulačního stroje a uživatelského rozhraní byla provedena v jazyce Smalltalk [PNtalk HP][5]. Druhá, spíše experimentální implementace, je realizována v jazyce Prolog a slouží pro vedení automatizovaných verifikací modelů v PNtalku. Později se stala základem vývoje distribuované verze PNtalku.

[5] V současné době probíhají práce na nové verzi interpretu PNtalku obohaceného o nové zkušenosti se simulačním modelováním [Janoušek 02].



Obrázek 2.1: Příklad modelu v jazyce PNtalk

Poznámka: Petriho sítě v diskrétním modelování

Petriho sítě neslouží přímo k budování simulačních modelů, ale především k vytváření silně formálních abstraktních modelů určených k automatizované formální analýze a verifikaci. Je to umožněno jednoduchou syntaxí a sémantikou Petriho sítě. Je třeba zdůraznit, že Petriho sítě v základní podobě (P/T) neuvažují modelový čas, je však možné je o modelový čas doplnit a tím zefektivnit využití Petriho sítí v simulaci.

Petriho sítě s časem

Časované Petriho sítě (Time(d) Petri Nets) jsou rozšířením Petriho sítí o modelový čas. Modelem akce (změny stavu) je v Petriho sítích *přechod*. Pokud chceme modelovat, že akce *trvá* jistou dobu, pak je nutné tuto informaci do přechodu vložit. Existují dvě základní možnosti provedení časovaného přechodu:

Uvažujme síť (definice v [Češka 98]) $S = (\{p_1, p_2\}, \{t_1\}, \{(p_1, t_1), (t_1, p_2)\}, (1, 0))$ a Δ_{t1} je zpoždění na přechodu t_1. V čase τ_0 je proveditelný přechod t_1.

1. Přechod t_1 vnitřně odpočítává dobu Δ_{t1} a v čase $\tau_0 + \Delta_{t1}$ realizuje přechod na značení $(0, 1)$. Po celou dobu odpočítávání času je akce přechodu t_1 přerušitelná.

2. Přechod t_1 odstraní značku z p_1 a v čase $\tau_0 + \Delta_{t1}$ ji přenese do p_2. Provádění přechodu není přímo přerušitelné. Tento způsob interpretace změny stavu systému není v dokonalém souladu s chápáním diskrétních systémů, kde ke změně stavu má dojít skokem v nulovém čase.

Je vidět, že zavedení času do Petriho sítí je komplikované a tím je i komplikované budování simulátorů založených na Petriho sítích.

2.7 Simulační knihovna SIMLIB/C++

Simulační knihovna SIMLIB/C++ [SIMLIB HP] vznikla na bývalém UIVT FEI VUT, který je dnes Ústavem inteligentních systémů FIT VUT[6]. Knihovna SIMLIB/C++ se využívá především pro experimentovaní s tvorbou simulačních modelů a ve výuce kurzu Modelování a simulace. Knihovnu SIMLIB/C++ zmiňuji proto, že je typickým představitelem využití univerzálního programovacího jazyka (C++) k účelům simulačního modelování a přizpůsobení univerzálního programovacího jazyka pro modelování (dále je to rozvedeno v kapitole 3).

Knihovna SIMLIB/C++ poskytuje prostředky pro modelování diskrétních, spojitých i kombinovaných systémů. V kapitole 3.4.1 je uveden příklad propojení SIMLIB s jazykem Prolog, což je vhodná demonstrace heterogenního propojení. Podobných knihoven existuje ve světě mnoho, zmiňme alespoň simulační knihovnu SIMPACK [SIMPACK] vytvořenou známým odborníkem v simulacích - Paulem Fishwickem[7].

V mé práci jsou některé fragmenty modelů psaných pro SIMLIB používány jako referenční.

2.8 Vlastnost perzistence

Na závěr této kapitoly se ještě vraťme k vlastnosti perzistence popsané společně s jazykem Smalltalk a Self. Uvědomme si hlavní výhodu perzistentního výpočetního (a aplikačního) prostředí typu Smalltalk vedle současných, například operačních, systémů:

Mějme aplikaci, která po svém skončení potřebuje zachovat jedno celé číslo (například integer) pro své další spuštění. V běžném operačním systému (Unix, Windows) k tomu potřebuje provést tyto kroky:

1. Určit jméno pro soubor a jeho umístění v adresářové struktuře.
2. Otevřít soubor pro zápis, ověřit jeho bezchybné otevření.
3. Zapsat číslo v nějakém formátu, ověřit zápis.
4. Soubor zavřít, ověřit bezchybné ukončení práce se souborem.

V dalším spuštění aplikace se provedou tyto kroky:

1. Nalezení cesty k souboru s číslem.
2. Otevření souboru, ověření jeho existence (mohl být třeba smazán).
3. Načtení dat, ověření platnosti formátu, ověření platnosti dat (rozsah hodnot).
4. Uzavření souboru.

Perzistentní aplikační systémy typu Smalltalk a Self tento problém neřeší, protože každá aplikace má svá data zahrnuta v paměti objektů systému Smalltalk až do okamžiku, kdy se je rozhodne sama zrušit. Systémy Smalltalk a Self ještě zcela nepřerostly do podoby operačních systémů, které by pokrývaly celý rozsah používání svého počítače, ale v současné

[6] WWW stránka FIT VUT: `http://www.fit.vutbr.cz/`
Domovská stránka Petra Peringera (SIMLIB): `http://www.fit.vutbr.cz/~peringer/`
[7] Domovská stránka Paula Fishwicka: `http://www.cise.ufl.edu/~fishwick/`

době se již o tom hovoří (projekt TUNES[8]). Zejména systém Squeak Smalltalk [Squeak HP] dnes určuje další vývoj jazyka Smalltalk. Pro výhody perzistentního prostředí byl jazyk HELEF implementován ve Squeak Smalltalku a z uváděných perzistentních systémů mnohé převzal.

2.8.1 Perzistence a interaktivnost v simulačních modelech

V základní podobě přistupujeme k modelování jako k dávkovému zpracování dat – vytvoříme simulační model, spustíme jeho simulaci a obdržíme výsledky. Simulace je stejný běh programu jako u kterékoliv jiné aplikace. Simulační modely mohou obsahovat stejné typy chyb a musí se ladit.

Je jasné, že kvalitní výsledky simulace získáme pouze z korektního experimentu se správným simulačním modelem. Pro *účely ladění* můžeme do modelu strukturálně a funkčně zasahovat v průběhu simulace a tak jej *doladit* (tady nacházíme první typ uplatnění interaktivního návrhu aplikace). V poslední fázi práce provedeme nepřerušený simulační experiment a získáme z něj data.

Druhým případem potřeby interaktivnosti je problém dlouhodobých simulací obsahujících cenná data. Tyto programy prostě nemůžeme destruktivně přerušit a pak modifikovat. Někdy je to však nutné - například pro účely rozšíření modelu. Uvažujme simulace rozvodných sítí elektřiny, vodohospodářské simulace, simulace nebezpečných fyzikálních procesů (jaderné reaktory), meteorologické simulace a dopravní simulace (v anglické literatuře se pro ně užívá pojem on-line simulation). Tyto aplikace mají společnou motivaci – existují souběžně s reálným časem a slouží pro podporu řízení reálného systému, k potvrzování hypotéz a k predikci budoucnosti na základě sledování vývoje přítomnosti. V těchto simulačních aplikacích je použití interaktivního vývojového prostředí velmi přínosné.

Pojem *model* v perzistentních interaktivních beztřídních OO jazycích

Pojetí simulačního modelu v perzistentních interaktivních OO jazycích je velmi zajímavé. Jedná se o prototypové vytváření simulačních modelů a modelování systémů s podporou jejich částečné (postupné) simulace.

Vycházejme z představy perzistentního interaktivního beztřídního OO prostředí - například jazyka Self. Simulační model je sítí provázaných objektů. Na rozdíl od jazyků zavádějících třídy, v beztřídních jazycích pracujeme výhradně s *existujícími perzistentními objekty*. Síť objektů v objektové paměti prostředí Selfu je přesnou reprezentací textu programu v jazyce Self a naopak. Každý krok vykonávání našeho simulačního modelu je popsatelný funkcí transformující model A na model B (model systému popisujeme jako množinu objektů OS a jejich vazeb R):

$$SimulationStep : Model \mapsto Model$$

$$Model = [OS, R]$$

Považujeme-li za simulační model text programu v simulačním jazyce a současně síť objektů v Selfu lze kdykoliv exportovat do textové podoby, pak kterýkoliv krok vykonávání běhu simulace lze exportovat do textu programu – simulačního modelu. Každý krok běhu simulace $SimulationStep : A \mapsto B$ dává opět fundamentálně totožný simulační model B jako předcházející A (s výjimkou modelového času). Takto jsme schopni evolucí simulačního modelu dojít do stavu modelu, který prohlásíme za startovní model pro korektní provedení

[8]Domovská stránka projektu TUNES: http://www.tunes.org.

finálního simulačního experimentu. V podstatě náš model vyvíjíme pomocí metod simulace. Tato vlastnost simulačního prostředí je podmíněna beztřídní objektovou orientací. V zásadě by to bylo emulovatelné i v jiných prostředích, ale asi mnohem komplikovaněji.

Ještě jednou bych rád zdůraznil, že vztah mezi textovým programem v jazyce Self a objektovou sítí z něj vzniklou, je izomorfní. Tento izomorfismus není až tak platný v případě třídních objektově orientovaných jazyků. Opět to ovšem nijak nepreferuje beztřídní přístup a vše je dáno *druhem aplikace* a navíc i pocitem uživatelského komfortu. Tento přístup k ladění modelu nemusí být absolutně platný pro všechny typy modelovaných problémů, přesto je myšlenka interaktivního budování simulačních modelů zajímavá a je jednou z nosných myšlenek vývoje jazyka HELEF.

Kapitola 3

Heterogenní modelování

Kapitola uvede pojmy heterogenní systém, heterogenní model a heterogenní modelování. Začíná uvedením motivací pro heterogenní modelování, analýzou typických heterogenní systémů a příkladů jejich heterogenních modelů. Cílem kapitoly je uvést teorii heterogenních modelů a poukázat na problémy jejich realizace.

Cílem heterogenního modelování[1] je rozklad systému do různých charakteristických tříd – subsystémů, aby bylo možno každou třídu modelovat samostatně s využitím pro ni charakteristického formalismu. Výsledné submodely mohou být opět zapouzdřitelné a instanciovatelné a mohou spolu *komunikovat*. Je proto vhodné kombinovat heterogenní a objektově orientovaný návrh modelů.

Tato práce byla inspirována myšlenkou výběru vhodné metody modelování pro zadaný problém – řekněme mírou odporu, který nám určitý formalismus (například simulační jazyk) klade při pokusech se jeho prostřednictvím vyjádřit. Zřejmě se pokusíme vždy použít ten nejvhodnější přístup, spojený s nejnižší mírou kladeného odporu. Určité úlohy mají své charakteristické metody řešení. Pro popis procesů a jejich vzájemné synchronizace použijeme pravděpodobně Petriho síť. Disertační práce dr. Janouška [PNtalk] ukazuje, že je možné Petriho sítí normálně *programovat*. Popis logických úloh provedeme logickým programovacím jazykem typu Prolog. Pro řešení diferenciálních rovnic najdeme spojitý simulační jazyk a podobně. Tyto metody a nástroje jsou z větší části jednoúčelové, a je proto potřeba realizovat jejich propojení.

3.1 Motivace: zvýšení produktivity práce v programování a modelování

Důležitým aspektem při programování je produktivita práce [Hrubý 03]. V této kapitole bych chtěl ukázat vztah mezi pojmy *univerzálnost nástroje* a *produktivita práce*, které jsou do jisté míry protichůdné. Předpokládáme, že pracujeme s nástroji, které mají modelovací sílu Turingova stroje, kde nejsme modelovací silou omezováni. Nacházíme však omezení *praktické*. Začněme následující úvahou na základě předpokladu, že požadujeme *maximálně univerzální simulační jazyk*. Dospějeme pouze ke kompromisnímu řešení – využití heterogenního simulačního jazyka.

[1] V anglické odborné literatuře se (kromě pojmu *heterogeneous modelling*) spíš setkáme s termíny *multi-modelling*, *multi-paradigm* a *multi-formalism modelling*, ale ve všech případech se jedná o různě specifické pohledy na *heterogenitu*.

Sledujme *hledisko vynaloženého úsilí* pro vytvoření programu. Bylo by asi zbytečné vymýšlet *objektivní* jednotku práce při programování, protože vynaložené úsilí je vždy subjektivní a odvislé od schopností programátora. Produktivitu programování lze však smysluplně porovnávat mezi různymi jazyky. Rozhodující je druh aplikace a použitý jazyk. Jeden problém popíšeme v různých jazycích s různými nároky na vynaložené úsilí.

Můžeme rozlišit tyto třídy jazyků:

- *Univerzální* programovací jazyky (C/C++, Java, Pascal, Smalltalk, ...). Tyto jazyky budeme považovat za referenční při posuzování náročnosti programovací práce.
- *Logické a funkcionální* – matematicky orientované jazyky. Funkcionální jazyky se používají i v oblastech s velmi vysokým nárokem na formální přesnost v kódování, v umělé inteligenci a jako jazyky pro programování rozšiřitelných systémů (např. AutoCAD a LISP).
- *Databázové a dotazovací* jazyky jsou určeny pro manipulaci s daty (SQL), včetně jejich rozšíření na univerzálnější programování (programovací jazyky databázových návrhových systémů – např. český PCFand). Ty bývají tak úzce spojeny se systémem řízení databáze, že jejich nasazení jinde, je prakticky nemyslitelné. Existují i speciální jazyky pro formulování dotazů při metodách známých jako dolování dat (data mining).
- *Jazyky pro programování HW* určené pro modelování a návrh elektronických obvodů (VHDL, Verilog). Obvykle bývají spojovány i se simulováním navržených obvodů. Z dnešního pohledu sem patří strojově orientované jazyky (assemblery).
- *Prototypovací* jazyky pro extrémně rychlý návrh aplikací – typicky spojené s návrhem grafického uživatelského rozhraní.
- *Skriptovací* jazyky mohou patřit i mezi prototypovací jazyky. Jsou určeny pro zápis dávek příkazů v řádkově ovládaném operačním systému (typicky Unix a shell) a jednoduchých aplikací. Můžeme sem řadit i jazyky Perl a Python.
- *Simulační a speciální modelovací* jazyky. Simulační jazyky jsou v této práci rozebírány podrobněji. Speciální modelovací jazyk je například VRML (jazyk pro popis fyzických objektů ve virtuální realitě) nebo UML.
- *Jazyky pro uložení dat a formátování textu.* Je to v dnešní době populární jazyk XML pro uložení strukturovaných dat, hypertextový formátovací jazyk HTML, typograficky orientovaný TeX a LaTeX.

Smyslem tohoto výčtu bylo ukázat vztah mezi specializovaným jazykem navrženým pro zjednodušení programování jednoho druhu aplikace a univerzálním jazykem určeným pro neomezené programování. Univerzální jazyky bývají často doplňovány o *knihovny* (již hotové aplikace zobecněné na sadu funkcí).

Knihovna rozšiřuje univerzální jazyk o další funkce a jazyk tak specializuje (třeba na simulační nebo databázový). Nikdy však neposkytuje takový komfort jako specializovaný jazyk, neboť knihovní funkce obvykle vyžadují zavedení spousty datových typů, konstant, velké škály obtížně zapamatovatelných funkcí se spoustou parametrů a podobně (příklad – knihovny WinAPI a OWL pro psaní aplikací na MS-Windows OS v C++).

Tušíme, že univerzálnost programovacího jazyka je diskutabilní. Kromě výhod shrňme i její nevýhody:

- Nutná investice do výroby knihovny. Knihovnu implementujeme s ohledem na její *vlastní univerzálnost.*
- Nutná znalost knihovny a zápis komplikovanějšího kódu.

Obecně vzato, knihovna nemusí být vždy přínosem co do čitelnosti kódu a co do nákladů na pořízení programu. Stojíme nyní před dilematem:

- Investovat úsilí do vývoje knihovny v univerzálním programovacím jazyce a do nepřirozeného kódu programu.
- Riskovat omezenou aplikovatelnost specializovaného jazyka a kódovat elegantně přímo v něm.

Můžeme ještě omezit okruh našich aplikací na počítačové simulace. I v těch najdeme rozsáhlou množinu velmi vzdálených problémů – od virtuální reality (kde je doménou grafika a prostorové modelování) po simulaci fyzikálních polí. Jazyk Simula 67 může být považován za univerzální simulační jazyk. Při dalších úvahách však zjistíme, že jeho univerzálnost končí stejně, jako u univerzálního programovacího jazyka.

Jakési univerzálnosti s minimálním dopadem na produktivitu dosáhneme kombinací kódů ve více jazycích. V jednom programu budou různé jeho části implementovány různým stylem. Každá část bude reprezentovat efektivní přístup k řešení daného podproblému.

Když tuto myšlenku zobecníme, dostáváme pravý heterogenní systém, kde každou jeho část popisuje původní autonomní programovací jazyk nebo formalismus. Zřejmě tak dosáhneme nejvýhodnějšího poměru cena/výkon při vytváření našich aplikací. Důležitou výhodou je znovupoužitelnost již hotových programů – v heterogenním prostředí je totiž nemusíme přepisovat.

Dostáváme se k problému *realizace heterogenního prostředí*, které by mělo být schopno zapojit libovolný jazyk nebo nástroj s minimálními nároky na jeho přizpůsobení a zvýšení produktivity programování.

3.2 Pojmy v heterogenitě

Prof. Tuncer Ören[2] ve svém článku [Oren 02] *Future of Modelling and Simulation: Some Development Areas* naznačil budoucí směry vývoje oboru modelování a simulace a postavil heterogenní přístupy na první místo. V kategorii *Science, methodology and technology of modelling and simulation* vyjmenoval tyto termíny (Článek je převzat a přeložen. Zdůrazním termíny relevantní pro tuto práci):

1. Multimodels – seskupení úzce svázaných *modulů* jednoho modelu. Běh simulace přepíná mezi jednotlivými moduly tak, že jeden je vždy *aktivní.*
2. Multiaspect, multistage, multiperspective, multiresolution, and multiparadigm modelling – multiaspect model je speciální případ multimodelu, kdy více modulů může existovat simultánně. *Multiresolution* modelování je přístup, kdy se pro jeden problém pořídí více různých modelů – každý na jiné úrovni rozlišení (abstrakce, přesnosti) a uživatel si vybírá podle svých požadavků jeden z nich.

[2] Ören, T. I. - Professor Emeritus, Ottawa Center of the McLeod Institute of Simulation Sciences of the SCS, University of Ottawa, Ontario, Canada.

3. Variable structure models.
4. Mixed formalism simulation – simulace je spojena s jiným typem zpracování znalostí (například expertní systém dodávající konkrétní parametry do simulace).
5. Multisimulation – zcela nový koncept, kde lze experimentovat s více aspekty reality souběžně.
6. Concurrent simulation – simulace se odvíjí souběžně s existencí své reálné předlohy. Využití je pro predikci situací v realitě (například systémy řízení rozvodů energie, dopravy). Tento případ je zde popisován jako jedna z aplikací interaktivní rozšiřitelné simulace.
7. Goal processing in modelling and simulation.
8. Automation of design of experiments.
9. Agent-directed simulation – modely založené na komunikujících (heterogenních) agentech, které jsou v poslední době velmi populární.
10. Holonic agent simulation (for cooperative systems) – speciální typ agentní simulace, kdy agenti (nazvaní *holoni*) mění dynamicky svou strategii v případech potřeby celku řešit prioritní úlohu.
11. Specification languages and environments for interoperability – interoperabilita (propojovanost, propojitelnost, spolupráce) je důležitá vlastnost v integrování simulačních modelů. Ze zkušeností plyne, že *kombinování existujících programových systémů je mnohem efektivnější než budování nových.*

Je vidět, že existuje velká škála specifických přístupů v modelování, kde vystupuje modularita, provázání se znalostmi, souběžné a interaktivní simulace, interoperabilita a specifikační (propojovací) jazyky vysoké abstrakce. V odborné literatuře vystupuje ještě pojem *multi-formalismus*, který má asi nejblíže našemu pojmu *heterogenity*. V článku je též zmíněna platforma HLA jako možný nástroj propojování komunikujících modulů modelu.

3.2.1 Platformy HLA a CORBA

Existují propojovací platformy pro vytváření specializovaných distribuovaných výpočtů. Tyto systémové architektury (HLA, CORBA) jsou obvykle i heterogenní ve smyslu propojování různorodých systémů na úrovni například programovacích jazyků.

Problém je však jiný. Tyto platformy zabezpečují fyzický přenos informací z jednoho systému na druhý. Aplikace A hodlá přes speciální rozhraní zaslat aplikaci B zprávu: rozhraní převede zprávu na univerzální formát (především čísla na binární úrovni) a zašle rozhraní na další straně, které ji upraví do podoby srozumitelné pro aplikaci B.

Platformy HLA a CORBA však v porovnání s obecnou heterogenitou:

- Nepřinášejí obecnou metodologii budování heterogenního systému.
- Omezují sadu zapojitelných subsystémů na několik podporovaných jazyků a vývojových prostředí.

HLA a CORBA mohou sloužit jako ustálená a odladěná komunikační vrstva pro vyšší heterogenní technologii. Nejdůležitějším aspektem fyzické komunikace je totiž sjednocení formátů dat na binární úrovni a formát zprávy zasílané mezi subsystémy. V dnešní době se velmi prosazuje jazyk XML. V zásadě jde o dohodu, že nejlépe přenositelná podoba dat je textová. Jazyk XML zabezpečuje strukturování textu do typovaných datových položek a zpráva v XML je vlastně strukturovaný a všeobecně srozumitelný text. Opět se jedná o použitelnou fyzickou vrstvu pro heterogenní komunikaci.

3.2.2 Kompatibilita počítačových systémů

Pokusme se rozšířit aplikaci heterogenity na veškeré počítačové systémy do oblastí:

- *Operační systémy*, kde je kompatibilita mezi různými systémy skutečně důležitým prvkem.
- *Aplikace* v rámci operačního systému, kde je propojitelnost stále nedořešeným problémem. Přenos dat mezi aplikacemi má omezenou podobu – v zásadě lze mluvit pouze o možnosti kopírovat text mezi aplikacemi. Data lze přenášet například v kódování podle normy XML, ale to pouze v případě, že obě strany komunikace jsou k tomu programovány (podobnou funkci mají objektové technologie OLE a COM).

Jedním z řešení je všechny systémy založit na jediném jazyce a paradigmatu. Prostředí jazyků Smalltalk, či Self to naznačuje. Smalltalk má od počátků svého vývoje určitou tendenci být operačním systémem. Svět počítačů postavených na jednom jazyce, či paradigmatu je však iluzorní a nejspíše i pro mnoho uživatelů nepřijatelný. Kompromisem by bylo vytvářet aplikace s ohledem na jednotný propojovací standard. Následující kapitola 4 jeden možný přístup popisuje.

3.2.3 Multi-paradigm jazyky

V současné době probíhá vývoj tak zvaných *multi-paradigm jazyků* – jazyků podporujících více programovacích paradigmat (logické, funkcionální, constraint-bases, objektově orientované, procedurální i paralelní). Ole Madsen (spoluautor jazyka Beta) ve svém článku [Madsen 00] píše: “Cíl výzkumu v programovacích jazycích by měl být vývoj jazyků integrujících to nejlepší z konceptů a konstrukcí různých programovacích paradigma”. Ve většině případů se jedná o propojení objektově orientovaného procedurálního jazyka s logickým, či funkcionálním programovacím jazykem. Podobné spojení prezentuje i tato práce, ale na obecnější úrovni.

Velmi populární je například jazyk Oz [Muller 95] implementovaný pomocí vývojového prostředí Mozart [Roy 99]. Výhodou jazyků typu Oz (objektově orientovaný jazyk s prvky logického programovacího jazyka) je zavedení mechanismu *inference* do výpočtu a budování programu na základě deterministického a nedeterministického výpočtu. Například Oz ukazuje deterministický logický program jako:

```
declare
proc {Append Xs Yz Zs}
  case Xs
  of nil Zs=Ys
  [] X|Xr then Zr in
     Zs=X|Zr {Append Xr Ys Zr}
```

```
  end
end
```

A nedeterministická varianta předchozího algoritmu:

```
proc {Append Xs Ys Zs}
  choice        Xs=nil Zs=Ys
  [] X Xr Zr in Xs=X|Xr Zs=X|Zr {Append Xr Ys Zr}
  end
end
```

Podobný je jazyk OLI (Objects and Logic Integration) [Lee I., Lee II.] odvozený od jazyka Smalltalk.

Všem těmto jazykům je společné *sekvenční* vedení výpočtu – běh programu se přepíná mezi sekvencí příkazů a inferenčním výpočtem. Takové pojetí jazyka je sice nové a přínosné, ale nevede k dokonalému splynutí dvou paradigmat (výpočet se pouze *přepíná* mezi dvěma izolovanými moduly programu). Nesmíme zapomínat na výsledný uživatelský dojem z mutli-paradigm prostředku – uživatel se neučí nový jazyk, ale integruje své původní znalosti do multi-paradigma stylu.

Cílem předložené práce je návrh propojení paradigmat na nejnižší úrovni, které vede k vytvoření jediného odvozeného paradigmatu – výpočetního modelu AIN popsaného ve čtvrté kapitole.

3.2.4 Standard DEVS

DEVS [DEVS, Praehofer DEVS] – Discrete Event Systems Specification – je již poměrně klasický standard v diskrétním modelování. Byl mnohokrát publikován v monografiích o modelování, na konferencích a ve vědeckých pracech. DEVS definuje obecný (v základní podobě) diskrétní simulační formalismus, který slouží jako řídicí prvek nad konkrétním modelem. Ukáže se, že DEVS může být použit i pro řízení heterogenních modelů.

Základem definice DEVSu je pojem *atomického bloku*. Atomický blok (atomický model AM) je

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

kde

X je množina *vstupních událostí*

S je množina *vnitřních stavů*

Y je množina *výstupních událostí*

$\delta_{ext} : Q \times X \to S$ je přechodová funkce na vnější událost. $Q = \{(s, e) | s \in S\, a\, 0 \leq e \leq ta(s)\}$ je úplný stavový prostor bloku. Pro jednoduchost berme $Q = S$

$\delta_{int} : S \to S$ je interní přechodová funkce

$\lambda : S \to Y$ je výstupní funkce

$ta : S \to R^+$ (R^+ je kladné reálné číslo včetně $\{0\}$) je funkce časového kroku (time advance)

DEVS je systém založený na stavovém stroji a obsluze událostí. DEVS blok modeluje systém se vstupy a výstupy, kde vstupy se nazývají *množina vstupních portů* a výstupy *množina výstupních portů*. Název port pramení z funkce – je určen pro napojení na porty jiných bloků. Největší přínos DEVSu je v modularizování typických prvků simulačních modelů do bloků, které pak lze jednoduše skládat a vytvářet složitější bloky.

Blok je tedy v určitém stavu a očekává jeden ze dvou druhů událostí:

1. *Externí* událost $x \in X$, kdy na port vstupu odpovídající události x přijde zpráva o události a blok přejde ze stavu $s \in S$ do stavu $\delta_{ext}(s, x)$.

2. *Interní* událost, kdy bloku přijde zpráva o dosažení naplánovaného časového okamžiku (podle time advance). Blok přejde ze stavu $s \in S$ do stavu daného interní přechodovou funkcí $\delta_{int}(s)$.

Pro implementaci DEVSu každý blok deklaruje rozhraní tvořené obsluhou čtyř zpráv (interní a externí přechodová funkce, časový krok a výstupní funkce) a uvidíme, že konkrétní realizace bloku není závislá na platformě, což vede k heterogenitě. Otázkou je, nakolik je tato vlastnost využívána a zda nezůstává pouze v teoretické rovině [Praehofer 96].

Příklad DEVS bloku

Generátor procesů – obsluhuje pouze vnitřní přechod.

```
class DEVSgenerator : public DEVSblock {
...
      void IntTransition() {
           first = FALSE;
      }
      void outputFunction() {
          send(OUTPORT, newJob());
      }
      time_t timeAdvance() {
          if (first) return 0; else return Uniform(A,B);
      }
} ;
```

3.3 Úvod do architektury heterogenního systému

V současné době je heterogenita souborem specifických přístupů a metod k budování modelů. V odborné literatuře se setkáme s polemikou, nápady, ale málokdy s konkrétním řešením, natož s vybudovanou teorií o architektuře heterogenního systému. Zdůrazněme, že heterogenitou rozumíme neomezenou propojitelnost mezi subsystémy.

Pokud chceme formálně popsat typický heterogenní systém, pak musíme začít popisem funkcí typického jednoho (autonomního) subsystému. Vycházejme z předpokladu, že každý subsystém je původem autonomní systém. Každý subsystém:

- Je schopen vlastní činnosti – říkáme, že má vlastní chování nebo vlastní hlavní program.

- Obsahuje sadu funkcí (služeb). Jeho data budeme zpřístupňovat voláním podprogramů (nebo zasílání zpráv) a obecně tuto situaci budeme označovat jako užívání jeho služeb.

Ukažme si například pojetí konečného automatu přizpůsobeného pro zapojení do heterogenního systému. Konečný automat $Automat = (Q, \delta, \Sigma, q_0, Q_F)$[3] převedeme na ovládání sadou funkcí (pro jednoduchost ve stylu jazyka C, kde Σ a Q jsou výčtové datové typy):

Q $actualState()$ vrací aktuální stav,

Σ $getNextInput()$ vrací další symbol ze vstupu,

void $setState(Q)$ nastaví aktuální stav,

void $doStep(Q, \Sigma)$ provede jeden krok na daný vstup,

void $initialize()$ nastaví aktuální stav na q_0,

void $Main()$ vlastní program řešící algoritmus:

1. proveď inicializaci – $initialize()$
2. dokud $actualState() \notin Q_F$, opakuj:
   (a) $doStep(actualState(), getNextInput())$.

Podobným způsobem přizpůsobíme každý subsystém, který má být zapojen do heterogenního modelování. Pro řízení heterogenního systému je důležité rozhodnutí, který ze subsystémů bude mít funkci hlavního řídicího subsystému. To je silně závislé na druhu aplikace. O heterogenních systémech zatím neuvažujeme jako o distribuovaných systémech nebo dokonce jako o paralelních systémech.

3.4 Demonstrační příklady

V následujícím textu uvedu dva příklady. První je ukázkou analýzy systému, která rozliší jeho procesní a znalostní složku. Druhý příklad není přímo ukázkou heterogenního modelu, ale souvisí s metodologií komunikace mezi částmi modelu.

3.4.1 Procesy a znalosti

Rád bych zmínil článek [Hrubý MOS01], ve kterém byly poprvé publikovány základní myšlenky mé práce. Článek vychází ze simulační knihovny SIMLIB/C++ [SIMLIB HP] a logického programovacího jazyka Prolog [SWI-Prolog HP], který v oblasti modelování můžeme chápat jako jazyk pro modelování znalostí a jejich vztahů[4]. Popsali jsme způsob propojení těchto dvou nástrojů, princip a smysl propojení jejich odlišných modelovacích přístupů a výsledek jsme ukázali na jednoduchém příkladu modelu dopravy ve městě.

[3] kde Q je množina stavů, δ je přechodová funkce $Q \times \Sigma \to Q$, $q \in Q$ je počáteční stav a $Q_F \subseteq Q$ je množina koncových stavů.

[4] V článku [Madsen 00] je zmíněn jazyk Primula, který vznikl spojením Prologu a Simuly 67. Salvesen, A. Wang, A.: Primula - An Integration of Prolog and Simula. Report no. 781, Norwegian Computing Center, 1986.

Křižovatky – model

Mějme úkol modelovat město popsané sítí silnic a křižovatek. Potřebujeme také modelovat auta pohybující se z místa A do místa B. Dostáváme klasický neorientovaný graf a úlohu hledání cesty v grafu.

Problém obsahuje dvě složky: znalostní a procesní. Znalostní částí je samotný graf – tedy databáze silnic a jejich křižování, případně délek silnic. Procesní částí je popis chování pohybujícího se auta. Je samozřejmě možné celý problém modelovat s využitím knihovny SIMLIB/C++ (příp. ekvivalentního jazyka SIMULA 67). Já však chci ukázat, že:

1. Pro vkládání znalostních dat do modelu je jazyk Prolog ideální.
2. Jazyk Prolog je ideální pro formulování algoritmu prohledávání grafu.
3. Knihovna SIMLIB je vhodná pro popis chování procesu *auto* a pro řízení celé simulace.
4. S existujícím nástrojem tvořeným SIMLIB/C++ a Prologem je sestavení modelu triviální.

Znalostní část jednoduše naznačíme v Prologu takto:

```
find(From, To, To) :-
      road(From, To, _).
find(From, To, Next) :-
      road(From, Next, _),
      find(Next, To, _).
findroad(From, From, []).
findroad(From, To, W) :-
      find(From, To, Next),
      findroad(Next, To, NN),
      append([Next], NN, W).
```

A procesní část v SIMLIB:

```
class Car : public Process {
public:
      Car(term_t from, term_t to) {
         Act=From=from; To=to; // proces auto
           // jedoucí z místa From do místa To
      };
      void Behavior(){
         double dist;
         term_t next;
         while (Act != To) {
           // volání Prologu, které zadá
           // dotaz na predikát findroad. Prolog
           // naváže odpověď na proměnné next a dist
           findinc(Act, To, &next, &dist);
           Wait(dist/50); // čekání odpovídající
              // jízdě na vzdálenost dist
```

```
                Act=next; // dosažení křižovatky next
            }
        }
};
```

Každou silnici (dlouhou 200 metrů) z křižovatky A do křižovatky B zadáme:

```
road(a,b,200).
```

Tento způsob modelování problému *Křižovatky* je velmi pohodlný, rychlý a flexibilní.

Při propojení C++/Prolog můžeme využít tří modelů spolupráce:

1. Simulace je záležitostí C++, Prolog je volán pro rozhodování na základě znalostí v databázi.

2. Kontrola simulace je v Prologu, Prolog volá SIMLIB pro dílčí výpočty. Výhodou je mechanismus inference v Prologu, který v simulaci může sloužit pro řešení úloh (jako v umělé inteligneci) na základě simulace.

3. Simulační proces (tok simulačních experimentů) *volně přechází mezi oběma jazyky.* Tento model řízení simulace je samozřejmě ideální a měl by být cílem vývoje heterogenních simulačních nástrojů. Aby byl však technicky realizovatelný, musel by se přizpůsobit přístup k datům a podprogramům v obou subsystémech.

3.4.2 Procesy a události

Následující příklad (publikován na konferenci ASU'02 – [Hrubý ASU02]) ukazuje model situace, kdy spolu mají komunikovat dva prvky modelu.

Čtenář a slunce

Mějme za úkol modelovat problém Čtenář a slunce. Slunce čtenáři dodává světlo potřebné ke čtení. Aktivní entita *čtenář* zde vystupuje jako proces. Pro demonstrování zmiňovaného přístupu je podstatný moment, kdy slunce zapadne. Tento okamžik budeme abstrahovat jako diskrétní událost a potřebujeme modelovat změnu stavu entity čtenář. V konvenčních simulačních jazycích orientovaných na procesy a události máme tyto možnosti:

1. Objekt slunce zašle objektu čtenář zprávu (událost) – je však nutné, aby objekt slunce znal referenci na objekt čtenáře. Navíc složitost roste s počtem objektů čtenář.

2. Objekt čtenář obsahuje podproces, který v pravidelných intervalech sleduje stav objektu slunce (složitost opět roste s každou podobnou vazbou).

Popisovaný princip lze zobecnit na libovolnou vazbu mezi procesem a měnícím se stavem jeho okolí. Ve většině případů zjistíme, že klasický procesní přístup se zasíláním zpráv je nedostatečný a že se příliš odlišuje od přirozeného způsobu modelovat[5] tento problém.

Příklad *Čtenář a slunce* patří již ke klasickým ukázkám teorie kolem jazyka HELEF. Bude rozveden v následujících kapitolách a v příloze je možno najít ukázku jeho řešení. Jádrem řešení je komunikační síť AIN, která z deklarovaného vztahu slunce a čtenáře vytvoří mezi nimi propojení a automaticky kontroluje platnost zadané podmínky.

[5] Ve světě návrhových vzorů (design patterns) existuje návrhový vzor *Observer*, který by tento problém mohl elegantně řešit první variantou.

3.5 Návrh heterogenního modelu

Z předchozího textu plyne motivace pro programování rozsáhlejších aplikací heterogenním způsobem a také dva přístupy k heterogenitě:

1. **Jednotná platforma pro spojování subsystémů** – ta se řeší v kapitole 4. Výhodou je sjednocený princip komunikace mezi subsystémy a formálnější základ architektury výsledného systému. Nevýhodou je striktní přístup k přizpůsobení subsystému do celku, který může být v některých výjimečných případech omezující.

2. **Živelný přístup** – každé propojení dvou subsystémů v rámci celku je odlišné. Neexistuje jednotná platforma komunikace, komunikace je omezená na subsytémy s vytvořeným vzájemným rozhraním. Výhodou může být přizpůsobení komunikace zvláštním charakteristickým znakům subsystému a zdánlivá rychlost realizace. Nevýhodou je pochopitelně roztříštěnost komunikace a nekompatibilita.

Máme specifikovaný problém, který je třeba implementovat. Problém obsahuje rysy různých paradigmat programování. Musíme si uvědomit, že rozložení implementace na různé platformy může zlepšit i ladění aplikace a její budoucí rozšiřování. Například vyčlenění některých funkcí z C++ pro implementaci do LISPu může být výhodné.

Prvním krokem je tedy analýza problému a jeho rozložení do podproblémů pro implementaci v subsystémech. Tento krok je klíčový pro návrh systému a bohužel jej nelze zde podrobně rozebírat. Jeho řešení závisí na dostupných subsystémech, na jejich ovládnutí a na celkových znalostech uživatele. Metodologie rozboru problému a aplikování heterogenity přesahují rámec této práce.

Druhým krokem je přizpůsobení subystémů pro propojení a návrh komunikačního rozhraní (jednotná platforma, živelný přístup). Zdůrazněme, že cílem je udržovat komunikaci mezi subsystémy na vyšší úrovni abstrakce (například pomocí proxy objektů).

Posledním krokem je implementace heterogenního systému. Tento krok může být velmi náročný a projeví se zde univerzálnost navrženého komunikačního rozhraní. Zřejmě cesta jednotné platformy bude nejschůdnější.

Problémy řešení

Byly podány názory, motivace a částečná řešení heterogenních modelů. Pro optimální propojování systémů je zřejmě nejvýhodnější variantou *vytvoření jednotné komunikační a řídicí platformy*, která by zabezpečovala propojení subsystémů na úrovních:

1. *Metodologické* – problém nalezení společných prvků u naprosto odlišných výpočetních paradigmat.

2. *Realizační* – návrh subsystémů, stylu komunikace a řízení mezi subsystémy.

3. *Fyzické (implementační)* – komunikace mezi subsystémy na úrovni datových přenosů a volání podprogramů.

V následující kapitole je popsána metoda univerzální jednodné propojovací platformy, která řeší metodologickou a realizační úroveň budování heterogenních modelů.

Kapitola 4

Univerzální propojovací platforma AIN

V této kapitole, která obsahuje jádro disertační práce, je popsáno nové vypočetní paradigma založené na speciální síti AIN tvořící výpočetní model obecné architektury heterogenního systému. Následující definice nejsou stavěny jako formálně precizní popis AIN, ale formalismy jsou používány jen ke zkrácení popisu některých jevů.

Cílem metody AIN je převést libovolný způsob vyčíslování v libovolném výpočetním systému na jednotnou platformu (síť AIN). Každý submodel heterogenního modelu je možno převést na síť AIN a tím jej propojit s ostatními submodely. Síť AIN tedy vytváří univerzální propojovací platformu pro tvorbu heterogenních modelů.

V obvyklých imperativních jazycích a částečně i v praktických implementacích deklarativních jazyků specifikujeme explicitně *pořadí* výpočtů[1]. Program je *sekvencí výpočetních kroků*. Programátor má velkou kontrolu nad efektivností výpočtu a nad jeho průběhem. Sekvenční výpočet však může nakonec program komplikovat.

Problém si naznačíme na příkladě křížových datových vazeb mezi proměnnými.

Příklad 1.

```
int a,b,c;
...
c = expression(a,b);
```

Má-li mít proměnná c vždy hodnotu odvozenou z hodnot proměnných a a b, pak musíme zachytit každou změnu těchto proměnných a vyvolat výpočet *expression* (libovolný výpočet nad proměnnými). To může v některých případech komplikovat program (testování, volání funkcí, ...). Představme si, že by prostý zápis $c = expression(a, b)$ garantoval v každém okamžiku běhu programu *platnost deklarovaného vztahu mezi proměnnými*.

Výpočet v procedurálním jazyce je řízen explicitně stanovenými kroky. Výpočet v (ideálním) deklarativním jazyce je řízen určitým vyšším principem, který respektuje předloženou sadu deklarací – a pak by nemělo teoreticky záležet na *pořadí provádění* automaticky volených kroků výpočtu. Kompromisem je zde prezentované výpočetní paradigma, které se stalo základem jazyka *HELEF* a které je vhodnější pro řízení simulace heterogenního modelu.

[1] Například předpis pro `assert` v Prologu ukazuje, že Prolog není zcela deklarativní jazyk. Záleží na pořadí predikátů v databázi.

Metoda založená na AIN (Automatic Information Net) přináší *opačný* přístup k programování – vazby mezi proměnnými jsou pouze *deklarovány* (sada deklarací) a řídící systém AIN udržuje jejich stálou konzistenci. Součástí řešení sady deklarací metodou AIN je dekompozice modelu na množinu objektů propojených do sítě AIN sledující datové závislosti mezi objekty.

V této kapitole budeme často pracovat s pojmem *objekt*. Řekněme tedy, že z hlediska metody AIN je objekt o datový element obsahující atribut $value(o)$, který nese aktuální hodnotu objektu o – objekt je paměťovým elementem systému. Objektem s hodnotou je například i *výraz* (například $expression(a, b)$). V programovacích jazycích pro uchování hodnot používáme *proměnné*. Ukažme základní rozdíl mezi metodou AIN a programovacím jazykem:

- V programovacím jazyce dotazem na hodnotu výrazu začne výpočet jeho vyhodnocení (volání funkce nebo přiřazení do proměnné).
- V metodě AIN se dotazem na hodnotu objektu o získá obsah $value(o)$. Obsah $value(o)$ se vyčíslí v tom okamžiku, kdy v systému AIN nastanou předpoklady ke *změně hodnoty* objektu o. V případě našeho příkladu je to situace objektu *expression(a,b)*, když došlo ke změně jednoho z objektů $\{a, b\}$.

Princip vyhodnocování AIN v závislosti na probíhajících změnách v systému může být v některých případech považován za efektivnější, než je konvenční sekvenční výpočet[2].

4.1 Úvod: Rozložení data × procesy

Metoda AIN vznikla propojením deklarativních a imperativních způsobů výpočtu. Proto metoda AIN kromě sítě provázaných výrazů zavádí i sekvenční výpočty formou posloupnosti příkazů, které mohou v každém kroku modifikovat kontext a strukturu sítě (tak zvané *akce* nebo také *procesy*), což rozšiřuje koncept systému *proměnná+výraz* na systém *proměnná+výraz+akce*.

V systému *proměnná+výraz* příliš nezáleželo na pořadí vyhodnocování změn, v systému *proměnná+výraz+akce* se zavádí mechanismus, který umožňuje spustit běh akce v *přesně stanoveném okamžiku*. Následující definice metody AIN je založená na pojmu *simulační objekt*. Simulační objekt je objekt, který je buď objekt systému *proměnná+výraz* (datový simulační objekt) nebo je akcí (procesní simulační objekt).

Pro označení oborů hodnot všech primitivních datových typů zaveďme pojem *universa* $\mathcal{U}$, zahrnující například veškerá čísla a prázdnou hodnotu označovanou ϵ.

Definice 1. *Element so nazveme simulačním objektem, pokud obsahuje (vnitřní) atributy:*

- Id – *identita objektu,* $id \in \mathbb{N}$.
- Value – *hodnota objektu,* $value(so) \in \mathcal{U}$.
- Type – *nabývá hodnoty výčtového typu* $\{data, process\}$ *pro rozlišení datového a procesního simulačního objektu.*
- Contents – *blíže neurčený předpis pro odvození hodnoty* value.

[2]Projeví se to zejména u implementace klasického synchronizačního prostředku *waituntil(podmínka)*.

K atributům objektu so je přistupováno zápisem $value(so)$, případně $type(so)$ a *contents(so)*. Každý objekt má svou identitu (atribut id). V dalším textu bude objekt ztotožňován s jeho identitou (kromě míst, kde by to mohlo být matoucí). V následujících definicích nelze zcela spojovat pojem objekt s představou konkrétního objektu v paměti počítače. Dva objekty o_1 a o_2 mající $id(o_1) = id(o_2)$ budou považovány za identické (včetně všech jejich atributů) a píšeme $o_1 \approx o_2$.

Budeme pracovat s množinami simulačních objektů. Aktuální množinu všech simulačních objektů v systému budeme označovat OS (Objects in System).

4.2 Automatická informační síť (AIN)

Popis metody AIN bude strukturován podle určitých úrovní abstrakce v krocích:

1. *Síť datových objektů* – zavedení datového simulačního objektu a jeho vazb k ostatním objektům. Množina objektů a jejich vazeb tvoří strukturu sítě AIN.

2. *Evoluce sítě, kontext evoluce sítě* – princip vyčíslování hodnot objektů v kontextu sítě AIN.

3. *Procesní objekty, kontext procesní části* – zavedení procesních simulačních objektů a řízení činnosti procesů.

4. *Celkový kontext (data, procesy)* – definice celkového kontextu evoluce sítě AIN zahrnující vyčíslování datových objektů a řízení procesů.

5. *Běh a ukončení výpočtu* – evoluce systému sítě AIN od počátečního celkového kontextu po koncový kontext.

Datové simulační objekty

Vycházejme z aktuální konečné množiny OS simulačních objektů. Každý simulační objekt $so_d \in OS$, $type(so_d) = data$ můžeme popsat výrazem

$$so_d = expression(so_{d_1}, ..., so_{d_{dmax}});\ dmax \in \mathbb{N} \tag{4.1}$$

kde $\{so_{d_1}, ..., so_{d_{dmax}}\} \subseteq OS$ jsou také simulační objekty, ze kterých so_d odvozuje svou hodnotu ($contents(so_d) = expression(so_{d_1}, ..., so_{d_{dmax}})$). Takovou množinu $\{so_{d_1}, ..., so_{d_{dmax}}\}$ budeme nazývat *báze objektu* so_d a označovat $Base(so_d)$.

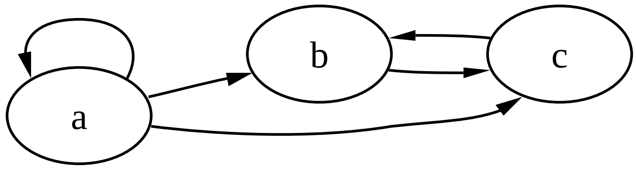
$$Base(so_d) = \{so_{d_1}, ..., so_{d_{dmax}}\} \subseteq OS \tag{4.2}$$

Říkáme, že objekt so_d je *závislý* na objektech z množiny $Base(so_d)$. Pro označení vazeb mezi $so_d \in OS$ a $Base(so_d)$ zavedeme:

$$Relations(so_d) = \{(b, so_d) | b \in Base(so_d)\} \subseteq OS \times OS \tag{4.3}$$

Všechny vazby všech $o \in OS$ (kromě vazeb mezi datovými objekty existují navíc vazby z datového objektu na objekt akce) jsou:

$$R = \bigcup_{o \in OS} Relations(o) \tag{4.4}$$



Obrázek 4.1: Příklad sítě

Příklad 2. *Uveďme příklady datových objektů:* $OS = \{1, (1+2), a, (a+b)\}$. *Objekt* $a+b$ *má atributy:* $type(a+b) = data$, $contents(a+b) = a+b$ *a* $value(a+b)$ *je zatím neznámé* $(= \epsilon)$.

Přistupme k definici sítě AIN složené ze simulačních datových a procesních objektů a jejich závislostí:

Definice 2. *Síť AIN je struktura* $S = [OS, R] = [OS \cup \{AINnoevent\}, R]$, *kde*

1. *OS je množina simulačních objektů. OS obsahuje význačný prvek* $AINnoevent$.[3]
2. *R je relace závislosti mezi simulačními objekty* $R \subseteq OS \times (OS \setminus \{AINnoevent\})$.

V této fázi definice AIN budeme hovořit pouze o *struktuře sítě* vyjádřené vazbami závislosti. Konkrétní vyčíslení výrazu $contents(so_d)$ (4.1) objektu $so_d \in OS$ bude popsáno později. Všimněme si, že síť AIN tvoří *orientovaný graf.* V síti $[OS, R]$ budeme psát $o \in OS$ a je tím míněno, že *objekt identity* $id(o)$ *je obsažen v OS.*

Příklad 3. *Obrázek 4.1 ukazuje síť* $S = [OS_1, R_1]$, *kde:*

$$OS_1 = \{a, b, c\}$$
$$R_1 = \{(a,a), (a,b), (a,c), (b,c), (c,b)\}$$

Notace: Zavedeme notaci podobnou Petriho sítím. Je-li $S = [OS, R]$ síť, pak:

- $\bullet o = \{x \in OS | (x, o) \in R\}$,
- $o\bullet = \{y \in OS | (o, y) \in R\}$,
- $ndep(o) = |o \bullet|$, pro $o \in OS$,
- $nread(o) = |\bullet o|$, pro $o \in OS$,
- je-li $o_1, o_2 \in OS$ a platí, že $(o_1, o_2) \in R$, tedy že $o_2 \in o_1\bullet$, pak píšeme $o_1 \to o_2$,
- v síti $S = [OS, R]$, je-li $o \in OS$, budeme pro jednoduchost zápisu psát $o \in S$.

Říkáme, že simulační objekt o je v síti $[OS, R]$:

- *Užitečný*, pokud platí $ndep(o) > 0$.
- *Vyčíslitelný*, pokud platí $\{(from, o) | from \in Base(o)\} \subseteq R$ (píšeme $computable(o, OS, R)$).
- *Prázdný*, značíme ϵ. Síť $S = [OS, R]$ vždy obsahuje ϵ, stejně tak $\epsilon \in \{\mathcal{U}\}$.

Síť $S = [OS, R]$ se nazývá *úplná*, pokud $\forall o \in OS : o$ je vyčíslitelný.

[3] Význam prvku $AINnoevent$ se objasní v následující kapitole

4.2.1 Evoluce sítě AIN

Přistupujme k síti $AIN = [OS, R]$ jako k systému AIN $\Sigma = (OS, R)$ s atributy OS a R. *Síť AIN* vyjadřuje statickou strukturu systému, *systém AIN* vyjadřuje *dynamický stav* sítě (hodnoty atributů objektů). V dalších kapitolách budou do systému přidávány další atributy tak, jak se bude definice systému AIN rozvíjet.

Definice 3. *Mějme síť* $AIN = [OS, R]$. *Výpočetní AIN-systém* $\Sigma = (OS, R)$ *představuje stav sítě* AIN.

Změnou stavu AIN-systému $\Sigma = (OS, R)$ na $\Sigma_2 = (OS_2, R_2)$ rozumíme výskyt jedné nebo více z následujících možností:

- Změnu (vnitřního) atributu simulačního objektu $o \in OS$ na $o_2 \in OS_2$ tak, že $o \approx o_2$.
- Změnu počtu objektů nebo identit v OS tak, že $\exists o \in OS : \nexists o_2 \in OS_2, o_2 \approx o$ nebo naopak.
- Změnu množiny závislostí R na R_2, že $R \neq R_2$. Vazby (a, b) a (a_2, b_2), kde $a \approx a_2$ a $b \approx b_2$ chápeme jako identické.

Pro změnu hodnoty (atributu *value*) objektu $o \in OS$ v systému $\Sigma = (OS, R)$ zavedeme funkci

$$chValue : (\Sigma, o, val) \mapsto \Sigma_2, val \in \mathcal{U} \tag{4.5}$$

takovou, že v $\Sigma_2 = (OS \setminus \{o\} \cup \{o_2\}, R)$ existuje objekt $o_2 \in OS_2$:

1. $o_2 \approx o$.
2. $value(o_2) = val$.

V následujících definicích bude zaveden pojem *kontext*, který není přímo vázán na *stav* sítě, ale je součástí výpočtu stavu sítě. Kontext výpočtu sítě AIN je neustálým překlápěním mezi dvěma stavy určenými dvojicí množin $Marked$ a $Changed$, kde:

- $Marked$ vyjadřuje množinu objektů, jejichž báze (hodnoty *value* bázových objektů) se v předchozím kroce změnila, a proto objekty z $Marked$ musí být přehodnoceny.
- $Changed$ vyjadřuje množinu objektů, které při přehodnocování změnily svou hodnotu *value* (a v dalším kroku zavedou přehodnocování na nich závislých objektů).

Definujme *kontext systému AIN*:

Definice 4. *Mějme síť AIN:* $AIN = [OS, R]$. *Struktura* $AC = (\Sigma, Marked, Changed)$ *se nazývá kontextem (evolučním stavem) systému AIN, pokud platí:*

1. $\Sigma = (OS, R)$ *je AIN-systém.*
2. $Marked \subseteq OS$, $Changed \subseteq OS$.

Definice 5. *Kontext systému AIN* $AC = (\Sigma, Marked, Changed)$ *se nazývá* platný, *pokud platí právě jeden ze stavů:*

1. $Marked = \emptyset \wedge Changed \neq \emptyset$.

2. $Marked \neq \emptyset \wedge Changed = \emptyset$.

3. $Marked = Changed = \emptyset$.

Platný kontext systému AIN AC se nazývá:

- *Ustálený*, pokud platí $Marked = Changed = \emptyset$ (Bude označován AC_U).
- *Označený*, pokud platí $Marked \neq \emptyset \wedge Changed = \emptyset$.
- *Změněný*, pokud platí $Marked = \emptyset \wedge Changed \neq \emptyset$.

Principem činnosti systému AIN je provádění evolučních kroků z jednoho kontextu AC_1 na kontext AC_2. Přistupme k definici jednoho evolučního kroku systému AIN.

Definice 6. *Platný kontext* $AC_1 = (\Sigma_1, M_1, CH_1)$ *odvozuje platný kontext* $AC_2 = (\Sigma_2, M_2, CH_2)$ *pokud platí:*

1. AC_1 *je ustálený,* $M_2 = AINnoevent\bullet$, $CH_2 = \emptyset$ *a* $\Sigma_2 = \Sigma_1$,

2. AC_1 *je změněný,* $M_2 = \bigcup_{i \in CH_1} i\bullet$, $CH_2 = \emptyset$ *a* $\Sigma_2 = \Sigma_1$,

3. AC_1 *je označený a atributy* M_2, CH_2 *a* Σ_2 *získáme algoritmem:*

   - $M_2 := \emptyset$, $\Sigma := \Sigma_1, CH_2 := \emptyset$
   - $while(M_1 \neq \emptyset)\, begin$
   - $m :=$*vyjmi prvek z* M_1
   - $(flag, \Sigma) := reeval(\Sigma, m)$[4]
   - $if\ (flag = TRUE)\, then\ CH_2 := CH_2 \cup \{m\}$
   - end
   - $\Sigma_2 := \Sigma$

Přepsání kontextu AC_1 na AC_2 zapisujeme $AC_1 \vdash AC_2$. Operace $\vdash$ definuje jeden evoluční krok sítě. Můžeme také psát, že kontexty AC_1 a AC_2 jsou v relaci evoluce. Vzhledem k tomu, že u metody AIN má smysl pracovat pouze s *platným* kontextem, pak bude dále slovem *kontext* míněn *platný kontext*.

4.2.2 Reeval: $(\Sigma, so) \mapsto (Boolean, \Sigma_2), so \in \Sigma$

Bylo již zmíněno, že základním atributem simulačního objektu so je jeho *aktualní hodnota* – atribut $value(so)$. Každý simulační objekt kromě toho obsahuje předpis $contents(so)$, podle kterého odvodí svou hodnotu z hodnot objektů své báze.

Funkce *reeval* pro daný AIN-systém Σ a datový objekt so sítě vrací nový systém Σ_2 s přehodnoceným objektem $so_2 \approx so$ a příznak změny hodnoty objektu:

$$reeval : (\Sigma, so) \mapsto (Boolean, \Sigma_2) \tag{4.6}$$

kde $so \in OS$, $\Sigma = (OS, R)$ a $\Sigma_2 = (OS_2, R)$ jsou AIN-systémy.

Zaveďme pomocnou funkci:

[4]Funkce $reeval : (\Sigma, m) \mapsto (Boolean, \Sigma_2)$ je definována v následující kapitole (4.2.2).

$$doeval : (\Sigma, so) \mapsto \mathcal{U} \tag{4.7}$$

$$doeval(\Sigma, so) = \begin{cases} contents(so) & ; computable(so, OS, R) \\ \epsilon \,;\, jinak & \end{cases} \tag{4.8}$$

kde předpis $contents(so)$ provádí výpočet hodnoty objektu so v kontextu systému (OS, R) odvozením z hodnot datových objektů báze. Je-li hodnota $value(sb) = \epsilon$ pro nějaké $sb \in Base(so)$, pak vyhodnocení $contents(so)$ vrací hodnotu ϵ.

Zaveďme pomocnou funkci $reevalWith(\Sigma, so, val)$:

$$reevalWith : (\Sigma, so, val) \mapsto (Boolean, \Sigma_2) \tag{4.9}$$

$$reevalWith(\Sigma, so, val) = \begin{cases} (FALSE, \Sigma) & ; value(so) = val \\ (TRUE, chValue(\Sigma, so, val)) & ; value(so) \neq val \end{cases} \tag{4.10}$$

Pak je funkce $reeval$ dána:

$$reeval(\Sigma, so) = reevalWith(\Sigma, so, doeval(\Sigma, so)) \tag{4.11}$$

Smyslem funkce $reeval$ je přehodnotit aktuální hodnotu objektu a vrátit hodnotu $TRUE$, pokud se přehodnocením změnila hodnota objektu (resp. jeho atributu $value$). Tím, že se za hodnotu výrazu so bere jeho poslední vyčíslená hodnota $value(so)$, nemůže dojít k zacyklení při výpočtu vzájemně křížových odkazů (x je odvozeno z y a naopak). Rozložení funkce *reeval* do tří funkcí respektuje budoucí objektově orientovanou implementaci metody AIN.

4.2.3 Proces evoluce sítě

Evoluce sítě je pro metodu AIN způsobem vedení výpočtu. Evoluce systému AIN má svůj počátek v počátečním kontextu, má svůj průběh a lze určit i její koncový stav (kontext).

Definice 7. *Počáteční stav systému* $\Sigma = (OS, R)$ *je kontext* $AC_0 = (\Sigma, M_0, \emptyset)$ *takový, že:*

1. $M_0 = \{o \in OS | \bullet o = \emptyset\} \setminus \{AINnoevent\}$.
2. $\forall o \in OS : value(o) = \epsilon$.

Definice 8. *Evoluce systému* $\Sigma = (OS, R)$ *je posloupností kontextů* $AC_0, AC_1, AC_2, ...,$ *kde platí:*

1. $AC_0 = (\Sigma, M_0, \emptyset)$ *je počáteční označený stav systému AIN.*
2. $AC_i \vdash AC_{i+1}$ *pro všechny* $i = 0, 1, 2, ...;$ $i \in \mathbb{N}$.

Evoluce nemá implicitně definovanovatelný koncový kontext. Respektive, pokud sledujeme ukončení *pouze evoluce* sítě AIN, a nezahrnujeme zatím procesy, pak situace ukončení výpočtu sítě je ustálený stav $AC_F = (\Sigma_F, \emptyset, \emptyset)$ sítě takový, že:

$$AC_F \vdash (\Sigma_{F1}, AINnoevent\bullet, \emptyset) \vdash (\Sigma_{F2}, M_x, \emptyset) \vdash (\Sigma_{FF}, \emptyset, \emptyset) = AC_{FF} \tag{4.12}$$

Jinak řečeno, hodnoty objektů z OS jsou takové, že z ustáleného stavu $(\Sigma_F, \emptyset, \emptyset)$ dojdeme do označeného stavu $(\Sigma_{F1}, AINnoevent\bullet, \emptyset)$, ale již nedojde v množině objektů $AINnoevent\bullet$ k žádné změně výpočtem *reeval*. Je zřejmé, že $OS(\Sigma_F) = OS(\Sigma_{F1}) = OS(\Sigma_{F2}) = OS(\Sigma_{FF})$, proto $AC_F = AC_{FF}$ a systém cyklí ustáleném stavu.

Příklad 4. *V síti* $OS = [OS, R] = [\{1, a\}, \{(1, a)\}], \Sigma = (OS, R)$ *je* $AC_0 = (\Sigma, \{1\}, \emptyset)$ *a* $AC_0 \vdash (\Sigma_1, \emptyset, \{1\}) \vdash (\Sigma_2, \{a\}, \emptyset) \vdash (\Sigma_3, \emptyset, \{a\}) \vdash (\Sigma_4, \emptyset, \emptyset) \vdash (\Sigma_5, AINnoevent\bullet, \emptyset) = (\Sigma_6, \emptyset, \emptyset)$*. V kontextu* AC_0 *se volá reeval pro objekt* 1 *(v* AC_0 *má* $value(1) = \epsilon$*). Vyčíslením reeval dostává objekt hodnotu jedna, tím mění svou hodnotu z* ϵ *na* 1.

V praxi však konec běhu evoluce definuje až model sám. Nyní můžeme zavést tranzitivní uzávěr relace $\vdash$.

$AC_s \vdash^+ AC_f$, existuje-li posloupnost kontextů $AC_1, AC_2, ..., AC_{x-1}, AC_x$ taková, že $AC_s = AC_1$, $AC_1 \vdash AC_2$, ..., $AC_{x-1} \vdash AC_x$ a $AC_x = AC_f$. Pokud existuje pro stav AC_s stav AC_{sf} takový, že $AC_s \vdash^+ AC_{sf}$ a AC_{sf} splňuje podmínku 4.12, pak píšeme $AC_s \vdash_F^+ AC_{sf}$.

Poznámka: Může se vyskytnout zvláštní stav, kdy $M_0 = \emptyset$, který se projeví v silně cyklické síti. Takový model však nemá praktický smysl.

Příklad 5. *Mějme síť* $S = [OS, R]$*, kde* $OS = \{1, b, b < 3, b + 1\}$ *a* $R = \{(1, b), (b < 3, b), (b, b < 3)\}$*. Počáteční kontext* $AC_{start} = (\Sigma_{start} = (OS, R), M_{start}, \emptyset)$ *a* $M_{start} = \{1\}$*. Evoluce proběhne následovně:*

$$(\Sigma_1, M_{start}, \emptyset) \vdash (\Sigma_2, \emptyset, \{1\}) \vdash (\Sigma_3, \{b\}, \emptyset) \vdash (\Sigma_5, \emptyset, \{b\}) \vdash (\Sigma_6, \{b < 3\}, \emptyset) \vdash$$

$$(\Sigma_7, \emptyset, \{b < 3\}) \vdash (\Sigma_8, \{b\}, \emptyset) \vdash (\Sigma_9, \emptyset, \{b\}) \vdash (\Sigma_{10}, \{b < 3\}, \emptyset) \vdash (\Sigma_{11}, \emptyset, \emptyset) \vdash$$

$$(\Sigma_{12}, \emptyset, \emptyset) = AC_{NatTerminate}$$

4.2.4 Podsíť

Definice 9. *Síť* $S_1 = [OS_1, R_1]$ *je* podsítí *sítě* $S = [OS, R]$*, pokud*

1. $OS_1 \subseteq OS$ *a* $R_1 \subseteq R$.
2. $\forall (o_F, o_T) \in R_1 : o_F \in OS_1 \vee o_T \in OS_1$.

Definice 10. *Síť* $S_1 = [OS_1, R_1]$ *tvoří* samostatnou podsíť *sítě* $S = [OS, R]$*, pokud*

1. S_1 *je podsíť* S.
2. $\forall (o_F, o_T) \in R_1 : o_F, o_T \in OS_1$.

Definice 11. *Síť* $S_1 = [OS_1, R_1]$ *tvoří* izolovanou podsíť *sítě* $S = [OS, R]$*, pokud*

1. S_1 *je samostatná podsíť* S.

2. neexistuje $(o_F, o_T) \in R$ *taková, že* $o_F \in S_1$ *a* $o_T \in OS \setminus OS_1$.

Definice 12. *V síti* $S = [OS, R]$ *tvoří objekt* $o \in OS$ specifickou podsíť $S_1 = [OS_1, R_1]$, *pokud*

1. S_1 *je samostatná podsíť* S.
2. OS_1 *obsahuje pouze objekt* o *a objekty, na které* o *odkazuje ve svých atributech.*

4.2.5 Procesní část

V této části práce zavedeme procesní složku metody AIN. Procesy jsou simulační objekty napojené na síť AIN, které se v přesně definovaných okamžicích aktivují. V průběhu jejich aktivity je evoluce $\vdash$ sítě pozastavena. Procesy umožňují formulovat sekvenčně prováděné části modelu a definovat strukturální zásahy do sítě AIN (přidání, odebrání a přemístění objektů).

Definice 13. *Proces* $P = (Body, StartC, ContinueC)$ *je simulační objekt sítě AIN s atributy:*

- *Body – tělo procesu tvořené konečnou posloupností příkazů.* $Body = (c_1, c_2, ..., c_{pc})$.
- *StartC – datový simulační objekt tvořící podmínku pro spuštění procesu.* $Value(StartC) \in \{TRUE, FALSE, \epsilon\}$.
- *ContinueC – datový simulační objekt tvořící podmínku pro běh procesu.* $Value(ContinueC) \in \{TRUE, FALSE, \epsilon\}$.

Objekty $StartC$ a $ContinueC$ mohou být prázdné (jsou rovny ϵ). Je-li v $[OS, R]$ proces $P \in OS$, pak i $StartC \in OS$ a $ContinueC \in OS$. Doplňme předpis pro množinu vazeb procesního simulačního objektu $P \in OS$:

$$Relations(P) = \{(StartC, P), (ContinueC, P)\} \setminus \{(\epsilon, P)\} \tag{4.13}$$

Kontext procesů

V následující definici použijeme pojem *fronta*. Frontou zde rozumíme homogenní seznam, ke kterému se přistupuje způsobem FIFO (přidává se zásadně na konec seznamu a prvky se berou zásadně ze začátku seznamu). Zápisem $<>$ bude označována *prázdná fronta*. Fronta procesů $p_1, p_2, ..., p_n$, $n \in \mathbb{N}$ bude zapisována $< p_1, p_2, ..., p_n >$. Je-li $queue =< i_1, i_2, ..., i_m >$, $m \in \mathbb{N}$ fronta, pak $set(queue)$ označuje množinu $\{i_1, i_2, ..., i_m\}$. Pro simulační objekt P typu proces zaveďme odvozený atribut $canContinue(P)$:

$$canContinue(P) = \begin{cases} value(ContinueC(P));\ ContinueC(P) \neq \epsilon \\ TRUE;\ jinak \end{cases} \tag{4.14}$$

Definice 14. *Nechť* $AIN = [OS, R]$ *je síť AIN. Kontext správy procesů sítě AIN je trojice* $PM = (ActP, ReadyQ, WaitingQ)$, *kde* $\{ActP\} \cup set(ReadyQ) \cup set(WaitingQ) \subseteq OS$ *a:*

- *ActP se nazývá* aktuálně běžící proces. $canContinue(ActP) = TRUE$.

- *ReadyQ se nazývá fronta* připravených procesů.
  $\forall p \in ReadyQ : canContinue(p) = TRUE$.
- *WaitingC se nazývá fronta* pozastavených procesů.
  $\forall p \in WaitingQ : canContinue(p) = FALSE$.

Notace: $PM_E = (\epsilon, <>, <>)$ je kontext bez procesů. Zápisem $p \in PM$ se rozumí: $p \in ReadyQ(PM) \vee p \in WaitingQ(PM) \vee p = ActP(PM)$.

Nechť $AIN = [OS, R]$ je síť AIN a $PM = (ActP, ReadyQ, WaitingQ)$ je kontext správy procesů. Každý simulační objekt $p \in OS$ typu proces se nazývá *aktivovaný*, pokud $p \in PM$. Když platí $p \notin PM$, pak se nazývá *neaktivovaný*. Je-li PM tvaru $PM = (\epsilon, <>, WaitingQ)$, pak se kontext správy procesů nazývá *neaktivní* (bude označován PM_{NonAct}), v případě $|set(ReadyQ)| > 0$ se kontext správy procesů nazývá *aktivní*.

4.3 Celkový kontext výpočtu

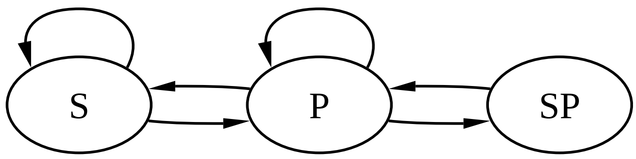
V předchozích kapitolách byla definována propojovací síť objektů AIN, která obsahuje simulační objekty typu data a procesy. Byl definován evoluční krok sítě AIN a kontext správy procesů. V následující definici se oba kontexty spojí do jediného – *celkového kontextu výpočtu*.

Pro rozlišení datové části AIN a části řízení procesů budeme označovat *síťovou část* a *procesní část*. Rozšíříme atributy AIN-systému (dynamické podoby sítě AIN) o atributy $ReadyQ$ a $WaitingQ$ pro vedení front procesní části metody AIN. AIN-systém zahrnující procesy je čtveřice $\Sigma = (OS, R, ReadyQ, WaitingQ)$, kde $[OS, R]$ je síť AIN.

Definice 15. *Nechť $AIN = [OS, R]$ je síť AIN, $\Sigma = (OS, R, ReadyQ, WaitingQ)$ je AIN systém a $AC = (\Sigma, Marked, Changed)$ značí platný evoluční stav sítě AIN. Celkový kontext výpočtu (simulace) – (CSC – Complete Simulation Context) je CSC=(State, AC, ActP), kde $State \in \{q_S, q_P, q_{SP}\}$ a $ActP \in OS$ je procesní objekt.*

Stav $State(CSC)$ určuje podobu celkového kontextu výpočtu CSC a činnosti systému AIN. V jednotlivých stavech nabývá CSC následujících tvarů:

- $State(CSC) = q_S$. Procesní část je *neaktivní*. CSC nabývá obecně tvaru:
$$(q_S,\ ((OS, R, <>, WaitingQ), M, CH), \epsilon). \tag{4.15}$$
- $State(CSC) = q_P$. Procesní část je *aktivní* a běží aktuální proces. CSC nabývá obecně tvaru:
$$(q_P,\ ((OS, R, ReadyQ, WaitingQ), M, CH), ActP). \tag{4.16}$$
- $State(CSC) q_{SP}$. Procesní část je *aktivní*, ale aktuální proces je suspendován. CSC nabývá obecně tvaru:
$$(q_{SP},\ ((OS, R, ReadyQ, WaitingQ), M, CH), ActP). \tag{4.17}$$



Obrázek 4.2: Stavový diagram přechodů CSC

4.4 Simulační stroj AIN

V této kapitole bude definována relace evoluce $\vdash_C$ nad celkovým kontextem CSC, funkce $reeval$ pro objekty typu proces a popíšeme chování celého systému složeného z datových a procesních simulačních objektů. Startovní CSC-kontext sítě $[OS, R]$ je

$$CSC_0 = (q_s,\ ((OS, R, <>, <>)M_{start}, \emptyset),\ \epsilon). \tag{4.18}$$

4.4.1 Sled výpočtu

Při krocích evoluce AIN-systému se přepíná mezi síťovou a procesní částí (odpovídá to stavům q_S a q_P). Je definován i mezistav (q_{SP}), který bude popsán odděleně. Běh výpočtu bude definován obecným přechodem ze CSC-stavu CSC_1 do CSC-stavu CSC_2 operací $\vdash_C$. CSC evoluce sítě přechází mezi stavy $\{q_P, q_S, q_{SP}\}$ podle schématu na obrázku 4.2.

Pro vedení procesů ve frontách zaveďme operace $(\Sigma = (OS, R, ReadyQ, WaitingQ), p) \mapsto \Sigma_2$:

- $removeProces(\Sigma, p)$ – odstraňuje proces p z front.
  $\Sigma_2 = (OS, R, ReadyQ \setminus \{p\}, WaitingQ \setminus \{p\})$.
- $setProcessReady(\Sigma, p)$ – vloží proces p do fronty $ReadyQ(\Sigma)$ (a případně jej odstraní z jiné fronty).
  - Je-li $p \in set(ReadyQ)$, pak $\Sigma_2 = \Sigma$.
  - Je-li $p \in set(WaitingQ)$, pak $\Sigma_2 = setProcessReady(removeProcess(\Sigma, p))$.
  - Jinak $\Sigma_2 = (OS, R, < ReadyQ, p >, WaitingQ)$.
- $setProcessWaiting(\Sigma, p)$ – vloží process p do fronty $WaitingQ(\Sigma)$ obdobně jako u operace $setProcessReady$.

Reeval funkce procesu ($reeval_{process}$), $reeval_{process} : (\Sigma, so) \mapsto (Boolean, \Sigma_2), so \in \Sigma$

```
if (this ∈ ReadyQ(Σ) or this ∈ WaitingQ(Σ) or this = ActP) {
   if (ContinueC == TRUE)
        return (FALSE, setProcessReady(Σ, this));
      else
        return (FALSE, setProcessWaiting(Σ, this));
}
if (value(StartC) == TRUE) {
   Σ2 :=setProcessReady(Σ, this);
   Σ3 := chValue(Σ2, StartC, FALSE);
   return (TRUE,Σ3);
}
```

Funkci $reeval$ procesu vyvolá změna jednoho z atributů procesu ($StartC$ nebo $ContinueC$). Situace $StartC = TRUE$ zajistí *spuštění* procesu. Stav $ContinueC$ určuje, zda má být proces ve frontě připravených nebo pozastavených procesů.

Popišme chování stroje AIN v jeho jednotlivých stavech.

Chování stroje ve stavu q_S

Stav q_S je bez aktivních procesů, kdy se odvíjí pouze síťová část AIN, ale procesy se mohou aktivovat – při evoluci AIN dojde k překlopení hodnoty $StartC(p)$ z $FALSE \to TRUE$ u nějakého procesu $p \in OS$ (obecně u libovolného počtu procesů), případně odblokovat (přejít z fronty $WaitingQ$ do $ReadyQ$). Protože $StartC(p) \to p$ (příp. $ContinueC \to p$), pak se p dostane v dalším kroku do množiny $Marked$ kontextu AC. Tím se volá funkce $reeval_{process}$ procesu a systém se může dostat do stavu s neprázdnou frontou $ReadyQ$.

1. CSC_1 tvaru $(q_S, AC = ((OS, R, <>, WaitingQ), M, CH), \epsilon) \vdash_C CSC_2$ takový, že
$$CSC_2 = (q_S, AC_2, \epsilon) \tag{4.19}$$
a $AC \vdash AC_2$.
2. $CSC_1 = (q_S, AC = ((OS, R, ReadyQ, WaitingQ), M, CH), \epsilon)$, $set(ReadyQ) \neq \emptyset$, pak
$$CSC_2 = (q_P, AC, \epsilon). \tag{4.20}$$

Chování stroje ve stavu q_P

Stav q_P je stav, kdy běží proces $ActP$. CSC kontext má obecně tvar:

$$(q_P, AC = ((OS, R, ReadyQ, WaitingQ), M, CH), ActP). \tag{4.21}$$

Proces při běhu může modifikovat objekt z OS (zapsat do atributu $value$ datových simulačních objektů). Modifikace dat dočasně přeruší jeho běh a CSC se překlopí do stavu q_{SP}. Ukončí-li se proces bez zásahu do dat z OS, pak CSC může pokračovat v q_P stavu.

1. CSC_1 tvaru $(q_P, AC = ((OS, R, <>, WaitingQ), M, CH), \epsilon) \vdash_C CSC_2$,
$$CSC_2 = (q_S, AC, \epsilon). \tag{4.22}$$
2. CSC_1 tvaru $(q_P, AC = ((OS, R, < p_1, p_2, ... >, WaitingQ), M, CH), \epsilon) \vdash_C CSC_2$,
$$CSC_2 = (q_P, AC = ((OS, R, < p_2, ... >, WaitingQ), M, CH), p_1). \tag{4.23}$$
3. CSC_1 tvaru $(q_P, AC = ((OS, R, ReadyQ, WaitingQ), M, CH), ActP) \vdash_C CSC_2$,
$$CSC_2 = \{runUntilChanged(ActP)\}CSC_2. \tag{4.24}$$

CSC_2 v třetím bodě závisí na výsledku operace $runUntilChanged(ActP)$ – spustí proces $ActP$ nebo obnoví běh přerušeného procesu $ActP$. Výsledkem operace je množina změn (referát změn) $PChanges = \{(object_1, newValue_1), (object_2, newValue_2), ...\}$. Podle referátu změn se dokončí třetí bod evoluce ve stavu q_P takto:

1. $|PChanges| = 0$. Pak $CSC_2 = (q_P, AC, \epsilon)$

2. $|PChanges| \geq 1$. Pak se změny $\{newValue_1, newValue_2, ...\}$ musí zapsat do objektů $\{object_1, object_2, ...\} \subseteq OS$ funkcí $\Sigma_2 := chValueSet(\Sigma, PChanges)$.

$$chValueSet(\Sigma, PChanges) \mapsto \Sigma_2 \tag{4.25}$$

$$chValueSet(\Sigma, \emptyset) = \Sigma$$

$$chValueSet(\Sigma, (so, val) \cup pchanges) = chValueSet(chValue(\Sigma, so, val), pchanges)$$

AIN systém přechází do stavu q_{SP}:

$$CSC_2 = (q_{SP}, (\Sigma_2, \emptyset, CH(AC) \cup \{object_1, object_2, ...\}), ActP). \tag{4.26}$$

Chování stroje v mezistavu q_{SP}

Mezistav q_{SP} tvoří přechod z procesní do síťové části výpočtu, kdy síť přehodnocuje modifikované objekty $\{object_1, object_2, ...\}$ a jejich navazující datové objekty $object_1 \bullet \cup object_2 \bullet \cup ....$ (včetně $StartC$ a $ContinueC$ podmínek procesů).

Mezistav začíná CSC-kontextem (4.26), speciálně jeho síťovou částí $AC = (\Sigma, \emptyset, CH \cup \{object_1, object_2, ...\})$. Evoluce ve stavu q_{SP} končí v ustáleném kontextu, tedy:

1. $(q_{SP}, AC_U, ActP) \vdash_C (q_P, AC_U, ActP)$.
2. jinak $(q_{SP}, AC, ActP) \vdash_C (q_{SP}, AC_2, ActP)$ takový, že $AC \vdash AC_2$.

4.4.2 Ukončení výpočtu

Přirozeným koncem výpočtu je CSC-stav $CSC_{NatTermination} = (q_S, AC_F, \epsilon)$[5]. Druhým způsobem ukončení je ukončení závislé na platnosti předem explicitně zvolené podmínky. Zvolme simulační objekt $termination$ a upravme evoluční krok na $\vdash_{C-until(termination)}$ definovaný takto:

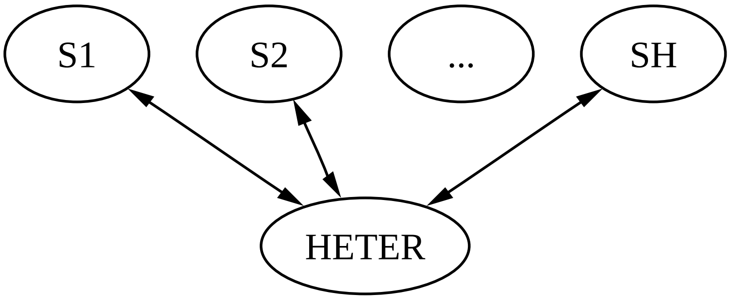
Definice 16. *Nechť $CSC_1 = (q, AC = ((OS, R, ReadyQ, WaitingQ), M, CH), ActP)$ je CSC-stav a $name \in OS$. Operace $CSC_1 \vdash_{C-until(name)} CSC_2$ je operace evolučního kroku taková, že:*

1. *Je-li $value(name) = TRUE$, pak $CSC_2 = CSC_1$.*
2. *Je-li $value(name) = FALSE$, pak $CSC_1 \vdash_C CSC_2$.*

Říkáme, že výpočet je *přirozeně ukončující*, pokud ze startovního stavu CSC_S dostáváme $CSC_S \vdash^+ CSC_{NatTermination}$. Výpočet je tak zvaně *ukončující na podmínku*, pokud pro specifikovanou podmínku $name$ dostáváme $CSC_S \vdash^+_{C-until(name)} CSC_{any}$ takový, že $CSC_{any} \vdash^+_{C-until(name)} CSC_{any}$ (stav, který se už nebude měnit). Evoluce $\vdash_{C-until(name)}$ má právě zachovat ten stav, kdy se hodnota objektu $name$ převrátila na $TRUE$.

Poznamenejme, že typickým představitelem ukončovací podmínky simulace je *časový rozsah simulace* – doba, kdy se má simulace ukončit. Podmínka $termination$ by pak měla podobu $termination = (time >= timeStop)$, máme-li rozsah modelového času definován jako interval $(0, timeStop)$.

[5] $AC_F = ((OS, R, <>, WaitingQ), \emptyset, \emptyset)$, kde se připouští neprázdná fronta $WaitingQ$.



Obrázek 4.3: Heterogenní systém $\{S_1, S_2, ..., S_H, HETER\}$

4.5 Heterogenní systém založený na AIN

V předchozím textu jsme definovali univerzální výpočetní systém AIN založený na komunikační síti AIN. Takovou síť $S_1 = [OS, R_1]$ lze spojit s jinou sítí $S_2 = [OS, R_2]$ tak, že vznikne jeden celek[6] $S = [OS_1 \cup OS_2, R_1 \cup R_2]$, ale pouze pokud jsou S_1 a S_2 izolované podsítě (definice 11).

Naším cílem je však budovat *provázané* systémy složené z komunikujících podsystémů. Takové subsystémy $[OS_s, R_s]$ se vyznačují existencí objektů $obj \in OS_s$, $\exists o \in Base(obj) : o \notin OS_s$ – objekt obj je závislý na objektu z jiného subsystému.

Mějme heterogenní (multi-) systém:

$$\{S_1, S_2, ..., S_H, HETER\} \tag{4.27}$$

složený z různých subsystémů $S_1, ..., S_H$ a řídicího systému $HETER$ (obrázek 4.3).

Smyslem tohoto rozhraní není zabezpečit *technicky* komunikaci mezi jeho částmi, ale položit platformu pro obecné propojení výpočtů. Rozhraní mezi subsystémy definujme jako sadu zpráv, kterým subsystémy rozumí.

4.5.1 Inicializace heterogenního systému

Nechť každý subsystém $S_i, i = 1..H$ obsahuje *lokální* a *globální* objekty (datové a procesní). Do propojení jsou zahrnuty pouze globální objekty $S_i globals = \{o_{i,1}, ..., o_{i,max_i}\}$.

Systém $HETER$ rozešle každému subsystému zprávu $S_i.getYourObjects()$, která po vyhodnocení vrátí množinu $S_i globals$. Celková množina všech globálních objektů je:

$$allObjects = \bigcup_{i=1..H} S_i globals \tag{4.28}$$

Subsystémy musí též přijímat zprávy adresující jejich globální objekty. Jsou to zprávy:

- *GetValue(o)* – vrať hodnotu atributu $value$ objektu o.
- *SetValue(o,val)* – nastav hodnotu atributu $value$ objektu $o,val \in \mathcal{U}$.
- *Name(o)* – vrať jméno objektu o.
- *RequiredNames(o)* – vrať množinu $\{name(obj) | obj \in Base(o)\}$ jmen objektů $Base(o)$.

[6] Pro jednoduchost budou $AINnoevent$ objekty z OS_1 a OS_2 představovat stejný objekt.

- *Reeval(o)* – proveď *reeval* pro objekt o. Vrať příznak $\{TRUE, FALSE\}$.
- *RunUntilChanged(op)* – prováděj proces *op* do jeho přerušení nebo ukončení. Vrací případný *referát změn*.

Řídicí systém $HETER$ sloučí všechny globální objekty všech subsystémů a vytvoří množinu simulačních objektů $OS = allObjects$. V této fázi specifikace již budeme s objekty pracovat pomocí jejich *jmen*, které objekty v heterogenním systému používají pro svou jednoznačnou identifikaci. Prostor jmen nechť je prozatím lineární, zavedení hierarchie prostorů jmen přináší až nadřazené vrstvy specifikované v dalších kapitolách.

Každý objekt má tedy jméno a vyžaduje množinu objektů daných jeho bází. Tu vrací obsluha zprávy $requiredNames(o)$. Systém $HETER$ v druhé fázi inicializace musí najít objekty s požadovanými jmény. Zaveďme funkci $findObject(name)$ vracející referenci na objekt jména $oName$:

$$findObject(oName) = \begin{cases} object & object \in allObjects, name(object) = oName \\ \epsilon & jinak \end{cases} \tag{4.29}$$

Objekt jména $oName$ nemusí být v systému $allObjects$ zastoupen, pak by samozřejmě heterogenní systém jako celek nebyl funkční. Dále však předpokládejme, že systém je *úplný* (viz. úplnost sítě AIN). Nechť je funkce $findObjects(names) = \{findObject(n) | n \in names\}$.

Relace vazeb R pro množinu objektů $OS = allObjects$ se vytvoří podle předpisu:

$$relationsOf(o) = \{(x, o) | x \in findObjects(requiredNames(o))\} \tag{4.30}$$

$$R = \bigcup_{o \in OS} relationsOf(o) \tag{4.31}$$

Tím je sestavena síť $[OS, R]$ všech globálních simulačních objektů zapojených do heterogenního systému.

Startovním kontextem CSC je $CSC_S = (q_S, ((OS, R, <>, <>), M_{Start}, \emptyset), \epsilon)$. Simulace heterogenního modelu začíná krokem $CSC_S \vdash_C CSC_{S+1}$ a může končit ve stavu $CSC_{NatTermination}$ nebo $CSC_{until(name)}$ s ukončovací podmínkou $name$.

4.5.2 Naznačení implementace

V předchozí kapitole se s objekty manipulovalo jako s pojmenovanými symboly. Tady naznačíme implementaci řídicího jádra systému *HETER*, tedy úroveň abstrakce o stupeň nižší.

Předpokládejme, že:

1. Subsystémy $S_1 ... S_H$ (dále jen subsystémy) jsou naprosto libovolné programové systémy. Budou adresovány podle svého čísla (v indexu).
2. Subsystémy běží na stejném počítači jako *HETER* nebo na jiném (distribuovaně).
3. Není žádné technické omezení v komunikaci mezi *HETER* a subsystémy. Neuvažujeme žádné specifické kódování zpráv a protokoly.
4. Simulační objekty jsou umístěny v jednotlivých subsystémech, *HETER* pouze *registruje* jejich zařazení do heterogenního modelu.

5. Každý subsystém rozumí sadě definovaných zpráv tvořících rozhraní mezi *HETER* a subsystémy.

6. Komunikace je pouze směrem *HETER* $\rightarrow$ subsystém. Subsystém zprávu přijme, zpracuje a vrátí výslednou hodnotu. *HETER čeká* na návratovou hodnotu.

Definujme rozhraní subsystému potřebné pro propojení se systémem *HETER*.

Rozhraní zpráv

- *GetYourObjects()* – vrací seznam dvojic $(name, type)$, kde $name$ je jméno objektu a $type \in \{data, process\}$.
- *GetValue(name)* – vrací obsah atributu $value$ objektu nazvaného $name$. Návratová hodnota je primitivní hodnota.
- *SetValue(name, val)* – opačná funkce ke $getValue(name)$. $val \in \mathcal{U}$ je primitivní hodnota. Návratová hodnota je val.
- *RequiredNames(name)* – vrací seznam jmen odpovídající bázi objektu se jménem $name$.
- *Reeval(name, values)* – aktivuje funkci $reeval$ pro simulační objekt jména $name$. Vrací primitivní hodnotu typu *Boolean*. Argument $values$ je seznam dvojic $(name,\ value) \subseteq \{(name, value) | name \in OS, value = value(name)\}$.
- *RunUntilChanged(name, values)* – subsystém spustí proces $name$. Proces běží, dokud nemodifikuje globální objekt. Vrací seznam dvojic $(name, value)$ jako referát změn. Pokud byl proces $name$ přerušen, pak při tomto volání pokračuje z místa přerušení.

Algoritmus inicializace

1. Sestav seznam OS jako seznam trojic $ProxyO = (subsystem, name, type)$. Zdroj dat je volání $getYourObjects()$ pro všechny subsystémy, kde $subsystem$ je číslo subsystému, kterému prvek náleží. Trojice $ProxyO$ bude zástupný (proxy) objekt všem fyzicky vyskytujícím se objektům v subsystémech.

2. Sestav R podle předpisu z předchozí kapitoly.

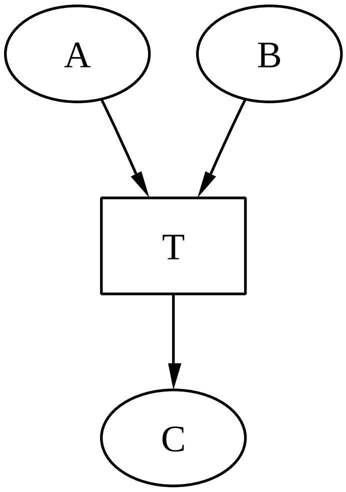
Běh simulace

Zůstává prakticky beze změny. Pro každé volání $reeval$ a $runUntilChanged$ je nutné sestavit seznam $(name, value)$ obsahující pro každé jméno z $requiredNames$ jeho aktuální hodnotu (ty se získají sérií volání $getValue(name)$).

Tato implementace je vytvořena bez nároků na efektivnost provozu. Zdrojem neefektivnosti je umístění hodnot objektů distribuovaně v jejich subsystémech a minimalizace paměťových nároků systému *HETER* (udržování seznamů žádaných jmen, hodnot).

4.6 Ukázky přizpůsobení systémů na specifikaci AIN

Metoda AIN ukazuje společné prvky všech výpočetních systémů. Aby byly propojitelné, musí se pro AIN upravit – nejedná se o *překlad* subsystému do podoby AIN, ale spíše o vytvoření *rozhraní* popsaného v kapitole 4.5.2.



Obrázek 4.4: P/T Petriho síť – příklad

4.6.1 P/T Petriho síť

Ukážeme přizpůsobení klasické P/T Petriho sítě pro AIN (obrázek 4.4). V následujícím převodu budeme místa P/T sítě považovat za celočíselné proměnné a přechody za akce (procesy).

V P/T Petriho síti $P = (\{a,b,c\},\{t\},\{(a,t),(b,t),(t,c)\}$ máme data $\{a,b,c\}$ a akci popsatelnou programem (kapacity míst pro jednoduchost vypouštíme):

```
if (a>0 and b>0) then
   begin atomic
     a := a - 1;
     b := b - 1;
     c := c + 1;
   end
```

kde tělo akce musí být provedeno atomicky.

Vyjděme z definice P/T Petriho sítě: $\Sigma = (P,T,F,M_0,W,C)$. P je množina míst, T je množina přechodů, $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ a $M_0 : P \rightarrow N \cup \{0\}$ je počáteční značení. $W : F \rightarrow N$ je ohodnocení hran (implicitně je jedna) a $C : P \rightarrow N$ je kapacita míst. $N = \{0,1,2,3,....\}$

Pro každou P/T síť vytvoříme počáteční registraci pro AIN takto:

1. Každé místo $p \in P$, $P = \{p_1, p_2, ...\}$ bude vystupovat jako datový objekt se jménem p.
2. Podobně pro každý přechod $t \in T$ vytvoříme process s objektem $StartC$ podle definice proveditelnosti přechodu.

Přechod $t \in T$ v síti Σ je proveditelný pokud $\forall p \in \bullet t : M(p) \geq W(p,t)$ a současně $\forall p \in t\bullet : C(p) \leq M(p) + W(t,p)$. To formulujeme podmínkou $StartC$ procesu, která má obecný tvar $StartC(t) = presetCond(t) \wedge posetCond(t)$, kde

$$presetCond(t) = \bigwedge_{p \in \bullet t} (M(p) > W(p,t)) \tag{4.32}$$

$$posetCond(t) = \bigwedge_{p \in t\bullet} (C(p) \leq M(p) + W(t,p)) \tag{4.33}$$

Vytvoří se základ podsítě $AIN_{PT} = [OS_{PT}, R_{PT}]$ takové, že

$$OS_{PT} = P \cup T \cup \{StartC(t) | t \in T\} \tag{4.34}$$

a $R_{PT} = \{(StartC(t), t) | t \in T\}$. Provázání mezi datovými objekty P/T sítě a startovními podmínkami procesů zabezpečí inicializační algoritmus z kapitoly 4.5.1.

Proto můžeme prohlásit, že libovolná P/T Petriho síť je přímo zobrazitelná na síť AIN. Jediná námitka by se mohla vyskytnout při interpretování přechodů Petriho sítě, kdy se z množiny $T_{now} \subseteq T$ aktuálně proveditelných přechodů vybírá náhodně jeden pro provedení. V AIN CSC-kontextu to odpovídá zařazení nově spuštěných procesů do fronty *ReadyQ*, kde pořadí také není předem určeno. Pokud by bylo striktně vyžadováno náhodné vybírání přechodu k provedení, pak lze P/T síť převést na systém

$$AIN_{NPT} = [OS_{NPT} \cup \{GlobalStartC, GlobalProcess\}, \{(GlobalStartC, GlobalProcess)\}] \tag{4.35}$$

takový, že $GlobalStartC = \bigvee_{t \in T} StartC(t)$. Pak je výběr a provádění přechodů zcela v režii procesu $GlobalProcess$.

4.6.2 Prolog

V Prologu budeme chtít využít především jeho mechanismu inference a vyhledávání v datech. Jako rozhraní k Prologu můžeme využít procesy.

Mějme predikát $prologp(inpupset, outputset)$, kde

$$inputset(prologp) = \{inp_1, ..., inp_{pimax}\} \tag{4.36}$$

je množina navázaných proměnných a

$$outputset(prologp) = \{outp_1, ..., outp_{pomax}\} \tag{4.37}$$

je množina volných proměnných. Prolog bude registrovat proces *prologp* s

$$requiredNames(prologp) = inputset(prologp) \tag{4.38}$$

a množinu $inputset(prologp) \cup outputset(prologp)$ datových položek.

Predikát *prologp* vytvoří specifickou podsíť

$$[\{prologp\} \cup inputset(prologp) \cup outputset(prologp), \{(b, prologp) | b \in inputset(prologp)\}] \tag{4.39}$$

S každou změnou báze $inputset(prologp)$ se zařadí výpočet predikátu *prologp* do správy procesů a po vyhodnocení se v proměnných $outputset(prologp)$ objeví nové hodnoty.

4.7 Simulační základ heterogenních modelů

V této kapitole budeme zkoumat klasické prvky spojité, diskrétní a kombinované simulace v heterogenních modelech ve formě sítě AIN.

V heterogenních modelech chceme inteligentním způsobem propojovat jednotlivé submodely tak, že sdílejí určité informace (objekty). Nejdůležitějším prvkem simulace je bezesporu *modelový čas*. V kapitole 5 o modelování objektů reálného světa se tato problematika rozebírá podrobněji. Pro celou teorii kolem metody AIN a jazyka HELEF je typické, že nespecifikuje implicitně přístup k řízení času, ale specifikaci šasové osy ponechává v kompetenci modelu. Má pro řízení diskrétních a spojitých simulačních modelů pouze určité doporučení.

Modelový čas je datový objekt *time* vytvářející specifickou podsíť

$$[\{time, timestep, AINnoevent\}, \{(AINnoevent, time)\}] \tag{4.40}$$

Je definovaný jako $time(AINnoevent) = time + timestep$. V CSC-ustáleném stavu se aktivují všechny objekty $AINnoevent\bullet$ a tím i posun času. Pak se zase aktivují všechny objekty množiny $time\bullet$. Procesy řídí svůj běh nastavováním podmínky $ContinueC$, která může specifikovat časový okamžik zařazení do fronty $ReadyQ$. Tento způsob řízení simulace umožňuje provádět libovolný kombinovaný (diskrétní a spojitý) simulační experiment.

V systému $S = [OS, R]$ se mohou objekty pracující s časem chovat takto:

1. Integrátory $I_{cont} = \{i_1, i_2, ...\}$; $I_{cont} \subseteq OS$ spojité části modelu S jsou napojeny na $time$. $\{(time, i) | i \in I_{cont}\} \subseteq R$

2. Procesy specifikují svou podmínku běhu $ContinueC$. Například, pro čekání t_w bude

$$ContinueC = time \geq timeNow + t_w \tag{4.41}$$

3. Čas *time* se inkrementuje v ustáleném stavu evoluce sítě.

4.8 Příklad jazyka pro programování AIN

Krátce naznačíme jednoduchý jazyk pro programování modelů ve stylu AIN. Jeho konečná podoba, která bude prezentována v kapitole 6, byla nazvána HELEF (*He*terogeneous *L*anguage *E*nvironment, *F*inal version). Metoda AIN definuje sémantiku jazyka HELEF.

Deklarace

```
name = expression
```

vytvoří podsíť $[\{name\}, \{(expression, name)\} \cup Relations(expression)]$. Podobná deklarace, ovšem bez vazby $expression \rightarrow name$ se zapíše:

```
name(condition) = expression
```

která vytvoří podsíť

$$[\{name, condition, expression\}, \{(condition, name)\} \cup Relations(condition)] \tag{4.42}$$

4.9 Cykly

U výpočetních systémů založených na grafech se objevuje námitka o možných *cyklech*, případně *rychlých smyčkách*. Může se soudit, že cykly ohrožují stabilitu výpočtu a mohou vést k *zacyklení výpočtu*. Pod pojmem cyklus rozumíme:

- Užitečný výpočet – opakovaně prováděný úsek programu. Chceme mít kontrolu nad počtem opakování cyklu (příkazy *for, while* v jazycích typu C; rekurze).

- Smyčka ve spojitých simulacích s integrátorem (v čase přerušovaný cyklus), správně prováděná rychlá smyčka ve spojitých simulacích.
- Chyba v programu (chybně stanovená podmínka opakování cyklu).
- Nekonečný cyklus – proces opakující sekvenci kroků, u kterého neuvažujeme jeho ukončení

Kterýkoliv programovací prostředek umožňuje napsat chybný program, a tedy i program obsahující chybný cyklus.

Síť $S = [OS, R]$, kde $OS = \{a\}$ a $R = \{(a, a)\}$ je jednoduchá ukázka cyklu nebo také rychlé smyčky. Je-li funkcí pro výpočet a funkce $a + 1$, pak se síť nikdy nedostane do ustáleného stavu. Evoluce bude probíhat takto:

$$((OS, R), \{a\}, \emptyset) \vdash ((OS, R), \emptyset, \{a\}) \vdash ((OS, R), \{a\}, \emptyset) \vdash ... \tag{4.43}$$

Pokud je S podsítí větší sítě, pak cyklus (a, a) v žádném případě neovlivní evoluci zbytku sítě, protože ta běží v krocích $Marked \to Changed \to Marked$. Otázkou je však smysluplnost takové deklarace $a = a+1$. Celou situaci lze řešit řízením cyklu explicitní podmínkou přehodnocení deklarace.

Deklaraci

```
a=a+1
```

nahradíme deklarací s řídící podmínkou (explicitní ...)

```
a(condition)=a+1
// například
a(a<100)=a+1
```

a v síti AIN propojíme objekty $\{a, condition, a+1\}$ do sítě $\{(a, condition), (condition, a)\}$, která tvoří cyklus s pevným bodem.

Pokud však síť S obsahuje cyklus, pak cyklus nezastaví běh simulace, pouze nemusí dojít k události $AINnoevent$ (plyne z definice operace $\vdash_C$). K podobné situaci může dojít i v modelech DEVS.

4.10 Srovnání metod AIN a heterogenity DEVSu

DEVS naznačuje možnost heterogenního propojování prostřednictvím obsluhy čtyř základních akcí. Mnohem důležitější než definovat rozhraní je ukázat, jak navrhnout systém zapojitelný do celku, respektive jak *modifikovat existující* systém. Základní rozdíl mezi DEVS a AIN je rozdíl mezi stavově (a událostně) a procesně vedenou simulací.

DEVS má jako základní atribut modelový čas – ten slouží pro řazení událostí do kalendáře událostí. AIN čas implicitně neobsahuje a řazení ve frontách je dáno vazbami mezi objekty, ze kterých plyne časová souslednost. V tomto ohledu je AIN mnohem obecnější.

Sledujeme-li vnořené simulace, pak oba formalismy umožňují vnořovat časovou osu – instanciovat celou simulaci a pracovat s ní jako s jedním blokem či objektem. Vedle toho AIN umožní vést několik nezávislých časových os v jedné úrovni simulace – určitá aplikovatelnost, i když není běžná, se diskutuje v kapitole 5.6.

4.10.1 Ukázka transformace AIN na DEVS

Vycházejme z existující sítě $AIN = [OS, R]$ obsahující datové a procesní simulační objekty. Ukážeme část převedení modelu v AIN na model složený z atomických bloků DEVS, kde zachováme původní centralizované řízení výpočtu metodou evolučních kroků.

Transformací z AIN do DEVS rozumíme převod $AIN = [OS, R]$ na množinu bloků $DataAB \cup ProcessAB \cup ControlB$, kde:

- $DataAB$ – je množina atomických bloků datových objektů.
- $ProcessAB$ – je množina atomických bloků procesních objektů.
- $ControlB$ – je řídicí DEVS-blok s funkcí ekvivalentní operaci $\vdash_C$.

Pro každý datový simulační objekt $obj \in OS$ přidáme do $DataAB$ blok:

$$DataDEVS(obj) = (\{reeval, step, m0\} \cup Base(obj), \{q_{idle}, q_{dirty}, q_{reeval}\}, \{value\}, \quad (4.44)$$
$$, \delta_{ext}, \delta_{int}, ta_{data}).$$

Vnější přechodová funkce δ_{ext} bloku $DataDEVS(obj)$ je definována:

- $\delta_{ext}(q_{idle}, base) = q_{dirty}$ pro $\forall base \in Base(obj)$.
- $\delta_{ext}(q_{dirty}, step) = q_{reeval}$.
- $\delta_{ext}(q_{reeval}, reeval) = q_{idle}$.

Objekty obj,$Base(obj) = \emptyset$ přidají přechody:

- $\delta_{ext}(q_{idle}, m0) = q_{dirty}$.
- $\delta_{int}(s) = s$, $\forall s \in S$.

Výstupní funkce λ provede výstup $value(obj)$ během přehodnocení objektu obj ze stavu q_{reeval} do stavu q_{idle}. Funkce *time advance* se pro blok $DataDEVS(obj)$ nedefinuje (vrací prázdnou hodnotu).

Řídicí blok $AinDEVS$ vede výpočet přes všechny objekty $obj \in OS$ algoritmem $devsEval$:

1. Hromadné rozeslání zprávy $m0$ (reagují všechny objekty $o \in OS, Base(o) = \emptyset$).
2. Hromadné rozeslání zprávy *step*.
3. Hromadné rozeslání zprávy $reeval$.

Algoritmus *devsEval* simuluje operaci evolučního kroku $\vdash$. Toto je neúplná varianta přepisu výpočetního modelu AIN na výpočetní model DEVS. Slouží jako ukázka srovnání metody AIN s formalismem DEVS.

Kapitola 5

Analýza modelování objektů reálného světa

V této kapitole bude popsána metoda modelování struktury reálného světa složeného z objektů typu prostředí, entita a vlastnost. V kontextu heterogenního modelování se jedná o způsob sestavování heterogenních modelů z jednotlivých submodelů. Jsou rozlišeny metody modelování vedoucí k jednoúčelovým a znovupoužitelným modelům a motivace pro vytváření znovupoužitelných modelů. Ukáže se, že pro znovupoužitelnost modelu je důležité vytvářet modely objektů světa způsobem, který zapouzdřuje pouze jejich vnitřní atributy nepopisující vztah objektu k jeho okolí. Vazba mezi objektem (entitou) a okolím (prostředím) bude specifikována třetím typem nezávislého objektu, který je zde nazýván pravidlo nebo také vlastnost.

V této kapitole bych rád analyzoval několik problémů souvisejících s modelováním struktury reálného světa na abtraktnější úrovni, které by mělo v budoucnosti vést na modely s prvky umělé inteligence, umělého života a robotiky. Pro začátek si připomeňme definici modelu systému:

Model systému A je jiný systém B, který je mu z určitého hlediska podobný.

Nepřímým důsledkem této definice je skutečnost, že každý člověk nahlíží na modelování poněkud jiným způsobem – jeho systém B může být velmi odlišný od systému B jiného modelujícího člověka. Nejde tak ani o formální *přesnost* modelu, ale o způsob jeho vytvoření. Tato kapitola se zabývá z větší části pouze vytvářením modelů bez jejich simulace. Nejedná se nám o experimentování s modelem v čase, ale o určitou formu automatizované analýzy modelu složeného z objektů a deklarovaných pravidel pro jejich provázání. Inspirací pro tyto metody může být například constraint-based programming nebo simulační nástroj Labfit vyvinutý ve Smalltalku.

V této kapitole vycházíme z předpokladu, že jsme již dnes schopni na velmi teoretické úrovni vytvořit model libovolného diskrétního nebo spojitého systému. Jediným omezujícím faktorem je jeho cena jako souhrn finančních a časových nákladů na jeho vytvoření. V předchozích dvou kapitolách byla zavedena metodologie heterogenního modelování, které nám umožní lépe využít dostupné modelovací prostředky a modelovat produktivněji. Tato kapitola rozšiřuje metodologii heterogenního modelování o další přístupy v modelování objektů reálného světa a zavádí metodologii organizace modelu do vzájemně provázaných struktur nazývaných prostředí, entita a vlastnost. Budeme vycházet z určitého pozorování reality a na jeho základě povedeme určitou úvahu o vhodném stylu modelování reality, které je zde nazýváno *přirozeným (reálným) modelováním reality.*

5.1 Motivace pro přirozené modelování reality[1]

Jedním z velmi důležitých faktorů kvality (možná lépe – užitečnosti) modelu je případná *znovupoužitelnost modelu.* Znovupoužitelnost, kromě usnadnění práce a zvýšení produktivity, přispívá ještě jedním důležitým vlivem – znovupoužitelný model definuje základní stavební bloky modelů, ze kterých je možné vybudovat knihovnu základních simulačních abstrakcí vhodných pro standardizaci určitých postupů v modelování. Inspiraci berme například z formalismu DEVS a jeho modularity.

> Model (nebo jeho fragment – submodel) je znovupoužitelný (přenositelný), pokud se může okamžitě stát součástí jiného modelu s minimálním nárokem na jeho úpravu.

Vezměme si příklad generátoru transakcí v systémech hromadné obsluhy (řešeno v SIMLIB/C++):

```
class Generator : public Event {
      void Behavior() {
           // vlastní akce
           ...
           // naplánování dalšího výskytu události
           Activate(Time+Exponential(X));
      }
}
```

Tento zápis je vyjádřen kódem v jazyce C++, ve kterém je knihovna SIMLIB/C++ implementována. Objektová orientace dodává do modelování základní prostředky znovupoužitelnosti, ale zde uvedená třída *Generator* zcela určitě přenositelná není (specifikace "vlastní akce" a přeplánování úlohy). Teoreticky je však možné implementovat *vzor generátoru* a specifika konkrétního generátoru přenést do jeho parametrů:

```
Generator myGen({akce}, Exponential(X));
```

což je v C++ obtížně řešitelné (možné řešení je s využitím maker (templates)).

Pro znovupoužitelnost modelů je důležité budovat modely s *minimálním zabudovaným vztahem* k jejich okolí (současné simulační modely jsou opakem tohoto přístupu). Tuto skutečnost bych v modelování považoval za nejpodstatnější a jeho analýza je náplní této kapitoly. Modelování objektu s výhradně jeho vlastními atributy je v této práci nazýváno *reálným modelováním.* Pojmem *vlastní atribut* objektu rozumíme minimální množinu atributů, které objekt má nezávisle na svém okolí. Pokud budeme tento objekt modelovat v různých prostředích, pak jeho vlastní atributy získáme průnikem všech těchto modelů. V této kapitole se pokusíme ukázat, že reálné modelování zvyšuje předpoklady pro znovupoužitelnost vytvořených modelů.

[1] V anglické literatuře nacházíme pojem *realistic modelling.* Například článek http://www.fz-juelich.de/nic-series/volume9/bluemer.pdf/ ukazuje realistické modely ve fyzice polovodičů. Pojem realistického modelování se objevuje i v oblasti virtuální reality (http://www.realisticmodelling.com/).

Jako základní platforma pro reálné modelování poslouží objektově orientované prostředí rozšířené o paradigma AIN z kapitoly 4. V tomto bodě bych se rád vrátil k otázce beztřídních OO jazyků (kapitola 2.1.1 a 2.4 – jazyk Self). Vhodnost třídního nebo beztřídního objektového přístupu k reálnému modelování nelze posuzovat obecně metodologicky, ale podle praxe existujících OO simulačních jazyků. Rád bych ukázal, že navzdory zkušenostem a praxi s třídními OO jazyky je beztřídní objektová orientace bližší úvahám této kapitoly.

Srovnáme-li simulační model systému vytvořený v beztřídním a třídním OO jazyce, pak je rozdíl následující: beztřídní OO jazyk popisuje množinu objektů s přímou návazností na modelovaný systém (každý existující model systému má ekvivalent v simulačním modelu). Třídní OO jazyk je stavěn pro popis obecné existence věcí a objekt vzniká až instanciací. Je tedy realitou objekt nebo třída? Mluvíme-li o znovupoužitelnosti a očekávané konkrétnosti beztřídního OO jazyka (která vedy spíše k jednoúčelovosti), pak docházíme k rozporu. Uvidíme, že tento paradox je pouze zdánlivý a stejně, jako je předmět v realitě znovupoužitelný, tak i objekt-model může být použitelný opakovaně. Zkusme alespoň na moment nebrat modelování jako striktně formální disciplínu a připusťme, že si s modely můžeme hrát jako s věcmi běžného života.

Na modelování můžeme nahlížet z mnoha pohledů abstrakce, což také ovlivňuje přesnost modelu v závislosti na požadované aplikaci. Tato práce není o modelování náročných fyzikálních jevů, ale o popisu obecných logických vztahů mezi objekty.

Prvotní úmluvou této kapitoly je skutečnost, že nediktujeme stav reality, ale snažíme se realitu popsat. Proto nelze vytvořit umělou teorii o modelování reality, ale je nutné z reality vytvořit teorii, která ji má šanci popsat.

Vymezení rozsahu aplikovatelnosti

Teorie uvedená v této kapitole dává návod k modelování *nevýpočetní* části modelu – k té, kterou nazýváme *znalostní část modelu*. Cílem je realisticky popsat obtížně popsatelný reálný i abstraktní svět lidsky přijatelnou formou.

Tyto modely (přímo) nevedou k masivním výpočtům soustav diferenciálních rovnic, kde je nejdůležitější složkou práce zajistit dostatečný výpočetní výkon. Tato teorie je spíše úvodem do modelů inteligentních entit, umělého života, řízení, robotiky a případně i sociotechnických systémů. Je nadstavbou metody AIN, čerpá však i hodně ze závěru druhé kapitoly o interaktivních a perzistentních simulacích. Většina úvah této práce sice vede k realizaci s využitím jazyka HELEF, ale předpokládá se i jejich obecnější platnost.

5.2 Základní koncept

Modelujeme-li reálný svět, docházíme k poznání, že je složen z *fyzických objektů* a *pravidel*. V OO modelování používáme *objekty* k programování, veškeré naše konstrukce jsou *objekty*, a proto i pravidla můžeme specifikovat pomocí objektů – nazveme je například *abstraktní objekty*. V praktickém modelování můžeme rozlišit tři typy objektů:

1. *Fyzický objekt* – hmotný objekt systému. Jeho model je pravděpodobně jednoznačně formulovatelný (objekt reality přenesený na objekt v modelu).

2. *Abstraktní objekt* – nehmotný objekt systému. Abstraktní objekt představuje snahu objektově realizovat neobjektový problém (vazby, pravidla). Jejich vliv na celkovou podobu modelu je hlavním tématem této kapitoly.

3. *Pseudo-abstraktní objekt* – abstraktní objekt, kterému dáváme z pragmatických důvodů (omezenou) fyzickou podobu (například model integrátoru). V zásadě je tato třída okrajová a uvádíme ji pouze pro úplnost.

Pro začátek zaveďme pojem množiny $World$ objektů reálného světa, respektive jeho aktuální části tvořící modelovaný systém. Svět $World = \{o_1, o_2, ...\}$ je konečná množina fyzických $phys(World) \subseteq World$ a abstraktních $abstr(World) \subseteq World$ objektů. Množiny $phys(World)$ a $abstr(World)$ jsou disjunktní (se pseudo-abstraktními objekty budeme pracovat jako s fyzickými objekty).

Modelování fyzických objektů je poměrně zvládnutý problém, jenž je v současné době založen především na modelování příslušných přímých fyzikálních zákonů (jevů) – vznikají tak různě složité, ale *jednoúčelové* modely. Zavedení abstraktních objektů do modelování zobecňuje *celý model* na jeho reálnou a znovupoužitelnou podobu.

Okolí objektu – směrem k pojmu *prostředí*

Každý fyzický objekt o světa $World$ má své okolí – množinu objektů $nei(o) = \{no_1, no_2, ...\}$, $nei \subseteq World$, které ho ovlivňují. Vztah objektu k jeho okolí můžeme modelovat relací

$$neiRelations(o) = \{(o, no_i) | no_i \in nei(o)\} \subseteq World \times World \tag{5.1}$$

Relace $neiRelations(o)$ musí být chápaná jako *naše pozorování světa* (z pozice jednoho pozorovatele). Udělejme si pro každý fyzický objekt světa soupis objektů v jeho okolí (vazeb "být v okolí"). Na základě našeho pozorování bude posléze postavena logická struktura modelu světa.

Všechny vztahy mezi všemi objekty světa $World$ tvoří relaci

$$neiAll(World) = \bigcup_{i \in World} neiRelations(i) \tag{5.2}$$

která bude zřejmě reflexivní, tranzitivní a symetrická. Tranzitivita relace $neiAll$ je ovšem pravdivá pouze do jisté míry – sedím-li v místnosti bez oken, pak zdi místnosti jsou pochopitelně mým okolím. Ta zeď je však i okolím objektů ve vedlejší místnosti. Je proto nutné odlišit v relaci "být-v-okolí" prvky tvořící jistou *hranici* mezi dvěma podmnožinami světa.

Definice 17. *Mějme svět $World$ a jeho dvě disjunktní podmnožiny s_1 a s_2. Objekt $border \in World$ se nazývá* hranicí mezi prostředími *tvořenými množinami s_1* a s_2 *objektů, když platí:*

$$\{\forall i \in s_1 : border \in neiRelations(i)\} \wedge \{\forall j \in s_2 : border \in neiRelations(j)\} \wedge \tag{5.3}$$

$$\wedge \{not \exists a \in s_1, b \in s_2 : [a \in neiRelations(b) \vee b \in neiRelations(a)]\}$$

Řečeno slovy, *border* je objekt, který je v okolí prvků z množin s_1 a s_2, ale neexistují prvky $a \in s_1$ a $b \in s_2$ takové, že by byly vzájemně v okolí. Objekt *border* je jediný společný prvek pro dvě seskupení prvků s_1 a s_2. Tvoří mezi nimi *hranici*.

Definice 18. *Mějme svět $World$, množinu $subw \subset World$ a objekt $border \in subw$. Množina $subw$ tvoří* ohraničené prostředí s hranicí <u>border</u>, *pokud současně platí:*

1. $\forall o \in World \setminus subw : not(\exists) s_o \in subw \setminus \{border\} : o \in neiRelations(s_o)$
2. $\exists o \in World \setminus subw : o \in neiRelations(border)$

Komentováno slovy:

1. Relace "být v prostředí" nespojuje *vnitřní prvky* prostředí s prvky *mimo prostředí* (až na hranici) – prostředí je uzavřeno a zapouzdřeno.
2. Mimo rozsah prostředí *subw* se očekává, že bude existovat prvek o tvořící (vnější) okolí prvku ohraničení *border* – prostředí nevyplňuje celý prostor světa a není od něho izolováno.

Tím je definován základ konceptu prostředí jako struktury, ze kterých je složen reálný svět. Smyslem bylo konstatování, že v OO modelu reality vystupují objekty tvořící hranice mezi jinými objekty, které v důsledku *ohraničují* skupiny objektů do *prostředí objektů*.

Prvky prostředí

Prostředí je především *okolím objektů*. Prostředí $env \in World$ definujme jako abstrakní objekt, který má ve svém okolí hraniční objekt $border(env)$ a množinu objektů

$$objects(env) = \{border(env), o_1, o_2, ...\} \tag{5.4}$$

Tím, že ve světě existuje objekt env_2 takový, že $border(env)$ je jeho prvkem, vzniká další odvozená vazba mezi objekty – "být podprostředím prostředí".

Definice 19. *Prostředí env_{sub} je podprostředím jiného prostředí env, pokud platí:*

1. $border(env_{sub}) \in objects(env)$.
2. $border(env_{sub}) \neq border(env)$.

Tím vytváříme jednoduchý model struktury reality jako *hierarchie vnořených prostředí*.

Závěr základního konceptu

V tomto okamžiku můžeme shrnout záměr základního konceptu reálného modelování reálného světa (RMRW - Real Modelling of Real World) do bodů:

- Rozlišujeme fyzické a abstraktní objekty (případně i pseudo-abstraktní).
- Objekty mají spolu základní vztah "být v okolí" (objekt o je v okolí objektu A píšeme $o \in neiRelations(A)$ a platí i $A \in neiRelations(o)$).
- Tento vztah vytváří pojmy *prostředí, podprostředí, hierarchie prostředí.*

V dalším textu budou zavedeny pojmy *entita* a *vlastnost* modelující pohyblivé objekty a dynamické vazby mezi objekty. Teorie budovaná v této kapitole je založena na sledování reality přes charakteristiku "být v okolí". Je to pouze jedna z mnoha vhodných projekcí reálného světa do abstraktního modelu.

5.3 Hierarchie prostředí

V předchozí kapitole bylo prostředí studováno z hlediska fyzických objektů a jejich vztahů "být v okolí". Druhou linií vztahů mezi objekty je jejich napojení na objekty abstraktní. Hlavní účel zavedení prostředí je v seskupování množiny pravidel – abstraktních objektů, které budou zprostředkovány obsaženým fyzickým objektům. Pravidla prostředí $env = (objects, border)$ nechť tvoří množinu abstraktních objektů $rules(env) \subseteq objects(env)$.

Pro účely další manipulace s pravidly definujme:

Definice 20. *Pravidlo rule je abstraktní objekt s atributy:*

1. *Name* – identifikace *pravidla (pravidlo je* pojmenovaný objekt*).*
2. *Value – obsah pravidla.*

Stejně, jako fyzický objekt reprezentuje *přítomnost* věci v prostředí, tak i abstraktní objekt reprezentuje určitou znalost o prostředí, která je v něm zahrnuta. Znalost je určena svou hodnotou (například teplota). Pravidlo budeme chápat jako dynamický atribut prostředí, který je v prostředí platný dočasně. Přistupme ke konkretizované definici pojmu prostředí:

Definice 21. *Prostředí env je pojmenovaný objekt s atributy:*

1. *Name – jméno prostředí.*
2. *Objects – fyzické a abstraktní objekty v prostředí (entity, podprostředí, pravidla).*
3. *Border – objekt hranice prostředí s jeho nadprostředím.*
4. *Sup – reference na objekt nadprostředí $sup(env)$ prostředí env. $Sup(env) \in World \cup \{\epsilon\}$.*

V dalším textu budeme zkoumat umístění konkrétních objektů v hierarchii prostředí a rozsah jejich působnosti. Protože v prostředí identifikujeme objekty jejich *jmény*, pak pro prostředí *env* zavedeme pojem *prostor jmen* – $namespace(env) = \{name(i) | i \in objects\}$.

Studujme vliv pravidel prostředí *env* na jeho podprostředí env_{sub}. OO technologie inspirované pozorováním reality zavádějí platnost atributu prostředí i v jeho podprostředích (bloková struktura, dědičnost). Z toho plyne, že aktuální stav pravidel prostředí *env* je:

$$allRules(env) = rules(env) \cup allRules(sup(env)) \tag{5.5}$$

$$allRules(\epsilon) = \emptyset$$

Prostory jmen

K pravidlům budeme přistupovat prostřednictvím jejich *jmen*. Na základě množiny jmen zaveďme následující podmnožiny (charakteristiky):

- $Generated(env) = \{name(i) | i \in objects\}$.
- $Inherited(env) = allRules(sup(env)) \setminus local(env)$.
- $Local(env)$ výčet jmen lokálních pravidel (množina lokálních jmen se bude podrobně specifikovat v kapitole 6).

Charakteristika *generated*(*env*) zahrnuje všechny pojmenované objekty umístěné do prostředí, charakteristika *inherited*(*env*) určuje objekty převzaté (zpřístupněné, zděděné) z nadprostředí, které jsou omezovány explicitně výčtem charakteristiky *local*(*env*). Podrobnější popis prostorů jmen má smysl vázat až na konkrétní modelovací nástroj, proto jsou podrobnější definice umístěny až v kapitole 6.4 o jazyce HELEF. Pro ukázku uveďme příklad vztahu dvou prostředí. V jazyce HELEF prostředí modelujeme zápisem:

```
environment {
   objects;
}
```

Prostředí *room* ukazuje velmi jednoduché řešení hranice mezi prostředími, která omezuje platnost objektu *light* v prostředí *room*.

```
world = environment { // prostředí svět
   entity sun {} // obsahuje entitu sun
   room = environment { // podprostředí room
      local light; // light je zde lokální objekt
      // do room nepronikᚠvliv light z entity sun
   }
}
sun = entity {
   exports light; // vytváří objekt light a předává do world
   light = TRUE
}
```

Tento příklad zároveň ukazuje odlišnou reprezentaci stavu znalostí v modelu. Obvykle každou znalost doplňujeme o metaznalost – příznak její platnosti. V této metodě je neplatnost znalosti vyjadřována její nedostupností v daném prostředí.

Shrnutí

Prostředí modeluje strukturu a pravidla světa, zatím bez objektů typu entita. Sekundárním smyslem prostředí je skutečnost, že prostředí tvoří hierarchie prostředí oddělené hranicemi (ohraničením). Hraniční objekt specifikuje přenesení (dědění) vlastností mezi prostředími. Příkladem je hraniční objekt zeď, který specifikuje, že nepropouští světlo.

5.4 Entity

Entita je fyzický nebo (pseudo-)abstraktní objekt, který odvozuje svůj stav z pravidel prostředí, v němž je umístěn. Základní vlastnost entity je, že smí přejít z jednoho prostředí do druhého (entita migruje).

Můžeme rozlišovat entity aktivní a pasivní. Aktivní entita je prvek vykonávající nějakou *činnost* – říkáme, že entita má své vlastní chování. Řekněme, že vlastní chování entity je projevem její existence bez toho, aniž by byla v interakci s jiným objektem. Pojem *chování* zastřešuje pojem *proces* jako souhrn vlastního chování entity a jejích reakcí na podněty z okolí. Příkladem aktivní entity je transakce v systémech hromadné obsluhy. Pasivní entita nemá aktivní (vlastní) chování, ale může nějak reagovat na vnější podněty[2].

Toto dělení entit uvádím na podporu neformální definice entity (případná formální definice by totiž nebyla přínosná). Základní aspekty entity (pohyblivost, chování a vazba na prostředí) vyplynou z následujícího textu.

Vztah entity a prostředí

Smyslem modelování by mělo být vytváření efektivních, maximálně obecných a současně přesných modelů. Případnou obecnost modelu snižuje právě modelování vztahu modelu k jeho okolí.

Entita odvozuje své chování z množiny pravidel svého prostředí. V OO modelování objekt definuje *komunikační rozhraní* a přes něj dotazuje okolí o informace. V konvenčním OO modelování však musí být rozhraní předem definováno. To omezuje submodel na implementačně kompatibilní okolí submodelu.

Zkusme přirozeně modelovat peříčko poletující ve větru. Pro tentokrát se nezajímejme konkrétně o jeho *pohybovou rovnici*, ale o vztah peříčka a jeho prostředí. Kdo v tomto vztahu ovlivňuje koho z hlediska poletování peříčka? Je implicitní vlastností peříčka pohybovat se ve větru? Je implicitní vlastností peříčka mít souřadnici ve 3D-geometrickém prostoru nebo dokonce padat vlivem gravitace?

Model peříčka by byl následující posloupností deklarací: peříčko je hmotný objekt (implicitně), jeho hmotnost je projevem gravitace v jeho prostředí (získáno), v 3D prostoru má své souřadnice (získáno), na jeho výškovou souřadnici má vliv gravitace (získáno), na všechny složky jeho souřadnice má vliv směr a rychlost větru (získáno), na polohu peříčka má vliv součet všech vlivů (odvozeno ze získaných).

Základní vlastnost entity byla stanovena na její pohyblivost v rámci hierarchie prostředí. Proto by měla být vazba mezi entitou a prostředím dynamická (nefixovaná v implementaci modelu). Entita vstoupí do prostředí a určitý mechanismus automaticky prováže její existenci s podobou prostředí. Modelovací základ a způsob provedení provázání bude rozebrán v následujících kapitolách o vlastnostech prostředí.

5.5 Vlastnosti

Vlastnosti jsou zobecněná pravidla aplikovatelná na objekty prostředí a entity. Rozlišujeme vlastnosti:

- Implicitní – množina vlastních atributů objektu *object*.
- Získané – entita nebo prostředí získá ve vzájemném vztahu novou odvozenou vlastnost[3].

Vlastnost musí být modelována jako strukturovaný objekt, protože její součástí jsou i určitá *data* (například souřadnice).

[2]Otázkou je způsob modelování vztahu mezi aktivní a pasivní entitou. V situaci, kdy pasivní entitou je mechanická překážka (třeba židle) a aktivní entitou je ve tmě tápající člověk, řešíme, jak modelovat jejich srážku. Pro tyto účely (a další) slouží *prostředí* jako komunikační platforma pro objekty. Tento příklad je dalším rozvinutím zavedeného problému "Čtenář a slunce".

[3]Případně lze rozlišit získané vlastnosti trvalé a dočasné.

```
3dcoord = property {
  local x,y,z;
}
```

Vstup entity do prostředí je *událost* – okamžik, kdy entita získá soubor získaných vlastností. Podobně je vystoupení entity z prostředí událost, kdy se získané vlastnosti zruší. *Aktuální soubor všech vlastností však musí být vždy logickým důsledkem souboru implicitních vlastností a sloučení získaných vlastností ze všech zdrojů.*

> Problém je v simulačním stroji, který by stále udržoval platné vztahy mezi pravidly. Určitým řešením je využití metody AIN a mechanizmu vyhodnocování změn v síti objektů.

Příklad 6. *Mějme prostředí, ve kterém jsou entity lampa a detektor světla. S ohledem na znovupoužitelnost a reálnost modelu modelujeme systém takto:*

```
world = environment { // svět spojující lamp a detector
   entity lamp {}
   entity detector {}
}
lamp = entity {
   export light; // lampa generuje objekt light do nadprostředí
   light = TRUE
}
detector = entity {
   // test na přítomnost objektu light v prostředí
   detected(defined(light)) = TRUE
   detected(!defined(light)) = FALSE
}
```

V takovém případě slouží prostředí jako komunikační médium mezi entitami, které nebyly navrženy pro vzájemnou komunikaci. Po odstranění entity lamp z prostředí přestane být v prostředí definováno light a přehodnotí se status detected v detektoru světla.

5.6 Čas

V této kapitole bych rád uvedl některé myšlenky jazyka HELEF, zatím stále velmi obecně a bez nutnosti jeho znalosti.

Čas jako vlastnost prostředí

Uvažujme o existenci času a o jeho modelování. V běžné praxi se používá jedna časová osa – v diskrétní simulaci s kalendářem událostí je to dokonce relativně krátká posloupnost naplánovaných událostí.

V úvahách o reálném modelování reality se pokoušíme řešit existenci času. Máme tyto možnosti abstrahování času:

1. Čas je vlastnost každého jednotlivého hmotného elementu. Kde není hmota, tam ani nemá smysl řešit čas a čas tam není.

2. Čas je vlastnost entity. Entitou abstrahujeme jeden objekt reality (bez předpokladu dělení).

3. Čas je vlastnost *prostředí* – je to jedno z pravidel dostupných od prostředí.

Chtěl bych ukázat, že z hlediska vnímámí času v našem reálném světě existuje více časových os. Kromě té jedné, kterou tušíme jako přírodní-fyzikální, je tu ještě rozdělení do geografických časových pásem. Jednotnost časové osy má význam pouze pro případnou synchronizaci a komunikaci entit mezi sebou. Nakonec má však každý své hodinky – tedy časovou osu, která je obecně odlišná (nesynchronizovaná) s ostatními lidmi. Synchronizace se děje v rámci určité *tolerance*. Diference mezi různými osami času různých entit je

- v posunutí os času,
- v rychlosti ubíhání času.

Entita tedy (reálně) vnímá především svůj lokální čas. V případě modelu zahrnujícího například několik časových pásem bychom se běžně doupouštěli chyby ve vztahu modelového času k pravé realitě.

V případě *vnořených simulací* musíme každopádně implementovat vnořenost časové osy, která je vlastní součástí vnořené simulace. Inspirovat se můžeme například řešením standardních simulačních knihoven jazyka SIMULA 67. V případě knihovny SIMLIB/C++ je modelový čas neinstancovatelná globální součást modelu.

Příkladem modelu s různou rychlostí ubíhání času může být model zahrnující relativitu času.

5.7 Vazba na metodu AIN

Je několik způsobů, jak modelovat pomocí AIN odvozené vztahy mezi částmi modelu (prostředími a entitami). Základem je deklarování funkce vracející množinu nových získaných pravidel odvozených z *aktuální kombinace* prostředí a obsažených entit. Tím, že je tato kombinace považovaná za obecně dynamickou (proměnlivou), pak je typická pro styl vyhodnocování v AIN (podle změn). AIN pak nevyhodnocuje pouze změny hodnot objektů ve statické síti, ale vyhodnocuje navíc aktuální strukturu sítě (množinu objektů a vazeb – $[OS, R]$). Model je schopen přetvářet sám sebe (reflektivita).

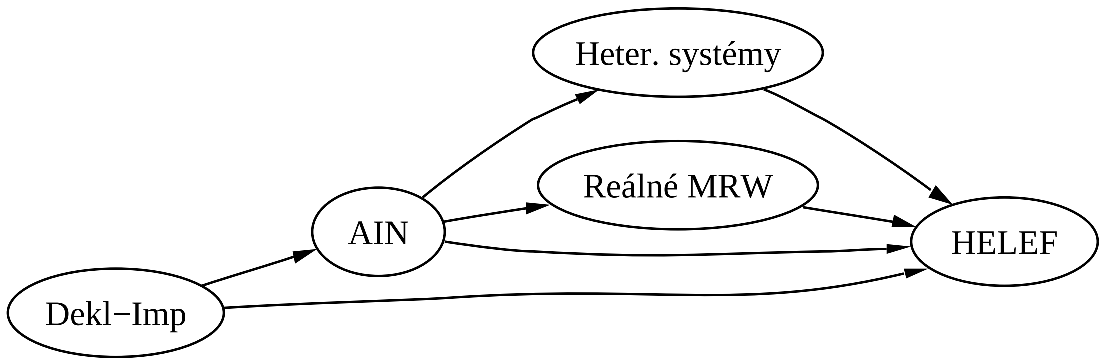
5.7.1 Příklad 3D-prostor

Mějme prostředí 3Dspace s pravidlem “hmotné entity mají souřadnice x,y,z”. Mějme míček *ball* s pravidlem $physical = TRUE$. Implicitní pravidlo prostředí zapíšeme:

```
3dcoord = property { local x,y,z; }
objects(physical=TRUE) = property @3dcoord
```

Prostředí bude případně obsahovat i patřičné pohybové rovnice. Vyhodnocení této deklarace rozšíří $ball \in objects(3Dspace)$ o kopii objektu $3dcoord$, respektive všechny objekty obj, $obj \in objects(3Dspace) : physical = TRUE$.

Je to stále balancování mezi styly: třídní OO $\times$ beztřídní OO, statický (budovaný během překladu) $\times$ dynamický (interaktivní), procedurální $\times$ deklarativní. Zopakujme si, že naším cílem je maximální obecnost modelu a z toho čerpaná znovupoužitelnost (přenositelnost).



Obrázek 5.1: Schéma vývoje práce

Míček v 3D prostoru je jen příkladem pro demonstrování těchto pojmů. Dynamičnost struktury modelu je pochopitelně v klasických překládaných jazycích emulovatelná – můžeme si představit každý objekt statického programu jako síť AIN, která přenáší strukturu modelu do podoby dynamických dat. Taková implementace by však byla příliš náročná, protože jazyky typu C++ ani nebyly konstruovány pro interaktivní běh aplikací.

Částečnou možnost nabízí jazyky Smalltalk a Self – například ve Squeak Smalltalku právě probíhá projekt vytváření další úrovně nad konceptem tříd – tak zvaných *classboxes*[4]. Classbox plní funkci prostředí aplikace, které aplikaci dodává sadu modifikací globálních tříd. Chce-li aplikace v globálním prostoru tříd nějakou třídu modifikovat, musí podle třídního objektového paradigma vytvořit novou, odvozenou třídu, a to s sebou nese i vytvoření nového identifikátoru třídy (jména). Classbox je naopak prostředím omezujícím platnost modifikace globální třídy na aplikaci a modifikace neovlivní nadprostředí classboxu.

5.8 Dotazování na vlastnosti modelu

Mechanismus automatického sestavování simulačního modelu z deklarací může být z jistého hlediska považován za určitý typ experimentování s modelem. Sestavený simulační model nese totiž jistou *odvozenou informaci o modelovaném systému.* Konkrétně to je například množina získaných vlastností jednotlivých prostředí a entit.

Nabízí se na sestavený simulační model aplikovat dotazy, jako třeba na databázi faktů v Prologu. To znamená, že předložíme simulátoru simulační modely a sledujeme způsob jejich automatického provázání.

5.9 Shrnutí kapitoly

Obrázek 5.1 ukazuje vývoj myšlenek této práce. Z původního záměru kombinovat deklarativní a imperativní programování vznikla metoda AIN provádějící obecný deklarativně-procedurální výpočet. Metoda AIN současně položila základ obecného propojování systémů do heterogenních systémů.

Druhou linií práce bylo reálné modelování reality (RMRW) – snaha o popis struktury a chování světa. Strukturou světa míníme však i strukturu heterogenního modelu. Metoda prostředí-entita-vlastnost je v podstatě rozšířením organizace programů a modelů. V teorii

[4]Jedná se o velmi čerstvou novinku, o které se zatím jen uvažuje v jedné větvi vývoje systému Squeak.

programování nacházíme podobné dohody – například vytváření modulů (jazyk Modula) a jejich využívání, objektovou orientaci a odvozování dědiců tříd. Organizací heterogenních modelů se rozumí zejména rozdělení prostorů jmen a deklarování vztahů mezi částmi modelu. Jazyk HELEF, který je důsledkem celého vývoje této práce, pak vystupuje jako řídící element celé heterogenní simulace. Popis metody RMRW je v této kapitole poměrně obecný a jeho plná implementace v jazyce HELEF je předmětem budoucího vývoje jazyka.

Kapitola 6

Jazyk HELEF

*V této kapitole bude představen heterogenní simulační jazyk HELEF (**HE**terogeneous **L**anguage **E**nvironment, **F**inal version), který vznikl během mé práce. Je založen na metodách popsaných v kapitolách 4 a 5. Jazyk HELEF byl velmi inspirován jazyky Self, Smalltalk a Beta, což se projevuje na jeho syntaxi, sémantice a přístupu k vytváření modelů a vedení simulačních experimentů. Vzhledem k tomu, že jeho sémantika byla již z velké části popsána v předchozím textu, bude jeho specifikace velmi zkrácená. Cílem této kapitoly a celé práce není podat čtenáři referenční manuál o jazyce HELEF, ale zdůraznit jeho filozofii a význam.*

Jazyk HELEF [Hrubý MOS02, Hrubý MOS03, Hrubý ASU02] vznikl několikaletým vývojem s cílem inteligentně a funkčně propojit koncept deklarativního a imperativního programování. Při vývoji tohoto jazyka byly odvozeny dvě nové metodologie modelování:

- Obecné heterogenní modelování (metoda AIN).
- Reálné modelování objektů reálného světa (RWRM).

Metoda AIN definuje koncept deklaracemi řízeného programování pro využití v modelování a metoda přirozeného modelování světa (RWRM) koncept AIN rozšiřuje o metodologii modelování struktury a chování reálného světa. Jazyk HELEF byl navíc využívá interaktivnost, perzistenci a reflektivitu.

Souhrnně jazyk HELEF kombinuje několik konceptů programování a simulačního modelování:

- Deklarativní a imperativní programování.
- Heterogenní modelování (multi-paradigm programování) založené na metodě AIN.
- Beztřídní objektovou orientaci.
- Interaktivní budování modelů.
- Perzistenci modelů.
- Distribuovanost.
- Vnořené simulace, nedeterministický výpočet a řešení úloh.

Z hlediska teorie heterogenního modelování tvoří HELEF slučující jazyk pro budování heterogenních modelů a řízení heterogenní simulace. Všechny pojmy z kapitol 4 a 5 jsou platné i zde.

6.1 Definice datových simulačních objektů

Vycházejme ze simulačního modelu v podobě textu v jazyce HELEF (analogie na Self a poznámku v kapitole 2.8.1). Výsledkem překladu programu v jazyce HELEF je síť simulačních objektů propojených podle specifikace metody AIN. Metoda AIN byla ve čtvrté kapitole definována obecně – bez rozlišení datových simulačních objektů. Pro účely praktického programování však potřebujeme několik různých typů simulačních objektů, které se liší strukturou atributů a sémantikou funkce *reeval* (kapitola 4.2.2). Jsou to datové simulační objekty typu *výraz*, *informace* a *jmeno*, jejichž definice následuje.

6.1.1 Simulační objekt typu výraz (HFExpr)

Matematický výraz so_d je nejjednodušší simulační objekt se základní funkcí *reeval* (kapitola 4.2.2). Zopakujme hlavní pravidla pro simulační objekt typu výraz:

- Sémantika výrazu je popsána v kapitole 4.
- Pokud výraz so_d není v $[OS, R]$ vyčíslitelný, pak $value(so_d) = \epsilon$.
- Výraz vytváří specifickou podsíť $[\{so_d\}, \emptyset]$.
- Objekt výraz je vnitřním atributem složitějšího objektu (informace, proces). Proto v $[OS, R]$ vždy platí: $\exists o \in OS : so_d \to o$.
- Předpis pro získání hodnoty výrazu je obsažen v jeho atributu $contents(so_d)$.

Příklad 7. *Příklady výrazů:* 1, $a + b$,*time* $>= 10$.

Nebudeme zde uvádět syntaxi matematických výrazů, jelikož je obecně známá z běžných programovacích jazyků. Úplná specifikace gramatiky jazyka HELEF je v příloze textu práce.

6.1.2 Simulační objekt typu informace (HFInfo)

Informace je stavebním prvkem modelů v HELEF. Informace je *pojmenovaná* znalost – nebo také pojmenovaný předpis k odvození hodnoty z jiných informací. Informace tedy rozšiřuje strukturu atributů simulačního objektu o atribut *name* – jméno objektu. Jménem se rozumí jednoduchý identifikátor (specifikováno v gramatice jazyka).

Syntaxe informací v HELEF je následující:

```
name[”(“condition”)”] (“=” | “:=”) object
```

Simulační objekt typu informace obsahuje atributy:

- *Name* – jméno informace (identifikátor).
- *Condition* – explicitní podmínka přehodnocení informace (implicitní podmínkou je změna báze). Tento atribut je nepovinný a ovlivňuje strukturu specifické podsítě objektu informace.
- *Relation* – operátor vztahu mezi objektem hodnoty a informací. Nabývá hodnoty výčtového typu $\{linked, assigned\}$. Vztah mezi objektem *object* a *name* je *linked* (resp. *assigned*), pokud byl uveden relační operátor = (resp. :=).

- *Object* – předpis pro odvození hodnoty z jiných informací (například reference na objekt typu výraz).

V tom obyčejném případě, kdy není specifikována podmínka $so_i(condition)$, se objekt informace so_i chová jako pojmenovaná hodnota – odkazuje na objekt $so_i(object)$. Změna $value(object)$ se pouze přenese na so_i. Objekt so_i vytváří specifickou podsíť

$$[\{so_i, so_i(object)\},\ \{(so_i(object), so_i)\}].$$

Druhý případ, kdy je $so_i(condition) \neq \epsilon$, vytváří specifickou podsíť

$$[\{so_i, so_i(condition), so_i(object)\},\ \{(so_i(condition), so_i)\}].$$

Sémantiku explicitní a implicitní podmínky přehodnocení informace ukazuje následující algoritmus metody $reeval$ pro objekty typu informace:

1. Je-li $condition \neq \epsilon$ a $value(condition) = TRUE$, pak $return(reevalWith(object.doEval()))$.
2. Je-li $condition = \epsilon$, pak $return(reevalWith(value(object)))$.
3. Jinak $return(FALSE)$.

Atribut *relation* určuje vztah mezi pravou a levou stranou informace. Operátor = deklaruje odvozovací vazbu (tok změn) a operátor := deklaruje přiřazení hodnoty (počáteční hodnotu informace).

Příklad 8. *Příklady informací* $a = 1$, $b(a = 1) = 2 + c$, $world = environment\,\{\}$

Obor hodnot informace zahrnuje primitivní datové typy a simulační objekty všech typů.

6.1.3 Simulační objekt typu jméno (HFName)

Poslední objekt z této řady – objekt HFName – slučuje několik objektů informace stejného jména a umožňuje tak strukturované deklarování hodnoty informace pro více explicitních podmínek.

Jméno (HFName) je objekt nadřazený objektu informace. Jeho atributem je množina $rinfos = \{i_1, i_2, ...\}$ registrovaných informací stejného jména. V systému $AIN = [OS, R]$ tedy pro jméno $n \in OS$ platí:

$$\forall i_1, i_2 \in rinfos(n) : name(i_1) = name(i_2)$$

Navíc musí platit, že nejvýše jedna registrovaná informace i jména n má atribut $relation(i) = assigned$

$$not \exists i_1, i_2 \in rinfos(n), i_1 \neq i_2 : relation(i_1) = relation(i_2) = assign$$

Objekt typu jméno je automaticky generován pro každou informaci v programu. Dále pro sadu informací daných:

```
name(condition1) = expr1
name(condition2) = expr2
...
name(conditionN) = exprN
name := expr
```

V síti $S = [OS, R]$ pro objekt typu jméno so_{name} platí, že $\forall info \in rinfos(so_{name})$: $info \rightarrow so_{name}$. Objekt so_{name} nabývá hodnoty poslední změny v množině $rinfos(so_{name})$. To znamená, že při evoluci $((OS, R), \emptyset, \{..., RI, ...\}) \vdash ((OS, R), \{..., so_{name}, ...\}, \emptyset)$, kde $RI \subseteq rinfos(so_{name})$:

1. $|RI| = 1$, pak *doeval* objektu so_{name} vrací $value(ri)$, kde $RI = \{ri\}$.
2. $|RI| > 1$, pak *doeval* vrací ϵ (neurčitá hodnota).

Příklad 9. *Příklad sloučení informací*

```
q:=0
delta(q=0 & input='a')=1
delta(q=1 & input='b')=1
delta(q=1 & input='c')=2
q(AINnoevent)=delta
```

6.2 Vrstva AIN

V této fázi specifikace stále ještě vycházíme z existující vytvořené sítě $AIN = [OS, R]$, kde OS je množina všech simulačních objektů specifikovaných v modelu:

- $HFExpr$ – všechny matematické výrazy na pravých stranách informací a všechny explicitní podmínky informace (condition).
- $HFInfo$ – všechny informace a $StartC$ podmínka procesů.
- $HFName$ – všechna jména v modelu (generuje se automaticky z objektů typu informace).
- $HFProc$ – procesy v modelu.
- $HFStruct$ – objekty typu struktura tvořící *prostory jmen.*

Simulační objekty jsou provázány podle specifických podsítí, které vytvářejí. To znamená, že všechny $HFExpr \rightarrow HFInfo$, $HFInfo \rightarrow HFName$ nebo $HFInfo \rightarrow HFProc$, $HFName \rightarrow HFStruct$. Vrstva AIN zabezpečuje evoluci výpočtu podle definice z kapitoly 4. Dynamické zásahy do AIN a vytváření AIN (podle prostorů jmen) ze simulačního modelu řeší nadřazené vrstvy.

Příklad 10. *Mějme následující kód:*

```
time(AINnoevent)=time+1
p=process(time==3) { }
```

který vygeneruje síť $S = [OS, R]$, *kde*

$$OS = \{time_{info}, time_{name}, AINnoevent, cond(time_{info}), expr(time_{info}),\\ p, startCond(p), expr(startCond(p))\}$$

s $cond(time_{info}) = AINnoevent$ *a* $expr(time_{info}) = time_{name}+1$. *Vazby R jsou zobrazeny na obrázku 6.1.*

Obrázek 6.1: Vazby R z příkladu 10

6.3 Procesy

Procesy v jazyce HELEF tvoří sekvenční složku výpočtu. Řízení jejich činnosti bylo popsáno v kapitole 4. V této kapitole doplníme několik podrobností o jejich příkazech. Syntaxe procesu je následující:

```
process [”(“ StartCondition”)” ] “{”
   [ “local” l1 {”,”l2} “;” ]
   { Command }
“}”
```

Simulační objekt typu proces má atributy:

- $StartC$ – reference na objekt informace se $StartC$ podmínkou.
- $ContinueC$ – reference na objekt informace se $ContinueC$ podmínkou.
- $Locals$ – množina lokálních objektů. V systému $AIN = [OS, R]$ platí, že $Locals \cap OS = \emptyset$.
- $Body$ – seznam příkazů procesu.
- $BodyP$ – index aktuálního příkazu.

Funkce $reeval$ procesu byla popsána v kapitole 4.4.1. Běh procesu (spuštěný voláním $runUntilChanged$) se řídí algoritmem pro chování stroje AIN ve stavu q_P.

6.3.1 Příkazy procesu

Procesní část jazyka HELEF je stavěna na obvyklém imperativním jazyce, jako je třeba jazyk C. Proto by v této kapitole nebylo přínosné formálně definovat syntaxi a sémantiku každého příkazu procesu. Jazyk HELEF obsahuje klasické příkazy přiřazení, větvení a cyklu. Pro popis běhu programu jsou podstatná tato pravidla:

- Proces pracuje nad lokálním ($Locals(p)$) a globálním ($[OS, R]$) prostorem dat.
- Proces běží *nepřerušeně* až do okamžiku, kdy modifikuje objekty globálních dat nebo se korektně ukončí (operace $runUntilChanged$).
- V síti $[OS, R]$ pro $Locals(p)$ procesu p platí, že $\forall l \in Locals(p) : |\bullet l| = |l \bullet| = 0$. K vyhodnocení výrazu $expr$ v těle procesu jsou použity hodnoty $\{value(o) | o \in Base(expr)\}$ z aktuálního AIN systému $\Sigma = (OS, R)$.

Jazyk HELEF nezavádí obvyklé zasílání zpráv. Synchronizace mezi procesy je to řešena kombinací podmínek $StartC$ a $ContinueC$. Kontext běhu procesu ($BodyP, Locals$) poprvé spuštěného procesu p je $(1, Locals(p))$, kde $\forall i \in Locals(p) : value(i) = \epsilon$. Je třeba zopakovat, že proces je simulační *objekt* a jeho přítomnost v AIN je trvalá – při dalším jeho spuštění tedy $Locals(p)$ obsahují hodnoty (ale $Body(p)$ je opět nastaven na 1).

Pro vyhodnocení příkazu se zavádí návratová informace zvaná *referát změn* (popsaná v kapitole 4). Referát změn obsahuje výsledek zápisu do globálních dat – identifikaci objektu a jeho novou hodnotu[1]. Proto je *operace přiřazení* prakticky jediná operace, která může běh procesu přerušit.

```
obj := expr // operace přiřazení
```

V procesu p sítě $[OS, R]$ nechť je `obj`$\in OS \setminus \{p\} \cup locals$. Při operaci přiřazení $(n, locals) \vdash_P (n+1, locals)$ a $Body(p, n)$ je příkaz přiřazení do objektu $obj(Body(p, n)) \in OS$, pak dojde ke změně stavu systému podle (4.24) na (4.26) definované v kapitole 4.

Příklad 11. *Příklad procesu:*

```
n:=10
fac=process(n>0) {
  local b;
  b:=1
  while (n>=1) {
    b:=b*n
    n:=n-1
  }
  facres:=b
}
```

Dynamičnost modelu

Proces může přidat nebo odebrat objekt ze své nadřazené struktury.

```
++object [-> place]
```

Příkaz umístí objekt *object* do struktury *place* nebo implicitně do struktury, ve které se nachází proces.

```
--name [-> place]
```

Obdobně – ze struktury se odstraní hierarchie objektů *name* začínající jménem *name*. Provedení příkazu odebrání a přidání tvoří akci přesunu objektu mezi strukturami – mobilita objektů. V síti $[OS, R]$ platí, že rušený objekt *name* způsobí změnu vyčíslitelnosti objektů $name\bullet$.

6.4 Úroveň struktur

Struktury tvoří seskupené celky simulačních objektů. Jejich seskupení není v žádném případě *zapouzdření* – to v jazyce není definováno (stejně jako v Betě a Selfu). Hlavní význam struktur je ohraničovat a seskupovat simulační objekty, definovat strukturu modelu a *hranice viditelnosti* (platnosti) množin informací (prostory jmen).

[1]Do referátu změn patří i objekty *ContinueC* a *StartC*

```
structureType ["@" Id ] "{"
   [ local Id{,Id} ";"]
   [ export Id{,Id} ";"]
   { Declaration }
"}"
```

Jazyk HELEF definuje tři základní typy struktur:

- *Property* – modeluje určitou zobecnitelnou vlastnost.
- *Entity* – modeluje pasivní nebo aktivní přemístitelný prvek simulace.
- *Environment* – modeluje konkrétní nebo abstraktní prostředí pro entity.

Objekt S typu struktura má atributy:

- $TypeS$ – výčtový typ, $typeS \in \{environment, entity, property\}$.
- $Inherites$ – jméno nadřazeného objektu (identifikátor).
- $InheritesRef$ – reference na nadřazený objekt.
- $Locals$ – množina identifikátorů lokálních jmen (má význam při budování *prostoru jmen*).
- $Exports$ – množina identifikátorů exportovaných jmen (má význam při budování *prostoru jmen*).
- $Objects$ – množina obsažených pojmenovaných a nepojmenovaných objektů.

V sítí $AIN = [OS, R]$ pro strukturu S platí: $\forall o \in objects(S) : (o, S) \in R$.

6.4.1 Prostor jmen

Struktura S obsahuje objekty $objects(S)$. Tyto objekty $o \in objects(S)$ mají svá jména $name(o)$ a odkazují se na jiná jména $\{name(b)|b \in Base(o)\}$. Prostor jmen je abstraktní ohraničení objektů, které může překročit rámec jedné struktury, a modeluje *významové ohraničení viditelnosti* určitých faktů ve struktuře modelu (případně v realitě). V jazyce HELEF vytváří prostory jmen struktury a procesy.

Prostory jmen mají význam při vyhledávání jmen pro navázání vazby s jinými objekty. Vzhledem k tomu, že modely v HELEF mají dynamickou strukturu, nelze zcela vždy vyhledávat ve jménech staticky – na úrovni zdrojového textu modelu v době překladu. Model v HELEF se může kdykoliv za běhu jeho simulace rekonfigurovat podle aktuálního stavu prostorů jmen – speciálně to platí v případě dynamické objektové dědičnosti.

Definice 22. *V síti $[OS, R]$ je* prostor jmen *objektu S (struktury nebo procesu – strukturované objekty) množina dvojic $NS(S) \subseteq Name \times OS$.*

Notace: pro jméno $name$ píšeme $name \in NS$ a je tím míněno $\exists nsi \in NS : name(nsi) = name$

Implicitně platí, že se prostor jmen dědí do vnořených struktur. Máme-li prostor jmen $NS(env)$ prostředí env, pak všechna podprostředí $subenvs \subseteq objects(env)$ mají implicitně $NS(env)$ jako podmnožinu.

Definujme výpočet aktuálního prostoru jmen pro strukturu S (procesy doplníme později). Zaveďme množiny:

- $Gener(S) = \{o \in objects(S) | name(o) \neq \epsilon\}$.
- $SubStructures(S) = \{o \in objects(S) | o$ je typu struktura$\}$.

Podle specifikace HELEF může struktura deklarovat množiny *local* a *export* jmen. Prostor jmen struktury S_{Parent} nadřazené S ($S \in objects(S_{Parent})$) označme $NS(S_{Parent})$. $NS(S_{Parent}) = \emptyset$ pro vrchol *model* hierarchie objektů.

Zavedeme význačné podmnožiny jmen a jejich implicitní hodnoty:

- $ChGener(S) = gener(S)$.
- $ChLocal(S) = Local(S)$.
- $ChExport(S) = Export(S)$.
- $ChInherited(S) = \begin{cases} chVisible(inheritesRef) & ; inheritesRef \neq \epsilon \\ \emptyset & jinak \end{cases}$
- $ChVisible(S) = chVisible(S_{Parent}) \setminus chLocal \cup$
  - $\cup chGener(S) \cup chInherited(S) \cup$
  - $\cup \{\bigcup_{o \in subStructures(S)} chExport(o)\}$.

Jednotlivé typy struktur pracují s význačnými podmnožinami stejně, až na struktury *property*, které implicitně kladou $chExport = chGener \setminus chLocal$. Objekt struktury má podle specifikace $chVisible$ čtyři zdroje jmen: svou strukturu, vnořené struktury, nadřazenou strukturu v hierarchii a nadřazenou strukturu děděním. Charakteristika $chVisible(S)$ představuje sloučení všech jmen dostupných v dané struktuře. Pro sestavování dvojic *(name, object)* prostoru jmen $NS(S) = \{(name(obj), find(name(obj))) | obj \in chVisible(S)\}$ se hledá konkrétní objekt se jménem $name \in chVisible$ podle pořadí (algoritmus $find(name)$):

1. Prvek je obsažen ve struktuře $chGener(S)$,
2. ve vnořené struktuře $\bigcup_{o \in subStructures(S)} chExport(o)$,
3. v nadřazené struktuře v hierarchie $chVisible(S_{Parent})$,
4. ve struktuře nadřazené děděním $chInherited(S)$.

Příklad 12. *V následující ukázce se prvek b naváže na hodnotu* 2.

```
world = environment {
  a:=1
  sub1=environment {
    a:=3
  }
  sub2=environment sub1 {
```

```
        local a;
        property {
            a:=2
        }
        b = a
      }
    }
```

6.4.2 Dědičnost (delegace)

Vycházejme z kapitoly 2.4 o jazyce Self, kde je dědění atributů řešeno delegací. Objekt obsahuje atributy, které slouží pouze pro hledání atributů v nadřazených objektech. V jazyce HELEF je dědičnost podobná delegaci v jazyce Self. Struktura může odkazovat na jinou strukturu se jménem *inhName*.

```
structureType [@] inhName
```

Odkaz na nadřazenou strukturu má dvě podoby:

1. Delegace na objekt *inhName*.
2. Delegace na kopii (@) objektu *inhName* – vytvoří se kopie nadřazeného objektu *inhName* a na ten se deleguje. V porovnání s jazykem Self to zjednodušuje zápis.

```
integrator = entity {
}
myin1 = entity integrator {}
myin2 = entity @integrator {}
```

Objekt *myin1* odkazuje na *integrator* a v podstatě s ním sdílí atributy. Objekt *myin2* odkazuje na kopii objektu *integrator* – atributy integratoru si klonuje pro výlučné použití. Obě varianty delegace jsou popsány v kapitole 2.4, HELEF je pouze zavedením operátoru @ zpřístupňuje.

Sémantika delegace v HELEF je totožná s odpovídající sémantikou jazyka Self. V okamžiku zařazení objektu do kontextu je nalezen objekt se jménem *inhName*, případně zkopírován a navázán na objekt.

6.4.3 Odvozování objektů (struktur a procesů)

```
s1 = environment {
  local ls1,ls2,...;
  o1;o2;...
}
s2 = environment s1 {
  local lb1,lb2,...;
  b1;b2;..
}
```

Vycházejme ze struktury S_1 s prvky $objects(S_1) = \{o_1, o_2, ...\}$. Zápis *environment s1* vytváří *nový objekt* odkazující na S_1:

1. $inherites(S_2) = S_1$
2. $local(S_2) = local(S_1) \cup \{lb_1, lb_2, ...\}$
3. $type(S_2) = type(S_1)$ (typ struktury se nesmí měnit)
4. $objects(S_2) = doInherite(S_1, \{b_1, b_2, ...\})$

Funkce $resObjects = doInherite(source, newObjects)$ je definována:

1. $resObjects := objects(source)$
2. pro všechny $o \in newObjects$: (body 3 a 4)
3. je-li $name(o)$ neprázdné, pak
   (a) je-li $relation(o) = assigned$ pak nahraď případný ekvivalent v $resObjects$
   (b) je-li $relation(o) = linked$, pak
      i. je-li $condition(o) \neq \epsilon$, pak o přidej do $resObjects$
      ii. jinak nahraď případný ekvivalent v $resObjects$
4. je-li $name(o)$ prázdné, pak přidej o do $resObjects$

Příklady jsou v kapitole 7 a přílohách. Podobné je odvozování procesů (inspirace z jazyka Beta a Simula 67). Následující konstrukce definuje procesy *p1* a *p2*.

```
p1=process {
}
p2=process p1 {
}
```

Odvozování procesů využívá přístup *inner* se sémantikou definovanou v jazyce Simula 67. Proces *p2* se spustí v kontextu procesu *p1* s atributem *self* (reference na sebe) ukazujícím na *p2*. Tato část specifikace jazyka HELEF ještě není zcela dokončena.

6.5 Překlad a spuštění simulačního modelu v HELEF

Simulační model v jazyce HELEF můžeme zapsat textově v podobě tradičního zdrojového textu programu nebo budovat interaktivně podle vzoru jazyků Smalltalk a Self.

Systém HELEF je koncipován jako beztřídní OO jazyk podle vzoru jazyka Self. Překlad textu lze aplikovat na libovolný fragment textu – v implementaci jazyka HELEF je překladač *konstruktorem* objektů[2].

```
i := HFInfo fromText: 'a := 0'
```

[2] Jazyk HELEF je implementován ve Smalltalku. Proto budou některé implementační detaily ukazovány na tomto jazyce

Vznikne objekt i typu $HFInfo$ (a $HFExpr$) vytvářející podsíť:

$$[\{HFInfo_a, HFExpr_0\}, \{(HFExpr_0, HFInfo_a)\}].$$

Přeložením zdrojového textu objektu i se v systému Smalltalk (nebo Self) objeví objektová reprezentace deklarace $a := 0$, která může přetrvat perzistentně libovolně dlouho a objekt i pak může být vložen do konkrétního modelu. Takovým způsobem lze dynamicky modifikovat model. Dokonce je možné objekt z modelu vyjmout a exportovat zpět do textové formy (zdrojového textu programu – více v kapitole 2.8.1). Podobným způsobem může být řešena i mobilita objektů v distribuovaných systémech.

Podobně dostaneme strukturu s:

```
s := HFStruct fromText: 'environment { }'
```

Prvek i se vloží do struktury s, která se stane základem modelu (*model* typu $MHELEF$):

```
s insert: i.
model := s becomeModel.
```

Pro HELEF je typické, že textová reprezentace modelu se chápe jako *startovní kontext simulace*. Kterýkoliv stav simulace je stejně tak modelem jako startovní kontext simulace. Může být kdykoliv pozastaven, uložen, obnoven, exportován do textové podoby a kombinován.

6.5.1 Sestavení vazeb

Startovní kontext

Startovní kontext odpovídá CSC_{start} z kapitoly 4. V implementaci je to řešeno zasláním zprávy *makeStartingContext* objektu vrcholu modelu (objekt *model)* – vrací objekt s kontextem.

```
asys := model makeStartingContext.
asys step. "krok ⊢C"
```

Běh simulace

Běh simulace je definován evolucí CSC-stavu. $CSC_{start} \vdash_C CSC_{next}$. V implementaci se objektu s kontextem zašle zpráva *step*. Výsledkem je další kontext CSC_{next}. Simulace běží až do kontextu, který je považován za konečný (kapitola 4.4.2).

6.5.2 Konvence pro modely v HELEF

Z předchozího textu plyne, že každý objekt může být v jedné z fází svého životního cyklu:

1. Textová podoba v souboru na disku počítače.

2. Objekt v paměti (vzniká překladem textové podoby).

3. Objekt zařazen do kontextu struktury (vytvořeno propojení na další objekty).

4. Objekt je součástí simulace.

V množině objektů vystupují dva hlavní vrcholy hierarchie objektů:

- *World* – prostředí simulačního modelu (v něm se odehrává simulace).
- *Model* – systémový objekt zastřešující celý obsah objektové paměti.

Objekt *model* je v podstatě prostředí, jehož obsahem je implicitně $AINnoevent$ a simulační model $world$.

```
model = environment {
  // vlasní model a jeho world
  world = environment {
  }
}
```

6.6 Heterogenní a distribuovaná simulace

Principem jazyka HELEF je definovat síť provázaných informací vytvořených z programových struktur. Mluvíme-li o distribuovaných výpočtech nebo simulacích, musíme specifikovat dva hlavní důvody pro distribuci výpočtu:

1. Zapojením více výpočetních jednotek do výpočtu získáváme teoreticky silnější výpočetní aparát a tedy i efektivnější řešení problému
2. Spojením více různých výpočetních stanic slučujeme jejich informační zdroje a další užitečné prostředky (technická zařízení jako disky, mechaniky diskových médií, tiskárny a podobně)

Sloučením obecně různých výpočetních platforem však vzniká řada problémů. Speciálně v oblasti distribuované simulace musíme řešit problémy:

- Synchronizace procesů a jejich komunikace.
- Sjednocení datových typů a převody binárních reprezentací datových typů mezi různými platformami.
- Distribuce simulačního času.
- V případě simulace simulace řízené událostmi je nutná i určitá distribuovaná správa kalendáře událostí.

Hlavním problémem distribuované simulace je nejspíše distribuce modelového času. Přitom čas je pouze jednou z mnoha globálních (sdílených) informací. Podaří-li se smysluplně a transparentně vyřešit způsob distribuce jakékoliv informace, pak se podaří i efektivně řídit simulační čas. HELEF chápe čas jako jednu z informací, která je součástí globální informační sítě.

Představme si, že určitá část AIN je umístěná na jiném počítači a že interpret HELEF je schopen zabezpečit komunikaci mezi více počítači a zároveň abstrahovat rozložení AIN tak, že se jeví uživateli jako jeden celek. Pak není rozdíl mezi výpočtem AIN na jednom počítači nebo na více počítačích, sémantika AIN to podporuje. Existuje pak mnoho možností, jak toho využít při programování. Rozložení výpočtu je možné specifikovat buď v programu nebo až během simulace.

```
world = environment {
 property time;
 s1 = distributed environment env1;
 s2 = distributed environment env2;
 ...
}
```

Příklad ukazuje jednu z možností specifikace rozložení simulace. Prostředí *env1* a *env2* budou umístěna na separovaných výpočetních stanicích zapojených do simulace. Přesunutí části modelu na jiný uzel systému lze provést i během simulace.

Budeme-li pokračovat v analýze podstaty HELEF, což je sdílení a propojování informací, pak se nabízí chápat AIN jako databázi informací se specifickým algoritmem jejich propojování. Je možné připojit vzdálenou AIN jako uzel lokální AIN, tyto dvě AIN se automaticky propojí podle specifikace AIN a sdílejí navzájem informace. Směřuje to k populárním výpočetním architekturám založeným na komunikaci agentů. Je to také přirozený způsob propojování informačních systémů.

Kapitola 7

Modelování v jazyce HELEF

V této kapitole bych rád ukázal řešení typických simulačních problémů s využitím jazyka HELEF, který je vysoce univerzálním simulačním nástrojem. Na závěr kapitoly je uveden rozbor složitějšího příkladu, který dokumentuje zajímavé vlastnosti jazyka HELEF a jeho přínos do simulačního modelování.

7.1 Modelový čas

V jazyce HELEF je každý simulační model strukturou typu *environment* a podle konvence HELEF se nazýva *world.*

```
world = environment { // prostředí simulačního modelu
}
```

Modelový čas není implicitně definován, základní časová osa však může být specifikována jako vlastnost *basicTimeProperty*

```
basicTimeProperty = property {
    local timestep; // definice časové osy
    time := 0
    time(AINnoevent) = time + timestep
}
```

a specializovaná při vložení do modelu světa jako:

```
world = environment {
    property basicTimeProperty {
       timestep := 0.01 // vložení času do prostředí
    }
}
```

V modelu může existovat více časových os s různou definicí. Například posunutí času pro určitou část modelu zapíšeme:

```
world = environment {
   property basicTimeProperty {}
   Prague = environment {
```

```
        local time; // jiná časová osa
        time = parent.time + oneHour
    }
}
```

7.2 Diskrétní simulace

Událostmi řízená simulace

Událost je definovaná procesem se spouštěcí časovou podmínkou $StartC$. Mějme sadu bloků $\{eventA, eventB, ...\}$ pro obsluhu událostí ve formě procesů.

```
eventA = process { // proces nazvaný eventA představující
                   // obsluhu události eventA
}
```

a eventA naplánujeme na čas $time = 0$.

```
world = environment { // v rámci modelu zavedeme proces
  // eventA a specifikujeme jeho objekt StartC nastavením
  // time = 0
  process eventA { // odvození nového objektu rozšířeného
                   // o startovní podmínku
     StartC = (time=0);
  }
}
```

Blok události *eventA* se spustí s platností *StartC* (simulační stroj AIN blok přesune do fronty připravených procesů). Provede svou činnost – naplánuje další události a buď se ukončí nebo přeplánuje nastavením *StartC* na další spuštění nebo *ContinueC* na pokračování. Naplánování další události *eventX* lze zapsat například takto:

```
eventX.StartC = (facility.Free=true); // spuštění události v okamžiku,
   // kdy se entita facility uvolní
eventX.StartC = (time>=at); // spuštění na časovou podmínku
```

Události – příklad *Generátor*

```
gener=entity { // entita generátor transakcí
   local timeadvance,action,at; // vymezení rozsahu platnosti jmen
   process(at) { // vzor akce generátoru
     do action; // na událost specifikovanou v at provede action a
        // přeplánuje se
     StartC=(time>=^time+timeadvance);
   }
}
world=environment { // prostředí simulačního modelu
   entity @gener { // klon generátoru s specifikovanými atributy
      at:=(time=0);
```

```
      action:=process { // proces vloží klon transakce do
           // prostředí modelu
         ++@transaction->world;
      }
      timeadvance=Exponential { m=10 };
   }
   transaction = process {
   }
}
```

Procesně orientovaná simulace

Vyjděme z reprezentace procesů v jazyce Simula 67 – tam jsou procesy řízeny dvojicí příkazů `activate` a `passivate`. V HELEF proces vzniká vytvořením svého objektu, spouští se na podmínku $StartC$ a běží podle podmínky $ContinueC$:

```
facility=entity {
   inuse:=nil
   last:=nil
   first:=nil
   free = (inuse = nil)
   seize = process { // self je zde reference na process
                     // entity transaction
       if (free=false) {
          if (last=nil) first:=last:=self
          else { last.next := self; last:=self }
       } else inuse := self
   }
   release = process {
       ...
   }
}
transaction=process {
   local next;
   next:=nil
   do process facility.size {
      // odvozuje proces facility.seize
      // a definuje hodnotu atributu self
   }
   // v osbluze
   do process facility.release {
   }
}
world=environment {
    process @transaction {
      StartC:=true
    }
}
```

7.3 Spojitá simulace

Model integrátoru

Síť AIN a způsob vyhodnocování AIN je přímým modelem diagramů spojitých modelů s integrátory. Jediným problémem je řešení rychlé smyčky, která se musí vyhodnocovat pouze jednou během časového kroku (synchronizuje se na změnu času).

Blok integrátoru naprogramujeme:

```
integrator = entity {
  local input;
  output(time.event) = output+timestep*input
  output := 0
}
```

Příklad použití integrátoru:

```
block = environment {
  i = entity @integrator {
    output := 1
    input = x + (-0.3*output)
  }
  x := 0
  x(time=1) = 1
}
```

Objekt `integrator` je deklarací entity pro numerickou integraci (pochopitelně je možno numerickou integraci naprogramovat lépe a přesněji). Výstup `output` se přehodnotí v každém kroku času (když dojde k jeho změně).

`Block` konkretizuje obecný objekt integrátoru vytvořením objektu `i`, kde se již specifikuje `input` a počáteční hodnota výstupu. Vstup `x` má nulovou počáteční hodnotu a v okamžiku $time = 1$ se změní na `1`.

Všimněme si, že AIN dokáže vytvořit platformu pro implementaci libovolného simulačního stroje.

7.3.1 Kombinovaná simulace

Navážeme na předchozí příklad – přidáním procesu se stavovou podmínkou $StartC$.

```
process(i.output>=0.8) {
  x := 0;
}
```

Přidáním procesu do prostředí `block` definujeme stavovou podmínku pro diskrétní událost a dostáváme tak *kombinovanou* simulaci.

7.4 Distribuované zpracování

Relace mezi dvěma simulačními objekty lze abstrahovat na relaci mezi objekty na různých počítačích. Pak se v návrhu algoritmů nic nemění a simulační model je přirozeně distribuován.

```
world = environment {
   property basicTimeProperty {}
   part1 = distributed environment part {
   }
   ...
}
```

Model je přeložen na hlavním uzlu distribuovaného simulačního zařízení a jeho části jsou rozděleny do zapojených uzlů, kde každý z nich může tvořit lokální síť AIN.

7.5 Rozšířené prohledávání – nedeterministický výpočet

V určitém ohledu můžeme simulaci chápat jako hledání kombinace určitých atributů modelu takových, aby platila specifikovaná podmínka. Zavedeme do jazyka HELEF prvky řešení úloh jako v umělé inteligenci.

Search-experiment

Search-experiment je velmi silná vlastnost jazyka HELEF. Umožňuje nedeterministické výpočty (prohledávání stavového prostoru) a vnořené simulace. Příkaz *search* má podobu:

```
search <range> <stopCondition> <resultCond>
```

Search prochází polem hodnot $range$ a pro každou hodnotu $val \in range$ provede:

1. Uloží kompletní stav simulace.
2. Vygeneruje objekty *stopCondition* a *resultCondition*.
3. Návratová hodnota příkazu je val, evoluce simulace dál pokračuje až do okamžiku platnosti *stopCondition*.
4. Je-li *resultCondition* platný objekt s hodnotou *true*, pak se *search* ukončí – obnoví kontext simulace a vrací hodnotu val.
5. Jinak se obnoví kontext simulace a pokračuje search s další hodnotou.

Stejného výsledku dosáhneme procesem:

```
process {
   local tmp;
   tmp := @model
   forall i in range {
       result := i
       continue := stopCondition
       if (resultCondition) {
           model := tmp
           result := i
           exit
       }
   }
}
```

Simulační experiment lze libovolně modifikovat – například, aby se po dosažení platného výsledku pokračovalo v evoluci bez návratu k původnímu kontextu. Tímto způsobem můžeme programovat i simulátory schopné se *vracet v čase*.

Rozšířené simulační experimenty, optimalizace

Mějme situaci: řešíme systém hromadné obsluhy obvyklým způsobem (například s použitím knihovny SIMLIB nebo v jazyce Simula). Ze simulace obdržíme statistiky využití modelovaných zařízení a sledujeme jednu konkrétní statistiku s. Potřebujeme ladit parametr p tak, aby platila podmínka $cond(s)$. Například kapacitu skladu c_s, kde průměrná doba čekání ve frontě byla menší než T_w. Kromě obvyklé metody opakovaných simulací můžeme problém simulace formulovat jako: “nalezni nejmenší kapacitu skladu c_s, aby platilo $cond(s)$ po provedení n přístupů na zařízení”.

```
Cs = search [1..] (acc==n) (cond==true)
```

7.6 Rozbor příkladu

Zadání: Mějme čipovou kartu pro výběr peněz v bankomatu. Její funkci zjednodušme na zkontrolování zadaného čísla *pin* a vydání důvěrné informace - tajného kódu karty. Jednou z technik zjišťování tajného čísla *pin* je útok hrubou silou, spojený se sledováním odběru elektrického proudu kartou. Algoritmus karty kontrolující zadaný pin nechť je následující:

```
cycles := flashread(cycles); // načti do registru čítač pokusů
if (cycles==0) return ;
pin := flashread(pin); // načti pin do registru
while (cycles>0) { // dokud je počet pokusů nenulový
  if (inputpin==pin) break; // správný pin
  else { // chybný pin
     cycles := cycles - 1;
     flashwrite(cycles); // dekrement čítače v paměti, sledované místo
     return ;
  }
}
secret := flashread(secret); // odeslání tajné informace do
   // vnějšího prostředí
cycles := 3;
flashwrite(cycles); // znovunastavení čítače pokusů
```

Instrukce zápisu do paměti flash odebírá nejvíce proudu a lze ji takto vysledovat z vnějšího nepřátelského prostředí. V okamžiku charakteristického odběru útočník automaticky odpojí napájení karty a operace se nedokončí. Tak má útočník neomezeně mnoho pokusů zjistit kód pin karty. Dnešní karty mají již modifikovaný algoritmus autentizace, kde se počítadlo pokusů dekrementuje ještě před porovnáním kódů pin.

Bezpečné prostředí bankomatu:

```
cashm = environment {
   sup := true // napájení zapnuto
   clk := 0
```

```
    clk(AINnoevent) = clk + 1 // taktovací hodiny
}
```

Prostředí útočníka:

```
hack = environment @cashm {
   sup(power>level) = false // vypni, pokud odběr proudu v prostředí
          // přesáhne úroveň level
   process(defined(card)) { // po vložení karty
      for inputpin := (0, 9999) { // útok hrubou silou
          // proces se pozastaví do okamžiku, kdy karta odpoví tajný
          // kód nebo se automaticky odpojí napájení
          sup := true // sepni napájení
          continue = ((sup=false) || defined(secret))
      }
   }
   // entity card {}
}
```

Akce:

Hlavním cílem této ukázky modelu v jazyce HELEF je ukázat způsob implementace modelu v duchu kapitoly 5 o reálném modelování. Jádrem modelu je rozlišení pevné struktury modelu a části modelu dočasně závislé na podmínce napájení. V okamžiku vypnutí proudu totiž některé objekty modelu přestanou existovat, stejně jako když se paměti RAM vypne napájení.

Entita $onoff$ je základem objektu závislého na informaci sup. Objekt $onoff$ představuje naplánovanou událost, která se sama zruší v okamžiku $sup = false$:

```
onoff=entity {
  local at, todo;
  process(at) { // v okamžiku at
    do todo; // proveď akci todo
  }
  process(sup=false) { // napájení odpojeno. Odstraň celý
         // objekt z modelu
     remove parentStructure;
  }
}
```

Jednotka Flash paměti:

```
flashitem = entity { // buňka flash paměti
   process(sup=true) { // spuštění nastaví hodnoty rozhraní
      output := nil
      power := 0
      ready := nil
   }
   process(sup=false) {
      output := nil
```

```
         ready := nil
      }
      read = process { // po aktivování procesu
         power := 1 // spustí odběr proudu
         ++ entity @onoff { // naplánuje událost
            at = (clk= ^clk+1) // další krok času
            todo = process {
                power := 0
                output := value
                ready := true
            }
         }
      }
      // write process obdobně, s power := 10
   }
```

Karta:

```
   card = entity {
      // objekt pin paměti flash
      pin = entity @flashitem { value := 1234 }
      // objekty cycles a secret
      cycles = entity @flashitem { value := 3 }
      secret = entity @flashitem { value := 1234567890 }
      // exportuj objekt power do nadřazené struktury
      export power;
      // odběr je součtem odběrů všech částí
      power = pin.power + cycles.power + secret.power + cpupower;
      cpupower := 0
      process(sup=true) {
         // algoritmus autentizace
      }
   }
```

Model karty ve čtecím zařízení útočníka vznikne vložením objektu *card* do prostředí *hack*. V objektu karty začne existovat informace *sup* s hodnotou *true*, spustí se proces autentizace a proces útoků na kartu. S každým neúspěšným pokusem o pin se hodnota *sup* převrátí na *false* a veškeré části karty, závislé na proudu, přestanou mít definovanou hodnotu. Model karty ukazuje rozdíl mezi prvky tvořícími neměnnou strukturu objektu a prvky dočasnými. Příklad je rozšířením příkladu "Čtenář a slunce", na kterém byla ukázána potřeba nepřímé komunikace mezi entitami v modelu systému – model čtecího zařízení neovlivňuje stav hodnot vnitřních atributů karty.

Kapitola 8

Závěr

Ve své disertační práci jsem se pokusil o analyticko-syntetický přístup, v němž po rozboru vlastností několika programovacích a simulačních OO jazyků následuje návrh metodické a programové podpory pro vytváření modelů heterogenních systémů. Popis několika OO jazyků, multi-paradigma jazyků a jejich vlastností tvoří popis současného stavu výzkumu v metodách modelování, ze kterého jsem ve své práci vycházel. Z řady OO jazyků mě ovlivnil zejména jazyk Smalltalk pro svou interaktivnost, perzistenci, reflektivitu a čistou objektovou orientaci. Interaktivnost a perzistence významným způsobem ulehčuje návrh simulačních modelů s dynamicky modifikovatelnou strukturou.

Heterogenní modelování je zřejmě budoucím trendem ve vývoji metod modelování systémů. Navíc se domnívám, že nalezne uplatnění i v programování rozsáhlejších programových produktů, jako jsou operační systémy, informační systémy, systémy umělé inteligence a další aplikace počítačů v lidském životě. Jedinou bariérou v praktické realizaci heterogenních modelů je způsob jejich efektivního propojení. Ve své práci jsem navrhl metodu dekompozice modelovaného systému na heterogenní systém zpracovatelný s využitím AIN. Navržená univerzální propojovací platforma AIN je hlavním původním přínosem této práce. V textu práce a přílohách byl uveden konkrétní návod k přizpůsobení různých autonomních systémů pro metodu AIN – například procesně a událostně orientovaných simulačních jazyků, spojitých simulátorů, konečných automatů, Petriho sítí, logických programovacích jazyků a dalších podsystémů zahrnujících vstupy a výstupy systému, statistiky a vizualizace.

Dalším výsledkem práce byla analýza struktury a chování objektů reálného světa. Motivací pro tuto analýzu byla snaha o maximální obecnost modelu vedoucí k jeho vysoké znovupoužitelnosti. Přínosem této části práce je návrh metody přirozeného modelování světa založené na jeho strukturování do hierarchie prostředí a pohyblivých entit, které vzájemnou interakcí získávají odvozené atributy. Vložením entity do prostředí získají obě strany, prostředí i entity, množinu dočasných nebo trvalých odvozených atributů. Myšlenkou metody bylo omezit model objektu světa na ty atributy (vlastní atributy), které obsahuje bez ohledu na své okolí. Významné je vytvoření teoretického zázemí pro realizaci interaktivního a perzistentního simulačního nástroje spojeného s rozšiřovatelnou knihovnou znovupoužitelných simulačních modelů. Takové prostředí by bylo velmi přínosné pro zefektivnění simulačního modelování systémů.

Teoretické přínosy metody AIN (kapitola 4) a metody přirozeného modelování reálného světa (kapitola 5) vedly k návrhu nového heterogenního simulačního jazyka HELEF. Vzhledem k možnému rozsahu práce byl jazyk HELEF popsán zkráceně s důrazem na provázanost kapitol 4 a 5 s cílem ukázat praktickou aplikaci těchto metod v modelování.

Specifikace jazyka je stále ve vývoji. Jedná se však již pouze o drobné úpravy. V době dokončování práce je jazyk HELEF implementován v prostředí jazyka Squeak Smalltalk.

Předpokládám další vývoj jazyka HELEF – dokončení jeho specifikace, vytvoření funkční implementace jádra interpretu jazyka a uživatelského prostředí a základu knihovny simulačních abstrakcí.

Literatura

[Benda 78] Benda, Z., Staudek, J.: Programování v jazyku Simula 67, SNTL, Praha, 1978

[Beta Mjolner] Beta Language Introduction, Mjolner Informatics Report, MIA 94-26, September 2001

[Beta HP] Beta HP: http://www.daimi.au.dk/~beta/

[BetaBook] Beta Book, ftp://ftp.mjolner.com/BETA/BOOK/betabook.pdf

[Blazevic 01] Blazevic, M., Budimac, Z.: An Environment-based Multiparadigm Language, In: Proceedings of the 1st Workshop on Multiparadigm Programming with Object Oriented Languages, Budapest, Hungary, 2001, ISBN 3-00-007968-8

[Bobillier 76] Bobillier, P. A., Kahan, B. C., Probst, A. R.: Simulation with GPSS and GPSS V, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976

[Buss 98] Buss, A., Jackson, L.: Distrubuted Simulation Modeling: A Comparison of HLA, CORBA, and RMI, In: Proceedings of 1998 Winter Simulation Conference, 1998

[Češka 98] Češka, M.: Petriho sítě, nakladatelství VUT Brno, Brno, 1998

[DEMOS] DEMOS Project HP: www.dcs.shef.ac.uk/~graham/research/demos.pdf

[DEVS] Conception, A. I., Zeigler, B. P.: DEVS Formalism: A Framework for Heirarchical Model Development, IEEE Trans. on Software Engineering, vol. 14 no. 2, únor 1988

[Drakos 92] Drakos, N.: Object Orientation and Visual Programming, In: Proceedings of OOPSLA'92 Workshop on Object-Oriented Languages: The Next Generation, rozšířený abstrakt, 1992

[Duchier TR] Duchier, D., Gardent, C.: A constraint-based treatment of descriptions, technická zpráva, University of the Saarland

[Erwig 95] Erwig, M., Meyer, B.: Heterogeneous Visual Languages - Integrating Visual and Textual Programming, In: Proceedings of 11th IEEE Symp. on Visual Languages, Darmstadt, 1995, pp. 318-325

[EASEL] Fisher, D. A.: Design and Implementation of EASEL - A Language for Simulating Highly Distributed Systems, technická zpráva, Carnegie Mellon University, Pittsburgh

[Fishwick TR] Fishwick, P., Zeigler, B. P.: A Multimodel Methodology for Qualitative Model Engineering, technická zpráva

[Gollu SHIFT] Gollu, A.: SHIFT: A New Programming Language for Simulation, technická zpráva, University of California, Berkeley

[Goldberg 84] Goldberg, A.: Smalltalk-80: The Interactive Programming Environment, Addison-Wesley series in Computer Science, 1984, Xerox, ISBN 0-21-11372-4

[Goldberg 89] Goldberg, A., Robson, D.: Smalltalk-80: The Language, Addison-Wesley series in Computer Science, 1989, ParcPlace Systems, ISBN 0-20113688-0

[Hrubý MOS01] Hrubý, M., Peringer, P., Rábová, Z.: Knowledge-Based Simulator in Object-Oriented Environment, In: Proceedings of the 35th Spring International Conference MOSIS 2001, Hradec nad Moravicí, CZ, MARQ, 2001, p. 75-80, ISBN 80-214-1860-5

[Hrubý MOS02] Hrubý, M., Rábová, Z.: HELEF: A New Simulation Language, In: Proceedings of the 36th Spring International Conference MOSIS 2002, Rožnov pod Radhoštěm, CZ, MARQ, 2002, p. 39-47, ISBN 80-85988-71-2

[Hrubý MOS03] Hrubý, M.: Formal Specification of the HELEF Simulation Language, In: Proceedings of 37th Internation Conference MOSIS'03, Ostrava, CZ, 2003, p. 143-148, ISBN 80-85988-86-0

[Hrubý ASU01] Hrubý, M., Rábová, Z.: Object Oriented Programming Languages in Modelling, In: Proceedings of The 27th ASU Conference, Rattwik, SE, ASU, 2001, p. 55-65, ISSN 1102-593X

[Hrubý ASU02] Hrubý, M., Rábová, Z.: Modelling of Real-world Objects using the HELEF Language, In: ASU Newsletter: a publication of the Association of SIMULA Users., roc. 28, c. 1, Stockholm, SE, p. 47-57, ISSN 1102-593X

[Hrubý 01] Hrubý, M., Rábová, Z.: Processes and Knowledge in One Simulation, In: Proceedings of 7th Conference STUDENT FEI 2001, FEI VUT Brno, Brno, 2001, p. 333-338, ISBN 80-214-1860-5

[Hrubý 02] Hrubý, M., Kočí, R., Peringer, P., Rábová, Z.: Tools for Creating of Multimodels, In: Kybernetes: An International Journal of Systems & Cybernetics, roc. 2002, c. 9, GB, p. 1391-1400, ISSN 0368-492X

[Hrubý 03] Hrubý, M., Kočí, R.: Modern Simulation Techniques and Tools, In: Proceedings of the International Workshop MOSMIC'2003, Zilina, SK, FRI ZU, 2003, p. 7-15, ISBN 80-8070-139-3

[Chambers 92] Chambers, C.: The Design and Implementation of the SELF compiler, an Optimizing Compiler for Object Oriented Languages, disertační práce, Stanford University, 1992

[Janoušek 02] Janoušek, V., Kočí, R.: PNtalk - An Open System for Prototyping and Simulation, In: Proceedings of The 28th ASU Conference, Brno, CZ, FIT VUT, 2002, p. 133-146, ISSN 1102-593X

[Kindler 80] Kindler, E.: Simulační programovací jazyky, SNTL, Praha, 1980

[Lee I.] Lee, J. H. M., Pun, P. K. C.: On Overview of the OLI Multiparadigm Language and Its Semantics, technická zpráva, The Chinese University of Hong Kong

[Lee II.] Lee, J. H. M., Pun, P. K. C.: Object Logic Integration: a Multiparadigm Design Methodology and a Programming Language, technická zpráva, The Chinese University of Hong Kong

[Liu 96] Liu, Ch.: Smalltalk, Objects, and Design, iUniverse.com, USA, dotisk vydání z roku 1996, ISBN 1-58348-490-6

[Madsen 00] Madsen, O. L.: Towards a Unified Programming Language, ECOOP 2000, LNCS 1850, pp.1-26, Springer-Verlag Berlin Heidelberg 2000

[Mohring MIMOSE] Mohring, M.: Social Science Multilevel Simulation with MIMOSE, technická zpráva, University of Koblenz-Landau

[Muller 95] Muller, M., Muller, T., Roy, P. V.: Multiparadigm Programming in Oz, výzkumná zpráva RR 95-16, In: "Workshop Visions for the Future of Logic Programming: Laying the Foundations for a Modern Succesor to Prolog", Portland, Oregon, 1995

[Nierstrasz 95] Nierstrasz, O., Tsichritzis, D.: Object-Oriented Software Composition, Prentice Hall, 1995, ISBN 0-13-220674-9

[Oren 02] \"{O}ren, T.: Future of Modelling and Simulation: Some Development Areas, Proceedings of the 2002 Summer Computer Simulation Conference

[Pawletta 96] Pawletta, T., Lampe, B., Pawletta, S., Drewelow, W.: An Object Oriented Framework for Modeling and Simulation of Variable Structure Systems, In: Proceedings of Summer Simulation Conf.'96, Portland, Oregon, 1996

[Peringer HP] Peringer HP: http://www.fit.vutbr.cz/~peringer/

[PNtalk] Janoušek, V.: Modelování objektů Petriho sítěmi, disertační práce, UIVT FEI VUT, Brno, 1998

[PNtalk HP] PNtalk project HP: http://www.fit.vutbr.cz/~janousek/pntalk/

[Praehofer 96] Praehofer, H.: An Environment for DEVS-Based Multi-Formalism Modeling and Simulation in C++, In: Proceedings of AIS, San Diego, 1996

[Praehofer DEVS] Praehofer, H., Zeigler, B. P.: On the Expressibility of Discrete Event Specified Systems, technická zpráva

[Roy 99] Roy, P. V., Haridi, S.: Mozart: A Programming System for Agent Applications, technická zpráva, http://www.mozart-oz.org/, 1999

[Rumbauch 91] Rumbauch, J. a kol.: Object-Oriented Modeling and Design, Prentice Hall, 1991, ISBN 0-13-630054-5

[SELF HP] Self HP: http://www.cs.ucsb.edu/labs/oocsb/self/

[Self4Linux HP] Self4Linux HP: http://www.gliebe.de/self/

[Shaalan 95] Shaalan, K. F. S.: Software Development Environment Based on Object-Oriented and Logic Programming Paradigms, disertační práce, Cairo University, Egypt, 1995

[SIMLIB HP] SIMLIB HP: http://www.fit.vutbr.cz/~peringer/SIMLIB

[SIMPACK] Fishwick, P. A.: SIMPACK: Getting Started with Simulation Programming in C and C++, technická zpráva, University of Florida

[Smalltalk HP] Smalltalk HP: http://www.smalltalk.org/

[Squeak HP] Squeak Smalltalk HP: http://www.squeak.org

[SWI-Prolog HP] SWI-Prolog HP: ftp://swi.psy.uva.nl/pub/SWI-Prolog/

[Ungar 91] Ungar, D., Chambers, C., Chang, B., Holzle, U.: Organizing Programs Without Classes, In: LISP and Symbolic Computation: An International Journal, Holandsko, 1991

[Vangheluwe TR] Vangheluwe, H. I., Vansteenkiste, G. C.: A Multi-Paradigm Modelling and Simulation Methodology: Formalisms and Languages, University of Gent, Belgie

[VisualWorks HP] Cincom HP: http://smalltalk.cincom.com/

[Wadler 92] Wadler,P.: The essence of functional programming, invited talk, In: Proceedings of 19'th Annual Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992

[Zeigler 02] Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation - Integrating Discrete Event and Continuous Complex Dynamic Systems, druhé vydání, Academic Press, USA, ISBN 0-12-778455-1

Publikace autora

- Hrubý, M., Peringer, P., Rábová, Z.: Knowledge-Based Simulator in Object-Oriented Environment, In: Proceedings of the 35th Spring International Conference MOSIS 2001, Hradec nad Moravicí, CZ, MARQ, 2001, p. 75-80, ISBN 80-214-1860-5
- Hrubý, M., Rábová, Z.: HELEF: A New Simulation Language, In: Proceedings of the 36th Spring International Conference MOSIS 2002, Rožnov pod Radhoštěm, CZ, MARQ, 2002, p. 39-47, ISBN 80-85988-71-2
- Hrubý, M.: Formal Specification of the HELEF Simulation Language, In: Proceedings of 37th Internation Conference MOSIS'03, Ostrava, CZ, 2003, p. 143-148, ISBN 80-85988-86-0
- Hrubý, M., Rábová, Z.: Object Oriented Programming Languages in Modelling, In: Proceedings of The 27th ASU Conference, Rattwik, SE, ASU, 2001, p. 55-65, ISSN 1102-593X
- Hrubý, M., Rábová, Z.: Modelling of Real-world Objects using the HELEF Language, In: ASU Newsletter: a publication of the Association of SIMULA Users., roc. 28, c. 1, Stockholm, SE, p. 47-57, ISSN 1102-593X
- Hrubý, M., Rábová, Z.: Processes and Knowledge in One Simulation, In: Proceedings of 7th Conference STUDENT FEI 2001, FEI VUT Brno, Brno, 2001, p. 333-338, ISBN 80-214-1860-5
- Hrubý, M., Kočí, R., Peringer, P., Rábová, Z.: Tools for Creating of Multimodels, In: Kybernetes: An International Journal of Systems & Cybernetics, roc. 2002, c. 9, GB, p. 1391-1400, ISSN 0368-492X
- Hrubý, M., Kočí, R.: Modern Simulation Techniques and Tools, In: Proceedings of the International Workshop MOSMIC'2003, Zilina, SK, FRI ZU, 2003, p. 7-15, ISBN 80-8070-139-3

Dodatek A

Syntaxe jazyka HELEF

```
Start ::= { Decl }
Decl ::= Info | Struct | Process | “(*” Id “*)”
Info ::= Id [ Cond ] Relation ( MExpr | Struct | Process )
Struct ::= StrType [ Inh ] “{” SDecl  “}”
Process ::= “process” [ Inh ] [ “(“ [ MExpr ] “)” ]  “{” PDecl “}”
Cond ::= “(“ MExpr “)”
Relation ::= “:=” | “=”
StrType ::= “environment” | “property” | “entity”
SDecl ::= { (Locals | Exports) } { Decl }
PDecl ::= { (Locals | Exports) } { Command }
Locals ::= “local” Id {, Id } “;”
Exports ::= “export” Id {, Id } “;”
Inh ::= [”@”] Id
PBlock ::= “{” { Command } “}”
Command ::= PBlock | If | While | For | AddRem
If ::= “if” Cond Command [ “else” Command ]
While ::= “while” Cond Command
For ::= “for” Id “:=” Range
AddRem ::= “++” Decl [”->” Id] | “--” Id [“->” Id]
MExpr ::= MObject { BinOper MExpr }
MObject ::= (UnOper MObj) | “(“ MExpr “)”
MObj ::= Const | Id | Special
Const ::= {Digit} [”.” {Digit}]
Id ::= Alpha {Digit|Alpha}
Alpha ::= ’a’..’z’ | ’A’..’Z’ | ’_’
Digit ::= ’0’..’9’
UnOper ::= “~” | “-”
BinOper ::= “+” | “-” | “*” | ...
Special ::= nil | true | false | self
```

Dodatek B

Příklady

B.1 Příklad modelu systému hromadné obsluhy v SIMLIB

```
#include "simlib.h"
Facility fac("A processor"); // objekt obslužné linky
Histogram table("Time in system", 0, 50, 20); // statistika
class Task : public Process { // třída transakce
      void Behavior() { // chování transakce
         double t = Time;
         Seize(fac);
         Wait(Uniform(5,10));
         Release(fac);
         table(Time-t);
      }
} ;
class Generator : public Event { // periodicky opakovaná událost
      void Behavior() {
         (new Task)->Activate();
         Activate(Time + Exponential(10));
      }
} ;
int main() { // simulační program
  Init(0, 10000);
  (new Generator)->Activate();
  Run();
}
```

B.2 Čtenář a slunce

```
sun = entity { // entita slunce
   isShinning(dayTime) = true // objekty dayTime a nightTime berme
   isShinning(nightTime) = false // jako externě dodávané
}
ableToRead = property { // vlastnost
   canIRead = anyLight // vyplyne dále
}
```

```
world = environment { // prostředí modelu
   entity sun { } // vložení slunce do modelu
   anyLight = isShinning
   man = entity {
      property ableToRead { }
      process (canIRead == false) {
         moveto house;
      }
   }
   house = environment {
      anyLight := true
   }
}
```

B.3 Křižovatky

Kód Prologu zveřejní proces `find` a jeho starovní podmínku `StartC=From.event` a data `To`, `W` (při změně `From` se volá proces `find`). Sekce (*∗id∗*) označuje místo vložení submodelu *id*.

```
car=entity{
  local At, Tp;
  (*prolog-find*)
  process() {
      while (At != Tp) {
        atomic { From=At; To=null; }
        ContinueC=(To!=null); /* bezi proces find */
        ContinueC=(time>^time+delay); /* jizda */
        At := To;
      }
  }
}
world = environment {
  (*prolog-roads*)
  car1=entity @ car {
    Fp=A; /* z bodu A do bodu Z */
    Tp=Z;
  }
}
```

B.4 Ukázka přenosu pravidel

```
okno=property { // vlastnost, která přenáší atribut
   // sunlight v závislosti na closed
   local closed;
   sunlight(closed=false)=parent.light
   sunlight(closed=true)=false
   closed:=false
```

```
}
room = environment {
   local light;
   light = sunlight OR artlight
}
world = environment {
   entity sun {}
   roomA = environment @ room {
      property okno {}
   }
}
```

B.5 Kombinovaná simulace se zapojeným spojitým externím simulátorem

Diskrétní jádro řídí spínání a vypínání topení, TKSL/C simuluje kapacitní jevy při akumulaci tepla v topném systému.

```
world = environment {
   (*TKSLC*) // generuje inp,out a time
   inp(out>=20)=0
   inp(out<20)=1
}
```