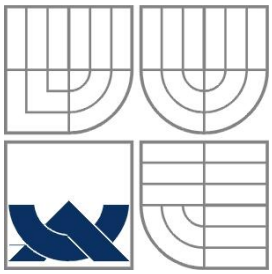
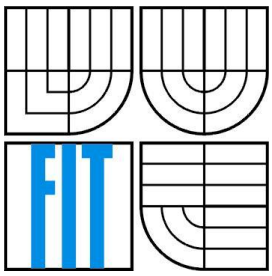
BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY

DEPARTMENT OF INTELLIGENT SYSTEMS

VYUŽITÍ METOD AGENTNÍHO PROGRAMOVÁNÍ V OBLASTI NÁVRHU A PROTOTYPOVÁNÍ DISTRIBUOVANÝCH SYSTÉMŮ

HABILITAČNÍ PRÁCE
HABILITATION THESIS

AUTOR PRÁCE
AUTHOR

Ing. FRANTIŠEK ZBOŘIL, Ph.D.

Věnováno Haničce, Terezce, Jakubovi a Haně

Abstract:

Topic of this work is programming of heterogeneous distributed computer systems with possible resource-limited nodes. A solution that is presented here combines agent-oriented programming with paradigmas of programming of systems with mobile codes. These codes are designed with respect to possible memory, energy and computing resources limitations on a hosting platform. Main part of this text is devoted to description of formal systems that we call ALLL systems. Along with these systems we introduce languages that are used for their programming. A string from an ALLL language represents a sequence of actions that transforms configurations of an ALLL system and that may also change configurations of other ALLL systems in a distributed system if there are multiple such systems. ALLL sytems that work with linear and hierarchical plans are introduced here. The linear plans represent some sequences of actions that transform bases of an ALLL system only. The hierarchical plans also contain actions that may extend the plan with a sub-plan and thus may modify hierarchy of the plan at runtime. ALLL systems working with hierarchical plans have then universal computation strength. Usage of these systems in real applications is done with connection with an agent platform that interprets the ALLL plans and also provides services to the agent as are communication and computational services. Concrete realization of such system is shown on examples of wireless sensor networks and the T-Mass simulation tool for which architecture with ALLL systems and supporting platforms has been implemented.

KEYWORDS: AGENT PROGRAMMING, MODELLING SUPPORT OF SYSTEM DEVELOPMENT, DISTRIBUTED SYSTEMS, WIRELESS SENSOR NETWORKS, MOBILIE CODES

Abstrakt:

Tématem práce jsou prostředky pro programování heterogenních distribuovaných počítačových systémů s ohledem na možné výskyty uzlů s omezenými zdroji. Prezentované řešení v sobě spojuje agentně orientovaný přístup k programování distribuovaných systémů a mobilními kódy a to i pro systémy s takovými uzly, které mají energetické, výpočetní a paměťové prostředky malé. Podstatnou část textu tvoří popis formálních systémů, které nazýváme ALLL systémy. Společně s těmito systémy jsou představeny jazyky, ve kterých se zapisuje jejich řídící kód. Řetězec ALLL jazyka reprezentuje posloupnost akcí, které transformují konfiguraci těchto systémů a které také mohou mít za následek změnu konfigurací jiných systémů v rámci nadřazeného distribuovaného systému. V textu jsou představeny ALLL systémy řízené lineárními a hierarchickými plány. Lineární plány jsou posloupnosti akcí, které manipulují s bázemi ALLL systému. U hierarchických plánů může být plán rozšiřován o podplány, pokud je provedena akce vyvolání plánů. Lze ukázat, že ALLL systémy interpretující hierarchické plány mají výpočetní sílu univerzálního výpočetního stroje. Využití ALLL systémů pro realizaci multiagentních systémů je v obecné rovině ukázáno jako spojení těchto systémů s platformou, která interpretuje ALLL kód a poskytuje další služby. Agent je díky platformě schopen komunikovat s okolím, využívat prostředky hostitelského systému, provádět komplexní výpočty a tak dále. Konkrétní nasazení ALLL systémů pro realizaci mobilních agentů v distribuovaných systémech je ukázáno na systémech bezdrátových senzorových sítí a na simulačním nástroji T-Mass .

KLÍČOVÁ SLOVA: AGENTNÍ PROGRAMOVÁNÍ, PODPORA VÝVOJE SYSTÉMŮ MODELOVÁNÍM, DISTRIBUOVANÉ SYSTÉMY, BEZDRÁTOVÉ SENZOROVÉ SÍTĚ, MOBILNÍ KÓDY

Obsah

Použité symboly

$\cdot$	Operace spojení dvou obecných prvků
$\perp$	Symbol neúspěšné operace
~	Symbol anonymního prvku
+	Operace přidávání do báze bez uspořádání
-	Operace odstranění z báze bez uspořádání
$\prec$	Relace specifikace
$\not\prec$	Relace 'není specifikací'
$\prod$	Operace restrikce
$\odot$	Symbol abstraktní operace báze
$\rightsquigarrow$	Relace uspořádání nad stavem báze s uspořádáním
$\hookrightarrow$	Operace vkládání pro bázi s uspořádáním
$\hookleftarrow$	Operace výběru pro bázi s uspořádáním
$\swarrow$	Operace odstranění pro bázi s uspořádáním
$\uparrow$	Relace maturace registrů
$\mapsto, \mathcal{S}_H \mapsto \gamma, n$	Transformace $\mathcal{S}_H$ systému podle stavu báze γ a výpočtu n
$\vdash_A$	Přechodová relace $\mathcal{S}_O$ systému
$\vdash_{A,INP}$	Funkce zpracování vstupů $\mathcal{S}_O$ systému
$\vdash_{A,APP}, \vdash_{A,TST},$	
$\vdash_{A,FIL}, \vdash_{A,EXE},$	
$\vdash_{A,SND}, \vdash_{A,RCV}$	Relace podle pravidel definujících aplikace akcí v $\mathcal{S}_O$ systému
$\vdash_{A,ACT}$	Relace aplikace atomických akcí $\mathcal{S}_O$ systému
$\vdash_{A,EVL}$	Relace aplikace obecných akcí $\mathcal{S}_O$ systému
$\vdash_{A,E}$	Relace provedení atomických akcí $\mathcal{S}_O$ systému
$\vdash_H$	Přechodová relace $\mathcal{S}_H$ systému
$\vdash_H^*$	Tranzitivní uzávěr přechodové relace $\mathcal{S}_H$ systému
$\vdash_{H,CMP}$	Přechodová relace $\mathcal{S}_H$ systému
$\vdash_{H,APP}, \vdash_{H,TST},$	
$\vdash_{H,FIL}, \vdash_{H,EXE}$	Relace podle pravidel definujících přechodovou relaci $\mathcal{S}_H$ systému
$\vdash_L$	Přechodová relace $\mathcal{S}_\mathrm{L}$ systému
$\vdash_L^*$	Tranzitivní uzávěr přechodové relace $\mathcal{S}_\mathrm{L}$ systému

$\ltimes_{L,I}$	Vnitřní přechodová funkce báze bez uspořádání
$\ltimes_{L,O}$	Výstupní funkce báze bez uspořádání
$\ltimes_{Q,I}$	Vnitřní přechodová funkce báze s uspořádáním
$\ltimes_{Q,O}$	Výstupní funkce báze s uspořádáním
$\rhd_L$	Krok výpočtu $\mathcal{S}_L$ systému
$\rhd_H$	Krok výpočtu $\mathcal{S}_H$ systému
$\rhd$	Krok výpočtu $\mathcal{S}_O$ systému
$\rhd_{DS}$	Přechodová relace $\mathcal{S}_{DS}$ systému
$\rhd_{DS,MSG}$	Relace předávání zpráv v $\mathcal{S}_{DS}$ systému
$\rhd_{DS,EVL}$	Krok výpočtu prvků $\mathcal{S}_{DS}$ systému
β_L	Systém ALLL báze bez uspořádání
β_Q	Systém ALLL báze s uspořádáním
Γ	Doména ALLL bází
γ	Vnitřní stav ALLL báze bez uspořádání
γ_0	Počáteční stav ALLL báze bez uspořádání
$\Delta^{i,n}(\Sigma)$	Doména obecných prvků i-té úrovně zanoření délky n nad množinou atomů Σ
$\Delta^{i}(\Sigma)$	Doména obecných prvků i-té úrovně zanoření nad množinou atomů Σ
$\Delta(\Sigma)$	Doména obecných prvků nad množinou atomů Σ
εx	Operace vyvolání podplánu
ϕ	Symbol prázdného vstupu na branách $\mathcal{S}_O$ systému
Φ	Výpočetní kontext $\mathcal{S}_H$ nebo $\mathcal{S}_O$ systémů
$\Psi(\{\Phi_1, \Phi_2 \dots \Phi_n\})$	Množina aktivních dat v množině výpočetních kontextů
ι	Stav na vstupech $\mathcal{S}_O$ systému jako prvku distribuovného systému
$\mu(\mathcal{R}^T, t)$	Maturační množina systému registrů $\mathcal{R}^T$
$\mu^{\sim}$	Uspořádaná množina substitucí
$\mu^{\mathrm{T,A}}$	Uspořádaná substituční množina se symboly registrů T a atomy A
η	Jméno $\mathcal{S}_O$ systému jako prvku distribuovného systému
o	Stav na výstupech $\mathcal{S}_O$ systému jako prvku distribuovaného systému
$\Pi(\mathrm{A})$	Množina plánů nad množinou akcí A
π	Plán ALLL systémů
σ^+	Operaci výběru podle předpony pro bázi s uspořádáním
Σ	Množina atomů ALLL systémů
τ	Registr věku 0, aktivní registr
${}^{n}\tau$	Registr věku n

Ω	Akce (atom) ukončeného plánu
Ω_T	Akce (atom) úspěšně ukončeného plánu
Ω_F	Akce (atom) neúspěšně ukončeného plánu
$\left[\frac{e}{t}\right]$	Operace substituce prvku e za atom t
$\mathrm{A}(\mathcal{S}_\mathrm{L}, \pi)$	Aplikace plánu π na $\mathcal{S}_\mathrm{L}$ systém
C	Konfigurace báze s uspořádáním
$\mathrm{C}(\Gamma)$	Doména konfigurace báze s uspořádáním
C_0	Počáteční konfigurace báze s uspořádáním
C_{0L}^{π}	Počáteční konfigurace $\mathcal{S}_\mathrm{L}$ systému s aplikovaným plánem π
C_L	Konfigurace $\mathcal{S}_\mathrm{L}$ systému s bází bez uspořádání
$D^n(\Sigma)$	Doména lineárních prvků nad množinou atomů Σ délky n
$D(\Sigma)$	Doména lineárních prvků nad množinou atomů Σ
$Data(\{\Phi_1, \dots \Phi_n\}, rc)$	Množina zpráv adresovaných prvku rc z výpočetních kontextů $\Phi_1, \dots \Phi_n$
$Data_\eta^{\bar{\Phi}}$	Seřazené zprávy prvku η výpočetních kontextů $\Phi_1, \dots \Phi_n$
$Dom_\mathrm{K}(\boldsymbol{\mathcal{S}_{DS}})$	Doména stavů $\boldsymbol{\mathcal{S}_{DS}}$ systému
$Dom_\mathrm{C}(\mathcal{S}_H)$	Doména konfigurací $\mathcal{S}_\mathrm{H}$ systému
$Dom_\Phi(\mathcal{S}_H)$	Doména výpočetních kontextů $\mathcal{S}_\mathrm{H}$ systému
$Dom_\mathrm{C}(\mathcal{S}_L)$	Doména konfigurací $\mathcal{S}_\mathrm{L}$ systému
$Dom_\mathrm{C}(\mathcal{S}_O)$	Doména konfigurací $\mathcal{S}_\mathrm{O}$ systému
$Dom_\Phi(\mathcal{S}_O)$	Doména výpočetních kontextů $\mathcal{S}_\mathrm{O}$ systému
$\mathcal{F}_{SL}(\mathcal{S}_\mathrm{L}, \pi)$	Výsledek aplikace plánu π na $\mathcal{S}_\mathrm{L}$ systém
$\mathcal{F}_{SH}(\mathcal{S}_\mathrm{H}, \pi)$	Rekurzivní funkce výpočtu koncové konfigurace $\mathcal{S}_\mathrm{H}$ systémů s plánem π
fst	Výběr prvního prvku z množiny obecných prvků podle uspořádání
$H(\Sigma, \mathrm{T}, O)$	Množina hierarchických plánů s množnou atomů Σ, symb. registrů T a akcí O
$H^i(\mathrm{T}, \Sigma, O)$	Množina hierarchických plánů s množinami Σ, T a O *i*-té úrovně
K	Konfigurace $\mathcal{S}_H$ nebo $\mathcal{S}_O$ systémů
$K_{0H}(\mathcal{S}_H, \pi)$	Počáteční konfigurace systému $\mathcal{S}_\mathrm{H}$ s aplikovaným plánem π
$L(\Sigma)$	Množina lineárních plánů nad množinou atomů Σ
$L^n(\Sigma)$	Množina lineárních plánů nad množinou atomů Σ délky n
$M(T, \Sigma)$	Doména uspořád. subst. množin pro množiny symbolů registrů T a atomů Σ
$M(\pi, \mathcal{R}^T, t)$	Operace maturace plánu π systémem registrů $\mathcal{R}^T$ a prvkem t
n_0	Počáteční stav výpočtu ALLL systémů
$\mathrm{O}(\mathrm{T}, \Sigma)$	Množina akcí pro množinu registrů T a množinu atomů Σ (platí I pro množiny akcí uvedené níže, varianta zápisu $\mathrm{O}(\mathrm{T} \cup \Sigma)$)

$\mathrm{O}_L(\mathrm{A})$	Množina akcí báze bez uspořádání nad množinou atomů A
$\mathrm{O}_Q(\mathrm{A})$	Množina akcí báze s uspořádáním nad množinou atomů A
$O_{RC}(\mathrm{A})$	Množina akcí přijímání zpráv z vyrovnávací paměti pro množinu atomů A
$O_{SN}(\mathrm{A})$	Množina akcí odesílání zpráv pro množinu atomů A
$O_{DC}(\mathrm{T}, \Sigma, O)$	Množina akcí přímého vyvolání podplánu s registry T, atomy Σ a akcemi O
$O_{DC}^{i}(\mathrm{T}, \Sigma, O)$	Množina akcí přímého vyvolání podplánu i-té úrovně
$O_{IC}(\mathrm{T}, \Sigma)$	Množina akcí nepřímého vyvolání podplánu
Op	Množina symbolů operátorů
$out(\Phi)$	Zobrazení data na vstupu prvku s výpočetním kontextem Φ
$\mathrm{P}(A)$	Množina substitucí pro množinu atomů A
$\mathrm{P}(T, \Sigma)$	Substituční množina pro množinu atomů Σ a množinu symbolů registrů T
fst	Funkce výběru prvního prvku z konfigurace báze s uspořádáním
pos_p^{γ}	Relace předcházení prvku p prvkům z množiny γ
pre_p^{γ}	Relace předcházení prvků z množiny γ prvku p
R	Charakteristika obecného systému
$\mathcal{R}^T$	Systém registrů s množinou symbolů registrů T
$\mathcal{R}un_{SH}^{\pi}$	Běh $\mathcal{S}_{\mathrm{H}}$ systému pro aplikaci plánu π
$\mathcal{S}$	Obecný systém
$\mathcal{S}_O$	Otevřený ALLL systém
$\mathcal{S}_{DS}$	Distribuovaný homogenní synchronní ALLL systém
$\mathcal{S}_H$	ALLL systém s hierarchickými plány
$\mathcal{S}_{\mathrm{L}}$	Lineární ALLL systém s bází bez uspořádání
sel	Funkce nedeterministického výběru z množiny obecných prvků
$sort\big(\mathrm{Data}(\overline{\Phi}, \eta)\big)$	Operace seřazení vstupních dat z množiny kontextů $\overline{\Phi}$ pro prvek η
T	Množina symbolů registrů
U	Univerzum obecného systému

Předmluva

V předkládané habilitační práci shrnuji své zkušenosti a poznatky získané několikaletou výzkumnou prací, která byla zaměřena na modelování a realizace distribuovaných systémů s malými výpočetními jednotkami s přímým využitím metod z teorie multiagentních systémů. Tematicky tato práce navazuje na mou disertační práci z roku 2004 [1], ve které jsem se zabýval návrhem agentně zaměřeného jazyka na nízké úrovni abstrakce a jeho použitím pro modelování distribuovaných systémů s agenty. V následujících letech jsem společně se svými studenty zkoumal různé přístupy a následně jsme realizovali několik různých systémů využívajících tento jazyk - právě zkušenost nabytá při tomto výzkumu spojeném s návrhem praktických aplikací je základem této habilitační práce.

1 Úvod

Východiskem pro hlavní část předkládané práce je pojednání o současném stavu na poli programovacích jazyků navržených pro implementaci (zejména) inteligentních umělých agentů v distribuovaných systémech. Problematika, na kterou se práce především zaměřuje, je obecně problematika *implementace* mobilních kódů, a to ať je chápeme jako inteligentní interpretovatelné kódy na platformách, tedy agenty, nebo jako data resp. meta-data zasílaná mezi platformami. Převážně budeme považovat tyto kódy za realizace inteligentních mobilních agentů, kteří jsou interpretovatelní na malých mobilních zařízeních. Pokud dojde k odklonu od obecného chápání těchto agentů, bude v textu vysvětleno, proč daný kód nespadá pod určitou skupinu agentů, v čem se liší, a co z agentních paradigmat naopak splňuje.

V práci se budeme věnovat i dalšími oblastem, ve kterých jsme formální systémy, které v této práci představíme, realizovali. Jedná se hlavně o oblast modelování distribuovaných a multiagentních systémů a o oblast bezdrátových senzorových sítí. V obou dvou případech najdeme potřebu vyjádřit činnost uzlů distribuovaných systémů stručně a v kódu, který je snadno interpretovatelný bez enormních nároků na výpočetní zdroje, jakými jsou paměť, výpočetní výkon a podobně. Jelikož v současnosti se stále hledá řešení, které by spojovalo mobilní agenty se systémy s malými výpočetními zařízeními s omezenými zdroji a podporovalo vývoj aplikací pro takové systémy s podporou prostředí modelování a simulací, vznikla i tato práce, která takovéto řešení nabízí.

Motivacím a cílům této práce se budeme věnovat podrobněji v nadcházejících podkapitolách. Úvodní kapitolu uzavřeme přehledem struktury celého textu a pak již se budeme věnovat samotnému tématu.

1.1 Motivace

Ve 21. století není složité najít systémy, které obsahují prvky s umělými výpočetními jednotkami a ve kterých jsou tyto prvky schopny vzájemné komunikace. Téměř každý člověk dnes přichází do styku s mobilními telefony, pracuje s Internetem, nebo se nechává při cestách navigovat družicovými systémy. Nastupují také inteligentní budovy schopné zvládat některé z procesů, které doposud potřebovaly asistenci člověka, autonomně, nebo lze tyto procesy řídit na dálku. Například můžeme řídit chod topných systémů zasíláním SMS zpráv a stejným způsobem zjišťovat tepelné podmínky v budově, spotřebu, správnou činnost systémů a tak dále. Na svět také přichází RFID technologie pro identifikaci objektů, například zboží v obchodech, a zájem je také o použití senzorových sítí jak třeba v automobilech či letadlech, tak také pro monitorování rozsáhlých nepřístupných prostředí, a tak dále. Všechny takovéto systémy odborník chápe jako distribuované výpočetní systémy a jednotlivé

výpočetní jednotky jako uzly těchto systémů. Pro ukázku možností reálného využití současných moderních technologií by mohl dobře sloužit příklad inteligentní budovy řízené systémy založenými na počítačích. Uvedeme příklad nemocnice. V té lze vysledovat řadu scénářů, ve kterých by inteligentní počítačové systémy mohly sloužit ke zvýšení bezpečnosti osob, provozu a majetku. Na jedné straně jsou doménou zájmu osoby, které se z různých důvodů mohou v systému nemocnice vyskytovat a hrát zde některou z rolí. Jsou zde lékaři a pacienti, jsou zde také úředníci, technický personál, studenti medicíny jako stážisté, zdravotní sestry a sanitáři či sanitářky, a také návštěvníci pacientů. V [2] jsme provedli studii možností sledování pohybu majetku ve Všeobecné fakultní nemocnici v Praze. Zaměřili jsme se na procesy pohybu lůžkovin v rámci nemocnice a mezi nemocnicí a průmyslovou prádelnou. Studie se primárně soustředila na využití RFID technologií pro identifikaci lůžkovin, ale byly zde představeny i scénáře, ve kterých by s využitím dalších současných technologií, jako například senzorových sítí, bylo umožněno zkvalitnit proces monitorování pohybu osob a majetku v instituci.

Tyto osoby a majetky bychom mohli chápat jako součást distribuovaného systému, ve kterém působí jako aktivní jednotky, které svým jednáním sledují některou ze svých motivací, nebo je s nimi manipulováno za nějakým účelem. To přináší do jejich fungování v systému rysy, které jsou známé z agentních a multiagentních systémů. U agentů se běžně určují jejich role v systému jako soubor povinností, oprávnění, prostředků, schopností a komunikačních protokolů. Pro vymezení jejich rolí bychom mohli využít například metodiku GAIA [3] či Prometheus [4] a dále bychom je mohli realizovat v některých z existujících vývojových prostředí multiagentních systémů. Prometheus umožňuje návrh těchto systémů pro BDI [5] systémy a podporuje transformaci navržených modelů do systému JACK [6] . Alternativu systému JACK, který je komerční produkt, lze najít v systémech JASON [7], 2APL [8], či starším systému JAM! [9]. Podporu heterogenním multiagentním systémům nabízí systém JADE [10], který je implementován v souladu se standardy organizace FIPA [11]. Výše uvedené metody a systému jsou v současnosti dobře prozkoumány a jsou podporou pro vývoj distribuovaných systémů s inteligentními prvky.

Motivací pro výzkum v této oblasti jsou také některé typy distribuovaných systémů, ve kterých mají výpočetní jednotky omezení, která komplikují použití výše uvedených metod. Další motivací je vytvořit prostředí, ve kterém by bylo možné modelovat takovéto systémy a tím podpořit jejich vývoj. Výsledkem úsilí ve zpracování tohoto problému je vznik distribuovaného systému, ve kterém jsou prvky jako výpočetní systémy řízený jazykem s agentními rysy. Jazyk je snadno interpretovatelný v systémech s omezenými prostředky, ale zároveň umožňuje relativně snadnou realizaci distribuovaných a multiagentních systémů.

Cíle práce

Obecně je cílem této práce prozkoumat použití výsledků z oblasti distribuovaných a multiagentních systémů v současných distribuovaných systémech s mobilními zařízeními. Bude zde popsán nový jazyk pro programování distribuovaných systémů založený na procesních kalkulech a agentním programování. Budeme diskutovat jeho paradigmatické zařazení, definovat jeho syntaxi i sémantiku a způsob interpretace a realizace distribuovaných systémů s použitím tohoto jazyka. Pro přehlednost uvedeme konkrétní cíle v následujících bodech.

- Navrhnout agentní systém, který by byl řízen kódem v nějakém jazyce, a ve kterém by programy zapsané v tomto jazyce, jejich umístění, přenos mezi systémy, a jejich interpretace vyžadovaly minimální prostředky.
- Navrhnout způsob začlenění těchto systémů do rozsáhlejších distribuovaných a multiagentních systémů.
- Představit implementaci těchto systémů v zařízeních s kritickým omezením prostředků, obvykle založených na mikrokontrolérech.
- Demonstrovat využití těchto systémů pro modelování distribuovaných systémů s inteligentními prvky a využití modelování pro návrh reálných aplikací.

1.2 Struktura práce

Uvedení současného stavu výzkumu v oblasti programování distribuovaných inteligentních systémů bude obsahem druhé části této práce. Zmíněny budou hlavní paradigmata současných programovacích jazyků a poté bude pozornost zaměřena na mobilní kódy, distribuované a agentní systémy, a dále na agentně orientované jazyky. Právě jazyky, jejich paradigmata, a distribuované multiagentní systémy budou oblastmi, ve které se bude tematicky pohybovat jádro práce. Přehled současného stavu uzavřeme zmíněním způsobů modelování distribuovaných systémů a to jednak procesními kalkuly, a jednak Petriho sítěmi.

Jádro práce se bude věnovat původním výpočetním systémům souhrnně nazývaným ALLL (Agent Low Level Language) systémy. V názvu se odráží to, že tyto systémy interpretují jazyk na nízké úrovni abstrakce. Tento jazyk nese stejný název, tedy ALLL. Systémy, které tento jazyk, nebo jeho podmnožinu, interpretují, budou mít v následujícím textu přívlastek ALLL.

Kapitola 3 bude nejrozsáhlejší ze všech kapitol. Výklad zahájíme formálním pohledem na podsystémy ALLL bází a operace nad nimi. Tyto báze budou později využívány v navrhovaných ALLL agentních systémech, na jejichž popis dojde následně. Postupně formálně představíme několik variant ALLL systému, od systému jehož chování je dáno pouze lineárním plánem coby posloupnosti akcí, přes systém zpracovávající hierarchické struktury plánů, až po otevřený systém, jehož chování je dáno plánem a hodnotami na vstupech. Vždy uvedeme jejich strukturu a jejich fungování jako výpočetního systému.

Další částí tohoto tematického bloku bude zaměřena na popis multiagentního systému s ALLL prvky. K tomu bude třeba představit agentní platformu, její roli v multiagentním systému, a jak funguje vztah mezi agentem a platformou v ALLL multiagentním systému. Uvedeme základní služby, které platforma agentům nabízí, a jak je může agent využívat. Následovat bude definice jazyka ALLL agenta, zejména jeho syntaxe, protože sémantika většiny akcí, které lze v tomto jazyce zapsat, bude již představena v kapitole 3. Na závěr budou ukázány některé programovací techniky pro tento jazyk.

Poslední část práce se bude týkat aplikací s ALLL systémy. Bude zde představen simulačním nástroj T-Mass a systém WSAgeNt, což je implementace ALLL v prostředí bezdrátových senzorových sítí. Následovat budou ukázky konkrétních modelů a aplikací implementovaných v T-Mass a WSAgeNt. Závěrem zmíníme možnost propojení modelovacího nástroje, agentních systémů a bezdrátových senzorových sítí za účelem podpory vývoje, testování a správy aplikací pro bezdrátové senzorové sítě.

2 Výstavba a modelování systémů, distribuovaných systémů a multiagentních systémů

Název této kapitoly vymezuje značně širokou oblast. Uvedeme si v ní ale jen několik témat souvisejících se systémy, které jsou obsahem této práce. Předně uvedeme rozdělení programovacích jazyků z hlediska paradigmat. Námi navržený jazyk v sobě skloubí několik ze současných způsobů realizací programovacích jazyků a toto rozdělení nám umožní získat nadhled nad principy různých realizací programovacích jazyků a pochopit, na jakých základech budeme náš nový jazyk stavět. Dále se zaměříme na některé otázky související s distribuovanými systémy obsahujícími inteligentní uzly, s nimi souvisejícími multiagentními systémy, a na systémy, ve kterých se mohou procesy během jejich provádění v rámci distribuovaného systému přenášet mezi výpočetními jednotkami. Poté se zaměříme na oblast modelování distribuovaných systémů. Modelování těchto systémů a formální přístup k jejich strukturálním a výpočetním vlastnostem je vhodnou cestou, jak ukázat princip výstavby těchto systémů a jejich fungování. Oblast kalkulů, zejména procesních kalkulů, byla vedle oblasti programování agentních systémů jednou z hlavních inspirací pro naše řešení v podobě agentního řídícího jazyka, které bude představeno v dalších kapitolách.

2.1 Programování počítačových systémů

Programovací jazyky jsou formální jazyky pro zápis kódu, který je schopen stroj vykonávat. Význačné rysy těchto jazyků a principy tvorby programů v nich jsou dány jednak teoretickými základy, na kterých jsou postaveny, a dále také typy systémů, pro jejichž programování mají být použity. Tyto význačné vlastnosti a principy nazýváme paradigmaty a na základě těchto paradigmat můžeme provádět kategorizaci programovacích jazyků. Obvyklým způsobem programování je zápis *algoritmů* v nějakém imperativním programovacím jazyce. Programem je zde posloupnost příkazů sestavená podle zvoleného algoritmu. Výpočty lze ale provádět i jinými způsoby, jako například výpočtem funkcí, nebo deklarací formulí, ze kterých stroj vyvozuje platnost či neplatnost jiných formulí. Pro paralelní systémy, respektive pro programování takovýchto systémů, také existuje řada teoretických systémů, které poskytují základ pro programovací jazyky distribuovaných systémů.

Následující sekce představí několik v současnosti významných paradigmatických kategorii jazyků trochu blíže.

2.1.1 Imperativní a procedurální jazyky

Programování systémů zápisem algoritmu ve formě jednotlivých kroků výpočtu (příkazů) je základním paradigmatem imperativních jazyků. Myšlenka imperativního programování se nabízí vzhledem k používané von Neumannovské počítačové architektuře, která vykonává instrukce strojového kódu. Jazyky vyšších úrovní abstrakce zachovávají tento přístup, obohacují ale programátorské možnosti o deklarace strukturovaných příkazů, datových typů a programových konstrukcí. Podmnožinou imperativních jazyků jsou procedurální jazyky, které část výpočtu abstrahují v procedury použitelné ve více částech programu, a ve kterých se dále mohou vyskytovat moduly či knihovny procedur. Dlouhá léta v minulosti většina praktiků chápala imperativní přístup jako ten pravý pro tvorbu programů a i dnes zůstává algoritmický přístup k řešení problémů a zápis algoritmů v imperativních jazycích základní profesní výbavou programátorů.

2.1.2 Funkcionální programovací jazyky

Funkcionální jazyky se řadí mezi deklarativní jazyky a umožňují deklarovat a vyčíslovat rekurzivní funkce. Teoretickým základem pro funkcionální přístup je výpočetní model λ-kalkul, který byl vyvinut

Alonsem Churchem ve dvacátých letech minulého století. Na λ-kalkul navázal John McCarthey jazykem LISP (List Processing) [12]. Jelikož se jedná o výpočet rekurzivních funkcí, je v tomto jazyce rekurze základní programátorskou metodou. V některých rysech jazyka se odráží i imperativní přistup k psaní programů, a to například v možnosti přiřazovat proměnným objekty, který byl nejspíš do jazyka zanesen pro usnadnění použití jazyka programátorům, kteří běžně mají zažitý imperativní způsob programování. Z dalších funkcionálních jazyků se zde patří uvést populární jazyk Haskell [13], který pečlivěji dodržuje formální základ funkcionálního programování.

2.1.3 Logické deklarativní jazyky

Nejznámějším představitelem programovacích jazyků založených na formálních logických systémech je jazyk PROLOG (PROgramming in LOGic) [14]. Vyhodnocování formulí spadajících do podmnožiny jazyka predikátové logiky prvního řádu a majících formu Hornových klauzulí, to je klauzulí s jedním pozitivním literálem, probíhá postupným vyhodnocováním jednotlivých negativních literálů v klauzuli. Pokud je možné vyhodnotit jako pravdivé všechny negované predikáty v těchto literálech, musí být pravdivým i onen jeden pozitivní literál. Vyhodnocování klauzule obvykle probíhá algoritmem s navracením, a to proto, aby byl prozkoumán všechen prostor možných pozitivních vyhodnocení těchto predikátů v literálech. Jako dalšího zástupce deklarativních jazyků lze uvést jazyk Gödel [15], který se striktněji drží principů logického deklarativního programování. Tento systém také umožňuje vytváření programových modulů a jejich vzájemné propojování [16].

2.1.4 Objektově orientované jazyky

Objektově orientovaná paradigmata vychází z objektového přístupu k modelování systémů v prostředích SIMULA 67 [17] [18]. Od devadesátých let minulého století se jedná a hojně rozšířený a populární způsob programování aplikací. Mezi základní paradigmata se zde řadí abstrakce, zapouzdření, dědičnost a polymorfismus. K interakci mezi objekty dochází předáváním zpráv, které spouští metody deklarované v objektu příjemce. Systémy jsou vytvářeny jako programy, které deklarují třídy objektů a vztahy mezi nimi. Objekty jsou instance těchto tříd a chování celého systému je dáno chováním jednotlivých objektů v systému. Mezi reprezentanty čistě objektového přístupu patří jazyk Smalltalk, objektové rysy mají jazyky Java, C++ a další.

2.1.5 Jazyky pro programování distribuovaných systémů

Za distribuované systémy považujeme paralelní systémy, ve kterých procesy komunikují předáváním zpráv. Pro programování distribuovaných systémů existují programovací jazyky, které v sobě zahrnují konstrukce umožňující realizaci nebo modelování takovýchto systémů. Můžeme k nim řadit jazyk ADA umožňující interakci procesů metodou Rendezvous, což je metoda vzdáleného volání přístupových bodů. Dále sem patří jazyk pro modelování distribuovaných systém OCCAM [19] či jazyk LINDA [20] realizující interakci procesů přes nástěnku. V současnosti se pro realizaci používají knihovny MPI a MPI2 [21], které umožňují programovat meziprocesní komunikaci předáváním zpráv a kolektivní operace, jako jsou všesměrové vysílání, suma prefixů a tak dále. K teoretickým systémům se zde řadí CSP (Communicating Sequential Processes) [22] a procesní kalkuly, o kterých budeme podrobněji psát později.

2.1.6 Agentně orientované jazyky

Jako poslední zmíníme paradigmata programování inteligentních agentů. Zde se nacházíme na vyšší úrovni abstrakce specifikace systémů. Agentní program by měl být programem pro systém, který je schopen praktického usuzování v obecně dynamickém, nepřístupném a nedeterministickém prostředí. Wooldridge a Jennings v [23] uvádějí dnes většinově přijímaná paradigmata racionálního

agenta, která zahrnují autonomnost, reaktivitu, proaktivnost a sociální schopnosti agenta. Proaktivnost a sociální schopnosti nemusí být ale obsaženy u takzvaných reaktivních agentů. Reaktivním i racionálním agentům budeme věnovat podkapitolu, kde jejich principy vysvětlíme blíže.

Systém, který představíme v této práci, bude programován jazykem, jenž bychom mohli nazvat multiparadigmatickým. Bude totiž zahrnovat většinu z výše uvedených paradigmat. To není neobvyklé, zejména ve vztahu k agentním jazykům, které v sobě zahrnují potřebu komunikace v rámci multiagentního systému, deklarativní přístup pro práci s mentálními stavy agentů, funkcionální rysy při vytváření struktur záměrů a rovněž imperativní části při deklarování plánů agenta jako posloupnosti akcí k vykonání. Následující podkapitoly podrobněji představí některé z paradigmatických přístupů k vytváření programů, které v našem jazyce využijeme.

2.2 Distribuované systémy

Uvedli jsme, že distribuovanými systémy rozumíme takové paralelní systémy, které spolu interagují předáváním zpráv. Programování distribuovaných systémů znamená programování systémů, které působí v rámci většího systému, což přináší výhody i nevýhody. Výhody distribuovaných systémů jsou obecně v tom, že systémy mohou spolupracovat při řešení úloh sdílením svých zdrojů a schopností. Na druhou stranu fungování v rámci většího systému může přinášet konflikty. Opět zmíníme sdílené zdroje, o které můžou procesy soupeřit, a systém by měl být zabezpečen proti nežádoucím důsledkům konfliktů, které mohou během soupeření vznikat. Jiným problémem paralelismu je synchronizace procesů, kdy požadujeme nějaké uspořádání mezi událostmi různých procesů v systému.

V případě distribuovaných systémů vychází metody řešení konfliktů z toho, že systémy k interakci potřebují buď prostředníka, nebo spolu komunikují zprávami. V sekci o jazycích pro distribuované systémy jsme uvedli několik prostředí, které umožňují programovat paralelní aplikace. V případě LINDAy se jedná o systém, který využívá jiný prvek systému, nástěnku, pro řešení problémů paralelismu. V systémech OCCAM a MPI lze realizovat zasílání zpráv mezi procesy a tím i využít metod pro řešení konfliktů a synchronizace založených na zasílání zpráv. Jednou z metod synchronizace procesů v distribuovaných systémech je použití logických hodin, například Lamportovým algoritmem [24]. K přístupu ke zdrojům můžeme použít systém služeb, kde zdroje spravuje agent. Tento agent registruje nabízené služby, včetně služeb poskytnutí zdrojů, a buď zprostředkovává služby jiným agentům odkazem na vhodné poskytovatele požadovaných služeb, například pomocí systému adresářů služeb, nebo přímo řídí jejich provádění poskytovatelem.

2.3 Systémy s mobilními kódy

Mobilita bude jednou z předpokládaných schopností agentů v systémech, které se chystáme představit v této práci. S ohledem na energetickou náročnost přenášení kódů u některých malých zařízení je mobilitou dán požadavek na co nejmenší velikost tohoto kódu. To znamená, že program agenta by měl být co nejmenší a přitom by v něm mělo být popsáno agentovo chování v systému včetně rozhodování, komunikace či zamýšleného pohybu v síti.

Nyní uvedeme několik argumentů, proč je dobré realizovat mobilitu při implementacích distribuovaných multiagentních systémů. Lange a Oshima shrnuli přínosy použití mobilních kódů v [25]. Identifikovali celkem sedm argumentů, které takovýto přístup k realizaci aplikací pro distribuované systémy podporují.

- *Redukce zátěže sítě* – data jsou zpracovávána na místě, kde byla pořízena, namísto toho, aby byla odesílána na server. Agent, který data zpracovává, má menší velikost, než mají zpracovávaná data a proto je redukována zátěž sítě.
- *Prevence zpožďování odezvy v síťovém provoz* – agent může jednat přímo na vzdáleném zařízení místo toho, aby řídil toto zařízení zasíláním řídících zpráv.
- *Zapouzdřování protokolů* – agent je schopen se přemístit na některé z uzlů, ustanovit mezi nimi komunikační kanál, a realizovat specifické komunikační protokoly.
- *Autonomní a asynchronní provádění kódu* – agenti se mohou lépe adoptovat v mobilních systémech, které nemají spolehlivé a stálé komunikační a přenosové kanály. Agenti jsou schopni vykonávat úlohy autonomně a zasílat výsledky svého konání zpět na domovské zařízení v okamžicích, kdy je spojení dostupné.
- *Dynamická adaptace agentů* – agenti se mohou sami autonomně rozmisťovat v síti způsobem, který považují za optimální vzhledem k řešené úloze.
- *Heterogenita* – jak z pohledu softwarového, tak i hardwarového, systémy počítačových sítí jsou heterogenní. Agenti jsou schopni se adaptovat na různé typy agentních platforem a provádět své úlohy v takovýchto heterogenních systémech.
- *Robustnost a odolnost proti chybám* – agenti jsou schopni čelit chybám během komunikace a chybám či výjimkám způsobeným vlivy okolního prostředí.

Nicméně negativa použití mobilních agentů v některých typech distribuovaných systémů, jako jsou bezdrátové senzorové, převažují, a proto v současnosti nejsou hlavním proudem v návrhu těchto systémů. To ovšem neznamená, že by nebyli využíváni zejména ve fázi prototypování těchto systémů.

2.4 Agentní systémy

V dosavadním textu jsme několikrát zmínili pojem agent v souvislosti s distribuovanými systémy. V současnosti jsou agenti, nebo přesněji umělí softwaroví agenti, chápáni jako procesy, které jsou schopny samostatného jednání v systému a jsou schopni pracovat v prostředí, ve kterém jsou umístěni tak, aby jejich jednání vedlo k naplnění jejich cílů. Přestože tato práce není primárně o umělých agentech, budeme v ní využívat některé agentní techniky pro realizaci našich ALLL systémů. Proto v této podkapitole poskytneme přehled základních agentních systémů a přístupů k agentně orientovanému programování.

Agentní systém je běžně chápán jako systém s umělými agenty, kteří jednají v nějakém obvykle dynamickém a nepřístupném prostředí ve prospěch svého klienta. Definice obecného agentního chování [26] používá zobrazení sekvence vstupních podnětů na agentových senzorech na jeho výstupní akce. Formálně to lze napsat následovně.

$$Ag: R^E \rightarrow A$$

V tomto zápise je R^E množina běhů systému končících ve stavu prostředí E a A je množina akcí agenta.

Uvedené zobrazení modeluje chování jakéhokoliv prvku, který reaguje na přijaté podněty nějakou akcí. Může se jednat o model kombinačních nebo sekvenčních obvodů, reaktivních prvků jako například tepelných čidel a podobně.

Agent je tedy prvek systému s nějakým chováním. Vágní definice by agenty od běžných prvků systému vymezovala tím, že agent by měl vykazovat známky inteligentního chování. Jeho reakce na vstupní podměty by měly být vzhledem k účelu, pro který se v systému nachází, racionální, a akce by měl agent vybírány podobně, jak by je volila inteligentní bytost.

2.4.1 Reaktivní agenti

Reaktivní agenti, jak naznačuje jejich název, jsou agenti, kteří nemají symbolickou reprezentaci prostředí, ve kterém se nachází, ale pouze reagují na obdržené vjemy na senzorech podle svých vnitřních pravidel. Respektovanými texty, které popisují přínos reaktivních agentů pro umělou inteligenci, jsou publikace Raymonda Brookse [27] [28]. Brooks v těchto textech kritizuje klasický přístup k řešení problémů zahrnutých do oblasti umělé inteligence zejména proto, že jednotlivé podoblasti jsou zkoumány odděleně, a k jejich propojení má dojít až po jejich dostatečném zvládnutí. Brooks tvrdí, že toto není zárukou správného chování inteligentního systému jako celku a je i skeptický k možnostem takovéhoto propojení. Naopak zastává názor, že bychom měli postupovat od jednoduchých systémů ke složitějším a mít k dispozici v každé fázi vývoje hotový inteligentní prvek schopný správného chování v reálném prostředí.

Brooks také prezentuje myšlenku, že prostředí samo je svojí nejlepší reprezentací a že agent si nemusí uchovávat model okolního prostředí v nějaké symbolické podobě a odvozovat svoje chování z tohoto modelu. Přesto je reaktivní agent schopen chování, které projevuje známky inteligence. Brooksovým návrhem na realizaci reaktivního agenta byl robot s takzvanou subsumpční architekturou [29]. Tato architektura byla inspirována poznatky ze studia chování hmyzu. Je realizována jako horizontálně vrstvená architektura, ve které jsou jednotlivé vrstvy schopny vykonávat chování podle vestavěného stavového automatu a ve které jednotlivé vrstvy mají hierarchické uspořádání. Hierarchicky nadřazené vrstvy mohou modifikovat vstupy na nižších vrstvách a také blokovat výstupy těchto vrstev. Pokud hierarchicky nadřazené vrstvy nemají patřičné vstupy, na které dokáží reagovat, může chování agenta řídit vrstva na nižší úrovni.

2.4.2 Racionální agenti

Racionálním agentem rozumíme agenta, který je schopen následovat svůj cíl a to tak, že jeho chování vůči tomuto cíli je racionální. To znamená, že pokud koná, tak koná tak, aby cíle dosáhl, nebo přiblížil jeho dosažení. Racionální agenti překračují pole vymezené reaktivními agenty také v tom, že pracují se symbolickou reprezentaci prostředí, ve kterém se nachází. Nad touto reprezentací provádí usuzování jak jednat v tomto systému vzhledem k přijatým cílům svého chování. U racionálních agentů dochází k takzvanému praktickému rozhodování, které má dvě části. První z částí je volba cíle, který má být následován, a druhou částí je pak výběr akce nebo sestavení plánu k dosažení či přiblížení tohoto cíle. Reprezentaci prostředí, nebo lépe řečeno agentovy představy o tom, jak vypadá okolní prostředí, je jedním z takzvaných mentálních stavů, které utváří vnitřní stav agenta. Dalšími mentálními stavy mohou být agentovy cíle, záměry, jeho povědomí o závazcích, které má v rámci systému, a tak dále.

Do oblasti racionálních agentů spadá několik architektur agentních systémů, které byly vyvinuty v posledních dvaceti letech. Mezi nejvýznačnější představitele těchto architektur patří systémy Agentně orientovaného programování [30] a BDI agentní systémy [31] [32].

2.4.3 Agentně orientované programování

Prvním ze dvou systémů, a dovolíme si tvrdit, že méně reflektovaný, je systém pro agentně orientované programování, který popsal Yoav Shoham [30]. Myšlenka agentně orientovaného programování je založena na přechodu od objektového programování k agentnímu a od deklarativního k postdeklarativnímu. Postdeklarativní přístup má být takový, že v programu jsou deklarovány pouze cíle, nebo zadání problému, a agent sám volí způsob jejich dosažení. Systém Agent-0, představený Shohamem v tomto článku, je ukázkou přístupu ke psaní programů, který se přibližuje k myšlenkám postdeklarativního programování.

Mentálními stavy agenta v tomto systému jsou jeho představy o stavu prostředí, znalosti o svých schopnostech a závazky vůči sobě nebo ostatním agentů v systému. Programování zde znamená deklarování podmínek, za jakých agent přijímá závazek vůči jinému agentovi, nebo i vůči sobě samému. Shohamův původní návrh počítal s deklarováním podmínek pro přijímání závazku takovým způsobem, že v podmínkách budou specifikovány jednak typy zpráv, které vyvolají úvahy o přijetí závazku (informování, žádost …), a jednak mentální stavy, které musí být platné, aby agent závazek přijal. Podmínka přijetí závazku v systému Agent-0 by pak mohla vypadat následovně.

```
(COMMIT
    (AND(
        (REQUEST ?time ?agt
    (INFORM ?agt ?time, 'time is'))
        (NOT (CMT ?agt2)
        (REFRAIN (INFORM ?ag3 ?tm2 'time is'))))
    (INFORM ?agt (+ time (*100 m)) 'time is' now))
```

Podmínka je v tomto případě taková, že platí, pokud je agent požádán jiným agentem, aby mu poskytl informaci o aktuálním čase, a zároveň agent dříve neslíbil nějakému jinému agentovi, že nebude přijímat takovýto závazek. Pokud je podmínka splněna, agent se zaváže, že bude za 100 minut od obdržení žádosti žadatele informovat o aktuálním čase. V zápise pravidla se objevují konstrukce logických vazeb mezi agentovými mentálními postoji. Konkrétně takové, že závazek platí, pokud agent ví, že obdržel požadavek, a zároveň také ví, že nedal dříve závazek tyto požadavky odmítat. Dále jsou zde proměnné, jejichž identifikátory následují za symboly otazníku. Také je zde použita systémová proměnná 'now', která obsahuje aktuální systémový čas, operace sčítání a násobení v prefixovém tvaru, či konstanta 'm' obsahující číslo odpovídající jedné minutě času.

Kromě pravidel pro přijímání závazků může program v systému Agent-0 obsahovat i deklaraci agentových představ.

Ačkoli od doby svého zveřejnění v roce 1993 byl tento systém řadu let vnímán spíše jako teoretická studie bez nějaké v širší míře používané realizace, s nástupem systému Agent Factory [33] se Shohamův přístup agentně orientovaného programování založený na přijímání závazků objevil v podobě programovacího jazyka AFAPL2 (Agent Factory Programming Language) pro agentní toto programovací prostředí.

2.4.4 Agenti řízení záměrem

Jiným, a dnes nejrozšířenějším agentním systémem z rodiny agentních systémů založených na mentálních stavech, jsou BDI agenti. Mentálními stavy jsou v tomto případě představy agenta o stavu

systému, jeho přání a záměry. Tento systém je založen na publikaci filosofa Michaela Bratmana [34], ze které přejímá její myšlenky ohledně řízení lidského chování. Zatímco první dva mentální stavy člověka, respektive umělého agenta, korespondují s pojmy z klasické umělé inteligence, konkrétně se znalostmi a cíli inteligentních systémů, záměr je něco nového, co BDI systémy přináší do oblasti umělé inteligence. Podle Bratmana je záměrem nějaké přání přijaté jako cíl, a to v okamžiku, kdy jedinec uvěří, že lze tohoto cíle dosáhnout. Dále je přijetím záměru dán závazek sobě samému, že další chování bude vedeno racionálně vzhledem k tomuto cíli. Záměr je odložen pouze v případě, že je záměru dosaženo, nebo jedinec pozná, že záměru nemůže být nikdy v budoucnu dosaženo. Jedno z pravidel definující záměr je i to, že agent nutně nemusí zamýšlet všechny efekty, které během sledování záměru způsobuje. Jako příklad je uváděn záměr dosáhnout vyléčení zubního kazu návštěvou zubaře, a to i za cenu bolestivého zákroku, který není přáním agenta.

Na teoretické rovině se BDI systémy zabývali zejména Anand Rao a Michael Georgeff, a to ve svých pracích [31, 32], kde jednak představili formální model pro reprezentaci těchto mentálních stavů a také jejich vzájemné působení pomocí multimodálních logik. V navazující práci představili systém AgentSpeak(L) [35], který je dodnes obecně přijímaným výpočetním modelem pro agentní systémy řízené záměrem. Téma BDI systémů zpracovává i Michael Wooldridge v [5]. Následují implementace BDI systémů JAM! [9], 3APL [36] a 2APL [8], JACK [6] či JASON [7]. Všechny tyto systémy umožňují programovat agenty právě podle principů systémů AgentSpeak(L). Jako další teoretický výpočetní systém založený na BDI principech můžeme jmenovat systém dMars [37, 38], který na rozdíl od systému AgentSpeak(L) umožňuje definovat plány jako stromové struktury. Společné rysy s BDI systémy vykazuje i systém procedurálního plánování PRS (Procedural Reasoning System) [39].

Agentní program je u BDI systémů deklarací znalostí a počátečních cílů agenta ve formě predikátů, a dále také databází plánů, které mohou být vykonávány pro dosažení nějakého cíle nebo podcíle. Záměrem, nebo lépe řečeno modelem záměru, je zde struktura složená z plánů, které můžou vést ke splnění celého nebo části záměru, a které se do struktury záměru přidávají za běhu agenta. Každý z plánů má deklarováno, k jakým cílům se vztahuje, jaké jsou podmínky jeho použití, a co se má vykonat, aby byly uvedené cíle dosaženy. Pokud je nějaký plán zvolen jako prostředek k dosažení cíle, pak je přidán do struktury záměru, který podcíl stanovil. Pokud je cíl stanoven samostatně, je pro něj vytvořen nový záměr a zvolený plán je plánem na nejvyšší úrovni záměru. Záměry jsou prováděny od plánů na nejnižší úrovni. Plány jsou vykonávány, dokud není vykonáno celé tělo plánu, nebo pokud nějaká akce či stanovený podcíl z těla plánu neskončí neúspěchem. Tím je agent schopen reagovat na nepředpokládané situace, které zabrání dosažení cíle zvoleným plánem. Více o tom, jak fungují BDI systémy, a jak je programovat, lze nalézt v [7].

Právě princip rušení části záměru v případě neúspěchu je jedním z principů, který promítneme do systémů ALLL, jejichž popis bude tvořit jádro této práce. Od agentních systémů s jedním agentem nyní přejdeme k systémům, ve kterých se může vyskytovat agentů více.

2.4.5 Multiagentní systémy

Agentní systém lze rozdělit na dva podsystémy – na podsystém prostředí a podsystém agentní. Agentní systém zkoumá chování jednoho agenta v prostředí, které nemusí být agentovi zcela přístupné, a jehož chování agent nemusí znát. Přístupy k realizaci takových agentních systémů byly stručně představeny v předchozích sekcích. Pokud ale uvažujeme více než jednoho agenta v systému, pak z pohledu jednoho agenta obsahuje jeho prostředí i několik jiných agentů, a pak hovoříme o

multiagentním systému. S tím přichází i další otázky a problémy k řešení. Konkrétně jde o to, jak by měly tyto agentní jednotky navzájem interagovat.

Studium multiagentních systémů je prováděno na úrovni interakční a úrovni organizací. Interakční úrovní je myšlena komunikace v systému, komunikační jazyky, přenášení zpráv v multiagentním systému a sémantika jednotlivých zpráv z hlediska jejich dopadu na mentální nebo jiné stavy agentů. Současný přístup k řešení komunikace v multiagentních systémech je založen na myšlence řečových aktů, které teoreticky studoval John Austin [40]. Na základě těchto teorií vznikly agentní komunikačních jazyky, které řečové akty používají k vyjádření agentova postoje k obsahu zprávy. V současnosti se jedná o jazyk KQML [41], který je již ale v mnoha ohledech zastaralý, a pak hlavně o jazyk ACL [11]. Studium sociálních schopností agentů se provádí na úrovni jejich organizací. Tyto schopnosti zahrnují schopnosti rozpoznat a řešit situace, kdy mezi agenty dochází ke konfliktu [42, 43], nebo kdy je v systému situace výhodná pro vzájemnou spolupráci. Organizační pohled zahrnuje modely s agenty poskytujícími závazky, vytvářejícími koalice a sociální skupiny, respektujícími normy v koalicích a tak dále. FIPA [11] poskytuje standardy pro heterogenní multiagentní systémy, kde definuje strukturu takových systémů jako systémů s prostředím, agentními platformami a agenty. Dále jsou v těchto standardech definovány komunikační jazyky na různých úrovních, interakční protokoly, nebo se zde diskutují otázky bezpečnosti v heterogenních multiagentních systémech.

Pro bližší pojednání o multiagentních systémech a metodách jejich návrhu a realizací není bohužel v tomto textu prostor. Kvalitní shrnutí současného stavu v oblasti multiagentních systémů nabízí například kniha již dříve zmíněného Yoava Shohama [44].

2.5 Modelování distribuovaných systémů

Modelování distribuovaných systémů také hraje důležitou roli v návrhu systémů a řídících jazyků, které budeme prezentovat dále v tomto textu. Jelikož náš jazyk má v řadě ohledů společné rysy s procesními kalkuly, budeme procesním kalkulům věnovat následující podkapitolu. V ní ukážeme, jak takové kalkuly vypadají, a jakou mají procesy a jejich modely souvislost s výpočtem funkcí.

2.5.1 Kalkuly pro paralelní, mobilní a agentní systémy

π-kalkul, představený Robinem Milnerem koncem osmdesátých let minulého století [45], je jedním ze současných obecně uznávaných přístupů k modelování distribuovaných systémů s mobilními procesy, které komunikují předáváním zpráv. Mobilita procesů je v modelech tohoto kalkulu realizována předáváním jmen komunikačních kanálů mezi procesy. Po pojmenovaných komunikačních kanálech také probíhá synchronní komunikace mezi procesy v rámci distribuovaného systému

Tento kalkul byl inspirován do té doby známými metodami, jako byly například Hoareho komunikující sekvenční procesy z roku 1978, v pozdějším vydání [22]. Nejdřív ale začneme sekcí o λ-kalkulu, skrz který později ukážeme vztah mezi procesními kalkuly a kalkulem pro vyčíslení funkcí a tím i vztah mezi procesy a funkcemi.

2.5.2 λ-kalkul

λ-kalkul je od padesátých let minulého století, kdy byl představen Alonzo Churchem, významným představitelem výpočetních modelů parciálních rekurzivních funkcí. Je to také výpočetní systém s univerzální výpočetní silou. Ačkoli je λ-kalkul dobře známým systémem, představíme jeho základy i v tomto textu, protože později přes λ-kalkul ukážeme výpočetní sílu procesních kalkulů, které mají blízký vztah k systémům, na které bude zaměřeno jádro této práci.

Termy λ-kalkulu jsou definovány následovně.

$$M = k \mid \lambda x.M \mid M\,M'$$

Redukční, nebo také někdy nazývaná jako konverzní, pravidla jsou definovaná jako vztah mezi termy. Konkrétně jsou v λ-kalkulu k dispozici následující tři redukční pravidla.

α - konverze: lze provést pro zápis termu ve formě abstrakce $\lambda v.E$, kde v je proměnná vázaná v E touto abstrakcí. Redukci se provede substitucí $\lambda v'.E[v'/v]$, pokud proměnná v' není vázaná v E. Celé redukční pravidlo má tvar $\lambda v.E \Longrightarrow_{\alpha} \lambda v'.(E[v'/v])$

β – konverze: podstatná redukce pro nějakou abstrakci. Pravidlo je ve formě $(\lambda v.E)E' \Longrightarrow_{\beta} E[E'/v]$. Význam je takový, že vázaná proměnná je nahrazena ve všech svých výskytech za term, který je na abstrakci aplikovaný.

η – konverze: Pravidlo této konverze je ve tvaru $\lambda v.Ev = E$ a musí platit, že v není volnou proměnnou v termu E.

α - konverze slouží k přejmenování proměnných a bývá obvykle používána, pokud nějaká substituce není aplikovatelná kvůli konfliktu jmen vázaných proměnných. Druhá z konverzí, β – konverze, je důležitou konverzí a její použití odpovídá aplikaci argumentu na funkci. Poslední z konverzí, η – konverze, je také nazývána pravidlem extenzionality, a uvádí, že pokud dvě funkce pro stejné argumenty zobrazují stejný výsledek, lze je syntakticky převádět jednu na druhou.

Část termu, na kterou lze uplatnit redukční pravidlo, nazýváme redex. Pojem redex lze také rozlišit přívlastkem podle toho, o jaké pravidlo se jedná. Pak v termech máme α - redexy, β -redexy, případně η -redexy. Pokud je ale použit pojem redex bez přívlastku, bývá pojmem redex označován β -redex.

Redukční strategie dávají řád v jinak nedeterministickém postupu výpočtu. Výpočtem je myšleno hledání normálního tvaru termu, to je takového tvaru, kterého lze dosáhnout redukcemi, ale který nelze dále redukovat. Mezi základní redukční strategie patří volání jménem a volání hodnotou.

2.5.2.1 Volání jménem

Jedna z redukčních strategií, která je často používaná v λ-kalkulu, ale i ve výpočetních systémech, které z něj vycházejí, je strategie volání jménem. Při této strategii je vždy vyhodnocován nejlevější redex. Pravidla pro aplikaci redukčních pravidel jsou pro tuto redukční strategii následující.

$$(n1)\quad \frac{}{(\lambda x.M)N \longrightarrow_N M\left\{\frac{N}{x}\right\}} \qquad (n2)\quad \frac{M \longrightarrow_N M'}{MN \longrightarrow_N M'N}$$

Pokud lze na nejlevější abstrakci aplikovat nějaký term, aplikujeme jej. Term po provedení aplikace může být dále redukovatelný. Pravidlo $\mathrm{n1}$ definuje vztah mezi abstrakcí před aplikací termu N a po jeho aplikování. Pravidlo $\mathrm{n2}$ definuje to, že vždy je zpracováván nejlevější redex.

2.5.2.2 Volání hodnotou

Na rozdíl od volání jménem, nebo také líného vyhodnocování, je strategie volání hodnotou založena na tom, že aplikace β-konverze je prováděna jen tehdy, když argumentem je hodnota. To znamená,

že případné abstrakce musí být v aplikovaném termu vyhodnoceny před tím, než je term použit jako argument. V následujících zápisech je term V ve svém normálním tvaru, to znamená, že je dále neredukovatelný, a takový term považujeme tedy za aplikovatelnou hodnotu.

$$(v1)\ \frac{N \longrightarrow_V N'}{VN \longrightarrow_V VN'} \quad (v2)\ \frac{M \longrightarrow_V M'}{MN \longrightarrow_V M'N} \quad (v3)\ \frac{}{(\lambda x.M)V \longrightarrow_V M\{V/x\}}$$

Pravidlo v1 umožňuje provádět redukce i na části termu, které vlevo obsahují neredukovatelné části. Pravidlem v2 provádíme redukce u redukovatelných částí a pravidlo v3 umožňuje provést aplikaci hodnoty V na abstrakci.

Nezávisle na zvolené redukční strategii je výsledná normální forma stejná. Ovšem může nastat situace, kdy je posloupnost redukcí nekonečná a zvolenou redukční strategií se nelze k normální formě dobrat. Také volba redukční strategie může ovlivnit paměťové a výpočetní nároky na systém, který tyto redukce interpretuje.

2.5.3 π -kalkul

Od doby svého publikování počátkem devadesátých let [45], π-kalkul dodnes představuje nejvýznačnějšího reprezentanta kalkulů pro paralelní a mobilní procesy. Dá se také říct, že se jedná o dobře prozkoumaný a ověřený formalismus. V současnosti existuje řada variant a rozšíření tohoto kalkulu, mezi které patří polyadické kalkuly, asynchronní π-kalkuly, kalkuly s prostředím, nebo agentní kalkuly. Některé z nich si za chvíli stručně představíme. Nyní uvedeme formální základy π-kalkul, jeho jazyk a redukční pravidla.

π-kalkul vychází z množiny jmen, která v systému zastupují jak proměnné, tak i konstanty a komunikační kanály. Obvykle procesy symbolicky reprezentujeme velkými písmeny (P,Q…), nebo kompozicí předpony a symbolu procesu. Syntaxe π-kalkulu je definována následovně.

Předpona π je v jednom z následujících tvarů $\pi = \bar{x}y \mid x(z) \mid \tau \mid [x=y]$

Proces P je pak zapsán jako

$$P ::= M \mid P|P' \mid \vartheta z P \mid !\,P$$

$$M ::= 0 \mid \pi.P \mid M + M'$$

Strukturální kongruence lze chápat jako relaci mezi procesy, které jsou stejné, jen zapsány rozdílným způsobem. Pro dva procesy, pro které by platila tato relace, platí i to, že jeden může být transformován na druhý za použití následujících pravidel.

$M + 0 \equiv M$	$\mathrm{M}_1 + \mathrm{M}_2 \equiv \mathrm{M}_2 + \mathrm{M}_1$	$\mathrm{M}_1 + (\mathrm{M}_2 + \mathrm{M}_3) \equiv (\mathrm{M}_1 + \mathrm{M}_2 + \mathrm{M}_3$
$P\|0 \equiv P$	$P_1\|P_2 \equiv P_2\|P_1$	$(P_1\|P_2)\|P_3 \equiv P_1\|(P_2\|P_3)$
$!\,P \equiv P\|!\,P$	$(\vartheta a)0 \equiv 0$	$(\vartheta a)(\vartheta b)P \equiv (\vartheta b)(\vartheta a)P$
$[\mathrm{x} = \mathrm{x}]\ \pi.\mathrm{P} \equiv \pi.\mathrm{P}$	$\mathrm{P}_1\|(\vartheta a)P_2\ pokud\ a \in fn(P_1)$	

Redukční pravidla π-kalkulu jsou tvořena s ohledem na možné použitelné akce v jednotlivých procesech. Jsou značena jako $\xrightarrow{\alpha}$ pro nějakou akci α. Základní π-kalkul definuje redukční pravidla pro komunikaci a to pro příjem jména po komunikačním kanálu $\xrightarrow{\bar{p}x}$, poslání jména po komunikačním

kanálu $\xrightarrow{(p)x}$, dale vázání jména v rámci podprocesu $\xrightarrow{(\nu x)}$ a pro tichou akci, která není pozorovatelná vně procesu $\xrightarrow{\tau}$.

Redukční pravidla se mohou v jednotlivých textech lišit. Představíme si ta, která se vyskytují obvykle.

$$(INTER)\ \frac{}{\bar{x}\langle z\rangle.P|x(y).Q \rightarrow P|Q\left[\frac{z}{y}\right]}\quad (R-PAR)\ \frac{P \rightarrow Q}{P|R \rightarrow Q|R}$$

$$(R-RES)\ \frac{P \rightarrow Q}{(\vartheta x)P \rightarrow (\vartheta x)Q}\quad (R-STRUCT)\ \frac{P \equiv P' \rightarrow Q \equiv Q'}{P \rightarrow Q'}$$

$$(RESTRUCT)\ \frac{}{\tau.P + M \rightarrow P}$$

2.5.4 Procesy jako funkce: kódování λ–kalkulu v π-kalkulu

Jelikož λ–kalkul je dobře známý Churchův systém s obecnou výpočetní silou, vztah mezi tímto kalkulem a π-kalkulem vede k prokázání výpočetní síly π-kalkulu. Vztah mezi procesním a lambda kalkulem byl Milnerem zkoumán již v době prvního uvedení π-kalkulu a převod mezi nimi byl učiněn jak pro λ–kalkul se strategií volání jménem, tak i se strategii volání hodnotou [46]. Převod mezi λ–kalkulem a procesním π-kalkulem je proveden transformací λ termu na tvar CSP, což je forma zápisu termu pro provádění výpočtu s kontinuem [47]. Kontinuum je zbytek výpočtu po provedení nějaké operace, do kterého se předává aktuální mezivýsledek. Z CSP pak lze provést převod mezi λ termem a procesem π-kalkulu, ve kterém bude kontinuum nějaký další proces, který přijímá po komunikačním kanále aktuální stav výpočtu po provedení kroku výpočtu v předchozím procesu.

Nejprve je tedy třeba přeložit jazyk λ–kalkulu do odpovídajícího tvaru pro CSP. Podle toho, jestli výsledkem má být proces π-kalkulu provádějící redukční strategii výpočtu voláním jménem či volání hodnotou, budou se lišit i překlady do CSP. Pro strategii volání hodnotou budou hodnoty *V* (konstanta, abstrakce) překládány funkcí $C_V[V]$, zkráceně $[V]$, a termy λ-kalkulu překládány funkcí $C_V[\![V]\!]$, zkráceně $[\![V]\!]$. Tyto funkce jsou definovány následovně.

$$[x] = x,\ \ [\lambda x.M] = \lambda x.[\![M]\!]$$

$$[\![x]\!] = \lambda k.k\ x, \qquad [\![\lambda x.M]\!] = \lambda k.k\ \lambda x.[\![M]\!],\ \ [\![MN]\!] = \lambda k.[\![M]\!](\lambda v.[\![N]\!](\lambda w.vwk))$$

Proměnná ‘k’ je vyhrazena pro kontinuum funkce, na které je aplikován výsledek daného kroku výpočtu. V případě proměnné je aplikace kontinua jednoduše abstrakcí transformována na aplikaci proměnné na toto kontinuum. Stejně tak pro abstrakci, u které ale je třeba nejprve přeložit její tělo. U aplikace je zápis trochu složitější, ale i tak lze na něm vidět, že po vyhodnocení termu *M* je výsledek navázán na proměnnou *v* a stejně tak po vyhodnocení termu *N* je výsledek navázán na proměnnou *w*. Tímto term přejde do tvaru aplikace $M\ N$ a výsledek této aplikace je aplikován na kontinuum *k*.

CSP tvar umožní převést zápis funkcí do ekvivalentního zápisu procesů. Pro jednotlivé konstrukce *M* představující konstantu, abstrakci, nebo aplikaci, je překladová funkce značená $\Upsilon[\![M]\!]$ a je definována následovně.

$$\Upsilon[\![x]\!](p) = \bar{p}(x)$$

$$\Upsilon[\![\lambda x.M]\!](p) = \bar{p}(y).!\,y(x,q)\Upsilon[\![M]\!](q)$$

$$\Upsilon[\![MN]\!](p) = \upsilon q(\Upsilon[\![N]\!]q \,|\, q(v).\upsilon r(\Upsilon[\![M]\!]r \,|\, r(w).\bar{w}\langle v,p\rangle))$$

Procesy komunikuj po kanále uvedeném ve dvojitých závorkách. Konstantě s uvedeným kanálem odpovídá proces, který pouze odešle tuto konstantu po zadaném komunikačním kanále. Proces pro abstrakci funguje tak, že nejprve odešle po komunikačním kanále jméno privátního kanálu pro danou abstrakci a pak po tomto kanále přijímá hodnotu, která je substituovaná v procesu pro výrazy v *M*. Výsledek vyhodnocení *M* je nakonec odeslán po původním komunikačním kanále.

Kódování aplikace jako procesu je na pohled nejsložitější, ale zároveň snadno pochopitelné. Nejprve je vyhodnocen výraz *N* a výsledek je odeslán po privátním kanále *q*. Pro další postup předpokládejme, že hodnota je aplikovaná na abstrakci, což je oprávněný předpoklad, jelikož jen taková aplikace se β redukcí v λ-kalkulu vyhodnocuje. Pak je abstrakce provedena odesláním svého privátního kanálu po *r*. Po tomto privátním kanále jsou předány hodnoty pro aplikaci a jméno kanálu, po kterém má být odeslán výsledek aplikace, čímž je původní kanál pro celý výraz původní aplikace.

Obdobně je možné nalézt zápisy procesů pro λ-výrazy s redukční strategií volání jménem. Jak je nalézt je uvedeno například v [46].

2.5.5 Varianty π -kalkulu

Od publikování původního π -kalkulu do dnešních dnů byla vytvořena řada dalších kalkulů, které vychází z původního π -kalkulu a rozšiřují jej. Polyadický π –kalkul zavádí možnost pracovat s datovými strukturami. Dalšími odvozenými kalkuly jsou kalkuly zahrnující prostředí, kalkuly pro klient-server aplikace, nebo v i kalkuly pro agentní systémy. Mezi procesní kalkuly se řadí také například PSI-kalkul [48, 49] využívaný v oblasti kryptografie a další [50, 51]. Některé z nich si nyní stručně představíme.

2.5.5.1 Polyadický π-kalkul

Původní π-kalkul pracoval s množinou jmen pro identifikaci komunikačních kanálů, ale zároveň i pro veškerá další data, se kterými procesy pracují. Polyadický π-kalkul rozšiřuje možnosti používat jako data struktury, což znamená nějak uspořádanou množinu více jmen. Takové struktury se pak mohou používat jako argumenty pro operace v procesech. Byl vytvořen Milnerem nedlouho po publikování původního π-kalkulu [52]. V tomto textu je představen způsob realizace datových struktur v kalkulu přes abstrakce a konkretizace pro přijímání a odesílaní datových struktur.

2.5.5.2 Asynchronní π-kalkul

Přestože π-kalkul je definován jako model se synchronní komunikací, je v něm možné asynchronní komunikaci simulovat [53]. Proces může replikovat své částí, které přijímají zprávy, a může být zapsán tak, aby proces obsahoval paralelní konstrukci s částí odeslání zprávy a jinou částí, která má zprávy přijímat. Zpráva je odeslána z první paralelní části, až když v jiné části bude akce přijetí zprávy po kanále, který se shoduje s kanálem odeslání v první paralelní části procesu.

2.5.5.3 Kalkuly s prostředím

Ambientní kalkuly, nebo kalkuly s prostředím [54], přináší možnost zapisovat procesy v rámci nějaké skupiny procesů, do které během fungování systému mohou jednotlivé procesy vstupovat a ze které mohou vystupovat. Komunikace v těchto systémech může probíhat jen v rámci jednoho prostředí, které vymezuje část systému, jako třeba lokální síť, agentní platformu a podobně. Možné akce, které

mohou být v procesu zapsány, obsahují oproti klasickému π-kalkulu akce pro vstoupení do nějakého prostředí, nebo vystoupení z prostředí, ve kterém se proces aktuálně nachází.

2.5.5.4 Agentní kalkul

Některé agentní rysy přináší do procesních kalkulů API-kalkul [55, 56]. Oproti kalkulům s prostředím, které umožňuje procesům seskupovat se v jednom prostoru, přidává možnost pracovat v rámci procesů s jejich bázemi představ. Práce s bází představ znamená možnost přidávání a odebírání faktů do a z báze, provádění procesů uložených v bázi a zasílání a přijímání obsahu bází mezi agenty. Tyto akce rozšiřují množinu akcí z předchozích kalkulů.

Kalkuly představené v této kapitole jsou jedním ze základů jazyka pro programování malých zařízení, jehož návrh patří mezi cíle této práce. V něm budeme také řešit mobilitu kódu, sdružování procesů v nějakých prostředích a práci agenta se svou bází dat. Nově v něm budeme tvořit i strukturované procesy, které se vyskytují například v systémech řízených záměrem, a které budou rozšířením možnosti zápisů procesů oproti současným přístupům uvedeným výše.

2.6 Petriho sítě

Druhým způsobem modelování distribuovaných systémů jsou Petriho sítě, které v roce 1962 Carl Adam Petri představil ve své disertaci [57]. Model systému je zde budován jako propojení míst a přechodů, přičemž místa udržují značení udávající stav systému a přechody mohou transformovat tento stav, pokud jsou splněny podmínky jejich provedení. Tyto sítě se během let ukázaly jako vhodný nástroj pro modelování paralelních systémů a pro svoji jednoduchost, dobré formální založení a vyjadřovací schopnosti patří dodnes vedle procesních kalkulů mezi nejpopulárnější modely paralelismu. Lze nalézt i polemiky o vhodnosti či nevhodnosti jednoho či druhého přístupu, to znamená procesních kalkulů či Petriho sítí, pro modelování různých typů paralelních systémů [58].

Pro modelování agentních systémů a to jak na úrovni agentních systémů, tak na úrovních interakcí mezi agenty a organizačních úrovních, jsou obvykle využívány Petriho sítě vyšší úrovně. Mezi ty se řadí například barvené Petriho sítě [59], Objektově orientované Petriho sítě [60] a další.

2.6.1 Využití Petriho sítí pro modelování a realizaci multiagentních systémů

Aplikaci Petriho sítí do oblasti agentních a multiagentních systémů lze najít například v pracích Jacquese Febera [61], nebo Daniela Moldta [62]. První z autorů se více zaměřuje na modelování multiagentních systému na úrovních interakčních a organizačních. Moldt ukázal realizaci Shohamova Agent-0 systému Petriho sítěmi a jeho skupina se soustředí na využití Petriho sítí pro modelování procesů s agentními rysy. Jedním z výsledků jejich snažení je multiagentní simulační systém založený na Petriho sítích nazvaný Mulan [63].

2.6.2 PN-Agent

Originálním výsledkem, na kterém jsem se podílel jako spoluautor, je implementace BDI agentů v prostředí Objektově orientovaných Petriho sítí. Konkrétněji se jedná o jejich realizaci v rámci systému Objektově orientovaných sítí PNtalk [60], nazvanou PNagent. Koncept tohoto systému byl navržen v [64] a následně zpracována Zdeňkem Mazalem, který výsledný systém představil ve své disertaci [65]. Výsledek byl dále publikován v odborném časopise [66]. PNagent je postaven podle současných architektur BDI agentů, které jsme stručně popsali v kapitole 2.4.4. Jelikož Objektově orientované Petriho sítě umožňují vytvářet a modifikovat strukturu sítě za běhu systému, bylo možné realizovat

tohoto agenta podle principů, které jsou u realizací BDI systémů běžné. Chování agenta je řízeno na základě struktur záměrů, které jsou také modelovány Petriho sítěmi.

PN-Agent byl začleněn do systému SmallDevs [67], což je implementace systému DEVS v prostředí Smalltalk. V současnosti dochází k dalšímu rozšiřování tohoto systému směrem k zavedení agentních platforem podle standardů FIPA. To má umožnit začlenění systémů PNagent a SmallDevs do větších heterogenních multiagentních systémů realizovaných podle těchto standardů.

3 Teorie ALLL systémů

Teoretická kapitola o ALLL systémech tvoří jádro této práce. Je v ní popsáno několik ALLL systémů z hlediska jejich struktury a jazyka, kterým je řízeno jejich chování, a také jsou zde uvedeny způsoby interpretace těchto jazyků.

ALLL systémy vychází ze systému t-Sapi, který byl popsán v disertaci [1]. První implementací těchto systémů byl simulační nástroj T-Mass (Tool for Multiagent System Simulation) [68, 69], který simuloval chování populace t-Sapi agentů, ovšem bez možnosti jejich interakcí s nějakým podsystémem, který by reprezentoval prostředí, ve kterém se tito agenti nachází. Následné práce se týkaly jednak podrobnější specifikace ALLL systémů. Například byl představen přístup k metarozhodování o prováděných plánech v těchto systémech [70]. Následovaly jeho aplikace do oblastí simulací, tentokrát pro novou generaci systému T-Mass(X), a také pro malá zařízení, jakými jsou uzly bezdrátové senzorové sítě. Pro ty byl ALLL systém implementován jako část middleware řešení a byl nazván WSAgeNt [71, 72, 73].

3.1 Některé vlastnosti ALLL systémů

Na úvod probereme vlastnosti ALLL systémů z hlediska jejich struktury a dále je budeme zkoumat jako výpočetní systémy. Začneme rozborem struktur systémů, ve kterých se ALLL systémy mohou vyskytovat buď jako podsystémy, nebo je tvořit zcela. Dále si všimneme paradigmat, která by se měla objevit v návrhu ALLL výpočetního prvku. Poté se podíváme na ALLL prvek jako výpočetní systém a budeme diskutovat, jak bude probíhat výpočet tímto systémem, nebo z jiného úhlu pohledu jak je řízeno chování ALLL prvku nějakým kódem. To nám poskytne dobrý úvod pro následující formální kapitolu, která představí některé konkrétní realizace systémů s ALLL prvky.

3.1.1 Struktura ALLL systémů

Pokud budeme mluvit o ALLL systémech, můžeme mít na mysli systém, který tvoří ALLL výpočetní prvek. Ten provádí nějaký program, který v případě těchto systémů nazýváme plán. ALLL systémem také můžeme rozumět ALLL distribuovaný systém, který zahrnuje distribuované prvky, z nichž některé jsou ALLL agenty. Za ALLL agenty budeme v těchto systémech považovat takové ALLL výpočetní prvky, které jsou otevřenými systémy a mohou v rámci většího distribuovaného systému jednat, přijímat z něj podněty a ovlivňovat jej svými akcemi.

Vztah mezi ALLL výpočetním prvkem a okolím se tvoří přes jejich vzájemné vazby a to skrz agentovy sensory a efektory, které jsou z hlediska systému jeho vstupními a výstupními branami. Ve struktuře ALLL prvku najdeme báze, které mohou obsahovat data reprezentující agentovy představy, plány, nebo přijaté zprávy. Součástí stavu prvku bude i informace, která reprezentuje aktuální hodnotu výpočtu po posledním vykonaném výpočetním kroku. Vzhledem k obvykle používaným pojmům budeme této hodnotě rozumět jako stavu registru ALLL prvku. V teoretických systémech je registr jen jeden, ale v praktických realizacích jich kvůli programátorskému pohodlí používáme více.

3.1.2 ALLL výpočetní prvky z pohledu paradigmat programovacích jazyků

Jazyk, který bude navržen pro řízení ALLL agentů, kombinuje několik výpočetních paradigmat. Půjde jednak o agentní paradigmata, paradigmata kalkulů paralelních procesů a také jsme adoptovali některá paradigmata z funkcionálního přístupu k výstavbě jazyků.

Z hlediska agentních paradigmat je situace taková, že ne všechny požadavky na systém jako agenta jsou u ALLL agentů splněny, ale ve většině ALLL systém agentní rysy naplňuje. Nejprve bude dobré

uvést, proč může být problematické nazývat tyto prvky agenty. Jednou ze základních vlastností agentních prvků je jejich schopnost pohotově a adekvátně reagovat na změny prostředí. ALLL systémy sice mají ve svých realizacích takovou možnost, ta je ale dána schopností platforem spouštět konkrétní plány ALLL prvků, které tyto platformy spravují, na základě definovaných událostí. Reaktivita ale není zohledněna v návrhu vlastního ALLL systému, a proto v tomto ohledu těmto systémům chybí základní předpoklad pro to, abychom je mohli bezezbytku považovat za agenty jako takové. Na druhou stranu mají některé agentní rysy jako jsou struktury plánů, byť v nejjednodušších verzích ALLL systémů pouze lineární, které jsou vykonatelné, pokud jsou úspěšně vykonány jednotlivé akce v aktuálním stavu prostředí. Dále agentní rysy lze u ALLL agentů vidět v jejich mobilitě, nebo v deklarativním přístupu ke specifikaci úloh, které ALLL systémy vykonávají.

3.1.3 ALLL prvek jako výpočetní systém

Výpočet systémem ALLL prvku, který bude prezentován v této kapitole, bude prováděn v krocích, které budou měnit jeho vnitřní stav i stav celého systému, ve kterém se nachází. Stav ALLL prvku bude zahrnovat i mezivýsledek výpočtu, který může být předáván do jeho výpočetního kontinua. Výsledkem celého výpočtu pak bude stav výpočtu po provedení posledního výpočetního kroku ALLL prvku. Jeho chování můžeme modelovat funkcí.

$$c' = \chi(a, c)$$

Symbol c' označuje nový vnitřní stav výpočtu ALLL prvku po provedení akce a za vnitřního stavu výpočtu c. Vnitřní stav výpočtu budeme později u jednotlivých formálních ALLL prvků také nazývat výpočetním kontextem nebo konfigurací výpočtu prvku. V rámci této kapitoly budeme definovat funkci chování ALLL prvků pro všechny akce, které můžou být zapsány v plánech, které jsou prvkem prováděny.

V distribuovaných systémech ALLL prvek pracuje v prostředí jako část výpočetního systému, u něhož stav výpočtu reprezentuje aktuální vnitřní stavy všech prvků, které obsahuje. Jednotlivé prvky ovlivňují další běh celého systému svými výstupy a mohou být samy ovlivňovány hodnotami na svých vstupech. ALLL prvek je také systém pracující na základě vjemů a ovlivňující okolí svými akcemi. Vnitřně je ALL prvek navržen jako systém tak, aby byl jednoduchý na implementaci a plán, který zpracovává, byl jednoduchý na interpretaci a přitom zahrnoval podstatné rysy současných agentních systémů.

3.1.4 Struktura kapitoly

Struktura ALLL prvků zahrnuje bází nebo báze ve smyslu bází dat, se kterými agent pracuje. Tato kapitola formálně představí jak tyto báze, tak ALLL prvky jako systémy, a nakonec i distribuované systémy, ve kterých se ALLL prvky vyskytují jako výpočetní podsystémy. Začneme daty a operacemi, které se v systému bází vyskytují. Poté budeme definovat přechodové a výstupní funkce těchto systémů a to systémů bází obsahující jednak data bez uspořádání a jednak s uspořádáním. Po představení systémů bází přejdeme k systémům, u kterých je chování řízeno nějakými plány jako lineárními posloupnostmi akcí. Poté přejdeme k složitějším plánům, které budou nazývány hierarchické plány a budou to strukturované prvky tvořené akcemi a podplány. Ukážeme princip intepretace takového plánu pro uzavřený ALLL prvek, který neinteraguje s okolím. Dále bude následovat otevřený ALLL prvek, který dokáže pracovat s daty na svých vstupních a výstupních branách. Poslední částí této teoretické kapitoly se bude věnovat ALLL systémům, ve kterých budou ALLL prvky pracovat paralelně a budou spolu komunikovat předáváním zpráv. Takové systémy nám

dají možnost ukázat principy jednání multiagentní skupiny a přejít k další kapitole, která bude věnována konkrétním realizacím ALLL agentů a multiagentních systémů s těmito agenty.

3.2 Datové struktury ALLL systému

Během provádění systémů přestavovaných v této práci budeme obecný prvek považovat za základní výpočetní konstrukci a budeme jej používat podobně, jak bývá například používán obecný seznam v jazyce LISP, nebo n-tice v nástěnkových systémech LINDA. ALLL systém navíc využívá postupy z obou těchto systémů, to znamená jak rekurzivní zpracování seznamů, tak i ukládání, dotazování a unifikace do prostoru n-tic.

3.2.1 Obecný prvek ALLL systémů

Nejprve zavedeme pojmy prvek, lineární prvek a obecný prvek ALLL systémů a zavedeme množiny těchto prvků různých délek a úrovní zanoření. Atomy a vyhrazené symboly ALLL systémů budeme uvádět v apostrofech a řetězce nad těmito prvky budeme uzavírat do uvozovek.

Definice 1: Množina atomů ALLL systému je značena Σ a obsahuje všechny atomy daného ALLL systému a neobsahuje znaky uvedené v apostrofech, ‘(’ , ‘) ‘ , ‘~ ‘, ‘⊥‘ $\notin \Sigma$.

Vyhrazené atomy, které podle definice 1 nemohou být součástí množiny atomů, budou použity ke konstrukci základních datových struktur a plánů ALLL systémů v podobě obecných seznamů uzavřených v kulatých závorkách. V dalším textu mohou být atomy i řetězce, které ale budeme chápat jako nedělitelné atomy a budou použity pro lepší názornost. Nebudeme se zabývat zpracováním řetězců a možnosti skládání jednoho atomu jako spojení druhého (například ‘to’ ‘mas’ a ‘tomas’), což by později komplikovalo vyhledávání prvků v bázi. Pokud budeme mít takový řetězec, budeme předpokládat, že je zpracovatelný jako jeden symbol.

Jelikož seznamy mohou obsahovat zanořené seznamy, musí být jednotlivé podstruktury odděleny a právě použití závorek nám toto umožní. Prvky vyšších úrovní mohou jako atomy používat i prvky na nižších úrovních, což je sice v rozporu s chápáním pojmu atom, ale my tento název budeme i tak používat, jelikož se bude jednat o základní stavební kameny oněch prvků na vyšších úrovních.

Následující definice zavede lineární prvek ALLL systému ve formě posloupnosti atomů uzavřené do závorek. Následně budou definovány prvky jako seznamy s hierarchickou strukturou.

Definice 2: Doména $D^n(\Sigma)$ lineárních prvků nad množinou atomů Σ délky $n \geq 1$ je množina všech uspořádaných posloupností s opakováním délky n (n-tic) nad množinou atomů Σ uzavřená do závorek.

$$D^n(\Sigma) = \{'(' w ')' | w \in \Sigma^n\}$$

Doména $D^n(\Sigma)$ je množina a její zápis má tvar funkce s argumentem vymezujícím množinu atomů, nad kterými se doména vytváří. V tomto textu si dovolíme takovýto zápis pro reprezentaci množin, abychom odlišili od sebe množiny právě podle prvků, ze kterých jsou konstruovány.

Definice 3: Doména $D(\Sigma)$ lineárních prvků nad množinou atomů Σ je definována následovně.

$$D(\Sigma) = \bigcup_{n \geq 1} D^n(\Sigma)$$

Následující definice zavedeme pojem obecný prvek.

Definice 4: Doména obecných prvků i-té úrovně délky n je značená jako $\Delta^{i,n}$ a je definována takto.

$$\Delta^{1,n} = D^n(\Sigma)$$

$$\Delta^{i,n}(\Sigma) = D(\Delta^{i-1}(\Sigma) \cup \Sigma)^n \; pro \; i \geqq 1$$

V definici 4 byl použit zápis obecného prvku i-té úrovně, jenž si zavedeme nyní.

Definice 5: Doména obecných prvků i-té úrovně obecné délky je značená jako Δ^i a je definována jako $\Delta^i(\Sigma) = \bigcup_{n \geq 1} \Delta^{i,n}(\Sigma)$.

Poznamenáváme, že doménu obecných prvků by šlo také definovat podobně jako v definici 3 takto.

$$\Delta^1 = D(\Sigma) \text{ a } \Delta^i(\Sigma) = D(\Delta^{i-1}(\Sigma) \cup \Sigma) \; pro \; i \geqq 2.$$

Definice 6: Doména obecných prvků je značená jako $\Delta(\Sigma)$ a je definována jako

$$\Delta(\Sigma) = \bigcup_{i \geqq 0} \Delta^i(\Sigma)$$

Definice pro obecné hierarchické prvky podle definice 4 by v případě, že by nebyly lineární prvky uzavřeny v závorkách mimo množinu atomických prvků, hierarchii nevytvářela, jelikož by výsledkem byla opět jen množina uspořádaných prvků s opakováním nad Σ.

Později u některých příkladů a dále také u jazyka ALLL postaveného na zde představovaných principech budou jednotlivé prvky n-tice oddělovány čárkou. Bude to zejména kvůli lepší čitelnosti datových struktury ALLL prvku. To se také nevylučuje s definicí 4, když čárka bude jedním z atomů a n-tice tak bude zahrnovat jak prvky nesoucí nějakou informaci, tak i čárky, které tyto prvky budou oddělovat.

Příklad 1: Pro množinu atomů $\Sigma = \{'a', 'b', 'c'\}$ jmohou být jednotlivé množiny například následovně.

$D^1(\Sigma) = \{"(a)", "(b)", "(c)"\}$,
$D^2(\Sigma) = \{"(aa)", "(ab)", "(ac)", "(ba)", "(bb)", "(bc)", "(ca)", "(cb)", "(cc)"\}$ …

$\Delta^1(\Sigma) = D(\Sigma) = \{"(a)", "(b)", "(c)", "(aa)", "(ab)" \ldots "(abca)", "(abcb)" \ldots\}$.

$\Delta^2(\Sigma) = \Delta^1(D(\Sigma) \cup \Sigma) = \{(a), (b), (c), \big((a)\big), \big((b)\big) \ldots "\big((abcb)\big)" \ldots$

$$\ldots "(aa)" \ldots "(cc)", "\big(a(a)\big)", "\big(a(b)\big)" \ldots "\big(a(cc)\big)", "\big(b(a)\big)" \ldots "(a(a)a)" \ldots\}$$

Množina obecných prvků pak je

$\Delta(\Sigma) = \{"(a)", "(b)", "(c)", "\big((a)a\big)", "\big((a)b\big)" \ldots "\Big(\big(a(bc)\big)a\Big)", "\Big(\big(a(bc)\big)b\Big)" \ldots\}$

O prvku i-té úrovně hovoříme, pokud obsahuje alespoň jeden prvek, který je i-1ní úrovně a neobsahuje prvek, který je vyšší úrovně. Pokud seznam obsahuje pouze atomy, nazýváme jej lineárním prvkem, nebo prvkem první úrovně.

Následující Lemmata uvedou vztah mezi obecnými prvky a množinami obecných prvků nějaké úrovně.

Lemma 1: Množina $\Delta^i(\Sigma)$ obsahuje všechny obecné prvky nad abecedou Σ do i-té úrovně zanoření.

Důkaz: Pro první úroveň platí, že všechny prvky n-tice jsou atomy. Pak podle definic 2 a 3 jsou všechny tyto prvky z množiny $D(\Sigma)$ a podle definic 4 a 5 i z množiny $\Delta^1(\Sigma)$. Dále pokud jsou všechny prvky (i-1)-ní úrovně z množiny $\Delta^{i-1}(\Sigma)$, pak musí být všechny prvky i-té úrovně v $\Delta^i(\Sigma)$. Ukážeme to indukcí. Pokud je prvek $\delta = '('\ x\ y\ z\ ')'$ prvkem délky n i-té úrovně, pak *x y z* jsou obecné prvky z $\Delta^{i-1}(\Sigma)$, nebo atomy, a podle definice 4 prvky z množiny, nad kterou se tvoří prvky i-té úrovně. Pak tedy i δ je podle definic 2,3 a 4 z $\Delta^i(\Sigma)$.

Ještě ukážeme, že doména obecných prvků i-té úrovně opravdu obsahuje všechny prvky podle následující Lemmy:

Lemma 2: Množina $\Delta(\Sigma)$ je tvořena všemi obecnými prvky nad množinou atomů Σ .

Důkaz: To, že obsahuje všechny obecné seznamy nad množinou atomů plyne z Lemmy 1 a z definice 6, vztahy mezi množinami obecných prvků a operacemi nad množinami.

Pro množiny obecných prvků nyní budeme sledovat některé jejich vzájemné vztahy a uvedeme několik operací nad nimi.

Lemma 3: Pro množiny atomů Σ, Σ', kde $\Sigma \subset \Sigma'$ platí, že $\Delta(\Sigma) \subset \Delta(\Sigma')$

Důkaz: Ukažme, že pokud $n \in \Delta(\Sigma)$, pak musí platit i $n \in \Delta(\Sigma')$. Podle Lemmy 2 je *n* obecný seznam nad množinou atomů Σ. Pokud by platilo $n \notin \Delta(\Sigma')$, pak $\Delta(\Sigma')$ neobsahuje obecný seznam nad množinou atomů Σ a jelikož $\Sigma \subset \Sigma'$ tak ani nad množinou atomů Σ', což je v rozporu s Lemmou 2. Opačně pro prvek "(a)", $'a' \in (\Sigma' - \Sigma)$ můžeme říct, že se nejedná o prvek nad množinou atomů Σ a tudíž nepatří do $\Delta(\Sigma)$, ale patří do $\Delta(\Sigma')$, jelikož $'a' \in \Sigma'$.

Definice 7: Operace spojení dvou obecných prvků je struktura $\cdot : \wp(\Delta(\Sigma)) \times \wp(\Delta(\Sigma)) \to \wp(\Delta(\Sigma))$ a je definována pro nějakou množinu atomů Σ tak, že platí

$$\forall n_1 n_2 \in \Delta(\Sigma)(n_1 = '('n_{11}n_{12} \dots n_{1i}')', n_2 = '('n_{21}n_{22} \dots n_{2j}')',$$

$$n_1 \cdot n_2 = '('n_{11}n_{12} \dots n_{1i}\ n_{21}\ n_{22} \dots n_{2j}')')$$

Lemma 4: Spojení obecných prvků i-té a j-té úrovně, vznikne obecný prvek i té úrovně, pokud i>=j, nebo j-té úrovně, pokud i<j.

Důkaz: Spojené obecné prvky obsahují jako své členy prvky z množiny vzniklé sjednocením množin atomů obou spojovaných obecných prvků. Prvek s nejvyšším zanořením udávající stupeň zanoření spojeného seznamu je prvek s nejvyšším zanořením z obou původních seznamů.

Lemma 5: Pro konstruktory množin obecných prvků platí vztah $\Delta^{a+b}(\Sigma) = \Delta^a\left(\Sigma \cup \Delta^b(\Sigma)\right)$

Důkaz: Pokud máme $\Delta^m(\Sigma \cup \Delta^n(\Sigma))$, a označíme $\Sigma' = \Sigma \cup \Delta^n(\Sigma)$ pak $\Delta^m(\Sigma') = \Delta^{m-1}\left(\Sigma' \cup \Delta(\Sigma')\right)$ a po nahrazení za původní výraz $\Delta^m(\Sigma \cup \Delta^n(\Sigma)) = \Delta^{m-1}\left(\Sigma \cup \Delta^n(\Sigma) \cup \Delta(\Sigma \cup \Delta^n(\Sigma))\right) = \Delta^{m-1}(\Sigma \cup \Delta^n(\Sigma) \cup \Delta^{n+1}(\Sigma)) = \Delta^{m-1}(\Sigma \cup \Delta^{n+1}(\Sigma))$. Obdobně dále $\Delta^{m-2}(\Sigma \cup \Delta^{n+2}(\Sigma))$ atd. Obecně po a

krocích $\Delta^{m}(\Sigma \cup \Delta^{n}(\Sigma)) = \Delta^{m-a}(\Sigma \cup \Delta^{n+a}(\Sigma))$. Pokud $m=a+b$ pak $\Delta^{a+b}(\Sigma) = \Delta^{m}(\Sigma) = \Delta^{m-1}(\Sigma \cup \Delta^{1}(\Sigma))$ a po b-1 krocích předchozího výpočtu $\Delta^{m-1-b+1}\left(\Sigma \cup \Delta^{1+b-1}(\Sigma)\right) = \Delta^{m-b}\left(\Sigma \cup \Delta^{b}(\Sigma)\right) = \Delta^{a}\left(\Sigma \cup \Delta^{b}(\Sigma)\right) = \Delta^{a+b}(\Sigma)$.

Lemma 6: Pro každou neprázdnou množinu atomů Σ platí mezi úrovněmi prvků vztahy $\ldots \subset \Delta^{i}(\Sigma) \subset \Delta^{i-1}(\Sigma) \subset \cdots \Delta^{1}(\Sigma) \subset \Delta^{0}(\Sigma) = D(\Sigma)$

Důkaz: I když je věta zřejmá, uvedeme pro ni důkaz. Vztah $\Delta^{0}(\Sigma) = D(\Sigma)$ platí z definice 4. Pro vztah množin $\Delta^{i}(\Sigma) \subset \Delta^{i-1}(\Sigma)$ nejprve ukážeme, (a) že neexistuje prvek z $\Delta^{i-1}(\Sigma)$, který by nebyl $\Delta^{i}(\Sigma)$. Dále ukážeme (b) , že existuje prvek z $\Delta^{i}(\Sigma)$, který není v $\Delta^{i-1}(\Sigma)$.

(a) Z Lemmy 5 plyne, že platí $\Delta^{2}(\Sigma) = \Delta^{1}(\Sigma \cup \Delta^{1}(\Sigma))$. Jelikož musí platit vztah $\Sigma \subseteq (\Sigma \cup \Delta^{1}(\Sigma))$, pak podle Lemmy 3 platí i $\Delta^{1}(\Sigma) \subseteq \Delta^{1}(\Sigma \cup \Delta^{1}(\Sigma))$ a tedy $\Delta^{1}(\Sigma) \subseteq \Delta^{2}(\Sigma)$. Obdobně pro $\Delta^{3}(\Sigma) = \Delta^{2}(\Sigma \cup \Delta^{1}(\Sigma)) = \Delta^{1}\left(\Sigma \cup \Delta^{1}(\Sigma \cup \Delta^{1}(\Sigma))\right)$ a platí vztah $(\Sigma \cup \Delta^{1}(\Sigma)) \subseteq \left(\Sigma \cup \Delta^{1}(\Sigma \cup \Delta^{1}(\Sigma))\right)$, pak i $\Delta^{2}(\Sigma) \subseteq \Delta^{3}(\Sigma)$ atd.

(b) Nechť platí, že $\Delta^{i}(\Sigma) = \Delta^{i-1}(\Sigma)$ pro nějaké *i* a má platit $p \in \Delta^{i}(\Sigma) \rightarrow p \in \Delta^{i-1}(\Sigma)$. Pro $p' = "\left((..(a)\ldots)\right)"$ i-té úrovně zanoření, kde a je nějaký atom z Σ, platí podle Lemmy 1, že $p' \in \Delta^{i}(\Sigma)$ a zároveň $p' \notin \Delta^{i-1}(\Sigma)$, jelikož množina atomů Σ není zanořená a s každou novou úrovní vzrůstá maximální možná úroveň zanoření o jedna. Tento vztah je v rozporu s původním předpokladem.

Výše uvedená tvrzení jsme uvedli pro přehlednost, abychom dále mohli zacházet s těmito strukturami s vědomím, že obsahují všechny možné obecné prvky, abychom znali konstrukci spojení obecných prvků a také znali některé vztahy mezi množinami obecných prvků.

Nyní budou následovat systémy, které výše uvedené prvky používají pro svoji činnost. Začneme bázemi, které budou později využity u ALLL výpočetních prvků.

3.3 Systém ALLL báze bez uspořádání

Systémy báze tvoří datovou základnu pro ALLL výpočetní prvky a vnitřní stav bází je součástí konfigurací a výpočetních kontextů ALLL prvků. Báze slouží jednak k reprezentaci znalostí či představ agenta, k uchovávání plánů, které agent může využít k řízení své činnosti, a také fungují jako vstupní vyrovnávací paměti pro asynchronní komunikaci. Následující kapitoly představí dva typy bází, a to bázi bez uspořádání pro data a plány a bázi s uspořádáním pro příchozí zprávy.

3.3.1 Struktura systému ALLL báze bez uspořádání

Nejprve budeme definovat systém ALLL báze bez uspořádání (dále jen bázi, nebo bázi bez uspořádání) a operace nad ní. Báze je stavový systém, který pracuje nad množinou obecných prvků, n-tic, z množiny atomů. Báze přijímá akce z množiny akcí báze a provádí přechod mezi jednotlivými stavy na základě své přechodové funkce.

Definice 8: Systém báze bez uspořádání β_L je pětice $\beta_L = (\Sigma, \Gamma, \gamma_0, \ltimes_{L,I}, \ltimes_{L,O})$, kde Σ je množina atomů báze, Γ je doména báze, $\Gamma \subseteq \wp(\Delta(\Sigma)) \cup {}' \perp {}', {}' \perp {}' \in \Gamma$, $' \perp '$ je symbol neúspěšné operace, γ je stav báze, $\gamma \in \Gamma$, $\ltimes_{L,I}$ je vnitřní přechodová funkce báze a $\ltimes_{L,O}$ je výstupní funkce báze.

Množinu atomů již známe z předchozích kapitol. Doména báze obsahuje všechny možné stavy systému báze. Stavem je množina obecných prvků nad množinou atomů Σ, a proto doména obsahuje všechny takové množiny, které pro daný systém jsou přípustné jako stav systému. Můžou to být všechny možné množiny obecných prvků nad atomy, nebo jen některé z nich. Proto také je doména definovaná jako podmnožina potenční množiny množiny obecných prvků nad Σ. Vnitřní přechodová a výstupní funkce báze realizují chování systému a prezentaci jeho vnitřního stavu navenek.

Lemma 7: Doména báze není uzavřena vůči operaci sjednocení.

Důkaz: Podmnožina potenční množiny nemusí být uzavřená proti množinovým operacím.

3.3.2 Operace nad ALLL bází bez uspořádání

Následující odstavce se zaměřují na definice vnitřní přechodovou a výstupní funkci systému báze bez uspořádání. Tyto funkce vychází z operací, které lze na bázi aplikovat. Jedná se konkrétně o operace vkládání, odstranění a výběru prvku z báze. Jednotlivé operace reprezentují symboly operací $'+', '-', '\sigma'$ společně s nějakým argumentem ve formě obecného prvku.

Nejprve představíme řetězce, kterými budeme operace zapisovat. Symboly operací mohou, ale nemusí být součástí množiny atomů. Argumenty jsou obvykle obecnými prvky nad množinou atomů, ale v některých případech mohou obsahovat i jiné atomy mimo množinu atomů, například právě symboly operací. Pokud jsou symboly operací a i všechny atomy v argumentech součástí množiny atomů, mohou být zápisy operací uloženy v bázi. To ale budeme diskutovat až později.

Nyní budeme jednotlivé operace definovat. K tomu, abychom mohli definovat tyto operace, je třeba představit anonymní a specifikující prvky a jejich role v zápisech operací. V následujících definicích je anonymní prvek zapisován jako symbol $'\sim'$. Anonymní prvek bude hrát v ALLL systémech podobnou roli, jakou hraje anonymní proměnná v jazyce PROLOG, tedy že bude zastupovat jakýkoli (pod)prvek na daném místě zápisu prvku.

Definice 9: Specifikující prvek je obecný prvek nad množinou atomů, která obsahuje i anonymní prvek ' ~ ', nebo je to přímo anonymní prvek. To znamená, že pro nějakou množinu atomů Σ je specifikující prvek z množiny $\Delta(\Sigma \cup \{'\sim'\}) \cup \{'\sim'\}$.

Díky specifikujícím prvkům můžeme definovat relaci specifikace, která je důležitá pro následné operace odstranění a výběru prvku z báze. Relace specifikace je vztah mezi obecnými a specifikujícími prvky definovaný následovně.

Definice 10: Relace specifikace je struktura $\prec \subseteq \Delta(\Sigma) \times \Delta(\Sigma \cup \{'\sim'\}) \cup '\sim'$ definovaná jako

$\forall n \in \Delta(\Sigma) \cdot (n \prec '\sim')$

$\forall n_1 \in \Delta(\Sigma), n_2 \in \Delta(\Sigma \cup '\sim') \exists i \cdot$

$$(n_1 \prec n_2 \leftrightarrow n_1 = (a_1, a_2 \dots a_i) \wedge n_2 = (b_1, b_2 \dots b_i) \wedge \forall j \cdot$$

$$(0 \leq j \leq i \wedge (a_j \prec b_j) \vee (a_j = b_j)))$$

Tato relace umožní výběr podmnožiny z domény báze s použitím symbolu anonymního prvku. V relaci s nějakým specifikujícím prvkem budou takové prvky, které mají stejné prvky na všech místech kromě místa, na kterém je ve specifikujícím prvku anonymní prvek. Na místě anonymního

prvku může být v obecném prvku jakýkoli obecný podprvek z $\Delta(\Sigma)$. Zároveň také platí, že jeden obecný prvek je specifikací druhého, pokud druhým prvkem je anonymní prvek.

Symbolem $\prec$ budeme značit negaci pro relaci specifikace, tj. relace 'není specifikací', která je doplňkem k relaci specifikace. To znamená, že platí $\prec = \Delta(\Sigma) \times \Delta(\Sigma \cup \{'\sim'\}) \cup '\sim' - \prec$ pro nějakou množinu atomů Σ.

Pro nějakou množinu atomů $\sum$ je množina specifikujících prvků nadmnožinou domény obecných prvků nad touto množinou atomů. Množina atomů specifikujících prvků oproti $\sum$ obsahuje samotný anonymní prvek a pak podle Lemmy 3 musí platit, že množina obecných prvků je podmnožinou specifikujících prvků pro nějaké $\sum$.

Po operaci specifikace následuje další operace, která z ní vychází a která bude definována pro množinu obecných prvků a specifikující prvek. Jedná se o operaci restrikce.

Definice 11: Restrikci na množině je struktura $\sqcap: \wp(\Delta(\Sigma)) \times \Delta(\Sigma \cup \{\sim\}) \cup '\sim' \longrightarrow \wp(\Delta(\Sigma))$, kterou definujeme jako

$$A \sqcap r = \{d : d \in A \cap \Gamma, (d \prec r)\}.$$

Restrikce pro nějakou množinu a specifikující prvek zobrazí množinu redukovanou o všechny původní prvky množiny, které jsou v relaci specifikace s prvkem, který je uveden jako druhý argument operace.

Jelikož relace specifikace a operace restrikce hrají důležitou úlohu ve výpočtech ALLL systémů, ilustrujeme je několika příklady.

Příklad 2: Mějme stav báze $\gamma = \{"(abc)", "(a(b)c)", "\left(a\left(b(c)\right)\right)", "\left(a(bc)\right)"\}$ a specifikující prvek $r = "(a \sim c)"$. Výsledkem operace restrikce je množina prvků z *γ*, které nejsou v relaci specifikace *r*. V této relaci jsou prvky $"(abc)"$ a $"(a(b)c)"$ a výsledkem operace restrikce je pak množina o dvou zbývajících prvcích $\gamma \sqcap r = \{"\left(a\left(b(c)\right)\right)", "\left(a(bc)\right)"\}$.

Příklad 3: Lze ukázat, že jeden prvek z báze může mít více specifikujících prvků. Pro prvek $"\left(a\left(b(c)\right)\right)"$ existuje relace specifikace se všemi specifikujícími prvky $"\sim"$, $"(a\sim)"$, $"\left(\sim\left(b(c)\right)\right)"$, $"(\sim\sim)"$, $"\left(a(b\sim)\right)"$, $"\left(a\left(\sim(c)\right)\right)"$, $"\left(a(\sim\sim)\right)"$, $"(\sim(b\sim))"$, $"\left(\sim\left(\sim(c)\right)\right)"$, $"(\sim(\sim\sim))"$, $"\left(a\left(b(c)\right)\right)"$, $"\left(a\left(b(\sim)\right)\right)"$, $"\left(a\left(\sim(\sim)\right)\right)"$, $"\left(\sim\left(b(\sim)\right)\right)"$ a $"\left(\sim\left(\sim(\sim)\right)\right)"$.

Nyní si řekneme dvě Lemmata, která se budou týkat uzavřenosti operace restrikce ve vztahu k doménám báze.

Lemma 8: Operace restrikce $A \sqcap r$ není uzavřena pro libovolnou doménu báze.

Lemma 9: Operace $A - A \sqcap r$ není uzavřena pro libovolnou doménu báze.

Důkaz: Operace restrikce je množinovým rozdílem množiny a její podmnožiny, která obsahuje prvky, které jsou v relaci specifikace s druhým argumentem operace. Operace množinového rozdílu není uzavřena pro libovolnou podmnožinu potenční množiny množiny atomů. Stejně tak operace rozdílu prvku báze domény s jeho restrikcí nějakým specifikujícím prvkem.

Příklad 4: Pro množinu atomů $\{\{'a', 'b', 'c'\}\}$ a doménu báze $\{\{"(a,b)"\},\{"(a,c)"\},\{"(a,d)"\}\}$ je výsledkem operace množinového rozdílu $\{"(a,b)"\} - "(a,\sim)" = \{\}$, ale prázdná množina není prvkem báze domény.

Lemmata 8 a 9 byla uvedena pro demonstraci toho, že pokud by agent pracoval s nějakou bází domény, která vymezuje všechny možné množiny obecných prvků, jenž mohou být stavem báze, tak po provedení některých z uvedených operací by mohlo dojít ke změně stavu báze mimo definovanou doménu báze. Proto následující operace nad bází budou toto zohledňovat a výsledky operací, které jsou mimo doménu báze zobrazí na symbol neúspěšné operace (viz. definice 8), který je vždy prvkem domény báze.

Další operace bude sloužit ke vkládání prvku do báze, a následující dvě budou vyhledávat v bázi specifické prvky a případně je mazat, nebo je prezentovat navenek jako vyhledané hodnoty v bázi.

Definice 12: Binární operaci vkládání obecného prvku nad množinou atomů Σ do báze β_L v nějakém stavu báze z domény báze Γ je definována jako zobrazení $+: \Gamma \times \Delta(\Sigma) \longrightarrow \Gamma \cup ' \perp '$, následovně:

$$\forall \gamma \in \Gamma, d \in \Delta(\Sigma): \left(\gamma + d = \begin{cases} (\gamma \cup \{d\}), (\gamma \cup \{d\}) \in \Gamma \\ ' \perp ', jinak \end{cases}\right)$$

Z definice je patrné, že operaci vkládané lze chápat jako sjednocení množin, nebo jako přidání prvku do množiny. Jelikož ani operace sjednocení není uzavřena vůči doméně jako podmnožině potenční množiny, tak i operace vkládání může neuspět, pokud je výsledek sjednocení mimo doménu báze. Pak je výsledkem této operace symbol neúspěšné operace.

Operace výběru z báze je definována pomocí operace restrikce a výsledkem restrikce je množina obecných prvků. U ALLL systémů budeme ale u všech operací požadovat jako výsledek pouze jeden obecný prvek. Proto i operace výběru musí vybrat pouze jeden prvek z množiny obecných prvků. Proto zavedeme funkci pro výběr z množiny *sel*.

Definice 13: Funkce *sel* je funkcí nedeterministického výběru prvku z množiny obecných prvků $sel: \wp(\Delta(\Sigma)) \rightarrow \Delta(\Sigma)$.

Konkrétní definici této funkce neuvedeme, bude záležet na způsobu implementace při realizacích jednotlivých ALLL systémů.

Operace výběru je nyní definována následovně.

Definice 14: Binární operace výběru prvku nad množinou atomů Σ z báze β_L v nějakém stavu z domény báze Γ definujeme jako zobrazení $\sigma: \Gamma \times \Delta(\Sigma \cup \{'\sim'\}) \cup '\sim' \longrightarrow \Gamma \cup ' \perp '$ takové, že platí

$$\forall \gamma \in \Gamma, r \in \Delta(\Sigma \cup \{ '\sim' \}) \cup '\sim': \left(\sigma(\gamma, r) = \begin{cases} sel(\gamma \sqcap r), \gamma \sqcap r \in \Gamma \\ ' \perp ', jinak \end{cases}\right)$$

Funkce nedeterministického výběru zanáší nedeterminismus do ALLL systémů. Tento problém je ale podobný i u dalších systémů, kde se pracuje s představami agenta a provádí se dotazováním do báze představ. Už v systému AgentSpeak(L) byly uvedeny hned tři funkce pro výběr události, substitucí (což je obdoba našeho případu) a zamýšleného prostředku pro dosažení události, které nebyly blíže definovány a jejich realizace byla ponechána autorům konkrétních nástrojů. V nich se většinou realizovaly jako FIFO či podle toho, jak byly plány, ze kterých se zamýšlené prostředky vybírají,

deklarovány v době návrhu systému. Ještě bychom rádi poznamenali, že problém výběru prostředků v AgentSpeak(L) systému JASON byl také tématem výzkumu autora této práce [74].

Příklad 5: Jako příklad uvedeme situaci, ve které stav báze je $\gamma = \{"(a\ b)", "(a\ c)", "\ (b\ (c\ d))", "\ ((a\ b)\ c)"\}$. Výsledek operace výběru pro specifikující prvek "$(\sim\ \sim)$" je prvek vybraný funkcí *sel* z celé množiny tohoto stavu báze. Specifikující prvek "$(a\ \sim)$" použitý v operaci výběru z báze by vedl k zúžení této množiny pouze na první dva uvedené prvky a z nich by bylo vybíráno funkcí *sel*. To ovšem jen v případě, že množina s těmito dvěma prvky by byla přípustná v doméně báze.

Definice 15: Binární operace odstranění prvku nad množinou atomů Σ z báze β_L definujeme jako zobrazení $-: \Gamma \times \Delta(\Sigma \cup \{'\sim'\}) \cup\ '\sim' \longrightarrow \Gamma$ takové, že platí následující vztah.

$$\forall \gamma \in \Gamma, r \in \Delta(\Sigma \cup \{'\sim'\}) \cup\ '\sim': \left(\gamma - \mathrm{r} = \begin{cases} \gamma - \sigma(\Gamma, r), \gamma - \sigma(\Gamma, r) \in \Gamma \\ '\perp', jinak \end{cases}\right)$$

Zatímco operace vkládání slouží k přidání jednoho prvků do aktuálního stavu báze, operace odstranění může mít za následek odebrání více než jednoho prvku z aktuálního stavu báze.

3.3.3 Akce nad ALLL bází bez uspořádání

Báze vymezuje doménu hodnot, které mohou být stavem báze a operace, které lze nad bází provádět. Dále jsme také hovořili o aktuálním stavu báze, který je dán nějakým prvkem z domény báze a reprezentuje její vnitřní stav. Pokud bychom uvažovali provádění výpočtů s bází ve formě posloupnosti akcí nad bází, tak pak by aktuální stav celého systému zahrnoval nejen bázi a její stav, ale i nějakou strukturu reprezentující postup výpočtu. Dvojici stavu báze jako systému, a plánu, jako posloupnosti akcí, které mají být na systém postupně aplikovány, budeme u ALLL prvků nazývat konfigurací systému. V případě bází samotných budeme také uvažovat takové konfigurace systému, ale jelikož báze budeme používat jen jako podsystémy větších systémů a stav bází s plánem k vykonání budou jen část kontextu výpočtu těchto systému, nebudeme nyní konfigurace pro samotné báze formálně definovat.

Nicméně kontext výpočtu báze se bude měnit tak, jak budou postupně jednotlivé akce plánu aplikovány na aktuální stav báze a to do té doby, než budou všechny akce vykonány a konfigurace systému bude zahrnovat pouze prázdný plán.

Abychom mohli definovat plán, musíme nejdřív definovat akce, které plán tvoří. Všechny operace, které jsme doposud uvedli, mají společné jednak to, že jsou definovány nad prvkem z báze domény nebo prvkem neúspěchu akce jako prvním argumentem a nad nějakým obecným nebo specifikujícím prvkem jako druhým argumentem. Také všechny operace mají pevný bod v podobě neúspěšné operace a to v případech, kdy prvním argumentem je symbol neúspěšné operace. Díky dříve uvedeným definicím můžeme nyní definovat množinu akcí nad bází pro nějakou množinu atomických prvků Σ, kterou značíme jako $\mathrm{O}_L(\Sigma)$. Uvedou ji následující definice.

Definice 16: Akce je ALLL obecný prvek složený z operátoru a argumentů.

Konvence 1: Akce jsou obecné prvky, které uvádíme v uvozovkách, a mají tvar "$(\odot\ p)$". V následujícím textu budeme tyto akce uvádět bez uvozovek, přesto že se jedná o obecný prvek,

který v uvozovkách zapisujeme. Také přidáme čárku mezi operátor a argument. Akce ALLL systému budeme tedy uvádět ve tvaru $(\odot, p)$.

Definice 17: Množinu všech proveditelných akcí nad nějakou bází β_L značíme jako $\mathrm{O}_L(\Sigma)$ a ta je tvořena jako obecný prvek následovně.

$$\mathrm{O}_L(\Sigma \cup \{'\sim'\}) = \{"(+)" \cdot a\colon a \in \Delta(\Sigma)\} \cup \{\mathrm{op} \cdot a\colon \mathrm{op} \in \{"(-)","(\sigma)"\}, a \in \Delta(\Sigma \cup \{'\sim'\})\}$$

Tedy podle definic 16 a 17 jsou akce báze bez uspořádání dvojice, kde prvním prvkem je symbol akce a druhým prvkem je argument akce, kterým je pro operaci vkládání obecný prvek a pro akce odstranění a výběru specifikující prvek. Množina akcí je pak množinou uspořádaných n-tic a tak lze uvést, jak taková množina souvisí s množinami lineárních nebo obecných prvků nad nějakou množinou atomů.

Lemma 10: Zápisy akcí výše uvedeným způsobem jsou obecné prvky a jsou podmnožinou $\mathrm{O}_L(\Sigma) \subseteq \Delta(\Sigma \cup \{' + ','\text{-}','\sigma','\sim'\})$.

Důkaz: Prvky $"(+)","(-)"$ a $"(\sigma)"$ patří do množiny $\Delta(\Sigma \cup \{'+','\text{-}','\sigma','\sim'\})$, což plyne z Lemmy 2. Množiny $\Delta(\Sigma)$ a $\Delta(\Sigma \cup \{'\sim'\})$ jsou podmnožinou $\Delta(\Sigma \cup \{'+','\text{-}','\sigma','\sim'\})$. Podle Lemmy 4 spojením dvou obecných prvků z množiny $\Delta(\Sigma \cup \{'+','\text{-}','\sigma','\sim'\})$ vznikne opět obecný prvek z této množiny.

Přiklad 6: Uvedeme příklad báze bez uspořádání a ukážeme množinu akcí nad ní.

Báze je postavena na množině atomů $\Sigma = \{'petr','pavel','cer','mod','zel'\}$ a množině operací $\mathrm{O}_L(\Sigma)$. Množina obecných prvků báze by byla například

$$\Delta(\Sigma) = \{"(\quad)","(petr)","(pavel)",\ldots$$

$$"(zel)","(petr\ petr)","(petr\ \ pavel)",\ldots"(zel\ zel)","\big((petr)\ pavel\big)"\ldots\}$$

Pro bázi, kterou chceme používat k uchovávání zpráv. Zpráva je vždy dvojice, ve které první prvek je adresát zprávy z množiny $\{"(petr)","(pavel)"\}$ a druhým prvkem je obsah zprávy ve formě obecného prvku. Doména báze je podmnožina $\Gamma \subset \wp\big(\Delta(\Sigma)\big)$. Konkrétněji $\Gamma = \{"(petr)","(pavel)"\} \times \Delta(\Sigma)$

Množina akcí by zahrnovala například akce $\mathrm{O}_L(\Sigma) = \big\{"\big(+\ \big((\mathrm{petr})\ (\mathrm{cer})\big)\big)","\big(?\ \big((\mathrm{petr})\ (\sim)\big)\big)","(\text{-}\sim)"\ldots\big\}$. První akce má za argument obecný prvek, ostatní dvě akce mají za argumenty specifikující prvky.

Příklad 7: Báze představ ALLL agenta pracujícího s množinou atomů Σ má doménu, která obsahuje všechny množiny obecných prvků, tedy $\Gamma = \wp\big(\Delta(\Sigma)\big)$. Doménou báze vstupních dat je ale pouze podmnožina $\Gamma \subseteq \wp\big(\Delta(\Sigma)\big)$ taková, že prvním prvkem obecného prvku je jméno agenta. Báze plánů agenta zase obsahuje pouze dvojice, ve kterých je prvním prvkem název plánu a druhým plán. Množina atomů zde ale musí zahrnovat i sjednocení množin plánů s množinou jmen agentů.

Uvedeme i příklad stavu báze. Z definice 8 plyne, že stavem báze je prvek z domény báze.

Příklad 8: Stavem báze představ z příkladu 7 může být například prvek

$$\gamma = \{"(petr\,(petr\,zel))","\,(petr\,(pavel\,zel))","(pavel\,(pavel\,mod))"\}$$

Takový stav báze by mohl sloužit k modelování agentových představ o barvě jiného agenta. Zatímco agent Petr věří, že oba agenti mají zelenou barvu, agent Pavel věří, že má barvu modrou a o barvě agenta Petra nemá představu.

3.3.4 Vnitřní přechodová a výstupní funkce ALLL báze bez uspořádání

Vztah mezi vstupy, vnitřním stavem a výstupy báze budeme definovat pomocí dvou funkcí, které jsme uvedli v definici báze. Obě funkce budou definovány pro stav báze a akci z množiny akcí báze. Celá přechodová relace systému by pak mohla být realizována pomocí těchto funkcí jako zobrazení vstupů báze na její výstup s příslušnou změnou vnitřního stavu báze. Pouze jednu přechodovou relaci budeme uvádět v případě komplexnějších systémů, a ta bude relací mezi konfiguracemi tohoto prvku, což znamená vztahem mezi jeho vnitřními stavy a plány, které jsou na prvek aplikovány. K tomu se ale dostaneme později.

Pokud jsou systémy, jako je nyní popisované báze, otevřené, pak pro jejich lepší zasazení do nějakého většího systému uvedeme funkci, které budou zobrazovat aktuální stav báze a akci na nový stav báze a dále také funkci, která na základě aktuálního vnitřního stavu zobrazí patřičnou výstupní hodnotu. To následně umožní přehlednější definice fungování hierarchicky nadřazených systémů.

Definice 17 zavedla množinu všech možných akcí nad bází bez upořádání. Tyto akce budou nějak transformovat stav báze, nebo ke stavu báze budou přistupovat. Nyní budeme definovat vnitřní přechodovou funkci báze pro jednotlivé její akce, pak ukážeme na příkladu, jak fungují, a dále přejdeme k definici výstupní funkce báze.

Definice 18: Vnitřní přechodová funkce báze β_L je zobrazení $\ltimes_{L,I}: \Gamma \times O_L(\Sigma) \to \Gamma$ a je definováné následovně

$$\forall \gamma \in \Gamma, (\odot, \mathrm{p}) \in O_L(\Sigma) \cdot \left(\gamma \ltimes_{L,I} (\odot, \mathrm{p}) = \begin{cases} \gamma + p \;\; iff \;\; \odot =' +', \gamma + p \neq ' \perp ' \\ \gamma - p \;\; iff \;\; \odot = ' -', \gamma - p \neq ' \perp ' \\ \gamma \,, jinak \end{cases} \right)$$

Pouze akce přidání a odebrání prvku do/z báze má za výsledek změnu báze samotné, což znamená, že výsledný stav zobrazený vnitřní přechodovou funkcí se liší od původního stavu báze. Akce výběru prvku ponechá bázi v původním stavu.

Příklad 9: Uvažujme stav báze z příkladu 6. Akce přidání prvku do báze a testování báze mohou být například následující.

$\gamma \ltimes_{L,I} "(+, (pavel, (petr, zel)))" =$
$\{"(petr, (petr, zel))","(petr, (pavel, zel))","(pavel, (pavel, mod))","(pavel, (petr, zel))"\}$

$$\gamma \ltimes_{L,I} (\sigma, (petr, \sim)) = \{"(petr, (petr, zel))","(petr, (pavel, zel))"\}$$

Definice 19: Výstupní funkce báze β_L je zobrazení $\ltimes_{L,O}: \Gamma \times O_L(\Sigma) \to \Gamma \cup \{' \perp '\}$ definované následovně

$$\forall \gamma \in \Gamma, (\odot, \mathrm{p}) \in \mathrm{O}_L(\Sigma) \cdot \left(\gamma \ltimes_{L,O} (\odot, \mathrm{p}) = \begin{cases} \gamma + p \ \ iff \ \ \odot = '+' \\ \gamma - p \ \ iff \ \odot = ' - ' \\ \gamma \ \sigma \ p \ \ iff \ \odot = '\sigma' \end{cases} \right)$$

Výstupní funkce na rozdíl od vnitřní přechodové funkce zobrazuje i symbol neúspěšné operace. V systémech, které budou tuto bázi zahrnovat, může proto docházet v interpretaci k zohlednění těchto neúspěšných operací nad bází a případnému přehodnocení prováděné činnosti. To ale bude náplní až dalších kapitol.

Nyní si ještě ukážeme, že pro některé různé báze jsou obě výše definované funkce stejné. O jaké báze se jedná, o tom hovoří následující dvě Lemmy.

Lemma 11: Pro dvě báze bez uspořádání $\beta_{L1} = (\Sigma_1, \Gamma_1, \gamma_{0_1}, \ltimes_{L,I1}, \ltimes_{L,O1})$ a $\beta_{L2} = (\Sigma_2, \Gamma_2, \gamma_{02}, \ltimes_{L,I2}, \ltimes_{L,O2})$ platí, že pokud $\Sigma_1 = \Sigma_2$ a $\Gamma_1 = \Gamma_2$, pak i $\ltimes_{L,I1} = \ltimes_{L,I2}$

Lemma 12: Pro dvě báze bez uspořádání $\beta_{L1} = (\Sigma_1, \Gamma_1, \gamma_{0_1}, \ltimes_{L,I1}, \ltimes_{L,O1})$ a $\beta_{L2} = (\Sigma_2, \Gamma_2, \gamma_{02}, \ltimes_{L,I2}, \ltimes_{L,O2})$ platí, že pokud $\Sigma_1 = \Sigma_2$ a $\Gamma_1 = \Gamma_2$, pak i $\ltimes_{L,O1} = \ltimes_{L,O2}$

Důkaz: Funkce se rovnají, pokud se rovnají jejich definiční obory a zobrazované prvky pro každý vzor z definičního oboru. Definičním oborem pro funkce $\ltimes_{L,O1}, \ltimes_{L,O2}, \ltimes_{L,I1}$ a $\ltimes_{L,I2}$ je vždy množina $\Gamma_1 \times \mathrm{O}_L(\Sigma_1) = \Gamma_2 \times \mathrm{O}_L(\Sigma_2)$. Operace vkládání, odstranění a výběru jsou definovány definicemi 12, 14 a 15 pro dvojice stavu báze a argumentu operace. Každá z operací je definována jen na základě stavu báze, argumentu operace a domény báze. Jelikož domény bází jsou si rovny, pak i pro stejné prvky z definičního oboru funkcí báze jsou zobrazené prvky těmito operacemi stejné a dále jsou podle definic 18 a 19 stejné i tyto funkce.

3.4 Systém ALLL báze s uspořádáním

Báze s uspořádáním umožní do systémů zavést princip vkládání a výběru prvků známý z abstraktního datového typu fronta. Příchozí zprávy jsou řazeny do báze podle okamžiku příchodu a výběr z báze bude na rozdíl od báze bez uspořádání probíhat po jednotlivých prvcích podle tohoto uspořádání. Abychom měli možnost identifikovat zprávy podle jména odesilatele, umožní operace nad touto bází zúžit výběr podle předpony, čímž myslíme nějakou část obecného prvku zleva. To znamená, že budou vybírány pouze takové prvky, které jsou složeny z dvou částí, kde část na začátku prvku lze vybrat podle nějakého specifikujícího prvku. A z těchto prvků bude vybrán ten, který byl vložený nejdříve.

3.4.1 Struktura systému ALLL báze s uspořádáním

Stejně jako v případě báze bez uspořádání začneme i v tomto případě definicí struktury báze

Definice 20: Systém báze s uspořádáním β_Q je pětice $\beta_Q = (\Sigma, \Gamma, \mathrm{C}_0, \ltimes_{Q,I}, \ltimes_{Q,O})$, kde Σ je množina atomů, Γ je doména báze, $\Gamma \subseteq \wp(\Delta(\Sigma)) \cup ' \perp ', ' \perp ' \in \Gamma$, C_0 je počáteční konfigurace báze, $\ltimes_{Q,I}$ je vnitřní přechodová funkce báze a $\ltimes_{Q,O}$ je výstupní funkce báze.

Struktura báze s uspořádáním je obdobná struktuře báze bez uspořádání. Pouze množina akcí a přechodové funkce se liší a ty budou definovány na základě nových operací, které si pro tuto bázi nyní představíme.

3.4.2 Vnitřní stav ALLL báze s uspořádáním

Nejdříve ale uvedeme, jak bude reprezentován vnitřní stav báze s uspořádáním. Na rozdíl od stavu báze bez uspořádání, kde stavem byl pouze nějaký prvek domény ve tvaru množiny obecných prvků, u stavu báze s uspořádáním to bude dvojice, kterou nazýváme konfigurace báze s uspořádáním. Konfigurace zde znamená něco jiného, než bude znamenat u ALLL prvků, kde, jak jsme již zmínili, bude konfigurací systému vnitřní stav systému a plán aplikovaný na tento systém.

Definice 21: Konfigurace báze s uspořádáním β_Q je značena jako C a má formu dvojice $(\gamma, \rightsquigarrow)$, kde γ je stav báze, $\gamma \in \Gamma$, a $\rightsquigarrow$ je relace ostrého uspořádání nad γ.

Definice 22: Doménu konfigurací báze s uspořádáním pro doménu báze Γ budeme značit $\mathrm{C}(\Gamma)$ a definujeme ji jako $\mathrm{C}(\Gamma) = \{(\gamma, \sim) : \gamma \in \Gamma, \sim je\ relace\ ostrého\ uspořádání\ na\ \gamma\}$

Relace ostrého uspořádání je relace antireflexivní, antisymetrická a tranzitivní. Relace $\rightsquigarrow$ se vytváří během provádění akcí nad bází. Vztah mezi relací uspořádání a stavem báze bude dán následujícími předpoklady. Pro prázdný stav báze, kdy $\gamma = \{\}$, bude i relace uspořádání $\rightsquigarrow = \{\}$. Další vztahy mezi stavy báze s uspořádáním a relací uspořádání budou vznikat tak, jak budou prováděny operace v bázi. Každý nově vložený prvek bude v relaci se svými předchůdci tím způsobem, že každý z původních prvků předchází vložený prvek.

Tato doména obsahuje konfigurace se všemi prvky z domény báze a to ve všech možných ostrých uspořádáních. Poznamenáváme, že pro jeden prvek z γ existuje více ostrých uspořádání, konkrétně pro množinu o *n* prvcích existuje *n!* permutací a stejný počet je i relací ostrého uspořádání.

3.4.3 Operace nad ALLL bází s uspořádáním

Operacemi pro tento typ báze jsou operace vkládání, odstranění a výběru prvku, což jsou názvy operací stejné jako v případě báze bez uspořádání. V tomto případě se ale operace budou lišit tím, že budou respektovat relaci uspořádání nad stavem báze a budou měnit její konfiguraci, jak budou prvky vkládány a odstraňovány. Jako symboly pro jednotlivé operace použijeme symbol $\hookrightarrow$ pro vkládání, $\hookleftarrow$ pro výběr a $\swarrow$ pro odstranění prvku do/z báze s uspořádáním.

Pro zjednodušení zápisu definic jednotlivých operací si nejprve uvedeme zápisy relací pre_p^γ a pos_p^γ pro nějaký stav γ a prvek *p*, to znamená relace na množině γ ∪ $\{p\}$. Tyto relace jsou množiny dvojic $pre_p^\gamma = \{(g,p) | g \in \gamma\}$ a $pos_p^\gamma = \{(p,g) | g \in \gamma\}$

Uvedené relace jsou vztahy mezi stavem báze γ a prvkem *p* a to takové, že každý prvek ze stavu báze γ předchází prvek *p* v případě relace pre_p^γ, nebo jej následuje v případě relace pos_p^γ. Nyní již můžeme definovat jednotlivé operace báze s uspořádáním.

Definice 23: Operace vkládání prvku do báze s uspořádáním β_Q je zobrazení $\hookrightarrow \subseteq \Delta(\Sigma) \times \mathrm{C}(\Gamma) \rightarrow \mathrm{C}(\Gamma) \cup {}' \perp {}'$ a je definováno formulí

$$\forall p \in \Delta(\Sigma), (\gamma, \rightsquigarrow) \in \mathrm{C}(\Gamma) \cdot \left(p \hookrightarrow (\gamma, \rightsquigarrow) = \begin{cases} (\gamma \cup \{\mathrm{p}\}, \rightsquigarrow \cup\ pre_p^\gamma) & iff\ p \notin \gamma, \gamma \cup p \in \Gamma \\ {}' \perp {}' & jinak \end{cases} \right)$$

Nová konfigurace kromě nového prvku ve stavu báze obsahuje i příslušné nové relace. Princip definice operace odebrání prvku z báze s uspořádáním je podobný.

Definice 24: Operace odebrání prvku z báze s uspořádáním β_Q je zobrazení $\swarrow \subseteq \Delta(\Sigma \cup \{"\sim"\}) \cup {}'\sim' \times \mathbb{C}(\Gamma) \to \mathbb{C}(\Gamma) \cup {}' \perp {}'$ a je definováno formulí

$$\forall p \in \Delta(\Sigma \cup \{'\sim'\} \cup' \sim',$$
$$(\gamma, \rightsquigarrow) \in \mathbb{C}(\Gamma) \cdot \left(p \swarrow (\gamma, \rightsquigarrow) = \begin{cases} (\gamma - \{\mathrm{p}\}, \rightsquigarrow - pre_p^{\gamma} - pos_p^{\gamma}) & iff \ \ \gamma - p \in \Gamma \\ {}' \perp {}' & jinak \end{cases} \right)$$

Operace výběru prvku z báze s uspořádáním se od předchozích dvou bude lišit. Tato operace nebude sloužit k transformaci konfigurace báze, ale bude zobrazovat konfiguraci báze a specifikující prvek pouze na jeden z prvků báze, pokud existuje odpovídající prvek, nebo na symbol neúspěšné operace, pokud takový neexistuje. Liší se také od operace výběru z báze s uspořádáním v tom, že výsledný prvek nemusí být prvek v relaci specifikace s druhým argumentem operace, ale stačí, když jeho předpona bude v takovéto relaci. Budeme-li mít specifikující prvek *s* a prvek *o*, které jsou v relaci specifikace $o \prec s$, pak jakýkoli prvek vzniklý spojením $o \cdot o'$ může být z konfigurace báze vybrán operací výběru s argumentem *s*. Abychom tuto funkci mohli elegantně definovat, definujeme nejprve funkci *fst* nad množinou s uspořádáním, která zobrazí takovou množinu na první prvek z ní vzhledem k tomuto uspořádání.

Definice 25: Funkce *fst* je struktura $\Gamma \times \wp(\Delta(\Sigma) \times \Delta(\Sigma)) \to \Delta(\Sigma) \cup \perp$ a je definována tak, že jejím výsledkem je obecný prvek p nebo symbol neúspěšné operace $' \perp '$ podle formule

$$\forall(\gamma, \rightsquigarrow) \in \mathbb{C}(\Gamma) \cdot \left(fst(\gamma, \rightsquigarrow) = \begin{cases} p \ \ iff \ \ \gamma \neq \{\}, \forall q \in (\gamma - p) \cdot ((p, q) \in \rightsquigarrow) \\ {}' \perp {}' \qquad\qquad\qquad\qquad\qquad\qquad iff \ \ \gamma = \{\} \end{cases} \right)$$

Funkce *fst* zobrazí nějaký prvek, pokud je argumentem neprázdná množina a všechny ostatní prvky v konfiguraci zobrazený prvek předchází podle dané relace ostrého uspořádání.

Funkce výběru podle předpony bude zobrazovat konfiguraci s množinou prvků, jejichž délka je stejná nebo větší než je délka specifikujícího prvku. Bude se jednat o prvky vybrané podle předpony tímto specifikujícím prvkem. Uspořádání pak bude podmnožinou původního uspořádání a to takovou, že bude obsahovat všechny dvojice z původního uspořádání pro prvky, které byly vybrány. Formálně tuto funkci definujeme následovně.

Definice 26: Operaci výběru podle předpony budeme značit podobně jako operaci výběru v bázích bez uspořádání, a to konkrétně jako σ^+. Je to zobrazení $\sigma^+: \mathbb{C}(\Gamma) \times \Delta(\Sigma \cup \{ '\sim' \}) \cup {}'\sim' \longrightarrow \mathbb{C}(\Gamma)$ definované následovně.

$$\forall p \in \Delta(\Sigma \cup \{'\sim'\}) \cup {}'\sim', (\gamma, \rightsquigarrow) \in \mathbb{C}(\Gamma): (\sigma^+\big((\gamma, \rightsquigarrow), p\big) = (\gamma', \rightsquigarrow'),$$

$$\gamma' = \left\{ q \middle| \begin{cases} \exists i: p, q \in \Delta^i(\Sigma), q \prec p \ \vee \\ \exists i, j \exists q1, q2 \in \Delta(\Sigma): p \in \Delta^i(\Sigma \cup \{'\sim'\}), q \in \Delta^{i+j}(\Sigma), q = q1 \cdot q2, q1 \in \Delta^i(\Sigma), q1 \prec p \end{cases} \right\}$$

$$\rightsquigarrow' = \{(q, r): (q, r) \in \rightsquigarrow, q, r \in \gamma'\}$$

Nyní lze definovat operaci výběru prvku jako složenou funkci.

Definice 27: Operace výběru prvku z báze s uspořádáním β_Q je zobrazení $\hookleftarrow \subseteq \Delta(\Sigma \cup \{"\sim"\}) \cup {}'\sim' \times \mathbb{C}(\Gamma) \to \Delta(\Sigma)$ a je definována složením dvou předchozích funkcí.

$$\forall p \in \Delta(\Sigma \cup \{"\sim"\}) \cup '\sim', (\gamma, \rightsquigarrow) \in \mathrm{C}(\Gamma) \cdot (\, p \hookleftarrow (\gamma, \rightsquigarrow) = fst(\, \sigma^{+}\big((\gamma, \rightsquigarrow), p\big))$$

Pokud konfigurace po operaci výběru podle předpony obsahuje alespoň jeden prvek, pak je vybrán první podle relace uspořádání báze. Pro ilustraci vyhodnocování funkcí jako operací nad bází s uspořádáním uvedeme několik příkladů, ale až poté, co definujeme akce báze s uspořádáním.

3.4.4 Akce ALLL báze s uspořádáním

Akce, které lze aplikovat na bázi s uspořádáním, budou opět dvojice v souladu s definicí 16. V tomto případě to budou akce pro přidání, odstranění a výběr do/z báze s uspořádáním a tyto akce budou tvořit množinu akcí podle následující definice.

Defnice 28: Množina všech možných akcí nad bází s uspořádáním β_Q je značena jako $O_Q(\Sigma)$ a je to množina

$$\mathrm{O}_Q(\Sigma) = \{"(\hookrightarrow)" \cdot a \colon a \in \Delta(\Sigma)\} \cup \{\mathrm{op} \cdot a \colon \mathrm{op} \in \{"(\swarrow)", "(\hookleftarrow)"\}, a \in \Delta(\Sigma \cup \{'\sim'\}) \cup '\sim'\}$$

Souvislost mezi množinou akcí báze s uspořádáním a množinou lineárních prvků nad nějakou abecedou uvede následující Lemma.

Lemma 13: Zápis akcí ve výše uvedeném zápise jsou obecné prvky a jsou podmnožinou $\mathrm{O}_Q(\Sigma) \subseteq \Delta(\Sigma \cup \{'\hookrightarrow', '\swarrow', '\hookleftarrow', '\sim'\})$.

Důkaz: Obdobný důkazu Lemmy 10

Akce budou aplikovány na bázi a výsledkem aplikace bude stejně jako v případě báze bez uspořádání změna konfigurace báze vnitřní přechodovou funkcí. Výsledek akce, kterým je zobrazená konfigurace, je dostupný přes výstupní funkci báze bez uspořádání.

3.4.5 Vnitřní přechodová a výstupní funkce ALLL báze s uspořádáním

Nejprve se zaměříme na vnitřní přechodovou funkci. Stejně jako v případě báze bez uspořádání, i zde jsou pro definici této funkce použity pouze dvě operace ze tří. Třetí z operací, to je výběr prvku z báze, bude sloužit jako výstupní funkce tohoto systému a bude zpřístupňovat vnitřní stav báze jeho okolí. Transformovat bázi budou pouze akce vkládání a odstranění, které v následující definici spojíme do vnitřní přechodové funkce následovně.

Definice 29: Vnitřní přechodová funkce báze s uspořádáním je funkce $\ltimes_{Q,I} \colon \mathrm{C}(\Gamma) \times \mathrm{O}_Q(\Sigma) \to \Gamma$ a je definovaná takto

$$\forall \mathrm{C} \in \mathrm{C}(\Gamma), (\odot, \mathrm{p}) \in \mathrm{O}_Q(\Sigma) \colon \left(\mathrm{C} \ltimes_{Q,I} (\odot, \mathrm{p}) = \begin{cases} p \hookrightarrow \gamma \;\; iff \;\; \odot = '\hookrightarrow', \gamma \hookrightarrow p \neq '\perp' \\ p \swarrow \gamma \;\; iff \;\; \odot = '\swarrow', \gamma \swarrow p \neq '\perp' \\ \gamma \, , jinak \end{cases} \right)$$

Opět bude mezi vnitřní přechodovou funkcí a výstupní funkcí systému báze rozdíl v tom, že výstupní funkce neúspěšnou operaci zobrazuje na příslušný atom, kdežto vnitřní funkce ponechávala předchozí stav. Okolní systémy mohou na základě výsledku výstupní funkce patřičně reagovat.

Definice 30: Výstupní funkce báze s uspořádáním je funkce $\ltimes_{Q,O} \colon \mathrm{C}(\Gamma) \times \mathrm{O}_Q(\Sigma) \to \Gamma \cup \{\perp\}$ a je definovaná takto

$$\forall \mathrm{C} \in \mathrm{C}(\Gamma), (\odot, \mathrm{p}) \in \mathrm{O}_Q(\Sigma): \left(\mathrm{C} \ltimes_{Q,O} (\odot, \mathrm{p}) = \begin{cases} p \hookrightarrow \gamma \ \ iff \ \ \odot = ' \hookrightarrow ' \\ p \swarrow \gamma \ \ iff \ \ \odot = ' \swarrow ' \\ p \hookleftarrow \gamma \ \ iff \ \ \odot = ' \hookleftarrow ' \end{cases} \right)$$

Jak je vidět, výstupní funkce obou bází provedou pro aplikovanou akci příslušné operace tak, jak jsou definované. Nyní si jednotlivé akce báze s uspořádáním představíme na příkladech.

Příklad 10: Na bázi s uspořádáním postupně aplikujeme čtyři akce. První dvě akce budou akce vkládání, poté bude následovat akce výběru a na závěr akce odstranění. Počáteční konfigurace báze bude následující.

$$\mathrm{C}_0 = (\gamma_0, \rightsquigarrow_0) = \left(\{"(\mathrm{m},37,(\mathrm{f},\mathrm{z}))", "(m, 60, (b, b))"\}, \{("(m, 37, (f, z))", "(m, 60, (b, b))")\}\right).$$

Stav báze tvoří dva prvky a v relaci je jedna dvojice. Akce, které budeme na tuto a následné konfigurace postupně aplikovat, budou následující

"(↪ ,(z,36,(h,b)))","(↪ ,(m,17,(j,z)))","(↩ ,(m))" a "(↙ ,(m,~,(~,z)))".

Budeme provádět transformaci konfigurací vnitřní přechodovou funkcí. Výsledkem operací vkládání a odstraňování budou nové konfigurace báze podle operací, které odpovídají jednotlivým akcím.

$$\begin{aligned} &(\mathrm{z},36,(\mathrm{h},\mathrm{b})) \hookrightarrow \left(\{"(m, 37, (f, z))", " (m, 60, (b, b))"\}, \{("(m, 37, (f, z))", " (m, 60, (b, b))")\}\right) \\ &= \left(\{"(m, 37, (f, z))", "(m, 60, (b, b))", "(\mathrm{z},36,(\mathrm{h},\mathrm{b}))\}, \{("(m, 37, (f, z))", "(m, 60, (b, b))"),\right. \\ &\left.("(m, 37, (f, z))", "(z, 36, (h, b))"), ("(m, 60, (b, b))", "(z, 36, (h, b))")\}\right) = (\gamma', \rightsquigarrow_1) \end{aligned}$$

$$\begin{aligned} (\mathrm{m},17,(\mathrm{j},\mathrm{z})) \hookrightarrow\ & (\gamma', \rightsquigarrow_1)(\gamma', \sim') \\ &= \left(\{"(m, 37, (f, z))", "(m, 60, (b, b))", "(\mathrm{z},36,(\mathrm{h},\mathrm{b}))", "(\mathrm{m},17,(\mathrm{j},\mathrm{z}))"\},\right. \\ &\{("(m, 37, (f, z))", "(m, 60, (b, b))"), ("(m, 37, (f, z))", "(z, 36, (h, b))"), \\ &("(m, 37, (f, z))", "(m, 17, (j, z))"), ("(m, 60, (b, b))", "(z, 36, (h, b))"), \\ &\left.("(m, 60, (b, b))", "(m, 17, (j, z))"), ("(z, 36, (h, b))", "(m, 17, (j, z))")\}\right) \\ &= (\gamma', \rightsquigarrow_2) \end{aligned}$$

Za této konfigurace aplikujeme akci výběru a provedeme výstupní funkci, která využije operaci výběru. Argumentem akce je specifikující prvek a výsledkem akce první prvek z konfigurace, ve které mají všechny prvky jako první podprvek atom 'm'.

$$\begin{aligned} &(\mathrm{m}) \hookleftarrow (\gamma', \rightsquigarrow_2) \\ &= fst\left(\left(\{"(m, 37, (f, z))", "(m, 60, (b, b))", "(\mathrm{m},17,(\mathrm{j},\mathrm{z}))"\}, \{("(m, 37, (f, z))", "(m, 60, (b, b))"),\right.\right. \\ &\left.\left.("(m, 37, (f, z))", "(m, 17, (j, z))"), ("(m, 60, (b, b))", "(m, 17, (j, z))")\}\right)\right) = "(m, 37, (f, z))" \end{aligned}$$

Konfigurace je stále $(\gamma', \rightsquigarrow_2)$, což nyní změní provedení poslední akce. Vrátíme se k vnitřní přechodové funkci a ukážeme operaci odstranění.

$$(\mathrm{m},\sim,(\sim,\mathrm{z})) \swarrow (\gamma', \rightsquigarrow_2) = \left(\{"(m, 60, (b, b))", "(\mathrm{z},36,(\mathrm{h},\mathrm{b}))"\}, \{("(m, 60, (b, b))", "(z, 36, (h, b))")\}\right)$$

Následující dvě Lemmy budou obdobou Lemmat 11 a 12, které vyjadřovaly rovnost výstupních a vnitřních přechodových funkcí pro různé báze, pokud pracují se stejnou množinou atomů a mají stejnou doménu.

Lemma 14: Pro dvě báze bez uspořádání $\beta_{Q1} = (\Sigma_1, \Gamma_1, \mathrm{C}_{01}, \ltimes_{Q,I1}, \ltimes_{Q,O1})$ a $\beta_{Q2} = (\Sigma_2, \Gamma_2, \mathrm{C}_{02}, \ltimes_{Q,I2}, \ltimes_{Q,O2})$ platí, že pokud $\Sigma_1 = \Sigma_2$ a $\Gamma_1 = \Gamma_2$, pak i $\ltimes_{Q,I1} = \ltimes_{Q,I2}$

Lemma 15: Pro dvě báze bez uspořádání $\beta_{Q1} = (\Sigma_1, \Gamma_1, \mathrm{C}_{01}, \ltimes_{Q,I1}, \ltimes_{Q,O1})$ a $\beta_{Q2} = (\Sigma_2, \Gamma_2, \mathrm{C}_{02}, \ltimes_{Q,I2}, \ltimes_{Q,O2})$ platí, že pokud $\Sigma_1 = \Sigma_2$ a $\Gamma_1 = \Gamma_2$, pak i $\ltimes_{Q,O1} = \ltimes_{Q,O2}$

Důkaz: Obdobný důkazu Lemmat 11 a 12

3.5 Plány jako struktury akcí a operace nad nimi

Vnitřní přechodová a výstupní funkce bází pomohou realizovat systémy s nějakým chováním, ve kterých jsou báze podsystémy a na které je aplikováno více akcí za sebou. Posloupnost akcí budeme považovat za plán, který je aplikován na systém. Výpočet v takových systémech bude probíhat tak, jak se bude měnit jejich konfigurace včetně stavů a konfigurací bází. Kromě stavu báze se v každém kroku bude měnit i plán tím, jak budou jednotlivé akce po vykonání odebírány z plánu. Plán v případě hierarchického uspořádání bude moct obsahovat také akce, které budou mít dopad na jeho strukturu, ale o tom až později. Nyní přejdeme k obecným definicím ohledně plánů.

3.5.1 Plány jako obecné prvky první úrovně nad množinou akcí

Doposud byly akce tvořeny pro nějakou množinu atomů systému Σ. Nad touto množinou atomů se také tvoří argumenty akce. Tak tomu bylo v případě akcí nad bázemi, které podle definic 17 a 28 byly zahrnuty v množinách $\mathrm{O}_Q(\Sigma)$ a $\mathrm{O}_L(\Sigma)$. Jelikož v dalším textu budeme mít i jiné množiny akcí, budeme obecně akce nad abecedou Σ značit jako $\mathrm{O}(\Sigma)$, nebo jako $O(\mathrm{A})$, pokud množina atomů, ze kterých se budou akce skládat, se nebude rovnat množině atomů toho kterého systému. Pro jakoukoliv množinu akcí platí, že struktura jejich prvků je podle následující definice.

Definice 31: Množna akcí ALLL systému s množinou atomů A je značena $O(\mathrm{A})$ a platí, že $O(\mathrm{A}) \subseteq \{'('op\ arg')' : op \subseteq \mathrm{Op}, arg \subseteq \Delta(\mathrm{A})\}$, kde Op je množina operátorů, které jsou ve formě atomů.

Definice 31 je v souladu s dřívější definicí 16, která zaváděla strukturu akcí. Množina A je obvykle rovna, nebo je nadmnožinou množiny Σ toho kterého systému. Může navíc obsahovat symboly pro registry, pro anonymní prvek nebo další atomy, které budou tvořit argumenty pro akce nad systémy. Množiny operátorů se liší podle konkrétního systému, pro které jsou akce definovány. Operátory můžou nebo nemusí být součástí množiny A.

Nyní definujme plán ALLL systémů.

Definice 32: Množina plánu obsahuje lineární prvky nad nějakou množinou akcí $\mathrm{O}(\mathrm{A})$ a označíme ji jako $\Pi(\mathrm{O}(\mathrm{A}))$, a platí vztah $\Pi(\mathrm{O}(\mathrm{A})) \subseteq \mathrm{D}(\mathrm{O}(\mathrm{A}))$

Definice uvádějící struktury plánu nám umožní definovat komplexnější systémy s bází, které byly předeslány na začátku této kapitoly. Jelikož už máme k dispozici hned dvě množiny akcí, a to množinu akcí báze bez uspořádání $O_L(\mathrm{A})$ a množinu akcí báze s uspořádáním $O_Q(\mathrm{A})$, můžeme si ukázat, jak by mohl plán pro tyto báze vypadat. Pro tuto ukázku vezmeme množinu akcí $O_L(A)$ a systém s bází bez uspořádání.

Příklad 11: Plán v systému s bází z příkladu 10 může být například

$$\pi = "\Big(\big(+\,(\mathrm{pavel}, (\mathrm{petr}, \mathrm{cer}))\big)\,\big(-\,(pavel, \sim)\big)\Big)"$$

Dvě akce, které mají být v systému vykonány, provádí přidání jednoho obecného prvku do báze a odstranění obecného prvku. Akce odstranění se týká prvků, které jsou dvojicí, a ve kterých je první podprvkem atom ‘pavel’ a druhý podprvek této dvojice může být libovolný.

Plán je lineární posloupnost akcí podle definice 2. Jelikož akce je obecný prvek minimálně druhé úrovně, plán musí být obecným prvkem minimálně třetí úrovně. To si zavedeme formálně.

Lemma 16: Plán z množiny plánů $\Pi(\mathrm{O}(\mathrm{A}))$ je obecným prvkem nejméně třetí úrovně nad množinou atomů A.

Důkaz: Z definice 31 plyne, že množina akcí je obecný prvek obsahující za podprvky operátory, což jsou atomy, a za argumenty obecné prvky. Jelikož obecný prvek je nejméně první úrovně, podle definice 4 jsou akce nejméně druhé úrovně. Stejně tak plány jsou konstruovány z prvků nejméně druhé úrovně a tak plány jsou obecnými prvky nejméně třetí úrovně, opět v souladu s definicí 4.

3.5.2 Registry, systém registrů, akce s registry

V úvodu této teoretické kapitoly jsme zmiňovali pojem registr v souvislosti s ALLL výpočetními systémy. Registr, jak jej známe z architektur počítačů, je místo pro dočasné uložení nějaké informace, která je během výpočtu používána. Registr tedy bude také součástí ALLL systémů a jeho účel bude podobný. Pro jeho zavedení do zápisu plánů ALLL systémů rozšíříme množinu atomů akcí o symboly registrů z nějaké množiny symbolů registrů, kterou budeme značit *T*. Množinu akcí s registry budeme značit $\mathrm{O}_L\ (\mathrm{T}, \Sigma)$ pro nějakou množinu atomů báze Σ. Stejně tak budeme používat toto značení pro ostatní množiny akcí, například rozšířenou množinu akcí nad bázemi, nebo množiny zahrnující operace jiných ALLL systémů. Formálně si toto uvedeme následující konvenci.

Konvence 2: Zápis množin akcí pro množinu atomů Σ a množinu symbolů registrů T lze uvádět v následující formě.

$$\mathrm{O}(\mathrm{T}, \Sigma) = \mathrm{O}(\Sigma \cup \mathrm{T} \cup' \sim')$$

A další konvence zavede symbol, který budeme v dalších systémech používat jako symbol registru, jenž budeme považovat za aktivní, a který bude substituováván za aktuální stavy výpočtů před provedením akce, v jejímž argumentu je použit.

Konvence 3: Jeden symbol registru bude povinně součástí *T*. Bude se jednat o tzv. aktivní registr, budeme jej značit τ, a musí tedy platit $\tau \in T$.

Symbol registru může, ale nemusí, být součástí množiny atomů báze. Pokud báze má uchovávat data reprezentující fakta, nebo agentovy představy, registry v množině atomů báze faktů nebývají. Pokud ale báze má obsahovat plány, potom jsou registry součástí množiny atomů báze, protože plány symboly registrů běžně obsahují.

3.5.3 Operace substituce v ALLL systémech

Než přejdeme k definici lineárních plánů a jejich provádění, je třeba formálně uvést operaci substituce. Operace substituce je známa z predikátového počtu jako zobrazení proměnných na termy a obdobně tomu bude i v případě ALLL systémů. Bude se jednat o operaci, kdy nějaký atom v obecném prvku je nahrazen jiným obecným prvkem. Touto záměnou dojde k změně původního obecného prvku za jiný.

Operaci substituce jsme neuváděli v souvislosti se systémy bází, jelikož pro realizaci vnitřní přechodové ani výstupní funkce báze není substituce použita. V situaci, kdy budeme pracovat s plány jako programy popisujícími výpočet, budeme substituci využívat pro zanesení mezivýsledků do kontinua, tedy do transformovaného plánu po provedení aktuální akce.

Nejprve uvedeme definice pro substituce atomu, atomů obecného prvku, atomů v akci a atomů v plánu. Dále také uvedeme případy, kdy bude substituce prováděna za implicitně daný symbol aktivního registru, který jsme představili v konvenci 3.

Definice 33: Substituce atomu je zobrazení $sub: \Sigma \times \Sigma \times \Delta(\Sigma) \rightarrow \Delta(\Sigma) \cup \Sigma$ a zapisovat jí budeme v obvyklé notaci jako $a\left[\frac{n}{e}\right] = n'$, kde Σ je množina atomů, $n, n' \epsilon \Delta(\Sigma)$ jsou obecné prvky nad množinou atomů Σ, a dále $a, e \epsilon \Sigma$ jsou atomy, za které se bude případně substituovat.

$$\forall n \in \Delta(\Sigma), a, e \in \Sigma \left(a\left[\frac{n}{e}\right] = n', n' = \begin{cases} a, & iff\ a \neq e \\ n, & iff\ a = e \end{cases} \right)$$

Definice substituce atomů obecného prvku je následující.

Definice 34: Operace substituce atomů v obecném prvku je zobrazení $sub: \Delta(\mathrm{A}) \times \Sigma \times \Delta(\mathrm{A}) \rightarrow \Delta(\mathrm{A})$ a opět jí budeme zapisovat jako $n\left[\frac{n'}{a}\right] = n''$, kde Σ je množina atomů, $n, n', n'' \epsilon \Delta(\mathrm{A})$ jsou obecné prvku nad množinou atomů A, a dále $a \epsilon \Sigma$ je atom, který bude substituován. Operace substituce nahradí všechny výskyty atomu *a* v obecném prvku *n* za obecný prvek *n'*. Zápis $(t_1 \dots t_i)$ zahrnuje všechny obecné prvky, tj. index $i \geq 1$. Operaci substituce pak definujeme následovně.

$$\forall (t_1 \dots t_i), n' \in \Delta(\mathrm{A}), a \in \mathrm{A} \left((t_1 \dots t_i)\left[\frac{n'}{a}\right] = \left({t_1}'\left[\frac{n'}{a}\right] \dots {t_i}'\left[\frac{n'}{a}\right] \right) \right)$$

V případě, že jsou podprvky v obecném prvku atomy, budou substituovány podle definice 33. V případě vnořených obecných seznamů je substituce prováděna podle definice 34. Ve výsledku se provede substituce každého atomu na všech úrovních zanoření, což nebudeme dále dokazovat s tím, že to plyne z rekurzivního charakteru těchto definic.

Substituce je nyní definována obecně pro nějakou množinu atomů a obecné prvky nad touto množinou. Systém ji ale bude aplikovat výhradně na plány, které bude takto modifikovat na základě aktuální hodnoty výpočtu. Operaci substituce budeme vykonávat buďto na nějaké části aktuálního plánu, nebo na první akci v plánu.

Operace substituce v akcích i v plánech může substituovat buď uvedený prvek za jiný, pokud je substituce uvedena jako dvojice v obvyklé formě, nebo je prvek nahrazen za symbol aktivního registru. V systémech s hierarchickými plány budeme substituovat symboly registrů z množiny symbolů registrů T navzájem. To ale vždy jen v nějakém plánu, a proto nyní uvedeme operace substituce pro akce a plány.

Definice 35: Operace substituce v akci z množiny akcí $o \in \mathrm{O}(\mathrm{A})$ definované nad množinou atomů A jako dvojice $\left[\frac{e}{t}\right], e \in A,\ t \in \Delta(A)$ je zobrazení $\mathrm{O}(\mathrm{A}) \times \mathrm{A} \times \Delta(\mathrm{A}) \rightarrow O(\mathrm{A})$ definované jako

$$\forall (\odot, p) \in \mathrm{O}(\mathrm{A}), \forall t \in \Delta(\mathrm{A}), \forall e \in \mathrm{A}: ((\odot, p)\left[\frac{t}{e}\right] = (\odot, p\left[\frac{t}{e}\right]))$$

Provedení substituce v nějaké akci má za následek provedení substituce v argumentu akce. Výše uvedené zápisy představují operaci substituce za libovolný atom. Následující konvence umožní zkrátit zápis, pokud se bude substituovat za symbol aktuálního registru.

Konvence 4: Operace substituce v akci z množiny akcí $o \in \mathrm{O(A)}$ definované nad množinou atomů A a obecný prvek $t \in \Delta(A)$ je zobrazení $\mathrm{O(A)} \times \Delta\mathrm{(A)} \rightarrow O\mathrm{(A)}$ definované jako

$$\forall(\odot, p) \in \mathrm{O(A)}, \forall t \in \Delta\mathrm{(A)}: ((\odot, p)[t] = (\odot, p[\frac{t}{\tau'}])\})$$

Nyní vyjádříme množinu všech možných dvojic, které jsou argumentem operace substituce spolu s atomem či prvkem, na kterém se substituce má provádět.

Definice 36: Množina všech možných substitucí pro nějakou množinu atomů A bude označena jako

$$\mathrm{P}(A) = \left\{\left[\frac{t}{e}\right], t \in \Delta\mathrm{(A)}, e \in \mathrm{A}\right\}.$$

Další definice zavedou operaci aplikace množiny uspořádané posloupnosti substitucí na akci.

Definice 37: Uspořádanou posloupnost substitucí s relací ostrého uspořádání $\rightsquigarrow$ označíme μ^A,

$$\mu^A = \left(\left[\frac{t_1}{e_1}\right], \left[\frac{t_2}{e_2}\right], \dots \left[\frac{t_i}{e_i}\right]\right), [\frac{t_1}{e_1}][\frac{t_2}{e_2}] \dots [\frac{t_i}{e_i}] \in \mathrm{P}(A)\ .$$

Definice 38: Operace aplikace uspořádané množiny substitucí na akci je zobrazení $\mathrm{O(A)} \times \wp(\mathrm{P}^A) \rightarrow O\mathrm{(A)}$, které je definováno takto.

$$\forall(\odot, p) \in \mathrm{O(A)}, \forall \mu^A \in \wp(\mathrm{P}(A), \rightsquigarrow)\left((\odot, p)\mu = (\odot, p[\frac{t_1}{e_1}][\frac{t_2}{e_2}] \dots [\frac{t_i}{e_i}]), \mu^A = \left([\frac{t_1}{e_1}][\frac{t_2}{e_2}] \dots [\frac{t_i}{e_i}]\right)\right)$$

Lze snadno ukázat, že pro různá uspořádání posloupnosti těch samých substitucí vede jejich aplikace k jiným výsledkům. Například pro prvek "(x)" a dvojici substitucí $[\frac{"(a)"}{\prime x\ \prime}][\frac{"(b)"}{\prime x\ \prime}]$ jsou výsledky aplikací rozdílné podle toho, v jakém pořadí budou substituce aplikovány, jelikož $"(x)"[\frac{"(a)"}{\prime x\ \prime}][\frac{"(b)"}{\prime x\ \prime}] =$ "$\big((a)\big)$" a $"(x)"[\frac{"(b)"}{\prime x\ \prime}][\frac{"(a)"}{\prime x\ \prime}] =$ "$\big((b)\big)$". Pokud pro množinu substitucí $M = \left\{[\frac{t_1}{e_1}][\frac{t_2}{e_2}] \dots [\frac{t_i}{e_i}]\right\}$ platí, že $e_1, e_2 \dots e_i \in B,\ t_1, t_2 \dots t_i \in \Delta(C),\ B \cup C = \{\ \}$ a $e_1 \neq e_2 \neq \dots \neq e_i$, pak nezáleží na pořadí aplikace substitucí M na jakýkoli prvek z $\Delta(B \cup C)$.

Nyní formálně zavedeme pojem substituční množina.

Definice 39: Doména možných substitucí registrů z množiny registrů T za množinu atomů A nezahrnující tyto registry, $T \cap \mathrm{A} = \{\ \}$, se nazývá substituční množina a je konstruována jako $\mathrm{P}(T, \mathrm{A}) = \left\{\left[\frac{t}{e}\right], t \in \Delta\mathrm{(A)}, e \in T\right\}\backslash$

Uspořádaná substituční množina kromě relace ostrého uspořádání bude mít i tu vlastnost, že každý registr z množiny registrů může být substituován nanejvýš jednou. Definice uspořádané substituční množiny následuje.

Definice 40: Doménu všech uspořádaných substitučních množin nad množinou atomů Σ a množinou symbolů registrů *T* budeme značit a definovat jako

$$\mathrm{M}(T,\mathrm{A}) = \left\{\left(\left[\frac{t_1}{e_1}\right],\left[\frac{t_2}{e_2}\right],\dots\left[\frac{t_i}{e_i}\right]\right) : \left[\frac{t}{e}\right]\left[\frac{t_2}{e_2}\right]\left[\frac{t\prime}{e\prime}\right] \in \mathrm{P}(T,\mathrm{A}), \forall m,n \in \langle 1..i\rangle (e_m = e_n \to m = n)\right\}$$

Poslední z definic uvádíme pro operace, které aplikují jednu substituci nebo množinu substitucí na celý plán.

Definice 41: Operace substituce plánu je zobrazení $\Pi(\mathrm{O}(\mathrm{A})) \times \wp(\mathrm{P}(A)) \to \Pi(\mathrm{O}(\mathrm{A}))$ a je definovaná jako

$$\forall (o_1, o_2 \dots o_n) \in \Pi(\mathrm{O}(\mathrm{A})), \left[\frac{t}{e}\right] \in \mathrm{P}(T,\mathrm{A})\left(((o_1, o_2 \dots o_n)\left[\frac{t}{e}\right] = \left(o_1\left[\frac{t}{e}\right], o_2\left[\frac{t}{e}\right] \dots o_n\left[\frac{t}{e}\right]\right))\right)$$

Poslední dvě definice týkající se substitucí definují aplikace uspořádaných množin substitucí na akce a plány.

Definice 42: Operace aplikace uspořádané substituční množiny na akci je zobrazení $\mathrm{O}(\Sigma) \times \mathrm{M}(T,\Sigma) \to \mathrm{O}(\Sigma)$ takové, že uspořádaná substituční množina je aplikována na argument akce.

Definice 43: Operace aplikace uspořádané substituční množiny na plán je zobrazení $\Pi(\mathrm{O}(\mathrm{A})) \times \mathrm{M}(T,\mathrm{A}) \to \Pi(\mathrm{O}(\mathrm{A}))$, které budeme zapisovat jako $p\,\mu^{T,A}$, $p \in \Pi(\mathrm{O}(\mathrm{A})), \mu^{\mathrm{T,A}} \in M(T,\mathrm{A})$ a je definované tak, že na každou akci z plánu *p* bude aplikována substituční množina $\mu^{\mathrm{T,A}}$.

$$\forall (o_1, o_2 \dots o_n) \in \Pi(\mathrm{O}(\mathrm{A})), \mu^{\sim} : ((o_1, o_2 \dots o_n)\mu^{\mathrm{T,A}} = (o_1\mu^{\mathrm{T,A}}, o_2\mu^{\mathrm{T,A}} \dots o_n\mu^{\mathrm{T,A}}))$$

Po teoretickém výkladu je vhodné ilustrovat principy substitucí v ALLL systémech na příkladu.

Příklad 12: Uvažujme plán ve tvaru "$\left((+(a\ \tau)),\ (@(\sim \tau')), (-(a\ \tau)), (-\tau')\right)$". Substituční množina nechť je $\left\{\left[\frac{"(a)"}{'\tau'}\right],\left[\frac{"\tau"}{'\tau'\ '}\right]\right\}$. Po aplikaci této množiny na plán vznikne plán "$\left((+(a\ (a))), (@(\sim \tau)), (-(a\ (a))), (-\tau)\right)$". Po provedení první akce a aplikaci stejné substituční množiny, tj. po provedení $\left((@(\sim \tau)), (-(a\ (a))), (-\tau)\right)\left\{\left[\frac{"(a)"}{"\tau"}\right],\left[\frac{"\tau"}{"\tau'\ "}\right]\right\}$ by vznikl plán $\left((@(\sim (a))), \left(-(a\ (a))\right), (-(a))\right)$. V případě změny substituční množiny na $\left\{\left[\frac{"(b\ c)"}{"\tau"}\right],\left[\frac{"\tau"}{"\tau'\ "}\right]\right\}$ bychom před vykonáním druhé akce měli plán $\left(\left(@(\sim (b\ c))\right), \left(-(a\ (a))\right), (-(b\ c))\right)$.

3.6 ALLL systém s lineárními plány

První výpočetní systém, který v této práci uvedeme, pracuje pouze s plány, které reprezentuje lineární struktura akcí báze bez uspořádání. Ukážeme, jak takové plány vypadají a jak probíhá jejich aplikace na systém báze. To má demonstrovat princip provádění výpočtů v ALLL systémech, který bude podobný i u ostatních systémů, které si představíme dále.

3.6.1 Lineární plány

Přestože plán je vždy syntakticky n-tice, tedy lineární struktura definovaná jako množina $\Pi(\mathrm{O}(\mathrm{A}))$ vytvořená nad množinou operací $\mathrm{O}(\mathrm{A})$, budou se od sebe lišit lineární plány a strukturované plány, které mohou reprezentovat hierarchii plánů a podplánů. Lineární plány mají za prvky pouze akce nad

bází, kdežto později strukturované plány budou obsahovat i jiné akce, které umožní právě vytváření hierarchické struktury. Lineární plány mají vždy také jednu speciální akci, kterou je buď akce úspěšně, nebo neúspěšně ukončeného plánu. Podle těch později určíme, jestli systém plán přijal, nebo nikoli. Nyní tedy definujme lineární plány formálně. Uvedeme nejdřív definici lineárních plánů jako takových, poté definici lineárních plánů pro báze bez uspořádání.

Definice 44: Množina lineárních plánů nad nějakou množinu akcí $O(A)$ je značena $\mathrm{L}\big(O(A)\big)$ a jedná se buď o prvek s jedním atomem zastupujícím akci úspěšné ukončeného nebo neúspěšně ukončeného plánu $\mathrm{L}\big(O(A)\big) \subseteq \{"(\Omega_F)", "(\Omega_T)"\}$, nebo o prvek s více atomy. Lineární prvky delší než jedna jsou lineární plány končící akcí úspěšně vykonaného plánu, $\mathrm{L}\big(O(A)\big) \subseteq \{o \cdot "(\Omega_T)", o \in D\big(\mathrm{O}(A)\big)\}$.

Definice lineárních plánů rozšíříme o definici zápisů plánů pro bázi bez uspořádání konkrétních délek.

Definice 45: Množina lineárních plánů pro systém báze bez uspořádání β_L s jedním aktivním registrem délky 0 je $\mathrm{L}^0\left(\mathrm{O}_L\left(\{\tau\},\Sigma\right)\right) = \{"(\Omega_T)", "(\Omega_F)"\}$. Pro délky n≥1 a pro nějakou množinu akcí báze $O_L\left(\{\tau\},\Sigma\right)$ je $\mathrm{L}^n\left(\mathrm{O}_L\left(\{\tau\},\Sigma\right)\right) = \left\{o \cdot "(\Omega_T)", o \in D^n\left(\mathrm{O}_L\left(\{\tau\},\Sigma\right)\right)\right\}$. Množina lineárních plánů pro systém báze bez uspořádání libovolné délky je $\mathrm{L}\left(\mathrm{O}_L\left(\{\tau\},\Sigma\right)\right) = \sum_{i\geq 0} \mathrm{L}^i\left(\mathrm{O}_L\left(\{\tau\},\Sigma\right)\right)$.

3.6.1.1 Princip provádění lineárních plánu nad bází bez uspořádání

Pokud budeme mít bázi bez uspořádání $\beta_L = \left(\Sigma,\Gamma,\gamma_0,\ltimes_{L,I},\ltimes_{L,O}\right)$ a plán $\pi \in \mathrm{L}(\mathrm{O}_L\left(\{\tau\},\Sigma\right))$ složený z akcí nad bází s jedním registrem $\mathrm{O}_L\left(\{\tau\},\Sigma\right)$, pak tento proces můžeme aplikovat na systém, jehož součástí je tato báze. Báze se nachází v nějakém počátečním stavu a ten může být akcemi modifikován podle toho, jak jsme definovali její vnitřní přechodové funkce. Kromě změny stavu báze dojde po provedení akce i ke změně plánu k vykonání a to jednak tím, že bude jedna nebo všechny akce z plánu odebrány, a dále také možnou substitucí některých části plánu za aktuální stav výpočtu. Stav výpočtu pak bude zobrazovat výstupní funkce báze.

Aplikaci plánu na bázi provedeme postupnou aplikaci akcí plánu na stavy báze. Z kapitoly o systému báze víme, že aplikací akce provedeme přechod stavu systému, v tomto případě přechod z jednoho stavu báze do druhého. Pokud máme lineární plán $\pi = \left((\odot_1,p_1)\,(\odot_2,p_2)\ldots(\odot_m,p_m)\,(\Omega_T)\right)$ a aplikujeme ho postupně na nějaký stavy báze γ, pak by výsledným stavem báze byl stav zobrazený vnitřní přechodovou funkcí následovně

$$\gamma' = \left(\left(\left(\gamma \ltimes_{L,I} (\odot_1,p_1)\right) \ltimes_{L,I} (\odot_2,p_2)\right) \ldots \ltimes_{L,I} (\odot_m,p_m)\right)$$

Výstupní funkce po provedení plánu by zobrazila transformovaný stav báze

$$n = \left(\left(\left(\gamma \ltimes_{L,I} (\odot_1,p_1)\right) \ltimes_{L,I} (\odot_2,p_2)\right) \ldots \ltimes_{L,O} (\odot_m,p_m)\right)$$

Jediný rozdíl mezi výše uvedenými vztahy je ten, že poslední akce je zobrazena výstupní funkcí, namísto vnitřní přechodové funkce.

3.6.1.2 Výpočet aplikací lineárního plánu na báze bez uspořádání

Výpočtem budeme rozumět transformaci počáteční konfigurace obsahující vstupní data, stav báze a plán na nějakou konečnou konfiguraci, ve které bude plán zpracován, báze bude v nějakém koncovém stavu a aktuální stav výpočtu bude udávat výsledek výpočtu.

Jak už jsme uvedli, vykonávání jednotlivých akcí plánu nebude transformovat jenom stav báze, ale i plán samotný. K transformaci plánu dojde po provedení každé z akcí. Navíc plán by měl být schopen zpracovat výsledek naposledy provedené akce. U agentních systému se za takový výsledek považuje výsledek dotazu do báze na přítomnost nějaké informace, přijetí zprávy, nebo změna struktury záměrů. V ALLL systémech bude výsledkem každé akce obecný prvek nad množinou atomů systému a tím je dán i aktuální stav výpočtu plánem, který je výsledkem posledně provedené akce. Aktuální stav výpočtu se poté může substituovat v některých částech plánů za symboly registrů. K tomu použijeme operace substitucí definované v kapitole 3.5.3.

V systémech s lineárními plány nebude zatím docházet ani ke komunikaci mezi systémy, ani k provádění akcí pro změny struktur plánů. Všechny akce, ze kterých tyto plány mohou být sestaveny, jsou pouze akcemi nad bází. Jedinou změnou plánu kromě odstranění akce či zbytku plánu po provedení akce bude provedení operace substituce na další akci plánu před jejím provedením. Pokud budeme chtít aplikovat další z akcí plánu na systém, nahradíme v této akci pomocí operace substituce výskyty symbolu registru za hodnotu aktuálního stavu výpočtu.

Na transformaci konfigurací opět použijeme již definované funkce báze $\ltimes_{L,I}$ a $\ltimes_{L,O}$. Před aplikací akcí dojde k patřičným substitucím. Navíc kromě změny stavu báze dojde v každém kroku i ke změně aktuálního stavu výpočtu, který budeme označovat znakem n a apostrofy. Postup aplikace plánu by pak byl následující.

$$\gamma' = (\gamma \ltimes_{L,I} (\odot_1, p_1)[n])$$

$$n' = (\gamma \ltimes_{L,O} (\odot_1, p_1)[n])$$

$$\gamma'' = (\gamma' \ltimes_{L,I} (\odot_2, p_2)[n'])$$

$$n'' = (\gamma' \ltimes_{L,O} (\odot_2, p_2)[n'])$$

…

$$\gamma^m = (\gamma^{m-1} \ltimes_{L,I} (\odot_m, p_m)[n^{m-1}])$$

$$n^m = (\gamma^{m-1} \ltimes_{L,O} (\odot_m, p_m)[n^{m-1}])$$

Tento princip aplikace akcí na báze je společný pro všechny systémy, které budou používat ALLL báze. Konkrétní chování systémů bude ale o něco složitější a budeme jej definovat pro každý systém zvlášť, nicméně u každého z nich nalezneme provádění substitucí registrů v akcích za aktuální stav výpočtu před aplikováním akcí na systém.

3.6.1.3 Poznámka k provádění a přijímání plánů obecnými ALLL systémy

Ukázali jsme si, že ALLL systém provádí výpočet transformací konfigurací systémů na základě zadaného plánu. Počáteční konfigurace je dána počátečními konfiguracemi systémů a jejich

podsystémů a plánem k aplikování. V otevřených systémech budou konfigurace obsahovat i vstupní a výstupní stavy, díky nimž bude možné z ALLL prvků vytvářet větší, hierarchicky nadřazené systémy, ve kterých budou tyto prvky komunikovat předáváním zpráv.

Plán je systémem přijímán, pokud po jeho aplikování nebude výsledná konfigurace systému obsahovat plán s akcí neúspěšně provedeného plánu. V naznačeném fungování systému však není řešena situace, kdy nějaká z akcí je zobrazena jako neúspěšná. Aktuálním stavem výpočtu je po nějakou část výpočtu symbol neúspěšné operace, ten však může být změněn a nahrazen nějakým obecným prvkem a celý plán po své aplikaci na systém skončí úspěšně. Zdali je takový plán přijat či nikoli se bude lišit v jednotlivých systémech. Jak tomu bude u nejjednoduššího z nich, systému nazvaného lineární ALLL systém s bází bez uspořádání, zjistíme v následující kapitole.

3.6.2 Struktura lineárního ALLL systému s bází bez uspořádání a jeho konfigurace

První z ALLL systémů, který si ukážeme, se příliš neliší strukturou ani činností od systému s bází popsaného v předcházejících odstavcích. Jeho struktura je následující.

Definice 46: Lineární ALLL systém $\mathcal{S}_{\mathrm{L}}$ s bází bez uspořádání $\beta_L = (\Sigma, \Gamma, \gamma_0, \ltimes_{L,I}, \ltimes_{L,O})$ je uspořádaná trojice $\mathcal{S}_{\mathrm{L}} = (\beta_L, \vdash_L, n_0)$, kde β_L je báze systému, $\vdash_L$ jeho přechodová relace a n_0 je počáteční stav výpočtu systému.

V této kapitole budeme předpokládat, že systém $\mathcal{S}_{\mathrm{L}}$ je lineární ALLL systém s bází bez uspořádání a budeme toto symbolické označení používat namísto víceslovného názvu.

Systém $\mathcal{S}_{\mathrm{L}}$ obsahuje bázi β_L jako podsystém, stav výpočtu ve formě obecného prvku nad abecedou báze, a přechodovou relaci systému $\vdash_S$, která transformuje stav celého systému tak, jak mění stav výpočtu systému a vnitřní stav báze. Onen celkový stav společně s aplikovaným plánem budeme nazývat konfigurací systému.

Definice 47: Konfigurace systému $\mathcal{S}_{\mathrm{L}}$ je trojice $\mathrm{C}_L = (\gamma, \pi, n)$, kde první prvek je z domény báze, $\gamma \in \Gamma$, druhým je lineární plán, $\pi \in L(\mathrm{O}_L\ (\{\tau\}, \Sigma))$ a třetí prvek je aktuální stav výpočtu, $n \in \Delta(\Sigma)$

Definice 48: Doména konfigurací systému $\mathcal{S}_H$ je podle předchozí definice $Dom_{\mathrm{C}}(\mathcal{S}_L) = \{(\gamma, \pi, n) : \gamma \in \Gamma, \pi \in L(\mathrm{O}_L\ (\{\tau\}, \Sigma)), n \in \Delta(\Sigma)\}$

V jaké konfiguraci se systém nachází je dáno jednak stavem jeho báze jako podsystému a dalšími stavy, kterými jsou aktuální hodnota výpočtu, což lze chápat jako hodnotu uloženou v jediném registru systému a také plán. Speciálním případem budou konfigurace, kde plán obsahuje pouze akci značící ukončení plánu, a to buď úspěšné, nebo neúspěšné.

Definice 49: Konfigurace $(\gamma, \pi, n) \in Dom_{\mathrm{C}}(\mathcal{S}_L)$ obsahující plán délky jedna, což podle definice 45 znamená $\pi \in \mathrm{L}^0\left(\left(\mathrm{O}_L\ (\{\tau\}, \Sigma)\right)\right)$, kde $\mathrm{L}^0\left(\left(\mathrm{O}_L\ (\{\tau\}, \Sigma)\right)\right) = \{"(\Omega_T)", "(\Omega_F)"\}$, je koncová konfigurace.

V dalším textu budeme používat symbol $'\Omega'$ zastupující obě akce ukončení plánu, tj. $'\Omega' \in \{'\Omega_T', '\Omega_F'\}$ a koncové konfigurace budou obsahovat plán s tímto symbolem.

Výpočet v systému probíhá jako posloupnost konfigurací, kde na počátku je počáteční konfigurace systému a poslední z konfigurací bude nějaký koncový stav. Počáteční konfigurace je dána počátečním stavem systému a aplikovaným plánem na tento systém.

Definice 50: Pro systém $\mathcal{S}_{\mathrm{L}}$ a plán $\pi \in L(\mathrm{O}_L(\{\tau\},\Sigma))$ je trojice $\mathrm{C}_{0L}^{\pi} = (\gamma_0, \pi, n_0) \in Dom_{\mathrm{C}}(\mathcal{S}_L)$ počáteční konfigurací.

Počáteční konfigurace je dána počátečním stavem báze, počátečním stavem výpočtu a aplikovaným plánem na systém. Formálně by mohla být definována funkce, která pro každý systém a plán zobrazí takovou konfiguraci, ve které by byly počáteční stav báze a počáteční hodnota výpočtu dány systémem. Definice 50 je ale pro účely dalšího textu dostatečná a umožní nám přejít k podkapitole, ve které budeme definovat chování $\mathcal{S}_{\mathrm{L}}$ systému.

3.6.3 Přechodová relace $\mathcal{S}_{\mathrm{L}}$ systémů

Nyní představíme přechodovou relaci systému $\mathcal{S}_{\mathrm{L}}$, která bude modelovat chování tohoto systému. Podle definice konfigurace bude systém pracovat s jednou bází, vstupními daty ve formě aktuálního stavu výpočtu a s plánem. Přechodová relace je pak zobrazením mezi dvěma konfiguracemi $\mathcal{S}_{\mathrm{L}}$ systému.

Definici této relace provedeme pomocí několika pravidel, které uvádějí změnu konfigurace systému úspěšným či neúspěšným provedením jednotlivých akcí. Relace transformuje jednu konfiguraci do druhé způsobem uvedeným pod čarou, pokud jsou splněna pravidla uvedená nad čarou. To je v souladu s tím, jak obvykle definujeme operační sémantiku výpočetního systému [75].

Definice 51: Relace $\vdash_s$ je přechodovou relací systému $\mathcal{S}_{\mathrm{L}}$ a je zobrazením $\vdash_s : \Gamma \times \mathrm{L}(\mathrm{O}_L(\{\tau\},\Sigma)) \times \Delta(\Sigma) \to \Gamma \times \mathrm{L}(\mathrm{O}_L(\{\tau\},\Sigma)) \times \Delta(\Sigma)$. Tato relace je definována pomocí následujících tří pravidel.

(APPLY) $$\frac{\gamma \ltimes_{L,O} (\odot,p)[n]=n',\ (\odot,p)\in L\left(\mathrm{O}_L(\{\tau\},\Sigma)\right), n' \neq' \perp', \odot \neq' \sigma'}{(\gamma,((\odot,p))\cdot o,n) \vdash_L (\gamma \ltimes_{L,I} (\odot,p)[n], o, \mathrm{n}')}$$

(TEST) $$\frac{\gamma \ltimes_{L,O} (\odot,p)[n]=n',\ (\odot,p)\in L\left(\mathrm{O}_L(\{\tau\},\Sigma)\right), n' \neq' \perp', \odot =' \sigma'}{(\gamma,((\odot,p))\cdot o,n) \vdash_L (\gamma, o, \mathrm{n}')}$$

(FAIL) $$\frac{\gamma \ltimes_{L,O} (\odot,p)[n]=' \perp'}{(\gamma,((\odot,p))\cdot o,n) \vdash_L (\gamma,"(\Omega_F)",\mathrm{n})}$$

Pomocí uvedených pravidel můžeme transformovat počáteční konfiguraci systému na nějakou koncovou konfiguraci. Pravidla APPLY a TEST by šla sloučit do jednoho, jelikož vstupní funkce báze $\ltimes_{L,I}$ nemění její stav v případě operace testování, a tak by v pravidle TEST mohla být uvedena stejně jako v pravidle APPLY, ale pro rozlišení akcí transformující a netransformující stav báze uvádíme pravidla dvě. Výpočet končí, respektive nedochází k další modifikaci ani stavu báze, ani aktuálního stavu výpočtu, pokud plán v konfiguraci obsahuje pouze akci ukončeného plánu, což je splněno pro koncové konfigurace. Pokud je součástí plánu nějaká jiná proveditelná, akce, lze aplikovat právě jedno z uvedených pravidel.

Přechodová relace zaslouží bližší studium. Z definic pravidel je patrné, že pouze jedno z pravidel je aplikovatelné pro každý stav báze. Které z pravidel se bude aplikovat, závisí na tom, zdali plán je nebo není ukončen. Pokud není ukončen, pak volba pravidla závisí na tom, zdali operace k provedení skončí či neskončí úspěšně a dále na tom, zdali se jedná o operaci výběru či nikoli. Navíc každé

pravidlo zobrazuje konfigurace na jednu následující konfiguraci a tedy tento systém je deterministický. Toto formálně uvádí následující lemma.

Lemma 17: Přechodová relace $\vdash_L$ je zobrazení.

Důkaz: Důkaz bude založen na tom, že pouze jedno pravidlo je použitelné pro jakoukoli konfiguraci systému a každá konfigurace je v relaci jen s jednou následující konfigurací. Pravidla jsou použitelná pro plán obsahující alespoň jednu akci kromě akce ukončeného plánu ve třech případech, a to za následujících podmínek: $Cond_1 \equiv \gamma \ltimes_{L,O} (\odot, p) \neq ' \perp '$, $\odot \neq '\sigma'$, $Cond_2 \equiv \gamma \ltimes_{L,O} (\odot, p) \neq ' \perp ', \odot = '\sigma'$ a $Cond_3 \equiv \gamma \ltimes_{L,O} (\odot, p) = ' \perp '$. Aby byly proveditelné alespoň dvě pravidla zároveň, musela by být pravdivá formule $(Cond_1 \wedge Cond_2) \vee (Cond_1 \wedge Cond_3) \vee (Cond_2 \wedge Cond_3)$, což není.

Další Lemma uvádí, že pro každou akci z množiny akcí nad bází bez uspořádání je použitelné nějaké pravidlo definující $\vdash_L$.

Lemma 18: Zobrazení $\vdash_L$ je definováno pro každou akci z $O_L(\{\tau\}, \Sigma)$.

Důkaz: Podle definice 17 je prvkem množiny $O_L(\{\tau\}, \Sigma_2)$ dvojice, kde první prvek dvojice je symbol akce '+', '-' či 'σ' a druhým prvkem je argument akce. Pokud je akce úspěšná po aplikaci na bázi, je použito pravidlo (TEST) pro symbol σ a pravidlo (APPLY) pro ostatní dva symboly. Pokud je aplikace neúspěšná, je definováno pravidlo (FAIL) pro jakýkoliv symbol a argument akce.

Další Lemma hovoří o rovnosti přechodových funkcí pro některé $\mathcal{S}_\mathrm{L}$ systémy.

Lemma 19: Pro dva systémy $\mathcal{S}_{\mathrm{L}1} = \left((\Sigma_1, \Gamma_1, \gamma_{01}, \ltimes_{L,I1}, \ltimes_{L,O1}), \vdash_{L1}, n_{01}\right)$ a $\mathcal{S}_{\mathrm{L}2} = \left((\Sigma_2, \Gamma_2, \gamma_{02}, \ltimes_{L,I2}, \ltimes_{L,O2}), \vdash_{L2}, n_{02}\right)$ platí, že pokud $\Sigma_1 = \Sigma_2$ a $\Gamma_1 = \Gamma_2$, pak i $\vdash_{L1} = \vdash_{L2}$

Důkaz: Systém obsahuje báze bez uspořádání a podle Lemmat 11 a 12 platí pro funkce těchto bází rovnosti $\ltimes_{L,I1} = \ltimes_{L,I2}$ a $\ltimes_{L,O1} = \ltimes_{L,O2}$. Definice přechodové relace uvádí definiční obor v obou případech jako domény jejich konfigurací, tedy $Dom_\mathrm{C}(\mathcal{S}_{L1}) = \{(\gamma_1, \pi_1, n_1) : \gamma_1 \in \Gamma_1, \pi_2 \in L(\mathrm{O}_{L1}(\{\tau\}, \Sigma_1)), n_2 \in \Delta(\Sigma_1)\}$ a $Dom_\mathrm{C}(\mathcal{S}_{L2}) = \{(\gamma_2, \pi_2, n_2) : \gamma_2 \in \Gamma_2, \pi_2 \in L(\mathrm{O}_{L2}(\{\tau\}, \Sigma_2)), n_2 \in \Delta(\Sigma_2)\}$. Na základě rovností uvedených jako předpoklady Lemmy musí platit $Dom_\mathrm{C}(\mathcal{S}_{L1}) = Dom_\mathrm{C}(\mathcal{S}_{L2})$. Pro každé dvě konfigurace z těchto domén má platit $\mathrm{C} \vdash_{L1} \mathrm{C}'$ pokud a pouze pokud $\mathrm{C} \vdash_{L2} \mathrm{C}'$. Pravidla, která tyto funkce definují, podmiňují existenci relací mezi konfiguracemi výsledky funkcí $\ltimes_{L,O1}$ resp. $\ltimes_{L,O2}$ a množinami $L\left(\mathrm{O}_L(\{\tau\}, \Sigma_1)\right)$ a $L\left(\mathrm{O}_L(\{\tau\}, \Sigma_2)\right)$. Tyto funkce a množiny se podle předpokladů pro oba systémy rovnají, proto i pravidla pro přechodové relace vytváří v obou případech stejné zobrazení.

Lemma 20: Pro jakoukoliv koncovou konfiguraci $\mathrm{C}_L, \mathrm{C}_L \in \{(\gamma, "(\Omega)", n) \in Dom_\mathrm{C}(\mathcal{S}_L)\}$, neexistuje konfigurace $\mathrm{C}_L' \in Dom_\mathrm{C}(\mathcal{S}_L)$ taková, že $\mathrm{C}_L \vdash_L \mathrm{C}_L'$

Důkaz: V definici přechodové funkce $\vdash_L$ ani jedno ze tří přechodových pravidel není aplikovatelné na konfigurace obsahující jako plán "(Ω)" a tedy pro konfigurace uvedené v Lemmatu není přechodová relace definována.

Lemma 21: Každá konfigurace obsahuje buďto plán začínající akcí z $O_L(\{\tau\}, \Sigma)$, nebo se jedná o koncovou konfiguraci.

Důkaz: To plyne z definic 45, 47 a 49. Plán v konfiguraci je buďto délky jedna, nebo delší. V případě plánu délky jedna se jedná o koncovou konfiguraci. Pro plán větší délky platí, že patří do množiny $\left\{\pi \cdot "(\Omega_T)", \pi \in D^n\left(\mathrm{O}_L\left(\{\tau\},\Sigma\right)\right)\right\}$ kde $\mathrm{n}\geq 1$ a první akce je prvním podprvkem lineárního prvku z $D^n\left(\mathrm{O}_L\left(\{\tau\},\Sigma\right)\right)$, tedy z $\mathrm{O}_L\left(\{\tau\},\Sigma\right)$.

Pro diskusi o dosažitelných stavech z nějakého stavu použijeme tranzitivní uzávěr přechodové relace.

Definice 52: Tranzitivní uzávěr $\vdash_L$ funkce budeme značit $\vdash_L^*$

Tranzitivní uzávěr přechodové relace systému lze chápat jako relaci mezi konfigurací a všemi konfiguracemi, do kterých se systém může během výpočtu dostat. Všechny takové konfigurace jsou součástí výpočtu a posledním krokem výpočtu je nějaká koncové konfigurace. Takovou posloupnost budeme chápat jako běh systému po aplikaci nějakého plánu.

3.6.4 Aplikace plánu a běh $\mathcal{S}_\mathrm{L}$ systému

Systém přijímá či nepřijímá plán z množiny lineárních plánů podle toho, zda jeho vykonávání skončí jeho transformací na koncovou konfiguraci obsahující plán s akcí úspěšného nebo neúspěšného ukončení plánu. Vykonání plánu bude znamenat postupnou transformaci počáteční konfigurace aplikací akcí z tohoto plánu až po dosažení nějaké z koncových konfigurací, ze kterých již dle Lemmy 19 nelze dosáhnout další konfigurace.

Definice 53: Posloupnost $(\mathrm{C}_{L1},\ldots,\mathrm{C}_{Lm})$, kde $\mathrm{C}_{L1}=\mathrm{C}_{0L}^{\pi}=(\gamma_0,\pi,n_0)$, je běh systému $\mathcal{S}_\mathrm{L}$, pokud $\pi \in L\left(\mathrm{O}_L\left(\{\tau\},\Sigma\right)\right)$ a pokud platí $\forall i,j,1\leq i<m\colon(\mathrm{C}_{Li}\vdash_L \mathrm{C}_{Li+1})$

Pro jednotlivé konfigurace v běhu systému také platí formule $\forall i,j,0\leq i<n,0\leq j\leq n\colon\left(\mathrm{C}_{Li}\vdash_S^* \mathrm{C}_{Lj}\right)$, která říká, že mezi každou předcházející a následující konfigurace v běhu systému platí relace $\vdash_L^*$.

Lemma 22: Každému systému $\mathcal{S}_\mathrm{L}$ s počáteční konfigurací C_{0L}^{π} odpovídá právě jeden koncový běh systému $(\mathrm{C}_{0L}^{\pi},\ldots,\mathrm{C}_{\mathrm{Ln}})$ takový, že C_{Ln} je koncová konfigurace a $\mathrm{C}_{\mathrm{Ln}-1}$ není koncová konfigurace.

Důkaz: Indukcí budeme sledovat běhy různých délek systému $\mathcal{S}_\mathrm{L}$. V jakékoli počáteční konfiguraci $\mathrm{C}_{\mathrm{L1}}=(\gamma_0,\pi,\mathrm{n}_0)$ existuje pouze jeden běh délky jedna obsahující pouze tuto počáteční konfiguraci. Běh délky n+1 vznikne přidáním konfigurací v relaci $\vdash_L$ s poslední konfigurací z běhů délky n. Pokud existuje jen jeden běh délky n, pak existuje jen jedna taková konfigurace $\mathrm{C}_{\mathrm{Ln}}=(\gamma,\pi,n)$ a jelikož podle Lemmy 17 je $\vdash_L$ zobrazení, pak existuje jedna konfigurace $\mathrm{C}_{\mathrm{Ln+1}}$, pokud $\pi=(o)\cdot\pi',o\in \mathrm{O}_L\left(\{\tau\},\Sigma\right),\pi\in L(O_L\left(\{\tau\},\Sigma\right))$, nebo neexistuje ani jedna taková konfigurace, pokud je C_{Ln} koncovou konfigurací. Indukcí lze ukázat, že existuje právě jeden běh C délky *m+1* obsahující na počátku běh C′ délky *m* s nekoncovou konfigurací jako poslední konfigurací běhu C′. Pokud běh C′ obsahuje koncovou konfiguraci jako svou poslední konfiguraci, neexistuje žádný další běh délky *m+i*, kde *i≥1*, který by na počátku obsahoval běh C′. Pro běh délky jedna s počáteční konfigurací proto existuje jen jeden koncový běh systému.

Jakou koncovou konfigurací aplikací plánu na systém dosáhneme, to zobrazuje funkce z následující definice.

Definice 54: Funkce $\mathrm{A}(\mathcal{S}_\mathrm{L}, \pi)$ pro daný systém a plán je struktura $\mathrm{A}: Dom(\mathcal{S}_L) \times L(O_L\ (\{\tau\}, \Sigma)) \rightarrow Dom_\mathrm{C}(\mathcal{S}_L)^*$, která zobrazí běh s jednou koncovou konfigurací $\mathrm{A}(\mathcal{S}_\mathrm{L}, \pi) = (\mathrm{C}_{0L}^{\pi}, \dots, \mathrm{C}_{\mathrm{L}\Omega})$ a tuto funkci nazýváme aplikací plánu π na systém $\mathcal{S}_\mathrm{L}$.

Následující Lemma vymezuje délku koncového běhu systému, to znamená počet kroků systému od nějaké počáteční konfigurace po koncovou konfiguraci, která je v běhu systému poslední.

Lemma 23: Koncový běh vzniklý aplikací plánu $\pi \in L^m(O_L\ (\{\tau\}, \Sigma))$ na systém $\mathcal{S}_L$ s počáteční konfiguraci C_{0L}^{π} je posloupnost délky menší nebo rovné *m+1*.

Důkaz: Pro konfiguraci s plánem délky 1, který je podle definice 45 $\pi^0 \in \mathrm{L}^0\left(O_L\ (\{\tau\}, \Sigma)\right) = \{"(\Omega_T)", "(\Omega_F)"\}$, a která je koncovou konfigurací, neexistuje dle Lemmy 20 jiná dostupná konfigurace. Proto koncový běh má pro tento plán délku 1 a obsahuje jen tuto počáteční konfiguraci. Pro konfigurace s plány $\pi^m \in \mathrm{L}^m\left(O_L\ (\{\tau\}, \Sigma)\right) = \left\{o \cdot "(\Omega_T)", o \in D^m\left(O_L^{\{\tau\}}\right)\right\}, m \geq 1$ je podle Lemmat 17 a 18 definováno zobrazení pro každou z akcí z plánu a to podle jednoho z pravidel z definice 51. Pravidla (APPLY) a (TEST), pokud jsou použita, transformují plány v konfiguracích z $((\odot, p)) \cdot \pi$ na π, tedy plán z $\mathrm{L}^m\left(O_L\ (\{\tau\}, \Sigma)\right)$ na plán o jednu akci kratší z $\mathrm{L}^{m-1}\left(O_L\ (\{\tau\}, \Sigma)\right)$. Pravidlo (FAIL) transformuje jakýkoli plán na plán z $\mathrm{L}^0\left(O_L\ (\{\tau\}, \Sigma)\right)$. Pro plán $\pi^0 \in \mathrm{L}^0\left(O_L\ (\{\tau\}, \Sigma)\right)$ je délka koncové konfigurace $|\mathrm{A}(\mathcal{S}_\mathrm{L}, \pi^0)| = 1$, plán $\pi^1 \in \mathrm{L}^1\left(O_L\ (\{\tau\}, \Sigma)\right)$ po transformaci kterýmkoli pravidlem přejde v jednom kroku na plán délky 0 a tedy $|\mathrm{A}(\mathcal{S}_\mathrm{L}, \pi^1)| = |\mathrm{A}(\mathcal{S}_\mathrm{L}, \pi^0)| + 1 = 2$, pro plán délky m, $\pi^m \in \mathrm{L}^m\left(O_L\ (\{\tau\}, \Sigma)\right), m \geq 1$ pak obecně $|\mathrm{A}(\mathcal{S}_\mathrm{L}, ((\odot, p)) \cdot \pi)| = |\mathrm{A}(\mathcal{S}_\mathrm{L}, \pi^0)| + 1 = 2$ nebo $|\mathrm{A}(\mathcal{S}_\mathrm{L}, ((\odot, p)) \cdot \pi)| = |\mathrm{A}(\mathcal{S}_\mathrm{L}, \pi)| + 1$. V prvním případě Lemma 23 platí, v druhém případě indukcí také.

Dosavadní výklad nyní budeme ilustrovat několika příklady.

Příklad 13: Pro systém $\mathcal{S}_\mathrm{L}$ s bází, která má doménu postavenou z dvojic z množiny jmen a množiny roků narození, a která neobsahuje prázdnou množinu, mějme počáteční konfiguraci $\mathrm{C}_{0L}^{\pi} = (\gamma_0, \pi, n_0)$, $\gamma_0 = \{"(jan\ 1983\ m)", "(marie\ 1965\ z)", "(frantisek\ 1974\ m)"\}$. Počátečním stav výpočtu je $n_0 = "(null)"$ a plán je $\pi = "((+\ (jan\ 1962\ m))\ (-\ (\sim \sim z))\ \left(\sigma\ (jan \sim m)\right)\ \Omega_T)"$. Systém pak bude mít jeden běh s konfiguracemi:

$(\{"(jan\ 1983\ m)", "(marie\ 1965\ z)", "(frantisek\ 1974\ m)"\},$

$"\left((+\ (jan\ 1962\ m))\ (-\ (\sim \sim z))\right)\ \left(\sigma\ (jan\ m)\right)\ \Omega_T)", "(null)") \vdash_{LAPPLY}$

$(\{"(jan\ 1983\ m)", "(marie\ 1965\ z)", "(frantisek\ 1974\ m)", "(jan\ 1962\ m)\},$
$"((-\ (\sim \sim z))\ \left(\sigma\ (jan\ m)\right)\ \Omega_T)", "(null)") \vdash_{LAPPLY}$

$\left(\{"(jan\ 1983\ m)", "(frantisek\ 1974\ m)", "(jan\ 1962\ m)"\}, "\left(\left(\sigma\ (jan \sim m)\right)\ \Omega_T\right)", "(null)"\right) \vdash_{SLTEST}$

$(\{"(jan\ 1983\ m)", "(frantisek\ 1974\ m)", "(jan\ 1962\ m)"\}, "(\Omega_T)", "\ (jan\ 1983\ m)")$

Pro názornost jsme symboly přechodové relace v místech indexu doprovodili názvem použitého pravidla (APPLY, TEST). Všechny tři akce, z toho dvě manipulující s bází a jedna testovací, projdou v

systému úspěšně, a jeho koncová konfigurace je taková, že po provedení akce testování je aktuálním výsledkem výpočtu výsledek testu.

Příklad 14: Další příklad poslouží k ukázce šíření mezivýsledků použitím substitucí. Stav báze a počáteční stav výpočtu budou shodné, jako tomu bylo v předchozím příkladu, změní se ale aplikovaný plán. Konkrétně budeme mít plán se čtyřmi akcemi $\pi =$ "$\left(\left(\sigma\ (\sim \sim z)\right)\left(-\ (\sim \sim z)\right)\left(+\ (zeny\ \tau)\right)\ \Omega_T\right)$". Běh systému by tentokrát byl následující:

$$\left(\begin{matrix}\{"(jan\ 1983\ m)","(marie\ 1965\ z)","(frantisek\ 1974\ m)"\},\\ "\left(\left(\sigma\ (\sim \sim z)\right)\left(-\ (\sim \sim z)\right)\left(+\ (zeny\ \tau)\right)\ \Omega_T\right)","(null)"\end{matrix}\right) \vdash_{L\ TEST}$$

$$\left(\begin{matrix}\{"(jan\ 1983\ m)","(marie\ 1965\ z)","(frantisek\ 1974\ m)"\},\left(\left(\text{- }(\sim \sim \text{z})\right)\ (+\ (\text{zeny}\ \tau))\ \Omega_{\text{T}}\right),\\ "(marie\ 1965\ z)"\end{matrix}\right) \vdash_{LAPPLY}$$

$$\left(\{"(jan\ 1983\ m)","(frantisek\ 1974\ m)"\},"\left(\left(+\ (zeny\ \tau)\right)\ \Omega_T\right)","(marie\ 1965\ z)"\right), \vdash_{LAPPLY}$$

$$\left(\{\text{(jan 1983 m)"},"(frantisek\ 1974\ m)","\left(zeny\ (marie\ 1965\ z)\right)"\},"(\Omega_T)","(marie\ 1965\ z)"\right)$$

Poslední z příkladů ukáže změnu plánu po akci, která neuspěje.

Příklad 15: Pro stejný počáteční stav báze a stejný počáteční stav výpočtu budeme uvažovat plán $\pi =$ "$((+\ (jan\ 1962\ m))\ (-\ (\sim \sim m)))\ \left(\sigma\ (jan \sim m)\right)\left(+\ (muzi\ n)\right)\ \Omega_T)$". Běh systému je pak následující:

$$\left(\begin{matrix}\{"(jan\ 1983\ m)","(marie\ 1965\ z)","(frantisek\ 1974\ m)"\},\\ "((+\ (jan\ 1962\ m))(-\ (\sim \sim m))\ \ (\sigma\ (jan \sim m))\ \ (+\ (muzi\ \tau))\ \ \Omega_T\ ))\ ","(null)"\end{matrix}\right) \vdash_{LAPPLY}$$

$$\left(\begin{matrix}\{"\text{(jan 1983 m)"},"\text{(marie 1965 z)"},"\text{(frantisek 1974 m)"},"(jan\ 1962\ m)"\},\\ \left(\left(-\ (\sim \sim m)\right)\left(\sigma\ (jan \sim m)\right)\left(+\ (muzi\ \tau)\right)\ \Omega_T\right),"(null)"\end{matrix}\right), \vdash_{LAPPLY}$$

$$\left(\begin{matrix}\{"(marie\ 1965\ z)"\},\\ "\left(\left(\sigma\ (jan \sim m)\right)\left(+\ (muzi\ \tau)\right)\ \Omega_T\right)"\ ,"(null)"\end{matrix}\right) \vdash_{L\ FAIL}$$

$$(\{"(marie\ 1965\ z)"\},"(\Omega_F)"\ ,"(null)")$$

Po vložení údaje o Janovi a následném odstranění všech mužů z databáze dojde k přechodu do chybového stavu, jelikož výsledkem akce výběru je prázdná množina, která není součástí domény báze. Pokud by ale akce výběru proběhla ještě před akcí, která odstraní všechny trojice s posledním prvkem ‘m’ z báze, systém by provedl i poslední ze čtyř uvedených akcí. Právě kvůli výběru a kvůli tomu, že jednotlivé akce mohou potenciálně skončit nezdarem, můžeme vidět, že při aplikaci plánu na systém záleží na pořadí akcí v plánu.

3.6.5 Provádění výpočtu a přijímání plánu $\mathcal{S}_L$ systémy

O úspěchu aplikace plánu můžeme rozhodovat podle části koncové konfigurace, konkrétně podle výsledného plánu. Pokud bychom hleděli na tento systém jako na výpočetní systém, který má zadanou úlohu počátečním stavem báze počátečním stavem výpočtu, pak výsledkem výpočtu aplikovaným plánem je stav báze a stav výpočtu v koncové konfiguraci. V případě systému $\mathcal{S}_{\text{L}}$, který slouží pouze pro demonstraci části principů výpočtů pozdějšího komplexního systému, ale budeme za výsledek výpočtu považovat pouze přijetí plánu, v případě výsledného plánu s akcí úspěšného ukončení plánu, nebo nepřijetí plánu, v případě výsledného plánu s akcí neúspěšného ukončení plánu.

Definice 55: Výsledek aplikace plánu π na systém $\mathcal{S}_{\mathrm{L}}$ je dán funkcí $\mathcal{F}_{SL}(\mathcal{S}_{\mathrm{L}},\pi)\colon Dom_{\mathrm{C}}(\mathcal{S}_{\mathrm{L}}) \times O_L\ (\{\tau\},\Sigma) \to Dom_{\mathrm{C}}(\mathcal{S}_L)$ která je definována jako zobrazení systému a aplikovaného plánu na koncovou konfiguraci

$$\mathcal{F}_{SL}(\mathcal{S}_{\mathrm{L}},\pi) = \mathrm{C}_F\ iff\ \mathrm{A}(\mathcal{S}_{\mathrm{L}},\pi) = (\mathrm{C}_{0L}^{\pi},\mathrm{C}_2 \dots \mathrm{C}_F)$$

, přičemž rozumíme, že pokud $\mathrm{C}_F = (\gamma,(\Omega_T),n)$ pro nějaké γ a n, pak systém plán přijímá, a pokud $\mathrm{C}_F = (\gamma,(\Omega_F),n)$, pak systém plán nepřijímá.

Výpočet výsledku aplikace plánu se provádí nalezením koncové konfigurace tím, že je proveden běh systému podle vstupní konfigurace. Z trochu jiného pohledu bychom mohli na výpočet pohlížet tak, že provedením každé akce se transformuje jeden systém na jiný s jinou počáteční konfigurací, na který je aplikován plán vzniklý z původního plánu odebráním první akce, pokud byla tato akce provedena úspěšně, nebo na systém s původním vnitřním stavem a plánem s jedinou akcí a to akcí neúspěšného provedení plánu. Tato transformace bude hrát důležitou roli spíše u dalších systémů, než v případě systému $\boldsymbol{\mathcal{S}_L}$, ale pro pořádek si ji popíšeme.

Můžeme ukázat, že dva systémy $\boldsymbol{\mathcal{S}_L}$ mohou mít mezi sebou vztah, pokud lze jeden systém transformovat na druhý vykonáním nějaké akce. To znamená, že existuje akce z množiny akcí, která počáteční konfiguraci prvního systému zobrazí přechodovou funkcí na počáteční konfiguraci druhého systému. Pro tuto transformaci budeme definovat funkci, kterou nazveme výpočetním krokem $\boldsymbol{\mathcal{S}_L}$ systému.

Definice 56: Výpočetní krok $\boldsymbol{\mathcal{S}_L}$ systému je funkce $\triangleright_L : Dom_{\mathrm{C}}(\mathcal{S}_{\mathrm{L}}) \times O_L\ (\{\tau\},\Sigma) \to Dom_{\mathrm{C}}(\mathcal{S}_{\mathrm{L}})$ definovaná tak, že platí

$$\forall \left(\left(\Gamma,\Sigma,\gamma_{01},\ltimes_{L,I},\ltimes_{L,O}\right),\vdash_L,n_{01}\right),\left(\left(\Gamma,\Sigma,\gamma_{02},\ltimes_{L,I},\ltimes_{L,O}\right),\vdash_L,n_{02}\right) \in Dom_{\mathrm{C}}(\mathcal{S}_{\mathrm{L}}),$$

$$\exists o \in O_L\ (\{\tau\},\Sigma), \exists \pi_1,\pi_2 \in L\left(O_L\ (\{\tau\},\Sigma)\right) \cdot$$

$$\left(\left((\gamma_{01},(o)\cdot\pi_1,n_{01}) \vdash_L (\gamma_{02},\pi_2,n_{02})\right)\right.$$
$$\left.\leftrightarrow \left(\triangleright_L \left(\left(\left(\Gamma,\Sigma,\gamma_{01},\ltimes_{L,I},\ltimes_{L,O}\right),\vdash_L,n_{01}\right),o\right) = \left(\left(\Gamma,\Sigma,\gamma_{01},\ltimes_{L,I},\ltimes_{L,O}\right),\vdash_L,n_{02}\right)\right)\right)$$

Smyslem výpočetního kroku je nalézt $\mathcal{S}_{\mathrm{L}}$ systém, jehož počáteční konfigurace by odpovídala počáteční konfiguraci původního $\mathcal{S}_{\mathrm{L}}$ systému po aplikaci plánu, který obsahuje pouze první akci původního aplikovaného plánu. Výsledek aplikace původního plánu bez této první akce na zobrazený $\mathcal{S}_{\mathrm{L}}$ systém by pak odpovídal výsledku aplikace původního plánu na původní $\mathcal{S}_{\mathrm{L}}$ systém.

Lemma 24: Uveďme vztah mezi $\mathcal{S}_{\mathrm{L}}$ systémy a výsledky aplikací plánů na ně následující formuli tak, že pokud jeden $\mathcal{S}_{\mathrm{L}}$ systém přechází na druhý aplikací nějaké akce, pak jsou jejich koncové konfigurace stejné, pokud je na první aplikován nějaký plán a na druhý plán po provedení své první akce.

$$\forall \boldsymbol{\mathcal{S}_{L1}\mathcal{S}_{L2}} \in Dom(\boldsymbol{\mathcal{S}_L})\ \forall o \in O_L\ (\{\tau\},\Sigma)\ \forall \pi \in L\left(O_L\ (\{\tau\},\Sigma)\right)\ \exists \pi \in L\left(O_L\ (\{\tau\},\Sigma)\right)$$

$$\cdot \begin{pmatrix} (\triangleright_L\ (\boldsymbol{\mathcal{S}_{L1}},o) = \boldsymbol{\mathcal{S}_{L2}}) \wedge \left(\mathrm{C}_{0L1}^{\pi} \vdash_L (\gamma',\pi',n')\right) \\ \rightarrow \left(\mathcal{F}_{SL}(\boldsymbol{\mathcal{S}_{L1}},(o)\cdot\pi) = \mathcal{F}_{SL}(\boldsymbol{\mathcal{S}_{L2}},\pi')\right) \end{pmatrix}$$

Důkaz: Pro oba systémy a aplikované plány existují běhy $\mathrm{A}(\boldsymbol{\mathcal{S}_{L1}}, (o)\cdot\pi) = (\mathrm{C}_{0L1}^{\pi}, \mathrm{C}_{12} \dots \mathrm{C}_{F1})$ a $\mathrm{A}(\boldsymbol{\mathcal{S}_{L2}}, \pi') = (\mathrm{C}_{0L2}^{\pi}, \mathrm{C}_{22} \dots \mathrm{C}_{F2})$. Jelikož $\boldsymbol{\mathcal{S}_{L1}} \rhd_L \boldsymbol{\mathcal{S}_{L2}}$, pak podle definice 56 a Lemmy 20 platí $\vdash_{L1} = \vdash_{L2}$. Jelikož počáteční konfigurace systému $\boldsymbol{\mathcal{S}_{L2}}$ je konfigurací, která je podle definice 56 získána z přechodové funkce $\vdash_{L1}$ a platí $\mathrm{C}_{0L1}^{\pi} \vdash_{L1} \mathrm{C}_{0L2}^{\pi}$, pak musí platit $\mathrm{C}_{12} = \mathrm{C}_{0L2}^{\pi}$. Běhy systému z obou konfigurací realizované přechodovými funkcemi, které jsou stejné, musí být také stejné, proto i $(\mathrm{C}_{12} \dots \mathrm{C}_{F1}) = (\mathrm{C}_{0L2}^{\pi}, \mathrm{C}_{22} \dots \mathrm{C}_{F2})$ a také $\mathrm{C}_{F2} = \mathrm{C}_{F2}$. Proto i $\mathrm{A}(\boldsymbol{\mathcal{S}_{L1}}, (o)\cdot\pi) = \mathrm{A}(\boldsymbol{\mathcal{S}_{L2}}, \pi')$, jelikož vztah mezi $(o)\cdot\pi$ a π' je dán přechodovou funkcí $\vdash_{L1}$.

Na závěr si povšimneme toho, že výsledek aplikace plánu v případě neúspěšného vykovávání závisí pouze na té části plánu, která předchází neúspěšné akci, a této neúspěšné akci. Formulujeme proto ještě jednu Lemmu, která hovoří o tom, že pro uvažovaný systém všechny plány, které mají stejnou předponu obsahující neúspěšnou akci, mají stejný výsledek své aplikace.

Lemma 25: Pokud aplikace plánu na nějaký $\boldsymbol{\mathcal{S}_L}$ systém vede k tomu, že plán není přijat, tak není tímto systémem přijat jakýkoli plán, který vznikne spojením tohoto plánu a nějakého dalšího plánu pro daný systém. Tedy platí formule $\forall\pi, \pi' \in L\left(O_L\ (\{\tau\}, \Sigma)\right), (\gamma, (\Omega_F), n) \in Dom_{\mathrm{C}}(\mathcal{S}_L)\colon \left(A(\mathcal{S}_{\mathrm{L}}, \pi) = (\gamma, "(\Omega_F)", n) \rightarrow A(\mathcal{S}_{\mathrm{L}}, \pi\cdot\pi') = (\gamma, "(\Omega_F)", n)\right)$

Důkaz: Aplikace plánu $\pi \in O_L\ (\{\tau\}, \Sigma)$ na nějaký $\boldsymbol{\mathcal{S}_L}$ systém vede k běhu $(\mathrm{C}_1, \mathrm{C}_2 \dots, \mathrm{C}_i, \mathrm{C}_F)$. Jelikož C_F je ve tvaru $(\gamma, "(\Omega_F)", \mathrm{n})$ a v definici 51 není pravidlo, které by tuto konfiguraci dále transformovalo. Jelikož původní plán je umístěn na začátku nového složeného plánu a běh systému je dán aplikací nejprve akcí z původního plánu, tak i v běhu systému po aplikování nového plánu bude pouze posloupnost konfigurací $(\mathrm{C}_1, \mathrm{C}_2 \dots, \mathrm{C}_i, \mathrm{C}_F)$ a jako poslední opět dojde k provedení přechodové relace pravidlem FAIL a další rozšíření běhu systému aplikací přechodové relace nejsou možná.

Výše uvedená Lemma také ukazuje to, že pokud se systém dostane do stavu, který chápeme jako chybový, potom už v tomto stavu zůstane. To nás zavádí i k stručnému pojednání o výpočetní síle těchto systémů, které nyní na závěr kapitoly o $\boldsymbol{\mathcal{S}_L}$ systémech provedeme.

3.6.6 Stručná diskuse o výpočetních vlastnostech $\mathcal{S}_L$ systému

Systém $\mathcal{S}_L$ chápeme jako systém provádějící výpočet podle plánu, který je na tento systém aplikován. Plán, jak bylo uvedeno v předchozí podkapitole, je vykonán buďto správně, nebo nesprávně. Výpočetní schopnosti tohoto systému bychom mohli analyzovat porovnáním s jinými systémy, například automaty, které se chovají podle přijímaného řetězce, a analyzovat jimi přijímaný jazyk. Nebo k nim můžeme přistupovat jako k systémům, které vykonávají algoritmus zapsaný ve formě plánu a pak studovat, jaké algoritmy pro jaké úlohy lze tímto systémem vykonat. Podstatné pro takovou analýzu je Lemma 23, která říká, že počet kroků systému je omezen délkou vykonávaného plánu. Za správně vykonanou úlohu bychom považovali takovou, pro kterou by plán v počáteční konfiguraci systému byl přijat a počátečním vnitřním stavům jako vstupním datům by odpovídaly stavy báze a výpočtu v koncové konfiguraci jako výsledná data. Analýzu převody na jiné systémy by bylo možné provádět s konečnými automaty a ukázat, že nelze provádět výpočet pro některé z nich, například pro ty, které obsahují cykly a naopak ukázat, že pro jakýkoli $\boldsymbol{\mathcal{S}_L}$ systém lze sestavit konečný automat, jelikož množina možných akcí je konečná a jejich počet v plánu také. Jeden nekoncový stav by sloužil pro případ neúspěšně provedené operace a ostatní stavy by byly koncové.

Jiným přístupem k studii síly těchto systémů je uvažovat úlohy řešitelné algoritmy s nekonstantní časovou složitostí. Počet kroků u těchto algoritmů je funkcí délky vstupu. Pro obvyklé třídy složitosti, logaritmické, lineární, polynomiální a vyšší, je nemožné realizovat takové algoritmy v systémech $\boldsymbol{S_L}$, protože pro jakýkoli plán tohoto systému, který by měl řešit úlohu, lze nalézt vstup nějaké délky, který je neřešitelný tímto plánem, jelikož počet kroků k jejímu vyřešení je vyšší, než je délka plánu.

Nyní se posuneme k systémům, které vychází ze systémů $\boldsymbol{S_L}$, ale jsou schopny vykonávat hierarchické plány, které ALLL systémům přinesou výpočetní možnosti univerzálního výpočetního stroje (v teoretickém případě s bázemi neomezených velikostí) a tím zařadí tyto systémy mezi reálně použitelné výpočetní systémy.

3.7 Hierarchické plány v ALLL systémech, operace substituce a maturace nad hierarchickými plány

Doposud bylo chování systému dáno zápisem plánu jako lineární posloupnosti akcí, který se vykoná buďto celý, nebo po první neúspěšnou akci. Plány, které budou řídit chování systému v dalších kapitolách, budou sloužit k výpočtům stejně tak, jako sloužily lineární plány z předchozích kapitol, avšak budeme u nich předpokládat možné hierarchické uspořádání jednotlivých jejich částí. Později uvidíme, že stejně tak jako tomu bylo u lineárních plánů, může vykonávání hierarchického plánu v nějaké jeho částí neuspět. Plán pak nemusí být vykonán celý. Zatímco u lineárních plánů zůstal nevykonán vždy celý jejich zbytek, který následoval po neúspěšné akci, u hierarchických plánů se bude díky jejich strukturování jednat jen o jistou část. Tento princip umožní zvýšit výpočetní sílu systému a výsledný systém, jak na závěr této kapitoly ukážeme, bude mít výpočetní sílu univerzálního výpočetního stroje. Nejprve ale začneme uvedením motivací pro zavedení hierarchických plánů, provedeme formální definici hierarchických plánů a pak ukážeme systém, který pracuje s těmito plány.

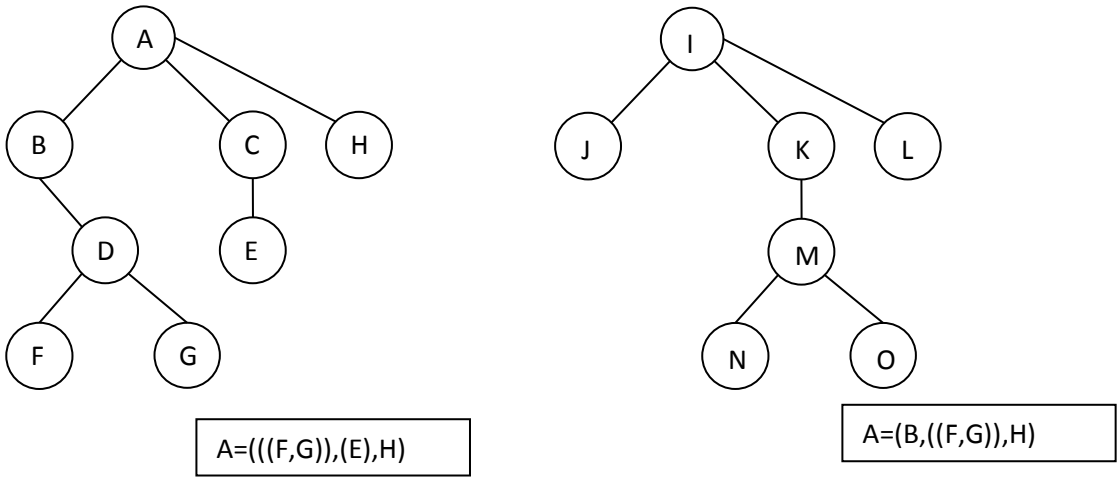
3.7.1 Motivace k zavedení hierarchických plánů do ALLL systémů

Hierarchické uspořádání v plánu, který řídí činnost prvku systému, umožní programovat komplexnější konstrukce a definovat části, které se agent pokusí vykonat, a v případě neúspěchu je tato snaha ukončena a plán může pokračovat na vyšší úrovni dále. Rozšíření ALLL výpočetního systému směrem k takovým strukturám plánů vykazuje některé rysy podobné BDI / AgentSpeak(L) systémům, kde části struktury záměrů, kterým by odpovídaly plány v ALLL systémech, nemusí být vykonány celé, pokud vykonání některé ze zamýšlených akcí v této struktuře selže. Řešení problémů tím, že je problém rozdělen na podproblémy a ty jsou řešeny postupně, nebo i paralelně, a výsledek řešení problému závisí na úspěchu či neúspěchu řešení podproblémů, je běžné v umělé inteligenci a výpočetních systémech už od padesátých let. V následujících odstavcích si představíme některé metody, ve kterých se rozklad úloh na podúlohy objevuje.

3.7.1.1 Řešení úloh rozkladem na podúlohy

Myšlenka hierarchického uspořádání procesů jako plánu také směřuje k možnosti programování agentů tím způsobem, že problém je řešen rozkladem původní úlohy na podproblémy. Řešení původní úlohy pak závisí na tom, jestli některé z podproblémů byly či nebyly úspěšně vyřešeny. V klasické umělé inteligence je tento způsob reprezentován pomocí AND/OR grafů. Mezi obvykle zmiňované příklady použití metod rozkladu na podproblémy patří úloha Hanojské věže, hraní jednoduchých her, nebo hraní složitých her metodami MiniMax / AlfaBeta.

Pro demonstraci ukážeme dva hypotetické příklady rozkladu úloh na podúlohy.



Obrázek 1: Příklady řešení problému rozkladem na podproblémy

První příklad je ukázán na obrázku 1. Úloha A se rozkládá na tří částí reprezentující podúlohy B, C a H. Podúloha B je řešena podplánem na druhé úrovni s jednou podúlohou řešenou podplánem D na další nižší úrovni. Tento podplán dále vykoná úlohy F a G bez dalšího rozkládání na podúlohy. Plán pro úlohu C obsahuje pouze jednu podúlohu E. Listy F, G, E a H stromu, který reprezentuje řešení úlohy A rozkladem na podúlohy, mohou být nějaké akce, nebo nerozložitelné úlohy na dané úrovni plánu.

Úloha I je reprezentována stromem se stejnou hloubkou, jakou měl strom reprezentující úlohu A. Pokud bychom ale prováděli výpočet nad tímto stromem a postupovali zleva, tak bychom se mohli dostat při řešení prvního podproblému do hloubky jedna. V případě stromu pro úlohu A by to byla hloubka tři. Druhý příklad demonstruje to, že úroveň zanoření plánu a hloubka stromu, která je v obou případech 3, nemusí být stejné, ale úroveň zanoření plánu je stejná nebo nižší, než je hloubka stromu. Hloubka stromu ale odpovídá úrovni plánu jako celku, to znamená, do jaké úrovně zanoření bychom za určitých okolností mohli dospět.

3.7.1.2 Implementace řešení rozkladem problémů na podproblémy

Jazyk LISP (List Processing) [12] byl vytvořen John McCarthym a jeho týmem na MIT v 50. letech 20. století. Jeho základ tkvěl v rekurzivním vyčíslování funkcí. Činnost LISPu spočívá ve vyhodnocování S-výrazů. S-výrazy jsou posloupnosti zpracovávané jako funkce, kde první prvek je funkční symbol, a ostatní prvky jsou argumenty funkce. Prvky S-výrazu mohou být konstanty nebo další S-výrazy. Ty jsou zpracovávány v době volání funkce, pokud se jedná o obvyklou implementaci, která využívá volání hodnotou.

Typické pro zpracování funkcí je rekurzivní přístup a zpracování seznamu, což koresponduje i s názvem tohoto jazyka. Běžně jsou funkce implementovány tak, že dochází ke zpracování prvního prvku, který pro seznam zpřístupní funkce car (content of adress) a zbývající část seznamu je zpracovávána funkcí, pro kterou je argumentem tento zbytek seznamu, zpřístupněný funkcí cdr (content of decrement part).

Rozklad na podproblémy je v jazyce LISP snadno implementovatelný. Jednotlivé podúlohy mohou být vyhodnocovány jako samostatné funkce a výsledek úlohy pak lze určit pomocí funkcí realizujících logické operace nad výsledky vyhodnocení jednotlivých funkcí.

3.7.1.3 Modely struktur záměrů jako hierarchie procesů

Jiný pohled na možnost použití hierarchických struktur plánů nabízí oblast výpočtů v dynamickém a nepřístupném prostředí, kde agent nemá svými senzory přístup k celkovému stavu prostředí. Výpočet může být odložen, pokud během jeho provádění dojde ke zjištění, že neodpovídá nově zjištěnému stavu prostředí. Několikrát zmíněné BDI struktury záměrů v obou uváděných realizacích, tedy v systémech AgentSpeak(L) a dMars, modelují způsob možného řešení úlohy grafovými strukturami, konkrétně lineární hierarchií plánů pro jednotlivé záměry u systému AgentSpeak(L) a stromovou strukturou plánů u systému dMars. V druhém případě opět není na jednotlivých úrovních stromu dáno pořadí provádění podcílů a tak může systém nedeterministicky volit větev ve struktuře záměrů, kterou bude provádět.

V realizaci systému AgentSpeak(L), která je nazvaná JASON, lze provádět výměnu zápisů plánů komunikací mezi agenty. Agent pro jistý typ řečového aktu předává svoji znalost postupu k naplnění nějakého cíle ve formě relevantního plánu. Relevantní plán je takový, který obsahuje jako spouštěcí událost predikát unifikovatelný pro dané cíle, které jsou v tomto systému zapsány také ve formě predikátů. V případě, že nastane situace, že je hledán plán k dosažení takového cíle, může být vybrán právě tento obdržený plán a agentovo chování je pak řízeno tímto plánem.

ALLL systémy se budou lišit od BDI agentních systémů mimo jiné v tom, že pořadí provádění jednotlivých podcílů bude určeno již v době napsání programu a nebudeme předpokládat nějaký rozhodovací mechanismus, který by z plánů vybíral v době běhu systému. Tedy nebude zde docházet k procesům výběru relevantního a aplikovatelného plánu, ale plány budou volány svým jménem. Programátor bude mít kontrolu nad pořadím vykonávání plánů, což může hrát roli a to z následujících důvodů:

- Agent během své činnosti proaktivně ovlivňuje a mění prostředí. Posloupnost změn prostředí může být důležitá pro dosažení cíle, a tak záměnou pořadí provádění plánů dojde k různým posloupnostem prováděných akcí systémem a to může vést k rozdílnému výslednému stavu prostředí.
- Pořadí plánů může znamenat ustavení preferencí možností, jak postupovat v předpokládaném stavu prostředí. Pokud nějaký plán neuspěje, je možné vyvolat plán jiný s jiným postupem, nebo plán pro řešení situací, ve kterých nešlo dosáhnout cíle původním plánem.
- Rovněž jde i o nutnost při snaze vybudovat agentní systém na nejnižší úrovni abstrakce a procesy unifikace, které jsou nezbytné pro nalezení relevantních a aplikovatelných plánů by do tohoto systému zanášely nežádoucí výpočty, které by kromě časové náročnosti vyžadovaly i paměťové nároky na uchovávání prostředí, substituovaných proměnných, aktuálních struktur záměrů apod.

Je poctivé zde zmínit i to, že vynechání možnosti agenta rozhodovat o postupech na základě vlastního výběru plánů za běhu systému posouvá ALLL systémy do oblasti mimo současné agentní systémy.

Co ale bude v nadcházejících ALLL systémech společné s BDI systémy bude právě hierarchické uspořádání plánu. Proč ale používat ve výpočtech hierarchické plány? Uveďme si i v tomto případě několik důvodů.

3.7.1.4 Shrnutí motivace k zavedení hierarchických plánů v ALLL systémech

Jako důvody proč používat plány v hierarchické podobě uvedeme následujících pět bodů.

- Z pohledu agentních technik lze podplány chápat jako možnosti, které agent může v daném stav výpočtu zvolit k dosažení svých cílů. V plánu na vyšší úrovni by mělo být možné vyhodnotit úspěšné či neúspěšné vykonání podplánu.
- Metody řešení úloh rozkladem na podúlohy spadají do oblasti klasické umělé inteligence. Programovací jazyky jako LISP jsou vhodné právě pro zápisy programů, ve kterých takové rozklady vedou k jednoduchému a elegantnímu strojovému řešení úloh.
- Předávání mezivýsledků po provedení plánu na dané úrovni na vyšší úroveň plánu má rysy funkcionálního programování a lze tak využít programátorských technik známých z funkcionálního přístupu k programování.
- Důvodem je i modularita a škálovatelnost agentního kódu. Agenti si mohou předávat schopnosti dosažení nějakého cíle, nebo řešení nějakého problému ve formě plánů. Záměnou jednoho plánu za jiný stejného jména lze změnit chování agenta v určitých situacích bez nutnosti jeho kompletního přepisování.
- Zvýšení abstraktní úrovně programování umožní lépe specifikovat činnost agenta. ALLL agentní jazyk se nachází na nízké úrovni abstrakce a programování systémů v tomto jazyce je málo pohodlné. Hierarchizace plánů toto lehce usnadňuje a umožňuje kvalitnější návrh a programování ALLL systémů.

V následujících podkapitolách nejprve představíme principy realizace hierarchických plánů a operací nad nimi a dále provedeme formální definice těchto struktur a operací, abychom v hlavní části kapitoly vytvořili ALLL systém, jehož chování bude řízeno hierarchickými plány.

3.7.2 Výpočty s hierarchickými plány

Realizace hierarchického uspořádání plánu v ALLL systému je prováděna zápisem plánu s hierarchickou strukturou. Z definic obecných prvků je patrné, že tyto prvky mají v sobě hierarchické uspořádání a stejně tak i plány jsou z definice také obecnými prvky, protože se skládají z akcí, které jsou obecnými prvky. Interpretace plánu je prováděna podle přechodové funkce systému s těmito plány, která je realizována s respektem k jejich hierarchickému uspořádání.

3.7.2.1 Realizace hierarchického plánu v ALLL systémech

Plán tedy bude opět posloupnost akcí, přičemž některé z akcí mohou jako argument mít informaci o plánu, který se má v rámci akce vykonat. Akci vykonání podplánu v daném místě podplánu budeme zapisovat jako "@ argument" a argumentem bude plán k vykonání, nebo jeho jméno.

Příklad 16: Struktura plánu s jedním hlavním plánem a třemi podplány může být následující:

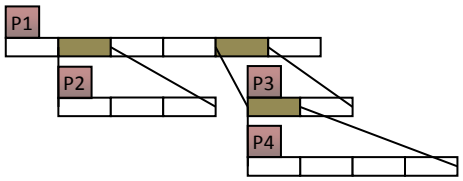
$$"\left((o_1,p_1),\left(@,\left((o_2,p_2),(o_3,p_3),(o_4,p_4)\right)\right),(o_5,p_5),(o_6,p_6),\left(@,\left(\left(@,\left((o_7,p_7),(o_8,p_8),(o_9,p_9),(o_{10},p_{10})\right)\right),(o_{11},p_{11})\right)\right),(o_{12},p_{12})\right)"$$

Při představování plánu z tohoto příkladu budeme používat pojmy atomická a neatomická akce. Atomická akce je interpretovatelná bez toho, aby bylo nutno vykonat nějaký podplán. Například všechny akce v plánech systému S_L byly atomické. Neatomickou akcí je myšlena akce, která spouští podplán. Tyto akce mohou mít daný podplán jako argument, nebo mohou podplán vyhledávat v bázi

plánů. Zápis plánu na nejvyšší úrovni zde obsahuje celkem šest akcí. První akce (s indexy 1) je akcí atomickou. Neatomickou akcí je následující akce, která spouští podplán se třemi atomickými akcemi 2, 3 a 4. a tak dále.

3.7.2.2 Interpretace hierarchického plánu

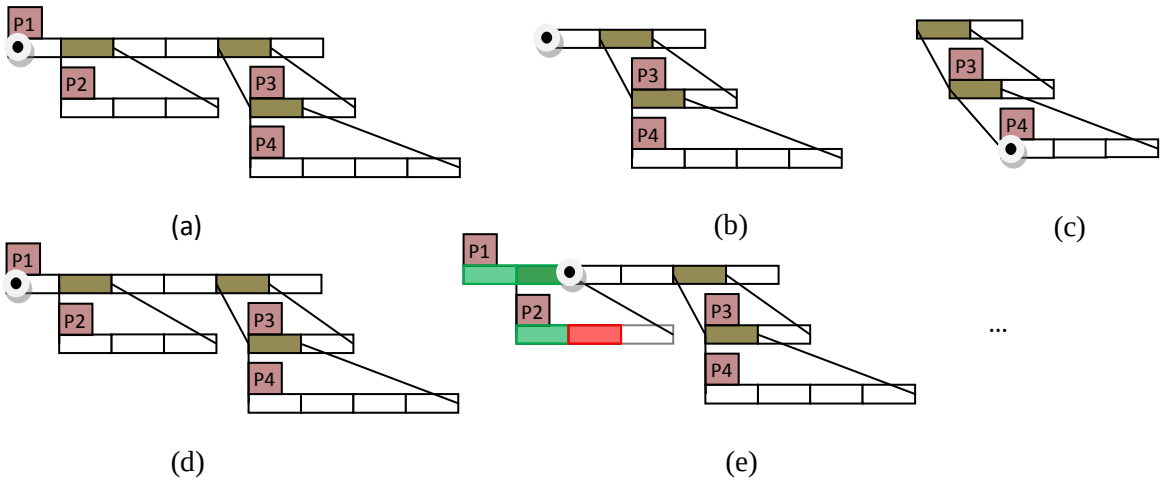
Neformálně by se dal výpočet v ALLL systémech s hierarchickým plánem představit tak, že bude probíhat pro nějaký plán, což bude struktura, kde jednotlivé plány mohou vedle akcí obsahovat i podplány. Příklad grafického znázornění hierarchického plánu je uveden na obrázku 2.



Obrázek 2: Hierarchická reprezentace plánu a odpovídající grafové znázornění

Pořadí vykonávání plánu je dáno pořadím průchodu touto hierarchií jako stromem. To ovšem platí, jen pokud nedojde k provedení akce, která skončí neúspěchem. V případě neúspěchu je na dané úrovni výpočet ukončen a pokračuje se na předchozí, nadřazené úrovni, vykonáním akce, která následuje za akcí, která vyvolala neúspěšný podplán. V případě uvedeném na obrázku bude vykonávání podplánů, pokud nedojde k provedení neúspěšné akce, v pořadí P2, P4, P3 a P1. Pokud by například neuspěla třetí akce plánu P1, vykonaly by se pouze plány P2 a P1.

Způsob vykonávání hierarchického plánu v případě, že některé atomické akce skončí neúspěchem, je ukázán na obrázku 3.



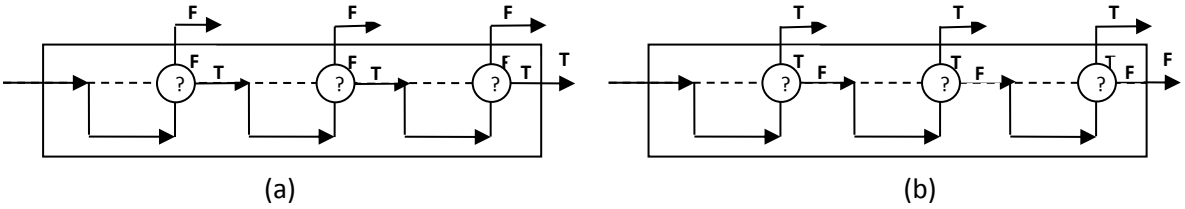
Obrázek 3: Ukázka průběhu transformace hierarchického plánu během výpočtu

Plán je transformován prováděním jednotlivých akcí. Na obrázcích (a) – (c) jsou ukázány plány pro tři fáze výpočtu. Obrázky (d) a (e) ukazují možný způsob transformace mezi prvními dvěma fázemi, kdyby spuštěný podplán P2 neuspěl svoji druhou akcí. Tento plán by byl následně ukončen jako neúspěšný. Akce, která tento plán spustila v plánu P1, by však neskončila jako neúspěšná a plán P1 by

pokračoval dále. To mác demonstrovat tu skutečnost, že neúspěch podplánu v hierarchii plánu ukončí vykonávání pouze tohoto podplánu a plány na vyšší úrovni mohou pokračovat dále.

3.7.2.3 Realizace AND/OR grafů ALLL systémem s hierarchickými plány

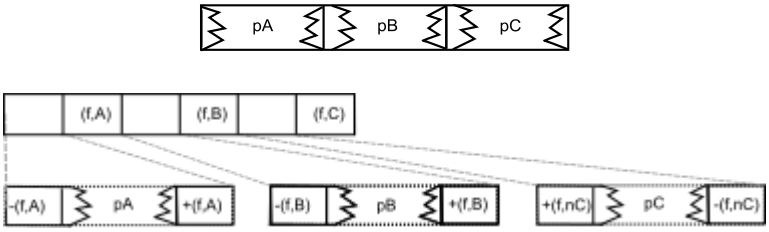
Jak by vypadala realizace řešení úloh rozkladem na podúlohy metodou AND/OR grafů ALLL systémem s hierarchickými plány je uvedeno na obrázku 4.



Obrázek 4: Princip řešení (a) AND a (b) OR úloh

Realizace uvedeného přístupu ALLL systémem je postavena na tom, že plán o svém úspěchu či neúspěchu rozhoduje pouze akcí testování báze. V případě, kdy výsledkem je neúspěch, musí neúspěchem skončit i operace testování. Vyhodnocováním podmínek se budeme zabývat později v části věnované programátorským technikám u ALLL agenta. Nyní naznačíme základní princip realizace AND/OR grafů ALLL systémy.

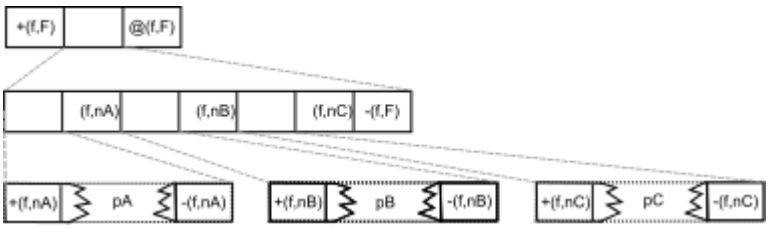
Pokud máme testovat úspěch či neúspěch provádění části plánu, pak před jeho provedením vložíme do báze vyhrazený prvek, například příznak se jménem plánu, a tento příznak v případě úspěšného provedení podplánu odstraníme jeho poslední akci. Nebo naopak lze tento příznak z báze na začátku provádění podplánu odstranit a po úspěšném provedení opět přidat. Potom lze použitím akce testování přítomnosti příznaku v bázi vyhodnotit úspěšnost nebo neúspěšnost podplánu. Obrázek 5 demonstruje tento princip v případě implementace AND uzlu.



Obrázek 5: Řešení rozkladem na podproblémy, AND uzel

Ve vrchní části obrázku je vedle jednodušší verze, kdy jednotlivé úlohy jsou umístěny za sebou, a první neúspěch u kterékoli z úloh znamená neúspěch celé úlohy. Pod ní je uveden i složitější případ, kde jednotlivé úlohy jsou zapsané samostatně a jsou spouštěné jako podúlohy v hierarchii plánu. Každý z plánů pro podúlohy, které mají být úspěšně splněny, nejprve odstraní z báze příznak a ten navrátí zpět až po úspěšném ukončení úlohy. Tento příznak je testován po každém navrácení do plánu vyšší úrovně, a pokud příznak v bázi není, celý výpočet končí neúspěchem. Pokud ovšem každá z podúloh uspěje, na konci skončí úspěšně i úloha jako celek. Akce jsou zapsány trochu odlišně od toho, jak byly definovány v kapitole 3.3.2. Operátory pro přidání a odstranění prvku jsou uvedeny před prvkem, pro akci testování operátor uveden není. Toto odpovídá tomu, jak budou akce zapisovány v jazyce ALLL, který bude tématem kapitoly 4.2.

Implementace OR uzlu je ukázána na obrázku 6. Využijeme zde vztahu mezi logickými formulemi daného de Morganovým zákonem, tedy že platí $A \vee B \vee C \equiv \overline{(\bar{A} \wedge \bar{B} \wedge \bar{C})}$. K implementacií takového rozkladu na podproblémy bude nyní třeba tří úrovní plánů.

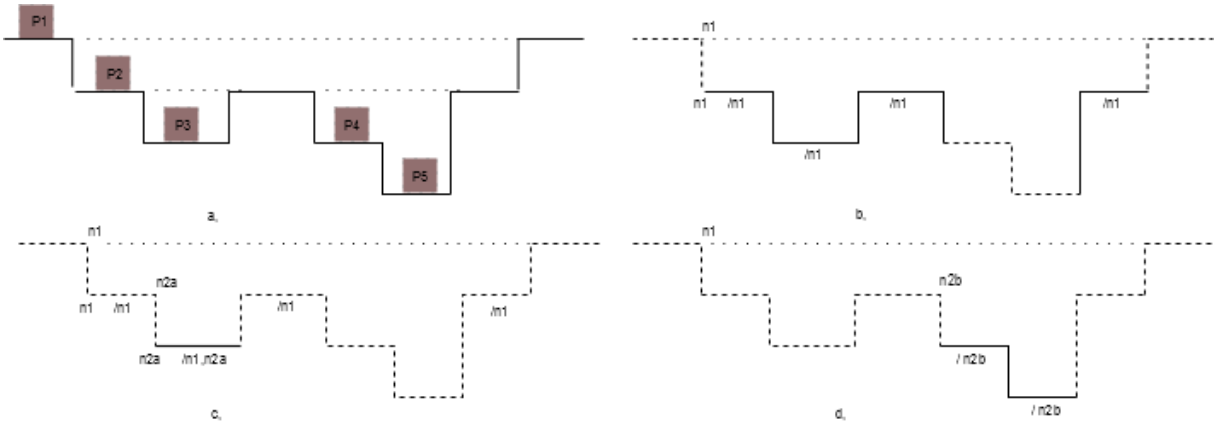


Obrázek 6: Řešení rozkladem na podproblémy, OR uzel

Nejvyšší úroveň plánu zavede do báze příznak (f,F) značící počáteční předpoklad pravdivosti vyhodnocované formule. Plán na druhé úrovni postupně volá jednotlivé podúlohy a testuje jejich splnění. V případě splnění kterékoli z úloh plánem na třetí úrovni dojde k neúspěšnému testování dvojice značící neúspěch úlohy (f,nA), (f,nB) atd. na druhé úrovni a výpočet končí na první úrovni úspěchem, protože dvojice (f,F) je stále přítomna v bázi. Tato dvojice je z báze odstraněna pouze, pokud ani jedna z podúloh neuspěje a plán druhé úrovně dospěje až do svého konce.

3.7.2.4 *Předávání výsledku výpočtu v hierarchických plánech na různých úrovních*

Jednou z významných nových vlastností systémů s hierarchickými plány je i to, že výsledky části výpočtu na různých úrovních hierarchického plánu je možné předávat dále do kontinua pomocí systému registrů, který nahrazuje jeden substituovatelný registr v systému. Použití více registrů na různých úrovních poskytne bohatší možnosti jak a kde substituovat aktuální hodnotu výpočtu dále do plánu. K pochopení těchto principů uvedeme následující příklad.



Obrázek 7: Příklad předávání mezivýsledků mezi jednotlivými úrovněmi plánu

Na obrázku 7 (a) je ukázán hierarchický plán se třemi podplány. Plán P2 je vykonáván po částech, protože během jeho vykonávání jsou vyvolány plány P3 a P4. Plán P5 je vyvolán na konci plánu P4. Podle úrovně zanoření můžeme považovat plán P1 za hlavní plán na nejvyšší úrovni, plán P2 za plán na první úrovni zanoření, plány P3 a P4 za plány na druhé úrovni zanoření a plán P5 za plán na třetí úrovni zanoření.

Podstatná místa výpočtu z hlediska hierarchického uspořádání plánu jsou ty, kde dochází k přechodu mezi jednotlivými úrovněmi. Po skončení výpočtu části plánu P1 je aktuálním stavem výpočtu prvek, který na obrázku reprezentuje symbol n1. (obrázek 7 (b)). Prvek n1 je v okamžiku spuštění plánu P2

substituován na všech úrovních kromě nejvyšší za symbol aktivního registru. Symboly registrů jsou změněny tak, že aktivním registrem se stává nějaký jiný symbol registru. Jedná se o takzvanou operaci maturace registrů, kterou si představíme za chvíli. Po skončení části plánu P2 a spuštění plánu P3 (obrázek 7 (c)) je aktuální hodnota výpočtu nahrazena za právě aktuální symboly registru v plánu P3. Poslední z obrázků (obrázek 7 (d)) demonstruje ukončení další části výpočtu plánu P2 a spuštění plánu P4, ve kterém mohly být také substituovány symboly aktivních registrů za aktuální hodnotu. V případě plánů P4 a P5 ale na obrázcích nedochází k nahrazování registrů před jejich spuštěním. Toto má demonstrovat spuštění externího plánu, který nebyl součástí struktury do ukončení druhé části plánu P2, nebo lépe řečeno plán P2 spustil externí plán P4 a až v tomto okamžiku je plán P4 součástí struktury a mohou v něm být substituovány aktivní registry za aktuální hodnoty výpočtů.

Následující kapitola zavede formálně akce hierarchických plánů, které vytváří podplány, a pak budou diskutovat principy předávání mezivýsledků výpočtů v ALLL systémech s hierarchickými plány podrobněji.

3.7.3 Hierarchické plány a akce nad hierarchickými plány

Hierarchické plány jsou opět strukturami podle definice 32. Tato definice však neurčuje, z jakých operací se plán skládá, a tak pro systém $\mathcal{S}_{\mathrm{H}}$, obdobně jako jsme to učinili v kapitolách 3.6.1 pro systém s bází bez uspořádání, a tím i pro systém $\mathcal{S}_{\mathrm{L}}$, uvedeme, z jakých akcí je hierarchický plán tvořen a jakou mají tyto akce strukturu.

Množinu akcí pro tento systém tvoříme na základě množiny atomů báze Σ a množiny symbolů registrů *T*. Množinu akcí můžeme také představit jako množinu zahrnující množinu akcí nad bází, a dále akce dosud neuvedené, které transformují strukturu plánu přechodem mezi úrovněmi plánů. To jsou akce přímého vyvolání podplánů a akce vkládající jednu struktury plánu do druhé, které chápeme jako akce nepřímého vyvolání podplánů. V praktických realizacích bude možné definovat i další specifické akce, které uvedeme až později v kapitole o reálných ALLL systémech.

Začneme definicí syntaxe akce nepřímého volání podplánů.

Definice 57: Akce nepřímého vyvolání podplánu pro ALLL systémy s množinou atomických prvků Σ a množinou symbol registrů T tvoří množinu

$$O_{IC}(\mathrm{T},\Sigma) = \{'('\,'@'\,'('jmenoplanu\;')'\,')', jmeno\_planu \in \Sigma \cup \mathrm{T}\}.$$

U akcí přímého volání plánů bude situace složitější. Tato akce má jako argument prvek z množiny hierarchických plánů a hierarchický plán je zase prvek nad množinou akcí včetně akcí přímého volání plánů. Budeme proto definovat tyto akce i plány postupně po jednotlivých úrovních.

Definice 58: Akce přímého vyvolání podplánu nulté úrovně pro ALLL systémy s množinou atomických prvků Σ, množinou symbol registrů T a množinu systémových akcí O tvoří množinu

$$O^0_{DC}(\mathrm{T},\Sigma,O) = \left\{'('\,'@'plan')': plan \in L\left(O \cup O_{IC}(\mathrm{T},\Sigma)\right)\right\}$$

Hierarchický plán první úrovně je pak plán, který kromě nějaké množiny akcí obsahuje i akce nepřímého volání plánu a akce přímého volání nulté úrovně. Jeho definice je následující.

Definice 59: Hierarchické plány první úrovně pro ALLL systémy s množinou atomických prvků Σ, množinou symbolů registrů T a množinu systémových akcí O tvoří množinu

$$H^1(\mathrm{T},\Sigma,O) = L\left(O \cup O_{IC}(\mathrm{T},\Sigma) \cup O_{DC}^0(\mathrm{T},\Sigma,O)\right)$$

Nyní můžeme dále definovat akce přímého volání plánů vyšších úrovní a také hierarchické plány vyšších úrovní.

Definice 60: Akce přímého vyvolání podplánu i-té úrovně pro ALLL systémy s množinou atomických prvků Σ, množinou symbol registrů T a množinu systémových akcí O tvoří množinu

$$O_{DC}^i(\mathrm{T},\Sigma,O) = \{'(\ '@'plan')', plan \in H^i(\mathrm{T},\Sigma,O)\}$$

Argumentem akce přímého volání podplánu i-té úrovně je hierarchický plán stejné úrovně. Akce přímého vyvolání plánu první úrovně spouští v systému hierarchický plán první úrovně jako podplán. Hierarchické plány vyšších úrovní jsou pak tyto.

Definice 61: Hierarchický plány i-té úrovně pro ALLL systémy s množinou atomických prvků Σ, množinou symbolů registrů T a množinu systémových akcí O tvoří množinu

$$H^i(\mathrm{T},\Sigma,O) = L\left(O \cup O_{IC}(\mathrm{T},\Sigma) \cup O_{DC}^{i-1}(\mathrm{T},\Sigma,O)\right)$$

Hierarchický plán druhé úrovně může obsahovat akce s přímým voláním plánu první úrovně. Stejně tak i vyšší úrovně plánů mohou vyvolávat podplány až po úroveň, do které patří.

Pokud budeme mluvit o hierarchických plánech, budeme mluvit o hierarchických plánech obecné, neomezené úrovně. Počet zanoření přes přímé volání takovýchto plánů nebude formálně omezen.

Definice 62: Obecné hierarchické plány pro ALLL systémy s množinou atomických prvků Σ, množinou symbolů registrů T a množinu systémových akcí O tvoří množinu

$$H(\mathrm{T},\Sigma,O) = \bigcup_{n>0} H^n(\mathrm{T},\Sigma,O)$$

Stejně tak budeme mít i akce přímého vyvolání obecného plánu.

Definice 63: Akce přímého vyvolání obecného podplánu pro ALLL systémy s množinou atomických prvků Σ, množinu symbolů registrů T a množinu systémových akcí O tvoří množinu

$$O_{DC}(\mathrm{T},\Sigma,O) = \bigcup_{n\geq 0} O_{DC}^n(\mathrm{T},\Sigma,O)$$

Význam nových akcí pro fungování celého systému bude objasněn v následujících podkapitolách, ve kterých rozšíříme definici přechodové relace systému $\mathcal{S}_{\mathrm{L}}$ z kapitoly 3.6.3 o pravidla, která se budou vztahovat k těmto dvěma novým typům akcí.

3.7.4 Registry, principy substitucí a maturací registrů

Nyní si představíme systém registrů a operace související s výpočtem v systémech s hierarchickými plány obsahujícími symboly ze systému registrů. Zmíníme zde také principy substituce registrů, které umožní předávat mezivýsledky výpočtů, jak bylo demonstrováno výše. Také zmíníme operaci maturace registrů, která je používána během přechodů z plánů vyšší úrovně na plány na nižších úrovních.

Program zapsaný jako hierarchický plán s registry můžeme považovat za abstraktní kvůli přítomnosti symbolů registrů v něm, které jsou během výpočtu nahrazovány mezivýsledky. Symboly registrů se mohou vyskytovat v argumentech akcí plánů, tedy v obecných prvcích reprezentujících data nebo plány, se kterými agent pracuje. Při srovnání s například π-kalkuly bychom mohli symboly registrů považovat za 'prázdná místa' v zápisech procesů, která jsou nahrazována během výpočtu hodnotami z aktuálního kontextu výpočtu. Toto zaplňování prázdných míst má v ALLL systémech obdobu právě v nahrazování registrů za mezivýsledky. Proto lze ALLL hierarchický plán považovat za abstraktní právě v tom smyslu, že jeho konkrétní forma při jeho vykonávání je dána výsledkem substitucí symbolů registrů, které obsahuje, za hodnoty předávané mezi plány na různých úrovních.

Následující odstavce představí principy vytváření hierarchie plánů a práce s registry v těchto strukturách prováděním příslušných operací. Uvidíme, že to je právě ten princip, který umožní agentům výpočetní možnosti univerzálního výpočetního systému a zároveň zavede do systému některé vlastnosti agentních systémů používajících struktury záměrů.

Symboly registrů budou uspořádány a toto uspořádání bude tvořit jejich hierarchii. Pro lepší názornost budeme v této kapitole symboly registrů znázorňovat jako nějaký symbol registru, obvykle τ, a hodnotu, která určuje *věk* registru a kterou budeme uvádět jako index u symbolu registru. V dalším textu budeme symbol registru a jeho věk nazývat zkráceně registr. Věk registru je možné chápat jako stupeň zanoření registru v hierarchii plánů s tím, že nejvíce zanořený registr má být substituován nejpozději. Proto by se jako indexy slušelo uvádět záporné hodnoty. Nejstarším z registrů by byl registr s nulový věkem, který také jako jediný je substituován za aktuální hodnotu výpočtu, pokud je součástí akce, která má být vykonána, nebo plánu, který je spouštěn. Registry vyššího věku jsou namísto substituce za aktuální hodnotu výpočtu pouze 'omlazeny' a jejich věk bude o jedna menší. Přesto budeme věk uvádět kladnými hodnotami a symboly ${}^{0}\tau$, ${}^{1}\tau$ … ${}^{i}\tau$ budou reprezentovat registry s tím, že horní přední indexy registrů označují věk registru. Registr s nulovým věkem budeme zapisovat bez indexu věku pouze jako τ.

Okamžik, kdy jsou symboly registry nahrazovány, závisí na jejich umístění ve struktuře plánů. Čím jsou nahrazovány, zase závisí na jejich pozici v hierarchii sytému registrů. K substitucím dochází jednak před vykonáváním akce, kdy je ale substituován pouze symbol registr nultého věku obsažený v argumentu této akce, nebo při spouštění podplánu, kdy dojde k substitucím registrů každého věku.

3.7.4.1 Příklady operací substituce a strukturálních operací v hierarchické reprezentaci plánů

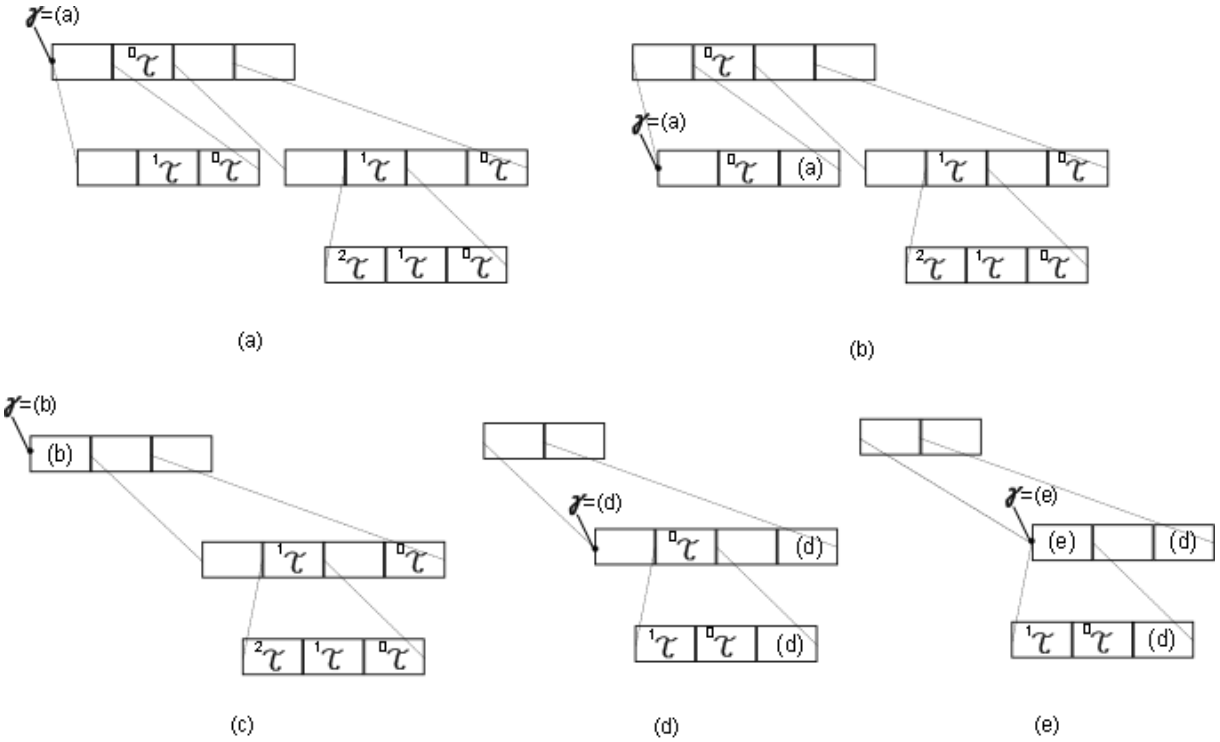
Operací maturace plánu je myšleno provedení takových substitucí, které všechny registry věku 1 a více nahradí registry s o jedna nižším věkem. Registr věku nula je pak nahrazen aktuální hodnotou výpočtu. Pokud označíme aktuální hodnotu výpočtu jako act, potom by se operace maturace dala chápat jako množina substitucí $\sigma = \{[act/\tau], [\tau/\ {}^{1}\tau], [\ {}^{1}\tau/\ {}^{2}\tau], \ldots [{}^{n-1}\tau/\ {}^{n}\tau]\}$, kde n je nejvyšší věk registru, který je obsažený v plánu. Operace maturace je pak aplikací této množiny substitucí na tento plán.

V následujících příkladech použijeme zápis ve tvaru $\langle\ {}^{a}\tau,\ {}^{b}\tau \ldots\rangle$, který bude reprezentovat plán, ve kterém se vyskytují symboly registru τ s věky a, b. Na nejvyšší úrovni plánu budou uvedena všechna stáří registrů, která jsou uvedena v akcích plánu na této úrovni. Pokud plán obsahuje podplán, bude jeho zápis obsahovat i zanořené plány s registry, které tyto podplány obsahují atd. Například zápis

$\langle {}^{a}\tau,\ {}^{b}\tau, \langle \tau,\ {}^{a}\tau,\ {}^{c}\tau \rangle \rangle$ reprezentuje plán, který na nejvyšší úrovni používá registry s věky a, b a potenciálně spouští podplán, který používá registry s věky 0, a, c. Nyní si uvedeme dva příklady, které na takových zápisech plánů ukáží princip operace maturace.

Příklad 17: Maturace plánu $\langle \tau, \langle \tau,\ {}^{1}\tau, \langle \tau,\ {}^{1}\tau,\ {}^{2}\tau \rangle \rangle \rangle$ za stavu výpočtu s hodnotou aktivního registru (a, b, c) vede k aplikaci substitucí $\langle \tau, \langle \tau,\ {}^{1}\tau, \langle \tau,\ {}^{1}\tau,\ {}^{2}\tau, \rangle \rangle \rangle \ \{[(a,b,c)/\tau][\tau/\ {}^{1}\tau][\ {}^{1}\tau/\ {}^{2}\tau]\}$ a výsledkem operace maturace je plán $\langle (a,b,c), \langle (a,b,c), \tau, \langle (a,b,c), \tau,\ {}^{1}\tau \rangle \rangle \rangle$. Po provedení první části plánu na nejvyšší úrovni se při spouštění podplánu s aktuálním stavem výpočtu, například (d, e) opět provede operace maturace, tentokrát ve tvaru $\langle (a,b,c), \tau, \langle (a,b,c), \tau,\ {}^{1}\tau \rangle \rangle \{[(d,e)/\tau][\tau/\ {}^{1}\tau][\ {}^{1}\tau/\ {}^{2}\tau]\}$ a výsledkem této operace je plán $\langle (a,b,c), (d,e), \langle (a,b,c), (d,e), \tau \rangle \rangle$

Příklad 18: Druhý příklad doprovodíme i obrázkem. Uvidíme na něm podrobněji, jak dochází k nahrazování symbolů registrů v jednotlivých částech výpočtu a jak jsou hodnot předávány ve struktuře plánu během jeho provádění. Na obrázku 8 (a) je uveden příklad plánu, ve kterém hlavní plán spouští dva podplány a druhý z těchto podplánů spouští další podplán. Struktura plánu a registrů je zde $\langle \langle \tau,\ {}^{1}\tau \rangle, \tau, \langle\ {}^{1}\tau, \langle \tau,\ {}^{1}\tau,\ {}^{2}\tau \rangle, \tau \rangle \rangle$. Aktuální stav výpočtu je "(a)" a akce, která má být vykonána, je akcí přímého volání podplánu. Právě v tomto okamžiku dojde k provedení operace maturace registrů a to před vlastním provedením spuštění podplánu. Prováděný plán po provedení spuštění podplánu neobsahuje další podplány a má strukturu registrů pouze $\langle \tau \rangle$. To je ukázána na obrázku 8 (b).



Obrázek 8: Příklad provádění operace maturace u hierarchických plánů

Registr, který byl ve spuštěném podplánu označen věkem nula, byl substituován, a registr původně s věkem jedna je nyní registr s věkem nula a bude substituován před provedením akce, ve které je použit. Na třetím obrázku (c) je ukázána situace po návratu z podplánu a v době, kdy bylo započato provádění druhé akce z plánu na nejvyšší úrovni. Odpovídající struktura plánů a registrů je nyní $\langle \langle\ {}^{1}\tau, \langle \tau,\ {}^{1}\tau,\ {}^{2}\tau \rangle, \tau \rangle \rangle$. Jelikož tento plán provádí akci, která obsahovala registr s věkem nula, a aktuální hodnota výpočtu je "(b)", registr v argumentu akce byl substituován za tuto hodnotu.

Provedení čtvrté akce plánu na nejvyšší úrovni vedlo ke spuštění podplánu, který je strukturou s jedním dalším podplánem, viz obrázek 8 (d). V struktuře spuštěného plánu se vyskytovaly dva registry s věkem nula, které byly nahrazeny aktuální hodnotou výpočtu v době, kdy byl plán vyvoláván. Zde je vidět, jak byla aktuální hodnota výpočtu předána nejen na úroveň spouštěného plánu, ale i dále ve struktuře na úroveň dalšího podplánu. Plán na této úrovni bude počítat s daty, které získal v době vyvolávání plánu na úrovni o jedna vyšší. Ve struktuře plánu byly také přítomny registry s věky jedna a dva a ty byly maturovány, což vedlo k situaci, že registry mají věky nula a jedna a jsou teď ve struktuře plánu $\langle \langle \tau, \langle \tau, {}^{1}\tau \rangle \rangle \rangle$. Poslední z obrázků, obrázek 8 (e), ukazuje situaci, kdy bylo zahájeno vyvolávání podplánu úrovně dva, ale nedošlo ještě k provedení operace maturace. Je zde zatím substituován pouze registr v rámci akce, ale na další substituce ve spouštěných podplánech teprve dojde.

3.8 ALLL systém s hierarchickými plány

Obdobně jako u systému s lineárními plány, i v případě systému s hierarchickými plány nejprve definujeme strukturu systému, pak přejdeme k diskusi ohledně formy aplikovatelných plánů na tento systém, tedy hierarchických plánů a akcí, které tyto plány mohou obsahovat. Dále uvedeme, jak se tyto plány vykonávají. Přechodová funkce bude používat systém registrů způsobem, který jsme demonstrováno výše. To následně umožní ukázat na příkladech způsoby vykonávání plánů tímto systémem a prozkoumat jeho výpočetní sílu.

3.8.1 Struktura ALLL systému s hierarchickými plány.

Systém bude opět obsahovat podsystém báze bez uspořádání, který tentokrát z názvu systému vypustíme a namísto toho do názvu zavedeme hierarchické plány, které je tento systém schopen interpretovat. V tomto případě bude systém čtveřice definovaná následovně.

Definice 64: ALLL systém s hierarchickými plány s bází bez uspořádání $\beta_L = (\Sigma, \Gamma, \gamma_0, \ltimes_{L,I}, \ltimes_{L,O})$ je uspořádaná čtveřice $\mathcal{S}_H = (\beta_L, \mathcal{R}^T, \vdash_H, n_0)$, kde β_L je báze systému, $\mathcal{R}^T$ je systém registrů, $\vdash_H$ jeho přechodová relace a $n_0 \in \Delta(\Sigma)$ je počáteční stav výpočtu.

V dalším textu bude zápis $\mathcal{S}_\mathrm{H}$ opět reprezentovat systém se strukturou uvedenou v předchozí definici a nebudeme ji dále rozepisovat. Báze systému bude také uváděna pouze jako β_L, a systém registrů jako $\mathcal{R}^T$. Posledně jmenovaný systém budeme blíže definovat v následujících odstavcích.

3.8.2 Systém registrů a jeho význam v $\mathcal{S}_H$ systémech

Systémem registrů budeme rozumět množinu symbolů, které mohou být substituovány v rámci tohoto systému navzájem podle relace, která je také součástí tohoto systému. Tato relace bude modelovat hierarchii registrů založenou na jejích věcích. Registr s věkem nula budeme značit symbolem τ. Celý systém formálně definujeme následovně.

Definice 65: Systém hierarchických plánů je uspořádaná dvojice $\mathcal{R}^T = (T, \uparrow)$, kde T je spočetná množina symbolů registrů obsahující symbol aktivního registru, $\tau \in T$, a $\uparrow$ je relace maturace registrů.

Množina symbolů registrů bude obsahovat atomy, které budou využívány k zápisu registrů ve strukturách plánů. Jak jsme již uvedli, těchto registrů se týká operace maturace. Operace maturace bude založena na relaci maturace v systému registrů, kterou si nyní nadefinujeme.

Definice 66: Relace maturace registrů v systému registrů $\mathcal{R}^T$ je bijektivní zobrazení množiny přirozených čísel na množinu symbolů registrů $\uparrow: \mathbb{N}^0 \to T$, pro které platí, že $(\uparrow (0) = \tau)$.

Relace maturace zobrazí každé přirozené číslo na jeden z prvků množiny symbolů registrů. Číslo, které zobrazují, odpovídá věku registru, který zobrazovaný symbol reprezentuje. Symbolem pro registr s věkem 0 zůstává symbol registru τ. Jelikož už máme systém registrů, můžeme přejít k novým operacím souvisejícím s hierarchickými plány.

3.8.3 Operace nad hierarchickými plány

Chování systému s hierarchickými plány bude vycházet z chování systémů s lineárními plány a rozšíří toto chování o dva nové způsoby přechodů mezi konfiguracemi systému. Pro oba tyto způsoby přechodů bude definováno odpovídající přechodové pravidlo, které bude součástí definice přechodové relace $\mathcal{S}_H$ systému a dalších systémů, které pracují s hierarchickými plány. První pravidlo se použije, pokud dojde k vyvolání podplánu akcí z kapitoly 3.7.3.. Během vyvolávání podplánu dojde k operaci maturace registrů podle přítomného systému registrů. Druhé pravidlo bude realizovat výběr podplánu z báze systému a jeho vyvolání. K oběma těmto pravidlům budeme potřebovat operace, které umožní patřičný přechod mezi konfiguracemi systémů a tyto operace představí následující odstavce.

3.8.3.1 Operace maturace plánů

V definici systémů registrů se vyskytla relace maturace registrů v systému registrů. Operace maturace bude vycházet z tohoto zobrazení a provede substituci některých atomů v plánu za jiné. Konkrétně se bude jednat o symboly z množiny symbolů registrů a ty budou nahrazeny jinými symboly z množiny registrů, nebo obecným prvkem nad množinou atomů systému. To už jsme ukázali dříve a nyní je třeba tuto operaci vytvořit formálně.

Operace maturace bude v podstatě aplikace množiny substitucí na plán podle definice 43. Jak bude substituční množina konkrétně vypadat, záleží na systému registrů. Substituční množinu ze systému registrů vytvoříme podle následující definice a nazveme ji maturační množinou.

Definice 67: Maturační množina systému $\mathcal{S}_H$ je definována pro systém registrů $\mathcal{R}^T$ a obecný prvek jako množina substitucí takto.

$$\mu(\mathcal{R}^T, t) = \left\{\left[\frac{\uparrow(n-1)}{\uparrow(n)}\right] : n \geq 2\right\} \cup \left\{\left[\frac{\tau}{\uparrow(1)}\right], \left[\frac{t}{\tau}\right]\right\}, t \in \Delta(\mathrm{T}, \Sigma)$$

Lemma 26: Maturační množina je substituční množina podle definice 40.

Důkaz: To, že se jedná o substituční množinu, pro kterou platí podmínky uvedené v definici 40, lze snadno ověřit. Substituční množina musí být uspořádaná a musí platit, že každý symbol je v této množině substituován maximálně jednou. Jelikož zobrazení ↑ je podle definice 66 bijekcí z množiny přirozených čísel, jsou obě tyto podmínky v definici 40 splněny.

Teď již můžeme definovat vlastní operaci maturace.

Definice 68: Operace maturace v S_H systému je struktura $M: H\big(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma)\big) \times \big((\Delta(\mathrm{T} \cup \Sigma) \cup T) \times T\big)^* \to H\big(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma)\big)$ definovaná jako

$$M(\pi, \mathcal{R}^T, t) = \pi\, \mu(\mathcal{R}^T, t)$$

Když už umíme provádět operaci maturace, můžeme přejít k operacím vyvolání plánu z báze, která hierarchické plány uchovává.

3.8.3.2 Báze pro uchování hierarchických plánů v S_H systémech

První možností, jak v systému S_H měnit strukturu aktuálně vykonávaného plánu, je umístit do něj jiný plán uložený v bázi plánů agenta. K tomu slouží akce nepřímého vyvolání plánu uvedená v definici 57. Aby báze mohla obsahovat prvky z množiny $H(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma))$, musí množina atomů báze zahrnovat celou množinu atomů, ze kterých se hierarchické plány tvoří. Podle definic 59, 61 a 62 se tvoří jako obecné prvky nad akcemi z množin $O_L(\mathrm{T}, \Sigma), O_{IC}(\mathrm{T}, \Sigma)$ a $O_{DC}^n(\mathrm{T}, \Sigma)$. Argumenty těchto akcí jsou obecné prvku nad množinami Σ a T. Symboly jednotlivých akcí jsou pak atomy z množiny $\{'+', '-', 'σ', '\sim', '@'\}$. Množina atomů pro zápis plánů jako obecného prvku je dána následující definicí.

Definice 69: Množina všech atomů pro zápis hierarchických plánu systému S_H je značena $\Sigma_H(\Sigma)$ a je definována takto.

$$\Sigma_H(\Sigma) = \Sigma \cup \mathrm{T} \cup \{'+', '-', 'σ', '\sim', '@', ' ' '\}$$

Lze uvést relaci mezi množinou hierarchických plánů a množinou obecných prvků nad množinou $\Sigma_H(\Sigma)$ jako $H(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma)) \subset \Delta(\Sigma_H(\Sigma))$. To znamená, že všechny hierarchické plány jsou obsaženy v množině obecných prvků nad atomy $\Sigma_H(\Sigma)$ a plyne to z definic hierarchických plánů a také z toho, že $\Delta(\Sigma_H(\Sigma))$ obsahuje všechny obecné prvky nad množinou atomů. $\Sigma_H(\Sigma)$.

Báze může být vytvořena jen pro nějakou podmnožinu z množiny hierarchických plánů, ovšem v tomto textu budeme dále uvažovat takové báze, které pokud slouží k uchovávání hierarchických plánů, pak umožňují vkládat, odstraňovat a vyhledávat kterýkoli z plánů nad danou množinou atomů Σ. Příslušná doména báze je proto celou potenční množinou nad množinou obecných prvků z dané množiny atomů. O tom, jaká musí být množina atomů takové báze, hovoří následující Lemma.

Lemma 27: Množina atomů Σ báze, která umožňuje ustanovení domény báze obsahující hierarchické plány, je nadmnožinou množiny atomů hierarchických plánů a množiny registrů, tedy $\Sigma \supseteq \mathrm{T} \cup \Sigma_H(\Sigma)$.

Důkaz: Množina atomů Σ báze by měla obsahovat všechny atomy pro tvorbu hierarchických plánů $\Sigma_H(\Sigma)$ a tedy $\Sigma \subseteq \Sigma_L(\Sigma)$. Jelikož podle definice 69 platí $\Sigma_H(\Sigma) \subseteq \Sigma$. Pokud bychom hledali Σ' takové, že $\Sigma_H(\Sigma') = \Sigma'$, pak hledáme pevný bod funkce Σ_H. Tímto pevným bodem jsou všechny množiny Σ', pro které plat, že $\Sigma' \subseteq \mathrm{T} \cup \{'+', '-', 'σ', '\sim', '@', ' ' '\}$. Tyto množiny pak jsou množinami atomů bází, které mohou obsahovat hierarchické plány nad jakoukoli množinou registrů T a množinou atomů Σ'', pro kterou platí $\Sigma'' \subseteq \Sigma'$.

3.8.3.3 Operace vyvolání podplánu v S_H systémech

Operace přímého vyvolání plánu bude jednodušší ze dvou operací, o které je systém s hierarchickými plány rozšířen oproti systému s lineárními plány. Výsledkem této operace bude samotný argument této operace. Základy pro operaci nepřímého vyvolání plánu z báze jsme představili před chvílí. Je prováděna tak, že vybereme vyvolávaný hierarchický plán z báze. K tomu využijeme operaci vyhledávání definovanou v kapitole o bázi bez uspořádání definicí 14. Budeme vybírat takový obecný prvek, jehož první podprvek není akce, ale jméno plánu, které je uvedeno jako argument akce nepřímého vyvolání podplánu. Výsledkem operace je v tomto případě hierarchický plán, který je druhým podprvkem vybraného prvku z báze.

Obě tyto operace budeme realizovat jednou funkcí, jejíž definice následuje.

Definice 70: Operace vyvolání plánu εx pro S_H systém a stav báze bez uspořádání γ je zobrazení $\varepsilon x: O_{DC}(\mathrm{T},\Sigma) \cup O_{IC}(\mathrm{T},\Sigma) \to H\big(\mathrm{T},\Sigma,O_L(\mathrm{T},\Sigma)\big)$ definované takto.

$\forall (op, arg) \in O_{DC}^{T}(\Sigma) \cup O_{IC}(\Sigma)$

$$\left(\varepsilon x\big((op,a)\big) = \begin{cases} \mathrm{a} \;\; \boldsymbol{iff} \;\; (op,a) \in O_{DC}(\mathrm{T},\Sigma), \;\; a \in H\big(\mathrm{T},\Sigma,O_L(\mathrm{T},\Sigma)\big) \\ \pi \;\; \boldsymbol{iff}\; (op,a) \in O_{IC}(\mathrm{T},\Sigma), \;\; \gamma \ltimes_{L,O} \Big(\sigma,\big(a\cdot(\sim)\big)\Big) = (ag,\pi), \pi \in H\big(\mathrm{T},\Sigma,O_L(\mathrm{T},\Sigma)\big) \\ (\Omega_F) \quad jinak \end{cases}\right)$$

Na výsledku této operace následně provedeme operaci maturace a poté maturovaný plán umístíme do právě vykonávaného plánu namísto akce, která tento plán vyvolávala. Toto ale bude až součástí pravidel v definicích patřičných přechodových funkcí systémů.

3.8.4 Konfigurace $\mathcal{S}_H$ systému

Konfigurace S_H systému bude obdobná struktura, jako byla konfigurace systému s lineárními plány. Jelikož se přeci jenom v jednom liší, uvedeme si ji novou definicí.

Definice 71: Konfigurace $\mathcal{S}_H$ systému je trojice (γ,π,n), kde první prvek je stav báze, $\gamma \in \Gamma$, druhým prvkem je hierarchický plán, $\pi \in H(\mathrm{T},\Sigma,O_L(\mathrm{T},\Sigma))$ a třetím prvkem je aktuální stav výpočtu, $n \in \Delta(\Sigma)$.

Všechny možné konfigurace tvoří doménu konfigurací.

Definice 72: Doména konfigurací $\mathcal{S}_H$ systému vyplývá z definice 71 a je to množina $Dom_C(\mathcal{S}_H) = \big\{(\gamma,\pi,n): \gamma \in \Gamma, \pi \in H\big(\mathrm{T},\Sigma,O_L(\mathrm{T},\Sigma)\big), n \in \Delta(\Sigma)\big\}$

Jedinou odlišností od konfigurace $\mathcal{S}_L$ systému z definice 47 je zde plán, který je tentokrát z množiny hierarchických plánů.

Běh $\mathcal{S}_H$ systému je posloupnost jeho konfigurací podle toho, jak bude probíhat výpočet na základě přechodové funkce $\mathcal{S}_H$ systému, kterou, jsme označili $\vdash_H$. Formálně se k běhu $\mathcal{S}_H$ systému dostaneme později. Zde si zatím uvedeme pouze to, že každý běh systému bude opět začínat v počáteční konfiguraci systému, která je dána strukturou $\mathcal{S}_H$ systému podle definice 64, a plánem, který je z množiny hierarchickým plánů a který je na tento systém aplikován. Počáteční a koncové konfiguraci tohoto systému zavedeme nyní formálně.

Definice 73: Pro $\mathcal{S}_H$ systém a plán $\pi \in H(\mathrm{T},\Sigma,O_L(\mathrm{T},\Sigma))$ je trojice $K_{0H}(\mathcal{S}_H,\pi) = (\gamma_0,\pi,n_0)$ počáteční konfigurací při aplikace plánu π na tento systém.

Definice 74: Konfigurace $(\gamma,\pi,n) \in Dom_C(\mathcal{S}_H)$ obsahující plán délky jedna, což podle definice 45 znamená $\pi \in \mathrm{L}^0\Big(\big(\mathrm{O}_L\,(\{\tau\},\Sigma)\big)\Big)$, kde $\mathrm{L}^0\Big(\big(\mathrm{O}_L\,(\{\tau\},\Sigma)\big)\Big) = \{(\Omega_T),(\Omega_F)\}$, je koncová konfigurace.

3.8.5 Přechodová relace $\mathcal{S}_H$ systémů

Výklad o systému s hierarchickými plány nyní zaměříme na definici přechodové relace tohoto systému. Přechodová relace $\mathcal{S}_H$ systému bude stejně jako přechodová relace $\mathcal{S}_L$ systému definována přechodovými pravidly. Tři přechodová pravidla jsou převzatá z lineárního sekvenčního systému, další dvě pravidla jsou do systému přidána, aby byl systém schopen zpracovávat hierarchii plánů.

Definice 75: Přechodová relace $\vdash_H \subseteq Dom_C(\mathcal{S}_H) \times Dom_C(\mathcal{S}_H)$ je definována následujícími pravidly uvádějícími jednotlivé vztahy mezi konfiguracemi systému pro možné aplikované akce. První tři pravidla se týkají akcí aplikovatelných na bázi systému.

(H-APPLY) $$\frac{\gamma \ltimes_{L,O}(\odot,p)[n]=n',\ (\odot,p)\in O_L\ (T,\Sigma),\ n' \neq \text{ı}\perp\text{ı},\odot\neq\text{ı}\sigma\text{ı}}{(\gamma,((\odot,p))\cdot\pi,n)\vdash_{H,APP}(\gamma \ltimes_{L,I}(\odot,p)[n],\pi,n)}$$

(H-TEST) $$\frac{\gamma \ltimes_{L,O}(\odot,p)[n]=n',\ (\odot,p)\in O_L\ (T,\Sigma),\ n'\neq\text{ı}\perp\text{ı},\odot=\text{ı}\sigma\text{ı}}{(\gamma,((\odot,p))\cdot\pi,n)\vdash_{H,TST}(\gamma,\pi,\mathrm{n}')}$$

(H-FAIL) $$\frac{\gamma \ltimes_{L,O}(\odot,p)[n]=\text{ı}\perp\text{ı}}{(\gamma,((\odot,p))\cdot\pi,n)\vdash_{H,FIL}(\gamma"(,\Omega_F)",\mathrm{n})}$$

Sjednocení relací budeme označovat výpočetní přechodovou relací, $\vdash_{H,CMP}$, a bude to relace

$$\vdash_{H,CMP}=\vdash_{H,APP}\cup\vdash_{H,TST}\cup\vdash_{H,FIL}$$

Další dvě pravidla jsou nová a budou použita při aplikacích akcí vyvolávání podplánů.

(H-PLAN_CALL_SUCC) $$\frac{(\odot,p)\in\left(O_{IC}(\mathrm{T},\Sigma)\cup O_{DC}(\mathrm{T},\Sigma)\right),\ \varepsilon x((\odot,p)[n])=p',\ (\gamma,p'\,\mu(\mathcal{R}^T,n),n)\vdash_H^*(\gamma\text{ı}",(\Omega_T)",n')}{(\gamma,((\odot,p))\cdot\pi,n)\vdash_{H,EXE}(\gamma',\pi,\mathrm{n}\text{ı})}$$

(H-PLAN_CALL_FAIL) $$\frac{(\odot,p)\in\left(O_{IC}(\mathrm{T},\Sigma)\cup O_{DC}(\mathrm{T},\Sigma)\right),\ \varepsilon x((\odot,p)[n])=p',p'="(\Omega_F)"\vee(\gamma,p'\mu(\mathcal{R}^T,n),n)\vdash_H^*(\gamma\text{ı},"(\Omega_F)",n')}{(\gamma,((\odot,p))\cdot\pi,n)\vdash_{H,EXE}(\gamma',\pi,\mathrm{n})}$$

Přechodová relace $\mathcal{S}_H$ systému je pak $\vdash_H=\vdash_{H,CMP}\cup\vdash_{H,EXE}$

Pravidla pro vyvolání podplánu jsou vytvořena tak, že uvádí vztah mezi dvěma konfiguracemi, kde v plánu první z konfigurací je první akcí akce vyvolání podplánu. Druhé konfigurace v těchto relacích jsou vytvořeny na základě toho, jaká je koncová konfigurace pro běh začínající v konfiguraci s aktuálním stavem báze, aktuálním stavem výpočtu a s podplánem, který byl vyvolán. Pokud koncová konfigurace obsahuje plán s akcí úspěšně ukončeného podplánu, pak platí pravidlo (H-PLAN_CALL_SUCC) a aktuální stav výpočtu je ten, kterým skončil výpočet s vyvolaným podplánem. Pokud vyvolaný plán neuspěje, platí pravidlo (H-PLAN_CALL_FAIL), a aktuální stavem výpočtu zůstane aktuální hodnota výpočtu původní. Báze je ale transformovaná vykonáváním podplánu v obou případech.

Lemma 28: Relace $\vdash_{H,CMP}$ je zobrazení.

Důkaz: Na základě podmínek u jednotlivých pravidel lze ukázat, že pro jakoukoli konfiguraci z domény konfigurací $Dom_C(\mathcal{S}_H)$ ve tvaru $(\gamma,((\odot,p))\cdot\pi,n)$ lze provést jen jedno z pravidel z definice 75. To je vybráno na základě výsledku aplikace bázové operace $\ltimes_{L,O}$ a dále podle symbolu akce v první akci plánu. Každé z pravidel pak jednoznačně určuje novou konfiguraci pro tuto relaci a proto je relace $\vdash_{H,CMP}$ zobrazením.

Na základě zápisu jednotlivých pravidel bychom mohli opět uvést, pro které $\mathcal{S}_H$ systémy se přechodové relace rovnají.

Lemma 29: Pro dva systémy $\mathcal{S}_{\mathrm{H1}} = \left((\Sigma_1, \Gamma_1, \gamma_{01}, \ltimes_{L,I1}, \ltimes_{L,O1}), \mathcal{R}_{T1}, \vdash_{H1}, n_{01}\right)$ a $\mathcal{S}_{\mathrm{H2}} = \left((\Sigma_2, \Gamma_2, \gamma_{02}, \ltimes_{L,I2}, \ltimes_{L,O2}), \mathcal{R}_{T2}, \vdash_{H2}, n_{02}\right)$ platí, že pokud $\Sigma_1 = \Sigma_2$, $\Gamma_1 = \Gamma_2$ a $\mathcal{R}_{T1} = \mathcal{R}_{T2}$ pak i $\vdash_{H1} = \vdash_{H2}$.

Důkaz: Stejně tak jako u důkazu Lemmy 19 začneme tím, že pro předpokládané rovností množin atomů a domén bází podle Lemmat 11 a 12 platí rovnosti $\ltimes_{L,I1} = \ltimes_{L,I2}$ a $\ltimes_{L,O1} = \ltimes_{L,O2}$. Domény obou přechodových relací $\{(\gamma_1, \pi_1, n_1) : \gamma_1 \in \Gamma_1, \pi_1 \in H(\mathrm{T}_1, \Sigma_1, O_L(\mathrm{T}_1, \Sigma_1)), n \in \Delta(\Sigma_1)\}$ a $\{(\gamma_2, \pi_2, n_2) : \gamma_2 \in \Gamma_2, \pi_2 \in H(\mathrm{T}_2, \Sigma_2, O_L(\mathrm{T}_2, \Sigma_2)), n_2 \in \Delta(\Sigma_2)\}$ se podle předpokládaných rovností opět rovnají. Pravidla pro definici relací jsou pro oba systémy stejná, protože se přechodové a výstupní funkce bází pro oba systémy rovnají a v pravidlech pro vyvolání plánů jsou použité systémy registrů také rovny. Díky těmto rovnostem lze i všechna pravidla považovat za stejná a tím i lze považovat za stejné přechodové relace, které definují.

Studium chování systému $\mathcal{S}_H$ nyní zaměříme na dva problémy a to jednak na to, zdali se jedná o deterministický nebo nedeterministický systém a poté na to, jaká je jeho výpočetní síla. Proto bude třeba umět simulovat provádění výpočtu takového systému.

3.8.6 Rozhodování o přijetí hierarchického plánu $\mathcal{S}_H$ systémy

Stejně jako u systémů s lineárními plány je i u systémů $\mathcal{S}_H$ důležité rozhodování o tom, zdali systém svoji počáteční konfiguraci přijímá, nebo nikoli. Myšlenka je opět založena na tom, že systém přijímá či nepřijímá svoji počáteční konfiguraci na základě koncové konfigurace, do které se může dostat přes svoji přechodovou relaci. Samozřejmě může nastat i případ, že počáteční konfigurace systému je zároveň jeho koncovou konfigurací, pak systém přijímá či nepřijímá tuto konfiguraci na základě akce, kterou obsahuje plán v této konfiguraci.

Nejprve zavedeme tranzitivní uzávěr přechodové relace systému.

Definice 76: Relace $\vdash_H^*$ je tranzitvní uzávěr přechodové relace $\vdash_H$.

A dále budeme formálně definovat, kdy systém $\mathcal{S}_H$ přijímá nebo nepřijímá plán.

Definice 77: Systém $\mathcal{S}_H$ je přijímající, nebo nepřijímající plán $\pi \in H\left(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma)\right)$, pokud $\pi = (\Omega_T)$ resp. $\pi = (\Omega_F)$, nebo pokud existuje relace $K_{0H}(\mathcal{S}_H, \pi) \vdash_H^* (\gamma', (\Omega_T), n')$ nebo $K_{0H}(\mathcal{S}_H, \pi) \vdash_H^* (\gamma', (\Omega_F), n')$.

Otázkou zůstává, jestli systém $\mathcal{S}_H$ může zároveň přijímat a nepřijímat nějaký plán. Abychom mohli ukázat, jestli je či není toto možné, uvedeme způsoby, jak lze koncové konfigurace $\mathcal{S}_H$ systému vypočítat. Potom uvidíme, že pokud je výpočet prováděn deterministickou funkcí, nebude možné, aby systém byl schopen nepřijmout plán, pokud plán je schopen přijmout a naopak.

3.8.6.1 Princip výpočtu koncových konfigurací $\mathcal{S}_H$ systémů

Pravidla z definice přechodové relace, která reprezentují vykonání podplánu, používají tranzitivní uzávěr přechodové relace, kterou spoludefinují. Na základě toho, že existuje taková relace pro odpovídající konfiguraci, která je dána aktuálními stavy báze a výpočtu a vyvolaným podplánem, a nějakou koncovou konfiguraci, je jedno ze dvou pravidel platné. Toto by se také dalo chápat tak, že pravidlo úspěšného volání plánu je použito, pokud nějaký jiný $\mathcal{S}_H$ systém přijímá onu konfiguraci po vyvolání podplánu, nebo pravidlo neúspěšného volání plánu je použito, pokud ji nepřijímá. Nyní je

třeba nalézt systém, na základě kterého bychom toto rozhodnutí o přijímání či nepřijímání uskutečňovali. To znamená, že bychom dokázali vypočítat, jestli taková relace v systému existuje, nebo neexistuje, a pokud ano, jestli ve své úspěšné nebo neúspěšné formě.

Následující Lemma představí systém, který danou přechodovou funkcí provádí výpočet z počáteční konfigurace, která obsahuje nějaký daný počáteční stav báze, počáteční stav výpočtu a hierarchický plán.

Lemma 30: Pokud systém $\mathcal{S}_H = ((\Sigma, \Gamma, \gamma_0, \ltimes_{L,I}, \ltimes_{L,O}), \mathcal{R}^T, \vdash_H, n_0)$ provádí výpočet pro nějaký plán $\pi \in H\big(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma)\big)$ přechodovou relací $\vdash_H$ ze své počáteční konfigurace (γ_0, π, n_0), pak systém, který provádí výpočet touto relací z počáteční konfigurace (γ_0', π', n_0'), kde $\gamma_0' \in \Gamma$, $\pi' \in H\big(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma)\big)$, $n_0' \in \Delta(\Sigma)$, je systémem $\mathcal{S}_H' = ((\Sigma, \Gamma, \gamma_0', \ltimes_{L,I}, \ltimes_{L,O}), \mathcal{R}^T, \vdash_H, n_0')$.

Důkaz: Podle Lemmy 29 platí pro systémy $\mathcal{S}_H = ((\Sigma, \Gamma, \gamma_0, \ltimes_{L,I}, \ltimes_{L,O}), \mathcal{R}^T, \vdash_H, n_0)$ a $\mathcal{S}_H' = ((\Sigma, \Gamma, \gamma_0', \ltimes_{L,I}, \ltimes_{L,O}), \mathcal{R}^T, \vdash_H', n_0')$ to, že $\vdash_H = \vdash_H'$. Proto také platí, že systém $\mathcal{S}_H'$ provádí výpočet přechodovou relací $\vdash_H$. Dále pro systém $\mathcal{S}_H'$ a plán $\pi' \in H(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma))$ je podle definice 73 počáteční konfigurace (γ_0', π', n_0'), pro kterou tento systém provádí výpočet, což je v souladu s uvedenou Lemmou.

Pokud budeme provádět výpočet nějakou přechodovou funkcí, pak pro různé počáteční konfigurace budou systémy, které tento výpočet provádějí, odlišné jen v počátečních vnitřních stavech báze a výpočtu. Následující konvence umožní vyjadřovat takovéto různé systémy jednoduchým zápisem.

Konvence 5: V dalších odstavcích budeme zápisem $\mathcal{S}_H \leftarrow \gamma, n$ vyjadřovat systém, který vznikne záměnou počátečních stavů báze a aktuálního stavu výpočtu systému $\mathcal{S}_H$ za hodnoty γ a n, pokud jsou tyto hodnoty z příslušných domén.

Základem výpočtu bude následujících pět formulí, které vychází z definice přechodové relace a vztahu uvedeného Lemmou 30. Tyto formule uvedeme poloformálně.

a) Systém $\mathcal{S}_H$ přijímá svoji počáteční konfiguraci (γ_0, π, n_0), pokud $\pi = "(\Omega_T)"$
b) Systém $\mathcal{S}_H$ nepřijímá svoji počáteční konfiguraci (γ_0, π, n_0), pokud $\pi = "(\Omega_F)"$
c) Systém $\mathcal{S}_H$ přijímá nebo nepřijímá svoji počáteční konfiguraci (γ_0, π, n_0), pokud existuje relace $(\gamma_0, \pi, n_0) \vdash_{H,CMP} (\gamma', \pi', n')$ a systém $\mathcal{S}_H' = ((\Sigma, \Gamma, \gamma', \ltimes_{L,I}, \ltimes_{L,O}), \mathcal{R}^T, \vdash_H, n')$ přijímá nebo nepřijímá konfiguraci (γ', π', n')
d) Systém $\mathcal{S}_H$ přijímá nebo nepřijímá svoji počáteční konfiguraci $(\gamma_0, (\odot, p) \cdot \pi, n_0)$, pokud $(\odot, p) \in O_{DC}^T(\Sigma) \cup O_{IC}^T(\Sigma)$ a systém $\mathcal{S}_H' = ((\Sigma, \Gamma, \gamma_0, \ltimes_{L,I}, \ltimes_{L,O}), \mathcal{R}^T, \vdash_H, n_0)$ přijímá svoji počáteční konfiguraci s plánem $\varepsilon\big((\odot, p)[n]\big)\, \mu(\mathcal{R}^T, n_0)$ v konfiguraci $(\gamma', "(\Omega_T)", n')$ a systém $((\Sigma, \Gamma, \gamma', \ltimes_{L,I}, \ltimes_{L,O}), \mathcal{R}^T, \vdash_H, n')$ přijímá nebo nepřijímá svou počáteční konfiguraci s plánem π.
e) Systém $\mathcal{S}_H$ přijímá nebo nepřijímá svoji počáteční konfiguraci $(\gamma_0, (\odot, p) \cdot \pi, n_0)$, pokud $(\odot, p) \in O_{DC}^T(\Sigma) \cup O_{IC}^T(\Sigma)$ a systém $\mathcal{S}_H' = ((\Sigma, \Gamma, \gamma_0, \ltimes_{L,I}, \ltimes_{L,O}), \mathcal{R}^T, \vdash_H, n_0)$ nepřijímá svoji počáteční konfiguraci s plánem $\varepsilon\big((\odot, p)[n]\big)\, \mu(\mathcal{R}^T, n_0)$ v konfiguraci $(\gamma', "(\Omega_F)", n')$ a systém $((\Sigma, \Gamma, \gamma', \ltimes_{L,I}, \ltimes_{L,O}), \mathcal{R}^T, \vdash_H, n)$ přijímá nebo nepřijímá svou počáteční konfiguraci s plánem π.

Uvedené formule nám umožní realizovat výpočet přechodových relací systému. V následujících podkapitolách si uvedeme dva přístupy k realizaci tohoto výpočtu. Prvním z nich bude zápis

rekurzivní funkce, která zobrazí výslednou koncovou konfiguraci systému, a druhým z nich bude rozvinutí této funkce pomocí kompozice $\mathcal{S}_H$ systémů.

3.8.6.2 Výpočty koncových konfigurací $\mathcal{S}_H$ systému rekurzivní funkcí

Rekurzivní funkci pro výpočet přechodové relace vytvoříme na základě formulí z předchozí podkapitoly. Její definice bude následující.

Definice 78: Funkce $\mathcal{F}_{SH}(\mathcal{S}_{\mathrm{H}}, \pi)$ pro daný systém a plán zobrazí koncovou konfiguraci, ve které systém přijímá či nepřijímá počáteční konfiguraci tohoto systému a plánu.

$$
\mathcal{F}_{SH}(\mathcal{S}_H, \pi)
$$

$$
= \begin{cases}
(\gamma_0, \pi, n_0)\ iff\ \pi \in \{"(\Omega_T)", "(\Omega_F)"\} \\
\quad \mathcal{F}_{SH}(\mathcal{S}_H \leftarrow (\gamma', n'), \pi')\ iff\ (\gamma, \pi, n) \vdash_{H,CMP} (\gamma', \pi', n') \\
\quad \mathcal{F}_{SH}(\mathcal{S}_H \leftarrow (\gamma', n'), \pi')\ iff \\
\qquad \pi = (\odot, p) \cdot \pi', (\odot, p) \in O_{IC}(T, \Sigma) \cup O_{DC}(T, \Sigma), \\
\qquad\quad \mathcal{F}_{SH}\left(\mathcal{S}_H \leftarrow (\gamma, n), \varepsilon x\big((\odot, p)[n]\big)\mu(\mathcal{R}^T, n)\right) = (\gamma', "(\Omega_T)", n') \\
\quad \mathcal{F}_{SH}(\mathcal{S}_H \leftarrow (\gamma', n), \pi')\ iff \\
\qquad \pi = (\odot, p) \cdot \pi', (\odot, p) \in O_{IC}(T, \Sigma) \cup O_{DC}(T, \Sigma), \\
\qquad\quad \mathcal{F}_{SH}\left(\mathcal{S}_H \leftarrow (\gamma, n), \varepsilon x\big((\odot, p)[n]\big)\mu(\mathcal{R}^T, n)\right) = (\gamma', "(\Omega_F)", n')
\end{cases}
$$

Způsob výpočtu koncové konfigurace rekurzivní funkcí $\mathcal{F}_{SH}$ je sice platný, ale není vhodný pro další výklad, protože tyto systémy budou pracovat v rámci větších systémů, ve kterých spolu budou vzájemně intereagovat. V některých fázích výpočtu bude třeba vzít ohled i na přijatá data z okolí. Budeme potom potřebovat využít jiný přístup k provádění plánu, který by umožňoval provádět výpočet po krocích, abychom mohli v těchto krocích reagovat na případné změny stavu okolí.

3.8.6.3 Výpočet koncové konfigurace $\mathcal{S}_H$ systémů a jejich kompozice

Pro účely realizace takového výpočtu si uvedeme ještě jednu funkci, která bude provádět vždy jeden krok výpočtu v systému $\mathcal{S}_H$. Tento krok výpočtu bude provádět nad nějakou posloupností konfigurací, která bude vyjadřovat stav výpočtu na jednotlivých úrovních rekurze. Využijeme známý princip převodu rekurzivního modelu výpočtu, který byl použit v předchozí podkapitole, na iterační způsob výpočtu, který použijeme nyní. Iterace zde bude obsažena v tom, že jednotlivé výpočetní kroky se provádí tak dlouho, dokud není dosažena koncová konfigurace a rekurzivní vyhodnocování podplánu je rozvinuto v posloupnost konfigurací. Každé spuštění podplánu znamená vytvoření nové konfigurace ke zpracování a její uložení do zásobníku konfigurací. Výpočet pak probíhá vždy pro konfiguraci na vrcholu zásobníku. Pokud je pro tuto konfiguraci nalezena koncová konfigurace, přejde se ke konfiguraci pod vrcholem zásobníku a tak dále, až po nalezení koncové konfigurace pro poslední konfiguraci v zásobníku.

Postupným prováděním kroků výpočtu vzniká běh systému. Běh v tomto případě je posloupnost kontextů výpočtů a kontext výpočtu je posloupnost konfigurací.

Definice 79: Množina $Dom_{\Phi}(\mathcal{S}_H) = Dom_{\mathrm{C}}(\mathcal{S}_H)^+$ je doménou kontextů výpočtu $\mathcal{S}_H$ systému

Konvence 6: Kontext výpočtu S_H nebo S_O systému budeme značit symbolem Φ, $\Phi \in Dom_{\Phi}(S_H) \cup Dom_{\Phi}(S_O)$ s případnými indexy.

Systém S_O a jeho kontext si představíme později. Nyní pokračujme s výkladem o S_H systémech.

Běh systému začíná kontextem obsahujícím počáteční konfiguraci systému, pro který se výpočet vykonává a prováděný plán, to znamená kontextem $K_{0H}(\mathcal{S}_H, \pi)$. V případě, že se dá aplikovat některé pravido pro relaci $\vdash_{H,CMP}$, zůstává počet konfigurací v kontextu stejný. Pokud v kontextu má dojít k provedení akce vyvolání podplánu, pak se kontext rozšíří o další konfiguraci systému, která pro konfiguraci ve tvaru $(\gamma_n, o \cdot \pi_n, n_n)$ odpovídá podle přpov9íslušných pravidel konfiguraci $(\gamma, \pi_n[n]\ \mu(\mathcal{R}^T, n), n)$. To znamená, že vznikne nová konfigurace pro stávající stav báze a výpočtu, ale pro nově vyvolaný plán, na kterém byla provedena operace maturace. Nový kontext výpočtu pak je v našem případě $\Phi'_H = \big((\gamma, \pi_n[n]\ \mu(\mathcal{R}^T, n), n), (\gamma_n, \pi_n, n_n)\big)$.

Běh skončí kontextem obsahujícím pouze některou z koncových konfigurací. Toto je obdobné tomu, jak je výpočetním krokem prováděn výpočet u systémů s lineárním plánem, viz. definice 56. I v případě S_H systému relací, kterou provádíme výpočet po krocích, říkáme relace výpočetního kroku, nebo výpočetní krok.

Zaveďme tedy relaci $\rhd_H$, která je výpočetním krokem systému S_H .

Definice 80: Pro systém S_H je výpočetním krokem struktura $\rhd_H \subseteq Dom_{\Phi}(S_H) \times Dom_{\Phi}(S_H)$ taková, že pro každou konfiguraci platí tato relace na základě některého z následujících pravidel.

$$(O - COMPUTE\ a)\quad \frac{\Phi_1 \vdash_{H,CMP} \Phi_1'}{(\Phi_1) \rhd_H (\Phi_1'), n \geq 1}$$

$$(O - COMPUTE\ b)\quad \frac{\Phi_n \vdash_{H,CMP} \Phi_n'}{(\Phi_n, \Phi_{n-1} \dots \Phi_1) \rhd_H (\Phi_n', \Phi_{n-1} \dots \Phi_1), n \geq 1}$$

$$(O - EXECUTE\ a)\quad \frac{(\odot, p) \in O_{IC}(T, \Sigma) \cup O_{DC}(T, \Sigma)}{\big((\gamma, ((\odot, p)) \cdot \pi, n)\big) \rhd_H \Big(\big(\gamma, \varepsilon\big((\odot, p)[n]\big)\mu(\mathcal{R}^T, n), n\big), (\gamma, \pi, n)\Big)}$$

$$(O - EXECUTE\ b)\quad \frac{(\odot, p) \in O_{IC}(T, \Sigma) \cup O_{DC}(T, \Sigma)}{\big((\gamma, ((\odot, p)) \cdot \pi, n), \dots \Phi_1\big) \rhd_H \Big(\big(\gamma, \varepsilon\big((\odot, p)[n]\big)\mu(\mathcal{R}^T, n), n\big), (\gamma, \pi, n), \dots \Phi_1\Big)}$$

$$(O - FINISH\_SUCC)\ \big((\gamma, "(\Omega_T)", n), (\gamma', \pi', n') \dots \Phi_1\big) \rhd_H \big((\gamma', \pi', n), \dots \Phi_1\big)$$

$$(O - FINISH\_FAIL)\ \big((\gamma, "(\Omega_F)", n), (\gamma', \pi', n') \dots \Phi_1\big) \rhd_H \big((\gamma', \pi', n'), \dots \Phi_1\big)$$

Výpočet aplikace plánu π systémem $\mathcal{S}_H$ probíhá od výpočetního kontextu s pouze počáteční konfigurací tohoto systému $K_{0H}(\mathcal{S}_H, \pi) = (\gamma_0, \pi, n_0)$ po jednotlivých krocích podle definice 80. Nyní si uvedeme formálně to, že běh systému je posloupnost kontextů výpočtu od počátečního kontextu výpočtu ke koncovému kontextu výpočtu, ze kterého již nelze učinit další výpočetní krok.

Definice 81: Běh systému $\mathcal{S}_H$ pro nějaký plán $\pi \in H(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma))$ je posloupnost $\mathcal{R}un_{SH}^{\pi} \in Dom_{\Phi}(S_H)^i, i \geq 1$ taková, že pokud $\mathcal{R}un_{SH}^{\pi} = (\Phi_{H1}, \Phi_{H2} \ldots \Phi_{Hn})$, pak $\Phi_{H1} = (K_{0H}(\mathcal{S}_H, \pi))$ a $\Phi_{Hn} = ((\gamma, \Omega, n))$ pro nějaká $\gamma \in \Gamma, n \in \Delta(\Sigma)$ a $\forall \Phi_{Hi}, \Phi_{Hi+1} \in Dom_{\Phi}(S_H)(\Phi_{Hi} \rhd_H \Phi_{Hi+1})$

Běh systému nám nyní umožní studovat determinismus chování $\mathcal{S}_H$ systému. V definici 64, která uvedla tento systém, hovoříme o přechodové relaci a ne o přechodovém zobrazení či funkci. Abychom přijali $\vdash_H$ za zobrazení a výpočet tímto zobrazením za deterministický, uvedeme si následující Lemmu.

Lemma 31: Krok výpočtu $\mathcal{S}_H$ systému $\rhd_H$ je zobrazení.

Důkaz: Definice 80 uvádí čtyři pravidla, které krok výpočtu definují. Rozdělíme množiny plánů $H(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma))$ na čtyři disjunktní podmnožiny $H = H_1 \cup H_2 \cup H_3 \cup H_4$ tak, že $H_1 = \{"(\Omega_T)"\}$, $H_2 = \{"(\Omega_F)"\}$ a množiny H_3 a H_4 obsahují plány rozdělené podle první akce plánu, a to zdali patří do množiny akcí $O_{IC}(T, \Sigma) \cup O_{DC}(T, \Sigma)$, či jde o množinu akcí nad bází $O_L(\mathrm{T}, \Sigma)$. To, že množina $H(\mathrm{T}, \Sigma, O_L(\mathrm{T}, \Sigma))$ je sjednocením těchto množin vyplývá z definic 59, 61 a 62. Množinu konfigurací systému jakožto relaci podle definic 71 a 72 lze obdobně rozdělit podle tohoto rozdělení množin plánů. Z definic 75 a 80 lze zjistit, že podmínky použití pravidel jsou takové, že pro jakýkoli kontext z množiny kontextů je použitelné nejvýš jedno pravidlo z definice 80 a to podle první konfigurace v tomto kontextu a podle toho, do jaké množiny plánů z H_1, H_2, H_3 nebo H_4 spadá plán v této konfiguraci. Jelikož platí Lemma 28, pak každé z těchto pravidel jednoznačně určuje nový kontext výpočtu pomocí přechodové relace $\mathcal{S}_H$ systémů.

Důsledkem Lemmy 31 je to, že systém nemůže zároveň přijímat a nepřijímat plán, jelikož koncová konfigurace pro daný systém a plán je jen jedna. Prováděním výpočetních kroků z počátečního kontextu výpočtu nelze dosáhnout kontextu s jedinou konfigurací a to takovou, že značí nepřijetí plánu a zároveň jiného kontextu s jedinou konfiguraci značící přijetí plánu. Důkaz by byl založen na tom, že z takovýchto konfigurací již nelze učinit výpočetní krok a tak při dosažení jedné není možné dosáhnout druhé a naopak.

Výše uvedené relace, zobrazení, běhy a další definice by si zasloužily ilustrační příklady. Ty poskytneme v další kapitole a zasadíme je do výkladu ohledně výpočetní síly systému $\mathcal{S}_H$.

3.8.7 Výpočetní síla $\mathcal{S}_H$ systémů

Výpočetní možnosti ALLL systémů prověříme pro nyní představovaný $\mathcal{S}_H$ systém a zjištění, která dosáhneme, nám umožní poznat sílu i následujících ALLL systémů. V této podkapitole si dokážeme následující Lemmu.

Lemma 32: Výpočetní síla ALLL systému s hierarchickými plány je stejná, jako je výpočetní síla univerzálního výpočetního systému.

Tato skutečnost bude platit i pro všechny další ALLL systémy, které v tomto textu uvedeme, protože budou zahrnovat výpočetní možnosti nyní zkoumaného systému. Pro dokázání Lemmy 32 ukážeme, jak by se ALLL systémem použil pro interpretaci skeletového jazyka, jenž je známým výpočetním systémem s univerzální výpočetní silou.

3.8.7.1 Reprezentace čísel v S_H systémech

Čísla jako taková nejsou v ALLL systémech přímo zavedena. Stejně tak nejsou zavedena například u λ-kalkulu a jsou kódována termy tohoto jazyka. V ALLL systémech budeme čísla kódovat jako plány tohoto systému. Budeme předpokládat, že v bázi systému jsou čísla ukládána společně s nějakým atomem, ke kterému jsou přiřazeny. Toto přiřazení je realizováno tím, že v bázi představ je uložen prvek (x), značící inicializaci proměnné x, a hodnota, která je v tomto případě z oboru přirozených čísel. V následujícím kódování čísel vždy uvedeme hodnotu, kterou reprezentují. Použijeme zkrácený tvar zápisu ALLL plánu, ve kterém akce nebudou uzavřeny do závorek a akce testování nebude mít uveden operátor σ. Pokud v plánu bude nějaká akce jen ve formě obecného prvku, budeme ji považovat za akci testování. Místo symbolu registru použijeme zápis 'tau', anonymní proměnná bude zapisována symbolem '_' a jednotlivé položky v plánu budou od sebe oddělovat čárky.

Kódování čísel v ALLL systémech je možné realizovat následovně.

0: (x)	((σ ((x))))
1: ((x),+((x)))	((σ ((x))) (+ (σ ((x)))))
2: ((x),+((x),+((x))))	((σ ((x))) (+ ((σ ((x))) (+ (σ ((x)))))))
3: ((x),+((x),+((x),+((x)))))	((σ ((x))) (+ ((σ ((x))) (+ ((σ ((x))) (+ (σ ((x)))))))))

… atd.

Každé číslo, až na nulu, je dvojice, kde prvním prvkem dvojice je akce testování na přítomnost inicializace dané proměnné a druhou je akce, která přidá do báze reprezentaci čísla o jedna menšího.

Každé číslo je tedy plán, a pokud je toto číslo uloženo v bázi, může být jako plán z báze vyvoláno. Struktura těchto plánů ale není vhodná pro to, aby se spouštěly nepřímým vyvoláním jménem plánu, a tak jsou volány akcí přímého spuštění plánu poté, co je plán přiřazen do registru po operaci testování báze. Jak toto konkrétně funguje, ukážeme dále. Abychom mohli ukázat, že ALLL systémy umožňují zapsat pomocí hierarchických plánů stejné výpočty, jaké umožňuje skeletový jazyk, musíme realizovat operace inkrementu proměnné, dekrementu proměnné a iterace "*while*" s podmínkou, že se proměnná nerovná nule. Před tím si ale ukážeme, jak lze v ALLL systémech realizovat podmíněný příkaz a jak iniciovat proměnnou.

Podmíněný příkaz je realizován jako plán se třemi akcemi. První akce přidává do báze příznak se jménem nějaké proměnné. Druhá akce vyvolá podplán o dalších třech akcích. První akce je abstraktní a má sloužit k vyhodnocení podmínky, například testováním báze. Pokud podmínka uspěje, je z báze odstraněn příznak a je proveden plán, který byl podmíněn. Pokud podmínka neuspěje, podplán neuspěje jako celek a příznak v bázi zůstává. Třetí akcí plánu na původní úrovni je opět akcí spuštění podplánu. Zde se testuje přítomnost příznaku v bázi, a pokud je příznak v bázi stále přítomen, odstraní se a provede se plán v 'Else' části podmíněného příkazu.

if *cond* then p1 else p2

$$(+(\mathrm{flg},x),@(\underline{cond},-(\mathrm{flg},x),p1),@((\mathrm{flg},x),-(\mathrm{flg},x),p2))$$

Inicializace proměnné bude složena také ze tří akcí. Předpokládáme, že aktuální hodnotou výpočtu je symbol proměnné v závorkách. První dvě akce plánu vymažou případnou existenci proměnné v bázi prvku a třetí akce vloží proměnnou inicializovanou na hodnotu 0. Pokud by se proměnná s hodnotou 0 v bázi nacházela, zůstane v bázi a inicializace i tak skončí úspěšně.

init tau

(init, ((tau,_),-(tau,_),+(tau)))

Dekrement proměnné je tou jednodušší z dvou operací s proměnnou, které ještě uvedeme. Čísla jsou v tomto textu kódována tak, aby právě dekrement šlo učinit pouze tím, že se kód, který číslo reprezentuje, provede. V okamžiku odstranění aktuální reprezentace čísla pro danou proměnnou je v registru uložena tato reprezentace a po jejím spuštění se do báze uloží reprezentace čísla o jedna menšího.

dec tau

(dec, (-(tau,_),@tau'))

Inkrement proměnné je realizován trochu složitěji. V tomto případě je nutné ošetřit situaci, kdy hodnota proměnné je rovna nule. Plán reprezentující nulu má totiž odlišnou formu od reprezentací ostatních čísel. Zde využijeme výše uvedené konstrukce podmíněného příkazu a vyrobíme následující plán.

inc tau

```
(inc , (
        +(flag,tau),
        @( (tau,_),-(tau,_), +(tau,+tau'), -(flag,tau) ) ,
        (flag,tau), +(tau, +(tau)) ,-(flag,tau))
)
```

Jako první nastavíme návěští v souladu s konstrukcí podmíněného příkazu. Dále spustíme podplán, ve kterém testujeme bázi na přítomnost čísla většího než nula provedením akce testování (tau,_). Pokud tato akce projde, odstraníme tento prvek z báze a přidáme do něj prvek reprezentující o jedničku větší prvek. Zároveň odstraníme návěští, abychom neprováděli plán v části ‘else’. Pokud ovšem návěští neodstraníme, což se stane, pokud test na přítomnost prvku většího než nula v bázi neprojde, pak provedeme test báze na existence nuly, to znamená, zdali prvek v bázi je inicializován, a pokud ano, tak do báze přidáme prvek reprezentující jedničku.

Poslední struktura, která nám chybí k tomu, abychom ukázali výpočetní univerzálnost systému, je konstrukce podmíněné iterace. Ta funguje tak, že plán, který iteraci reprezentuje, nejprve provede testování přítomnosti čísla většího než nula pro atom, který je aktuální hodnotou výpočtu, a pokud takové číslo existuje, provedeme tělo iterace a snížíme pro daný atom číslo o jedničku provedením operace dekrementu. Předtím si testováním inicializované proměnné nahradíme aktuální hodnotu výpočtu jménem proměnné, se kterou pracujeme. Poslední akce plánu tento plán znovu spustí.

while (tau!=0) do plan

(while, ((tau,_), … plan … , (tau), @(dec), @(while))

Fungování těchto operací ukážeme na příkladech inkrementace a dekrementace čísla.

Příklad 19: Jako první zkusíme provést operaci inkrementace v situaci, kdy v bázi je přítomen prvek (x), který značí, že proměnná x je inicializována a jelikož dále není v bázi žádný prvek odpovídající reprezentaci čísla pro atom x, znamená to, že hodnota x je nula. Budeme provádět výpočet pro následující počáteční konfiguraci.

$(\gamma, \pi, n)=$ ({(x)},
(
+(flag,tau),
@((tau,_),-(tau,_), +(tau,+tau''), -(flag,tau)) ,
(flag,tau), +(tau, +(tau)) ,-(flag,tau)
)
,(x))

Pro lepší přehlednost budeme jednotlivé kroky výpočtu zapisovat do tabulky. Začneme v okamžiku, kdy je plán spuštěn a je na něm provedena operace maturace.

Krok	Báze	Plán	Aktivní	Akce k provedení
1.	(x)	+(flag, (x)),@(((x),_),-((x),_), +((x),+tau'), -(flag, (x))) , (flag, (x)), +((x), +((x))) ,-(flag, (x)))	(x)	+(flag,(x))
2.	(x),(flag,(x))	(@(((x),_),-((x),_), +((x),+tau'), -(flag, (x))) , (flag, (x)), +((x), +((x))) ,-(flag, (x)))	(flag, (x))	@(((x),_),-((x),_), +((x),+ tau), -(flag, (x)))

Druhý krok způsobí rozšíření kontextu výpočtu o další konfiguraci, která bude reprezentovat výpočet podplánu. Následující tabulka uvede výpočet tohoto podplánu s tím, že se jedná jen o část celého kontextu výpočtu. K původní konfiguraci se vrátíme po skončení výpočtu podplánu.

Krok	Báze	Plán	Aktivní	Akce k provedení
2.1	(x),(flag,(x))	(((x),_),-((x),_), +((x),+tau), -(flag, (x)))	(flag, (x))	((x),_)
2.2	(x),(flag,(x))	(-((x),_), +((x),+tau), -(flag, (x)))	(flag, (x))	Fail

Testování na prvek větší než nula neuspělo a tak se kontext výpočtu redukuje opět pouze na původní konfiguraci. Výpočet pak pokračuje následovně.

Krok	Báze	Plán	Aktivní	Akce k provedení
3.	(x), (flag,(x))	((flag,(x)), +((x), +((x))) ,-(flag, (x)))	(flag, (x))	(flag, (x))
4.	(x),(flag,(x))	(+((x), +((x))) ,-((x), (x)))	(flag, (x))	+((x),+((x)))
5.	(x),(flag,(x)), ((x),+((x)))	(-(flag, (x)))	(flag, (x))	-(flag,(x))
6.	(x), ((x),+((x)))	(Ω_T)	(flag, (x))	Ω_T

V bázi se nachází prvek reprezentující jedničku pro atom x a výpočet skončil úspěšně.

Přiklad 20: Navážeme na předchozí příklad a zkusíme tentokrát provést operaci inkrementace čísla pro proměnnou x.

Krok	Báze	Plán	Aktivní	Akce k provedení
1.	(x), ((x),+((x)))	+(flag, (x)),@(((x),_),-((x),_), +((x),+tau'), -(flag, (x))) , (flag,tau), +(tau, +(tau)) ,-(flag,tau))	(flag, (x))	+(flag,(x))
2.	(x), ((x),+((x))) ,(flag,(x))	(@(((x),_),-((x),_), +((x),+tau'), -(flag, (x))) , (flag,tau), +(tau, +(tau)) ,-(flag,tau))	(flag, (x))	@(((x),_),-((x),_), +((x),+tau), -(flag,(x)))
2.1	(x), ((x),+((x))) ,(flag,(x))	(((x),_),-((x),_), +((x),+tau), -(flag,(x)))	(flag, (x))	((x),_)
2.2	(x), ((x),+((x))) ,(flag,(x))	(-((x),_), +((x),+tau), -(flag,(x)))	((x),+((x)))	-((x),_)
2.3	(x) ,(flag,(x))	(+((x),+tau), -(flag,(x)))	((x),+((x)))	+((x),+ ((x),+((x))))
2.4	(x) ,(flag,(x)), ((x),+ ((x),+((x))))	(-(flag,(x)))	((x),+((x)))	-(flag,(x))
2.5	(x) , ((x), + ((x),+((x))))	(Ω_T)	((x),+((x)))	(Ω_T)
3	(x) , ((x), + ((x),+((x))))	((flag, (x)), +((x), +((x))) ,-(flag, (x)))	((x),+((x)))	((flag, (x))
4	(x) , ((x), + ((x),+((x))))	(Ω_F)	((x),+((x)))	Ω_F

V bázi se nyní nachází atom inicializované proměnné v závorkách a hodnota tohoto atomu odpovídající číslu 2. Plán sice skončil jako neúspěšný, ale jen proto, že jsme prováděli akci testování na příznak pro případné provedení části 'else'. To by se dalo napravit tím, že by se operace inkrementu spustila jako podplán a jeho případný neúspěch nebude mít vliv na úspěšné dokončení plánu na vyšší úrovni.

3.9 Systém otevřeného ALLL prvku s hierarchickými plány a vstupní bází

Abychom mohli zasadit ALLL systém jako podsystém do nějakého většího systému, ve kterém budou tyto systémy pracovat paralelně a komunikovat zasíláním zpráv, vytvoříme třetí a v této kapitole poslední ALLL systém na úrovni prvku, který vykonává nějaký plán. Jeho název je "Otevřený ALLL systém s hierarchickými plány a vstupní bází". Název je to komplikovaný, proto jej budeme zkracovat na tvar "otevřený ALLL prvek", "otevřený prvek", nebo jej budeme reprezentovat symbolem S_O, obdobně jak jsme to dělali u předchozích systémů. Obdobný bude i sled výkladu, kdy začneme definicí struktury systému, přejdeme k sekvencím, které představují akce, jenž mohou být součástí plánu aplikovatelných na tento systém, dále uvedeme také patřičné přechodové funkce, abychom nakonec probrali plány jako struktury přijímaných akcí a způsob přijímání těchto plánů.

3.9.1 Struktura otevřeného ALLL systému

Oproti předchozím systémům bude tento systém obsahovat dvě báze, jednu pro reprezentaci dat a druhou jako vyrovnávací paměť pro zprávy, které obdrží od ostatních prvků v systému. Jelikož tento systém budeme používat v rámci většího systému, ve kterém budou tyto otevřené prvky adresovány podle svých jmen, budou i tato jména součástí systému otevřeného prvku. Přejděme nyní k definici jeho struktury.

Definice 82: Otevřený ALLL prvek je uspořádaná šestice $S_O = (\eta, \beta_L, \beta_Q, \mathcal{R}^T, \vdash_A, n_0)$ kde $\beta_A = (\Sigma, \Gamma_L, \gamma_0, \ltimes_{L,I}, \ltimes_{L,O})$ je báze systému, $\beta_Q = (\Sigma, \Gamma_Q, \mathrm{C}_0, \ltimes_{Q,I}, \ltimes_{Q,O})$ je vyrovnávací paměť zpráv, $\mathcal{R}^T$ je systém registrů, η je jméno z množiny atomů bází, $\eta \in \Sigma$, $\vdash_{A,I}$ je přechodová relace a n_0 je počáteční stav výpočtu.

Otevřený systém obsahuje dvě báze obou uvedených typů, to znamená báze bez uspořádání a s uspořádáním. Tyto báze budeme dále zkráceně nazývat báze a vyrovnávací paměť. Množina atomů báze a vyrovnávací paměti jsou stejné. Víme z předchozích kapitol, že tyto množiny také určují, jak vypadají množiny akcí a plánů, se kterými systém, který báze zahrnuje, pracuje. Domény báze a vyrovnávací paměti se ale mohou lišit (a také liší) a proto jsou označeny rozdílně. Zatímco doména báze není nějak dále specifikována, doména vyrovnávací paměti má svoji specifickou strukturu.

3.9.1.1 Doména vyrovnávací paměti zpráv a reprezentace zpráv S_O systémů

Podle definice 20 je systém báze s uspořádáním pětice $\beta_Q = (\Sigma, \Gamma_Q, \mathrm{C}_0, , \ltimes_{Q,I}, \ltimes_{Q,O})$ a je mimo jiné tvořen množinou atomických prvků Σ a doménou Γ_Q. Doménu tohoto systému budou tvořit množiny uspořádaných dvojic, kde prvním prvkem každé dvojice je jméno odesilatele z Σ a druhým je zpráva ve formě obecného prvku z $\Delta(\Sigma)$. Formálně bude doména podle následující definice.

Definice 83: Doména vyrovnávací paměti systému S_O je potenční množina $\Gamma_Q = \wp\left(D^1(\Sigma) \cdot D^1\left(\Delta(\Sigma)\right)\right)$

V definici jsme použili operaci spojení dvou lineárních prvků délky jedna a to nad množinou atomů a nad množinou obecných prvků nad množinou atomů. Takto vytvořená potenční množina je platnou doménou otevřeného prvku.

Lemma 33: Doména Γ_Q je platnou doménou báze s uspořádáním.

Důkaz: Jelikož podle definic 2 až 6 platí $D^1(\Sigma) \subseteq \Delta(\Sigma)$ a z definic 4,5 a 6 vyplývá, že $D^1\big(\Delta(\Sigma)\big) \subseteq \Delta(\Sigma)$. Pak i $D^1(\Sigma) \cdot D^1\big(\Delta(\Sigma)\big) \subseteq \Delta(\Sigma)$ podle Lemmy 4 a definic 2 až 6. Potenční množina $\wp\left(D^1(\Sigma) \cdot D^1\big(\Delta(\Sigma)\big)\right)$ je podmnožinou $\wp\big(\Delta(\Sigma)\big)$. Podle definice 20 je tedy Γ_Q platnou doménou báze s uspořádáním.

Připomínáme, že podle definice 21 jsou konfigurace báze s uspořádáním dvojice stavu této báze z množiny, která je prvkem domény báze Γ_Q, a relace ostrého uspořádání nad touto množinou. Zprávy v bázi jsou proto uspořádány a v reálných systémech jsou uspořádány podle toho, jak byly do báze umístěny podle principu FIFO.

3.9.2 Komunikační akce S_O systému

Ještě jednou a tentokrát naposledy budeme definovat množiny, které budou obsahovat akce a plány aplikovatelné na ALLL systémy. Tentokrát z množiny atomických prvků vytvoříme množiny pro komunikační akce, to je pro zasílání a přijímání zpráv, a tyto akce začleníme do hierarchického plánu.

3.9.2.1 Realizace komunikace v ALLL distribuovaných systémech

Nejprve je vhodné ukázat, jak je celý proces komunikace v ALLL systémech pojat. V distribuovaných ALLL systémech budou otevřené ALLL systémy tvořit prvky, které spolu komunikují skrz nějaký komunikační kanál. Adresace jednotlivých prvků je činěna uvedením jména prvku, které mohou mít ALLL prvky uloženo ve svých bázích. Směrování komunikace v případě, že neexistuje přímý komunikační kanál mezi prvkem, který zprávu odesílá, a prvkem, který je adresován, je nad rámec této práce. V systémech, které budeme dále v textu popisovat, budeme předpokládat, že pokud jeden ALLL prvek poslal zprávu s uvedeným jménem jiného ALLL prvku a tento prvek se v systému nachází, pak mu bude zpráva doručena.

3.9.2.2 Akce odeslání obecného prvku a vybrání z vyrovnávací paměti zpráv

Syntakticky budeme nové akce definovat stejně, jak jsme to dělali u předchozích systémů. Tyto dosud neuvedené akce budou dvou typů. První z těchto typů akcí bude na výstup systému předávat zprávu k zaslání, zatímco druhý typ bude vybírat přijatou zprávu z vyrovnávací paměti zpráv. Definici obou typů následují.

Definice 84: Akce odeslání zprávy v S_O systému s množinou atomických prvků Σ a množinou registrů T tvoří množinu $O_{SN}(,\Sigma) = \{'(' '!' '('name, msg')' ')' : name \in \mathrm{T}\cup\Sigma, msg \in \Delta(\mathrm{T}\cup\Sigma)\}$

Definice 85: Akce výběru zprávy z vyrovnávací paměti v S_O systému s množinou atomických prvků Σ a množinou registrů T tvoří množinu

$$O_{RC}(\mathrm{T},\Sigma) = \{'(' '?' '('name, msg')' ')' : name \in \mathrm{T}\cup\Sigma\cup\{\sim\}, msg \in \Delta(\mathrm{T}\cup\Sigma\cup\{\sim\})\cup\{\sim\}\}$$

3.9.2.3 *Hierarchické plány s komunikačními akcemi*

Množiny hierarchických plánů pro nějakou množinu akcí O, jsme získávali konstruktorem $H(\Sigma, \mathrm{T}, O)$ a tento konstruktor bude použit i pro definici množiny hierarchických plánů, které jsou aplikovatelné na S_O systém.

Definice 86: Množina hierarchických plánů aplikovatelná na S_O systém s množinou atomických prvků Σ a množinou registrů T je množina $H\left(\Sigma, \mathrm{T}, O_L\ (\mathrm{T}, \Sigma) \cup O_{RC}(\mathrm{T}, \Sigma) \cup O_{SN}(\mathrm{T}, \Sigma)\right)$

Podle této definice jsou aplikovatelné na plány S_O systém hierarchickými plány, které jako akce mohou kromě akcí pro práci s bází a akce vyvolávání podplánů obsahovat i akce pro odesílání zpráv a jejich výběr z vyrovnávací paměti.

3.9.3 Konfigurace S_O systémů

I v případě S_O systému bude přechodová relace zobrazením definovaným pro množiny konfigurací tohoto systému. U otevřených systémů zavedeme do struktury konfigurace tři nové prvky, a to jméno prvku, které bude sloužit k adresování zpráv, a dále aktuální data na vstupech a výstupech prvku. Definice konfigurace systému S_O je potom následující.

Definice 87: Konfigurace S_O systému je uspořádaná sedmice $(\eta, \iota, o, \gamma, \mathrm{C}, \pi, n)$, kde $\eta \in \Sigma$ je jméno systému jako prvku distribuovaného systému, $\iota \in \left(D^1(\Sigma) \cdot D^1\left(\Delta(\Sigma)\right)\right)^* \cup \phi$ je stav na vstupu, $o \in \left(D^1(\Sigma) \cdot D^1\left(\Delta(\Sigma)\right)\right) \cup \phi$ je stav na výstupu, $\gamma \in \Gamma_L$ a $\mathrm{C} \in \mathrm{C}(\Gamma)$ jsou konfigurace báze a vyrovnávací paměti, $\pi \in H\left(\Sigma, \mathrm{T}, O_L\ (\mathrm{T}, \Sigma) \cup O_{RC}(\mathrm{T}, \Sigma) \cup O_{SN}(\mathrm{T}, \Sigma)\right)$ je plán a $n \in \Delta(\Sigma)$ je stav výpočtu systému.

Použitý atom pro vstupní hodnoty ϕ znamená nepřítomnost dat na vstupu nebo výstupu systému. Jako symbol pro reprezentaci konfigurací otevřených systémů budeme opět používat symbol K s případnými indexy.

Pro pořádek a následné použití v definici přechodové funkce definujeme také doménu konfigurací otevřeného systému jako následující množinu.

Definice 88: Doména konfigurací $\mathcal{S}_O$ systému je množina

$$Dom_{\mathrm{C}}(\mathcal{S}_O) = \left\{ \begin{array}{c} (\eta, \iota, o, \gamma, \mathrm{C}, \pi, n): \eta \in \Sigma, \iota \in \left(D^1(\Sigma) \cdot D^1\left(\Delta(\Sigma)\right)\right)^* \cup \phi, o \in \left(D^1(\Sigma) \cdot D^1\left(\Delta(\Sigma)\right)\right) \cup \phi, \gamma \in \Gamma_L, \\ \mathrm{C} \in \mathrm{C}\left(\Gamma_Q\right), \pi \in H\left(\Sigma, \mathrm{T}, O_L\ (\mathrm{T}, \Sigma) \cup O_{RC}(\mathrm{T}, \Sigma) \cup O_{SN}(\mathrm{T}, \Sigma)\right), n \in \Delta(\Sigma) \end{array} \right\}$$

Pokud máme definované struktury konfiguraci systému, můžeme přejít k definici jeho přechodové relace.

3.9.4 Přechodová relace S_O systému

Jelikož se jedná o otevřený systém, mohli bychom přechodovou relaci definovat tak, že bychom ji definovali jako relaci složenou ze vstupní funkce, která zpracovává data na vstupu a funkce, která provádí transformaci stavu prvku a předává data na jeho výstup. Tak také i učiníme, jelikož takovéto rozdělení umožní lépe popsat chování systémů jako dvoukrokové. Přechodovou relaci otevřeného ALLL systému $\mathcal{S}_O$ označíme jako $\vdash_O$. relace, které budeme nyní definovat, označíme jako $\vdash_{O,INP}$ a

$\vdash_{O,EVL}$, kde první relace bude zpracovávat hodnoty na vstupech prvku a druhá bude provádět akce plánu a předávat data na výstup prvku.

3.9.4.1 Funkce zpracování vstupu S_O systémem

Vstupy budeme zpracovávat funkcí. Tato funkce má zobrazovat konfigurace tak, že případné vstupní data v konfiguraci uloží do vyrovnávací paměti a v následné konfiguraci systému se tato data na vstupech už neobjeví. Vstupní data i stav vyrovnávací paměti jsou podle definice 87 v konfiguracích zahrnuta a tak lze tyto funkce definovat jako zobrazení z nějaké konfigurace $\mathrm{K}_1 = (\eta, \iota, \gamma, \mathrm{C}, n, o)$ na konfiguraci $\mathrm{K}_2 = (\eta, \phi, \gamma, \mathrm{C}', n, o)$. Kromě nahrazení dat na vstupu symbolem značící nepřítomnost dat na vstupu dojde ke změně konfigurace vyrovnávací paměti. Formálně tuto funkci uvádí následující definice.

Definice 89: Funkce zpracování vstupu otevřeného ALLL systému je zobrazení $\vdash_{A,INP}: Dom_{\mathrm{C}}(\mathcal{S}_A) \rightarrow Dom_{\mathrm{C}}(\mathcal{S}_O)$, definovaná následovně

$$(\eta, (\iota_1, \iota_2 \ldots \iota_m), o, \gamma, \mathrm{C}, \pi, n) \vdash_{A,INP} (\eta, \phi, o, \gamma, \iota_m \hookrightarrow (\ldots \iota_2 \hookrightarrow (\iota_1 \hookrightarrow \mathrm{C})), \pi, n)$$

$$(\eta, \phi, o, \gamma, \mathrm{C}, \pi, n) \vdash_{A,INP} (\eta, \phi, o, \gamma, \mathrm{C}, \pi, n)$$

Data na vstupech jsou umístěna do vyrovnávací paměti postupnou aplikací operace vkládání do báze s uspořádáním a ve výsledné konfigurace se na místě dat na vstupu objeví symbol ϕ. Pokud na vstupech nejsou žádná data, konfigurace se aplikací funkce zpracování vstupu nezmění.

3.9.4.2 Relace aplikací akcí a výpočet S_O systémem

Definici relace aplikací akcí učiníme opět pomocí pravidel. Jednotlivá pravidla budou definovat relace provedení akcí pro jednotlivé akce, která jsou aplikovatelná na $\mathcal{S}_O$systém. Pět ze sedmi pravidel je převzato z předchozích systémů, nová jsou pravidla pro nově zavedené akce a jsou uvedena na konci. Jejich složením pak dostaneme pravidla pro aplikací skupin akcí na systém a z nich přechodovou relaci systému.

Definice 90: Relace aplikací akcí $\vdash_{A,APP}, \vdash_{A,TST}, \vdash_{A,FIL}, \vdash_{A,EXE}, \vdash_{A,SND}, \vdash_{A,RCV} \subseteq Dom_{\mathrm{C}}(\mathcal{S}_O) \times Dom_{\mathrm{C}}(\mathcal{S}_O)$ jsou definována pomocí následujících pravidly, která uvádí jednotlivé vztahy mezi konfiguracemi.

(APPLY-A) $$\frac{(\odot,p)\in O_L\ (T,\Sigma),\gamma\ltimes_{L,O}(\odot,p)[n]=n',\ \ n'\neq\text{ı}\perp\text{ı},\odot\neq\text{ı}\sigma\text{ı}}{(\eta,\phi,o,\gamma,\mathrm{C},((\odot,p))\cdot\pi,n)\vdash_{A,APP}(\eta,\phi,\phi,\gamma\ltimes_{L,I}(\odot,p)[n],\mathrm{C},\pi,\mathrm{n})}$$

(TEST-A) $$\frac{(\odot,p)\in O_L\ (T,\Sigma),\gamma\ltimes_{L,O}(\odot,p)[n]=n',\ \ n'\neq\text{ı}\perp\text{ı},\odot=\text{ı}\sigma\text{ı}}{(\eta,\phi,o,\gamma,\mathrm{C},((\odot,p))\cdot\pi,n)\vdash_{A,TST}(\eta,\phi,\phi,\gamma,\mathrm{C},\pi,\mathrm{n}')}$$

(FAIL-A) $$\frac{\gamma\ltimes_{L,O}(\odot,p)=\text{ı}\perp\text{ı}}{(\eta,\phi,o,\gamma,\mathrm{C},((\odot,p))\cdot\pi,n)\vdash_{A,FIL}(\eta,\phi,\phi,\gamma,\mathrm{C},\text{" }(\Omega_F)\text{"},\mathrm{n})}$$

(PCALL SUCC-A) $$\frac{(\odot,p)\in O_{IC}(T,\Sigma)\cup O_{DC}(T,\Sigma),\varepsilon x((\odot,p)[n])=p'\ \ (\eta,\phi,\phi,\gamma,\mathrm{C},p'\ \mu(\mathcal{R}^T,n),n)\vdash_A^*(\eta,\phi,\phi,\gamma\text{ı},\mathrm{C}',(\text{"}\Omega_T\text{"}),n')}{(\eta,\phi,\phi,\gamma,\mathrm{C},((\odot,p))\cdot\pi,n)\vdash_{A,EXE}(\eta,\phi,\phi,\gamma',\mathrm{C},\pi,\mathrm{n}\text{ı})}$$

(PCALL FAIL-A)

$$\frac{(\odot,p)\in O_{IC}(T,\Sigma)\cup O_{DC}(T,\Sigma),\ \varepsilon x((\odot,p)[n])=p',p'=(\text{"}\Omega_F\text{"})\ \vee\ (\eta,\phi,\phi,\gamma,\mathrm{C},p'\mu(\mathcal{R}^T,n),n)\vdash_A^*(\eta,\phi,\phi,\gamma',\mathrm{C}',(\text{"}\Omega_F\text{"}),n')}{(\eta,\phi,\phi,\gamma,\mathrm{C},((\odot,p))\cdot\pi,n)\vdash_{A,EXE}(\eta,\phi,\phi,\gamma',\mathrm{C},\pi,\ \mathrm{n})}$$

(SEND - A) $$\frac{(\odot,p)\in O_{SN}(T,\Sigma),\ (\odot,p)[n]=(\odot,p')}{(\eta,\phi,\phi,\gamma,\mathrm{C},((\odot,p))\cdot\pi,n)\vdash_{A,SND}(\eta,\phi,p',\gamma,\mathrm{C},\pi,n)}$$

(RECEIVE – A) $$\frac{(\odot,p)\in O_{RC}(T,\Sigma),\ (\odot,p)[n]=(\odot,p')}{(\eta,\phi,\phi,\gamma,\mathrm{C},((\odot,p))\cdot\pi,n)\vdash_{A,RCV}(\eta,\phi,\phi,\gamma,p\prime\swarrow C,\pi,p\prime\hookleftarrow C)}$$

Relace reprezentující provádění podplánů využívá tranzitivního uzávěru přechodové relace $\mathcal{S}_O$ systému, kterou teprve budeme definovat. Zdali tato relace existuje nebo neexistuje, závisí opět na tom, zdali lze pro nějaký jiný systém $\mathcal{S}_O$ s nějakou počáteční konfigurací nalézt koncovou konfiguraci nebo ne. To jsme blíže probírali v kapitole o $\mathcal{S}_H$ systémech a při hledání takové relace jsme dospěli k provádění kroků výpočtu. K těm také dospějeme i u $\mathcal{S}_O$ systému. Nejprve ale budeme definovat složené relace provádění akcí a z nich přechodovou relaci otevřeného ALLL systému.

Definice 91: Relace aplikace atomické akce $\mathcal{S}_O$ systému je $\vdash_{A,ACT}=\vdash_{A,CMP}\cup\vdash_{A,APP}\cup\vdash_{A,TST}\cup\vdash_{A,FIL}\cup\vdash_{A,SND}\cup\vdash_{A,RCV}$

Definice 92: Relace aplikace obecné akce $\mathcal{S}_O$ systému je $\vdash_{A,EVL}=\vdash_{A,ACT}\cup\vdash_{A,EXE}$

Opět budeme používat infixový zápis pro tyto relace, to znamená, že pokud jsou konfigurace K_1 a K_2 v relaci $(\mathrm{K}_1,\mathrm{K}_2)\in\vdash_{A,EVL}$, pak tuto relaci zapíšeme jako $\mathrm{K}_1\vdash_{A,EVL}\mathrm{K}_2$.

3.9.4.3 Přechodová relace $\mathcal{S}_O$ systému

Přechodová relace otevřeného ALLL systému je funkce složená z relace zpracování vstupů a relace aplikace obecné akce $\mathcal{S}_O$ systému.

Definice 93: Přechodová relace $\mathcal{S}_O$ systému je relace

$$\vdash_A=\{(\mathrm{K}_1,\mathrm{K}_2):\forall\mathrm{K}_1,\mathrm{K}_2\in Dom_{\mathrm{K}}(\mathcal{S}_O)\ \exists\mathrm{K}_3\in Dom_{\mathrm{K}}(\mathcal{S}_O):(\mathrm{K}_1\vdash_{A,INP}\mathrm{K}_3)\wedge(\mathrm{K}_3\vdash_{A,EVL}\mathrm{K}_2)\}$$

Lemma 34: Přechodová relace $\mathcal{S}_O$ systému je zobrazení

Důkaz: Relace $\vdash_{A,INP}$ je zobrazení. To plyne přímo z její definice 89. Pravidla definující relace $\vdash_{A,CMP},\vdash_{A,APP},\vdash_{A,TST},\vdash_{A,FIL}$a $\vdash_{A,EXE}$ vychází ze systému $\mathcal{S}_H$ a pro ten jsme ukázali, že jejich sjednocení tvoří zobrazení. Přidáním relací $\vdash_{A,SND}$ a $\vdash_{A,RCV}$ vzhledem k pravidlům a podmínkám aplikovatelnosti těchto pravidel zůstane výsledná relace zobrazením. Obě pravidla jsou aplikovatelná na plán s první akcí typu zaslání či vybrání zprávy a definují jednoznačně přechod mezi konfiguracemi systému.

Pro další výklad budeme potřebovat podmnožinu přechodové relace takovou, kterou můžeme bez dalších výpočtů v tomto systému určit a tím je kompozice funkce zpracování vstupu a relace aplikace atomické akce. Nazveme ji relace provedení atomické akce.

Definice 94: Relace provedení atomické akce ALLL systémem je relace

$$\vdash_{A,E}=\{(\mathrm{K}_1,\mathrm{K}_2):\forall\mathrm{K}_1,\mathrm{K}_2\in Dom_{\mathrm{K}}(\mathcal{S}_O)\ \exists\mathrm{K}_3\in Dom_{\mathrm{K}}(\mathcal{S}_O):(\mathrm{K}_1\vdash_{A,INP}\mathrm{K}_3)\wedge(\mathrm{K}_3\vdash_{A,ACT}\mathrm{K}_2)\}$$

Abychom dokázali reálně provádět výpočet tímto systémem, budeme potřebovat relaci výpočetního kroku, jejíž obdobu jsme k výpočtu používali i u systémů s hierarchickými plány. Takovou relaci si nyní vytvoříme.

3.9.4.4 *Krok výpočtu S_O systému*

Krok výpočtu $\mathcal{S}_{\mathrm{O}}$ systému bude definován stejně jako v případě systému $\mathcal{S}_{\mathrm{H}}$ na kontextech výpočtu systému. Kontext výpočtu je opět posloupnost konfigurací $\mathcal{S}_{\mathrm{O}}$ systému, což zavádí následující definice.

Definice 95: Množina $Dom_{\Phi}(\ S_O\ ) = Dom_{\mathrm{C}}(\ S_O\ )^{+}$ je doménou kontextů výpočtu systémem S_O

Princip definice relace kroku výpočtu pro systém S_O bude podobný principu definice relace kroku výpočtu pro systém $\mathcal{S}_H$.

Definice 96: Pro systém S_O je krok výpočtu struktura $\triangleright \subseteq Dom_{\Phi}(S_O) \times Dom_{\Phi}(S_O)$ taková, že pro dvojice kontextů výpočtů platí relace na základě některého z následujících pravidel.

$$(O-COMPUTE\ \mathrm{a})\quad \frac{(\mathrm{K} \vdash_{A,E} \mathrm{K}')}{(\mathrm{K}) \triangleright (\mathrm{K}')}$$

$$(O-COMPUTE\ b)\quad \frac{(\mathrm{K}_n \vdash_{A,E} \mathrm{K}_n')}{(\mathrm{K}_n, \mathrm{K}_{n-1} \dots \mathrm{K}_1) \triangleright (\mathrm{K}_n', \mathrm{K}_{n-1} \dots K), n \geq 1}$$

$$(O-EXECUTE\ a)$$

$$\frac{\begin{array}{c}(\odot, p) \in O_{IC}(T, \Sigma) \cup O_{DC}(T, \Sigma), \\ \Big((\eta, i, o, \gamma, \mathrm{C}, ((\odot, p)) \cdot \pi, n)\Big) \vdash_{A,INP} \Big((\eta, \phi, o, \gamma, \mathrm{C}', ((\odot, p)) \cdot \pi, n)\Big)\end{array}}{\begin{array}{c}\Big((\eta, i, o, \gamma, \mathrm{C}, ((\odot, p)) \cdot \pi, n)\Big) \triangleright \\ \left(\left(\eta, \phi, o, \gamma, \mathrm{C}', \varepsilon\left((\odot, p)\left[\frac{n}{\tau}\right]\right)\mu(\mathcal{R}^T, n), n\right), (\eta, \phi, o, \gamma, \mathrm{C}', \pi, n)\right)\end{array}}$$

$$(O-EXECUTE\ b)$$

$$\frac{\begin{array}{c}(\odot, p) \in O_{IC}(T, \Sigma) \cup O_{DC}(T, \Sigma), \\ \Big((\eta, i, o, \gamma, \mathrm{C}, ((\odot, p)) \cdot \pi, n), \dots K_1\Big) \vdash_{A,INP} \Big((\eta, \phi, o, \gamma, \mathrm{C}', ((\odot, p)) \cdot \pi, n), \dots K_1\Big)\end{array}}{\begin{array}{c}\Big((\eta, i, o, \gamma, \mathrm{C}, ((\odot, p)) \cdot \pi, n), \dots K_1\Big) \triangleright \\ \left(\left(\eta, \phi, o, \gamma, \mathrm{C}', \varepsilon\left((\odot, p)\left[\frac{n}{\tau}\right]\right)\mu(\mathcal{R}^T, n), n\right), (\eta, \phi, o, \gamma, \mathrm{C}', \pi, n) \dots K_1\right)\end{array}}$$

$$(O-FINISH\_SUCC)\ \big((\eta, \iota, o, \gamma, \mathrm{C}, "(\Omega_F)", n), (\eta, \iota', o', \gamma', \mathrm{C}', \pi', n') \dots\big) \triangleright \big((\eta, \iota, o, \gamma, \mathrm{C}, \pi', n), \dots\big)$$

$$(O-FINISH\_FAIL)\ \ \big((\eta, \iota, o, \gamma, \mathrm{C}, "(\Omega_F)", n), (\eta, \iota', o', \gamma', \mathrm{C}', \pi', n') \dots\big) \triangleright \big((\eta, \iota, o, \gamma, \mathrm{C}, \pi', n'), \dots\big)$$

$$(O-TERMINATED)\ \big((\eta, \iota, o, \gamma, \mathrm{C}, \Omega, n)\big) \triangleright \big((\eta, \iota, o, \gamma, \mathrm{C}, \Omega, n)\big)$$

Provedení výpočetního kroku skrz tyto relace má opět dvě části – zpracování vstupních dat a provedení akce. V případě atomických akcí je toto provedeno již v definici relace $\vdash_{A,E}$, pro zbývající 'neatomické' akce, to znamená pro akce vyvolání podplánu, je relace výpočtu definována nově, ale také v souladu s definicí 94. Poslední tři pravidla se týkají situace, kdy první z konfigurací je koncovou konfigurací. V případě, že v kontextu výpočtu existuje nějaká další konfigurace, jsou na ni přeneseny patřičné části. Výpočetní kroky jsou definovány i pro kontexty výpočtu, které obsahují pouze

koncovou konfiguraci. Je to z důvodu použití této relace pro činnost prvku v distribuovaných systémech, kde bude potřeba provádět nějaký výpočetní krok i u systémů, které zpracování plánů už ukončily.

Ještě než přejdeme k distribuovaným systémům, uvedeme si pár vět k otázce výpočetní síly otevřeného ALLL systému.

3.9.5 Přijímání plánu $\mathcal{S}_O$ systémem

Výpočetní síla otevřeného ALLL systému se neliší od výpočetní síly systému s hierarchickými plány. Přidáním dvou nových typů akcí dojde k dojde rozšíření množin plánů a podle Lemmy 3 platí, že $H\left(\Sigma, \mathrm{T}, O_L\ (\mathrm{T}, \Sigma)\right) \subset H\left(\Sigma, \mathrm{T}, O_L\ (\mathrm{T}, \Sigma) \cup O_{RC}(\mathrm{T}, \Sigma) \cup O_{SN}(\mathrm{T}, \Sigma)\right)$. Pokud je na vstupech symbol ϕ a plány jsou z množiny plánů hierarchického ALLL systému, která je podmnožinou množiny plánů tohoto otevřeného systému, pak každý $\mathcal{S}_O$ systém interpretuje plán z $H\left(\Sigma, \mathrm{T}, O_L\ (\mathrm{T}, \Sigma)\right)$ stejně, jak by jej interpretoval $\mathcal{S}_H$ systém. To proto, že pravidla pro akce, které můžou být součástí těchto plánů, jsou pro oba systémy stejná.

$\mathcal{S}_\mathrm{O}$ systémy v tomto textu uvádíme proto hlavně, protože budou součástí většího systému, který může zahrnovat více $\mathcal{S}_\mathrm{O}$ prvků. Výpočet bez interakcí s okolím, který jsme uvažovali dosud, je sice možný, ale nemá smysluplné použití. Proto na závěr této teoretické kapitoly zmíníme systémy, ve kterých by tyto otevřené ALLL systémy, které dále budeme nazývat ALLL prvky, fungovaly jako distribuované výpočetní jednotky. Uvidíme, že chování distribuovaného systému lze definovat podle výpočetních kroků jednotlivých prvků, kdy v každém kroku distribuovaného systému dojde k provedení výpočetního kroku jednoho nebo více ALLL prvků.

3.10 Distribuované ALLL systémy

Otevřený systém může plnohodnotně fungovat jen v hierarchicky nadřazeném systému, kde tvoří jen jeden z podsystémů tohoto systému. Teoretickou kapitolu zakončíme představením těchto distribuovaných systémů, kde opět ukážeme jejich strukturu a chování, včetně chování otevřených ALLL prvků, které obsahuje.

3.10.1 Principy distribuovaných ALLL systémů

V případě distribuovaných ALLL systémů postupujeme na vyšší úroveň v hierarchii systémů a budeme sledovat, jak jednotlivé prvky mezi sebou interagují. Interakce mezi jednotlivými ALLL prvky bude prováděna asynchronním předáváním zpráv, které je řešeno ukládáním zpráv do vyrovnávacích pamětí jiných prvků.

Pokud bychom měli tento systém klasifikovat a shrnout doposud představené části, pak by se jednalo o model paralelního distribuovaného systému s asynchronním předáváním zpráv. Jednotlivé prvky v systému mají nějaké chování, přičemž pokud jde o ALLL prvky, pak je jejich chování řízeno programem, který má formu hierarchických plánů s akcemi pro asynchronní komunikaci, manipulaci s bází dat a s akcemi modifikujícími strukturu plánu.

Distribuovaný ALLL systém bude tvořen množinou prvků, které budou spolu svázány vzájemnými vazbami. Obecně systém definujeme jako dvojici $S=(U,R)$, kde universem je množina prvků $U = \{u_1, u_2 \ldots u_n\}$, která v našem případě může obsahovat také otevřené ALLL systémy. V množině R se nachází relace mezi těmito prvky, které nyní budou reprezentovat komunikační kanály mezi nimi. Ve všech systémech, o kterých budeme v rámci této kapitoly hovořit, budeme předpokládat plně

propojený systém, tedy že existuje relace reprezentující komunikační kanál mezi každou dvojicí prvků v systému a to v obou směrech. Dále předpokládáme, že jména prvků jsou pro každý prvek unikátní.

Jednotlivé prvky budou mít nějaké chování. Pokud se jedná o ALLL prvky, je jejich chování dáno plánem, který interpretují. Jiné prvky v tomto systému mohou mít chování odlišné a pro účely následujících kapitol je budeme považovat za abstraktní v tom smyslu, že toto chování bude reprezentovat nějaká dále nedefinovaná funkce. V případě distribuovaných ALLL systémů nebudeme definovat přechodové relace způsobem, který jsme používali v případech výpočetních ALLL prvků, to jest jako vztah mezi konfiguracemi těchto prvků, ale uvedeme přímo relace výpočetních kroků pro složené systémy, nebo přesněji řečeno pro jejich výpočetní kontexty. Chování všech následujících systémů bude probíhat ve dvou krocích.

1. Předání zpráv na výstupech prvků na vstupy adresovaných prvků
2. Provedení jednoho výpočetního kroku všech nebo některých prvků systému

Následující odstavce představí celkem tři ALLL distribuované systémy. První dva z nich budou obsahovat pouze ALLL otevřené systémy jako své prvky. Ukážeme na nich principy interakce mezi jednotlivými prvky a to v případě synchronního systému, kdy každý prvek v systému provede jeden krok výpočtu během jednoho kroku provádění distribuovaného systému, a dále v případě asynchronního systému, kdy pouze část prvků může v daný okamžik provést výpočetní krok. Třetí distribuovaný systém, který zmíníme, může obsahovat i jiné prvky než jsou ALLL prvky.

3.10.2 Struktury a funkce v distribuovaných ALLL systémech

Některé formální struktury budou společné pro následující distribuované systémy, proto si je představíme zvlášť ještě před popisem konkrétních systémů.

Začneme funkcí, která pro každý výpočetní kontext ALLL prvku zobrazí data, která se nachází na jeho výstupech. K těm navíc přidá jméno prvku z první konfigurace kontextu jako jméno odesilatele těchto dat. Dále uvedeme množinu, která bude definována na základě množiny konfigurací prvků a bude obsahovat všechna výstupní data z těchto konfigurací. Poslední z konstrukcí budou množiny, které budou obsahovat výstupní data podle jména adresovaného prvku.

Definice 97: Funkce $out: Dom_{\Phi}(S_O) \rightarrow \left(D^1(\Sigma) \cdot D^1(\Delta(\Sigma))\right) \cup \phi$ zobrazí výpočetní kontext otevřeného ALLL prvku na data na jeho výstupu.

$$\forall\left((\eta, \iota, \phi, \gamma, \mathsf{C}, \pi, n)\right) \in Dom_{\mathsf{C}}(S_O) \cdot \left(out\left((\eta, \iota, \phi, \gamma, \mathsf{C}, \pi, n)\right) = \phi\right)$$

$$\forall\left((\eta, \iota, (rc, d), \gamma, \mathsf{C}, \pi, n)\right) \in Dom_{\mathsf{C}}(S_O) \cdot \left(out\left((\eta, \iota, (rc, d), \gamma, \mathsf{C}, \pi, n)\right) = (\eta, rc, d)\right)$$

$$\forall\left((\eta, \iota, (rc, d), \gamma, \mathsf{C}, \pi, n), K_2 \ldots K_n\right) \in Dom_{\Phi}(S_O) \cdot$$

$$\left(out\left(\left((\eta, \iota, (rc, d), \gamma, \mathsf{C}, \pi, n), K_2 \ldots K_n\right)\right) = (\eta, rc, d)\right), n \geq 2.$$

Definice 98: Pro množinu výpočetních kontextů $\{\Phi_1, \Phi_2 \ldots \Phi_n\}$ je množina aktivních dat definována jako $\Psi(\{\Phi_1, \Phi_2 \ldots \Phi_n\}) = \bigcup_{i=1..n} out(\Phi_i) - \phi$

Díky tomu, že každý prvek množiny $\Psi(\{\Phi_1, \Phi_2 \ldots \Phi_n\})$ je trojice, kde první prvek je jméno odesilatele a druhý je jméno adresáta, můžeme rozložit tuto množinu na disjunktní podmnožiny

právě podle těchto jmen. Následující definice uvede konstruktor množiny, která bude obsahovat aktivní data adresovaná prvku se jménem uvedeným jako argument konstruktoru.

Definice 99: Množina dat zasílaná prvku rc je

$$\mathrm{Data}(\{\Phi_1, \Phi_2 \dots \Phi_n\}, rc) = \{(\eta, d) : (\eta, rc, d) \in \Psi(\{\Phi_1, \Phi_2 \dots \Phi_n\})\}$$

Dvojice, které tvoří tuto množinu, a které obsahují jméno odesilatele a nějaká data, budeme nazývat zprávami.

Zprávy jsou předávaná na vstup adresovanému prvku ve formě uspořádané posloupnosti, která je ve formě obecného ALLL prvku. Abychom z množiny získali takovou uspořádanou posloupnost, použijeme abstraktní, blíže nedefinované zobrazení *sort*, které neprázdnou množinu zpráv $\{(sn_1, d_1), (sn_2, d_2) \dots\}$ zobrazí na uspořádanou posloupnost zpráv ve formě obecného ALLL prvku. To v našem případě bude prvek z $\left(D^1(\Sigma) \cdot D^1(\Delta(\Sigma))\right)^+$ ve formě $\left((sn_1, d_1), (sn_2, d_2) \dots\right)$

Definice 100: Operace seřazení vstupních dat *sort* je struktura $sort: \wp\left(D^1(\Sigma) \cdot D^1(\Delta(\Sigma))\right) \longrightarrow \left(D^1(\Sigma) \cdot D^1(\Delta(\Sigma))\right)^+$

Způsob uspořádání dat na vstupu ponecháme pro konkrétní realizace těchto systémů.

Konvence 7: Abychom v dalším textu zlepšili čitelnost zápisů množiny zasílaných zpráv, zavedeme zápis $Data_\eta^{\overline{\Phi}} \equiv sort\left(\mathrm{Data}(\overline{\Phi}, \eta)\right)$, kde $\overline{\Phi} = \{\Phi_1, \Phi_2 \dots \Phi_n\}$

3.10.3 Homogenní synchronní distribuovaný ALLL systém s ALLL prvky

Homogenní distribuovaný ALLL systém s ALLL prvky zkráceně nazýváme homogenním ALLL systémem a skládá se pouze z otevřených ALLL prvků. Kromě toho budeme ještě předpokládat, že počet prvků v systému se neměnní, že báze a vyrovnávací paměti těchto prvků budou mít stejné domény a také že množina atomických prvků bude u všech ALLL prvků stejná. V čem se prvky liší, jsou jejich počáteční konfigurace a také plány, které vykonávají.

3.10.3.1 Struktura $\mathcal{S}_{DS}$ systému

Homogenní synchronní systém je tvořen ALLL prvky a relací, která definuje chování tohoto systému. Relaci charakteristiky systému v tomto případě uvádět nebudeme, protože předpokládáme plně propojený systém. Definice homogenního synchronního distribuovaného systému je tato.

Definice 101: Homogenní synchronní distribuovaný systém $\boldsymbol{\mathcal{S}_{DS}} = (\mathcal{S}_{O1}, \mathcal{S}_{O2} \dots \mathcal{S}_{On}, \triangleright_{DS})$ je složen z ALLL otevřených prvků $\mathcal{S}_{O1}, \mathcal{S}_{O2} \dots \mathcal{S}_{On}$ a $\triangleright_{DS}$ je jeho přechodová relace.

Opět zápis $\boldsymbol{\mathcal{S}_{DS}}$ bude dále reprezentovat takovýto homogenní synchronní distribuovaný systém.

Zatímco ALLL otevřené prvky jsme již představili v minulé kapitole, přechodovou relaci a související struktury je třeba nyní definovat. Přechodová relace zde udává vztah mezi dvěma stavy výpočtu homogenního ALLL systému. V případě systému s více ALLL prvky je tento stav dán výpočetními kontexty jednotlivých prvků. Nejedná se tedy pouze o konfigurace těchto prvků, ale o celé jejich kontexty tak, jak vznikají během provádění jednotlivých výpočetních kroků těmito systémy. Doménu konfigurací pro systém s *n* ALLL otevřenými prvky uvádí následující definice.

Definice 102: Doména stavů $\boldsymbol{\mathcal{S}_{DS}}$ systému je značena $Dom_{\mathrm{K}}(\boldsymbol{\mathcal{S}_{DS}})$ a je to množina $Dom_{\mathrm{K}}(\boldsymbol{\mathcal{S}_{DS}}) = \{\{\Phi_{O1}, \Phi_{O2}, \dots \Phi_{On}\}: \Phi_{O1} \in Dom_{\Phi}(\mathcal{S}_{O1}), \Phi_{O2} \in Dom_{\Phi}(\mathcal{S}_{O2}) \dots \Phi_{On} \in Dom_{\Phi}(\mathcal{S}_{On})\}$

3.10.3.2 Relace předávání zpráv v $\mathcal{S}_{DS}$ systému

Předání zpráv je modelováno relací mezi stavy distribuovaného systému, tedy množinami kontextů jednotlivých ALLL prvků, které zahrnuje. Je to také vztah mezi daty na výstupech jednotlivých prvků a daty na vstupech adresovaných prvků. Doména přípustných adres v systému $\boldsymbol{\mathcal{S}_{DS}}$ je množinou všech jmen ALLL prvků v systému. Kdo je adresátem které zprávy, udává první prvek v odesílané zprávě. Pokud na výstupech nejsou žádná data, což reprezentuje atom ϕ, nebo pokud v systému není adresovaný prvek, k přenosu zpráv nedojde. Jelikož předpokládáme úplné propojení prvků komunikačními kanály, dojde k přenesení zprávy mezi libovolnou dvojicí prvků, pokud je mezi nimi zasílána zpráva a ta má adresáta.

Definice 103: Relace předávání zpráv je relace mezi stavy systému $\boldsymbol{\mathcal{S}_{DS}}$, a je to následující struktura.
$\triangleright_{DS,MSG}: Dom_{\mathrm{K}}(\boldsymbol{\mathcal{S}_{DS}}) \rightarrow Dom_{\mathrm{K}}(\boldsymbol{\mathcal{S}_{DS}})$

$$\{\Phi_{O1}, \Phi_{O2}, \dots \Phi_{On}\} \triangleright_{DS,MSG} \{\Phi_{O1}', \Phi_{O2}', \dots \Phi_{On}'\}$$

Kde $\Phi_{O1} = \big((\eta_1, \phi, o_1, \gamma_1, \mathrm{C}_1, \pi_1, n_1), \dots\big), \Phi_{O2} = \big((\eta_2, \phi, o_2, \gamma_2, \mathrm{C}_2, \pi_2, n_2), \dots\big) \dots$

Označíme $\bar{\Phi} = \{\Phi_{O1}, \Phi_{O2}, \dots \Phi_{On}\}$

$$Pak\ platí\quad \Phi_{Oi}' = \begin{cases} \big((\eta_i', Data_{\eta i}^{\bar{\Phi}}, \phi', \gamma_i', \mathrm{C}_i', \pi_i', n_i') \dots\big) & pokud\ Data_{\eta i}^{\bar{\Phi}} \neq \{\} \\ \big((\eta_i', \phi', \phi', \gamma_i', \mathrm{C}_i', \pi_i', n_i') \dots\big) & jinak \end{cases}, \boldsymbol{1 \leq i \leq n}$$

V této definici existuje relace pouze mezi takovými kontexty, kde v prvním kontextu aktuální konfigurace nemá data na vstupech. To znamená, že prvek nemůže mít nezpracovaná data na vstupech, aby mohl provést operaci modelovanou touto relací. To může být problém v případě asynchronních systémů, což probereme v kapitole o těchto systémech.

3.10.3.3 Relace provedení kroku prvků $\mathcal{S}_{DS}$ systému

Relace provedení kroku výpočtu prvků homogenního synchronního distribuovaného systému je definovaná pro kontexty výpočtů prvků z universa, kde první konfigurace těchto kontextů neobsahují data na výstupech. Předpokládá se, že tato data byla odstraněna při předávání dat na vstupy příslušných prvků. Realizace kroku výpočtu prvků je založena na výpočetních krocích jednotlivých prvků systému $\mathcal{S}_{DS}$ podle následující definice.

Definice 104: Relace provedení kroku výpočtu prvků v systému $\boldsymbol{\mathcal{S}_{DS}}$ je struktura $\boldsymbol{\mathcal{S}_{DS}}: \triangleright_{DS,EVL}: Dom_{\mathrm{K}}(\boldsymbol{\mathcal{S}_{DS}}) \rightarrow Dom_{\mathrm{K}}(\boldsymbol{\mathcal{S}_{DS}})$

$$\frac{\Phi_{A1} \triangleright \Phi_{A1}, \Phi_{A2} \triangleright \Phi_{A2} \dots \Phi_{An} \triangleright \Phi_{An}}{\{\Phi_{A1}, \Phi_{A2}, \dots \Phi_{An}\} \triangleright_{DS,EVL} \{\Phi_{A1}', \Phi_{A2}', \dots \Phi_{An}'\}}$$

K přechodu mezi stavy systému $\mathcal{S}_{DS}$ dochází, pokud existují relace výpočetního kroku mezi výpočetními kontexty jednotlivých prvků systému.

3.10.3.4 Přechodová relace S_{DS} systému

Přechodová relace $\boldsymbol{S_{DS}}$ systému je kompozicí předchozích dvou předchozích relací, tedy relace předávání zpráv a relace provedení kroku prvky $\boldsymbol{S_{DS}}$ systému.

Definice 105: Přechodová relace $\boldsymbol{S_{DS}}$ systému je struktura $\triangleright_{DS}: Dom_{\mathrm{K}}(\boldsymbol{S_{DS}}) \rightarrow Dom_{\mathrm{K}}(\boldsymbol{S_{DS}})$ a je definovaná jako množina

$$\triangleright_{DS} = \{(\Phi_1, \Phi_2): \forall \Phi_1, \Phi_2 \in Dom_{\mathrm{K}}(\boldsymbol{S_{DS}})\ \exists \Phi_3 \in Dom_{\mathrm{K}}(\boldsymbol{S_{DS}}): (\Phi_1 \triangleright_{DS,MSG} \Phi_3) \wedge (\Phi_3 \triangleright_{DS,EVL} \Phi_2)\}$$

Celý systém funguje ve dvou krocích, jak bylo uvedeno na začátku této kapitoly. Po přenesení dat z výstupů na vstupy dojde k provedení výpočetního kroku u všech prvků v systému. Pokud existuje relace přechodového kroku pro všechny kontexty ALLL prvku, což by se dalo ukázat z definice 96, lze pro jakýkoliv stav $\boldsymbol{S_{DS}}$ systému provést přechodovou relaci.

Fungování systému $\boldsymbol{S_{DS}}$ demonstrujeme na příkladu.

Příklad 21: Jako příklad uveďme dva $\boldsymbol{S_O}$ systémy $("ag1", \beta_L, \beta_Q, \mathcal{R}^T, \vdash_A, "(a)")$ a $("ag2", \beta_L, \beta_Q, \mathcal{R}^T, \vdash_A, "(a)")$ s nějakými bázemi a systémy registrů, kde jeden prvek zasílá druhému zprávu, druhý uloží nějaký prvek do báze, pak přijme zprávu z vyrovnávací paměti a bude testovat přítomnost zprávy ve své bázi znalostí. Plány π_1 a π_2 těchto prvků jsou následující.

$$\pi_1 = "\big(\big(!\ (ag2\ (o2\ jan\ 1962\ m))\big)\big)", \pi_2 = "\big(\big(+\ (b)\big)\ \big(?\ (\sim)\big)\ (\sigma\ \tau)\big)"$$

Počáteční konfigurace obou systémů budou

$$\big("ag1", \phi, \phi, \{\ \}, (\{\ \}, \{\ \}), "\big(\big(!\ (ag2\ (o2\ jan\ 1962\ m))\big)\big)", "(a)"\big)$$
$$\big("ag2", \phi, \phi, \{\ \}, (\{\ \}, \{\ \}), "\big(\big(+\ (b)\big)\ \big(?\ (\sim)\big)\ (\sigma\ \tau)\big)", "(a)"\big)$$

Výpočetní kontexty budou obsahovat pouze tyto konfigurace a dojde k následující sekvenci výpočtu.

$$\left\{\begin{array}{c}\big(\big("ag1", \phi, \phi, \{\ \}, (\{\ \}, \{\ \}), "\big(\big(!\ (ag2\ (o2\ jan\ 1962\ m))\big)\big)", "(a)"\big)\big), \\ \big(\big("ag2", \phi, \phi, \{\ \}, (\{\ \}, \{\ \}), "\big(\big(+\ (b)\big), \big(?\ (\sim)\big), (\sigma\ \tau)\big)", "(a)"\big)\big)\end{array}\right\}$$

$\triangleright_{DS}$

$$\left\{\begin{array}{c}\big(("ag1", \phi, \phi, \{\ \}, (\{\ \}, \{\ \}), "(\Omega_T)", "(a)")\big), \\ \big(\big("ag2", "\big(\big(ag1\ (o2\ jan\ 1962\ m)\big)\big)", \phi, \{"(b)"\}, (\{\ \}, \{\ \}), "\big(\big(?\ (\sim)\big)\ (\sigma\ \tau)\big)", "(a)"\big)\big)\end{array}\right\}$$

$\triangleright_{DS}$

$$\left\{\begin{array}{c}\big(("ag1", \phi, \phi, \{\ \}, (\{\ \}, \{\ \}), (\Omega_T), "(a)")\big), \\ \big(\big("ag2", \phi, \phi, \{"(b)"\}, (\{\ \}, \{\ \}), "\big((\sigma, \tau)\big)", "\big(ag1\ (o2\ jan\ 1962\ m)\big)"\big)\big)\end{array}\right\}$$

Výpočetní kroky byl podle přechodové funkce proveden ve dvou fázích. Podle relace $\vdash_{A,INP}$ došlo k přenesení dat ze vstupu prvku do jeho báze s uspřádáním a relace $\vdash_{A,ACT}$ provedla akci výběru zprávy. Přijatá zpráva je nyní aktivním stavem výpočtu adresáta zprávy, tedy prvku $"ag2"$.

V posledním výpočetním kroku dojde k provedení akce testování báze, která bude mít po substituci tvar "$\left(\sigma\ \ (ag1\ (o2\ jan\ 1962\ m))\right)$", a která neuspěje.

$\rhd_{DS}$

$$\left\{\begin{array}{c}\left(("ag1", \phi, \phi, \{\ \}, (\{\ \}, \{\ \}), "(\Omega_T)", "(a)")\right), \\ \left(\left(("ag2", \phi, \phi, \{(b)\}, (\{\ \}, \{\ \}), "(\Omega_F)", "\left(ag1\ (o2\ jan\ 1962\ m)\right)")\right)\right)\end{array}\right\}$$

3.10.4 Homogenní asynchronní distribuovaný ALLL systém s ALLL prvky

Podobně jako distribuovaný synchronní ALLL systém bychom mohli definovat i distribuovaný asynchronní ALLL systém. V tomto textu ale pouze zmíníme, v čem by se taková definice lišila od definice předchozího distribuovan0ho systémů a nebudeme ji celou formálně uvádět.

Ona definice asynchronního systému by se lišila pouze v definici přechodové relace tohoto systému. Množina prvků, které v daném okamžiku provádí výpočet, je nějakou podmnožinou množiny všech prvků v systému. Kvůli tomu by pravidla pro relaci předávání zpráv a relaci provedení kroku prvky měla být změněny tak, aby jednotlivé stavy systému na levé straně relace sice obsahovaly kontexty výpočtu všech prvků v systému, ale aby se výpočetní krok prováděl jen u některých z nich. To znamená, že se změní jen některé kontexty výpočtu ve stavech na pravé straně relace.

Co se týče relace předávání zpráv, je třeba správně modelovat situaci, kdy do vstupní báze přechází zprávy v různých okamžicích, a to aniž by byly pokaždé ihned zpracovány. Relace předávání zpráv by pak měla být definovaná tak, že první stav systému v relaci by mohl obsahovat i takové výpočetní kontexty, které mají jako svoji první konfiguraci takovou konfiguraci, která obsahuje nějaká data na vstupech. V definici 103 se předpokládalo, že v okamžiku předávání dat žádná data na vstupech prvků nejsou.

Přechodová relace asynchronního systému je i zde kompozicí předchozích dvou relací.

Distribuované asynchronní ALLL systémy by mohly být modely některých reálných distribuovaných systémů s ALLL prvky, pokud by se v těchto modelech nevyskytovaly jiné než ALLL prvky. Například pokud bychom modelovali bezdrátovou senzorovou síť a studovali směrování zpráv v ní, nemuseli bychom modelovat okolí senzorových uzlů, ale jenom tyto uzly jako právě například ALLL agenty. Obecně distribuované systémy jsou synchronizovány předáváním zpráv a reálně neprovádí výpočet v synchronizovaných krocích, takže distribuované ALLL asynchronní systémy by odpovídaly reálným distribuovaným systémům, které pracují tak, že každý prvek systému provádí činnost jako otevřený podsystém. Pokud ovšem by jen část prvků systému pracovala jako otevřený ALLL systém, což odpovídá i zamýšlenému nasazení realizací ALLL prvků, potom by odpovídající model takového distribuovaného systému musel zohledňovat i to, že některé prvky nemají chování přesně podle definic chování otevřených ALLL prvků. Jak by takový model měl vypadat, uvedeme v poslední části této kapitoly.

3.10.5 Heterogenní distribuovaný ALLL systém

Pod pojmem heterogenní distribuovaný ALLL systém rozumíme distribuovaný systém, který může obsahovat i jiné než ALLL otevřené prvky, ale tyto jiné prvky musí respektovat formát dat a princip fungování ALLL systémů. Tyto distribuované systémy jsou také těmi, které byly realizovány jako

skutečné systémy a které budeme popisovat v kapitole 4. Na tomto místě pouze zmíníme obecné požadavky na prvky, které jsou součástí takového systému.

Jelikož heterogenní distribuovaný ALLL systém vychází z homogenních distribuovaných ALLL systémů, které obsahovaly pouze otevřené ALLL prvky, budou prvky v tomto systému mít z vnějšího pohledu chování odpovídající otevřeným ALLL prvkům. Z vnějšího pohledu to znamená především to, že pro fungování prvku v rámci ALLL systému je třeba patřičně realizovat jeho

- formát přenášených dat ve formě ALLL obecných prvků
- chování ve fázích zpracování ALLL dat na vstupech prvku, provedení nějakého výpočtu a delegaci ALLL dat na výstupy prvku
- Interpretaci výše uvedených kroků v rámci distribuovaného systému

Další kapitoly ukáží realizace takových systémů na konkrétních případech.

4 Realizace ALLL multiagentních systémů

Od teoretického přístupu k definici struktur a chování ALLL systémů a prvků přejdeme k pojednání o jejich realizaci v rámci multiagentních distribuovaných systémů. Chování reálných ALLL prvků je řízeno jazykem ALLL. Jazyk ALLL je formálním jazykem pro programování systémů založených na formálních systémech uvedených v předchozích kapitolách. V následujícím textu půjde hlavně o specifikaci tohoto jazyka, jehož sémantika odpovídá principům výpočtů ALLL systémů a jeho následné použití v reálných systémech. Jazyk jako takový je pod názvem t-Sapi používán už od roku 2004, kdy jeho zjednodušená verze byla použita v první verzi nástroje T-Mass, o kterém bude pojednávat kapitola 4.4. Nyní představíme jazyk ALLL tak, jak jej používáme v současnosti, a ukážeme zkušenosti s programováním v tomto jazyce.

Text v minulé kapitole směřoval od základních systémů k složitějším, až jsme se dobrali k distribuovaným ALLL systémům. Distribuované ALLL systémy mohou obsahovat jak otevřené ALLL prvky, tak i jiné prvky, pro které jsou definovány příslušné funkce chování a které komunikují zprávami ve formátu ALLL obecných prvků. V této kapitole se budeme zabývat právě takovýmito heterogenními systémy. Na začátku kapitoly si uvedeme obecnou strukturu systému jako multiagentního systému s agentními platformami a základní principy jeho fungování. To nám umožní přejít ve výkladu k definici ALLL jazyka, na němž stojí realizace ALLL agentních prvků. V ALLL jazyce budeme zapisovat kompletní stav ALLL agenta zápisem plánů, dat a zpráv, jak jsme je představili v minulé kapitole. Jazyk se ale bude v některých aspektech lišit od dosavadního pojetí zápisů těchto struktur. Důvody k těmto změnám uvedeme na patřičných místech, a dále formálně uvedeme syntaxi jazyka v rozšířené Backus-Naurově formě. Jeho sémantiku si dovolíme uvést tentokrát méně formálně a to buďto s odkazem na poznatky z minulé kapitoly, a u těch akcí, které budou odlišné oproti akcím ze systémů z kapitoly 4, vysvětlíme sémantiku podrobněji.

Druhá polovina této kapitoly bude věnována dvěma systémům, ve kterých jsme realizovali ALLL agenty v rámci distribuovaných heterogenních ALLL systémů. Již jsme zmínili dříve, že těmito realizacemi jsou simulační nástroj T-Mass a systém bezdrátové senzorové sítě s ALLL agenty, který jsme nazvali WSAgeNt. U obou těchto systémů uvedeme příklady konkrétních aplikací, ze kterých budou patrny principy jejich fungování.

4.1.1 Obecná struktura a principy ALLL multiagentního systému

V současné době se obvykle předpokládá taková struktura multiagentního systému, ve které agenti existují na agentních platformách, které řídí životní cyklus těchto agentů, zpřístupňují jím okolní prostředí a poskytují jim své služby. Standardizace tohoto principu lze nalézt v dokumentech FIPA [11], ve kterých je uvedeno, jak by taková platforma měla vypadat a jaké služby by měla poskytovat. Tyto platformy mimo jiné poskytují adresářové služby, umožňují komunikaci mezi agenty v rámci platformy a i mezi platformami navzájem a to vše pomocí jazyka ACL (Agent Communication Language) [11]. Agentní platformy pro ALLL agenty ale nebudou zcela respektovat FIPA standardy. Je to hlavně proto, že zaměření ALLL systémů je na modelování distribuovaných systémů a jejich realizaci v malých zařízeních, pro která je zbytečné a v některých ohledech, jako je například spotřeba energie v bezdrátových senzorových sítích, i neefektivní, implementovat všechny požadavky z FIPA specifikací.

Agent bude mít formu kódu, který je na platformě uložen a který popisuje aktuální stav agenta. V návaznosti na předchozí kapitolu můžeme platformu a agenta vnímat tak, že agent reprezentuje

aktuální kontext výpočtu ALLL prvku a platforma realizuje přechodovou funkci tohoto ALLL prvku. Navíc je platforma také prvkem distribuovaného ALLL systému a tak může docházet ke komunikaci mezi agentem a platformou v rámci tohoto systému. Následující odstavce představí, jak agentní kód vypadá, jak jej platforma interpretuje a jak probíhá komunikace mezi platformou a agentem.

4.1.2 ALLL agent jako prvek multiagentního systému

Tedy pod pojmem ALLL agent budeme rozumět výpočetní kontext ALLL prvku, který je otevřeným prvkem a v základních principech své struktury a interpretace plánů funguje jako otevřený ALLL prvek z kapitoly 3.9. V souladu s teoretickými kapitolami je agent jako kontext výpočtu posloupnost jeho konfigurací, ale kvůli zestručnění zápisu nebudeme v jednotlivých kontextech opakovat báze a aktuální stavy výpočtu. Bude stačit, když pro jednotlivé úrovně kontextu uvedeme pouze prováděný plán. Jeho reprezentaci a některé způsoby odlišné interpretace od interpretací představených v teoretické části uvedeme později. Nyní konkrétně představíme strukturu ALLL agenta.

ALLL agenta reprezentuje šestice $Ag =< id, PB, KB, IB, I, R >$, která téměř odpovídá výpočetnímu kontextu ALLL otevřeného prvku. V této šestici je '*id* ' řetězec identifikující agenta a každý identifikátor agenta je jednoznačný v celém systému. Struktury PB, BB a IB jsou báze, I je záměr a R je množina hodnot registrů. Báze jsme definovali jako systémy, jejichž stavem je množina prvků, která musí patřit do domény báze. V tomto agentním prvku by bázi bez uspořádání odpovídalo sjednocení báze pro práci s plány (PB, Plan Base) a báze představ (BB, Belief Base). Vyrovnávací paměť pro příjem zpráv, za kterou považujeme bázi bez uspořádání, zde zastupuje vstupní báze (IB, Input Base). Domény bází utvoříme jako množiny všech podmnožin obecných prvků nad běžnou abecedou, která bude zahrnovat čísla, znaky a některé další symboly podle syntaktické konstrukce ALLL agentního jazyka, kterou uvedeme v nadcházejících podkapitolách. Vstupní báze bude uchovávat dvojice se jménem odesilatele jako prvním prvkem a obsahem zprávy ve formě obecného prvku na druhé pozici. Registry R jsou množinou s n obecnými prvky, kde n udává počet registrů dostupných pro daného agenta.

4.1.3 ALLL agentní platforma

Účel platformy je takový, že je na ní interpretován ALLL agentní kód, zajišťuje agentům zpřístupnění okolního prostředí a také jim poskytuje služby. Pokud je agent platformou přijat, je jeho kód uložen na této platformě a je platformou interpretován v nějakých výpočetních krocích. To znamená, že platforma je schopna vypočíst na základě tohoto kódu nový kód agenta. Jelikož v rámci distribuovaného nadsystému je platforma stejně jako agent jedním z jeho prvků, může mezi platformou a agentem probíhat interakce, i v tomto případě formou předávání zpráv. Agent posílá platformě zprávy, které obsahují požadavek na provedení nějaké služby, a platforma posílá zpět výsledky po provedení požadované služby.

Vztah mezi hostitelskou agentní platformou a agentem je natolik úzký a daný, že pro komunikaci mezi nimi budeme využívat jiné jazykové i interpretační metody, než jaké byly uvedeny v teoretické části. Namísto pojmenování platformy jménem a využití tohoto jména jako adresy při odesláni zprávy agentem, umožníme agentům používat akci, která bude přímo volat nějakou službu platformy. Výsledek provedení služby se nebude odesílat do vyrovnávací paměti agenta, ale většinou se umístí přímo do jeho aktivního registru. Co myslíme pojmem aktivní registr, uvedeme dále v kapitole o ALLL agentním jazyce. Nyní si ještě rozdělíme služby platforem do několika skupin.

4.1.4 Služby ALLL agentní platformy

Služby agentní platformy rozdělujeme na implicitní a explicitní, ty druhé pak dále na výpočetní a řídící. Implicitními službami myslíme takové, které jsou agentem využívány během jeho existence v systému bez toho, aby si je sám vyžádal. Mezi výpočetní služby řadíme takové, které mají formu komunikace mezi agentem a platformou takovou, že agent pošle platformě požadavek na výpočet ve formě obecného prvku a platforma po provedení výpočtu agentu odpoví zprávou ve formě obecného prvku. Tato zpráva, jak bylo uvedeno před chvílí, se objeví jako aktuální stav výpočtu agenta v jeho aktivním registru. Naproti tomu řídící služby mohou ovlivňovat interpretaci agenta, nebo znamenají nějakou aktivitu v rámci širšího systému. Platforma může pozdržet interpretaci agentního kódu, může jej přesunout na jinou platformu, vyhledávat dostupné agentní platformy, nebo i modifikovat či odstranit část nebo celý agentní kód.

4.1.4.1 Implicitní služby ALLL platformy

To, že implicitní služby nejsou agentem přímo vyžadovány, neznamená, že nejsou důležité pro jeho správné fungování v rámci platformy. Naopak se jedná o služby, bez kterých by agent nebyl schopen běžného provozu. Tyto služby proto musí být součástí všech ALLL platforem. Implicitní služby ALLL platformy jsou následující.

- Schopnost uchovávat a interpretovat agentní kód. To znamená, že platforma je realizována na nějakém výpočetním zařízení, nebo v rámci programového systému, a je schopna provádět výpočetní kroky agentů, kteří jsou na platformě umístěni.
- Schopnost přijímat a směrovat zprávy zasílané agentem.
- Schopnost přijímat a zpracovávat požadavky na provedení explicitních služeb.
- Schopnost zprostředkovávat agentům vjemy z prostředí a provádět jimi zvolené akce z a vůči prostředí, ve kterém se platforma s agenty nachází.

4.1.4.2 Synchronní a asynchronní služby platforem

Rozdělení na synchronní a asynchronní služby platforem provádíme podle toho, kdy jsou tyto služby prováděny. Synchronní služby jsou takové, které jsou platformou vykonány ihned po obdržení žádosti o službu. Pokud agent v jednom kroku požádá platformu o provedení synchronní služby, platforma tuto službu provede a ihned po provedení zprostředkuje agentovi výsledek. V bezprostředně následujícím interpretačním kroku pak má agent výsledek této služby k dispozici jako stav svého aktuálního registru. Žádost o službu nijak jinak neovlivní interpretační proces agenta. Oproti tomu asynchronní služby platformy mohou ovlivnit interpretační proces, nebo výsledky nemusí mít agent k dispozici již v následujícím kroku výpočtu. Asynchronní služby dále dělíme na blokující a démony. Blokující služby pozastavují interpretaci agenta, nebo ji oddalují. Démoni jsou takové služby, které jsou po vyžádání spuštěny na platformě jako samostatné procesy a běží paralelně s procesem interpretace agenta.

4.1.4.3 Výpočetní služby ALLL platformy

Mezi typické výpočetní služby řadíme aritmetické, logické a relační operace. Dále ve všech dosavadních realizacích zde jsou operace pro práci se seznamy, práci s řetězci a také služby provádějící složitější výpočty nad daty. Výpočetní služby platformy jsou vždy synchronní, tedy výpočet je platformou proveden ihned po obdržení žádosti a výsledek je ihned poskytnut agentovi zpět. Některé konkrétní výpočetní služby uvedeme u obou realizovaných systémů, které budeme popisovat v závěrečných částech této práce.

4.1.4.4 *Řídící služby ALLL platformy*

Za řídící služby považujeme všechny služby, jejichž efekt je širší, než je provedení výpočtu v rámci platformy. Řídící služby se neobejdou bez další komunikace s okolím, nebo bez omezení či modifikací prováděného agentního kódu. Řídící služby mohou být jak synchronní, tak i asynchronní blokující, nebo ve formě démona. Může zde jít o monitorování sítě, pozastavení interpretace agenta do výskytu definované události, nebo o ukončení agenta a jeho odstranění z platformy. U bezdrátových senzorových sítí patří mezi důležitou řídící službu transport agenta mezi platformami, což většinou znamená přenos agentních kódů mezi senzorovými uzly. I v tomto případě uvedeme u konkrétních realizací konkrétní poskytované řídící služby.

4.1.5 Komunikace v multiagentním ALLL systému

Zajišťování komunikace patří mezi implicitní služby platforem. Systém platforem by měl být schopen přenést zprávu od agenta k adresovanému agentovi, pokud se oba nachází na jedné platformě, a také toto činit i mezi agenty na různých platformách. Komunikaci v prostředí heterogenních multiagentních systémů je věnována značná pozornost v již několikrát zmíněných FIPA specifikacích [11], například jsou zde specifikovány komunikační jazyky, jazyky pro obsah zpráv, ontologie, komunikační protokoly atd.. Přenášení zpráv mezi agenty a mezi platformami je realizováno vysíláním zpráv k sousedním uzlům v případě bezdrátových senzorových sítí a v případě nástroje T-Mass tak, že prvky prostředí systému mohou hrát roli směrovačů v síti. Díky nim je možné zprávy přenášet nejen v rámci platforem, ale i mezi platformami napojenými společně na nějaký prvek prostředí, který vysílá zprávu od platformy všem ostatním připojeným platformám. Ty pak ověřují, jestli je adresovaný agent napojen na danou platformu nebo ne. Pokud agenta nenaleznou, je zpráva zahozena. Pokud ano, zprávu mu doručí.

Pokročilejší metody směrování v ALLL systémech jsou v současnosti vyvíjeny jak pro bezdrátové senzorové sítě, kde je realizován způsob šíření zpráv na základě poziciování uzlu, tzv. georoutingu [76], tak pro systém T-Mass ve verzi (M), kde realizujeme všesměrové spolehlivé vysílání pro obecné topologie sítí.

4.2 Jazyk ALLL agentního systému

Jazyk ALLL, který je používán v současnosti realizovaných ALLL systémech, se od jazyka, ve kterém jsme zapisovali konfiguraci či výpočetní kontexty u formálních ALLL systémů, mírně liší. Důvody, proč tomu tak je, uvádíme v následujících bodech.

- Jelikož čitelnost kódu, pokud by byl navržen podle teoretické specifikace, je relativně špatná, stejně tak jako programování v tomto typu jazyků je z počátku příliš obtížné, jsou některé struktury syntakticky odlišeny. Konkrétně se jedná o plány, které jsou uvedeny v lomených závorkách a které jsou uvozeny symbolem stříšky. Dotazovací seznam pro testování bází a volání platforem je uváděn v hranatých závorkách.
- Programátor má k dispozici více registrů, které mu umožní pohodlnější programování ALLL systémů oproti situaci, kdy byl v systému pouze jeden registr. Jeden z registrů je vždy nastaven jako aktivní a stav aktivního registru se může měnit prováděním akcí. V zápisech akcí může být zapsán symbol aktuálního registru, nebo symbol registru s indexem, který reprezentuje jeden z registrů v systému agenta. Každý z registrů je dále označen věkem pomocí apostrofů tak, že počet apostrofů před symbolem registru udává jeho věk.

- Oddělení bází plánů a představ zpřehlední vnitřní stav agenta. Operace vkládání a výběru do a z báze plánů či dat je také syntakticky rozlišena.
- Aritmetické a další výpočty budeme provádět v systémech s platformami. Ty, jak jsme uvedli výše, poskytují agentům služby, které umožňují aritmetické výpočty, provádění porovnání hodnot, práce se seznamy, řetězci a práci s proměnnými platformy. Jazyk bude umožňovat deklarovat akce, které přímo provádí volání služeb platformy

Abeceda jazyka bude sjednocením množiny velkých a malých písmen, symbolu registru τ, množiny přirozených čísel *N* a pomocných symbolů. Následující text představí gramatiku ALLL agentního jazyka a neformálně uvede sémantiku některých nových konstrukcí.

4.2.1 Gramatika programovacího jazyka ALLL systémů

Jazyk ALLL představíme gramatickými pravidly v EBNF. Prvky zde budeme dělit na obecné prvky a dotazovací prvky, které mohou navíc obsahovat symboly anonymního prvku. Na rozdíl od teoretické části jsou v současných realizacích a také v tomto jazyce dotazovací prvky pouze lineární. To vše je patrné z následujících pravidel.

```
<number>::= ["0"]|["1".. "9"]["0"..”9”]*

<string>::= { ("a"..”z”|”A”..”Z”|<number>|”_”|”-"|”+”|”.”) }

<atom>::= <number> | <string> | <reg>

<atom_ae>::= <atom> | "_”

<gntuple>::= "(" ( <atom> | <gntuple> )

                        {",” ( <atom> | <gntuple> ) } ")”

<antuple>::= "[" <atom_ae> { ",” <atom_ae> } "]”
```

Atom je základní prvek konstrukce agenta. Jedná se o kombinaci alfanumerických znaků a speciálních symbolů. Atom potenciálně obsahující anonymní prvek “_” je použit u dotazovacího prvku **antuple**. Obecné prvky anonymní prvek nezahrnují a jsou využity tam, kde operace neumožňují provádět dotazování, například při vkládání prvků do bází. V předchozí kapitole jsme už zmínili, že zápis dotazovacího prvku se také liší v použitých závorkách. Dotazovací prvky jsou uzavřeny v hranatých závorkách a obecné prvky v kulatých.

4.2.2 Systémy registrů u ALLL agentů

V základní verzi jazyka ALLL, který v počátcích nesl název t-sapi podle předpon, které se pro jednotlivé akce používaly, a také v teoriích představených v kapitole 3, byl pro označení stavu výpočtu použit jen jeden symbol registru, i když mohl mít různá ohodnocení věkem. Jednotlivé akce pak mohly tento registr používat ve svých zápisech a býval v nich nahrazován aktuálním stavem výpočtu před provedením této akce či při vyvolání podplánu. Výsledkem provádění akcí pak byla potenciální změna aktuálního stavu výpočtu.

Systém registrů je v reálných implementacích rozšířen o další symboly registrů. Obvyklým počtem registrů jsou tři obecné a dva speciální registry. Obecné registry slouží pro uchovávání mezivýsledků výpočtu a další registry mohou mít speciální funkci, jakou je třeba uchovávání hodnoty zanoření

právě vykonávaného plánu v hierarchii plánů či uchovávání jména agenta. Syntaktické pravidlo pro zápis registrů je následující.

<reg> ::= "&" **<number>** | " ' " <**reg**> | "&&" | "&L" | "&N"

Obecný registr má vždy formu znaku ampersand a symbolu. Systém s *n* registry umí pracovat s registry &1, &2 .. &*n* , kde *n* je přirozené číslo. Jak jsme již uvedli, symbol apostrofu, respektive jejich počet, udává věk registrů. Registr bez apostrofu, zapsán jen jako ampersand a symbol, je registr s věkem nula. Registr s jedním apostrofem má věk jedna, se dvěma apostrofy věk dva atd. Například &3 je registr číslo tři s věkem nula, "' &1 je registr číslo jedna s věkem tři.

Zbývá vysvětlit význam zápisů registrů *&&*, *&L* a *&N*. Dva ampersandy s případnými apostrofy je zápis aktuálního registru s příslušným věkem. Při zahájení výpočtu je aktivním registrem registr číslo jedna. Některé akce, které ukážeme za chvíli, mohou změnit aktivní registr na jiný. Ostatní dva zápisy jsou zápisy registrů, které mají hodnotu nastavenou systémem na úroveň zanožení aktuálně vykonávaného plánu v případě registru &L, a na jméno agenta v případě registru &N. Tyto registry nemohou být aktivní a nemohou být substituovány během operace maturace. Označení těchto registrů věkem je proto bezpředmětné a je systémy, který interpretují tento jazyk, ignorováno.

4.2.3 Akce ALLL agenta

Akce agenta jsou ve většině akcemi, se kterými jsme se setkali u teoretických systémů. Navíc zde najdeme takzvané interní akce, které popíšeme v další kapitole. Gramatika akcí je zde uvedena tak, aby byla přehledná. V praxi, například při programování systému T-Mass, bylo třeba gramatiku přepsat do gramatiky LL1, aby nástroj ANTLR [77] byl schopen správně vygenerovat kód překladače.

<action> ::=

"@" "^" "(" <**atom**> ")" | "@" <**plans**>

| "!" "(" <**atom**> "," <**gnlist**> ")"

| "?" <**anlist**>

| "+" <**gnlist**> | "+" <**dplan**>

| "-" <**anlist**> | "-" "^" "(" <**atom**> ")"

| "#" <**anlist**> | "#" <**gnlist**>

| <**internal_action**>

Pro pořádek uvedeme, o jaké akce se jedná. První dva řádky pravidla slouží k zápisu akce pro spouštění plánů. V prvním případě je vyvolán plán z báze plánu identifikovaný svým jménem, ve druhém případě zápis vyvolávaného plánu následuje bezprostředně po symbolu zavináče. Další řádky jsou pro generování akcí příjmu a zaslání zprávy. Akce jsou tvořeny předponou vykřičníku a otazníku v souladu s předchozími systémy a jsou následovány argumenty. Při odesílání zprávy je třeba zapsat jméno agenta, kterému je zpráva adresována a obsah zprávy ve formě obecného prvku. Pro přijímání zprávy je nutné uvést dotazovací prvek udávající vyhledávaného odesilatele zprávy ve vstupní bázi agenta. Operace přidávání prvku do báze dělíme podle toho, jestli do báze přidáváme znalost, nebo plán. Znalost je dána obecným prvkem, který následuje po předponě plus. V případě plánu za tímto znakem následuje dvojice udávající jméno plánu a samotný plán. Tyto dva prvky dohromady tvoří obecný prvek s tím, že první prvek musí být atom a nikoli seznam. Akce odebrání prvku z bází jsou

opět rozděleny na odebírání znalosti z báze představ, nebo na odebírání plánu z báze plánu. V prvním případě následuje po předponě, kterou je znak minus, dotazovací prvek. V případě odebrání plánu následuje symbol stříšky a prvek uvádějící jméno plánu. Kromě interních akcí, kterým budeme věnovat zvláštní odstavec, zůstává k představení akce dotazování do báze, která je uvedena předponou mřížky následovanou dotazovacím nebo obecným prvkem. Dotazování do báze může být také interpretováno jako akce volání služby platformy, pokud se první prvek v argumentu dotazu shoduje se jménem služby, která je platformou prováděna. Sémantika těchto akcí je v ALLL jazyce odlišná od uvedených teoretických systémů pouze v tom, že akce volání platformy může skončit neúspěchem, nebo výsledkem požadované služby, který je umístěn jako stav aktuálního registru.

4.2.4 Interní akce

Interní akce jsou v tomto systému navíc oproti teoretickým systémům z kapitoly 3. V ALLL jazyce bude interní akci reprezentovat zápis začínající znakem dolaru $.

<internal_action> ::= "$" <**number**>

Interní akcí, která je společná pro všechny ALLL agenty, kteří obsahují více než jeden registr, je akce přepínání aktivity registrů. Je také jedinou interní akci, kterou zde uvedeme. Jedná se o akci, která nastaví nějaký registr jako aktivní. Pokud za zápisem znaku dolaru následuje číslo, akce přepne aktivitu na registr s tímto číslem, a skončí úspěchem. Pokud číslo neodpovídá registru v systému, akce skončí neúspěchem.

4.2.5 ALLL plány a deklarace plánu

Syntakticky je ALLL plán posloupností akcí oddělených čárkami. Oproti teoretickým kapitolám, ve kterých byl plán jako každý jiný prvek uzavřen v kulatých závorkách, je zde kromě čárek mezi akcemi ještě navíc plán uzavřen do lomených závorek, což ho odlišuje od ostatních struktur jazyka.

<plan> ::= ("<" ">")

| ("<" <**action**> { "," <**action**> } ">")

Plán v ALLL systému je podle definice 32 lineární prvek obsahující akce a také toto pravidlo pro generování plánu až na zmíněné odlišnosti odpovídá této definici. Jednotlivé akce se pak provádí postupně, a jak jsou prováděny, je plán modifikován buď jejich odstraněním, rozšířením plánu o vyvolaný podplán, nebo přechodem na prázdný plán v případě neúspěchu provedení akce.

Pro práci s bází plánu, pro jejich ukládání, odebírání a nepřímého volání plánu z báze, je třeba plány pojmenovat. Struktuře, která bude obsahovat jméno plánu a samotný plán, říkáme deklarace plánu. Má formu danou tímto pravidlem.

<dplan> ::= "^" "(" <**atom**> "," <**plan**> ")"

To, že jde o deklaraci plánu, vyjádříme symbolem stříšky, který uvedeme jako první znak v deklaraci. Stříška jako znak uvádějící plán byla v pravidle definující akci použita také při deklaraci akce odstranění plánu z báze plánů, ve které jsme ale neuváděli celou deklaraci plánu, ale pouze jeho jméno.

4.2.5.1 ALLL Záměr

U teoretických ALLL systémů jsme uvedli, že přechodová relace je definovaná pro nějaké konfigurace ALLL prvků. Relace výpočetního kroku je pak definována pro výpočetní kontexty, pokud používáme

ALLL prvky s hierarchickými plány, a od nich odvozené prvky. V případě výpočetního kontextu se jedná o posloupnost konfigurací, a jak jsme uvedli v kapitole 4.1.2., budeme v našem případě výpočetní kontext reprezentovat posloupností plánů a podplánů s tím, že registry a báze nebudeme zmiňovat pro každou konfiguraci v kontextu zvlášť. Posloupnosti plánů budeme říkat záměr a bude to hierarchie plánů umístěná v zásobníku.

Záměr je syntakticky vytvořen nad plány a jeho definice je následující:

intention ::= plan ";" | intention ";" plan

Záměrem může tedy být jeden plán samotný, nebo více plánů seřazených v zásobníku, kde jednotlivé položky, plány, jsou odděleny středníkem. Prázdný záměr odpovídá jednomu prázdnému plánu.

Sémantická odchylka od teoretické kapitoly je taková, že na rozdíl od teorie, kde při provedení neúspěšné akce byl podle definice pravidla (FAIL) stav výpočtu obnoven na stav, který zde byl před spuštěním neúspěšného podplánu, v případě reálných systémů nastavujeme při neúspěchu podplánu hodnotu aktivního registru na hodnotu 'Fail'.

4.2.5.2 Báze ALLL agenta

Mezi báze dat patří báze pro představy agenta, jeho plány a vstupní báze pro asynchronní komunikaci mezi agenty, kterou zde nazýváme zkráceně vstupní báze. Ve všech třech případech to budou množiny syntaktických konstrukcí jednoho typu.

<BB> ::= { **<gntuple>** }

<IB> ::= { "(" **<atom>** "," **<gntuple>** ")" }

<PB> ::= { **<plan>** }

V případě vstupní báze a báze představ se bude jednat o obecné n-tice. Rozdíl mezi oběma bázemi bude v tom, že u vstupní báze bude prvním prvkem n-tice povinně identifikátor odesilatele. V případě báze plánu se bude jednat o dvojici definující ALLL plány, která bude jako první prvek obsahovat název plánu a jako druhý vlastní plán.

4.2.5.3 ALLL Agent

Z doposud uvedených syntaktických konstrukcí jsme nyní schopni utvořit kód agenta. Co má být obsaženo v tomto kódu jsme stanovili v kapitole 4.1.2. Je to jméno agenta, jeho báze, registry a záměr. Syntaxe úplného ALLL agentního programu je následující

<ALLL> ::= **<atom>** ';' [**<ntuple>**]^{n} ';' **<IB>** ';' **<BB>** ';' **<PB>** ';' **<intention>**

Jedinou částí, která zůstává po celou dobu existence agenta v systému neměnná, je jeho jméno zapsané jako atom, obvykle ve formě řetězce. Následují stavy *n* registrů, které jsou ve formě *n* obecných prvků, dále báze jako spojení řetězců pro jednotlivé prvky báze a nakonec záměr jako posloupnosti aktuálně prováděných plánu.

Provádění agenta probíhá na základě jeho aktuálního záměru. Interpret provádí přechod z jednoho stavu agenta na druhý přepisem agentního kódu. Podle první akce plánu na vrcholu zásobníku záměru agenta se určí akce k provedení a tou může dojít ke změně v kterékoli části agentního kódu

kromě jeho jména. Přijímání zpráv mění stav vstupní báze, operace vkládání a odebírání dat a plánů báze představ a plánů, většina akcí mění i stav aktuálního registru a každá z akcí mění záměr agenta.

4.3 Některé programovací techniky v ALLL jazyce

Jazyk definovaný v předchozí kapitole nyní prostudujeme z hlediska možnosti vytváření distribuovaných aplikací a programování jednotlivých uzlů v nich. V kapitole 3.8.7 bylo ukázáno, že ALLL systém s hierarchickými plány je schopen výpočtů na úrovni univerzálních výpočetních systémů. V minulé kapitole jsme specifikovali jazyk, kterým můžeme uzly programovat a nyní si ukážeme několik programátorských technik, které se ukázaly jako užitečné pro programování ALLL systémů.

4.3.1 Předávání hodnot do podplánů

Nejprve ukážeme, jak je možné předávat více než jednu hodnotu do spouštěného podplánu. Předpokládejme, že v bázi dat jsou vyhrazeny trojice (jmeno_planu, jmeno_argumentu, hodnota) pro nějaké jméno plánu uvedené jako první prvek v trojici. Pak můžeme vytvořit plán podle následujícího zápisu.

```
<#(jmeno_planu,jmeno_argimentu1, _),#(lst,( tth,&&)),
        @(#(jmeno_planu,jmeno_argimentu2, _), #(lst,( tth,&&)),
                @(#(jmeno_planu,jmeno_argimentu3, _), #(lst,( tth,&&)),
                …
                        @(telo_planu)  // se zápisy registrů '&&,''&&,'''&& …
                …
)))>
```

Tento plán kaskádově spouští podplány s tím, že při každém spuštění je obsahem aktivního registru hodnota, která byla získána z báze jako argument určitého jména. To je dosaženo provedením služby platformy pro práci se seznamem a konkrétně operací "tth", což odpovídá operacím "tail","tail" a "head" aplikovaným na druhý argument služby. Tímto tato služba do aktivního registru umístí třetí prvek z původní trojice, což je hodnota argumentu. V okamžiku spuštění podplánu dochází k maturaci registrů na úrovni těla podplánů a pokaždé dojde k nahrazení registrů s věkem 0 za aktuální hodnotu registru. Tak se postupně registry '&&,''&&,'''&& … nahrazují hodnotami, které byly aktuální v okamžiku prvního, druhého, třetího, a případně dalších spuštění podplánu.

4.3.2 Obnovení registrů z podplánů

Rozdíl v sémantice interpretace podplánu a následných stavu registrů po skončení podplánu, kterou jsme uvedli v kapitole 4.2.5.1., vedou k tomu, že v případě chybného ukončení podplánu jsou původní hodnoty aktuálního výpočtu ztraceny. I v případě úspěšného dokončení podplánu můžeme jako programátoři chtít, aby se po návratu interpretace na vyšší úroveň plánu obnovila původní data v registrech. Následující zápis zajistí, že hodnoty v registrech budou obnoveny na stav před zahájením provádění podplánu a to v obou případech, kdy plán označený @(*telo_planu*) neuspěje, nebo neuspěje. V následujícím zápise budeme předpokládat, že systém používá tři registry.

```
<@(telo_planu) ,
        $1,(std,(s,&1)),$2,(std,(s,&2)),$3,(std,(s,&3))>
```

V okamžiku spuštění plánu jsou symboly registrů &1, &2 a &3 nahrazeny aktuálními hodnotami těchto registrů. Nadále jsou tyto hodnoty v plánu zachovány a to do okamžiku, kdy skončí výpočet

plánu “telo_planu”. Ten byl spuštěn jako podplán a tak ať uspěje či nikoli, proběhne provedení zbytku plánu ve formě

<$1,(std,(s,hodnota_1)),$2,(std,(s, hodnota_2)),$3,(std,(s, hodnota_3))>

V tomto zápisu plánu zápisy „hodnota_1“ atd. zastupují hodnoty příslušných registrů v době spuštění podplánu. Služba platformy označená ‘std’ je standardní systémová služba a operace ‘s’ znamená nastavení hodnoty aktivního registru na hodnotu, která je druhým prvkem argumentu služby. V plánu se pak postupně nastavují jednotlivé registry na aktivní a do nich jsou ukládány původní hodnoty registrů.

4.3.3 Opakované provádění plánu

Opakované provádění části plánu budeme nejprve demonstrovat na jednoduchém příkladu s abstraktním tělem plánu, které bude opět reprezentovat řetězec ‘*telo_planu*’.

{<iter,< #(test), @(*telo_planu*), @iter>}

Plán, který probíhá v iteračních krocích, musí být uložen v bázi plánů. Ve výše uvedeném příkladu iterace skončí, pokud skončí neúspěchem akce #(test), což je testování báze agenta, nebo volání služby platformy téhož jména, pokud jí má platforma v adresáři služeb. Vyhodnocování podmínky opakování iterace se většinou provádí voláním služby platformy, která buďto skončí úspěchem, nebo neúspěchem.

Další příklad ukáže plán, který provádí cyklus od zadané hodnoty do hodnoty nula. Budeme předpokládat, že v době spuštění plánu bude báze obsahovat dvojici ve tvaru ‘(iter, n)’, kde ‘n’ bude kladné číslo.

{<iter,< #[iter,_], #(lst,(th,'&&)), #(ari,(m,’&&,(1))), -[iter,_], +(iter,‘&&),
#(rel,(g,'&&,(0))), #(@(*telo_planu*), ,@iter>}

V tomto případě agent využívá hned tří služeb platforem. Služby ‘ari’, ‘rel’ a’lst’ jsou služby pro výpočet aritmetických operací, pro vyhodnocování relací a pro práci se seznamy. Po úspěšném provedení testování báze na dvojici, ve které je prvním prvkem atom ‘iter’, je vyvolána služba, která z obsahu registru vybere druhý prvek. Pokud předpokládáme, že v bázi byla odpovídající dvojice, například ‘(iter,5)‘, pak výsledkem operace testování je prvek ‘(iter,5) ‘ a výsledkem vyvolané služby je prvek ‘(5)‘.

Dále je vyvolána služba pro aritmetické výpočty a v argumentu je na prvním místě uvedena operace, v tomto případě operace odečítání, a za ní jsou uvedeny operandy. Symbol registru se nahradil hodnotou ’(5)’ a druhým operandem je prvek ‘(1)‘. Tato služba skončí úspěchem a výsledkem v tomto případě je nová hodnota aktivního registru ‘(4)‘. Nová dvojice je vložena do báze na místo původní a je požadována další služba platformy, která má porovnat, zdali druhý prvek argumentu služby je větší než prvek ‘(0)‘. Pokud by tomu tak nebylo, plán a celá iterace skončí. Pokud služba projde úspěšně, provede se tělo iterace a celý plán ‘iter’ je vyvolán znovu.

Některé další příklady použití jazyka ALLL pro programování systémů si ukážeme v souvislostech s reálnými ALLL systémy. Těmi se nyní budou zabývat závěrečné části této práce.

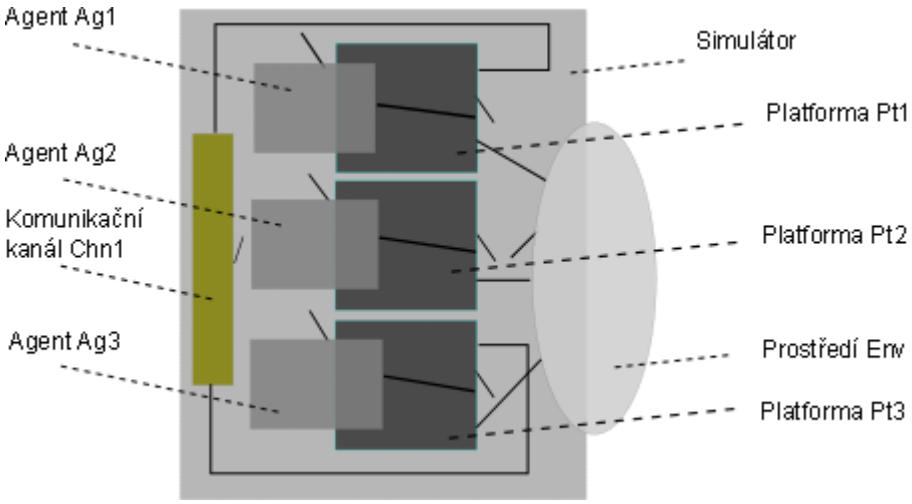
4.4 Agentně orientované modelování diskrétních distribuovaných systémů systémem T-Mass

Prvním reálným systémem, ve kterém jsou ALLL prvky použity, je už několikrát zmíněný simulační nástroj distribuovaných inteligentních systémů T-Mass (A Tool For Multiagent Systém Simulation). V disertaci autora [1] byl poprvé popsán jako nástroj pro provádění diskrétních simulačních systémů s agentními rysy. V roce 2005 byla k dispozici první verze tohoto systému, ve kterém ale ještě nebyly použity agentní platformy a prvky okolí. Systém zde tvořila populace agentů, která byla navzájem provázána komunikačními kanály. Agenti měli k dispozici některé výpočetní služby, které byly prováděny přímo interpretem agentního kódu v rámci simulačního systému. Tento systém byl použitelný pouze k ověření výpočetních schopností ALLL agentů na některých jednoduchých modelech. Od té doby dochází k rozvoji tohoto systému směrem k možnostem simulace reálných distribuovaných systémů s inteligentními prvky. Koncem roku 2010 byla publikována další verze systému nazvaná T-Mass(X), která je nyní k dispozici k volnému použití na stránkách FIT VUT v Brně, a kterou si v následujícím texu stručně představíme.

4.4.1 Struktura modelu v systému T-Mass

V současné verzi je simulační nástroj T-Mass(X) zaměřen na vytváření simulačních modelů s architekturami zahrnujícími prvky jako jsou agentní platformy, ALLL agenti a prostředí. Model je vytvářen pomocí metajazyka, ve kterém lze specifikovat jednotlivé prvky v systému, jejich funkčnost a jejich vzájemné propojení. Každý prvek má vstupní a výstupní brány určené k propojení s některým ze tří typů prvků. Skrz brány je také každý prvek propojený se simulátorem, což je vždy jeden z prvků systému, a který má za úkol řídit simulaci. Tyto brány bývají párovány do umístění (slotů) prvků, to znamená, že vždy jedna vstupní a jedna výstupní brána tvoří slot k propojení s nějakým typem prvku. V současné verzi je dáno, že agentní prvek může být kromě simulátoru propojen pouze s agentní platformou, prvek prostředí s agentní platformou nebo s jiným prvkem prostředí a konečně agentní platforma může mít sloty určené pro propojení s agenty, nebo s prvky prostředí.

Obrázek 9 ilustruje obecnou strukturu modelu v T-Mass(X). Obecně by se měl model skládat alespoň z jedné agentní platformy a jednoho agenta. V tomto případě jsou v modelu tři platformy P1, P2 a P3 které spravují vždy jednoho agenta Ag1, Ag2 a Ag3. Všechny platformy jsou propojeny s prvkem prostředí Env a první a třetí platforma jsou propojeny s jiným prvkem prostředí, který je nazván Chn1, a slouží jako komunikační kanál mezi těmito platformami. Všechny prvky v modelu jsou dále propojeny s prvkem simulátoru.



Obrázek 9: Koncepce struktury modelu ALLL systému v nástroji T-Mass(X)

4.4.2 Princip simulace v systému T-Mass

Simulace probíhá jako sekvence zpracování událostí v systému podle jejich zařazení do kalendáře událostí. Události se vztahují jak k simulátoru, tak k prvkům prostředí, platforem a agentů. Na počátku simulátor vykoná jeden krok a v tom nastaví události pro agentní prvky. Agenti pak provádí svojí činnost a obvykle vygenerují událost pro platformu, na které se nachází. Touto událostí dojde k oživení platforem, které události zpracují a obvykle vygenerují událost pro agenty a tak dále.

Interpretaci agentů v takovémto modelu provádí simulační systém a platforma zde funguje pouze jako prvek systému, který reaguje na zprávy od agentů a události od prvků prostředí, se kterými je spojena. Jde zde také o to, že T-Mass je navržen spíše s ohledem na realizaci simulačních nástrojů s diskrétními událostmi řízených kalendářem, než na striktně multiagentních principech.

Provedení simulačního kroku může mít za následek jednu nebo více z následujících změn.

- Změnu vnitřního stavu prvku. Prvek může provést výpočet, kterým změní struktury, které si uchovává jako svůj vnitřní stav.
- Změny hodnot na vstupech a výstupech prvků. Součástí procesu komunikace je vygenerování zprávy na výstupních bránách prvku a doručení těchto zpráv na vstupní brány jiných prvků. Provedením prvku můžou jednak vzniknout nové zprávy v systému, nebo můžou být zprávy zpracované prvkem, kterému byly doručeny na jeho vstupní bránu.
- Změnu struktury systému. Ve verzi T-Mass(X) migrace agentů a ani jiné způsoby změny struktury systému nejsou podporovány. Ve verzi vyvíjené od začátku roku 2012, která nese název T-Mass(M), je pohyb agenta mezi platformami podporován. Agent zde může zavolat službu platformy, jejímž výsledkem má být to, že umístění agenta v systému se změní a jeho fungování bude dále probíhat ve spojení s jinou platformou.
- Změnu v kalendáři událostí. Prvek může generovat nové události a ty vkládat do kalendáře událostí. Tím ovlivňuje běh simulaci v budoucích krocích.

První dva body jsou v souladu s teoretickými předpoklady ohledně ALLL prvků. Vnitřní stavy těchto prvků zahrnující i hodnoty na vstupních a výstupních branách se mohly měnit během interpretace aplikace plánů na tyto prvky. Druhé dva body souvisí s požadavky na mobilitu agentů v systému a s principy simulačního systému založenými na událostech a jejich kalendářích.

4.4.3 Události a kalendář událostí

Tak jak události vznikají, jsou řazeni do kalendáře událostí. U každé události je uvedeno, ve kterém modelovém čase a kým byla vygenerována a pro události týkající se simulace činnosti prvku také to, který prvek v rámci této události má konat a kdy má provést nějakou činnost. Interpret simulace během své činnosti vybírá událost k vykonání v nejnižším modelovém čase a tu vykonává. V události lze také specifikovat, jakou část nebo části výpočetního kroku prvku daná událost vyvolává. V T-Mass je totiž výpočetní krok každého prvku v modelu prováděn ve čtyřech fázích. Jde o fáze načtení hodnot ze vstupních bran, zpracování těchto vstupních hodnot prvkem, provedení výpočtu prvkem a nastavení hodnot na jeho výstupech. U události může být nastavena jakákoli kombinace těchto fází, i když obvykle dochází k provedení všech čtyř fází současně. Kromě událostí provádění prvků mohou být do kalendáře umístěna událost změny struktury modelu přesunutím agenta, ty ale pouze ve verzi T-Mass(M), a událost ukončení činnosti agenta.

Běžný simulační běh systému probíhá tak, že agent po provedení výpočtu vygeneruje událost provedení výpočtu platformou a naopak. Platforma také může generovat událost pro provedení kroku prvkem prostředí a prvek prostředí následně zavádí do kalendáře událost pro výpočetní krok platformy, aby ta mohla zareagovat na případné podněty z prostředí. Tak se obvykle střídají kroky mezi těmito dvěma podmnožinami prvků, množinou platforem a množinou ostatních prvků universa. Mohou ale nastat i jiné scénáře generování událostí, z nichž dva si pro ilustraci uvedeme.

Agent může platformu požádat, aby jeho další interpretační krok provedla po náhodně stanoveném okamžiku. Potom dojde k pozdržení interpretace agenta tím, že platforma po zpracování informace od agenta vygeneruje další událost provedení agenta s patřičným zpožděním, to znamená v nějakém modelovém čase po uplynutí náhodně vygenerovaného okamžiku. Ostatní prvky systému mohou provádět své výpočty v době, než nastane tento modelový čas, ale agent není vykonáván a na případnou snahu o interakci ze strany ostatních prvků nereaguje. Uchovává si ale zaslané informace ve své vstupní bázi, aniž by kromě zpracování vstupů prováděl další fáze výpočetního kroku.

Druhý příklad ukáže situaci obdobnou, ale z pohledu prostředí. Jedna platforma zasílá skrz prostředí informaci jiné platformě. V okamžiku doručení zprávy přes prostředí je také generována událost pro zpracování vstupu adresovanou platformou. Nemusí tedy dojít k provedení celého výpočetního kroku platformy, ale jen k přijetí zprávy z prostředí.

4.4.4 Jazyk ALLL agenta pro T-Mass

Jazyk používaný k programování agentů v modelech systému T-Mass(X) vychází z jazyka, který jsme představili v kapitolách 4.2 a 4.3. Jednou ze změn oproti tomuto jazyku je rozšíření množiny interních akcí agenta o další dvě, které uzavírají a otevírají agentovy vstupy. Akce *$FI* (Freeze Input) dovolí agentovi v nějaký okamžik svého působení v systému zablokovat doručování zpráv do své vstupní báze. Zprávy zůstávají na platformě a ta je doručí až po vykonání akce a *$OI* (Open Input) agentem, nebo po vyžádání služby platformy, které otevírají agentovy vstupy automaticky. Takovou službou je například služba čekání na příchod zprávy. Tato služba je také nejčastěji využívána dohromady s akcí zmrazení vstupu, protože pokud dochází k postupnému zpracovávání zpráv na vstupu, pak po doručení zprávy bývá vstup zmražen, zpráva zpracována, a po zpracování zprávy se opětovným zavoláním služby čekání na další zprávu agentovy vstupy otevřou. Během zpracovávání zprávy tedy nemůže dojít k příchodu nové zprávy do vstupní báze agenta a ten tak nemusí bázi testovat, ale

rovnou zadat službu čekání na zprávu. Příslušné syntaktické pravidlo pro obě nové interní akce je toto.

<internal_action> ::= "$" "FI" | "$" "OI" | "$" <**number**>

Druhá odlišnost ALLL jazyka v T-Mass(X) bude v použití dotazovacího seznamu. V současných verzích probíhá dotazování u ALLL jen s použitím lineárních seznamů, což se v praxi ukázalo jako dostačující, i když do budoucna počítáme s realizací dotazovacích seznamů ve formě obecných seznamů, jak jsme je používali v teoretických kapitolách.

<antuple>::= "[" <**atom_ae>** { "," <**atom_ae>** } "]"

Mezi drobné syntaktické odlišnosti patří i to, že u deklarací plánů jsou jména plánu a jejich těla odděleny středníkem a že deklarace je uzavřena ve složených závorkách namísto kulatých závorek.

Celý ALLL agent je zapsán ve formě deklarace záměru, vstupní báze a báze představ. Nedeklarujeme počáteční stav registrů, který bude implicitně nastaven na vyhrazený prvek (null), a nedeklarujeme ani počáteční stav vstupní báze. Pro přehlednost deklarace záměru a deklarace bází uvádíme řetězci >intention: nebo >database : .

Podstatnou změnou sémantiky oproti sémantice operací u formálních ALLL systémů přináší operace odebírání prvku z báze představ. V případě systému T-Mass(X) operace odebírání prvku, pokud nenajde prvek, který by šlo podle argumentu odstranit, skončí neúspěchem. Při praktickém programování se ukázalo užitečné to, aby akce odebírání fungovala zároveň i jako akce testování. Smysl této kombinace akcí odebírání a testování bude vidět na příkladech, které uvedeme za chvíli.

4.4.5 Módy ALLL agenta a provádění jeho výpočetního kroku

Interpretace agenta probíhá v souladu s návrhem otevřeného agentního prvku, to znamená zpracováním vstupu, provedením výpočtu, a nastavením hodnot na jeho výstupních branách. Navíc v simulačním modelu může být agent nastaven do nějakého módu a na základě tohoto módu může být ovlivněn způsob jeho interpretace, jeho grafické zobrazení a také může být prováděn zápis těchto módů do protokolu simulačního běhu. Mezi podporované módy patří mód, kdy je agent připraven k interpretaci, běží, čeká jako odložený proces, je probuzen platformou po čekání na událost, je ukončen a také mód, kdy má blokované vstupy a nezpracovává vstupní informace.

4.4.6 Jazyk pro specifikaci struktury modelu T-Mass Modelling Language

T-Mass Modelling Language (TML) je jazyk pro specifikaci struktury modelu v T-Mass. Struktura modelu musí odpovídat uvedeným vztahům mezi typy prvků v multiagentním systému, jak jsme je uvedli v kapitole 4.4.1. Agent je v systému vázán na agentní platformu a agentní platformy interagují s agenty a s prvky okolního systému. V jazyce TML může být proto deklarováno následující.

- Programy agentních prvků a umístění souborů s agentními programy
- Typy platforem v systému a umístění souborů s popisem struktur těchto platforem
- Počet a pojmenování jednotlivých agentů v systému
- Počet a pojmenování jednotlivých platforem v systému, služby platforem
- Prvky prostředí v systému a třídy popisující jejich chování

- Propojení mezi jednotlivými prvky v systému

4.4.7 Deklarace platforem v systému T-Mass a jejich služby

Deklarace platforem zahrnuje pojmenování platforem, počet slotů pro umístění agentů a počet slotů pro napojení prvků prostředí. Každý slot zahrnuje jednu vstupní a jednu výstupní bránu pro spojení s prvkem daného typu. Dále jsou u každé platformy uvedeny třídy, které reprezentují služby platforem. Tyto třídy jsou odvozeny od třídy služby platformy, které poskytuje systém T-Mass, a jsou pojmenovány nějakým atomem ve formě řetězce.

Uživatel si může sám deklarovat nové třídy, nebo využít tříd, které jsou poskytovány v distribucích T-Mass. Některé služby jsou obvyklé pro všechny distribuce, jako je například vyhodnocováni aritmetických výrazů, nebo práce s řetězci či se seznamy. Obvyklou službou platforem v tomto systému je také to, že spravují proměnné platformy a zpřístupňují je agentům. Každá služba je tedy popsána třídou a v deklaraci platformy se uvádí umístění třídy služby a jméno, pod kterým bude služba na platformě registrována. Na základě interakce s agentem platforma spouští tuto službu, pokud je jméno služby prvním prvkem obsahu požadavku od agenta. Druhý prvek této zprávy je argumentem služby. Obvykle v tomto argumentu je jako první prvek uvedena specifikace operace, tedy co konkrétně je od služby požadováno. Například zápis

```
#(lst,(hth,((a,b,c),(d,e)))
```

je požadavek na službu "lst", které je zde deklarována jako služba platformy a platforma zná třídu, která provádí požadovanou službu. Ten podle prvního prvku argumentu 'hth' provede postupně operace 'head' 'tail' a 'head' nad uvedeným seznamem a výsledkem je pak prvek'(b)'. Ostatně takovou službu jsme již použili v kapitole o jazyce ALLL.

Nyní přejdeme k ukázkám realizací modelů systémů v T-Mass(X).

4.4.8 Příklad jednoduchého agenta v T-Mass

První příklad ukáže způsob deklarace záměru a využití služeb platforem. Agentní kód obsahuje pouze deklaraci záměru a záměr obsahuje pouze dvě akce. Obě akce jsou voláním služby platformy. Předpokládáme, že v deklaracích platformy byly služby pro práci s řetězci registrovány pod jménem 'std' a služby terminálu pod jménem 'tml'. Každá služba obdrží argument ve formě obecného prvku a ten dále zpracovává. Připomínáme, že v systému T-Mass(X) je u argumentů akcí volání služeb obvyklé, že první prvek argumentu specifikuje operaci, kterou má služba vykonat. V našem případě jde o operaci 'c' – compare, porovnání, u služby práce s řetězci a 'pl' – printline, u služby terminálu. Kód záměru agenta je napsán tak, že je porovnáno jméno agenta, které je implicitním obsahem registru &N, s řetězcem 'agent1', a pokud tato operace uspěje, je vytištěno na terminál, že agentovo jméno je právě 'agent1'.

```
>intention : <
        #(std,(c,agent1,&N)),          // porovnání registru se jménem agenta s 'agent1'
        #(tml,(pl,(jaJsem, agent1))),  // pokud se jmenuji 'agent1', pochlubím se tím
        >;
```

Agent bude v okamžiku před zahájením své interpretace reprezentován kódem

```
"<#(std,(c,agent1,&N)), #(tml,(pl,(jaJsem, agent1)))>,null,null,null"
```

Tento kód obsahuje kód záměru a hodnoty tří registrů, které jsou na začátku nastaveny na ‘null’. Po provedení první akce je na platformě spuštěna odpovídající služba, jejímž výsledkem může být buď hodnota aktuálního registru a tato hodnota se tedy nezmění, nebo platforma bude interpretovat tuto akci jako chybnou. V prvním případě bude kód agenta “<#(tml,(pl,(jaJsem, agent1)))>,null,null,null“ a je připravena druhá z akcí k provedení. Agent skončí buď již po selhání první akce, nebo po provedení druhé akce, a to v obou případech jako kód “<>,null,null,null“.

4.4.9 Příklad modelu v T-Mass, synchronní komunikace mezi agenty

Druhý příklad bude trochu složitější a ukážeme v něm zápis kódu agenta, který spouští podplány a provádí výpočet v iteracích. V systému bude fungovat několik agentů v rámci multiagentního systému a uvedený agent bude zjišťovat, jací agenti v systému jsou k dispozici. K tomu využije službu platformy, která zjistí sousední agenty v multiagentním systému a informace o nich zašle agentovi do jeho báze poté, co je najde. Jedná se tedy o asynchronní službu, která běží souběžně s další interpretací agenta. Plán na nejvyšší úrovni vypadá následovně.

```
>intention : <
        #(tml,(pat)),   // služba terminálu, vytisknutí aktuálního času
        #(ntw,(af)),    // služba sítě, nalezne v systému sousední agenty
        #(wms,(150)),// čekání na zprávu po dobu 150ti simulačních cyklů
        @^(iter),       // plán pro zpracování vstuptní báze
        #(tml,(pat))    // vytisknutí aktuálního času před ukončením agenta
        >;
```

První a poslední akce jsou akce vyvolání služby terminálu, které na terminál vytisknou aktuální čas. Lze tak určit dobu, jak dlouho proces vyhledávání agentů v systému trval. Dalšími akcemi jsou akce vyvolání síťové služby vyhledávání okolních platforem a dále agent žádá platformu o službu čekání na zprávu. Ta zastaví interpretaci agenta buďto do okamžiku než do jeho vstupní báze dorazí zpráva, nebo do uplynutí 150 simulačních cyklů. Časový limit je zde učiněn pro případ, že by platforma neobjevila žádné další agenty v systému a proto nezaslala agentovi žádnou zprávu do báze. Po ukončení čekání je vyvolán podplán, který provede uložení informací o nalezených platformách ze vstupní báze do báze představ.

```
>database : {iter;
        <
        ?(#connected),      // služba sítě zaslala výsledky pod jménem ‘connected’
        +('&&),             // pokud výběr uspěl, výsledek uložíme do báze
        $1,@^(iter)         // spustíme plán pro výběr další zprávy
        >};
```

4.4.10 Příklad modelu v T-Mass, producent a vyrovnávací paměť

Posledním z příkladů bude kód zahrnující agentní programy pro komplexnější systém, ve kterém budou působit dva agenti s rolemi producenta a konzumenta. První agent nazvaný ‘Delnik’ posílá prvky ve formě zprávy druhému agentovi se jménem ‘Skladnik’. Po obdržení zprávy skladník uloží tuto zprávu, ale ne do své báze představ, ale na platformu tím, že zavolá službu ukládání do báze platformy. Pokud platforma potvrdí úspěšné uložení dat, dělník posílá další prvek. Pokud služba platformy neuspěje a prvek není správně uložen, informuje o tom skladník dělníka a oba agenti svoji činnost ukončí.

První část kódu agenta je tvořena pro skladníka, který obdržené informace od dělníka ukládá na platformu. Po uložení vyžaduje zaslání další informace, pokud bylo v paměti ještě místo, nebo zasílá informaci, že už nemůže informace ukládat. Prvotní záměr bude inicializovat datovou strukturu lokální datové vyrovnávací paměti platformy a potom spustí hlavní plán z báze plánů.

```
>intention :  <#(buf,(nb,(mybuf,10))),@^(cycle)> ;
```

Po inicializaci nové vyrovnávací paměti na platformě s názvem ‘mubuf’ záměr pokračuje spuštěním plánu ‘cycle’.

```
>database :   {cycle;
               < @^(waitmessage),
```

Plán ‘waitmessage’ je zde spuštěn hned první akcí plánu ‘cycle’. Jeho činnost bude popsaná níže. Zde si uvedeme pouze to, že po ukončení plánu bude v registru &1 uložena obdržená zpráva od dělníka. Druhou akcí je přímé spuštění podplánu. Důvod této akce je prostý a často přítomný při programování ALLL systémů. Některé akce mohou v tomto podplánu neuspět, a přitom si nepřejeme, aby skončil neúspěchem plán na původní úrovni. Spuštěním části plánu jako podplánu tomu zabráníme a můžeme postupovat v činnosti tak, že do registru 3 uložíme modelový čas a ten službou platformy ‘lst’ a operací ‘a’ (append) připojíme do seznamu za přijatou zprávu agenta.

```
@<
    $3, #(std,(mt)),              // do registru 3 uloží modelový čas
    $1, #(lst,(a,''&1,(''&3))),   // sestaví dvojici přijaté zprávy a modelového času
```

Následující tři akce provedou to, že bude vyvolána služba platformy, která má uložit sestavený prvek do vyrovnávací paměti, kterou jsme na platformě iniciovali. V okamžiku vyvolání této služby je registr č. 3 nastaven na hodnotu ‘fail’, která předjímá neúspěch operace. Připomínáme, že služba ‘std’ a její operace ‘s’ (set) nastaví aktivní registr na hodnotu následující za kódem operace ‘s’. Pokud dojde k neúspěchu, pak se tento registr nezmění, jelikož jsme nastavili aktivitu na registr &1. V případě úspěchu jej ale nastavíme na hodnotu ‘next’ a nastavíme příznak, který bude značit úspěch.

```
  $3,#(std,(s,fail)),             // nastaví aktuální registr č.3 na hodnotu ‘fail’
  $1,#(buf,(p,(mybuf,(''&1,)))), // uloží seznam do bufferu, pokud je v něm místo
  $3,#(std,(s,next)),             // operace s bufferem prošla, registr bude ‘next’
  +((flag,&L))                    // přidání příznaku úspěšného provedení operace
>,
```

Obsah registru &3 odešleme dělníkovi. Ten si již přebere obě situace, ať dostane žádost o další prvek ‘next’, nebo informaci o neúspěchu ‘fail‘. V případě úspěchu se odstraní příznak, tím se i provede testování jeho přítomnosti v bázi, a celý plán je spuštěn znovu.

```
 !(Delnik,'&3),          // odeslání žádosti nebo informace o neúspěchu
 -[flag,&L],             // operace s bufferem uspěla či ne?
 @^(cycle)               // pokud test projde, přijmeme další zprávu
> };
```

Ještě je třeba popsat plán, který čeká na zprávu od dělníka. Kód nyní uvedeme pouze s komentáři, které dostatečně uvedou smysl jednotlivých akcí.

```
        {waitmessage;
            <
             +((flag,&L)),      // nastavení příznaku, který bude použit dále
             @<
               $1,?(Delnik),    // pokus o vybrání zprávy od dělníka z vyrovnávací paměti
               $2,-[flag,&L]    // pokud uspěje, odebereme příznak z vyrovnávací paměti
               >,
               -[flag,&L],      // odebrání příznaku a zároveň test jeho přítomnosti v bázi
// pokud plán uspěje i po této akci, je třeba čekat na zprávu, jelikož doposud nedorazila
               #(wms,(20)),   // čekání na obdržení zprávy, maximálně 20 cyklů simulace
               @^(waitmessage)  // opakování tohoto plánu
            >};
```

Zbývá představit plány druhého agenta, kterým je dělník, jenž má za úkol na požádání produkovat zprávy pro skladníka. Na začátku jeho záměr pouze spouští hlavní plán agenta.

```
>intention : <@^(core)>;
```

V bázích jsou opět uloženy dva plány. První zasílá skladníkovi zprávu s aktuálním modelovým časem a volá druhý plán, který čeká na odpověď od skladníka. Pokud obsahem odpovědi je žádost o další prvek ‘next’, což zjistí díky službě platformy pro práci s řetězci a pro operaci porovnání, provede dělník znovu celý plán.

```
>database : {core;
               <
               #(std,(mt)),                  // aktivní registr bude obsahovat modelový čas
               @^(sendandwait),
// po ukončení podplánu ‘sendanwait’ čekáme v aktivním registru obsah zprávy od skladníka
               #(std,(c,'&&,(next))),        // pokud se jedná o žádost o další prvek
               @^(core)                      // opakujeme hlavní plán
               >
      };
```

Plán ‘sendandwait’ je krátký. Pouze naplněný aktivní registr modelovým časem odešle skladníkovi. Poté vyvolá podplán čekání na odpověď od skladníka a skončí.

```
       {sendandwait;
        <
               !(Skladnik,(&&)),       // pošleme data skladníkovi k uložení
               @^(waitskladnik),       // volaný plán počká na odpověď od skladníka
               $1                      // odpověď je v registru $1, nastavíme jej na aktivní
         >
      };
```

Díky nastavení příznaku opět zjistíme, jestli během čekání na zprávu zpráva dorazila nebo nedorazila. Pokud uspěje akce výběru zprávy ze vstupní báze zpráv, odstraníme příznak a tím zabráníme opakování tohoto plánu a novému čekání na zprávu.

```
{waitskladnik;
  <
        +((flag,'&L)),          // nastavení příznaku, který bude použit dále
        #(wms,(20)),            // čekání na zprávu po 20 cyklů
   @<
        $1,?(Skladnik),         // pokus o načtení zprávy od skladníka do registru $1
        $2,-[flag,'&L],         // odstranění příznaku, registr $1 ochráníme přepnutím
        $1,#(lst,(th,''&1))     // výsledek je vzat ze seznamu a uložen opět do $1
   >,
```

Přijatá zpráva byla ve formě dvojice (odesilatel, (telo_zpravy)*). Za výsledek zprávy považujeme první prvek v těle zprávy.

```
   $2, -[flag,'&4],         // test existence příznaku a ochránění registru přepnutím
   @^(waitskladnik)         // pokud prošel, zpráva nebyla doručena, provedeme plán znovu
  >
};
```

4.5 Implementace ALLL agentů pro bezdrátové senzorové sítě

Jedněmi ze systémů, pro které jsme také implementovali ALLL prvky, jsou systémy bezdrátových senzorových sítí (Wireless Sensor Networks, WSN). Tyto sítě jsou tvořeny malými výpočetními uzly, které jsou schopny snímat veličiny svými senzory, pracovat samostatně v neznámém prostředí, mají vlastní zdroj napájení a komunikují bezdrátově. Uzly bývají rozmístěny v nějakém prostředí, které může být člověku nedostupné, nebo těžko dostupné, a proto musí být schopny pracovat samostatně a s vlastním zdrojem napájení po dobu v řádu měsíců či let. Úlohou sítě je obvykle monitorovat toto prostředí a informovat, pokud v něm dojde k nějakým definovaným událostem.

Tato kapitola bude demonstrovat možnost implementace ALLL prvků v takovýchto systémech a ukáže na příkladu jejich praktické využití. Nyní stručně uvedeme základní informace o WSN systémech. Jejich podrobnější popis naleznete například v [78].

4.5.1 Struktura bezdrátových senzorových sítí

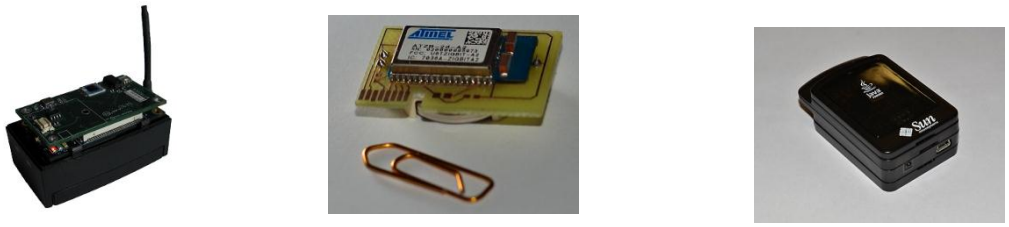
Rozmístění jednotlivých uzlů v prostředí je obecně libovolné. Jedná se proto o takzvané ad-hoc sítě, které kromě nespecifikované topologie mají schopnost se vyrovnat se změnou topologie posunutím, přidáním, nebo odstraněním uzlu ze sítě. Propojení mezi WSN a jinými sítěmi je realizováno přes základnové stanice, které bývají propojeny s řídícími počítači a skrz které probíhá komunikace s jinými podsystémy.

Mezi klasické přístupy k realizaci WSN patří jejich realizace podle specifikací ZigBee [79]. Tyto specifikace definují protokoly, které jsou na části linkové a síťové vrstvě protokolového zásobníku. ZigBee vytváří členění sítě tak, že rozdělí senzorové uzly na plně funkční a uzly s částečnou funkčností. Uzly s částečnou funkčností pouze měří veličiny a přes směrovací uzly posílají tyto veličiny řídícímu uzlu seskupení (klastru, cluster head). Směrovače i řídící uzly jsou uzly s plnou funkčností a nároky na jejich zdroje jsou vyšší, protože musí provádět více složitějších výpočtů, než uzly pouze

sbírající data. V případě ZigBee systémů je směrování dat v sítí dáno pozicí řídících uzlů. Obvykle se v aplikacích pro WSN směruje směrem od a k základnovým stanicím. Adresování v síti je na rozdíl od obvyklých systémů orientováno na data nebo na pozici výskytu uzlů v prostoru. Operátor má zájem získat určitá data, která například znamenají výjimečnou situaci v systému, nebo směrovat příkazy uzlům v určitém místě sítě či uzlům s požadovanými možnostmi a prostředky.

4.5.2 Zařízení pro bezdrátové senzorové sítě

Uzly senzorové sítě tvoři malá výpočetní zařízení. Senzorový uzel je obvykle tvořen mikrokontrolerem obsahující transciever (jednotku pro komunikaci rádiem) schopnou komunikovat na fyzické vrstvě protokoly IEEE 802.15.4, vlastním zdrojem energie a sensory. V současnosti jsou v odborné komunitě považovány za referenční senzorové uzly založené na mikro kontrolérech Améba, původně vyvinuté na univerzitě v Barkley, dále podporovány firmami Crossbow, a v posledních letech firmou MEMSIC. Tyto senzorové uzly jsou běžně nazývány anglickým termínem '*mote*'. Současnou podobu těchto uzlů ukazuje obrázek 10 (a). Jiným typem senzorových uzlů jsou uzly SunSPOT [80], které jsou založeny na mikroprocesorech ARM. Uzel SunSPOT je uveden na obrázku 10 (c). Obrázek 10 (b) ukazuje uzel FIT mote, který je stejně jako uzly Crossbow/MEMSIC založen na mikrokontrolerech ATMega a je vyvinut na FIT VUT v Brně. O dalších typech senzorových uzlů je možné se dočíst například v publikaci [81].



(a) (b) (c)

Obrázek 10: Moduly pro bezdrátové senzorové sítě. (a) IRIS mote, (b) FIT mote, (c) SunSPOT

Senzorové uzly jsou specifické hlavně tím, že jejich prostředky jsou omezeny. Dynamické paměti v mikrokontrolerech ATmega jsou v jednotkách kilobajtů, poslední verze pracují s mikrokontrolery ATMega 1281 s 8kB dynamické paměti. Omezené zdroje energie a požadavek na fungování senzorů v řádu let bez přímého zásahu člověka vedou k vyvíjení šetrných aplikací a také k hledání cest, jak sbírat energii z prostředí, ve kterém jsou senzorové uzly umístěny.

4.5.3 Vývoj aplikací pro bezdrátové senzorové sítě

Programování bezdrátových senzorových sítí nejčastěji bývalo a stále je prováděno v prostředí TinyOS (aktuální verze 2) v jazyce nesC [82]. TinyOS je spíše než operační systém knihovnou, skrz kterou lze přistupovat k hardware senzorových uzlů. Jazyk nesC je modulární jazyk s událostmi, které se mohou šířit buď od hardware k aplikacím, nebo naopak ve formě příkazů od aplikací směrem k hardware. V rámci sítě uzly interagují pomocí aktivních zpráv, které vzdáleně spouští procesy na straně příjemce. Tento způsob je využíván hlavně u uzlů a aplikací šetrných ke spotřebě energie, což je případ aplikací pro WSN. Existují i pokusy o vytvoření dynamické a mobilní verze systému TinyOS [83].

Vedle TinyOS existuje i řada jiných prostředí pro realizaci aplikací pro WSN. Například systém Contiki [84] je operačním systémem pro internet věcí a malá zařízení s omezenými energetickými zdroji, který podporuje komunikaci internetovými protokoly IPv6. Contiki lze implementovat pro uzly IRIS a

MICAz, jelikož existuje jeho verze pro mikrokontrolery Atmel AVR. Pro programování sítí používajících protokoly Zig-Bee lze také použít prostředí BitCloud [85].

4.5.4 Agenti v systémech WSN

Agentní paradigmata a WSN jako prostředí pro multiagentní systémy v současnosti nepatří mezi hlavní přístupy k realizaci systémů se senzorovými uzly. Přesto i tak existuje řada skupin, které se implementací agentních systémů pro senzorové sítě zabývají.

Pro a proti použití agentů v bezdrátových senzorových sítích lze nalézt řadu argumentů. Proti jejich použití hovoří hlavně energetická náročnost na spravování, intepretaci a hlavně přenos agentních kódů. Interpretace racionálních agentů je navíc na vyšší úrovni abstrakce, než jsou zmíněné prostředí TinyOS a další běžně používané prostředí pro realizace WSN systémů.

Argumenty podporující realizace agentů pro WSN jsou založeny na tom, že nasazení uzlů v reálných prostředích, ve kterých mají pracovat samostatně bez zásahu operátora v řádech měsíců a let, vyžaduje autonomnost těchto uzlů [86]. Během jejich fungování v systému může dojít k situacím, kdy je třeba přeprogramovat tyto uzly na dálku a na stálo nebo dočasně rozšířit jejich funkčnost bez nutnosti přímého kontaktu operátora s nimi. Další výhodu spatřujeme ve fázi vývoje aplikace, kdy vyvíjený systém a jeho chování lze nejprve provést podle multiagentních metod, ověřit správnost fungování tohoto systému například odpovídajícím simulačním modelem, a pak tento multiagentní prototyp převést na stejně funkční systém, kde je interakce mezi uzly realizována například již zmíněnými aktivními zprávami.

4.5.5 Přehled existujících řešení pro realizaci agentů pro WSN

Vývoj agentních aplikací v současnosti umožňuje několik systému, které přímo podporují realizaci agentů ve WSN, nebo realizaci systémů s některými rysy agentních systémů, jakými jsou například podpora služeb, migrace kódu, či programování systému inspirované agentním programováním. Některé takové systémy nyní stručně představíme.

4.5.5.1 AGILLA

Mezi reprezentanty prostředí pro implementaci agentů pro bezdrátové senzorové sítě patří systém Agilla [87]. V současnosti je k dispozici verze pro TinyOS v.1 a na verzi pro systém TinyOS v.2 se dle našich informací pracuje. Agilla je navržena v souladu s požadavky kladenými na implementace software pro WSN. Tedy těmi, že cena (energie) za transfer, uchovávání a interpretaci agentního kódu má být minimální. Stejně jako u většiny ostatních agentních systémů se i zde předpokládá, že agenti jsou mobilní. To znamená, že jsou schopni měnit svoji fyzickou pozici v systému tím, že přejdou z jednoho senzorového uzlu na jiný. U softwarových agentů to znamená také to, že řídící kód a data jsou udržovány v pamětech některého z uzlů v síti.

Instrukční sada systémů Agilla se skládá z instrukcí pro aritmetické, logické, řídící skokové operace, a také pro operace uspávající agenta na určitou dobu. Dalšími operacemi jsou operace pro přesun agentů v rámci sítě a pro klonování agentů. Výpočty probíhají na zásobníku. Operandy jsou umístěny na vrchol zásobníku a po výpočtu jsou nahrazeny výsledkem operace. Zápisem kódů Agilla připomíná spíš jazyk symbolických instrukcí, než agentní jazyky. Přesto se jedná v současnosti o nevýznačnějšího reprezentanta nasazení agentů ve WSN.

4.5.5.2 Sensorware

Sensorware [88], [89] je platforma, která je schopna interpretovat mobilní skripty v jazyce Tcl. Sensorware umožňuje zpracovávat události, na které je definován obslužný skript. Událostmi, které je možné definovat, jsou příchod zprávy, překročení nastavené meze snímané veličiny, naplnění vyrovnávací paměti, nebo uplynutí stanoveného času. Při zpracování skriptu je možné činit komunikaci v rámci systému, stanovovat nové události, či měnit stav platformy. Na rozdíl od systému Agilla je jeho nasazení směrováno do větších systémů s většími prostředky, než jsou MicaZ/IRIS senzorové uzly. Prototyp tohoto systému byl vyvinut pro iPAQ.

4.5.5.3 AFME

Systém, či lépe řečeno vývojové prostředí pro agentní systémy, nazvaný Agent Factory Micro Edition (AFME) [90] je založen na systému Agent Factory, který je vývojovým prostředím pro multiagentní systémy s podporou několika různých způsobů implementací agentů. Jedním z podporovaných způsobů je použití Agentně orientovaného programování a jazyka AFAPL2 [91], který jsme již zmínili v kapitole 2. Dalšími jazyky, které tu jsou pro programování agentů k dispozici, jsou jazyky AF-AgentSpeak vycházející z AgentSpeak(L) a jazyk pro specifikaci reaktivních agentů AF-TeleoReactive. AFME je malá verze AgentFactory pro programování aplikací pro malá zařízení včetně bezdrátových senzorových sítí a z uvedených technik jediná podporuje programování BDI agentů. Je zaměřena na jejich realizaci v senzorových uzlech SunSPOT díky jejich podpoře prostředí Java J2ME, které AFME využívá.

4.5.5.4 MAPS

Další middleware, který podporuje programování mobilních agentů, je MAPS (Mobile Agent Platform for SunSPOTs) [92]. Už z názvu plyne, že se opět jedná o řešení pro seznorové uzly SunSPOT. Platforma je implementována v prostředí Java s využitím knihoven SunSPOT k přístupu ke zdrojům senzorového uzlu. V agentním kódu jsou definované události, na které má agent reagovat, a akce, kterými má reagovat. Platforma zajišťuje migraci agentů, předávání zpráv, vymezuje jmenný prostor agentů a spravuje přístup agentů ke zdrojům na platformě.

4.5.5.5 HERA

HERA (Hardware Embedded Reactive Agent platform) [93] je agentní platforma založená na platformě služeb SYLPH [94], která implementuje servisně orientovanou architekturu do WSN senzorových uzlů. Agenti pro tuto platformu nejsou mobilní, ale může jich být vice na jednom uzlu, mohou spolu komunikovat a sjednávat si navzájem služby. Účel využití agentního principů je zde dynamická alokace zdrojů a služeb v systému bezdrátových senzorových sítí.

Další systémy, které jsou k dispozici pro použití jako middleware platformy pro mobilní kódy a agenty v systémech WSN, jsou diskutovány například v [95, 96, 97, 98, 99, 100]. Existují i simulační prostředí, které se přímo vztahují k simulacím agentních systémů v prostředích WSN. Jako zástupce takového simulačního prostředí můžeme uvést systém SAMSON [101], který je postaven na BDI agentním prostředí JASON.

4.6 Agentní aplikace v systému WSageNt

V této kapitole zmíníme implementace ALLL agenta pro malá zařízení, jakými jsou ‘mote’ uzly senzorové sítě. První verze takového systému vznikla v rámci diplomové práce Jana Horáčka [71]. Původně šlo o ověření fungování ALLL agentů v reálných systémech na reálných výpočetních zařízeních. ALLL agenti se nakonec ukázali jako velmi vhodní pro nasazení v systémech WSN. Jejich

výhodou byla velmi malá délka kódu, v kterém je možné napsat konstrukce řídící běh agenta v rámci sítě a také řídící jejich činnosti na uzlech. V našich experimentech jsme dokázali tvořit funkčně zajímavé aplikace, ve kterých agenti dosahovali délky stovek bajtů, maximálně několika málo kilobyte. Možnost snadno tvořit funkční distribuované aplikace pro WSN a mít přitom řešení, které respektuje omezené prostředky zařízení v této síti je pro nás potvrzením smysluplnosti pracovat na tomto řešení a prezentovat jej jako vhodný prostředek na poli programování systémů bezdrátových senzorových sítí. Proto také tento text zakončíme pojednáním o systému WSageNt, jak jsme nazvali realizaci agentní platformy pro interpretaci mobilních ALLL agentů implementovanou pro WSN senzorové uzly, který je schopen v tomto prostředí fungovat jako vzdáleně programovatelný agentní systém.

4.6.1 Principy činnosti systému WSAgeNt

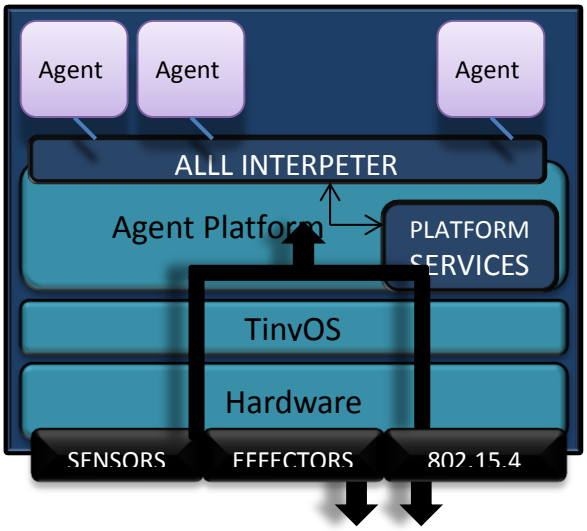
Kdybychom vztáhli systém WSageNt k modelům systémů v T-Mass, pak bychom opět viděli vazbu mezi systémovými prvky agenta a platformy, kteří se nachází v nějakém podsystému prostředí, které může obsahovat další platformy s agenty. Agentem je tady řetězec, který je uchováván v dynamické paměti některého ze senzorových uzlů, na kterém je implementována WSageNt platforma.

Platforma může obsahovat více agentů tím, že přidělí každému z agentů část dynamické paměti uzlu. Platforma také může manipulovat s agentním kódem během jeho interpretace, ale zároveň může kód měnit vkládáním dat přímo do bází agenta. Komunikace mezi agentem a platformou je jen virtuální. Platforma zprávy, které jsou pro ni určené, přebírá v okamžiku, kdy interpretuje příslušnou akci. Směrem k agentovi předává zprávy tím, že přímo mění stav jeho aktuálního registru. Oproti dosavadním systémům je intepretace agenta závislá na tom, zdali je po transformaci agentního kódu dostatek místa na jeho umístění. Některé akce mohou mít za následek zvětšení velikosti agenta, respektive jeho kódu, nad vymezený prostor v paměti. Tyto akce pak neuspějí a agent je také jako neúspěšné interpretuje. Dochází tím ke korektní interpretaci agenta pouze s takovým omezením, že některé zamýšlené akce nejde provést, což ovšem není v rozporu s principy agentního programování.

Jazyk, který je dialektem jazyka ALLL, je interpretován automatem. Jeden krok automatu provede jednu akci agenta a patřičně změní jeho stav. Vedle toho platforma snímá veličiny z prostředí, zpracovává je a případně je předává agentovi, pokud si agent u platformy zaregistroval takovou službu. Platformy též řídí migraci agentů mezi jednotlivými platformami a předávání zpráv mezi agenty. Kromě toho nabízí běžné platformní služby, o kterých jsme mluvili již dříve, a také některé služby specifické pro systémy WSN.

4.6.2 Architektura systému WSageNt

Senzorový uzel se v systému WSageNt skládá z hardwareové části, která je v současnosti systémem senzorového uzlu podle Barkley/Crossbow architektury. Prostředí, ve kterém je agentní platforma realizována, je programové prostředí TinyOS v.2. Vlastní implementací systému je agentní platforma, která má za úkol volat interpreter ALLL jazyka, jenž následně vykoná jeden interpretační krok. Platforma také zajišťuje řádnou interakci agenta s okolím. Struktura senzorového uzlu s agentní platformou systému WSageNt je ukázána na obrázku 11.



Obrázek 11: Struktura senzorového uzlu systému WSageNt

Intepretace kódu je prováděna automatem, který pracuje s jednotlivými částmi kódu jako s vyhledávacími tabulkami, to v případě bází, dále se zásobníkem, ve kterém je uložen aktuální záměr, a nakonec také s registry. Tím se liší od předchozího systému T-Mass(X), ve kterém jsou agenti přeloženi do objektové reprezentace, a agent je interpretován voláním metod objektů reprezentujících jednotlivé akce. Interpretační kroky jsou prováděny tak dlouho, dokud zásobník obsahuje alespoň jeden plán s alespoň jednou akcí.

Celou činnost WSageNt platformy lze vnímat jako dva paralelní procesy, kde jeden proces zpracovává data z okolního prostředí, a druhý interpretuje agentní kód a vykonává služby platformy. Logování dat na této platformě může probíhat pro všechny možné vstupy dat prvku, to znamená, že vedle přijímaných zpráv od ostatních prvků v systému je také možné logovat například i snímané veličiny ze senzorů. Paralelně s tím je volán interpret agentního kódu, pokud není jeho provádění zablokováno například službou čekání na specifikovanou událost. Po provedení interpretačního kroku jsou provedeny další činnosti, které souvisí se spuštěnými platformními službami, například manipulace s naměřenými hodnotami, počítání průměrné hodnoty nějaké veličiny atd.

4.6.3 Jazyk ALLL pro platformu na uzlech WSN

V systému WSageNt se syntaxe jazyka ALLL od původně deklarované syntaxe mírně liší. Největší změnou je to, že kvůli minimalizaci délky kódy byly odstraněny čárky mezi akcemi. Při nepřímém vyvolávání plánu je vynechán symbol zavináče a deklarace plánu se oproti systému T-Mass(X) vrací k syntaxi uvedené v kapitole 4.2.1. V některých verzích systému je pro volání služeb platforem namísto mřížky používán symbol dolaru a interní akce jsou zapsány také se symbolem dolaru, za kterým následuje identifikátor akce uzavřený v závorkách. Také některé systémy používaly jinou reprezentaci podplánů, ve kterých neuváděly začátek podplánu, ale pouze zarážky pro konec podplánu, respektive pro místa, po které má být plán zkrácen v případě neúspěšného provedení akce. Tato reprezentace ale neodpovídala teoretický předpokladům, které jsme v této práci prezentovali, a v současnosti se již nepoužívá. Jinak se ALLL pro systém WSageNt jazyk podobá ALLL jazyku pro systém T-Mass(X).

4.6.4 Služby agentní platformy v systému WSageNt

Vedle obvyklých služeb, které jsme zmiňovali v kapitole 4.1.4. je rejstřík služeb platforem v systému WSageNt rozšířen o některé specifické služby. Uvedeme si takové z nich, které jsou běžně využívány v

realizovaných implementacích tohoto systému a které budeme v dalších kapitolách využívat i u příkladů použití tohoto systému.

Zásadní pro fungování ALLL agentů ve WSN systémech je jejich mobilita. Agentní platforma umožňuje agentům pohybovat se v síti tím, že poskytuje službu přenosu agenta z této platformy na jinou, specifikovanou jménem platformy. K tomu, aby agent zjistil, kam je možné se přesunout, slouží služba platformy pro prohledávání okolí. Výsledkem této služby je seznam jmen platforem na dostupných uzlech sítě. Mobilita sebou nese další nároky na platformu a to směrování, nebo řízení pohybu agentů v rámci systému. Současný stav vývoje WSageNt je takový, že je podporováno zaznamenávání informaci o pohybech agentů na platformách ve formě jejich pachových stop a směrování agentů v rámci sítě probíhá na základě těchto pachových stop.

Představíme několik služby, které budeme potřebovat pro pochopení kódu příkladu realizace agenta pro systém WSageNt. Jsou to následující služby a jejich operace.

#(f,&1) – je jednou ze dvou výpočetních služeb, které zde představíme. Výsledkem provedení této služby je to, že první prvek z argumentu je umístěn do aktivního registru. U systému T-Mass jsme už takovou službu viděli, ale s trochu jiným kódováním.

Také jsme již uvedli, že pro fungování agentů ve WSN je důležitá služba, která zjistí platformy na bezprostředně dostupných uzlech. Tuto službu si nyní představíme.

#(n) – jedná se o řídící službu, jejímž výsledkem je naplnění báze představ trojicemi popisujícími okolní platformy. Trojice mají formát ‘(NB, id, strenght)‘, kde ‘id ‘ je jméno platformy a ‘strength’ je síla signálu, zjištěná mezi domovskou platformou a uvedenou vzdálenou platformou. ‘NB’ je konstantní prvek, který identifikuje tyto informace v agentově bázi představ.

Z ostatních použitých služeb platformy je jedna rozdělena podle tří svých operací. Tato služba je registrována jako ‘t’ a týká se podpory mobility agenta v rámci systému. V našem případě ji využijeme pro získání informací o platformě, na které se agent nachází, jména platformy, ze které agent na aktuální platformu přišel a zjištění, zdali na dostupné platformě daného jména již agent během svého působení v systému byl, nebo doposud ne.

#(t,(i)) - je to operace služby ‘t’, která naplní aktuální registr identifikátorem platformy, na které se agent právě nachází.

#(t,(b,f)) – jedná se o operaci sledování pohybu agentů na uzlu. Argumentem je dvojice, která určuje operaci zjištění platformy, ze které byl agent na aktuální platformu doručen. Jelikož agent může během své existence přijít na jednu platformu ve více okamžicích z různých platforem, udává druhý prvek argumentu, v tomto případě atom ‘f’, že žádáme platformu o první ze záznamů příchodů agenta.

#(t,(v,id)) – platforma zjistí, zdali tento agent byl již dříve na platformě adresované jako ‘id’. Tato služba skončí úspěchem, pokud agent na této platformě nebyl. Pokud byl, nebo platforma daného jména není v dosahu, služba skončí jako neúspěšná.

Jako poslední si uvedeme službu, která zajistí přenesení agenta v systému.

#(m,(&2,s)) – jedná se o službu přenesení agentního kódu na platformu, která je adresována prvním atomem v argumentu. Druhým atomem v argumentu je atom ‘s’, který znamená to, že po přenesení agenta se činnost tohoto agenta na původní platformě ukončí. Dojde tak k přechodu agenta z jedné platformy na druhou a ne k jeho klonování v systému.

4.6.5 Případová studie: průchod agenta ad-hoc sítí

Příkladem použití agentů v bezdrátových senzorových sítích bude průchod agenta touto sítí. Zadání pro algoritmus bude takové, že agent má navštívit každý uzel alespoň jednou a vykonat na něm nějakou činnost. Účel tohoto algoritmu byl původně ten, že agent měl zjistit dostupnost jednotlivých uzlů navzájem. Na každém uzlu, který poprvé navštívil, zjistil okolí včetně sil signálů k jednotlivým sousedním uzlům a tyto informace odeslal na základnovou stanici. Ze základnové stanice data přejímaly řídící stanice, které z těchto dat sestavovaly přibližnou topologii sítě. Nyní uvedeme algoritmus pro průchod sítí, jak byl popsán v publikaci [72] a z této publikace budeme v následujícím textu vycházet.

Algoritmus nejprve zapíšeme v pseudojazyce, který lépe ukáže jeho podstatu. Potom jej převedeme do jazyka ALLL.

```
Algoritmus:
    proveď činnost na platformě
    N = seznam jmen platforem na sousedních uzlech
    while(není prázdný N)
        platforma = první prvek z N
        odstraň tento prvek z N
        if(agent nebyl na platformě)
            přejdi na platformu
            spusť algoritmus
    vrať se na původní platformu
```

Pro podporu mobility agentů použijeme pachové stopy, které zaznamenávají výskyt agentů na platformách a se kterými pracují některé služby platforem, jak jsme je uvedly výše. Agent na první platformě spouští uvedený algoritmus a nejprve zde provede nějakou požadovanou činnost. Dále zjistí dostupné platformy v okolí a vytvoří z nich seznam. Z tohoto seznamu postupně vybírá jednotlivé platformy a pro každou z nich zjišťuje, zdali na ní již byl, nebo doposud nebyl. Pokud nebyl, přesune se na ni a spustí celý tento algoritmus od znova. Pokud nezbývá žádná platforma v okolí, na které by ještě nebyl, vrací se na platformu, ze které přišel.

Algoritmus se svým fungováním podobá algoritmu prohledávání stavového prostoru s navracením. Agent projde všechny platformy, které jsou dostupné, a vrátí se na výchozí platformu. Jak by tento algoritmus byl zapsán v jazyce ALLL ukážeme níže i s doprovodnými komentáři.

Kód agenta v jazyce ALLL začneme tím, že uvedeme záměr. Ten obsahuje jen jednu akci a to akci volání podplánu.

(^nb)

Jako první uvedeme plán, který je vyvolaný z prvotního záměru. Plán nazvaný ‘nb’ má čtyři akce, které až na jednu spouští podplány. Nejprve dojde ke zpracování dat na uzlu, to je k nějaké činnosti, kterou agenti na jednotlivých platformách mají vykonávat. Další akce vyvolá službu platformy, která naplní bázi informacemi o dostupných platformách v okolí. Následuje vyvolání podplánu, který provede přenos agenta na sousední platformy, na kterých doposud nebyl, a na závěr se agent vrátí na platformu, ze které na aktuální platformu přišel. Celý plán vypadá následovně.

(nb,(^(^(addblf)#(n)^(cycle))^(back)))

Další z plánů pouze přidá informaci o jménu platformy, na které se právě nachází, do agentovy báze představ. Po přepnutí aktivního registru na &1 zjistí jméno platformy zavolání příslušné služby a výsledek uloží do báze ve dvojici s atomem ‘vis’, (navštíven – visited).

(addblf,(&1#(t,(i))+(vis,&1)))

Plán ‘cycle‘ je plán, který v cyklu provede přechody na okolní platformy, na kterých agent doposud nebyl. Tělo plánu postupně vybere do registru &1 trojice popisující sousední platformy, z nich do registru &2 vybere první trojici, tuto odstraní z báze představ, a zavolá podplán, který případně na platformu uvedenou ve vybrané trojici agenta přesune a provede tam patřičnou činnost. Po skončení podplánu předpokládáme, že se agent vrátil na původní platformu a spustíme plán znovu, aby byl přesunut na případnou další nenavštívenou platformu v okolí.

(cycle,(&1[NB,_,_]&2#(f,&1)-&2^(testnmove)^(cycle)))

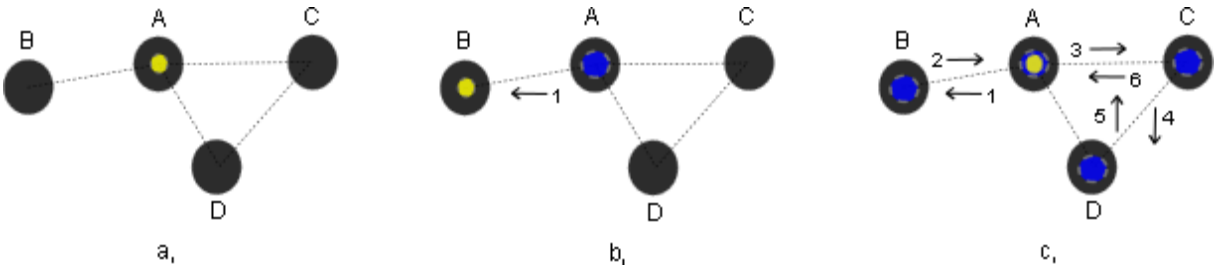
Zmíněný podplán má za úkol provést zjištění, zdali na vybrané sousední platformě agent již byl, nebo ne. Pokud ano, skončí akce vyvolání služby neúspěchem. Pokud služba skončí úspěšně, přeneseme agenta na tuto platformu. Nejdříve ale odstraníme z báze všechny informace o sousedech, protože tyto informace už nebudou na nové platformě platné. Poté přeneseme agenta na platformu nového uzlu a spustíme na něm celý algoritmus znovu.

(testnmove,(#(t,(v,&2))-[NB,_,_]#(m,(&2,s))^(nb)))

Poslední plán je volán, pokud již není na daném uzlu plánována další činnost a agent se má vrátit na uzel, ze kterého na dané místo přišel. Po přepnutí aktivity registrů je z platformy zjištěno jméno uzlu, ze kterého agent přišel, a další akce zavolá službu přesunu na tuto platformu. Na původní platformě se činnost agenta ukončí. Již na nové platformě agent znovu naplní svoji bázi představ dostupnými platformami.

(back,(&3#(t,(b,f))#(m,(&3,s)) #(n)))

Činnost agenta si ukážeme na jednoduchém příkladu. Na obrázku 12 (a), je ukázána senzorová síť se čtyřmi uzly. Na uzlu označeném jako ‘A’ se nachází náš agent. Agent je ve stavu, kdy kromě prvotního záměru a plánů v bázi plánů obsahuje pouze aktuální hodnoty registrů nastavené na ‘(null)’. Dále budeme demonstrovat činnost agenta v systému tak, že ukážeme jeho podobu v některých okamžicích jeho fungování. Uvedeme ale pouze stav agentova záměru a vynecháme báze představ, plánů a registry. Pokud by tyto součásti agenta hrály nějakou roli v aktuálním stavu výpočtu, uvedeme to v příslušných komentářích.



Obrázek 12: Příklad průchodu agenta senzorovou sítí

Agent je umístěn na prvním uzlu (například je sem zaslán ze základnové stanice) a jeho záměrem je vyvolán ‘nb’ pro průchod sítí. V dalších krocích dojde ke spuštění podplánu, který je vedle akce vyvolání plánu ‘back’ jedinou součástí plánu ‘nb’, a dále dojde k vyvolání plánu, který provede na uzlu nějakou činnost, v tomto případě si agent uloží do své báze představ informaci, že se nacházel na uzlu daného jména.

I0: (^(nb))

I1: ((^(^(addblf)#(n)^(cycle))) ^(back))

I2: (((&1#(t,(i))+(vis,&1)), #(n)^(cycle))^(back))

…

Nyní se přesuneme do stavu po vykonání podplánu uložení informace do báze. Agent pokračuje ve vykonávání plánu ‘nb’ vyžádáním služby platformy pro nalezení dostupných platforem v okolí. Služba bude mít za následek naplnění báze trojicemi s informacemi o těchto platformách. Poté je vyvolán plán, který přenese agenta na nějakou platformu, na které agent doposud nebyl.

I3: ((#(n)^(cycle))^(back))

I4: ((^(cycle)) ^(back))

I5: (((&1[NB,_,_]&2#(f,&1)-&2^(testnmove)^(cycle))) ^(back))

…

V plánu ‘cycle’ bylo provedeno prvních pět akcí a agent se nachází před spuštěním podplánu, který přesune agenta na sousední platformu, pokud na ní nebyl. V registru &1 se nachází jméno první z dostupných platforem, kterou nalezl ve své bázi představ. Tedy vyvolá plán pro případný přesun a rozšíří jím aktuální záměr agenta.

I6: (((^(testnmove)^(cycle))) ^(back))

I7: ((((#(t,(v,&2))-[NB,_,_]#(m,(&2,s))^(nb)) ^(cycle))) ^(back))

…

Agent provedením služby platformy (t,(v,&2)), která skončila úspěšně, zjistil, že na platformě, jejíž popis má ve svém registru, doposud nebyl , a chystá se provést vyvolání služby platformy, která jej přenese na požadovanou platformu a ukončí jeho činnost na stávající platformě.

I8: ((((#(m,(&2,s))^(nb)) ^(cycle))) ^(back))

Agent je přenesen (obrázek 12 (b)) a pokračuje na dané úrovni jedinou akcí a to vyvoláním plánu průchodu sítí, tak jak to činil i v kroku I0 v plánu prvotního záměru.

I9: (((^(neb)) ^(cykle)) #^(bací))

…

Na nové platformě se chová obdobně, jak se choval na původní platformě. Po spuštění plánu 'cycle' se agent chystá z báze představ vybrat informaci o okolních uzlech. V jednom cyklu vybere informaci o tom, že v okolí se nachází platforma 'A', ale plán 'testnmove' neprojde přes službu platformy na testování předchozí přítomnosti agenta na této platformě. V bodě I10 se nacházíme v druhé iteraci plánu 'cycle', kdy agent vyhledává nějakou další platformu na dostupném uzlu. Žádná takové se ale v bázi nenachází a plán končí neúspěchem. Na úrovni plánu 'nb' dojde k vyvolání podplánu 'back'.

I10: (((((((&1[NB,_,_]&2#(f,&1)-&2^(testnmove)^(cycle))) ^(back))) ^(cycle))) ^(back))

I11: ((((((^(back))) ^(cycle))) ^(back))

Vyvolaný podplán vyhledá pomocí příslušené služby platformy tu platformu, z níž na aktuální platformu přišel a vrátí se na ní.

I12: (((((((&3#(t,(b,f))#(m,(&3,s)) #(n)))) ^(cycle))) ^(back))

I13: ((((((#(m,(&3,s)) #(n))   )) ^(cycle))) ^(back))

Agent je přesunut zpět na původní uzel, naplní bázi představ dostupnými platformami a pokračuje spuštěním nového cyklu pro přesun na nenavštívenou platformu.

I14: (((((#(n))   )) ^(cycle))) #^(back))

Celý průběh průchodu agenta sítí je ukázán na obrázku 12 (c). Agent se dostane na obě zbývající platformy a nakonec se vrátí na původní platformu 'A', na které ukončí svoji činnost. V bázi představ ale bude mít v tom okamžiku informace o všech navštívených platformách, což jsou také všechny dostupné platformy v síti.

Na závěr chceme poukázat na jednu nepříjemnost, a to tu, že na takto implementovaném agentovi můžeme vidět, že v kroku I9 je agent v podobném stavu jako na počátku své činnosti, akorát s sebou nese dvě akce navíc. S každým dalším posunem vpřed se agentní kód zvětšuje takovýmto zvětšením záměru, což je nepříjemné mimo jiné kvůli nárokům na omezené zdroje hostitelských uzlů. Další diskuse jak transformovat kód na kód s ekvivalentním chováním, optimalizací kódů a podobně je již nad rámec tohoto textu.

5 Současný stav výzkumu v oblasti a směry do budoucna

Témata uvedená v této práci tvoří jen část z oblastí výzkumu, kterým jsme se v posledních letech věnovali. Systémy a jazyk ALLL byly často využívány při zpracování postupových prací, výzkumu v rámci skupiny inteligentních systémů i při řešení projektů, které se týkaly distribuovaných inteligentních systémů.

Pokud v přehledu dalších aktivit vyjdeme z příkladu uvedeného na konci předchozí kapitoly, tak ten byl otestován na reálné senzorové síti s několika desítkami uzlů a byla potvrzena jeho funkčnost. Aplikace byly rozšířeny o zjišťování síly signálů RSSI [102] od ostatních uzlů a jejich zpracování pro sestavení předpokládané topologie sítě. Dalšími oblastmi, kterými se zabýváme v souvislosti s agenty a systémy WSN, jsou oblasti bezpečnosti v těchto systémech [103, 104, 105, 106]. S tou souvisí reputační systémy a modely důvěry, které mohou být vhodné pro tuto oblast. Zkoumá se nasazení vícekontextového modelu důvěry [107, 108] pro rozhodování agentů v distribuovaných aplikacích. Zabýváme se také bezpečností mobilních kódů ve WSN [109] a realizací bezpečných agentních platforem [110, 111]. Také došlo k návrhu pokročilého algoritmu pro směrování agentů ve WSN na základě geografických informací o jejich poloze v prostředí [76]. Platformy pro tyto systémy byly vedle bezdrátových senzorových sítí založených na mikrokontrolerech Atmel implementovány i pro zařízení s operačním systémem Android a další. Dochází k návrhu a realizaci jazyka AHLL [112], který je postaven vyšší úrovni abstrakce než je jazyk ALLL, ale zachovává rysy ALLL jazyka v tom, že definuje aktuální agentův záměr, umožňuje práci s bázemi, asynchronní komunikaci mezi agenty a volání služeb platforem.

Dalším původním systémem, který je realizován na našem pracovišti a je nyní fázi vývoje, je systém JAWS (Jade Agents to Wireless Sensor Networks) [113]. Tento systém propojuje různé systémy, které mají jako společné to, že pracují s ALLL agenty. V současnosti je možné propojit simulační systémy T-Mass a PNtalk s bezdrátovou senzorovou síť. Jádro tohoto systému tvoří populace JADE agentů, kteří fungují jako správci ALLL agentů, přenášejí je mezi simulátorem a reálnou senzorovou sítí a sledují provoz této sítě, například odposloucháváním zpráv daného typu. Pokud si některý z agentů v JADE zaregistruje službu odposlouchávání zpráv ze sítě, pak jsou mu tyto zprávy předávány. JAWS rovněž spravuje síť servisními agenty, uchovává její topologii, zjišťuje možné výpadky v sítí atd.. Předpokládáme, že systém JAWS podpoří návrh aplikací pro WSN tím, že umožní modelovat tyto systémy a provádět simulace těchto modelů k ověření správnosti návrhu. ALLL agenti, u kterých bude simulací ověřena správnost jejich chování, mohou být ve stejné formě umístěny do reálných sítí.

6 Závěr

Programování distribuovaných inteligentních systémů je z pohledu vývojáře programováním paralelních systémů se samostatně pracujícími prvky, které spolu komunikují předáváním zpráv, a které jsou schopny proaktivně a flexibilně jednat v obvykle ne zcela přístupném a dynamickém prostředí. Předložený text měl za cíl představit nový přístup k programování těchto systémů v jazyce, který v sobě skloubí prvky funkcionálního, procedurálního a agentně orientovaného programování a programování distribuovaných systémů s asynchronně komunikujícími prvky. Výsledkem snažení je nový jazyk, který je na nízké úrovni abstrakce, kombinuje uvedená paradigmata a je snadno interpretovatelný uzly distribuovaného systému. Formální specifikace principů, na kterých je náš původní jazyk postaven, tvořila podstatnou část této práce. Postupně jsme po jednotlivých systémech ukazovali nové vlastnosti těchto systémů. Od formálně specifikovaných otevřených ALLL prvků jsme přešli k realizaci distribuovaných systémů s těmito prvky a ke konkrétním realizacím takovýchto distribuovaných systémů. Zde jsme již zvolili praktičtější formu výkladu a společně s principy systémů, ve kterých jsou realizovány ALLL prvky interpretující ALLL jazyk, jsme uvedli i několik příkladů ilustrujících jejich použití.

Jako jedny z možných oblastí aplikace byly představeny oblasti modelování systémů a oblast bezdrátových senzorových sítí. Existující aplikace potvrzují použitelnost a dobrou funkčnost při začlenění ALLL agentů do reálných distribuovaných systémů, pokud potřebujeme nasadit mobilní programy, které jednají autonomně, a dokáži se v rámci systému pohybovat a využívat možností a služeb jednotlivých uzlů v síti. Modelování systémů s těmito agenty podporuje vývoj distribuovaných systémů tím, že agentní ALLL kódy je možné vyzkoušet v modelech uvažovaných systémů, ve kterých mají být později použity, a následně simulací ověřit, že chování ALLL agentů v těchto modelech odpovídá předpokladům. Stejné ALLL agenty je pak možné nahrát do zařízení, které pracují v reálných systémech a očekávat jejich správné chování. Do budoucna plánujeme užší propojení reálných systémů s jejich modely a vytvoření spojeného systému, ve kterém by ALLL agenti pracovali v obou podsystémech, systému modelu i v reálném systému, jako v jednom celku.

Formální základ, poskytnutý v této práci, má zlepšit možnosti zkoumání a modelování ALLL agentů a systémů s těmito agenty. Na obecnější úrovni mají zkušenost s modelováním a realizaci distribuovaných systémů a zapojení vhodných simulačních technik do procesu jejich vývoje vést k pokroku ve výzkumu na poli distribuovaných inteligentních systémů na FIT VUT v Brně, k čemuž snad svůj díl přinesla i tato práce.

Literatura

[1] F. Zbořil, Plánování a komunikace v multiagentních systémech, Brno: Brno University of Technology, 2004.

[2] F. Zbořil, P. Hanáček, J. Žák, J. Horáček, M. Drahanský, J. Hájek a A. Marvan, „Studie využití autentizačně - bezpečnostních technologií pro automatizaci a zlepšení efektivity vybraných procesů sledování stavu a pohybu majetku a osob ve firmě nebo instituci," Siemens CZ, Brno, 2011.

[3] M. Wooldridge a N. R. K. D. Jenning, „The Gaia Methodology for Agent-Oriented Analysis and Design," *Autonomous Agents and Multi-Agent Systems,* sv. 3, pp. 285-312, 2000.

[4] L. Padgham a M. Winkoff, Developing Intelligent Agent Systems, John Wiley & Sons, 2004.

[5] M. Wooldridge, Reasoning about Rational Agents, Cambridge: MIT Press, 2000.

[6] M. Winikoff, „Jack™ Intelligent Agents: An Industrial Strength Platform," *Multiagent Systems, Artificial Societies, and Simulated Organizations,* Sv. 1 z 215, II, pp. 175-193, 2006.

[7] R. H. Bordini, J. F. Hübner a M. Wooldridge, Programming Multi-agent Systems in AgentSpeak using Jason, John Wiley & Sons, 2007.

[8] M. Dastani, „2APL: a practical agent programming language," *Autonomous Agents and Multi-Agent Systems,* sv. 16, č. 3, pp. 214-248, 2008.

[9] M. Huber, *JAM Agents in a Nutshell,* Oceanside, CA, 1999.

[10] „Java Agent DEvelopment Framework," [Online]. Available: jade.tilab.com.

[11] FIPA, „Foundation of Intelligent Physical Agents," [Online]. Available: www.fipa.org.

[12] J. McCarthy, Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, Cambridge: Massachusetts Institute of Technology, 1960.

[13] „The Haskell Programming Language," [Online]. Available: http://www.haskell.org/haskellwiki/Haskell.

[14] A. Colmerauer a P. Roussel, „The Birth of Prolog," v *History of programming languages---II*, ACM, 1996, pp. 331 - 367.

[15] J. Lloyd a P. Hill, The Gödel Programming Language, MIT Press, 1994.

[16] F. Zbořil, „Modulární systém v logickém programovacím jazyce Godel," *Sborník prací studentů a doktorandů,* pp. 270-272, 2000.

[17] O.-J. Dahl, „The Roots of Object Oriented Programming,“ *Software Pioneers,* pp. 78 - 90, 2002.

[18] O.-J. Dahl, B. Myhrhaug a K. Nygaard, SIMULA 67 Common Base Language, Oslo: Norwegian Computing Center, 1968.

[19] D. C. Hyde, „Introduction to the Programming Language Occam,“ Bucknell University, Lewisburg, 1995.

[20] D. Gelenter, „Generative communication in Linda,“ *ACM Transactions on Programming Languages and Systems vol. 7,* pp. 80-112, 1985.

[21] „MPI-2: Extensions to the Message-Passing Interface,“ University of Tennessee, 1997. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm. [Přístup získán 6 10 2012].

[22] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall International, 2004.

[23] M. Wooldridge a N. R. Jennings, „Intelligent agents: Theory and practice,“ *The Knowledge Engineering Review,* sv. Vol. 10:2, pp. 115-152, 1995.

[24] L. Lamport, „Time, Clocks, and the Ordering of Events in a Distributed System,“ *Communications of the ACM,* sv. 21, pp. 558-565, 1978.

[25] D. B. Lange a M. Oshima, „Seven Good Reasons for Mobile Agents,“ *Communications of the ACM,* sv. 3, pp. 88,89, March 1999.

[26] M. Wooldridge, An Introduction to MultiAgent Systems - Second Edition, John Wiley & Sons, 2009.

[27] R. Brooks, „Intelligence without Reason,“ v *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

[28] R. Brooks, „Intellligence without Representation,“ *Artificial Intelligence 47,* pp. 139-159, 1991.

[29] R. Brooks, „A robust layered control system for a mobile robot,“ *IEEE Journal of Robotics and Automation,* pp. 14-23, 1988.

[30] Y. Shoham, „Agent Oriented Programming,“ *Artificial Intelligence 60,* pp. 51-92, 1993.

[31] A. Rao, „Modeling Rational Agents within a BDI-Architecture,“ v *Proceedings of the Second International Conference on principles of Knowledge Representation and Reasoning*, San Mateo, 1991.

[32] A. Rao a M. Georgeff, „Formal Models and Decision Procedures for Multi-Agent Systems,“ Australian Artificial Intelligence Institute (61), , Melbourne, 1995.

[33] R. W. Collier a G. M. P. O'Hare, „Modeling and Programming with Commitment Rules in Agent

Factory,“ v *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, IGI Publishing, 2009.

[34] M. Bratman, Inentions, Plans and Practical Reason, Harvard University Press, 1987.

[35] A. Rao, „AgentSpeak(L): BDI Agents speak out in a logical computable language,“ v *Agents Breaking Away, Lecture Notes in Artificial Intelligence*, Amsterdam, 1996.

[36] M. Dastani, F. Dignum a J.-J. Meyer, „3APL: A Programming Language for Cognitive Agents,“ v *ERCIM News*, 2003.

[37] M. d'Iverno, L. M. Kinny a M. Georgeff, „The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System,“ v *Autonomous Agents and Multi-Agent Systems*, 2004.

[38] M. d'Iverno, D. Kinny a M. Luck, „A Formal Specification of dMARS,“ v *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, 1998.

[39] K. Myers, *User Guide for the Procedural Reasoning System,* Menlo Park: SRI International, Artificial Intelligence Center, 1997.

[40] J. L. Austin, How to Do Things With Words, Cambridge (Mass.): Harvard University Press, 1962.

[41] T. Finn, Y. Labrou a J. Mayfield, „KQML as an agent communication language,“ v *Software Agents*, 1996.

[42] S. Kraus, Strategic Negotiation in Multiagent Environment, Cambridge: MIT Press, 2001.

[43] M. P. Wellman, A. Greenwald a P. Stone, Autonomous Bidding Agents, Cambridge: MIT Press, 2007.

[44] Y. Shoham a K. Leyton-Brown, Multiagent Systems, New York: Cambridge University Press, 2009.

[45] R. Milner, J. Parrow a D. Walker, „A Calculus of Mobile Processes, I and II,“ *INFORMATION AND COMPUTATION,* 1989.

[46] R. Milner, „Function as processes,“ *Automata, Languages and Programming,* Sv. %1 z %2LNCS, Vol 443/1990, pp. 167-180, 1990.

[47] J. C. Reynolds, „The Discoveries of Continuations,“ *LISP AND SYMBOLIC COMPUTATION: An International Journal, 6,* pp. 233-247, 1993.

[48] J. Bengtson, M. Johansson, J. Parrow a B. Victor, „Psi-calculi: A FRAMEWORK FOR MOBILE PROCESSES,“ *Logical Methods in Computer Science vol 7,* pp. 1-44, 2011.

[49] J. Bengtson, M. Johansson, J. Parrow a B. Victor, „Psi-calculi: Mobile processes, nominal data, and logic,“ v *Proceedings of LICS*, Los Angeles, 2009.

[50] T. Clark, „Specification and Implementation of a Multi-Agent Calculus based on Higher-Order Functions,“ University of Bradford Technical Report, Bradford, 1999.

[51] C. Fournet, G. Gonthier, J.-J. Levy a M. R. D. Luc, „A Calculus of Mobile Agents,“ v *LNCS 1119*, Springer, 1996, pp. 406-421.

[52] R. Milner, „The Polyadic PI-Calculus: a Tutorial,“ *The Polyadic PI-Calculus: a Tutorial,* 1991.

[53] D. Sangiorgi a D. Walker, The PI-calculus, A Theory of Mobile Processes, Cambridge: Cambridge University Press, 2001.

[54] L. Cardelli a A. D. Gordon, „Mobile Ambients,“ *Proceedings of the First international Conference on Foundations of Software Science and Computation Structure,* pp. 140-155, 1998.

[55] S. Rahimi, M. Cobb, D. Ali a F. Petry, „A Modeling Tool for Intelligent-Agent Based Systems: the API-Calculus,“ *Soft Computing Agents: A New Perspective for Dynamic Systems, the International Series "Frontiers in Artificial Intelligence and Application",* pp. 165-186, 2002.

[56] S. Rahimi a R. Ahmad, „ACVizualizer: A Visualization Tool for Api-Calculus,“ *Multiagent and Grid Systems,* pp. 271-291, 2008.

[57] C. A. Petri, „Communication with Automata,“ *RADC-TR-65-377,* sv. Volume I, 1966.

[58] W. v. d. Aalst, „Pi calculus versus Petri nets: Let us eat “humble pie” rather than further inflate the “Pi hype”,“ *BPTrends,* sv. 5(30, pp. 1-11, 2005.

[59] K. Jensen, Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use, Berlin Heidelberg: Springer, 1997.

[60] V. Janoušek, Modelování objektů Petriho sítěmi, Brno: Brno University of Technology, 1998.

[61] . Ferber, Multi-Agent System: An Introduction to Distributed Artificial Intelligence, Harlow: Addison Wesley Longman, 1999.

[62] D. Moldt a F. Wienberg, „Multi-Agent-Systems based on Coloured Petri Nets,“ *Application and Theory of Petri Nets,* 1997.

[63] L. Cabac, T. Dörges, M. Duvigneau a D. Moldt, „Agent Models for Concurrent Software Systems,“ *Proceedings of the Sixth German Conference on Multiagent System Technologies, MATES'08,* sv. 5244, 2008.

[64] F. Zbořil a R. Kočí, „Intention Structures Modelling Using Object Oriented Petri Nets,“ *Proceedings of the 7th ISDA,* pp. 33-38, 2007.

[65] Z. Mazal, Modelování uvažujících agentů Petriho sítěmi, Brno: Brno University of Technology, 2008.

[66] Z. Mazal, R. Kočí, V. Janoušek a F. Zbořil, „Modelling intelligent agents for autonomic computing in the PNagent framework," *International Journal of Autonomic Computing,* pp. 121-139, 2009.

[67] V. Janoušek a R. Kočí, „The PNtalk/SmallDEVS Framework -- Meta-level Modeling Techniques," *Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering,* pp. 16-23, 2008.

[68] F. Zbořil a F. V. Zbořil, „Simulation of Wireless Sensor Networks with Intelligent Nodes," *10th International Conference on Computer Modelling and Simulation,* 2008.

[69] F. Zbořil, V. Janoušek, R. Kočí, Z. Mazal a F. V. Zbořil, „Framework for model-based design of mullti-agent systems," *International Journal of Autonomic Computing,* pp. 140-162, 2009.

[70] F. Zbořil, „Meta-interpretation of the t-Sapi Language," *Proceedings of the Sevnth International Scientific Conference ECI 2006,* pp. 76-81, 2006.

[71] J. Horáček, „Platforma pro mobilní agenty v bezdrátových senzorových sítích," FIT VUT v Brně, Brno, 2009.

[72] J. Horáček a F. Zbořil, „WSAgeNt: A case study," *Proceedings of CSE 2010 International Scientific Conference on Computer Science and Engineering,* pp. 258-264, 2010.

[73] F. Zbořil a P. Spáčil, „Automata for Agent Low Level Language Interpretation," *Proceedings of UKSim 2009,* 2009.

[74] F. Zbořil, R. Kočí, V. Janoušek a Z. Mazal, „Reactive Planning with Weak Plan Instances," *Proceedings of 8th ISDA,* p. 6, 2008.

[75] G. D. Plotkin, „A Structural Approach to Operational Semantics," *Tech. Rep. DAIMI FN-19,* pp. 17-139, 1981.

[76] J. Horáček, F. Zbořil a P. Hanáček, „Agent Aided Routing in Wireless Sensor Networks," *Proceedings of CSE 2012,* pp. 119-126, 2012.

[77] T. Parr, The Definitive ANTLR 4 Reference, 2012.

[78] K. Holger a A. Wilig, Protocols and Architectures for Wireless Sensor Networks, John Wiley & Sons, 2006.

[79] „ZigBee Aliance," ZigBee Aliance, [Online]. Available: http://www.zigbee.org/. [Přístup získán 17 10 2012].

[80] F. Aiello, A. Carbone, G. Fortino a S. Galzarano, „Java-based Mobile Agent Platforms for Wireless Sensor Networks," *Proceedings of the International Multiconference on Computer Science and Information Technology,* pp. 165-172, 2010.

[81] U. B. Desai, B. N. Jain a S. N. Merchant, „Wireless Sensor Networks: Technology Roadmap," Indian Institute of Technology, Madras, 2010.

[82] „TinyOS," [Online]. Available: www.tinyos.net. [Přístup získán 17 10 2012].

[83] W. Munawar, M. H. Alizai, O. Landsiedel a K. Wehrle, „Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks," v *Proceedings of the IEEE International Conference on Communications*, 2010.

[84] „Contiki, The Open Source OS for the Internet of Things," [Online]. Available: www.contiki-os.org. [Přístup získán 17 10 2012].

[85] „BitCloud User Guide," Atmel.

[86] K. Römer, O. Kasten a F. Matten, „Middleware challenges for wireless sensor networks," *Mobile Computing and Communications Review, Volume 6, Number 2,* pp. 1-2, 2002.

[87] C.-L. Fok, G.-C. Roman a C. Lu, „rapid development and flexible deployment of adaptive wireless sensor network applications," *Proceedings. 25th IEEE International Conference on Digital Object Identifier,* pp. 653-662, 2005.

[88] A. Boulis, C.-C. Han, R. Shea a M. Srivastava, „SensorWare: Programming sensor networks beyond code update and querying," sv. 3ugust, č. 4, pp. 386-412, 2007.

[89] A. Boulis, C.-C. Han a M. B. Srivastava, „Design and implementation of a framework for efficient and programmable sensor networks," v *Proceedings of the 1st international conference on Mobile systems, applications and services* , New York, 2003.

[90] C. Muldoon, G. M. P. O'hare, R. W. Collier a M. J. O'grady, „Agent Factory Micro Edition: A Framework for Ambient Applications," v *LNCS VOL. 3993/2006*, 2006.

[91] „AFAPL2," 2009. [Online]. Available: http://www.agentfactory.com/index.php/AFAPL2.

[92] Aiello, Fortino, Guerrieri a Gravina, „MAPS: A Mobile Agent Platform for WSNs Based on Java Sun Spots," v *Proceedings of ATSM*, 2009.

[93] D. Tapia, R. G. O. Alonso a J. Corchado, „HERA: Hardware-Embedded Reactive Agents Platform," v *Highliths in PAAMS*, Heideberg, 2011.

[94] D. I. Tapia, R. S. Alonso, F. D. Z. C. Prieta a e. al., „SYLPH: An Ambient Intelligence Based Platform for Integrating Heterogeneous Wireless Sensor Networks," *Journal of Ambient Computing Intelligence,* 2011.

[95] K. Henricksen a R. Robinson, „A Survey of Middleware for Sensor Networks: State of the Art and Future Directions," v *MidSens'06*, Melbourne, 2006.

[96] T. Liu a M. Martonosi, „Impala: a middleware system for managing autonomic, parallel sensor systems," v *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, 2003.

[97] Y. Kwon, S. Sundresh, K. Mechitov a G. Agha, „ActorNet: An Actor Platform for Wireless Sensor Networks," v *Proceedings of 5th AAMAS*, 2006.

[98] D. Georgoulas a K. Blow, „In-Motes: Intelligent Agent-Based Middleware for Wireless Sensor Networks," v *Proceedings of the 5th WSEAS international conference on Applications of electrical engineering*, 2006.

[99] M. R.-A. J. A. C. J. Vinyals, „A Survey on Sensor Networks from a Multiagent perspective," *The Computer Journal,* Sv. %1 z %2Volume 54, Issue 3, 2010.

[100] M.-M. Wang, J.-N. Cao, J. Li a S. Das, „Middleware for Wireless Sensor Networks: A Survey," *Journal of Computer Science and Technology,* pp. 305-326, May 2008.

[101] A. Morris, P. Giorgini a S. Abdel-Naby, „Simulating BDI-based Wireless Sensor Networks," v *wi-iat, vol. 2*, 2009.

[102] Y. Chen a A. Terzis, „On the Mechanism and Effects of Calibrating RSSI Measurements for 802.15.4 Radios," *LNCS vol. 5970/2010,* pp. 256-271, 2010.

[103] W. M. Farmer, J. D. Guttman a V. Swarup, „Security for Mobile Agents: Issues and Requirements," v *Security for Mobile Agents: Issues and Requirements*, 1996.

[104] G. C. Necula a P. Lee, „Safe, Untrusted Agents using Proof-Carrying Code," v *Mobile Agents and Security*, London, 1998.

[105] N. Sastry a D. Wagner, „Security Considerations for IEEE 802.15.4 Networks," v *WiSe '04: Proceedings of the 2004 ACM workshop on Wireless security* , 2004.

[106] M. Hefeeda a B. Bhagava, „On Mobile Code Security," *CERIAS TR 2001-46,* 2001.

[107] J. Samek and F. Zboril jr., "Algorithmic Evaluation of Trust in Multilevel Model," in *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation*, Praha, 2010.

[108] J. Samek and F. Zboril ml., "Hierarchical Model of Trust in Context," in *Networked Digital Technologies*, Heidelberg, 2010.

[109] F. Zbořil, J. Horáček, P. Hanáček a M. Drahanský, „Security in Wireless Sensor Networks with Mobile Codes," v *Threats, Countermeasures, and Advances in Applied Information Security*, Hershey, IGI Global, 2012, pp. 411-425.

[110] P. Pecho, F. Zbořil, M. Drahanský a P. Hanáček, „Agent Platform for Wireless Sensor Network with Support for Cryptographic Protocols,“ *Journal of Universal Computer Science,* pp. 992-1006, 2009.

[111] J. Horáček a F. Zbořil, „Secured Agent Platform for Wireless Sensor Networks,“ *Lecture Notes in Computer Science,* sv. 4, pp. 476-485, 2011.

[112] R. Kalmár, „Optimalizace překladu agentních jazyků různé úrovně abstrakce,“ FIT, VUT v Brně, Brno, 2012.

[113] J. Žák, F. Zbořil, R. Kočí a V. Janoušek, „Connecting Jade with PN agent,“ *Proceedings of Seventh EUROSIM Congress on Modelling and Simulation,* 2010.

[114] F. Zbořil, P. Spáčil a J. Horáček, „Intelligent Agent Platform and Control Language for Wireless Sensor Networks,“ *Proceedings of 3rd EMS,* 2009.