VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

DOKTORSKÁ DISERTAČNÍ PRÁCE

srpen 1996

Ing. Petr Hanáček

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Model pro vyjádření synchronizačních a komunikačních mechanismů

DOKTORSKÁ DISERTAČNÍ PRÁCE

Doc. RNDr. Milan Češka, CSc. Ing. Petr Hanáček

školitel doktorand

srpen 1996

Předkládaná doktorská práce vznikla na Ústavu informatiky a výpočetní techniky VUT v Brně jako výsledek intenzívního studia problematiky týkající se paralelního a distribuovaného programování. Práce vychází rovněž ze zkušeností, získaných výukou předmětu Paralelní a distribuované algoritmy.

Je mou povinností poděkovat všem pracovníkům ústavu za mnoho cenných rad a nezištné předávání svých zkušeností. Za všechny jmenuji Prof. Ing. Jana Blatného, DrSc, Prof. Ing. Václava Dvořáka, DrSc. a Doc. Ing. Zdenu Rábovou, CSc.

Zvláště pak děkuji svému školiteli Doc. RNDr. Milanu Češkovi, CSc., za vytvoření potřebných podmínek, za citlivé vedení a podporu v práci.

Prohlašuji, že jsem práci vypracoval samostatně a že jsem použil jen citovanou literaturu.

Petr Hanáček

Obsah

1. ÚVOD 8

1.1. Cíl práce 8

2. SEZNÁMENÍ S PROBLEMATIKOU 10

2.1 Paralelní počítačové architektury 10
2.1.1 Paralelní systémy MIMD se sdílenou pamětí 11
2.1.2 Systémy MIMD s distribuovanou pamětí 13

2.2 Interakce procesů se sdílenou pamětí 14
2.2.1 Sdílené proměnné 16
2.2.2 Semafory 18
2.2.3 Bariérová synchronizace 19
2.2.4 Podmíněné kritické oblasti 20
2.2.5 Monitory a podmínkové proměnné 21

2.3 Interakce procesů pomocí předávání zpráv 23
2.3.1 Asynchronní a synchronní předávání zpráv 24
2.3.2 RPC a rendezvous 25
2.3.3 Budoucí proměnné a Cboxy 26

2.4 Nástěnka v jazyce Linda 26

2.5 Klasifikace interakčních mechanismů 28

3. MODEL PRO VYJÁDŘENÍ INTERAKČNÍCH MECHANISMŮ 32

3.1 Použitý výpočetní model 32

3.2 Sdílený datový prostor 33

3.4 Interakční bod 35

3.5 Příklady implementace interakčních mechanismů 39

3.6 Jednoduché interakční mechanismy 40

3.7 Složené interakční mechanismy 42

3.8 Nalezení složených interakčních mechanismů 44

3.9 Interakční mechanismy s více interakčními body 45

4. OPTIMALIZACE IMPLEMENTACE INTERAKČNÍHO BODU 46

4.1 Výpočet délek front interakčního bodu 46

4.2 Použitá notace a operace 49
4.2.2 Sekvence a její vlastnosti 50

4.3 Skládání sekvencí **53**
4.3.1 Sekvenční provedení 53
4.3.2 Paralelní provedení 55
4.3.3 Podmíněné provedení 57
4.3.4 Alternativní provedení 58
4.3.5 Počítaná iterace 59
4.3.6 Nepočítaná iterace 60

4.4 Nepočítaná iterace s nekonečnou vstupní frontou **62**

4.5 Určení maximálních délek front v programu **63**

5. MOŽNÉ APLIKACE MODELU **67**

6. ZÁVĚR **69**

7. POUŽITÁ LITERATURA **70**

PŘÍLOHA A - PŘÍKLAD VÝPOČTU MAXIMÁLNÍCH DÉLEK FRONT **73**

PŘÍLOHA B - VYBRANÉ PUBLIKACE AUTORA **77**

Abstrakt

Tato práce se zabývá jedním z problémů paralelního a distribuovaného výpočetního prostředí, kterým je interakce mezi procesy. Interakce mezi procesy zahrnuje vzájemnou synchronizaci procesů, komunikaci mezi procesy a sdílení dat mezi procesy. Interakce mezi procesy je implementována v paralelních a distribuovaných operačních systémech a programovacích jazycích mnoha různými mechanismy. Jako příklad těchto mechanismů je možno uvést semafory, monitory, sdílené proměnné, synchronní a asynchronní kanál, synchronní a asynchronní přihrádka, volání vzdálených procedur, rendezvous a mnohé jiné. Každý z těchto mechanismů je založen na jiném principu, má jiné vlastnosti, syntaxi i sémantiku.

Cílem této práce je prezentovat formální model, vhodný pro jednotný popis všech běžně používaných prostředků pro interakci mezi paralelními a distribuovanými procesy. Jako základ pro tento formální model byla použita datová abstrakce, nazvaná *sdílený datový prostor*. Tento sdílený datový prostor obsahuje objekty, zvané *interakční body*. Pomocí těchto interakčních bodů a základních operací, které jsou nad nimi definovány, pak vyjádříme jednotlivé interakční mechanismy.

Nejdříve uvedeme definici modelu pro *jednoduché interakční mechanismy*, které je možno modelovat jedním interakčním bodem. Mezi ně patří zejména sdílená proměnná, semafor, jednoduchý monitor, asynchronní kanál a asynchronní přihrádka. Pak bude definován model pro *složené interakční mechanismy*, pro jejichž vyjádření je třeba dvou a více interakčních bodů. Příklady těchto mechanismů jsou synchronní kanál, synchronní přihrádka, volání vzdálených procedur, rendezvous a bariérová synchronizace.

Poslední část práce se věnuje problematice optimalizace implementace interakčních mechanismů, vytvořených na základě výše definovaných modelů.

Abstract

The purpose of this work is to present a formal model for description of the commonly used primitives for interprocess communication, process synchronization, and data sharing. The attempt is made to uniformly describe the most of common paradigms used in the programming languages for parallel and distributed computing.

The shared persistent data space model is used as a tool for describing all of the common communication and synchronization primitives, include semaphore, monitor, shared variable, synchronous and asynchronous channel, synchronous and asynchronous mailbox, remote procedure call, rendezvous, and so on. Because of unified approach to the synchronization and communication we will use the term interaction for both these kinds of primitives. As the basic interaction object is used so called *interaction point*.

In the first part of the work the computational model is described. The behavior and the properties of the persistent shared data space are formally expressed. Two basic operations *inspect* and *modify* are defined. The atomicity of the operations, the lifetime of the process and the persistence of the data space is defined and discussed as well. In the second part of the work the basic interaction object, *interaction point*, is defined. Interaction point is the object in the persistent shared data space. It consists of two queues: *entry queue* that holds the values or events stored in the interaction point and the *waiting queue* that contains the processes waiting for the interaction point.

Two basic, Linda-like operations with this object *in* and *out* are defined. The *out* operation is non-blocking and puts the data into the interaction point. The *in* operation is blocking and retrieves data from the interaction point. The statistical properties of the interaction point are defined and the model is extended to the *parametrized interaction point* model. This model is parametrized by three basic properties of the interaction primitive - the transmitted data size D, the maximal entry queue length E, and the maximal waiting queue length W.

Using this parametrized interaction point is in the following part of the work defined the class of the *simple interaction primitives*. Simple interaction primitives are those primitives that can be modeled by single interaction point. These include shared variable, semaphore, monitor, asynchronous channel, and asynchronous mailbox.

In the next section is defined the class of the *compound interaction primitives*. Compound interaction primitives are those primitives that can be modeled using two or more interaction points. This class include synchronous channel, synchronous mailbox, remote procedure call, rendezvous, and barrier synchronization. It also includes whole group of client-server-like primitives and a some primitives with no name. The interesting conclusion is that the hierarchy of both simple and compound primitives gives a lattice diagram.

The last part of the work deals with the implementation features of the primitives created from the interaction points. It is discussed how to implement the interaction point effectively using the information extracted from the code of the parallel processes. The optimization is concerned in problem how to determine the maximal length of the queues. The knowledge of the maximal lengths of the queues allows us in many cases to avoid one or both the queues. The goal of the optimization is to make the modeled interaction primitives at least as efficient as the original primitives were.

1. ÚVOD

Problémy paralelního a distribuovaného výpočetního prostředí patří v současné době k aktuálním směrům v informatice. Paralelní a distribuované výpočetní prostředí se liší od běžného sekvenčního prostředí především tím, že v sekvenčním prostředí se pro výpočty používá pouze jediný procesor, zatímco v paralelním prostředí je problém rozdělen na několik podproblémů, které jsou řešeny současně (paralelně) na několika běžících procesorech.

V konvenčním sekvenčním počítačovém prostředí byla a stále je věnována velká energie a úsilí zrychlování výpočtů. Rychlost procesorů a hustota integrace se stále pohybuje na jejich technologické hranici, avšak požadavky na výpočetní výkon stoupají stále rychleji. Některé aplikace, jako například předpovědi počasí nebo některé simulační experimenty, vyžadují větší a větší výpočetní výkon. Čím větší výpočetní výkon je k dispozici, tím lepší výsledky mohou být dosaženy v kratším čase. Existují také některé teoretické i praktické problémy, jejichž řešení je stále za hranicí dostupného výpočetního výkonu. Na druhé straně je pravdou, že vyšší výpočetní výkon, který má člověk k dispozici, přináší další a další problémy, které byly dříve zcela neřešitelné a nyní jsou "téměř" řešitelné. Výsledkem je, že stoupající výpočetní výkon počítačů vyvolává poptávku po jeho dalším zvyšování.

V paralelním výpočetním prostředí je řešený problém rozdělen mezi množství procesorů, které pracují paralelně na jednotlivých podproblémech. Každý procesor řeší pouze část problému a řešení celého problému získáme spojením řešení jednotlivých podproblémů. Jelikož jsou jednotlivé podproblémy řešeny současně, může být celkové řešení provedeno ve zlomku času, který by si vyžadoval sekvenční výpočet.

Jelikož v paralelním výpočetním prostředí je třeba problém rozdělit na podproblémy, které obvykle nejsou zcela nezávislé a výsledná řešení je třeba opět zkombinovat, vyskytují se zde značné požadavky na *komunikaci* mezi jednotlivými procesory a na jejich vzájemnou *synchronizaci*. U skutečných paralelních výpočetních zařízení je vzájemná komunikace a synchronizace procesorů jedním z nejzávažnějších problémů a často je také z hlediska výkonu úzkým místem celého systému.

Komunikace mezi procesory, synchronizace procesorů a (dosud nezmíněné) sdílení dat mezi procesory jsou pravděpodobně jedny z nejdůležitějších problémů v paralelním počítání. Náplní této práce je problematika mechanismů pro komunikaci, synchronizaci a sdílení dat. V následujícím textu budeme pojmy synchronizace, komunikace a sdílení dat nazývat pojmem jediným - interakce.

1.1. Cíl práce

V celé historii paralelního programování bylo vyvinuto mnoho interakčních mechanismů v mnoha programovacích jazycích. V současné době jsou vyvíjeny nové mechanismy (například takové, které jsou vhodné pro zabudování do objektově orientovaných paralelních jazyků). A v budoucnu budou pravděpodobně vyvinuty další mechanismy. Některé z těchto mechanismů se od sebe liší svou sémantikou, některé se však liší pouze svou syntaxí a některé se liší pouze svým názvem. Časem se také objevila

tradiční kritéria pro kategorizaci interakčních mechanismů. Zmíníme se o nich podrobněji v jedné z následujících kapitol.

Postupem času se ukazuje, že práce v oblasti interakce mezi paralelními procesy je spíše extenzivní než intenzivní. Jsou stále vytvářeny nové mechanismy, avšak je zde pouze několik ojedinělých pokusů o vytvoření nějakého společného modelu, vhodného pro všechny typy interakčních mechanismů. A pokud se takové modely objevily, zabývaly se převážně syntaktickou stránkou popisovaných mechanismů a nikoli stránkou sémantickou.

Cílem této práce je prezentovat společný model, vhodný pro popis všech běžně používaných mechanismů, používaných pro komunikaci, synchronizaci a sdílení dat. Práce se pokouší jednotně popsat většinu běžných paradigmat, používaných v paralelních programovacích jazycích a v jazycích pro distribuované výpočty. Jako základ popisu je použit model sdíleného datového prostoru, založeného na výpočetním modelu asynchronní PRAM (pojem bude vysvětlen později). Ten bude použit pro popis všech běžných interakčních mechanismů, jako je například semafor, monitor, sdílená proměnná, synchronní a asynchronní kanál, synchronní a asynchronní přihrádka, volání vzdálených procedur, rendezvous a podobně.

Další část této práce se zabývá problematikou implementace interakčních mechanismů, popsaných výše zmíněným modelem. Vzhledem k tomu, že použitý model je velmi obecný, byla by naivní implementace (naivní implementací rozumíme přímočarou implementaci bez jakékoli optimalizace) mechanismů, popsaných tímto modelem, velmi obtížná a v některých případech až nemožná. Proto jsou ve druhé části práce rozpracovány metody optimalizace, které mohou pomoci při implementaci mechanismů, popsaných tímto modelem.

2. SEZNÁMENÍ S PROBLEMATIKOU

V této kapitole je uveden krátký přehled paralelních a distribuovaných architektur, které budou naším předmětem zájmu. Pak následuje popis všech běžně používaných interakčních mechanismů, kterými se práce zabývá. Zvlášť se zmíníme o jednom interakčním mechanismu, o nástěnce v jazyce Linda, který má ve výkladu významnou úlohu. Nakonec se budeme zabývat společnými vlastnostmi popsaných interakčních mechanismů a tím naznačíme některé možnosti vytvoření společného modelu pro všechny tyto mechanismy.

Nyní několik slov k použité terminologii. Už dříve jsme uvedli, že pojmy *synchronizace*, *komunikace* a *sdílení dat* budeme nazývat společným pojmem *interakce*. Při popisu paralelních a distribuovaných architektur se někdy nevyhneme tomu, abychom použili názvy konkrétních interakčních mechanismů, přestože tyto názvy nebyly ještě vysvětleny a definovány. Jejich vysvětlení je uvedeno později, v podkapitole "Nejpoužívanější interakční mechanismy". Další terminologický problém je spojen s používáním pojmů *proces* a *procesor*, v případech jako je "*komunikace mezi procesy*" a "*komunikace mezi procesory*". Interakční mechanismy mohou být totiž použity jak na úrovni operačního systému (zde se bude jednat o interakci mezi procesory), tak na úrovni aplikační, kde se jedná o interakci mezi procesy. Jelikož se v této práci zabýváme interakčními mechanismy obecně, budeme chápat, že interakce může probíhat (až na výjimky) jak mezi procesy, tak i mezi procesory.

Poslední terminologický problém se týká pojmů *paralelní* a *distribuovaný*. V odborné literatuře různých autorů tyto pojmy mívají někdy různý význam. Někteří autoři (např. [Sno92]) chápou pod pojmem *paralelní systém* pouze systém se sdílenou pamětí (viz následující kapitola) a pod pojmem *distribuovaný systém* pouze systém s distribuovanou pamětí. Jiní autoři (např. [Wil90]) používají pojem *paralelní systém* pro všechny počítačové systémy s více procesory (to znamená, že tímto pojmem označují jak paralelní, tak i distribuované systémy z předchozí definice). Vzhledem k problematičnosti první definice (hranice mezi paralelním a distribuovaným systémem není vždy ostrá) se přikloníme k definici druhé. Pokud bude někde v textu použit kromě pojmu paralelní systém i pojem distribuovaný systém, je to pouze pro doplnění, neboť předpokládáme, že pojem paralelní systém pokrývá všechny počítačové systémy s několika paralelně spolupracujícími procesory.

2.1 Paralelní počítačové architektury

Přestože bylo provedeno mnoho pokusů o oddělení paralelního programovacího jazyka od počítačové architektury, na níž je tento jazyk implementován (např. *Mentat* [Gri91]) nebo *Linda* [Gel85], je většina interakčních mechanismů silně závislá na některé počítačové architektuře. Abychom tento problém poněkud osvětlili, ukážeme v následujících odstavcích některé nejrozšířenější paralelní architektury a spojení mezi těmito architekturami a mezi interakčními mechanismy, které jsou pro tyto architektury přirozené.

Pokud pro klasifikaci paralelních architektur použijeme Flynnovu klasifikaci [Fly72], můžeme paralelní počítačové systémy rozdělit podle počtu řídicích toků a podle počtu datových toků na čtyři skupiny:

- Architektury s jediným řídicím tokem a s jediným datovým tokem (single instruction and single data stream - *SISD*), neboli sekvenční architektury.
- Architektury s jediným řídicím tokem a s několika datovými toky (single instruction and multiple data stream - *SIMD*), mezi něž patří například vektorové a maticové procesory.
- Architektury s několika řídicími toky a s několika datovými toky (multiple instruction and multiple data stream - *MIMD*), neboli běžné multiprocesorové systémy.

Protože se v této práci chceme zabývat mechanismy pro synchronizaci a komunikaci, budeme se zabývat pouze architekturami MIMD. Tyto architektury mají několik řídicích toků, což vede k potřebě synchronizace a mají i několik datových toků, což vede k potřebě komunikace a sdílení dat. Architektury MIMD se dělí na dvě skupiny. První skupina má sdílenou paměť, která je přístupná všem procesorům. Příslušníci této skupiny se často nazývají *paralelní počítače*, *multiprocesory* nebo *systémy se sdílenou pamětí*. Druhá skupina nemá k dispozici sdílenou paměť a její příslušníci se nazývají *distribuované systémy*, *multicomputery* nebo *systémy s distribuovanou pamětí*.

Systémy se sdílenou pamětí mají jediný, globální adresový prostor, který je viditelný pro všechny procesory v systému. Každý z těchto procesorů může přečíst nebo zapsat slovo z/do paměti tak, že pouze přesune data z/do této paměti. Interakce mezi procesy se provádí pomocí této sdílené paměti. Systémy s distribuovanou pamětí nemají k dispozici sdílenou paměť a procesy v nich musí komunikovat předáváním zpráv.

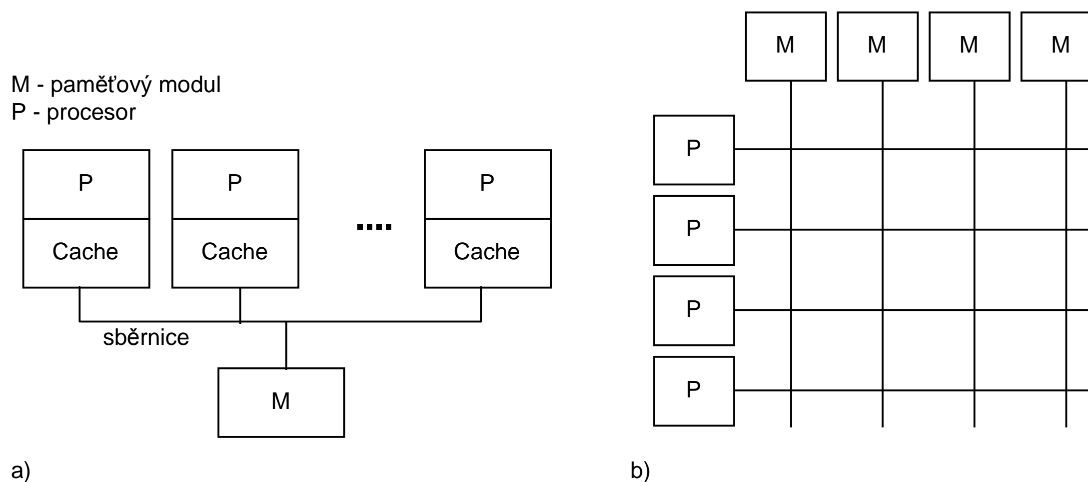
2.1.1 Paralelní systémy MIMD se sdílenou pamětí

Klíčovým požadavkem, kladeným na každý systém MIMD se sdílenou pamětí, je takzvaná *paměťová koherence*: Pokud kterýkoli procesor zapíše hodnotu *v* do paměti na adresu *m*, každý jiný procesor, který následně přečte slovo na adrese *m*, musí přečíst hodnotu *v*, bez ohledu na to, jak rychle čtení následuje po zápisu.

Existují dva základní způsoby, jak vytvořit počítačový systém se sdílenou pamětí ([Tan92]). První cestou je připojit všechny procesory na společnou sběrnici spolu s paměťovým modulem. Pokud chce procesor přečíst nebo zapsat data, provede normální paměťový požadavek přes sběrnici. Jelikož zde existuje pouze jediný paměťový modul a data v paměti existují pouze v jediném exempláři (nikde není uložena kopie těchto dat), paměť je vždy koherentní. Aby se snížilo zatížení společné sběrnice, jsou procesory často vybaveny rychlou vyrovnávací pamětí (cache), ve které jsou uloženy kopie těch paměťových slov, ke kterým bylo v poslední době přistupováno. Tato rychlá vyrovnávací paměť (RVP) musí být udržována v konzistentním stavu pomocí speciálního technického vybavení, které sleduje provoz na sběrnici a zneplatňuje ty položky RVP, které jsou modifikovány jiným procesorem.

Na obrázku 1a je znázorněna struktura paralelního systému se sdílenou pamětí. Tato architektura má špatnou rozšiřovatelnost, protože má pouze jednu společnou sběrnici a pouze jediný paměťový modul. Ty se při zvyšování počtu procesorů stávají úzkým místem systému. Zvýšení výpočetního výkonu pak neodpovídá přírůstku počtu procesorů.

Částečným řešením tohoto problému je zvýšení počtu a kapacit rychlých vyrovnávacích pamětí. Některé architektury dokonce ukládají všechna data pouze v rychlých vyrovnávacích pamětech a paměťový model u nich neexistuje (takzvané "allcache" architektury [Cas93]).



Obr. 1: Architektura se sdílenou pamětí
a) se společnou sběrnicí b) s propojovací sítí

Druhou cestou je vytvořit systém se sdílenou pamětí pomocí některé ze známých propojovacích sítí, například s křížovým propojovačem, jak je naznačeno na obrázku 1b. U tohoto způsobu je každý z n paměťových modulů spojen s každým z m procesorů pomocí matice elektronických spínačů. Pokud je spínač ij sepnut, pak je procesor i spojen s paměťovým modulem j a může do něj zapisovat nebo z něj číst data. Výhodou této architektury je to, že zátěž je rozložena mezi několik sběrnic a paměťových modulů. Tím se odstraní úzké místo, které bylo u předchozího způsobu. Tento způsob má však také některé nevýhody. Problematická je například skutečnost, že pro propojení n procesorů s n paměťovými moduly je potřeba n^2 spínačů. Pokud je n velké (například 1024 procesorů a 1024 paměťových modulů), křížový propojovač se stává příliš drahým a špatně řiditelným.

Místo křížového propojovače je také možno použít jinou propojovací síť, například síť omega. Síť omega je speciální propojovací síť, která spojuje procesory a paměťové moduly. Pokud chce procesor číst slovo z paměti, vyšle paket s požadavkem přes propojovací síť odpovídajícímu paměťovému modulu, který mu pošle zpět paket s odpovědí. Nevýhodou této architektury je větší zpoždění při přístupu k paměti a stále poměrně velký počet potřebných spínačů ($n\log_2 n$).

Interakční mechanismy, používané na architekturách se sdílenou pamětí, jsou přímočaré. Jelikož všechny procesory běží v jednom sdíleném adresovém prostoru, mohou snadno sdílet proměnné a datové struktury. Proto základním komunikačním mechanismem je *sdílená proměnná*. Pokud jeden procesor zapíše hodnotu do proměnné a jiný ji ihned přečte, vždy dostane hodnotu, která byla právě zapsána (paměťová koherence je zajištěna).

Aby paralelně běžící procesy mohly spolupracovat, musí synchronizovat svou činnost. Pokud například jeden proces modifikuje datovou strukturu seznam, je důležité, aby se během této modifikace žádný jiný proces nepokoušel ani o čtení ani o modifikaci

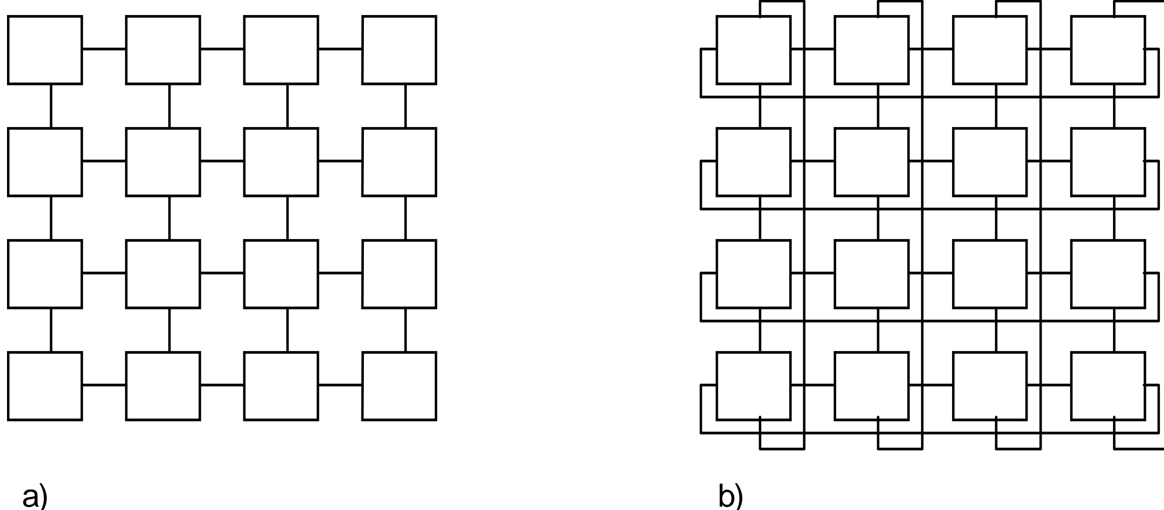
tohoto seznamu. Mechanismy, které se v těchto architekturách používají pro synchronizaci, jsou obvykle *semafory*, *monitory* a *bariéry*.

2.1.2 Systémy MIMD s distribuovanou pamětí

Na rozdíl od systémů se sdílenou pamětí, které, jak už říká jejich název, sdílejí paměť, každý procesor v systému MIMD s distribuovanou pamětí má svou vlastní lokální paměť, se kterou může pracovat pouze tento procesor. Tento rozdíl vede ke značně rozdílné architektuře jak z pohledu technického vybavení, tak i z pohledu programového vybavení.

Aby byla i u této architektury možná interakce mezi procesory, procesory musí být spojeny nějakou komunikační sítí. Těchto komunikačních sítí existuje velké množství. Obrázek 2 ukazuje dva příklady takové komunikační sítě: ortogonální mřížku a hyperkostku.

Architektura mřížka (Obr. 2a) je nejvhodnější pro řešení dvoudimenzionálních problémů, jako jsou například maticové operace nebo zpracování obrazu. Hyperkostka je vlastně k-rozměrná krychle, v jejíchž rozích jsou umístěny jednotlivé procesory a jejíž hrany jsou komunikační kanály. Na Obr. 2b je uveden příklad čtyřrozměrné hyperkostky. Příznivou vlastností hyperkostky je to, že počet kanálů, vedoucích z každého procesoru, roste logaritmicky s počtem procesorů, což dovoluje udržet celkový počet kanálů na rozumné úrovni. Nevýhodou ovšem je to, že komunikační zpoždění (tj. počet procesorů, kterými musí zpráva v nejhorším případě projít) roste opět logaritmicky s počtem procesorů. V případě mřížky roste komunikační zpoždění s odmocninou počtu procesorů, což je horší.



Obr. 2: Dva systémy MIMD s distribuovanou pamětí:
a) mřížka b) hyperkostka.

Protože systémy s distribuovanou pamětí už z principu nemají sdílenou paměť, je komunikace mezi jednotlivými procesory možná pouze *zasíláním zpráv* pomocí

komunikačních kanálů. Nejjednodušší komunikační mechanismus, který reprezentuje zasílání zpráv, používá dva příkazy: *send* a *receive*. Komunikační mechanismy, které pracují se zasíláním zpráv, mají mnoho variant, jako je například synchronní a asynchronní zasílání zpráv nebo blokující a neblokující zasílání zpráv. Jiné mechanismy se pokoušejí o skrytí samotného zasílání zpráv do nějaké vyšší jazykové struktury. Příkladem je volání vzdálených procedur (RPC, Remote Procedure Call), které skrývá zasílání zpráv do volání procedur.

2.2 Interakce procesů se sdílenou pamětí

V následujících kapitolách je uveden přehled většiny běžně používaných interakčních mechanismů, které jsou známy z literatury. Bylo by možné tuto kapitolu vynechat a přehled interakčních prostředků uvést odkazy na literaturu. Tento postup by však jednak ztížil čtenáři orientaci v práci a pak by mohl vést ke komplikacím. Tyto komplikace by především vznikly z toho, že v různých pramenech se některé interakční mechanismy nazývají rozdílně a naopak někteří autoři si pod stejným názvem interakčního mechanismu představují rozdílné mechanismy. Uvedeme zde tedy úplný přehled všech mechanismů, o kterých se budeme v práci zmiňovat. Musíme si však být vědomi toho, že se v jiné literatuře můžeme setkat s mírně odlišnou definicí těchto mechanismů.

Paralelní program se skládá z několika paralelně běžících a kooperujících procesů. Každý z těchto procesů provádí sekvenční činnost. V této kapitole budeme předpokládat, že procesy interagují tím, že mohou číst a zapisovat do/ze sdílených proměnných.

Pokud je vykonáván sekvenční program, existuje pouze jeden *řídicí tok*: programový čítač začíná první atomickou akcí (obvykle instrukcí nebo její částí) procesu a postupně se vykonávají další atomické akce procesu. Vykonávání paralelního programu znamená existenci několika řídicích toků - pro každý proces existuje jeden řídicí tok. Provádění paralelního programu je tedy možno chápat jako prokládání atomických akcí, prováděných jeho jednotlivými procesy. Pokud procesy spolu interagují, je pravděpodobné, že ne všechny možné varianty prokládání atomických akcí budou přijatelné. Synchronizace má za úkol zabránit těm variantám prokládání, které jsou nežádoucí. Synchronizace má typicky dva možné účely. Prvním je to, že umožňuje seskupit několik atomických akcí (instrukcí) do bloku, který se chová jako celek atomicky - tento problém se nazývá *vzájemné vyloučení* (mutual exclusion) nebo také *kritická sekce* (critical section). Druhým účelem je pozastavit vykonávání procesu až do doby, kdy je splněna jistá podmínka - jde o problém *synchronizace na podmínce*. V následujících odstavcích se budeme převážně zabývat problémem vzájemného vyloučení a atomických akcí.

Atomická akce je takový přechod z jednoho stavu do jiného, který je dále nedělitelný. Pokud ve skutečné implementaci atomické akce existují nějaké mezistavy, tyto mezistavy nesmí být viditelné pro ostatní procesy. Otázkou je, které základní operace počítačové architektury můžeme považovat za atomické. V sekvenčním programování je přiřazovací příkaz považován za atomický, protože nejsou viditelné žádné jeho mezistavy. V paralelním programování to obecně neplatí, protože přiřazovací příkaz je obvykle implementován jako posloupnost několika akcí. Například v následujícím programu pracují dva procesy paralelně s několika sdílenými proměnnými (předpokládejme, že počáteční hodnoty proměnných y a z jsou nulové):

```
Proces 1            Proces 2
x := y + z;         y := 1;
                    z := 2;
```

Pokud je operace *x := y + z* implementována tak, že se nejdříve načte do registru *y* a pak se k němu přičte *z*, může být výsledná hodnota proměnné *x* rovna 0, 1, 2 nebo 3.

V následujícím textu budeme muset zavést jisté předpoklady o počítačové architektuře z hlediska atomičnosti prováděných operací. Zavedeme proto předpoklady, které jsou rozumné a jsou splňovány většinou typických architektur:

- Čtení a zápis základních a ordinálních typů (např. integer) je prováděno jako atomická akce.
- Při práci s výrazy jsou hodnoty načteny do registrů, je s nimi manipulováno v těchto registrech a pak jsou zapsány zpět do paměti.
- Každý proces má svou vlastní sadu registrů.
- Každý mezistav při vyhodnocování složitějších výrazů je uložen pouze v registrech nebo v paměti, která je pro proces privátní.

Pokud architektura splňuje tyto předpoklady, znamená to následující skutečnost: Pokud se výraz *e* neodkazuje na proměnnou, která je jiným procesem modifikována, vyhodnocení výrazu *e* bude atomické, i když se skládá z několika akcí.

Definice synchronizace

Na základě výše uvedených předpokladů však nejsme schopni vytvořit rozumný paralelní program, protože prakticky v každém paralelním programu máme potřebu zajistit atomičnost posloupnosti několika příkazů. Proto potřebujeme mít k dispozici prostředek, který nám dovolí sdružit skupinu základních atomických akcí (typicky instrukcí) procesu v jednu větší atomickou akci, která se jeví jako nedělitelná - potřebujeme synchronizaci.

Pro formální zápis synchronizace použijeme tzv. notace s příkazem **await** (který je popsán např. v [And91]). V této notaci se atomičnost vyjadřuje pomocí dvojitých úhlových závorek << a >>. Například zápis <<*e*>> znamená, že výraz *e* bude vyhodnocen atomicky. Celý zápis notace příkazu **await** má tvar:

```
<<await B -> S>>
```

Booleovský výraz *B* definuje čekací podmínku, *S* je posloupnost sekvenčních příkazů, která zaručeně skončí (například posloupnost přiřazovacích příkazů). Příkaz **await** je umístěn v úhlových závorkách, protože je prováděn atomicky. Dále je zaručeno, že v okamžiku, kdy začne vykonávání příkazů *S*, je podmínka *B* splněna a platí, že žádný mezistav provádění příkazů *S* není viditelný pro ostatní procesy. Například příkaz

```
<<await s>0 -> s := s - 1>>
```

čeká, dokud není *s* kladné a pak dekrementuje *s*. Je zaručeno, že hodnota proměnné *s* je před dekrementací kladná.

Příkaz **await** je velmi mocný prostředek pro vyjádření synchronizace. Je proto vhodný pro popis některých synchronizačních mechanismů a problémů. Proto jej budeme používat v úvodních kapitolách této práce. Je však třeba si uvědomit, že právě proto, že příkaz **await** je velmi mocný, je prakticky efektivně neimplementovatelný. Pouze některé speciální případy příkazu **await** se dají implementovat efektivně.

Obecný tvar příkazu **await** definuje jak vzájemnou výlučnost, tak i synchronizaci na podmínce. Pokud chceme vyjádřit pouze vzájemnou výlučnost, je možné příkaz **await** zkrátit do tvaru:

```
<<S>>
```

Například následující příkaz atomicky inkrementuje proměnné *x* a *y*. Pokud chceme vyjádřit pouze synchronizaci na podmínce, můžeme zkrátit příkaz **await** do tvaru

```
<<await B>>
```

a například příkaz

```
<<await count>0>>
```

pozastaví proces, dokud nebude hodnota proměnné *count* kladná.

2.2.1 Sdílené proměnné

Sdílené proměnné (někdy nazývané také *sdílená paměť*) jsou nejjednodušším prostředkem pro interakci mezi procesy. Základní výhodou komunikace pomocí sdílených proměnných oproti komunikaci předáváním zpráv je to, že pro programátora je tento model mnohem jednodušší a snadněji srozumitelný. Způsob přístupu ke sdíleným proměnným je konzistentní s tím, jak běžné sekvenční programy přistupují ke svým datům a dovoluje snadnější přechod od sekvenční verze programu k jeho paralelní verzi.

Sdílená proměnná je proměnná, která je viditelná pro několik procesů současně. Každý z těchto procesů může hodnotu této sdílené proměnné číst a každý do ní může zapisovat. Komunikační model se sdílenými proměnnými používá dvě základní operace pro přístup k datům:

```
value := read (address)
write (address, value)
```

Operace *read* vrací datovou položku, umístěnou na adrese *address,* a operace *write* nastavuje obsah datové položky, umístěné na adrese *address* na hodnotu *value*.

Sdílené proměnné mají jednu nežádoucí vlastnost - žádným způsobem nezajišťují synchronizaci procesů. Nyní si ukážeme, jakým způsobem je možno ošetřit kritickou sekci, pokud máme k dispozici pouze sdílené proměnné.

Kritická sekce

Problém kritické sekce spočívá v tom, že *n* procesů opakovaně provádí jistou kritickou část svého kódu (tj. kritickou sekci) a pak část kódu, která není kritická. Požaduje

se, aby provedení kritické sekce bylo atomické. Před kritickou sekcí je posloupnost instrukcí, zvaná *vstupní protokol* a za ní je posloupnost instrukcí, zvaná *výstupní protokol*.

```
while true do
 begin
 vstupní protokol
 kritická sekce
 výstupní protokol
 nekritická sekce
 end;
```

Každá kritická sekce obsahuje posloupnost instrukcí, jejichž prostřednictvím se přistupuje ke sdílenému objektu. Každá nekritická sekce obsahuje skupinu jakýchkoli jiných příkazů. Předpokládáme, že proces, který vstoupí do kritické sekce, z ní také v konečném čase vystoupí - to znamená, že proces se může ukončit pouze mimo kritickou sekci.

Triviálním řešením je uzavřít kritickou sekci do úhlových závorek takto:

```
while true do
 begin
 <<kritická sekce>>
 nekritická sekce
 end;
```

Toto způsob však není řešením, protože neříká, jak implementovat výše použité úhlové závorky. Pro řešení kritické sekce je důležitá vlastnost vzájemné výlučnosti - pouze jeden proces může být v kritické sekci. Proto potřebujeme sdílenou proměnnou, která bude indikovat, že v kritické sekci se právě nachází nějaký proces. Tato proměnná se bude nazývat *lock*, bude Boolovská a její hodnota *true* znamená, že se v kritické sekci právě nachází nějaký proces. Pak výsledný algoritmus může mít tvar:

```
while true do
 begin
 <<await not lock -> lock := true>>
 kritická sekce
 lock := false;
 nekritická sekce
 end;
```

Toto řešení je už prakticky implementovatelné, protože většina procesorů (zvláště procesorů používaných v paralelních počítačích) má speciální instrukci, kterou lze použít pro implementaci atomické akce v algoritmu. Obecně se jedná o instrukci typu **Fetch-and-Something**, tj. instrukce atomicky přečte hodnotu z paměti a provede s ní nějakou operaci. Příklady takové instrukce mohou být **Test-and-Set**, **Fetch-and-Add** nebo **Compare-and-Swap** (bližší vysvětlení i s příklady je možno nalézt v [Goo89]).

Mějme například k dispozici instrukci Test-and-Set (budeme ji nazývat TS). Tato instrukce má dva Boolovské argumenty: proměnnou *lock* a podmínkový kód *cc*. Instrukce TS atomicky nastaví hodnotu parametru *cc* na hodnotu parametru *lock* a pak nastaví proměnnou *lock* na hodnotu *true*:

```
TS(lock, cc): <<cc := lock; lock := true;>>
```

S využitím instrukce TS můžeme implementovat výše uvedený algoritmus:

```
while true do
 begin
 while cc do TS (lock, cc);
 kritická sekce
 lock := false;
 nekritická sekce
 end;
```

Implementace kritické sekce s použitím instrukce Test-and-Set je poměrně jednoduchá. Může však nastat případ, že žádnou vhodnou instrukci typu Fetch-and-Something nemáme k dispozici. V tom případě je nutný složitější algoritmus s několika sdílenými proměnnými. Algoritmů, které je možno použít, je známo několik - například algoritmus Petersonův nebo algoritmus Dekkerův (viz. např. [Sno92]). Vzhledem k tomu, že prakticky každý procesor poskytuje nějakou instrukci typu Fetch-and-Something, je dnes praktická cena těchto algoritmů diskutabilní. Proto zde žádný z těchto algoritmů nebudeme uvádět. S podrobným popisem těchto algoritmů i s historií jejich vývoje se lze seznámit ve specializované publikaci [Ray86].

Jak již bylo řečeno, všechna výše uvedená řešení problému synchronizace pomocí sdílených proměnných používají tzv. *čekání naprázdno* a jsou proto velmi neefektivní. Neefektivnost je zvláště výrazná v okamžiku, kdy dva procesy navzájem soupeří o přístup do kritické sekce. Důvodem je to, že oba procesy neustále přistupují k jedné sdílené proměnné (nebo k několika málo sdíleným proměnným u algoritmů bez instrukce Fetch-and-Something) a tím vzniká tzv. "soupeření o paměť", které degraduje výkon paměťových jednotek a propojovacích sítí mezi procesory a pamětí. Vzhledem k této závažné nevýhodě se začaly používat jiné synchronizační prostředky, o kterých se zmíníme dále.

2.2.2 Semafory

V předchozí kapitole jsme ukázali, jak je možno pro synchronizaci procesů využít pouze sdílenou paměť bez dalších mechanismů, pouze s použitím čekání naprázdno. Synchronizační mechanismy, které používají pouze čekání naprázdno, jsou však obtížně navrhovatelné, obtížně srozumitelné a obtížně se prokazuje jejich správnost. Většina z těchto mechanismů je také poměrně složitá (viz. např. [Ray86]). Rovněž zde není u složitějších programů jasné rozdělení mezi sdílenými proměnnými, které se používají pro synchronizaci a těmi, které se používají pro jiné účely. Výsledkem je, že je velmi obtížné zajistit, že procesy budou správně synchronizovány.

Další nevýhodou čekání naprázdno je jeho neefektivnost. I v multiprocesorovém počítači obvykle na jednom procesoru běží několik procesů (obvykle totiž máme více procesů než kolik je k dispozici procesorů) a proces, který čeká naprázdno, zbytečně zatěžuje procesor.

Protože synchronizace je pro paralelní procesy zásadní, je třeba mít k dispozici jiný prostředek, který pomůže zajistit správnou synchronizaci a dovolí pozastavit procesy, které musí čekat. Jedním z těchto prostředků je *semafor* ([Sno92]). Semafor je jedním z nejstarších synchronizačních mechanismů a pravděpodobně také jedním z nejdůležitějších. Pomocí semaforu je možno snadno zajistit atomičnost kritické sekce a lze ho také využít pro čekání procesu na splnění podmínky. Semafor je možno implementovat několika

způsoby. Lze jej implementovat s využitím čekání naprázdno, ale je možno jej také implementovat tak, aby spolupracoval s operačním systémem (přesněji se správou procesů) a tím lze zajistit synchronizaci bez čekání naprázdno.

Semafor má dvě operace - **P** a **V**. Operace **V** signalizuje výskyt události. Operace **P** slouží k pozastavení procesu, dokud nenastane událost. Tyto dvě operace musí být implementovány tak, aby byla splněna následující podmínka:

Invariant semaforu: Nechť pro semafor *s* je *nP* počet dokončených **P** operací a *nV* je počet dokončených **V** operací. Pokud *init* je počáteční hodnota semaforu *s*, pak ve všech viditelných stavech programu musí platit $nP \leq nV = init$.

Semafor lze implementovat jako celočíselnou proměnnou, která na počátku obsahuje počáteční hodnotu *init* a zaznamenává rozdíl mezi počtem dokončených **P** a **V** operací. **V** operace semafor inkrementuje, **P** operace semafor dekrementuje. Operace **P** a **V** tedy definujeme takto:

```
P(s): <<await s>0 -> s := s - 1>>
V(s): <<s := s + 1>>
```

Operace **P** a **V** musí být samozřejmě implementovány jako atomické akce. Operace **P** a **V** jsou jediné operace nad semaforem - programy nesmí mít možnost jiného přístupu k proměnné semaforu.

Syntaktické zabudování semaforů do programovacího jazyka může být provedeno různě. Častá je například deklarace proměnné, která je typu **sem** a které může být přiřazena počáteční hodnota, jež je počáteční hodnotou semaforu. Deklarace bude například ve tvaru:

```
var mutex : sem := 1;
```

2.2.3 Bariérová synchronizace

Funkce bariérová synchronizace (nebo zkráceně bariéry, [And91]) je podobná funkci bariéry v reálném životě. Jakmile proces při svém vykonávání dorazí k bariéře, není schopen ji sám překonat a musí u ní počkat. Teprve když k bariéře dorazí ještě několik dalších procesů, jsou společnými silami schopny bariéru překonat a společně se přes ni dostat.

Bariéra je tedy mechanismus, který zajišťuje, že několik procesů současně dosáhne při svém vykonávání konkrétního bodu. Parametrem bariéry je její *výška*, která určuje, kolik procesů musí být přítomno u bariéry, aby ji mohli překonat.

Bariéra může být syntakticky implementována několika způsoby. Ukážeme alespoň typický příklad použití bariéry v programovacím jazyce Pascal:

```
  :
  :
Barrier (3); /* Bariéra o výšce 3. */
  :
  :
```

V tomto příkladě je pomocí příkazu *Barrier (3)* vytvořena bariéra o výšce 3. První proces, který dojde k tomuto příkazu je pozastaven a čeká dokud nedorazí ještě další dva procesy. Pak všechny tři procesy bariéru překonají.

2.2.4 Podmíněné kritické oblasti

Semafory jsou základními synchronizačními mechanismy a mohou být použity pro řešení prakticky jakéhokoli synchronizačního problému. Problematická je však skutečnost, že semafory jsou prostředky na značně nízké úrovni, a proto jejich použití snadno svádí k chybám a omylům. Programátor musí dávat velký pozor na to, aby v některém místě programu nezapomněl na operaci **P** nebo **V,** ani žádnou z těchto operací nepřidal. Každé takové opomenutí může vést k těžko odhalitelným chybám.

Podmíněné kritické oblasti (Conditional critical regions, CCR, [And91]) se snaží řešit tento problém tím, že pro vyjádření synchronizace zavádějí strukturovanou notaci. Při použití CCR jsou proměnné, které vyžadují vzájemně výlučný přístup, deklarovány společně jako tzv. prostředek (resource). Tyto proměnné mohou být zpřístupněny pouze příkazem **region**, jenž určuje prostředek, který bude používat. Vzájemné vyloučení je zajištěno tím, že provádění příkazu **region** nemůže probíhat současně s prováděním jiného příkazu **region**, který odkazuje na stejný prostředek. Synchronizace na podmínce je zajištěna pomocí Boolovské podmínky v příkazu **region**. Vzájemné vyloučení je tedy implicitní a synchronizace na podmínce musí být naprogramovaná explicitně. To způsobuje, že programování pomocí CCR je značně jednodušší než programování s využitím semaforů.

CCR zahrnují dva mechanismy: deklaraci prostředků pomocí klíčového slova **resource** a příkazy **region**. Prostředek je pojmenovaná množina sdílených proměnných, ke kterým je požadován vzájemně výlučný přístup. Deklarace prostředku má tvar:

```
resource r (deklarace proměnných);
```

Identifikátor *r* určuje jméno prostředku. Komponentami *r* je jedna nebo několik proměnných. Každá sdílená proměnná v programu musí patřit některému prostředku. Tyto proměnné mohou být zpřístupněny pouze uvnitř příkazů **region**, které odkazují prostředek, do kterého proměnná patří. Příkaz **region** má tvar:

```
region r when B -> S end;
```

kde *r* je jméno prostředku, *B* je Boolovská podmínka a *S* je seznam příkazů. Jak *B*, tak *S* smí přistupovat k proměnným, patřícím prostředku *r*. Klíčové slovo **when** je nepovinné. Lze je vynechat, pokud není třeba specifikovat podmínku *B*.

Vykonávání příkazu **region** je pozastaveno, dokud není splněna podmínka *B*. Jakmile je splněna, jsou provedeny příkazy *S*. Provádění příkazů **region**, které odkazují stejný prostředek, je vzájemně výlučné. Navíc je zajištěno, že před vykonáním příkazů *S* je

splněna podmínka *B*. Příkaz **region** je tedy velmi podobný příkazu **await**. Závažný rozdíl je však v tom, že v příkazu **await** je možno přistupovat k *jakýmkoli* proměnným a v příkazu **region** pouze k proměnným, které patří do odkazovaného prostředku. Tím bylo dosaženo toho, že příkaz **region** je, na rozdíl od příkazu **await**, mnohem snadněji implementovatelný.

Přesto je však příkaz **region** implementovatelný stále obtížně. Důvodem je skutečnost, že Boolovské výrazy *B* musí být vyhodnocovány vždy, když se mění hodnota některé ze sdílených proměnných. CCR jsou tedy poměrně nevýhodné z hlediska výkonnosti a jsou využívány poměrně velmi málo. CCR je možno simulovat pomocí několika semaforů. Naopak semafor je možno simulovat pomocí CCR. To znamená, že CCR jsou ze sémantického hlediska ekvivalentní semaforům. Simulace CCR pomocí semaforů je poměrně složitá; lze ji nalézt například v [And91].

2.2.5 Monitory a podmínkové proměnné

Při použití semaforů a CCR jsou sdílené proměnné stále viditelné globálně ve všech procesech. To znamená, že přístupy k těmto sdíleným proměnným mohou být rozptýleny kdekoli po celém programu. Aby programátor porozuměl tomu, jak jsou sdílené proměnné v programu používány, musí prozkoumat celý program. Pokud je do programu přidán nový proces, opět je třeba prozkoumat, zda používá sdílené proměnné správně.

Monitory ([And91]) jsou programové moduly, které jsou strukturovanější než CCR a mohou být implementovány stejně efektivně jako semafory. Monitory jsou navíc datovou abstrakcí: monitory zapouzdřují reprezentaci abstraktních prostředků a poskytují sadu operací, jejichž prostřednictvím (a nijak jinak) je možno manipulovat se zapouzdřenými prostředky. Proces interaguje s monitorem pouze tím, že volá jeho procedury. Vzájemná výlučnost je zajištěna tím, že provádění procedur téhož monitoru se nesmí časově překrývat. To je podobné jako implicitní vzájemné vyloučení, poskytované příkazem **region**. Na rozdíl od příkazu **region** je však synchronizace na podmínce zajišťována mechanismem na nízké úrovni, který se nazývá *podmínkové proměnné* (condition variables). Jak uvidíme dále, tyto podmínkové proměnné se používají podobně jako semafory.

Monitor obsahuje jak proměnné, tak i procedury, kterými se k těmto proměnným přistupuje. Deklarace monitoru má tvar:

```
monitor Mname
deklarace statických proměnných
inicializační příkazy
procedure op1 (parametry) tělo op1 end;
...
procedure opn (parametry) tělo opn end;
end;
```

Statické proměnné v monitoru reprezentují stav sdíleného prostředku. Tyto proměnné jsou nazývány statické, protože existují tak dlouho, jak dlouho existuje monitor. Pokud chce proces změnit stav sdíleného prostředku, musí zavolat některou proceduru monitoru. Syntakticky má volání procedury monitoru tvar:

```
call Mname.opi (parametry);
```

kde opi je jedna z procedur monitoru.

Vzájemná výlučnost je monitorem zajišťována implicitně. Synchronizace na podmínce musí být naprogramována explicitně pomocí podmínkových proměnných.

Podmínková proměnná se používá pro pozastavení procesu, který nemůže pokračovat ve své činnosti. Deklarace podmínkové proměnné má tvar:

```
var c: cond;
```

Protože podmínkové proměnné se používají pouze pro synchronizaci uvnitř monitoru, mohou být deklarovány pouze uvnitř monitoru. Pro pozastavení procesu na podmínkové proměnné proces provede příkaz:

```
wait (c);
```

Provedení příkazu **wait** způsobí, že volající proces bude pozastaven a umístěn na konec fronty procesů, čekajících na podmínku *c*. Protože zavoláním příkazu **wait** proces uvolňuje monitor, může v tomto okamžiku vstoupit do monitoru jiný proces a ten může také později probudit proces, který čeká na podmínku. Procesy, pozastavené na podmínkové proměnné, jsou probuzeny příkazem **signal**. Pokud na podmínkovou proměnnou *c* čeká alespoň jeden proces, pak zavolání příkazu

```
signal (c);
```

probudí tento proces. Pokud na podmínkovou proměnnou čekalo procesů několik, je vybrán proces ze začátku fronty čekajících. Probuzený proces bude pokračovat ve své činnosti, jakmile bude moci znovu získat výlučný přístup k monitoru. Pokud na podmínkovou proměnnou nečekal řádný proces, provedení příkazu **signal** nemá žádný efekt. Procesu, který provedl příkaz **signal**, zůstává výlučný přístup k monitoru a může pokračovat bez ohledu na to, zda se příkazem **signal** probudil některý jiný proces. Proto se někdy říká, že příkaz **signal** má sémantiku *signal-and-continue*.

Operace **wait** a **signal** jsou podobné operacím semaforu **P** a **V** - příkaz **wait** pozastavuje proces jako **P** a příkaz **signal** probouzí proces jako **V**. Mezi těmito operacemi však existuje několik důležitých rozdílů.

1. Operace **signal** nemá žádný účinek, pokud na podmínkové proměnné nečeká žádný proces - její provedení není nikde zapamatováno.
2. Operace **wait** vždy pozastaví proces, dokud není provedena operace **signal**.
3. Proces, provádějící operaci **signal** má vždy přednost (alespoň uvnitř monitoru) před procesem, který byl probuzen.

Pomocí monitoru je možno implementovat semafor. Implementaci lze provést několika způsoby. Jednu z možných implementací převzatou z [And91] ukážeme:

```
monitor Semaphore
var s:=0, pos : cond;
procedure P(); begin while s=0 do wait(pos); s:=s-1; end;
procedure V(); signal(pos); s:=s+1; end;
end;
```

Stejným způsobem můžeme pomocí semaforu implementovat monitor. Pro implementaci monitoru použijeme jeden semafor *e*, který bude sloužit jako vstupní semafor pro zajištění vzájemné výlučnosti. Jeho počáteční hodnota bude 1 a bude nabývat pouze hodnot 0 nebo 1.

Pro implementaci podmínkové proměnné potřebujeme také pouze jeden semafor *c* (doplněný celočíselnou proměnnou *nc*, která slouží jako čítač počtu procesů čekajících na podmínkovou proměnnou). Počáteční hodnota semaforu *c* je 0 a počáteční hodnota čítače *nc* je také 0. Implementace pak může mít například následující tvar ([And91]):

```
sdílené proměnné
        e : sem := 1;
        c : sem := 0;
        nc : integer := 0;

vstup do monitoru: P(e);
wait (cond): nc := nc + 1; V(e); P(c); P(e);
signal (cond): if nc > 0 then begin nc := nc - 1; V(c); end;

výstup z monitoru: V(e);
```

Vzhledem k tomu, že dokážeme pomocí monitoru simulovat semafor a pomocí semaforu monitor i podmínkovou proměnnou, můžeme tvrdit, že z hlediska vyjadřovací schopnosti je semafor a monitor ekvivalentní.

2.3 Interakce procesů předáváním zpráv

Interakční mechanismy, které jsme si uvedli doposud, byly založeny na sdílených proměnných. Tyto mechanismy jsou obvykle používány v architekturách, ve kterých procesory sdílejí paměť. Existuje však velké množství architektur, ve kterých jsou procesory spojeny pouze komunikační sítí. V těchto architekturách mohou procesory komunikovat pouze *předáváním zpráv*.

Aby bylo možno psát programy pro tyto architektury, je nutno definovat *rozhraní*, to znamená základní operace pro předávání zpráv. Bylo by samozřejmě možné použít operace *read* a *write*, podobně jako při práci se sdílenou pamětí. To by však znamenalo, že by procesy musely používat synchronizaci pomocí čekání naprázdno. Proto je výhodnější definovat speciální operace pro předávání zpráv. Tyto operace pak lze použít jak pro komunikaci mezi procesy, tak i pro synchronizaci procesů. Operace, které budeme používat, se nazývají **send** a **receive**.

Při interakci procesů pomocí předávání zpráv jsou data předávána mezi procesy explicitně. Procesy jsou spojeny pomocí komunikačních *kanálů* nebo komunikačních poštovních *přihrádek*.

Komunikační *kanál* spojuje dva procesy a je jednosměrný (jeden proces může pouze zasílat zprávy a druhý proces může pouze přijímat zprávy). Dva procesy mohou však

komunikovat pomocí několika kanálů. Z hlediska syntaktické identifikace kanálů se používá několik způsobů *pojmenování*. Při statickém pojmenování kanálů jsou kanály mezi procesy ustaveny už v okamžiku kompilace programu. Při dynamickém pojmenování kanálů mohou být kanály vytvářeny a rušeny až při běhu programu.

Podobným komunikačním prostředkem je *předávání zpráv s přímým adresováním*, při kterém proces-odesílatel zasílá zprávu konkrétnímu procesu-příjemci. V tomto případě se jedná o zvláštní případ komunikace pomocí kanálů, kdy mezi každými dvěma procesy může existovat pouze jeden kanál.

Poštovní *přihrádka* může být cílem jakéhokoli příkazu *send* nebo zdrojem jakéhokoli příkazu *receive*. Výhodou přihrádek (nebo také nevýhodou) je to, že odesílatel nemá kontrolu nad tím, který proces zprávu přijme. Přihrádky mohou být přístupné globálně všem procesům, obvykle jsou však sdíleny pouze mezi malou skupinou procesů.

Podle toho, kolik procesů při komunikaci zasílá zprávy a kolik jich zprávy čte, je možno vytvořit tyto typy komunikace:

- 1:1 - V tomto případě je pouze jeden odesílatel a pouze jeden příjemce. Jedná se o komunikaci pomocí kanálu.
- N:1 - Jeden proces přijímá zprávy od několika jiných procesů. Použitým komunikačním prostředkem musí být přihrádka.
- 1:N - V tomto případě je mnoho potenciálních příjemců (ale pouze jeden skutečný) a jeden odesílatel. Použitý mechanismus musí být přihrádka.
- N:N - V tomto případě je mnoho potenciálních příjemců a mnoho odesílatelů. Použitý mechanismus musí být přihrádka.

2.3.1 Asynchronní a synchronní předávání zpráv

Z hlediska synchronizace rozdělujeme mechanismy s předáváním zpráv na tři skupiny: asynchronní předávání zpráv, synchronní předávání zpráv a synchronní předávání zpráv s omezenou kapacitou.

Princip *asynchronního předávání* zpráv je jednoduchý: Odesílatel zašle zprávu a pokračuje ve svém běhu. Zprávu převezme systém a někdy později ji doručí příjemci. Proces, čekající na přijetí zprávy, je pozastaven, dokud k němu zpráva nedorazí. Operace *send* je vždy neblokující a operace *receive* je blokující. Reálným příkladem asynchronního předávání zpráv je například elektronická pošta. Asynchronní předávání zpráv má dvě nevýhody:

- Odesílatel neví, zda a kdy byla zpráva přijata.
- Pro uložení nedoručených zpráv je třeba velký paměťový prostor. Tento paměťový prostor musí mít neomezenou kapacitu.

Vzhledem k výše uvedeným dvěma nevýhodám neexistuje implementace předávání zpráv, která by zcela dodržovala princip asynchronnosti. V každé použitelné implementaci je jistý způsob potvrzení příjmu zprávy. A navíc žádná implementace nemá k dispozici neomezenou kapacitu pro dočasné uložení nedoručených zpráv. Proto každá reálná implementace asynchronního předávání zpráv je ve skutečnosti předáváním zpráv s omezenou (byť velmi velkou) kapacitou.

Při *synchronním předávání zpráv* může být jak příjemce, tak i odesílatel pozastaven, dokud není protistrana připravena ke komunikaci. Proto zde není potřeba žádného dočasného paměťového prostoru pro ukládání zpráv. V této variantě předávání zpráv jsou operace *send* a *receive* blokující. Příkladem synchronního předávání zpráv je mechanismus CSP (communicating sequential processes, viz. [Hoa85]) a jeho implementace - programovací jazyk OCCAM (viz. [In88b]).

Při *synchronním předávání zpráv s omezenou kapacitou* je mezi odesílatelem a příjemcem vyrovnávací paměť o pevné, omezené kapacitě. Dokud není tato vyrovnávací paměť naplněna, chová se tento mechanismus jako asynchronní předávání zpráv. Jakmile se vyrovnávací paměť naplní, operace *send* začne být blokující a mechanismus se chová jako synchronní předávání zpráv. Pokud je velikost vyrovnávací paměti nulová, mechanismus je ekvivalentní se synchronním předáváním zpráv.

2.3.2 RPC a rendezvous

Při vytváření paralelních programů se často používá konstrukce, nazývaná klient-server. Klient je proces, který požaduje službu a na základě jeho požadavku mu server tuto službu poskytuje. Klient iniciuje činnost a to v okamžiku, který si sám zvolí. Obvykle pak čeká, dokud jeho požadavek není vyřízen. Server pouze čeká na požadavky a pak na ně reaguje. Server je často stále běžící proces, který zpracovává požadavky od několika klientů.

Pro implementaci konstrukce klient-server lze samozřejmě použít jak asynchronní, tak i synchronní kanály. Protože při interakci klient-server se jedná o dvoucestnou komunikaci (požadavek - odpověď), musíme použít dvě samostatné zprávy, které jsou přenášeny dvěma kanály. Proto byly vytvořeny dva interakční mechanismy, které usnadňují realizaci konstrukce klient-server. Těmito mechanismy jsou *volání vzdálených procedur* (Remote Procedure Call, *RPC*) a *rendezvous* (viz. [Dei90]). Oba tyto mechanismy kombinují principy monitoru a synchronního předávání zpráv. Podobně jako u monitorů modul (nebo proces) vyváží několik operací a tyto operace jsou volány stejně jako běžné procedury. Novinkou je to, že při zavolání operace se použije obousměrný kanál mezi klientem a serverem, který přenáší jedním směrem požadavek a opačným směrem odpověď.

Rozdíl mezi RPC a rendezvous je ve způsobu, jak server čeká na požadavek. První možností je vytvořit pro každou službu jednu proceduru a při každém zavolání služby spustit proces, který požadavek zpracuje. Tento způsob se nazývá *volání vzdálených procedur* (*RPC*). Druhou možností je *rendezvous*, kdy proces serveru už existuje a čeká (prostřednictvím příkazu *input* nebo *accept*) na požadavek.

2.3.3 Budoucí proměnné a Cboxy

Dalšími dvěma možnostmi, jak zabudovat mechanismus klient-server do programovacího jazyka, jsou takzvané *budoucí proměnné* (future variables) a *Cboxy* [Kaf93].

Budoucí proměnná je proměnná, která je vytvořena v okamžiku zavolání vzdálené funkce a která bude v budoucnu (po dokončení vyhodnocování vzdálené funkce) obsahovat její návratovou hodnotu. Volající proces není při zavolání vzdálené funkce blokován a pokračuje ve svém běhu. Zablokován může být pouze v případě, když potřebuje hodnotu této budoucí proměnné a volaný proces ještě neskončil vyhodnocování vzdálené funkce. Použití budoucí proměnné může vypadat například takto:

```
future v := f(...);       volání funkce f
....                      probíhá současně
                          s vyhodnocováním funkce f
i := v;                   zablokování dokud neskončí f
```

```
Cbox je obecnějším případem budoucí proměnné. Cbox je proměnná,
která má podobné vlastnosti jako budoucí proměnná (tj. někdy v
budoucnu bude obsahovat hodnotu). Cbox je však možno předávat volané
vzdálené funkci jako parametr (to znamená, že funkce může mít jako
parametry i několik Cboxů). Dalším důležitým rozdílem je to, že
pokud vzdálená funkce volá jinou vzdálenou funkci, může jí jako
parametr předat tento Cbox, který si zachovává své vlastnosti. Nyní
uvedeme příklad použití Cboxu:
```

KLIENT

```
Cbox v1, v2;              klient vytvoří Cboxy
g (v1, v2);               Cboxy jsou předány jako argumenty
...
i := v1;                  blokuje dokud v1 neobsahuje hodnotu
...
i := v2;                  blokuje dokud v2 neobsahuje hodnotu
```

SERVER

```
function g (v1, v2);
...
v1 := 1;                  Cbox v1 dostává hodnotu
...
h (v2);                   Cbox v2 je předán funkci h
```

2.4 Nástěnka v jazyce Linda

Programovací jazyk Linda ([Gel85], [Car86], [Car88], [Car89], [Car90], [Car91]) je určen pro programování v paralelním a distribuovaném výpočetním prostředí. Jazyk Linda se podstatně liší od běžných programovacích jazyků. Neexistují v něm proměnné ani příkazy - tyto mechanismy jsou zajištěny jiným programovacím jazykem, který se nazývá *hostitelský jazyk*. Jazyk Linda je založen na modelu takzvané *generativní komunikace*. Základem modelu generativní komunikace ([Gel85]) je abstraktní, globální a datově nezávislá struktura, nazývaná *nástěnka* (anglicky tuple space). Datové elementy, které se vkládají do nástěnky, se nazývají *n-tice* (anglicky tuples). Každý z paralelně běžících

procesů může přistupovat k n-ticím v nástěnce, avšak n-tice nejsou nijak vázány na procesy - n-tice nemají vlastníka. N-tice jsou uspořádané posloupnosti *atomů*, ve kterých uvažujeme jak pořadí prvků, tak jejich násobnost.. Každý atom má jako atribut datový typ. Tento typ musí odpovídat některému z datových typů, definovaných v hostitelském programovacím jazyce. V nástěnce se může vyskytovat více n-tic se stejným obsahem.

Atomy mohou být trojího druhu:

- *Konstanty* - tyto elementy se zapisují stejně jako konstanty hostitelského jazyka a mají stejný význam.
- *Aktuální atomy (aktuály)* - jsou proměnné nebo výrazy, které slouží jako vstupní parametry operátorů jazyka Linda.
- *Formální atomy (formály)* - jsou proměnné, které slouží jako výstupní parametry operátorů jazyka Linda. Zápis aktuálu se od zápisu formálu liší tím, že před identifikátorem formálu se musí vyskytovat znak otazník '?'.

Při operacích s nástěnkou v jazyce Linda zavedeme pojem *srovnatelné n-tice*. Abychom dvě n-tice, uložené v nástěnce, považovali za ekvivalentní, nepožadujeme jejich shodnost (tzn. shodnost všech jejich atomů), ale postačuje nám jejich srovnatelnost.

Srovnatelnost dvou n-tic se určuje podle následujících pravidel:

- Pravidlo 1: Aby dvě n-tice byly srovnatelné, musí mít stejný počet atomů a odpovídající dvojice atomů musí být srovnatelné.
- Pravidlo 2: Aktuál je srovnatelný s aktuálem, konstanta je srovnatelná s konstantou a aktuál je srovnatelný s konstantou, pokud jsou stejného typu a vyjadřují stejnou hodnotu.
- Pravidlo 3: Aktuál je srovnatelný s formálem a konstanta je srovnatelná s formálem, pokud jsou stejného typu (formál nemá hodnotu).
- Pravidlo 4: Formál není srovnatelný s formálem.

Příklady srovnatelných n-tic:

```
('T1', 1) ('T1', 1)
('T1', 1) ('T1', ?A)
('T1', B) ('T1', ?A)
('T1', B) ('T1', A) pokud A=B
```

Příklady nesrovnatelných n-tic:

```
('T1', 1) ('T1', 2)
('T1', 1) ('T2', 1)
('T1', B) ('T1', A) pokud A<>B
```

V jazyce Linda je definováno pouze šest operátorů: *out (tuple)*, *rd (tuple)*, *in (tuple)*, *inp (tuple)*, *rdp (tuple)* a *eval (arguments).*

```
out (TUPLE)
```

Operátor **out** slouží pro vložení n-tice do nástěnky. Proces, který provádí tento operátor, není pozastaven a nemusí čekat na provedení operace. Například operace *out ('T1', 2)* vloží do nástěnky n-tici *('T1', 2).*

```
rd (TUPLE)
```

Operátor **rd** slouží pro přečtení hodnoty n-tice, umístěné na nástěnce. Při provádění operace rd se v nástěnce hledá n-tice, která je srovnatelná s n-ticí *TUPLE*. Výsledkem provedení příkazu *rd* je doplnění hodnot do všech formálů v n-tici *TUPLE* (pokud nějaké obsahuje). Není-li v nástěnce nalezena žádná n-tice, srovnatelná s *TUPLE*, pak proces, provádějící příkaz *rd*, je pozastaven až do doby, dokud žádaná n-tice není do nástěnky vložena. Například operace *rd('T1', ?x)* najde v nástěnce srovnatelnou n-tici *('T1', 2)* vloženou v předchozím příkladu a výsledkem operace je přiřazení hodnoty 2 proměnné *x*.

```
in (TUPLE)
```

Operátor **in** je podobný operátoru *rd*. Rozdíl mezi nimi je pouze v tom, že operátor *in* způsobí odstranění nalezené n-tice z nástěnky.

```
rdp (TUPLE), inp (TUPLE)
```

Operátory **rdp** a **inp** jsou predikátové verze operátorů *rd* a *in*. Proces, který provádí některý z těchto operátorů, není zablokován, pokud se nenajde srovnatelná n-tice v nástěnce. Místo toho tyto operace vracejí hodnotu *TRUE*, pokud byla n-tice nalezena a operace je úspěšná nebo hodnotu *FALSE* v případě, že operace je neúspěšná.

```
eval (ARGUMENT1, ARGUMENT2, ....ARGUMENTn)
```

Tento operátor je jediným prostředkem, který dovoluje v jazyce Linda vytvářet paralelně běžící procesy. Argumentem operátoru *eval* může být element (tak jako v operátoru *out*) nebo tzv. aplikace. Aplikace je uspořádaná posloupnost, která se skládá z funkce a z argumentů, na něž je tato funkce aplikována. Jazyk Linda při provádění tohoto operátoru zajistí paralelní provádění všech aplikací operátoru eval. Po vyhodnocení všech aplikací je do nástěnky vložena n-tice *(VAL1, VAL2, ...VALn)*, kde $VAL_i=ARGUMENT_i$ pokud $ARGUMENT_i$ byl element nebo VAL_i je výsledek vyhodnocení aplikace $ARGUMENT_i$, pokud $ARGUMENT_i$ byla aplikace. Z pohledu funkcionálního programování jsou operátory *eval* a *out* totožné. Rozdíl mezi nimi spočívá pouze v tom, že u operátoru *out* se provede striktní vyhodnocení (eager evaluation) argumentů a u operátoru *eval* nestriktní (lazy evaluation).

2.5 Klasifikace interakčních mechanismů

V předchozích kapitolách jsme uvedli stručný přehled běžně používaných interakčních mechanismů, jejich názvy a jejich funkci. Většina těchto mechanismů byla vytvořena ad hoc, bez toho, aby byl brán ohled na jejich vztah s jinými mechanismy. Výsledkem tohoto spontánního vývoje interakčních mechanismů je spousta existujících

interakčních mechanismů, které většinou nejsou propojeny na základě žádného systematického vztahu. Existovala zde však potřeba klasifikace těchto mechanismů - proto byly uskutečněny pokusy o systematickou klasifikaci těchto interakčních mechanismů. Většina z těchto známých pokusů o klasifikaci interakčních mechanismů je však buď příliš obecná (bez schopnosti popsat rozdíly mezi podobnými mechanismy) nebo příliš speciální (bez schopnosti popsat celý rozsah známých mechanismů). Tyto pokusy navíc zpravidla nejsou navzájem konzistentní a také nebyly obecně přijaty do povědomí informatické veřejnosti. Navzdory těmto nedostatkům je však třeba se o těchto pokusech o klasifikaci interakčních mechanismů zmínit.

Základním schématem pro klasifikaci interakčních mechanismů je jejich rozdělení do dvou tříd - na mechanismy se sdílenou pamětí a mechanismy s předáváním zpráv. Toto rozdělení je velmi praktické a logické, protože obě tyto skupiny interakčních mechanismů mají rozdílný charakter. Tato klasifikace interakčních mechanismů je široce akceptovaná a je také prakticky používaná. Má však také některé nedostatky.

Prvním nedostatkem této klasifikace je to, že klasifikace pouze na dvě skupiny je velmi hrubá a není schopna popsat rozdíly mezi mechanismy uvnitř těchto dvou skupin.

Druhým nedostatkem je skutečnost, že rozdělení interakčních mechanismů je provedeno pouze z implementačního hlediska a nikoli také z uživatelského hlediska. Výsledkem je, že nejsme schopni klasifikovat některé abstraktnější interakční mechanismy. Například mechanismus *virtuální sdílené paměti* je z hlediska uživatele mechanismem se sdílenou pamětí, avšak je implementován jako mechanismus s předáváním zpráv.

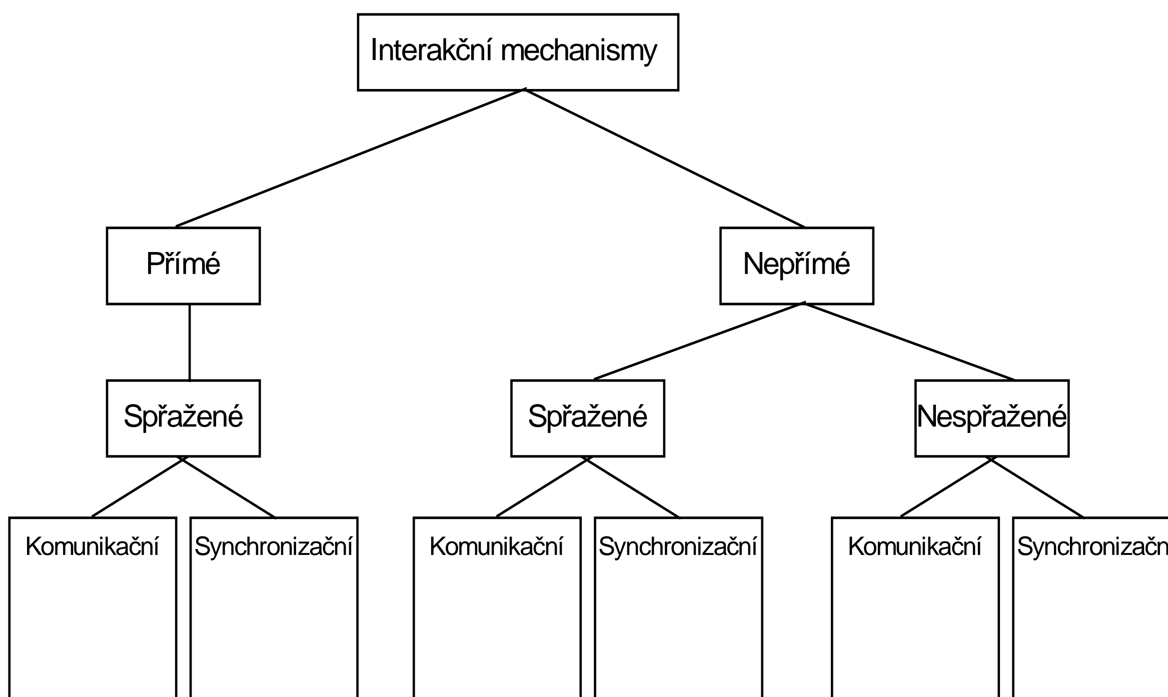
Druhým pokusem o klasifikaci interakčních mechanismů, který ukážeme, byl vyvinut jako součást projektu BACS [Gut93]. Při tomto způsobu klasifikace jsou použita pro rozlišování jednotlivých typů interakčních mechanismů následující tři kritéria:

Prvním kritériem je, zda je interakční mechanismus použit pro předávání dat, nebo pouze pro synchronizaci. Pomocí tohoto kritéria je možno rozdělit interakční mechanismy na dvě skupiny - na komunikační a na synchronizační mechanismy.

Druhým kritériem je odpověď na otázku, zda interakční mechanismus slouží pro interakci mezi dvěma procesy (*přímý interakční mechanismus*) nebo mezi více procesy (*globální interakční mechanismus*). Toto kritérium nám dovoluje rozlišovat například mezi kanálem, který je přímým interakčním mechanismem a schránkou, jež je nepřímým interakčním mechanismem. Některé interakční mechanismy, jako je například RPC nebo semafory se mohou chovat jako přímé i jako globální interakční mechanismy.

Třetím kritériem je to, zda se proces účastní interakce anonymně (*nespřažený interakční mechanismus*) nebo jeho účast je explicitní (*spřažený interakční mechanismus*). Přímé interakční mechanismy mohou být pouze spřažené. Globální interakční mechanismy mohou být nespřažené (např. semafor) nebo spřažené (např. bariéra). Toto kritérium je méně obecné než předchozí dvě kritéria a někdy popisuje pouze syntaktické rozdíly.

Výsledkem aplikace těchto tří kritérií je rozdělení interakčních mechanismů do osmi možných skupin. Protože některé z těchto skupin však nemají reálný význam, zůstává použitelných šest skupin interakčních mechanismů. Následující obrázek (převzatý z [Gut93]) ukazuje výsledné skupiny spolu s některými reprezentanty těchto skupin.



Obr. 3: Klasifikace interakčních mechanismů podle [Gut93]

Poslední způsob klasifikace interakčních mechanismů, který ukážeme, je popsán v článku [Kaf93]. Tato klasifikace je poměrně složitá, a proto popíšeme pouze její hlavní koncept. Podrobnosti je možno nalézt v materiálu [Kaf93]. Klasifikace se skládá ze tří modelů - z *modelu animace*, *modelu interakce* a *modelu synchronizace*. Model animace popisuje mechanismy realizace procesů a vláken (threads) a z našeho pohledu není příliš užitečný. Model interakce se zabývá sémantikou interakce mezi aktivním účastníkem interakce (klientem) a pasivním účastníkem (serverem). Navzdory svému pojmenování nemá model synchronizace nic společného se synchronizací v našem slova smyslu. Tento model se používá pro popis rozhraní mezi procesem a interakčním mechanismem a je vhodný pro popis takových mechanismů jako je příkaz *select* v jazyce ADA ([Sno92]), nebo strážený vstup v jazyce OCCAM ([In88b]). Poslední a nejdůležitější je model interakce. Tento model klasifikuje interakční mechanismy na základě tří vlastností: *sémantiky volání*, *návratové sémantiky* a *návratového adresování*. Sémantika volání může být buď synchronní nebo asynchronní. Synchronní a asynchronní volání se liší ve svém dopadu na klienta. Synchronní volání znamená, že klient je během zpracování svého požadavku serverem zablokován. Oproti tomu asynchronní volání znamená, že klient může během vyřizování svého požadavku běžet paralelně se serverem.

Návratová sémantika může být buď explicitní nebo implicitní. V případě explicitní návratové sémantiky je identifikace objektu, kterému bude zaslán výsledek, známa serveru explicitně. V případě implicitní návratové sémantiky je identifikace objektu, kterému bude zaslán výsledek, implicitní a nemůže být serverem ovlivněna.

Návratové adresování může být buď *hodnotou* nebo *místem určení*. V případě adresování místem určení je identita klienta pro server explicitně viditelná. V případě návratového adresování hodnotou je identita klienta zabudována v objektu, kterým se vrací

návratová hodnota. Server nemusí vědět, který objekt očekává návratovou hodnotu. Může však být schopen s tímto objektem manipulovat - například jej předat jinému serveru.

Tento způsob klasifikace interakčních mechanismů není však dostatečně obecný. Je vhodný pro popis interakčních mechanismů, založených na předávání zpráv nebo na konstrukci klient-server, avšak selhává při popisu jiných tříd interakčních mechanismů.

3. MODEL PRO VYJÁDŘENÍ INTERAKČNÍCH MECHANISMŮ

3.1 Použitý výpočetní model

Existuje mnoho teoretických modelů pro popis paralelních výpočtů. Tyto modely lze rozdělit do dvou tříd: modely s *distribuovanou pamětí* a modely se *sdílenou pamětí*. U modelů s distribuovanou pamětí je paměť paralelního počítače distribuována mezi jednotlivé procesory. Komunikace mezi procesory je zajištěna jistým komunikačním mechanismem. U modelů se sdílenou pamětí všechny procesory sdílejí jedinou paměť. Jeden z modelů se sdílenou pamětí je model PRAM, který je rozšířením modelu RAM pro paralelní výpočty.

Nejdříve se krátce zmíníme o výpočetním modelu RAM (Random Access Machine). RAM se skládá z programu, který běží na *procesoru* a z neomezeného počtu *registrů*, označených *1, 2,...*, z nichž každý je schopen uložit libovolnou celočíselnou hodnotu (tj. hodnotu s libovolným počtem bitů). V jednom kroku může základní model RAM provést přímý přístup k paměti (to znamená, že adresa registru, ke kterému se přistupuje, závisí pouze na *konstantě*, nikoli na obsahu jiného registru), zapsat konstantu do registru 1, sečíst nebo odečíst obsah dvou registrů nebo provést podmíněný skok na základě obsahu registru 1. Většina skutečně používaných modelů RAM má rozšířenou instrukční sadu. Oproti základnímu modelu může provádět nepřímý přístup k paměti, to znamená, že adresa může záviset na obsahu některého registru.

Výpočetní model PRAM ([Wil90]) je přímočarým zobecněním (paralelizací) modelu RAM. Model PRAM se skládá z *P* modelů RAM, které mají společnou paměť, ke které mohou jednotlivé modely RAM přistupovat nezávisle na sobě a v konstantním čase. Sdílená paměť slouží jako efektivní komunikační prostředek mezi procesory: komunikace mezi procesory se skládá z jedné čtecí a jedné zápisové operace, to znamená, že ji lze provést v konstantním čase. Díky tomuto rysu je model PRAM velmi mocný, dostatečně obecný a poměrně jednoduchý pro programování a analýzu. Na druhé straně tento rys způsobuje, že tento model je (alespoň současnou technologií) neimplementovatelný.

Model PRAM s *P* procesory P_1, ..., P_P se tedy skládá z *P* RAMů a neomezené sdílené paměti, skládající se ze sdílených registrů, označených 1, 2,..., z nichž každý je schopen pojmout jednu celočíselnou hodnotu. Každý procesor je schopen přistupovat přímo (a nepřímo) jak ke své lokální paměti, tak i ke sdílené paměti. Každý procesor má také takzvaný *registr signatury* SIG, který obsahuje jedinečný *identifikátor procesoru* (v intervalu od 1 do *P*). Každý RAM obsahuje stejný *program*, který všechny RAMy provádějí *synchronně*, instrukci po instrukci. Protože každý RAM má různý identifikátor procesoru, je také schopen sám sebe identifikovat a na základě této identifikace případně skočit do jiné části programu, než ostatní procesory.

Sdílená paměť se skládá z neomezeného počtu sdílených registrů, označených 1, 2, 3,.... Každý registr sdílené paměti může obsahovat libovolnou celočíselnou hodnotu. Podle toho, zda je povoleno současné čtení jednoho registru dvěma procesory, se modely PRAM

dělí na model se *současným čtením* (*concurrent read*, CR) a s *výhradním čtením* (*exclusive read*, ER). Podobně podle toho, zda je povolen současný zápis do jednoho registru dvěma procesory se modely PRAM dělí na model se *současným zápisem* (*concurrent write*, CW) a s *výhradním zápisem* (*exclusive* write, EW). Z těchto dvou možných zápisových a dvou čtecích modelů je možno vytvořit čtyři možné modely PRAM - EREW, ERCW, CREW a CRCW.

Model PRAM ve formě, která byla prezentována v předchozích odstavcích není pro naše účely příliš vhodný. Aby byl tento model realističtější, je v něm potřeba provést několik úprav. Především je třeba zrušit předpoklad, že instrukce všech procesorů se provádějí synchronně.

Tato modifikace modelu PRAM je motivována vlastnostmi reálných paralelních počítačů typu MIMD. Tyto počítače jsou asynchronní, to znamená, že procesory nemusí vykonávat instrukce synchronně. Každý procesor může běžet svou vlastní rychlostí a ostatními procesory je ovlivňován pouze v explicitních synchronizačních bodech. U modelu PRAM běží všechny procesory synchronně. Aby program vytvořený pro model PRAM mohl běžet na asynchronním víceprocesorovém počítači, za *každou* instrukcí původního modelu PRAM musí být vložen explicitní synchronizační bod. Synchronizace po každé instrukci je však velmi neefektivní, protože schopnost procesorů, běžet svou vlastní rychlostí, není využita a je zde poměrně značná režie, spojená se zajištěním synchronizace.

Výsledný model, zvaný *asynchronní PRAM* (APRAM), se skládá, podobně jako PRAM, z *P* modelů RAM. Na rozdíl od modelu PRAM běží procesory asynchronně, to znamená, že každý procesor vykonává své instrukce nezávisle na ostatních procesorech. Jakýkoli časový vztah mezi procesory musí být explicitně zabudován do programu synchronizačními mechanismy.

3.2 Sdílený datový prostor

V této kapitole budeme definovat nový pojem - *sdílený datový prostor*. Pro popis sdíleného datového prostoru využijeme model asynchronní PRAM, definovaný v předchozích odstavcích.

První pojem, který budeme definovat je *datový prostor* (DS, Data Space). Datový prostor je nekonečná množina sdílených registrů, definovaných v modelu APRAM. Obsah každého takového sdíleného registru je posloupnost bitů libovolné délky. Tyto posloupnosti bitů, které tvoří obsah datového prostoru, budeme nazývat *datové objekty* nebo zkráceně *objekty* (pojem objekt, použitý v této práci, nemá žádný vztah k pojmu objekt, používanému v objektově orientovaném programování). Je třeba poznamenat, že délka těchto posloupností bitů není pevná a může se měnit během procesu výpočtu.

Každý datový objekt může být jednoznačně identifikován svou ordinální pozicí ve výčtu, například 1, 2, 3 atd. Tato ordinální pozice je ekvivalentní číslu registru, používanému u výpočetního modelu APRAM. Ordinální pozici budeme nazývat *jedinečný identifikátor objektu*, nebo zkráceně *jedinečný identifikátor* a budeme jej označovat *id*. Tento jedinečný identifikátor bychom také mohli nazvat *adresou objektu*, protože určuje umístění objektu ve výčtu. Nebudeme jej tak však nazývat, protože pojmy umístění či adresa často vyvolávají dojem pevné délky a pevného umístění objektu na nějakém

konkrétním paměťovém médiu, čemuž bychom se rádi vyhnuli. Pro označení řetězce bitů, který patří objektu, použijeme v naší notaci příponu *object*. To znamená, že zápis *id.object* označuje řetězec bitů, který je spojen s identifikátorem *id*. Jedinečnost identifikátoru *id* vede k implikaci:

$$id_1 = id_2 \text{ --> } id_1.object = id_2.object$$

Jak již bylo řečeno, pro naše účely použijeme výpočetní model APRAM. Proto každý proces budeme modelovat jedním procesorem tohoto výpočetního modelu, jenž je typu RAM. Pro označení procesu (nebo jeho odpovídajícího modelu RAM) použijeme symbol P_i. Protože potřebujeme, aby tyto procesy pracovaly se společnými daty, budeme předpokládat, že všechny výše uvedené procesy (procesory RAM), mohou číst data ze sdíleného *datového prostoru* (DS) a mohou také data do tohoto datového prostoru zapisovat. Pro čtení a zápis do datového prostoru budou platit výše definovaná pravidla pro model APRAM.

Nyní provedeme definici některých operací, které pracují nad objekty ve sdíleném datovém prostoru. Vzhledem k tomu, že se budeme zabývat paralelními procesy, které provádějí tyto operace, je třeba zavést do naší notace pojem *času*. Pro jednoduchost budeme předpokládat, že čas, který označíme t_k, je diskrétní, a že žádné dvě operace nad jedním objektem ve sdíleném datovém prostoru nemohou probíhat současně. Podmínka, že dvě operace nad jedním objektem ve sdíleném datovém prostoru nemohou probíhat současně, odpovídá většině běžných implementací sdíleného datového prostoru a není příliš omezující.

Nejprve budeme definovat dvě základní operace nad sdíleným datovým prostorem - operace pro čtení a zápis řetězce bitů (hodnoty) objektu ve sdíleném datovém prostoru. Tyto dvě operace budeme nazveme *read* a *write*. Jejich zápis má tvar:

$$read\ (id_i,\ P_j,\ t_k)$$
$$write\ (id_i,\ P_j,\ t_k)$$

Tento zápis má následující význam: proces P_j čte (zapisuje) hodnotu objektu id_i (tj. řetězec bitů $id_i.object$) v diskrétním čase t_k. Operace *read* pouze čte hodnotu $id_i.object$ (nebo její část) bez jakékoli modifikace datového prostoru. Operace *write* modifikuje hodnotu $id_i.object$. Nová hodnota $id_i.object$ však nemusí mít stejnou délku jako hodnota původní.

Detailnější popis operací *read* a *write* zde nebudeme uvádět. Tyto dvě operace nebudeme později používat v žádné sémantické konstrukci. Pro účely popisu sémantiky budou později definovány jiné základní operace, u kterých bude uveden přesný sémantický popis. Operace *read* a *write* použijeme pouze pro vyjádření *chování* těchto operací.

Pokud budeme potřebovat označit jakoukoli operaci s datovým prostorem, bez ohledu na to, o kterou konkrétní operaci se jedná, použijeme označení

```
op (idi, Pj, tk)
```

Reference interakčního bodu

Pro další práci bude užitečné, abychom věděli, které procesy se odkazují na konkrétní objekt v datovém prostoru. Pro získání této informace budeme definovat funkci

```
ref (idi, operation)
```

která vrací množinu procesů, které obsahují ve svém těle operaci *operation*, která se odkazuje na objekt id_i. Touto operací *operation* může být operace *read*, *write* nebo jakákoli jiná operace nad sdíleným datovým prostorem, definovaná dříve. Pokud budeme chtít zjistit pouze počet procesů, jež se odkazují na konkrétní objekt ve sdíleném datovém prostoru, použijeme výraz

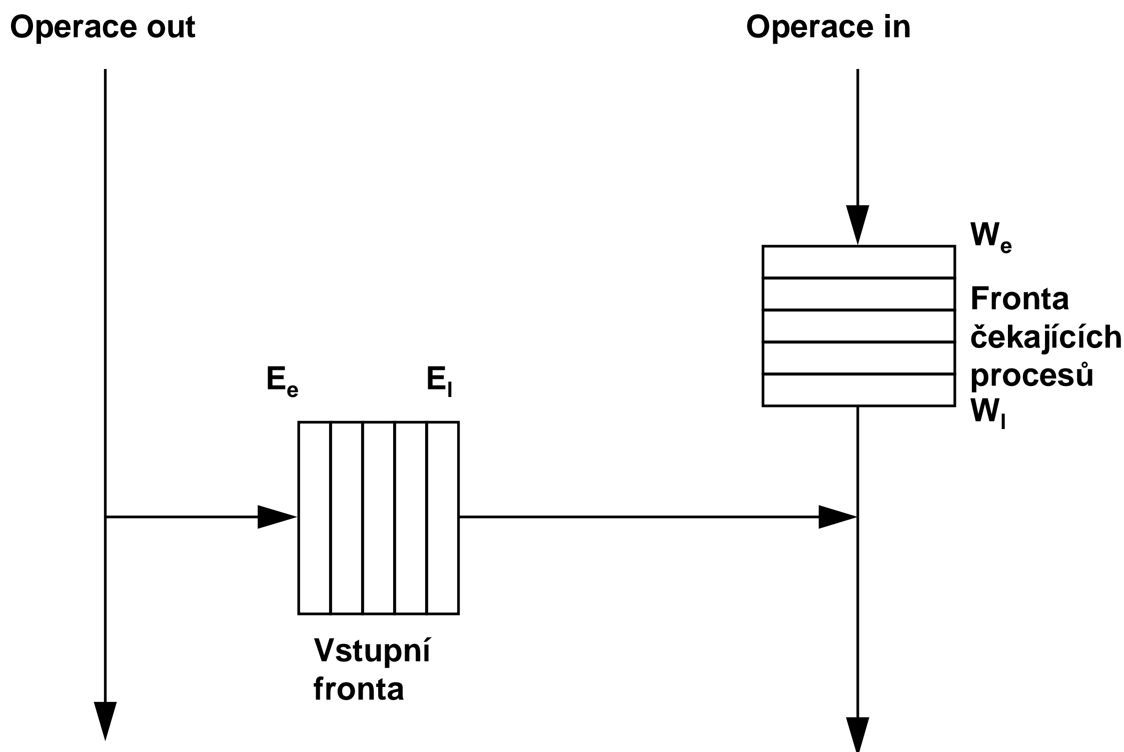
```
|ref (idi, operation)|
```

jehož hodnotou je počet procesů, jež se odkazují na objekt id_i prostřednictvím operace *operation* (notace /*S*/ značí kardinalitu množiny *S*).

3.4 Interakční bod

V následujícím textu se pokusíme definovat strukturu a chování základního objektu sdíleného datového prostoru, který budeme používat a který nazveme *interakční bod*. Poté ukážeme, jak je možné pomocí tohoto interakčního bodu (tedy s využitím modelu interakce se sdíleným datovým prostorem) modelovat chování běžně používaných interakčních mechanismů.

Základním objektem sdíleného datového prostoru, který budeme používat, je struktura, zvaná *interakční bod* (interaction point). Struktura interakčního bodu je znázorněna na Obr. 4. Interakční bod se skládá ze dvou front - ze *vstupní fronty* (input queue) a z *fronty čekajících procesů* (waiting queue). Pod pojmem fronta v této souvislosti rozumíme datovou abstrakci, která se často nazývá FIFO (First In First Out). Obě tyto fronty mají obecně neomezenou délku. Vstupní fronta obsahuje data, která byla vložena do interakčního bodu pomocí operace *out*. Fronta čekajících procesů obsahuje procesy (nebo spíše nějaký druh deskriptorů procesů), které provedly operaci *in* a které nyní čekají na dokončení této operace.



Obr. 4. Struktura interakčního bodu

Fronta čekajících procesů obsahuje pouze posloupnost čekajících procesů. Z hlediska použitého modelu PRAM neobsahuje žádná data. Jedná se o běžnou frontu FIFO bez priorit.

Vstupní fronta obsahuje data, která budeme nazývat *datové elementy* nebo zkráceně *elementy*. Pro naše účely budou datové elementy bitové řetězce o konečné a pevné délce. Délka datových elementů je jedním z důležitých atributů interakčního bodu, se kterým budeme dále pracovat. Proto definujeme funkci

```
elemwidth (idi)
```

která vrací číselnou hodnotu délky datového elementu, který může být uložen v interakčním bodu id_i. Tato velikost může být obecně v jakýchkoli jednotkách (například v bitech nebo v bajtech).

Pro práci s interakčním bodem jsou definovány dvě nové operace - operace *in* a *out*. Operace *in* má formu

```
in (idi, Pj, tk, Dl)
```

kterou je možno interpretovat takto: proces P_j čte data (tj. provádí vstupní operaci) z interakčního bodu id_i v diskrétním okamžiku t_k. Výsledkem operace *in* je hodnota D_l, která obsahuje načtená data. Přestože operace *in* provádí čtení dat, tato operace také modifikuje obsah interakčního bodu id_i.

Sémantika operace *in* je následující: Pokud při provádění operace *in* (id_i, P_j, t_k, D_l) je vstupní fronta interakčního bodu id_i neprázdná, je z této vstupní fronty vyjmut první element a jeho hodnota je vrácena jako návratová hodnota D_l operace *in*. Pokud vstupní fronta je prázdná, proces P_j je pozastaven a jeho deskriptor je vložen na konec fronty čekajících procesů interakčního bodu id_i.

Operace *in* je blokující. To znamená, že proces, provádějící operaci *in* může být pozastaven až do doby, než jsou požadovaná data ve vstupní frontě k dispozici.

Druhou definovanou operací je operace *out*. Tato operace má notaci

```
out (idi, Pj, tk, Dl)
```

která má následující význam: proces P_j provádí v diskrétním okamžiku t_k zápis datového elementu D_l do interakčního bodu id_i. Operace *out* modifikuje obsah interakčního bodu id_i.

Sémantiku operace *out* je možno neformálně popsat takto: datový element D_l je připojen na konec vstupní fronty interakčního bodu id_i. Pokud je fronta čekajících procesů interakčního bodu id_i prázdná, je operace *out* ukončena. Pokud tato fronta čekajících procesů není prázdná, je ze začátku této fronty vyjmut proces P_w, který je opět aktivován. Tento proces byl pozastaven proto, že nemohl dokončit operaci *in*. Nyní už má k dispozici data ve vstupní frontě a může operaci *in* dokončit.

Operace *out* je neblokující, to znamená, že proces, provádějící tuto operaci není nikdy pozastaven a tato operace se vždy ihned úspěšně ukončí.

V dalším textu se budeme zabývat obsahem obou front interakčního bodu v různých okamžicích provádění paralelního programu. K tomu použijeme některé z následujících statistických charakteristik front interakčního bodu id_i:

- E_e - označuje celkový počet položek, které byly od počátku existence interakčního bodu vloženy do vstupní fronty
- E_l - označuje celkový počet položek, které byly od počátku existence interakčního bodu vyjmuty ze vstupní fronty
- E_c - označuje počáteční počet položek, které byly v rámci inicializace vloženy do vstupní fronty interakčního bodu
- E_a - označuje počet položek, které jsou v tomto okamžiku ve vstupní frontě interakčního bodu. Platí, že $E_a = E_c + E_e - E_l$
- W_e - označuje celkový počet položek, které byly od počátku existence interakčního bodu vloženy do fronty čekajících procesů
- W_l - označuje celkový počet položek, které byly od počátku existence interakčního bodu vyjmuty z fronty čekajících procesů

- W_a - označuje počet položek, které jsou v tomto okamžiku ve frontě čekajících procesů interakčního bodu. Platí, že $W_a = W_e - W_l$

Nyní ještě několik slov o počátečním stavu interakčního bodu. Fronta čekajících procesů interakčního bodu je na počátku jeho existence prázdná. Neexistuje žádná možnost (a ani žádný důvod) jak naplnit při inicializaci tuto frontu. Vstupní fronta obecně může být při inicializaci naplněna E_c datovými elementy. Důvodem proto toto počáteční naplnění je to, že některé interakční mechanismy (například semafor) požadují, aby bylo možno provést tuto inicializaci vstupní fronty. Počet datových elementů, které byly při inicializaci vloženy do vstupní fronty interakčního bodu id_i budeme označovat notací *id.init*.

Výše uvedené statistické charakteristiky interakčního bodu E_a a W_a vyjadřují *okamžitý* počet položek ve vstupní frontě a ve frontě čekajících procesů. Nás však budou zajímat dvě jiné charakteristiky interakčního bodu - bude nás zajímat *maximální* počet položek v těchto frontách. Proto nadefinujeme další dva důležité atributy interakčního bodu, kterými budou maximální délka vstupní fronty a maximální délka fronty čekajících procesů. Pro jejich vyjádření použijeme následující dvě funkce

```
maxentry (idi)
```

a

```
maxwaiting (idi)
```

které vrací maximální délku obou front interakčního bodu id_i. Je zřejmé, že pro všechny možné hodnoty E_a and W_a musí platit vztahy

$$\text{maxentry}(id_i) >= E_a$$
$$\text{maxwaiting}(id_i) >= W_a$$

Minimální možná hodnota funkce *maxentry* (id_i) je rovna 1. Interakční bod, který by měl nulovou kapacitu vstupní fronty nemá žádný sémantický význam, protože podle definice operace *out* není možno operaci *out* nad tímto bodem provést. Maximální možná hodnota funkce *maxentry* (id_i) není podle definice interakčního bodu limitována.

Minimální možná hodnota funkce *maxwaiting* (id_i) je rovna 0 - pokud by vstupní fronta nebyla nikdy prázdná, nebylo by zde opodstatnění pro existenci fronty čekajících procesů. Maximální možná hodnota této funkce je rovna počtu procesů, které se odkazují na interakční bod id_i pomocí operace *in*. Formálně lze tedy maximální možnou hodnotu funkce *maxwaiting* (id_i) vyjádřit jako

$$|\text{ref}(id_i, \text{in})|.$$

Nyní jsme definovali vše, co budeme potřebovat pro zavedení pojmu *parametrizovaný interakční bod*. Parametrizovaný interakční bod je interakční bod id_i, který je charakterizovaný čtyřmi atributy:

```
elemwidth (id_i)
maxentry (id_i)
maxwaiting (id_i)
id.init (která se rovná hodnotě E_C)
```

Pomocí tohoto parametrizovaného bodu můžeme v následujících kapitolách vytvořit klasifikaci pro takzvané jednoduché interakční mechanismy a složené interakční mechanismy.

3.5 Příklady implementace interakčních mechanismů

Pro usnadnění pochopení následujícího textu uvedeme nyní některé příklady běžně používaných interakčních mechanismů, vyjádřených pomocí interakčních bodů a operací *in* a *out*. Pro každý interakční mechanismus uvedeme jeho jméno a jména jeho operací (v levém sloupci) a odpovídající implementaci pomocí interakčních bodů (v pravém sloupci).

Sdílená proměnná

```
změna hodnoty             in (Variable, ?Value)
                          out (Variable, NewValue)
```

Semafor

```
P (Sem1)                  in (Sem1)
V (Sem1)                  out (Sem1)
```

Asynchronní kanál (přihrádka)

```
Send (Chan1, Val)         out (Chan1, Val)
Recv (Chan1, Val)         in (Chan1, ?Val)
```

Synchronní kanál (přihrádka)

```
Send (Chan1, Val)         out (Chan1.1, Val)
                          in (Chan1.2)
Recv (Chan1, Val)         in (Chan1.1, ?Val)
                          out (Chan1.2)
```

Korespondence mezi operacemi semaforu P a V a jejich vyjádřeními pomocí operací *in* a *out* je zřejmá. V případě sdílené proměnné je naznačena atomická operace, která provádí modifikaci hodnoty sdílené proměnné. Tato modifikace je provedena přečtením původní hodnoty a bezprostředním zápisem hodnoty nové. Vzhledem ke vlastnostem operací *in* a *out* je tato operace atomická, tak jak je požadováno. Otazník, použitý v zápisu *in* (*Variable*, *?Value*) znamená, že parametr *Value* je (v pojmech programovacích jazyků) výstupním parametrem, jeho počáteční hodnota je bezvýznamná a po provedení operace *in* dostává novou hodnotu. Tato notace byla zvolena v souladu s notací, používanou v jazyce Linda.

Implementace asynchronního kanálu (a asynchronní schránky) je opět jednoduchá. Implementace synchronního kanálu (a synchronní schránky) se však výrazně liší od implementace asynchronních variant těchto mechanismů. V tomto případě pro implementaci jednoho synchronního kanálu (schránky) potřebujeme **dva** interakční body. Interakční mechanismus, pro jehož implementaci je třeba více než jeden interakční bod, budeme nazývat složený interakční mechanismus (na rozdíl od mechanismu, pro jehož implementaci postačuje jediný interakční bod a který budeme nazývat jednoduchý interakční mechanismus). Oba tyto druhy interakčních mechanismů detailněji popíšeme v následujících odstavcích.

3.6 Jednoduché interakční mechanismy

V této kapitole se pokusíme poněkud formálněji vyjádřit vztah mezi již dříve definovaným parametrizovaným interakčním bodem a reálnými interakčními mechanismy.

Máme-li k dispozici pouze jeden interakční bod, máme pouze jednu frontu čekajících procesů. To, znamená, že pomocí jednoho interakčního bodu jsme schopni modelovat pouze ty interakční mechanismy, které obsahují pouze jeden druh čekajících procesů. Příkladem takového interakčního mechanismu je asynchronní komunikační kanál. Tento kanál obsahuje pouze jeden druh čekajících procesů - procesy, čekající na čtení z kanálu.

Tyto interakční mechanismy, které je možno simulovat jediným interakčním bodem, budeme nazývat jednoduché interakční mechanismy. Opačným příkladem je složený interakční mechanismus. Je to takový interakční mechanismus, který může obsahovat dva (nebo více) druhy čekajících procesů. Příkladem složeného interakčního mechanismu je synchronní kanál. Ten může obsahovat dva druhy čekajících procesů - procesy, které čekají na čtení z kanálu a procesy, které čekají na zápis do kanálu. Složený interakční mechanismus musí být simulován více než jedním interakčním bodem. Složenými interakčními mechanismy se budeme zabývat později.

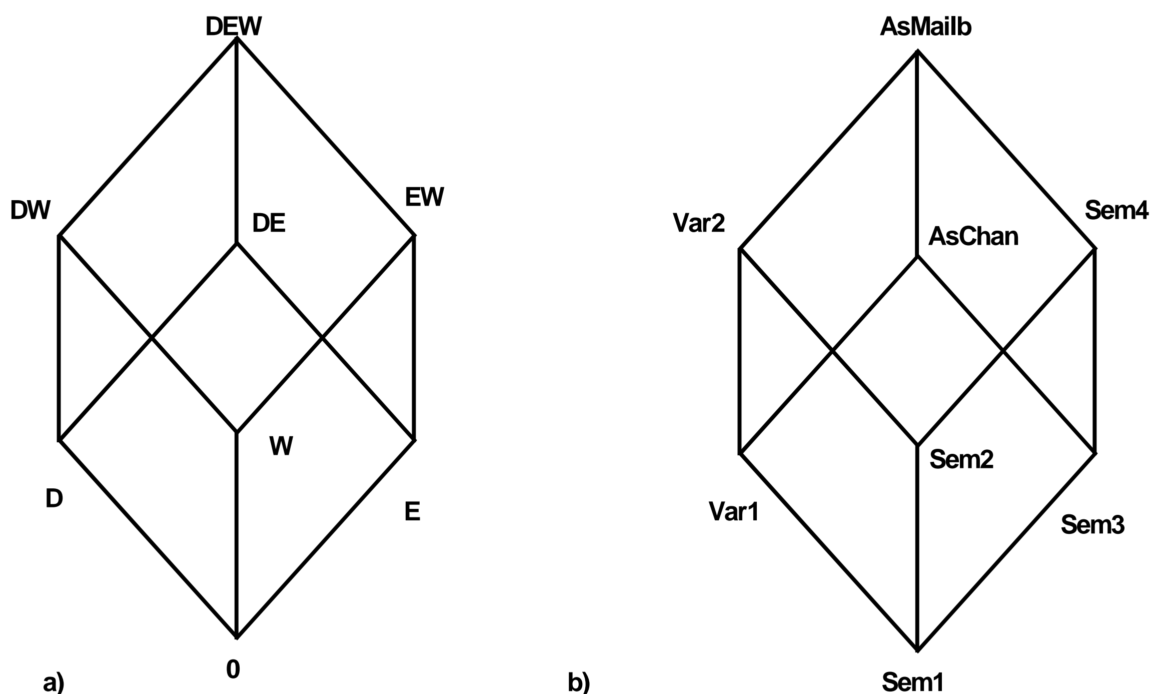
Nyní se pokusíme vytvořit klasifikaci jednoduchých interakčních mechanismů za pomoci parametrizovaného interakčního bodu. Z hlediska klasifikace jednoduchých interakčních mechanismů se musíme zabývat třemi důležitými otázkami:

- Provádí parametrizovaný interakční bod přenos dat, to znamená, platí *elemwidth* (id_i) >0?
- Používá parametrizovaný interakční bod vstupní frontu, tj. platí *maxentry* (id_i) >1?
- Používá parametrizovaný interakční bod frontu čekajících procesů, tj. platí *maxwaiting* (id_i) >1?

Na základě těchto tří kritérií můžeme definovat tři vlastnosti parametrizovaného interakčního bodu, které označíme D, E a W s následujícím významem:

```
D - elemwidth (idi) > 0
E - maxentry (idi) > 1
W - maxwaiting (idi) > 1
```

Intuitivně lze říci, že nejjednodušší parametrizovaný interakční bod (který budeme označovat symbolem 0), nebude mít žádnou z těchto vlastností a nejsložitější parametrizovaný interakční bod ((který budeme označovat symbolem DEW) má všechny tyto tři vlastnosti. Pomocí těchto tří můžeme rozlišit 8 různých typů jednoduchých interakčních mechanismů. Množinu těchto typů mechanismů je možno znázornit pomocí svazového diagramu - viz Obr. 5a. Pro snadnější porozumění je možné prvky tohoto svazu označit jmény typických interakčních mechanismů, které jsou reprezentanty tohoto typu. Je to však možné pouze u jednoduchých interakčních mechanismů, kde pro každý prvek svazu je skutečně možno nalézt reprezentanta. V případě složených interakčních mechanismů to už možné nebude, protože pro některé prvky svazu neexistuje název odpovídajícího interakčního mechanismu.



Obr. 5: Svazový diagram jednoduchých interakčních mechanismů

Pro popsání prvků svazu použijeme následující zkratky jmen interakčních mechanismů:

- *AsMailb* - asynchronní schránka s neomezenou kapacitou, používaná více než dvěma procesy
- *AsChan* - asynchronní kanál s neomezenou kapacitou, používaný právě dvěma procesy
- *Sem* - semafor

* *Var* - sdílená proměnná

Výsledný svaz, označený jmény jednoduchých interakčních mechanismů je znázorněn na Obr. 5b. Jména interakčních mechanismů musela být doplněna číslováním (např. Sem1, Sem2 atd.), protože naše klasifikace je podrobnější než slovní popis. Například prvek *Var1* (s vlastností D) označuje sdílenou proměnnou, která je používána právě dvěma procesy, zatímco *Var2* (s vlastností DW) označuje sdílenou proměnnou, která je používána více než dvěma procesy. Zachycení rozdílu mezi *Var1* a *Var2* je však důležité, protože mechanismus *Var1* může být implementován mnohem jednodušeji než mechanismus *Var2*.

Z implementačního hlediska je důležité, abychom byli schopni určit maximální hodnoty funkcí *elemwidth*, *maxwaiting* a *maxentry* pro každý použitý interakční bod. Pokud například víme, že maximální hodnota funkce *elemwidth* je rovna nule, můžeme při implementaci zcela vynechat vstupní frontu interakčního bodu a tuto vstupní frontu můžeme nahradit pouhým čítačem, který obsahuje počet elementů, uložených v této frontě. Podobně, je-li maximální hodnota funkce *maxentry* nebo *maxwaiting* rovna jedné, můžeme při implementaci nahradit vstupní frontu nebo frontu čekajících procesů jedinou sdílenou proměnnou. A i tehdy, pokud maximální hodnota funkce *maxentry* nebo *maxwaiting* není rovna jedné, ale je omezená (později ukážeme, že ve většině případů jsou tyto hodnoty skutečně omezené) může informace o maximální délce zjednodušit alokaci paměti a výrazně zvýšit efektivnost konkrétní implementace interakčního bodu.

Zjištění hodnoty funkce *elemwidth* je velmi jednoduché. Tuto hodnotu lze zjistit pro každý interakční bod pouhou statickou analýzou zdrojového textu všech procesů. Zjištění hodnot funkcí *maxentry* a *maxwaiting* je však mnohem obtížnější. Proto se touto problematikou budeme zabývat odděleně v kapitole 4 této práce.

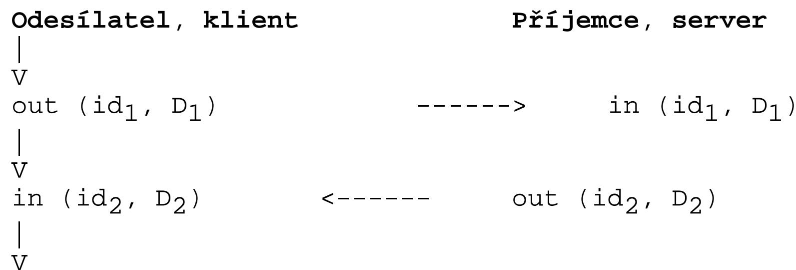
3.7 Složené interakční mechanismy

V předchozích odstavcích jsme se zabývali pouze těmi interakčními mechanismy, které bylo možno simulovat jedním interakčním bodem. Nyní se budeme zabývat tou skupinou, pro jejíž modelování je třeba více interakčních bodů. Tyto mechanismy budeme nazývat složené interakční mechanismy. Pro zjednodušení začneme popisem mechanismů, pro jejichž simulaci jsou třeba právě dva interakční body. Tyto interakční mechanismy mají následující vlastnosti:

* Skládají se ze dvou interakčních bodů id_1 a id_2.
* Zatímco u jednoduchých interakčních mechanismů může existovat pouze jeden druh čekajících procesů, u složených interakčních mechanismů jsou dva druhy čekajících procesů. V názvosloví paralelních a distribuovaných systémů se členové prvního druhu nazývají *odesílatel* nebo *klient* a členové druhého druhu *příjemce* nebo *server*.

Typická implementace primitivní operace, prováděné se složeným interakčním bodem se skládá ze dvou operací s interakčním bodem (z jedné operace *in* a z jedné

operace *out*). Pořadí těchto dvou operací závisí na tom, zda se jedná o proces odesílatel (klient) nebo o proces příjemce (server). Implementace je znázorněna na následujícím obrázku:



Lze dokázat, že pro složený interakční bod musí platit následující rovnosti:

$ref\ (id_1, out) = ref\ (id_2, in)$ (1)
$ref\ (id_1, in) = ref\ (id_2, out)$ (2)
$maxwaiting\ (id_1) = |ref\ (id_1, in)|$ (3)
$maxwaiting\ (id_2) = |ref\ (id_2, in)|$ (4)
$maxentry\ (id_1) = maxentry\ (id_2)$ (5)

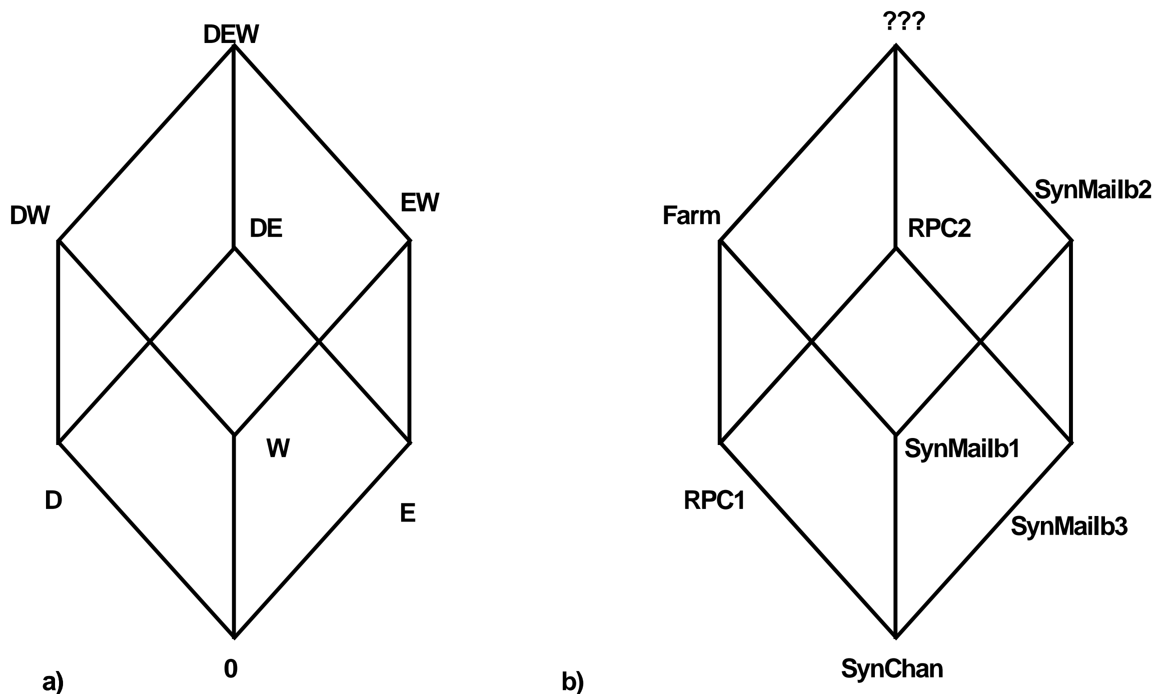
Stejně jak v případě jednoduchého interakčního bodu vytvoříme klasifikaci složených interakčních mechanismů, simulovaných pomocí dvou interakčních bodů. Z hlediska klasifikace jsou pro nás zajímavé tyto charakteristiky:

- Provádí interakční mechanismus přenos dat v obou směrech, tj. je ($elemwidth\ (id_1)>0$) a ($elemwidth\ (id_2)>0$)?
- Používá interakční mechanismus vstupní fronty, tj. je $maxentry\ (id_1) >1$?
- Používá interakční mechanismus frontu čekajících procesů interakčního bodu id_1, tj. je $maxwaiting\ (id_1) >1$?

Na základě těchto tří otázek definujeme následující tři vlastnosti složeného interakčního mechanismu, které označíme D, E a W:

D - $(elemwidth\ (id_1)>0)$ and $(elemwidth\ (id_2)>0)$
E - $maxentry\ (id_1) = maxentry\ (id_2) > 1$
W - $maxwaiting\ (id_1) > 1$

S pomocí těchto tří vlastností jsme schopni rozlišit 8 různých typů složených interakčních mechanismů. Množinu těchto typů mechanismů je možno opět znázornit pomocí svazového diagramu - viz Obr. 6a. Pro snadnější porozumění je opět možné alespoň některé prvky tohoto svazu označit jmény typických interakčních mechanismů, které jsou reprezentanty tohoto typu.



Obr. 6: Svazový diagram složených interakčních mechanismů

Pro pojmenování prvků svazu byly použity následující zkratky složených interakčních mechanismů:

- *SynMailb* - synchronní schránka s omezenou kapacitou, použitá více než dvěma procesy
- *SynChan* - synchronní kanál s omezenou kapacitou, použitý právě dvěma procesy
- *RPC* - volání vzdálených procedur (RPC, Remote Procedure Call)
- *Farm* - farma procesů
- *Barrier* - synchronizační bariéra

Výsledný svaz se jmény složených interakčních mechanismů je znázorněn na Obr. 6b. Pro rozlišení musela být jména interakčních mechanismů opět doplněna číslováním, protože naše klasifikace je podrobnější než slovní popis. Pro interakční mechanismus s vlastností DEW jsme dokonce nenalezli žádné reálné jméno.

3.8 Nalezení složených interakčních mechanismů

Pro identifikaci složených interakčních mechanismů (definovaných v předchozí kapitole) je třeba umět nalézt odpovídající dvojici interakčních bodů id_1 a id_2, ze kterých

se složený interakční mechanismus skládá. Tuto dvojici interakčních bodů je možno nalézt analýzou grafu toku řízení analyzovaného programu. V tomto grafu toku řízení je třeba nalézt ty dvojice interakčních bodů, které jsou procesy analyzovaného programu použity podle popisu v předchozí kapitole. Detailní popis algoritmu nalezení složených interakčních bodů se však vymyká zaměření této práce.

3.9 Interakční mechanismy s více interakčními body

Mimo zmiňované složené interakční mechanismy se dvěma interakčními body mohou existovat interakční mechanismy, pro jejichž modelování jsou třeba více než dva interakční body. Příkladem takového interakčního mechanismu může být podmíněná kritická oblast. Tyto interakční mechanismy však bývají pouze sémantickými konstrukcemi, pod nimiž se skrývá některý z jednodušších interakčních mechanismů, kterým je v tomto případě semafor. Tyto interakční mechanismy lze tedy převést na interakční mechanismy jednodušší a v praxi se pro implementaci těchto složitějších interakčních mechanismů skutečně používají právě tyto jednodušší interakční mechanismy. Proto se těmito interakčními mechanismy dále nebudeme podrobněji zabývat.

4. OPTIMALIZACE IMPLEMENTACE INTERAKČNÍHO BODU

V této části práce se budeme zabývat problematikou implementace interakčních mechanismů, popsaných pomocí výše zmíněného modelu. Vzhledem k tomu, že použitý model je velmi obecný, byla by naivní implementace (naivní implementací rozumíme přímočarou implementaci bez jakékoli optimalizace) mechanismů, popsaných tímto modelem, velmi obtížná a v některých případech až nemožná. Proto nyní popíšeme metody optimalizace, které mohou pomoci při implementaci mechanismů, popsaných tímto modelem.

Popisovaná optimalizace se bude zabývat problémem, který je při implementaci interakčních mechanismů s frontami značně kritický, a tím je optimalizace délek front interakčního bodu.

Znalost či neznalost maximálních délek front interakčního bodu může významně ovlivnit efektivitu výsledné implementace interakčního bodu. Pokud maximální délky front interakčního bodu nejsou při implementaci známy, je třeba dle definice interakčního bodu předpokládat, že maximální délky front mohou být nekonečné. Reálná implementace tedy musí být schopna implementovat interakční bod takovým způsobem, který dovolí narůstání délek front interakčního bodu bez jakýchkoli omezení, až do vyčerpání všech dostupných (paměťových) prostředků. Taková implementace musí značně využívat dynamické struktury, které jsou méně efektivní než struktury statické.

V případě, kdy známe maximální délky front interakčního bodu, je situace mnohem lepší. V tom případě jsme schopni pro implementaci interakčního bodu použít statické datové struktury, které jsou mnohem efektivnější.

V reálných programech nastávají často případy, kdy maximální délka některé fronty není větší než jedna. V takovém případě je implementace interakčního bodu ještě efektivnější, protože se velmi zjednodušuje implementace operací *in* a *out*.

Proto cílem následující části této práce bude problematika výpočtu maximálních délek front interakčního bodu.

4.1 Výpočet délek front interakčního bodu

Předmětem naší analýzy bude paralelní (nebo distribuovaný) program, který se skládá z libovolného počtu paralelně běžících procesů. Tyto procesy spolu interagují pouze pomocí sdíleného datového prostoru a to pouze prostřednictvím operací *in* a *out*. V okamžiku analýzy jsou známy zdrojové texty všech procesů, ze kterých se analyzovaný paralelní program skládá.

Analýzu provádíme odděleně pro každý interakční bod, který tvoří jednoduchý interakční mechanismus nebo pro každou dvojici interakčních bodů, která tvoří složený

interakční mechanismus. V následujícím textu bude vždy popisována analýza pouze pro jeden interakční mechanismus.

Uvažujeme-li tedy jeden interakční mechanismus, lze v logickou úvahou dojít ke vzorcům pro výpočet maximálních délek front odpovídajícího interakčního bodu (bodů). V případě jednoduchého interakčního bodu lze hodnoty funkcí *maxwaiting* a *maxentry* spočítat podle následujících vzorců:

```
maxentry (id)    = id.init + Σ Uo (Pj, id)
                             Pj ∈ ref(id, out)

maxwaiting (id)  =
    min (|ref (id, in)|, Σ Ui (Pj, id) - id.init)
                         Pj ∈ ref(id, in)
```

Pro složený interakční mechanismus platí následující vzorce pro výpočet hodnot funkcí *maxwaiting* a *maxentry*:

```
maxwaiting (id1)  = |ref (id1, in)|

maxwaiting (id2)  = |ref (id2, in)|

maxentry (id1) = maxentry (id2) = id1.init + id2.init +
= Σ Uo (Pj, id12)     + Σ Uo (Pj, id12)
  Pj ∈ ref(id1, out)    Pj ∈ ref(id2, out)
```

Funkce $U_i(P, id)$ a $U_o(P, id)$ jsou takzvané funkce *unbalance*. Tyto funkce, intuitivně řečeno, vyjadřují "příspěvek procesu *P* k požadavkům na délku vstupní fronty (funkce U_o) nebo fronty čekajících procesů (funkce U_i) interakčního bodu *id"*. Zápis U_o (P, id_{12}), použitý v případě složeného interakčního bodu znamená, že hodnota U_o je vztažena ke dvojici interakčních bodů id_1 a id_2, které tvoří složený interakční mechanismus.

Pokud například proces *P* obsahuje pouze jednu operaci *in*, následovanou jednou operací *out*, jde o proces, který požaduje jednu položku ve frontě čekajících procesů (první operace *in* může být blokující a proces může být pozastaven) a nepožaduje žádnou položku ve vstupní frontě (proces sice provádí operaci *out*, která vkládá jednu položku do vstupní fronty, ale před tím provedl operaci *in*, která jednu položku ze vstupní fronty odebrala). Pro tento proces je tedy hodnota U_i=1 a U_o=0.

Pokud proces obsahuje jednu operaci *out*, následovanou jednou operací *in*, jde o proces, který požaduje jednu položku ve vstupní frontě (první operace *out* vkládá jednu položku do vstupní fronty) a nepožaduje žádnou položku ve frontě čekajících procesů (proces sice provádí operaci *in*, která by mohla být blokující, ale před tím provedl operaci *out*, která zajišťuje, že operace *in* blokující nebude). Pro tento proces je tedy hodnota U_i=1 a U_o=0.

Výše uvedené vzorce byly vytvořeny logickou úvahou na základě zvážení požadavků jednotlivých procesů na délky front interakčního bodu. K formálnímu odvození těchto

vzorců je potřeba mít k dispozici aparát, který si vytvoříme později. Pak tyto vzorce odvodíme exaktně.

Největším problémem při výpočtu hodnot funkcí *maxentry* a *maxwaiting* je výpočet hodnot funkcí U_i a U_o. Pokud se na každý paralelní proces P_i díváme izolovaně, pak hodnoty funkcí U_i a U_o představují maximální požadavky procesu P_i na frontu čekajících procesů (resp. na vstupní frontu). Hodnoty těchto funkcí je tedy možno zjistit tak, že pro proces P_i nalezneme všechny možné sekvence provádění operací *in* a *out* a pro každou sekvenci nalezneme její požadavky na délky front interakčního bodu. Požadavky sekvence s největšími požadavky jsou pak maximálními požadavky celého procesu. Výpočet hodnot funkcí U_i a U_o je tedy možno vyjádřit pomocí následujících vzorců:

$$U_o(P,\ id) = \underset{*}{MAX}\left(\underset{i=1}{MAX}\left(\sum_{j=1} \begin{cases} \\ +1 & \text{if } op_j = out \end{cases}\right)\right)$$

$$U_i(P,\ id) = \underset{*}{MAX}\left(\underset{i=1}{\overset{K}{MAX}}\left(\sum_{j=1}^{i} \begin{cases} +1 & \text{if } op_j = in \\ -1 & \text{if } op_j = out \end{cases}\right)\right)$$

* maximum přes všechny sekvence operací $op_1 \ldots op_K$ s interakčním bodem *id* v procesu *P*

Výše uvedené vzorce jsou sice správné, ale prakticky bezcenné. Je velmi obtížné spočíst první MAXimum v těchto vzorcích, protože se jedná o maximum ze všech sekvencí operací $op_1 ... op_k$ s interakčním bodem *id* v procesu *P*. Tento výpočet je velmi obtížný už jen pro to, že v okamžiku analýzy nemusíme znát počet provedení cyklů v procesu a tedy počet všech možných sekvencí operací $op_1 ... op_k$ je nekonečný.

Jedno z možných řešení tohoto problému je následující: ze zdrojového textu procesu vytvoříme pro každý interakční bod odpovídající graf toku řízení. V tomto grafu toku řízení nás budou zajímat pouze řídící struktury a operace *in* a *out*. na tomto grafu toku řízení je možno provádět tzv. invariantní transformace. Invariantní transformace grafu toku řízení je taková transformace grafu, která neovlivňuje hodnoty funkcí *unbalance*. Příkladem takové invariantní transformace může být následující transformace

```
in
out => in
in
```

která redukuje graf toku řízení, ale zachovává hodnoty funkcí *unbalance*. Pomocí těchto invariantních transformací odstraníme z grafu toku řízení cykly a maximálně graf toku řízení zjednodušíme (nejlépe až do podoby grafu se dvěma vrcholy a s jednou hranou). Z výsledného grafu toku řízení pak snadno spočítáme hodnoty funkcí *unbalance*.

V následujících odstavcích tento poměrně vágně popsaný postup popíšeme formálněji.

4.2 Použitá notace a operace

Při analýze maximálních délek front interakčního bodu vyjdeme z grafu toku řízení programu, pro který analýzu provádíme. Při vytváření grafu toku řízení se provede nejdříve rozdělení každého procesu na skupiny příkazů, zvané *základní bloky*. Základní blok je maximální skupina příkazů, pro kterou platí:

- řízení přechází pouze na první příkaz této skupiny
- je-li řízení předáno na první příkaz této skupiny, všechny příkazy v základním bloku jsou vykonány sekvenčně

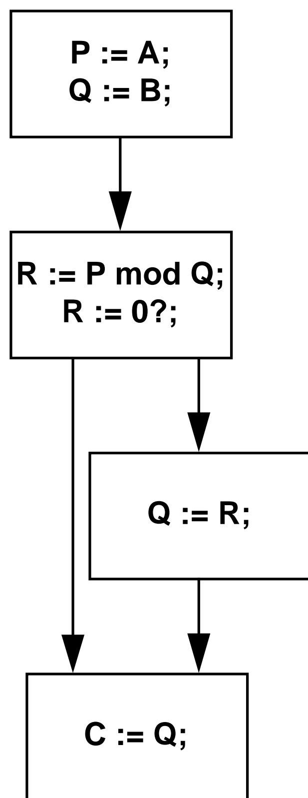
Ze základních bloků každého procesu pak vytvoříme orientovaný graf, který představuje tok řízení procesu. Každý vrchol tohoto grafu odpovídá základnímu bloku. Pokud existuje hrana z bloku *x* do bloku *y*, pak při provádění programu může být řízení předáno z bloku *x* do bloku *y*. Tento graf se nazývá *graf toku řízení*. Tento graf má *počáteční vrchol*, ve kterém začíná vykonávání programu a *terminální vrchol*, ve kterém končí vykonávání programu.

Tato podoba grafu toku řízení však není pro náš účel vhodná, protože v ní lze obtížně vyjádřit řídící konstrukce, jako je paralelní provedení, podmíněné provedení atd. Proto ji v následujícím textu upravíme na tzv. graf sekvencí. Graf sekvencí se od grafu toku řízení liší tím, že základní bloky nejsou vrcholy, nýbrž hrany grafu. Vrcholy grafu pak vyjadřují řídící konstrukce. Tyto řídicí konstrukce jsou sekvenční provedení dvou základních bloků (vrchol se neoznačuje), paralelní provedení základních bloků (vrchol se označuje PAR), podmíněné provedení základních bloků (vrchol se označuje COND), alternativní základních bloků (vrchol se označuje ALT), počítaná iterace základních bloků (vrchol se označuje ITERN), a nepočítaná iterace základních bloků (vrchol se označuje ITER). Vztah mezi grafem toku řízení a grafem sekvencí ukážeme na příkladě na následujícím obrázku. Na tomto obrázku je zdrojový text program a), jeho vyjádření grafem toku řízení b) a jeho vyjádření grafem sekvencí c).

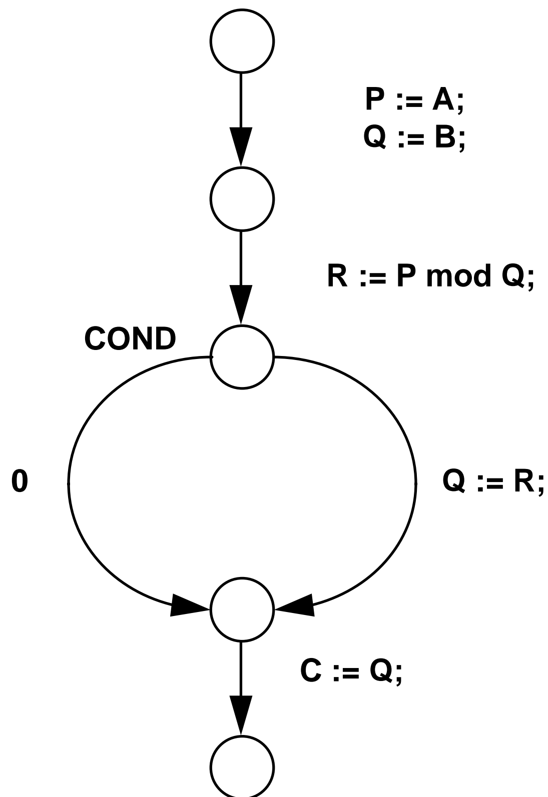
```
P:=A;
Q:=B;

R := P mod Q;
if R = 0 then
 begin
 P := Q;
 Q := R;
 end;
C:=Q;
```

a)



b)



c)

Obr. 7: Příklad grafu sekvencí

4.2.2 Sekvence a její vlastnosti

V dalším textu budeme používat speciální typ podgrafu grafu sekvencí, který budeme nazývat sekvence. *Sekvence* je souvislým podgrafem grafu sekvencí. Sekvence obsahuje dva významné vrcholy - vrchol počáteční (*source*) a vrchol koncový (*sink*). Z počátečního vrcholu hrany pouze vystupují a do koncového vrcholu hrany pouze vstupují. Každý z ostatních vrcholů v sekvenci obsahuje alespoň jednu vstupující hranu a alespoň jednu vystupující hranu. Nejjednodušší sekvence obsahuje pouze dva vrcholy (počáteční a koncový) a jednu hranu, která tyto vrcholy spojuje.

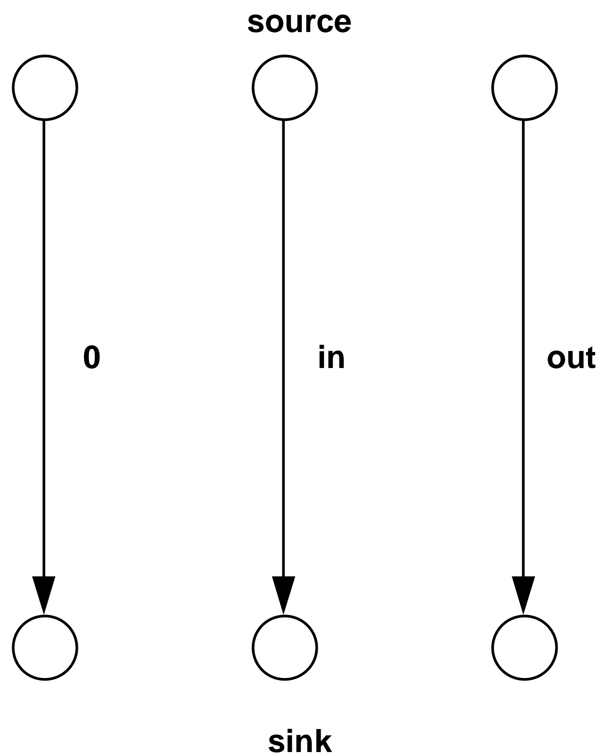
Hranami sekvence jsou opět jiné sekvence. Hranami sekvence mohou být buď elementární sekvence nebo obecné sekvence. Elementární sekvence odpovídá v grafu toku řízené čistě sekvenčnímu provedení libovolného množství akcí (tj. základnímu bloku), v nichž se:

a) Nenachází žádná operace *in* ani operace *out*. Pak se jedná nulovou elementární sekvenci, kterou budeme označovat *0*.

b) Nachází pouze jediná operace *in*. Tuto elementární sekvenci budeme označovat *in*.

c) Nachází pouze jediná operace *out*. Tuto elementární sekvenci budeme označovat *out*.

Grafy elementárních sekvencí jsou znázorněny na Obr. 8.



Obr. 8: Elementární sekvence

Hranou sekvence může být také jiná, obecná sekvence. Tuto obecnou sekvenci je možno označovat třemi způsoby:

a) Pokud jsme tuto sekvenci již dříve pojmenovali, můžeme ji nazývat symbolickým jménem.

b) Pokud je tato sekvence natolik jednoduchá, že se skládá pouze ze sekvenčního provedení několika operací *in* a *out*, můžeme ji označovat pomocí seznamu těchto operací. Například sekvenci, ve které se pouze sekvenčně provádí operace *in*, operace *out* a opět operace *in* můžeme označit *[in, out, in]*.

c) Pokud se jedná o složitější sekvenci a zajímají nás pouze její vlastnosti, můžeme ji označit uspořádanou čtveřicí (trojicí) jejích vlastností (viz dále).

Vrcholy sekvence obsahují informace, které běžně obsahují vrcholy grafů toku řízení. Jde o informace, které identifikují řídící struktury, ze kterých graf toku řízení vznikl (například sekvenční provedení, paralelní provedení, iteraci, podmíněné provedení atd.). Popisy těchto informací budou uvedeny později, až u popisů konkrétních řídících struktur.

Důležitou informací o sekvenci jsou její vlastnosti. Tyto vlastnosti nás budou zajímat jako výsledek analýzy těchto sekvencí budeme s nimi pracovat i při skládání sekvencí. *Vlastnosti* sekvence *S* budeme označovat *P*(*S*) a budeme je vyjadřovat jako následující uspořádanou čtveřici:

$P_S = (U_i, U_o, V_{min}, V_{max})$

kde

U_i je příspěvek sekvence k délce fronty čekajících procesů
U_o je příspěvek sekvence k délce vstupní fronty
V_{min} je minimální nesymetrie sekvence
V_{max} je maximální nesymetrie sekvence

U_i a U_o jsou takzvané funkce *unbalance*, které říkají, jak přispívají operace *in* a *out* obsažené v podgrafu P_S k požadavkům na frontu čekajících procesů (funkce U_i) a k požadavkům na vstupní frontu (funkce U_o). Vlastnosti V_{min} a V_{max} určují takzvanou nesymetrii sekvence. Nesymetrie je vlastnost sekvence, která říká, že v rámci sekvence není stejný počet operací *in* a operací *out* s daným interakčním bodem. Nesymetrie je vyjádřena celým číslem. Hodnota 0 znamená, že sekvence je symetrická (např. nulová sekvence nebo sekvence [*in*, *out*] má nulovou nesymetrii). Kladná hodnota znamená, že převažuje počet operací *out* (např. sekvence [*out*, *in*, *out*] má nesymetrii 1). Záporná hodnota znamená, že převažuje počet operací *in* (např. sekvence [*in*, *out*, *in*] má nesymetrii -1). Pokud sekvence neobsahuje podmíněné provedení, lze sekvenci vyjádřit jedinou hodnotou $V = V_{min} = V_{max}$. Obsahuje-li podmíněné provedení, jsou třeba pro její vyjádření dvě čísla V_{min} a V_{max}, která představují minimální a maximální hodnotu nesymetrie. V případě, že $V_{min} = V_{max}$, budeme pro zjednodušení vlastnosti podgrafu sekvence označovat jako uspořádanou trojici:

```
PS = (Ui, Uo, V)
```

Nyní je třeba definovat vlastnosti tří elementárních sekvencí - prázdnou sekvenci, označovanou *P*(*0*), sekvenci s jedinou operací *in*, označovanou *P*(*in*) a sekvenci s jedinou operací *out*, označovanou *P*(*out*). Pro tyto tři elementární sekvence platí:

```
P(0) = (0, 0, 0)
P(in) = (1, 0, -1)
P(out) = (0, 1, +1)
```

Hodnoty vlastností elementárních sekvencí vyplývají z definicí jednotlivých složek vlastností. Hodnoty vlastností složitějších sekvencí určíme pomocí skládání sekvencí jednodušších.

4.3 Skládání sekvencí

V následujících odstavcích si ukážeme, jaké vlastnosti má sekvence, která vznikne složením jednodušších sekvencí. Skládání sekvencí si ukážeme pro již dříve uvedené základní řídící konstrukce:

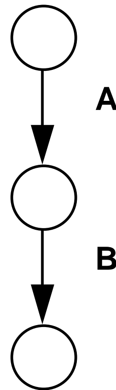
- sekvenční provedení dvou základních bloků (vrchol se neoznačuje)
- paralelní provedení základních bloků (vrchol se označuje PAR)
- podmíněné provedení základních bloků (vrchol se označuje COND)
- alternativní základních bloků (vrchol se označuje ALT)
- počítaná iterace základních bloků (vrchol se označuje ITERN)
- nepočítaná iterace základních bloků (vrchol se označuje ITER)

U každé z těchto konstrukcí si ukážeme výpočet vlastností U_i, U_o, V_{min} a V_{max}.

4.3.1 Sekvenční provedení

Sekvenční provedení je základním a nejčastějším typem skládání dvou sekvencí. Sekvenční provedení sekvencí *A* a *B* znamená, že nejdříve je provedena celá sekvence *A* a pak celá sekvence *B*. Běh sekvenčního provedení je ukončen, jakmile je ukončena sekvence *B*.

Graf sekvenčního provedení je znázorněn na následujícím obrázku. Na tomto obrázku je uveden příklad dvou sekvencí *A* a *B*, které jsou provedeny sekvenčně.



Obr. 9: Graf sekvenčního provedení

Nyní si ukážeme, jaké budou hodnoty vlastnosti (U_i, U_o, V_{min}, V_{max}) sekvenčního provedení dvou sekvencí *A* a *B*. Vzorce pro jejich výpočet jsou následující:

a) $U_i = \max(U_i(A), U_i(B) - V_{min}(A))$
b) $U_o = \max(U_o(A), U_o(B) + V_{max}(A))$
c) $V_{min} = V_{min}(A) + V_{min}(B)$
d) $V_{max} = V_{max}(A) + V_{max}(B)$

Nyní tyto vzorce zdůvodníme.

a) Pokud jsou požadavky sekvence A na frontu čekajících procesů větší než požadavky sekvence B, je výsledná hodnota U_i sekvenčního provedení obou sekvencí rovna požadavkům sekvence A, tj. hodnotě $U_i(A)$. Jsou-li požadavky sekvence A na frontu čekajících procesů menší než požadavky sekvence B, je výsledná hodnota U_i sekvenčního provedení obou sekvencí rovna požadavkům sekvence B, sníženým o minimální nesymetrii sekvence A (tj. pokud sekvence A přidala do vstupní fronty $V_{min}(A)$ položek, budou požadavky sekvence B na frontu čekajících procesů o tuto hodnotu nižší).

b) Totéž platí pro výpočet hodnoty U_o s tím rozdílem, že v případě, že požadavky sekvence B na vstupní frontu jsou větší než požadavky sekvence A, je výsledná hodnota U_i sekvenčního provedení obou sekvencí rovna požadavkům sekvence B, zvýšeným o maximální nesymetrii sekvence A (tj. pokud sekvence A přidala do vstupní fronty $V_{max}(A)$ položek, musí být požadavky sekvence B na frontu čekajících procesů o tuto hodnotu vyšší).

c) Minimální hodnoty nesymetrie dvou sekvencí se sčítají. Pokud např. sekvence A přidala do vstupní fronty minimálně $V_{min}(A)$ položek, a sekvence B přidala do vstupní

fronty minimálně $V_{min}(B)$ položek, je minimální počet položek přidaných do fronty oběma sekvencemi roven $V_{min}(A) + V_{min}(B)$.

d) Maximální hodnoty nesymetrie dvou sekvencí se opět sčítají. Viz vysvětlení k bodu c).

Pro ilustraci uvedeme dva příklady sekvenčního provedení dvou sekvencí.

Příklad 1:

Jsou-li vlastnosti elementárních sekvencí

```
P(in) = (1, 0, -1)
P(out) = (0, 1, +1)
```

pak vlastnosti sekvencí se dvěma operacemi s datovým prostorem jsou:

```
P ([in in ]) = ( 2, 0,-2,-2)
P ([in out ]) = ( 1, 0, 0, 0)
P ([out in ]) = ( 0, 1, 0, 0)
P ([out out ]) = ( 0, 2, 2, 2)
```

Příklad 2:

Jsou-li vlastnosti elementárních sekvencí

```
P(in) = (1, 0, -1)
P(out) = (0, 1, +1)
```

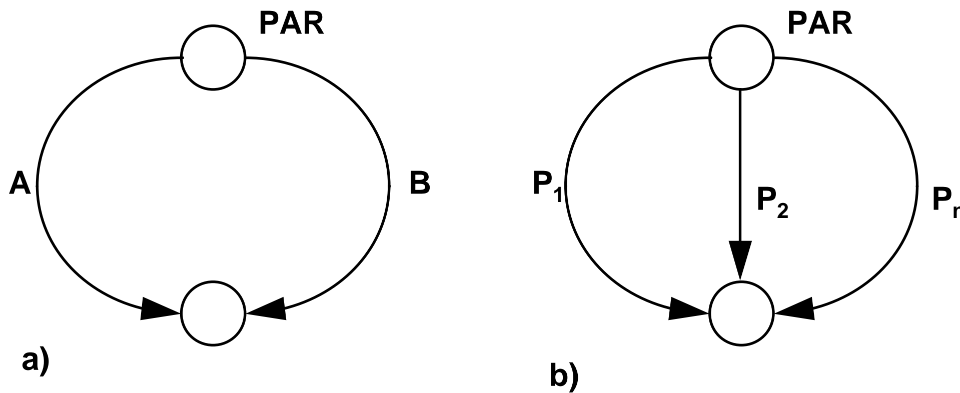
pak vlastnosti sekvencí se třemi operacemi s datovým prostorem jsou:

```
P ([in in in ]) = ( 3, 0,-3,-3)
P ([in in out]) = ( 2, 0,-1,-1)
P ([in out in ]) = ( 1, 0,-1,-1)
P ([in out out]) = ( 1, 1, 1, 1)
P ([out in in ]) = ( 1, 1,-1,-1)
P ([out in out]) = ( 0, 1, 1, 1)
P ([out out in ]) = ( 0, 2, 1, 1)
P ([out out out]) = ( 0, 3, 3, 3)
```

4.3.2 Paralelní provedení

Paralelní provedení je poměrně častým obratem, používaným v paralelních a distribuovaných programech. Paralelní provedení sekvencí *A* a *B* znamená, že obě dvě sekvence probíhají paralelně. Nelze vytvořit žádný předpoklad o tom, v jakém časovém vztahu budou akce, provedené v sekvenci *A* vzhledem k akcím, prováděným v sekvenci *B*. Běh paralelního provedení je ukončen, jakmile jsou ukončeny obě sekvence *A* i *B*.

Graf paralelního provedení je znázorněn na následujícím obrázku. Na tomto obrázku je uveden příklad dvou sekvencí *A* a *B*, které jsou provedeny paralelně.



Obr. 10: Graf paralelního provedení
a) Paralelní provedení dvou procesů
b) Paralelní provedení více procesů

V případě paralelního provedení dvou procesů se hodnoty vlastností spočítají následovně:

$$U_i = U_i(A) + U_i(B)$$
$$U_o = U_o(A) + U_o(B)$$
$$V_{min} = V_{min}(A) + V_{min}(B)$$
$$V_{max} = V_{max}(A) + V_{max}(B)$$

Pro všechny čtyři vlastnosti je hodnota výsledné vlastnosti spočtena jako součet vlastností obou paralelně prováděných sekvencí.

Častější než paralelní provedení dvou procesů bývá případ paralelního provedení *n* procesů. V tomto případě se hodnoty vlastností spočtou jako suma vlastností jednotlivých sekvencí:

$$U_i = \sum_{P_j \in ref(id,\ in)} U_i\ (P_j,\ id)$$

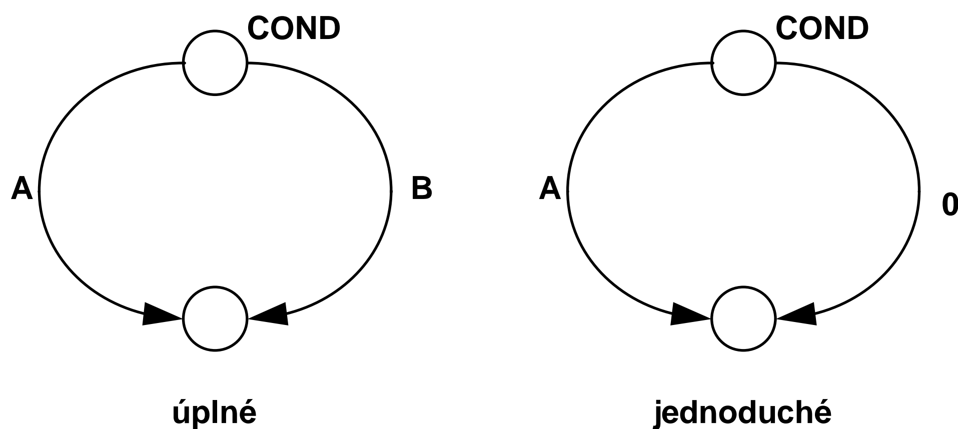
$$U_o = \sum_{P_j \in ref(id,\ out)} U_o\ (P_j)$$

$$V_{min} = \sum_{P_j \in ref(id,\ out)} V_{min}(P_j)$$

$$V_{max} = \sum_{P_j \in ref(id,\ out)} V_{max}(P_j)$$

4.3.3 Podmíněné provedení

Podmíněné provedení odpovídá podmíněnému příkazu v běžných programovacích jazycích. Rozeznáváme úplné podmíněné provedení a jednoduché podmíněné provedení. Úplné podmíněné provedení v případě splnění podmínky provede sekvenci *A* a v případě nesplnění podmínky provede sekvenci *B* (odpovídá jazykové konstrukci IF-THEN-ELSE). Jednoduché podmíněné provedení v případě splnění podmínky provede sekvenci *A* a v případě nesplnění podmínky neprovede žádnou sekvenci (a odpovídá jazykové konstrukci IF-THEN). Graf obou variant podmíněného provedení je znázorněn na následujícím obrázku.



Obr. 11: Graf podmíněného provedení

Pro výpočet vlastností úplného podmíněného provedení platí následující vzorce:

$$U_i = \max(U_i(A), U_i(B))$$
$$U_o = \max(U_o(A), U_o(B))$$
$$V_{min} = \min(V_{min}(A), V_{min}(B))$$
$$V_{max} = \max(V_{max}(A), V_{max}(B))$$

Jednoduché podmíněné provedení je speciálním případem úplného podmíněného provedení, ve kterém je sekvence *B* nahrazena nulovou sekvencí. Pro výpočet vlastností jednoduchého podmíněného provedení pak platí následující vzorce:

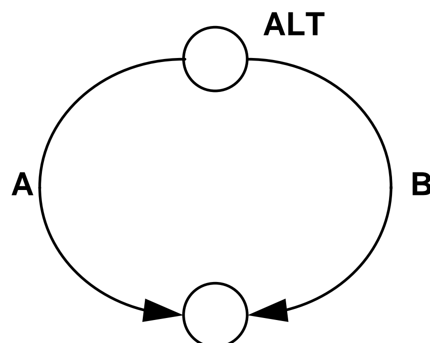
$$U_i = U_i(A)$$
$$U_o = U_o(A)$$
$$V_{min} = \min(V_{min}(A), 0)$$
$$V_{max} = \max(V_{max}(A), 0)$$

Podmíněné provedení je spolu s alternativním provedením jediným provedením sekvencí, které může způsobit to, že hodnota vlastnosti V_{min} je různá od hodnoty vlastnosti V_{max}.

4.3.4 Alternativní provedení

Alternativní provedení dvou sekvencí znamená, že ze dvou sekvencí *A* a *B* je nedeterministicky vybrána jedna, která je provedena a druhá provedena není. Alternativní provedení se často v jazycích pro paralelní a distribuované programování vyskytuje. Příkladem implementace alternativního provedení je příkaz ALT v programovacím jazyce OCCAM.

Graf alternativního provedení je znázorněn na následujícím obrázku. Na tomto obrázku je uveden příklad dvou sekvencí *A* a *B*, z nichž bude nedeterministicky jedna vybrána a ta bude provedena.



Obr. 12: Graf alternativního provedení

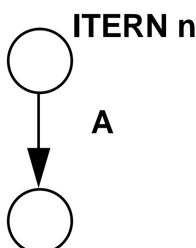
Vlastnosti alternativního provedení dvou sekvencí jsou stejné jako vlastnosti úplného podmíněného provedení. Vzorce pro výpočet vlastností alternativního provedení jsou tedy následující:

U_i = max (U_i(A), U_i(B))
U_o = max (U_o(A), U_o(B))
V_{min} = min (V_{min}(A), V_{min}(B))
V_{max} = max (V_{max}(A), V_{max}(B))

4.3.5 Počítaná iterace

Počítanou iterací se rozumí to, že sekvence je vykonána *n* krát, kde *1* <= *n* a kde *n* je známo v době analýzy grafu toku řízení. Jedná se především o obraty, které jsou v programovacích jazycích vyjadřovány pomocí počítaného cyklu (např. cyklus FOR) a u kterých je v době analýzy znám počet průchodů cyklem. Pokud by v době analýzy nebyl znám počet průchodů cyklem, je třeba tento cyklus interpretovat jako nepočítanou iteraci, která je popsána v následujících odstavcích.

Graf počítané iterace je znázorněn na následujícím obrázku. Na tomto obrázku je uveden příklad sekvence *A*, která je provedena *n* krát.



Obr. 13: Graf počítané iterace

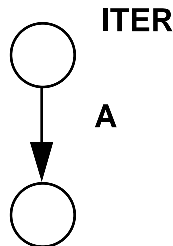
Vzorce pro výpočet vlastností počítané iterace sekvence A jsou tedy následující:

U_i = max (U_i(A), U_i(A))- (n-1) * V_{min}(A))
U_o = max (U_o(A), U_o(A) + (n-1) * V_{max}(A))
V_{min} = n * V_{min}(A)
V_{max} = n * V_{max}(A)

Protože počítaná iterace je vlastně sekvenčním provedením *n* sekvencí A, je možno tyto vzorce snadno odvodit ze vzorců pro výpočet vlastností sekvenčního provedení.

4.3.6 Nepočítaná iterace

Nepočítanou iterací se rozumí to, že sekvence je vykonána *n* krát, kde $1 <= n$ a kde *n* není známo v době analýzy grafu toku řízení. Jedná se především o obraty, které jsou v programovacích jazycích vyjadřovány pomocí nepočítaného cyklu (např. cyklus WHILE nebo REPEAT) a u kterých není v době analýzy znám počet průchodů cyklem. Graf nepočítané iterace je znázorněn na následujícím obrázku.



Obr. 14: Graf nepočítané iterace

Výpočet vlastností nepočítané iterace je poněkud obtížnější než v případě ostatních provedení sekvencí. Při výpočtu je třeba rozlišit následující tři případy:

a) Případ, kdy $V_{min}(A) = V_{max}(A) = 0$.

Jde o případ, kdy hodnoty vlastností $V_{min}(A)$ a $V_{max}(A)$ jsou nulové. V tomto případě výsledné hodnoty vlastností nepočítané iterace nejsou závislé na počtu iterací a je možno je vypočítat podle následujících vzorců, které byly odvozeny ze vzorců pro počítanou iterací dosazením hodnot $V_{min}(A) = V_{max}(A) = 0$.

$$U_i = U_i(A)$$
$$U_o = U_o(A)$$
$$V_{min} = 0$$
$$V_{max} = 0$$

a) Případ, kdy $V_{max}(A) > 0$

Jde o případ, kdy hodnota vlastnosti $V_{max}(A)$ je kladná. V tomto případě výsledné hodnoty vlastností nepočítané iterace jsou závislé na počtu iterací. Protože počet iterací není v době analýzy znám, je třeba předpokládat, že počet iterací je nekonečný. Hodnoty vlastností je možno vypočítat podle následujících vzorců, které byly odvozeny ze vzorců pro počítanou iterací dosazením hodnot $n=\infty$ a $V_{max}(A) > 0$.

$U_i = U_i(A)$

$U_o = +\infty$

$V_{min} = +\infty$

$V_{max} = +\infty$

a) Případ, kdy $V_{max}(A) < 0$

Jde o případ, kdy hodnota vlastnosti $V_{max}(A)$ je záporná. V tomto případě výsledné hodnoty vlastností nepočítané iterace jsou závislé na počtu iterací. Protože počet iterací není v době analýzy znám, je třeba předpokládat, že počet iterací je nekonečný. Hodnoty vlastností je možno vypočítat podle následujících vzorců, které byly odvozeny ze vzorců pro počítanou iterací dosazením hodnot $n=\infty$ a $V_{max}(A) < 0$.

$U_i = +\infty$

$U_o = U_o(A)$

$V_{min} = -\infty$

$V_{max} = -\infty$

Případ a) je z hlediska analýzy délek front interakčních bodů v pořádku, neboť hodnoty vlastností interakčního bodu nejsou závislé na počtu iterací.

Oproti tomu případy b) a c) jsou nepříjemné, neboť při neznalosti počtu iterací musíme předpokládat, že délka vstupní fronty (v případě b) nebo délka fronty čekajících procesů (v případě c) je nekonečná.

Případ c) není tak kritický, jako případ b). V případě c) vychází, že maximální délka fronty čekajících procesů musí být nekonečná. V rámci celého paralelního programu je však jasné, že maximální délka fronty čekajících procesů interakčního bodu *id* nemusí být větší, než celkový počat procesů, které s tímto interakčním bodem pracují pomocí operace *in* (jak si také ukážeme v jedné a následujících podkapitol). To znamená, že platí

$U_i <= |ref\ (id,\ in)|$

Vzhledem k tomu, že skutečný počet paralelních procesů v reálných programech bývá poměrně malý, je délka fronty čekajících procesů i v tomto nepříjemném případě poměrně omezená.

Vážnějším problémem je však případ b). V tomto případě vychází, že maximální délka vstupní fronty musí být nekonečná. Na rozdíl od případu c) však v rámci celého paralelního programu neexistuje žádné omezení na počet položek vstupní fronty interakčního bodu a při implementaci je nutno skutečně předpokládat nekonečnou délku vstupní fronty tohoto interakčního bodu. Protože tento případ by mohl narušit výsledky prováděné analýzy, budeme mu věnovat následující podkapitolu.

4.4 Nepočítaná iterace s nekonečnou vstupní frontou

Jak bylo uvedeno v předchozí podkapitole, může nastat případ, kdy se v paralelním programu nachází nepočítaná iterace a délka vstupní fronty této iterace je nekonečná. Protože tento případ je z implementačního hlediska nepříjemný, budeme se jím zabývat podrobněji. K výše zmíněné situaci může dojít ze tří možných příčin:

a) Jde o neodhalenou počítanou iteraci

První možnou příčinou je, že jde ve skutečnosti o počítanou iteraci, ale analýza toku řízení nebyla schopna tuto zjistit skutečnost zjistit. Příkladem může být počítaná iterace, vytvořená pomocí cyklu WHILE nebo počítaná iterace, vytvořená pomocí cyklu FOR, u které se nepodařilo v době analýzy zjistit počet průchodů cyklem. Problém tedy není v samotném programu, ale v nedostatečně kvalitní analýze. Jediným způsobem, jak tento případ odstranit, je zkvalitnit analýzu. To znamená, analýzu toku řízení doplnit i analýzou toku dat, která umožní identifikovat tuto počítanou iteraci.

b) Jde o neodhalený složený interakční mechanismus

Druhou možnou příčinou nekonečné délky vstupní fronty může být to, že interakční bod je jednou polovinou složeného interakčního mechanismu a tato skutečnost nebyla zjištěna. Při analýze může dojít situaci, že nebude správně identifikována dvojice interakčních bodů, které tvoří složený interakční mechanismus. Vzhledem k charakteru a k použití složených interakčních mechanismů je pravděpodobné, že takovéto interakční body se budou nacházet v nepočítané iteraci a budou požadovat vstupní frontu nekonečné délky. Protože je však identifikace interakčních bodů, které tvoří složený interakční mechanismu, poměrně jednoduchá, nemělo by v praxi k takovým případům docházet.

c) Jde o konstrukci se skutečně požadovanou nekonečnou kapacitou

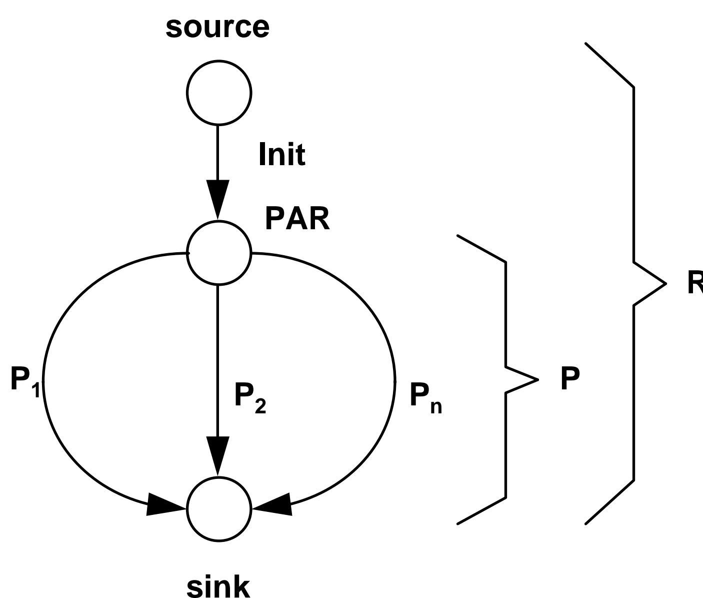
Poslední možnou příčinou nekonečné délky vstupní fronty může být to, že skutečně jde o interakční mechanismus, který obecně požaduje nekonečnou délku vstupní fronty. Může jít o dva druhy interakčních mechanismů: o asynchronní kanál a o asynchronní přihrádku. Pokud nastane tento případ, tak to znamená, že programátor skutečně požadoval interakční mechanismus s těmito vlastnostmi a v tomto případě žádná optimalizace není možná. Případ, kdy programátor požaduje skutečný asynchronní kanál nebo asynchronní přihrádku je však v paralelních programech poměrně vzácný. Důvodem je to, že tento mechanismus s nekonečnou kapacitou nelze pro fyzicky omezenou reálnou paměť nikdy plnohodnotně realizovat a skutečná realizace vždy skončí na kanálu nebo přihrádce s konečnou kapacitou. Častější je případ, kdy jde o omyl programátora. Proto je vhodné, aby proces analýzy uživatele na výskyt tohoto případu upozornil.

4.5 Určení maximálních délek front v programu

V této podkapitole se pokusíme pomocí dříve definovaného aparátu určit maximální délku vstupní fronty a fronty čekajících procesů daného interakčního bodu v paralelním programu.

Pro určení maximálních délek front interakčního bodu je třeba vzít v úvahu celý paralelní program. Obecná struktura toku řízení paralelního programu je znázorněna na Obr. 15. Obecný paralelní program *R* se skládá z inicializační sekvence *Init*, za kterou sekvenčně následuje paralelní provedení všech paralelních procesů tohoto programu *P1* až *Pn*.

Inicializační sekvence *Init* se skládá ze sekvenčního provedení několika příkazů *out*. Počet těchto příkazů *out* je roven počátečnímu počtu položek ve vstupní frontě, tj. hodnotě *id.init*. Vlastnosti inicializační sekvence P_{init} lze velmi snadno spočíst jako vlastnosti sekvenčního spojení několika elementárních sekvencí *out*, které má vlastnosti



Obr. 15: Graf paralelního programu R s procesy

Po inicializaci sekvenčně následuje paralelní běh několika procesů (viz Obr. 15). Hodnoty vlastností V_{min} a V_{max} tohoto paralelního spojení jsou nezajímavé, neboť nebudou v žádném následujícím výpočtu figurovat. Hodnoty vlastností U_o a U_i se spočtou podle pravidel pro paralelní provedení sekvencí:

$$
\begin{aligned}
U_i(P,id) &= \sum_{P_j \in ref(id,\ in)} U_i\ (P_j,\ id) \\
U_o(P,id) &= \sum_{P_j \in ref(id,\ out)} U_o\ (P_j)
\end{aligned}
$$

Vlastnosti celého programu jsou sekvenčním provedením inicializační sekvence a běhu několika paralelních procesů. Hodnoty vlastností V_{min} a V_{max} jsou opět nezajímavé a hodnoty vlastností U_o a U_i celého programu se spočtou podle pravidel pro sekvenční provedení sekvencí:

$$
\begin{aligned}
U_i(R,\ id) &= max\ (U_i(Init),\ U_i(P) - V_{min}(Init)) = \\
&= \sum_{P_j \in ref(id,\ in)} U_i\ (P_j,\ id) - id.init \\
\\
U_o(R,\ id) &= max\ (U_o(Init),\ U_o(P) + V_{max}(Init)) = \\
&= id.init + \sum_{P_j \in ref(id,\ out)} U_o\ (P_j,\ id)
\end{aligned}
$$

Hodnoty funkcí *maxentry* a *maxwaiting* jsou pak pro jednoduchý interakční bod rovny hodnotám funkcí U_o a U_i. Pouze u funkce *maxwaiting* je výsledná hodnota shora omezena celkovým počtem procesů, které přistupují pomocí operace *in* k interakčnímu bodu *id*. Je zřejmé, že celkové požadavky na frontu čekajících procesů interakčního bodu nemohou být nikdy větší, než je počet procesů, které provádí operaci *in* s tímto interakčním bodem. Výsledné vzorce pro výpočet hodnot funkcí *maxentry* a *maxwaiting* jednoduchého interakčního bodu mají tedy tvar:

$$
\begin{aligned}
&maxentry\ (id) = id.init + \sum_{P_j \in ref(id,\ out)} U_o\ (P_j,\ id) \\
\\
&maxwaiting\ (id) = \\
&\quad min\ (|ref\ (id,\ in)|,\ \sum_{P_j \in ref(id,\ in)} U_i\ (P_j,\ id) - id.init)
\end{aligned}
$$

V případě složeného interakčního bodu je situace poněkud složitější. Jak již bylo uvedeno dříve, složené interakční mechanismy se skládají ze dvou interakčních bodů id_1 a id_2. Typická implementace primitivní operace, prováděné se složeným interakčním bodem se skládá ze dvou operací s interakčním bodem (z jedné operace *in* a z jedné operace *out*). Pořadí těchto dvou operací závisí na tom, zda se jedná o proces odesílatel (klient) nebo o proces příjemce (server).

```
Odesílatel, klient Příjemce, server
|
V
out (id1, D1) ------> in (id1, D1)
|
V
in (id2, D2) <------ out (id2, D2)
|
V
```

Z hlediska hodnot funkcí *maxwaiting* je situace poměrně jednoduchá. Hodnoty funkcí *maxwaiting* interakčních bodů id_1 a id_2 se v případě složeného interakčního bodu spočtou podle následujících vztahů:

```
maxwaiting (id1) = |ref (id1, in)|
maxwaiting (id2) = |ref (id2, in)|
```

Na rozdíl od případu jednoduchého interakčního mechanismu je maximální délka fronty čekajících procesů interakčních obou bodů rovna počtu procesů, které se na tyto interakční body odkazují pomocí operace *in*. Je to způsobeno tím, že například proces klient mimo operace *in* s interakčním bodem id_2 provádí i operace *out* s interakčním bodem id_1, které prostřednictvím procesu server způsobí dříve nebo později odstranění položek z fronty čekajících procesů interakčního bodu id_2. Protože však proces server běží nezávisle na procesu klient může se stát, že je proces server pomalý (nebo zablokovaný) a nezpracovává po nějakou dobu požadavky klientů. Z hlediska procesu klient je pak situace ekvivalentní případu, kdy proces klient obsahuje pouze operace *in* a žádnou operaci *out* (protože provedení operace *out* nemusí mít žádný efekt na délku fronty čekajících procesů) a pak podle vzorce pro výpočet hodnoty maxwaiting jednoduchého interakčního bodu platí *maxwaiting* (id_2) = /*ref* (id_2, *in*)/. Totéž lze pak odvodit naopak pro případ procesu server, pro který platí *maxwaiting* (id_1) = /*ref* (id_1, *in*)/.

Zatímco v případě hodnot funkcí *maxwaiting* bylo možné brát v úvahu každý z interakčních bodů složeného mechanismu samostatně, u hodnot funkcí *maxentry* tomu tak není. Protože procesy klient i procesy server běží nezávisle na sobě, mohou se položky vstupních front libovolně přelévat z interakčního bodu id_1 do interakčního bodu id_2 (činností procesu server) a naopak (činností procesu klient). Požadavky na maximální délku front interakčních bodů id_1 a id_2 jsou tedy stejné a v nejobecnějším případě platí vztah

```
maxentry (id1) = maxentry (id2).
```

Hodnoty funkcí *maxentry* (id_1) a *maxentry* (id_2) jsou tedy součtem požadavků na vstupní fronty interakčních bodů id_1 a id_2 a spočtou se podle následujícího vztahu:

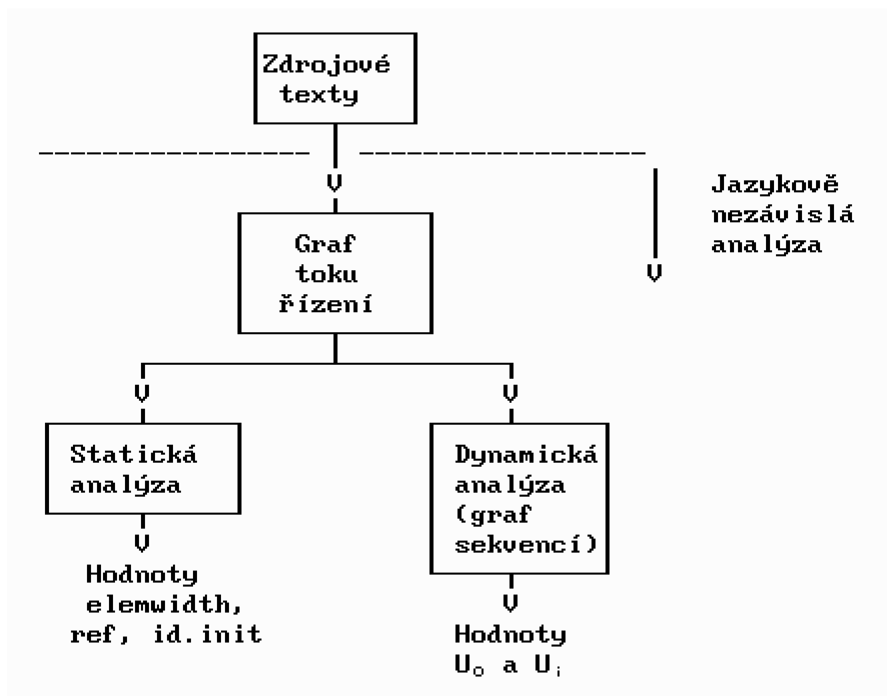
$$\text{maxentry}\,(id_1) = \text{maxentry}\,(id_2) = id_1.\text{init} + id_2.\text{init} + \\ = \sum_{P_j \in \text{ref}(id_1,\ \text{out})} U_o\,(P_j,\ id_{12}) + \sum_{P_j \in \text{ref}(id_2,\ \text{out})} U_o\,(P_j,\ id_{12})$$

v tomto vztahu zápis U_o (P, id_{12}) znamená, že hodnota U_o je vztažena ke dvojici interakčních bodů id_1 a id_2, které tvoří složený interakční mechanismus.

Příklad určení maximálních délek front v programu je uveden v příloze A této práce.

5. MOŽNÉ APLIKACE MODELU

V předchozí kapitole této práce byl popsán postup optimalizace implementace interakčního bodu. Nyní popíšeme celý proces analýzy a optimalizace interakčních bodů. Na Obr. 16 je tento proces znázorněn graficky.

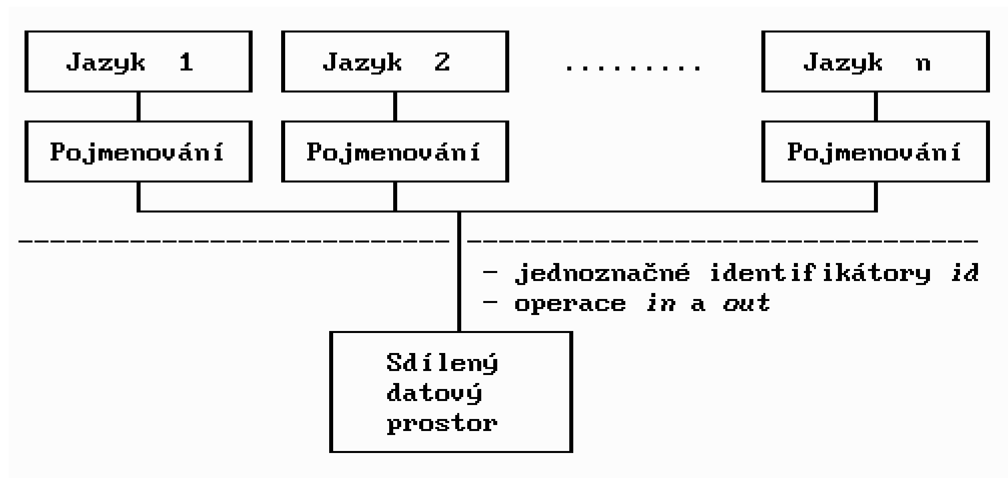


Obr. 16: Proces analýzy interakčního bodu

Vstupem tohoto procesu jsou zdrojové texty všech procesů paralelního programu. Tyto zdrojové texty jsou transformovány na graf toku řízení. Během této transformace jsou ze zdrojových textů odstraněna syntaktická informace. výsledný graf toku řízení obsahuje pouze sémantickou informaci a je nezávislý na použitém programovacím jazyce. Graf toku řízení je pak podroben statické a dynamické analýze. Během statické analýzy se z grafu získají hodnoty *elemwidth*, *ref* a *id.init*. Tato analýza je poměrně jednoduchá. Má charakter sběru statistických informací z grafu toku řízení bez toho, aby bylo potřeba analyzovat vztah mezi jednotlivými uzly tohoto grafu.

Dynamická analýza slouží k získání hodnot funkcí U_O a U_I, které jsou nezbytné pro výpočet hodnot *maxentry* a *maxwaiting*. Tato analýza je poměrně složitější, využívá dříve definovaného grafu sekvencí a byla popsána v předchozí kapitole.

Dalším možným využitím vypracovaného modelu interakčních mechanismů je vytvoření jednotného rozhraní různých paralelních a distribuovaných programovacích jazyků.



Obr. 17: Jednotné rozhraní jazyků

Předpokládejme, že máme k dispozici několik paralelních programovacích jazyků *Jazyk1* až *Jazykn* (viz. Obr. 17). Některé z těchto jazyků používají interakci pomocí předávání zpráv a některé používají sdílenou paměť. Každý z těchto jazyků používá jistý druh identifikátorů pro popsání identity konkrétního interakčního objektu. Pomocí procesu, zvaného *pojmenování* (naming) můžeme každému z těchto identifikátorů přiřadit interakční bod (nebo několik interakčních bodů), označených jedinečným identifikátorem *id*. Pro každý z použitých jazyků můžeme také transformovat jeho operace s interakčními objekty na operace *in* a *out*. Po takovéto transformaci můžeme provést analýzu vlastností každého interakčního bodu a můžeme zvolit optimální implementaci tohoto interakčního bodu pomocí dostupné fyzické implementace sdíleného datového prostoru. Pomocí této metody můžeme dosáhnout následujících dvou vlastností:

- Programovací jazyk může používat jiný princip interakce (např. předávání zpráv nebo sdílenou paměť) než fyzická nebo logická platforma, na které je implementován. Je možno snadno používat programovací jazyky, které mají kombinovaný charakter (tj. používají předávání zpráv i sdílenou paměť) nebo i jazyky, které používají některý virtuální interakční mechanismus (např. nástěnku nebo virtuální sdílenou paměť). Můžeme také snadno změnit implementační platformu bez nutnosti změn aplikačních programů.

- Můžeme používat heterogenní paralelní programy, tj. programy, které se skládají z procesů, napsaných v různých programovacích jazycích. Koncept interakčního bodu poskytuje jazykově nezávislou platformu pro zajištění interakce mezi procesy, napsanými v různých programovacích jazycích.

6. ZÁVĚR

Tato práce se zabývá jedním z problémů paralelního a distribuovaného výpočetního prostředí, kterým je interakce mezi procesy. Cílem této práce je prezentovat formální model, vhodný pro jednotný popis všech běžně používaných prostředků pro interakci mezi paralelními a distribuovanými procesy. Jako základ pro tento formální model byla použita datová abstrakce, nazvaná *sdílený datový prostor*. Tento sdílený datový prostor obsahuje objekty, zvané *interakční body*. Pomocí těchto interakčních bodů a základních operací, které jsou nad nimi definovány, budou pak vyjadřovány jednotlivé interakční mechanismy.

Byl definován model pro tzv. *jednoduché interakční mechanismy* (to jsou ty mechanismy, které je možno modelovat jedním interakčním bodem) a *složené interakční mechanismy* (což jsou mechanismy, pro jejichž vyjádření je třeba dvou a více interakčních bodů). Poslední část práce se věnuje problematice optimalizace implementace interakčních mechanismů, vytvořených na základě výše definovaných modelů.

Několik málo částí tohoto dokumentu čerpalo z literatury. Například výpočetní model RAM byl převzat z literatury ([Gib93], [Ham92]). Rovněž kapitola 2 čerpala z mnoha literárních pramenů. Popis jednotlivých interakčních mechanismů byl však prakticky u všech vytvořen znovu, aby se dosáhlo jednotnosti popisu. Jména a částečně i sémantika operací *in* a *out*, používaných v následujících kapitolách, byly vypůjčeny z programovacího jazyka Linda.

Ostatní části práce, tj. především kapitoly 3, 4 a 5 obsahují zcela původní myšlenky. Za nejvýznamnější přínos práce je považována myšlenka návrhu interakčního bodu spolu s vypracováním společného modelu a klasifikace většiny známých interakčních mechanismů. Dalším přínosem je rozpracování postupu analýzy optimalizace implementace tohoto modelu.

Základní myšlenky, uvedené v této práci, byly poprvé veřejně prezentovány v květnu 1993 na workshopu o paralelních a distribuovaných algoritmech, pořádaném Katedrou počítačů ČVUT Praha jako přednáška bez publikace a poté na mezinárodní konferenci SOFSEM'93. Téma pak bylo dále rozpracováváno a průběžně prezentováno na dalších seminářích a konferencích (viz příloha B).

7. POUŽITÁ LITERATURA

[And91] Andrews G. R.: Concurrent Programming: principles and practice, The Benjamin Cummings Publishing Company Inc., 1991

[Car86] Carriero, N., Gelernter, D.: The S/Net's Linda Kernel, ACM Trans. Comp. Sys. (May 1986)

[Car88] Carriero, N., Gelernter, D.: How to write parallel programs: a guide to the perplexed, Yale University Department of Computer Science, 1988

[Car89] Carriero, N., Gelernter, D.: Linda in Context, CACM, Volume 32, No 4, 1989, pp. 444--458

[Car90] Carriero N., Gelernter D.: How to write parallel programs: a first course, MIT Press, 1990

[Car91] Carriero N., Gelernter D.: Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler, in Advances in Languages and Compilers for Parallel Processing, Pitman Publishing, London 1991, ISBN 0953-7767

[Cas93] Casavant, T.: Architectures for Massively Parllel Computers, in proceedings of ISIPCALA'93, Czech Technical University, Prague, July 1993

[Cha84] Chambers, F.: Distributed Computing, Academic Press, 1984[Dei90] Deitel. H.M.: Operating Systems, Addison Wesley Publishung 1990, ISBN 0-201-18038-3

[Fly72] Flynn M.J.: Some computer organizations and their effectiveness, IEEE Trans. on Computers, C-21, 1972, 9, pp. 948-960

[Fuj90] Fujimoto, R.: Parallel Discrete-Event Simulation, Comm. of ACM vol. 33, No. 10, 30-53

[Gan93] Gannon, D.: Directions in Parallel Programming: HPF, Shared Virtual Memory and Object Parallelism in C++, in proceedings of ISIPCALA'93, Czech Technical University, Prague, July 1993

[Gel85] Gelernter D.: Generative Communication in Linda, in ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, January 1985

[Gib93] Gibbons P. B.: Asynchronous PRAM algorithms, in Syntesis of Parallel Algorithms, 1993

[Gib90] Gibbons A., Rytter, W.: Efficient Parallel Algorithms, Cambridge University Press 1990, ISBN 0-521-38841-4

[Goo89] Goodman, J.R., Vernon, M., Woest, P.J.: Efficient Synchronization primitives for Large-Scale Cache-Coherent Multiprocessors, in CACM 0-89791-300-0/89/0004/0064

[Gri91] Grimshaw A.: An Introduction to Parallel Object-Oriented Programming with Mentat, Computer Science Report No. TR-91-07, Department of Computer Science, University of Virginia, April 1991

[Gut93] Gutzwiller S., Ohnack P.: Basel Algorithm Classification Scheme, Report 92-1, Institut für Informatik der Universität Basel, 1993

[Ham92] Hamalainen, P.: PRAM emulator, Report B-1992-2, University of Joensuu, Department of Computer Science, 1992.

[Han92] Hanáček P., Přikryl P.: The Linda System in a Distributed Environment -- the Experimental Implementation, SOFSEM'92, Ždiar, Magura, 22.11. - 4.12.1992, 4 strany

[Ha93a] Hanáček P.: Parallel Simulation Using the Linda Language, 5th Moravo-Silesian International Symposium on Modelling and Simulation of Systems MOSIS'93, , Olomouc, June 1-4, 1993, sborník strana 263-267, pořadatel Dům techniky Ostrava

[Ha93b] Hanáček P.: The Common Model for the Communication and Synchronization Primitives, Mezinárodní konference SOFSEM'93, Hrdoňov, Šumava, 21.11.-3.12. 1993

[Han94] Hanáček P.: Optimalizace jazyka Linda pro paralelní simulaci, Modelling and Simulation of Systems MOSIS'94, House of Technology Ostrava, Zábřeh na Moravě 30.5.-2.6. 1994, str. 196-201

[Han96] Hanáček P.: Virtual Time for Parallel Simulation, MOSIS'96, Zábřeh na Moravě, 1996

[Hil85] Hillis, D.: The Connection Machine, The MIT Press, 1985

[HoU79] Hopcroft, J., E., Ullman, J., D.: Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979

[Hoa85] Hoare, C., A., R.: Communicating Sequential Processes, Prentice Hall 1985[Inm86] INMOS Ltd.: Transputer architecture reference manual., INMOS Ltd., 1986

[In88a] INMOS Ltd.: Transputer instruction set - a compiler writer's guide, Prentice Hall 1988, ISBN 0-13-929100-8

[In88b] INMOS Ltd.: occam 2 Reference Manual, Prentice Hall 1988

[Jag91] Jagannathan, S.: Optimizing Analysis for First-Class Tuple-Spaces, in Advances in Languages and Compilers for Parallel Processing, Pitman Publishing, London 1991, ISBN 0953-7767

[Jef85] Jefferson, D. R.: Virtual Time, ACM Transactions on Programming Languages and Systems, 7(3):404-425, July 1985

[Jef90] Jefferson, D. R.: Virtual Time II: The Cancelback protocol for storage management in distributed simulation, Proc. 9th Annual ACM Symposium on Principles of Distributed Computation, pages 75-90, August 1990

[Kaf93] Kafura, D., G., Lavender, G.: Concurrent Object-Oriented Languages and the Inheritance Anomaly, in proceedings of ISIPCALA'93, Czech Technical University, Prague, July 1993

[May86] May, D.: OCCAM 2 product definition, INMOS Ltd., 1986

[Pfa92] Pfaltz, John L.: Programming over a Persistent Data Space, IPC Technical Report 92-008, University of Virginia, 1992 [Pri95] Přikryl, P. - Hanáček,

P. - Ryšánek, M.: The Linda Language in Distributed Environment, Proceedings of WORKSHOP'95, January 23--26, 1995, Prague, Czech Republic, pp. 231-232

[Ray86] Raynal, M.: Algorithms for Mutual Exclusion, NORTH OXFORD ACADEMIC Publishers Limited, 1986

[She91] Shekhar, K., H, Srikant, Y., N.: Linda Sub System on Transputers, in Transputing 91, IOS Press 1991

[Sch93] Schill, A.: DCE - Das OSF Distributed Computing Environment, Springer Verlag, 1993, ISBN 3-540-55355-5

[Sno92] Snow, C., R.: Concurrent Programming, Cambridge Computer Science Texts, Cambridge University Press 1992, ISBN 0-521-33993-6

[Stu90] Stumm, M., Zhou, S.: Algorithms Implementing Distributed Shared Memory, University of Toronto, 1990

[Tan92] Tanenbaum A., Kaashoek F., Bal H.: Parallel Programming Using Shared Objects and Broadcasting, Comm. ACM, August 1992, pp. 10-19

[Wil90] Williams, Shirley A.: Programming models for parallel systems, John Wiley & Sons 1990, ISBN 0 471 92304 4

PŘÍLOHA A - PŘÍKLAD VÝPOČTU MAXIMÁLNÍCH DÉLEK FRONT

Pro ilustraci výpočtu délek front interakčních bodů uvedeme následující příklad paralelního programu se dvěma paralelně běžícími procesy P1 a P2. Program je popsán zdrojovým textem v jazyce Pascal, do kterého je přidána jazyková konstrukce *parbegin - parend*, která dovoluje spustit několik procesů paralelně. Styk programu se sdíleným datovým prostorem je proveden pomocí operací *in* a *out*.

Příklad 3:

```
Program Example3;

Procedure P1;       {První paralelní proces}
begin
   while true do
      begin
      ....
      in (id3);
      out (id1, D1);
      in (id2, D2);
      out (id3);
      end;
end;

Procedure P2;       {Druhý paralelní proces}
begin
   while true do
      begin
      in (id3);
      in (id1, D1);
      out (id2, D2);
      out (id3);
      ....
      end;
end;

begin {Inicializace}
   out (id3);
   parbegin         {Paralelní spuštění procesů P1 a P2}
   P1;
   P2;
   parend
end.
```

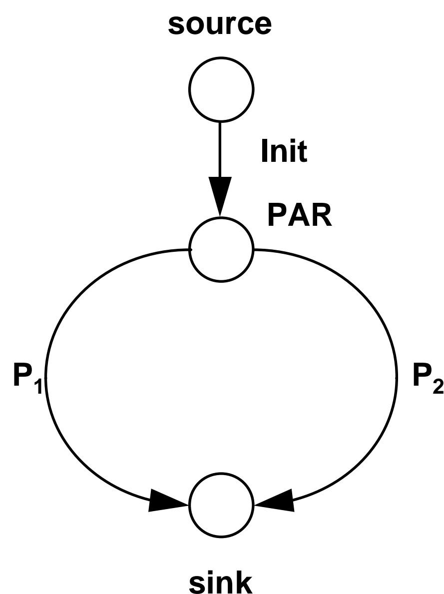
Postup analýzy a optimalizace

1. Zjistí se, že interakční body id_1 a id_2 tvoří složený interakční mechanismus a interakční bod id_3 tvoří jednoduchý interakční mechanismus (viz. kapitola 3.8).

2. Statickou analýzou se zjistí hodnoty funkcí *elemwidth* pro jednotlivé interakční body

```
elemwidth (id1) > 0
elemwidth (id2) > 0
elemwidth (id3) = 0
```

3. Vytvoří se graf sekvencí paralelního programu.



Obr. 1: Graf sekvencí pro příklad 3

4. Provede se výpočet pro jednoduchý interakční mechanismus s interakčním bodem id_3.

4a. Spočtou se hodnoty vlastností pro jednotlivé sekvence, z nichž se analyzovaný program skládá.

```
P (Init) = P (out) = ( 0, 1, +1)
P (P1) = P ([in out ]) = ( 1, 0, 0, 0)
P (P2) = P ([in out ]) = ( 1, 0, 0, 0)
```

4b. Spočtou se hodnoty funkcí *maxentry* a *maxwaiting*.

```
maxentry (id3)     = id3.init + Σ Uo (Pj, id3)       =
                              Pj ∈ ref(id3, out)

                   = 1 + 0 + 0 = 1


maxwaiting (id3)  =
    min (|ref (id3, in)|, Σ Ui (Pj, id3) - id3.init) =
                         Pj ∈ ref(id3, in)

 =  min (2, 1 + 1 - 1) = min (2, 1) = 1
```

4c. Určí se vlastnosti D, E a W interakčního mechanismu. Vlastnosti interakčního mechanismu tvořeného interakčním bodem id_3 jsou 0. Jde tedy o interakční mechanismus, označený ve svazovém diagramu na Obr. 5 symbolem *Sem1*. Tento interakční mechanismus je tedy jedním speciálním typem semaforu. Je možno jej implementovat jedním interakčním bodem, který má vlastnosti:

```
elemwidth (id3) = 0
maxentry (id3) = 1
maxwaiting (id3) = 1
```

Vzhledem k tomu, že tento interakční bod nepředává žádná data (elemwidth (id_3) = 0), obsahuje pouze jedinou hodnotu (maxentry (id_3) = 1) a může na něj čekat pouze jediný proces (maxwaiting (id_3) = 1), datová struktura pro jeho implementaci obsahuje pouze jediný příznak obsazenosti a prostor pro deskriptor jediného čekajícího procesu. Je zřejmé, že tato implementace je mnohem jednodušší a efektivnější než implementace obecného semaforu nebo dokonce implementace obecného interakčního bodu.

5. Provede se výpočet pro složený interakční mechanismus s interakčními body id_1 a id_2.

5a. Spočtou se hodnoty vlastností pro jednotlivé sekvence, z nichž se analyzovaný program skládá.

```
P (Init) = P (0) = ( 0, 0, 0)
P (P1) = P ([out in ]) = ( 0, 1, 0, 0)
P (P2) = P ([in out ]) = ( 1, 0, 0, 0)
```

5b. Spočtou se hodnoty funkcí *maxentry* a *maxwaiting*.

$$\mathrm{maxwaiting}\ (id_1) = |\mathrm{ref}\ (id_1, \mathrm{in})| = 1$$

$$\mathrm{maxwaiting}\ (id_2) = |\mathrm{ref}\ (id_2, \mathrm{in})| = 1$$

$$\mathrm{maxentry}\ (id_1) = \mathrm{maxentry}\ (id_2) = id_1.\mathrm{init} + id_2.\mathrm{init} + \sum_{P_j \in \mathrm{ref}(id_1,\ \mathrm{out})} U_o\ (P_j,\ id_{12}) + \sum_{P_j \in \mathrm{ref}(id_2,\ \mathrm{out})} U_o\ (P_j,\ id_{12}) =$$

$$= 0 + 0 + 1 + 0 = 1$$

5c. Určí se vlastnosti D, E a W interakčního mechanismu. Vlastnosti interakčního mechanismu tvořeného interakčními body id_1 a id_2 jsou D. Jde tedy o interakční mechanismus, označený ve svazovém diagramu na Obr. 6 symbolem RPC1. Tento interakční mechanismus je tedy jedním speciálním případem volání vzdálené procedury. Je možno jej implementovat dvěma interakčními body, které mají vlastnosti:

```
elemwidth (id1) > 0
maxwaiting (id1) = 1
maxentry (id1) = 1
```

a

```
elemwidth (id2) > 0
maxwaiting (id2) = 1
maxentry (id2) = 1
```

Oba interakční body předávají data (elemwidth > 0), obsahují pouze jedinou hodnotu (maxentry = 1) a může na něj čekat pouze jediný proces (maxwaiting = 1). Datová struktura pro jejich implementaci obsahuje pouze jediný prostor pro předávaná data a prostor pro deskriptor jediného čekajícího procesu. Je zřejmé, že tato implementace je mnohem jednodušší a efektivnější než implementace obecného volání vzdálené procedury nebo dokonce implementace obecného interakčního bodu.

PŘÍLOHA B - VYBRANÉ PUBLIKACE AUTORA

V této příloze jsou uvedeny kopie několika vybraných publikací autora, které mají přímý vztah k řešené problematice. jedná se o tyto publikace:

- [Han92] Hanáček P., Přikryl P.: The Linda System in a Distributed Environment -- the Experimental Implementation, SOFSEM'92, Ždiar, Magura, 22.11. - 4.12.1992, 4 strany

- [Ha93a] Hanáček P.: Parallel Simulation Using the Linda Language, 5th Moravo-Silesian International Symposium on Modelling and Simulation of Systems MOSIS'93, , Olomouc, June 1-4, 1993, sborník strana 263-267, pořadatel Dům techniky Ostrava

- [Ha93b] Hanáček P.: The Common Model for the Communication and Synchronization Primitives, Mezinárodní konference SOFSEM'93, Hrdoňov, Šumava, 21.11.-3.12. 1993

- [Han94] Hanáček P.: Optimalizace jazyka Linda pro paralelní simulaci, Modelling and Simulation of Systems MOSIS'94, House of Technology Ostrava, Zábřeh na Moravě 30.5.-2.6. 1994, str. 196-201

- [Pri95] Přikryl, P. - Hanáček, P. - Ryšánek, M.: The Linda Language in Distributed Environment, Proceedings of WORKSHOP'95, January 23--26, 1995, Prague, Czech Republic, pp. 231-232

--