VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘÍKAZOVÝ INTERPRET PRO VIRTUÁLNÍ STROJ PETRIHO SÍTÍ

COMMAND INTERPRETER FOR OOPN VIRTUAL MACHINE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ADAM HUSPENINA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. RADEK KOČÍ, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Huspenina Adam**
Program: Informační technologie
Název: **Příkazový interpret pro virtuální stroj Petriho sítí**
Command Interpreter for OOPN Virtual Machine
Kategorie: Operační systémy

Zadání:

1. Prostudujte problematiku tvorby interpretů a virtuálních strojů.
2. Seznamte se s aktuální implementací virtuálního stroje Petriho sítí (OOPNVM).
3. Navrhněte systém vzdálené správy virtuálního stroje pomocí textových příkazů. Návrh bude obsahovat příkazový interpret, komunikační protokol a klientskou a serverovou část.
4. Příkazový interpret musí pokrývat základní funkčnost systému (spuštění, zastavení a krokování simulace, načtení modelů, zjišťování a ukládání stavu). Zvažte možnost tvorby skriptů.
5. Navržený systém implementujte v jazyce C++.
6. Vytvořte dokumentaci nástroje a sadu testů ověřující funkčnost nástroje.

Literatura:

- V. Janoušek: Modelování objektů Petriho sítěmi. Disertační práce. VUT v Brně, 1998.
- PNtalk 2.0. http://perchta.fit.vutbr.cz/pntalk2k/, 2018.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Kočí Radek, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Tato práce se zabývá tvorbou systému pro vzdálenou správu simulačního nástroje PNtalk 2.0. PNtalk 2.0 je simulačním nástrojem Objektově orientovaných Petriho sítí, který je implementován ve jazyce Smalltalk. Vzdálená správa probíhá prostřednictvím textový příkazu, které jsou zpracovány příkazovým interpretem.

Abstract

This thesis deals with creation of a system for remote control of tool PNtalk 2.0. PNtalk 2.0 is simulation tool for Object oriented Petri nets and it's implemented in Smalltalk. Remote control is performed by text commands, which are processed by command-line interpreter.

Klíčová slova

PNtalk 2.0, příkazový interpret, klient, server, komunikační protokol

Keywords

PNtalk 2.0, command interpreter, client, server, communication protocol

Citace

HUSPENINA, Adam. *Příkazový interpret pro virtuální stroj Petriho sítí*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Příkazový interpret pro virtuální stroj Petriho sítí

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Kočího Ph.D.. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Adam Huspenina
15. 5.

Poděkování

Chtěl bych poděkovat vedoucímu mé bakalářské práce Ing. Radku Kočímu Ph.D. za jeho trpělivost, odborné vedení a rady při vypracování této práce.

Obsah

Kapitola 1

Úvod

Interpretem je označován v odvětví informačních technologií speciální počítačový program, který umožňuje přímé spouštění, resp. vykonávání instrukcí jiného programu zapsaného ve svém programovacím nebo skriptovacím jazyce. Interpretovaný program není nutné převádět do strojového kódu cílového procesoru. Interpret tak umožňuje vytvářet programový kód, který není vázán počítačovými platformami. Interpret je využíván v prostředí příkazové řádky, kde je znám také jako příkazový interpret.

Cílem této práce bude vytvoření systému, který umožní vzdálenou správu simulačního nástroje pro oběktově orientované Petriho sítě *PNtalk 2.0*, prostřednictvím textových příkazů. K vytvoření takového systému bude zapotřebí navrhnou vhodný komunikační prostředek, který bude složen z klientské a serverové části systému komunikujících prostřednictvím navrženého protokolu. Jádrem systému pak bude interpret textových příkazů. Systém bude napodobovat příkazovou řádku s textovým uživatelským rozhraním (tzv. CLI – *command-line-interface*).

Tato práce je rozdělena do několika kapito. V kapitole 2 je shrnuta základní teorie potřebná k pochopení pro jaký simulační nástroj je interpretační systém tvořen. Kapitola 3 se věnuje problematice využití interpretů a vysvětlení důvodů jejich použití v návrhu řešení tohoto systému. Kapitola 4 je stěžejní kapitolou této programové dokumentace. V této kapitole jsou analyzovány požadavky na výsledný systém, následovány popisy návrhů jednotlivých logických celků. V těchto podkapitolách jsou k nalezení návrhy částí systému vzhledem k definovaným požadavkům, společně s popisy konkrétní implementace těchto části. Kapitola 5 se věnuje provedenému testování systému. V poslední kapitole 6 je pak shrnuta tato práce s dosaženými výsledky.

Kapitola 2

Objektově orientované Petriho sítě

Tato kapitola uvádí do problematiky objektově orientovaného konceptu Petriho sítí jako základ k modelovacímu nástroji PNtalk 2.0. V souvislosti s nimi bude nejprve představen základní koncept Petriho sítí. Následně bude uveden formalismus Oběktově orientovaných Petriho sítí (Object-Oriented Petri nets – OOPN), jejichž konkrétním systémem a jazykem je PNtalk 2.3. Také bude představen simulační server *PNtalk 2.0*, který je konkrétní implementací jazyka PNtalk.

2.1 Petriho sítě

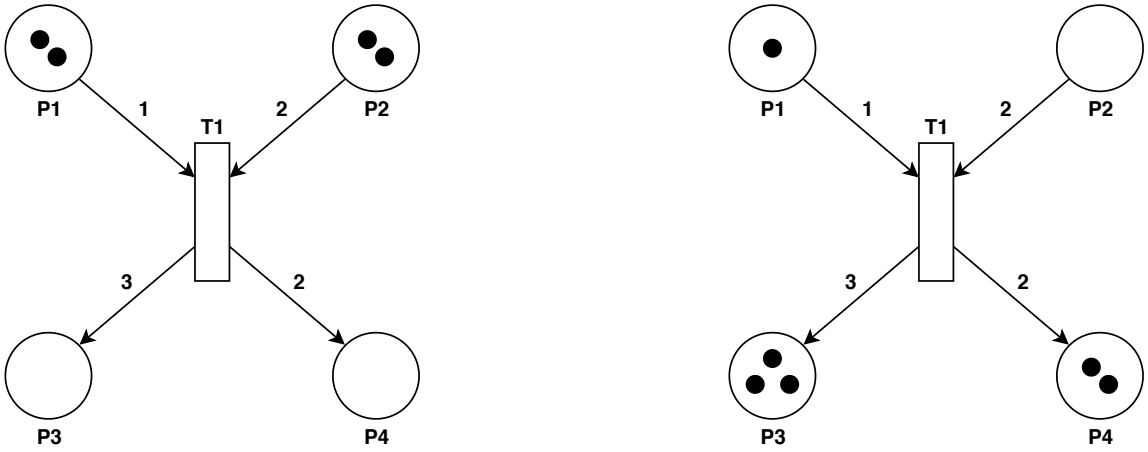
Petriho sítě (Petri nets – PN) představují populární formalismus pro modelování diskrétních (paralelních) systémů, který spojuje výhody srozumitelného grafického zápisu a možnosti simulace s dobrou formální analyzovatelností [1]. Provedení analýzy systémů modelovaných pomocí PN umožňuje identifikaci řady jejich vlastností již v počátečních fázích návrhu a vyhnout se tak problémům vzniklým později.

PN má přesný grafický zápis, který bude použit k neformálnímu zavedení jednotlivých prvků. PN se skládá ze dvou prvků: ze *sítě* a z *počátečního značení*. Síť představuje orientovaný graf se dvěma typy uzlů, kde mezi žádnými dvěma uzly stejného typu nemůže vzniknout hrana. Při vytváření sítě užíváme následující elementy:

- **Místa (places)** vyjadřují jednotlivé stavy modelovaného systému. K jejich značení se obvykle používá kružnice a pro označení pak písmeno P v kombinaci s pořadovým číslem místa v síti.
- **Přechody (transitions)** slouží k vyjádření změny v modelovaném systému. Obvykle se jako značka přechodu používá obdelník, příp. úsečka. Označují se složením písmene T a pořadového čísla přechodu v síti.
- **Hrany (arcs)** musejí být vždy orientované a také platí, že žádná hrana nesmí spojovat dva uzly stejného typu. Tedy musí vždy platit spojení *místa* a *přechodu*. Jedno *místo* s jedním *přechodem* může být spojeno pouze jednou *hranou* v každém z obou směrů. *Hrany* se zakreslují většinou úsečkami nebo křivkami zakončenými šipkou na jedné straně k udání směru. Tyto úsečky, resp. křivky mohou být dále označeny přirozeným číslem, které označuje *váhu hrany*, tedy jaké množství *značek* prochází *hranou*.
- **Značení sítě (tokens)** je vyjádřeno pomocí tzv. *značek (tokens)* aktuální stav modelovaného systému. Tyto *značky* jsou považovány za vzájemně nerozlišitelné. K jejich

zápisu se používají malé kruhy v jednotlivých místech sítě. V případě výššího počtu *značek* v jednom *místě* se použije přirozené číslo. Nachází-li se modelovaný systém v počátečním stavu, je takové *značení* označováno jako *počáteční značení*.

[2]



Obrázek 2.1: Petriho síť před a po provedení jednoho kroku simulace.

Každý *přechod* má definována vstupní a výstupní místa, která jsou v grafu PN vyjádřena *orientovanými hranami* mezi *místy* a *přechody*. Těmito místy je deklarováno, které aspekty stavu systému podmiňují výskyt odpovídajícího provedení přechodu, a které aspekty stavu jsou výskytem této události ovlivněny. Jinými slovy má tedy každý *přechod* definovány své vstupní a výstupní podmínky. Provedení přechodu (viz obr. 2.1) se tedy projeví odstraněním *značky*, resp. značek, ze vstupních míst, což zapříčiní neplatnost vstupních podmínek, a umístěním značek do výstupních míst, čímž se uplatní výstupní podmínky. Jak bylo popsáno výše (viz Hrany 2.1), tak vstupní a výstupní podmínky *přechodů* upřesňují počet *značek*, které budou odebrány, resp. přidány. Provedení přechodu je atomickou operací odpovídající výskytu události. [1]

2.2 Objektově orientované Petriho sítě

Oběktově orientované Petriho sítě (OOPN) představují formalismus spojující výhody Petriho sítí a všechny atributy objektově orientovaného jazyka, tj. umožňuje polymorfismus, zapouzdření a dědičnost. Dále také OOPN poskytuje výkonné prostředky k popisu paralelismu.

Model systému OOPN, ve kterém mohou dynamicky vznikat a zanikat objekty, jenž komunikují prostřednictvím zasílání zpráv, je specifikován množinou tříd těchto objektů. Tato množina tříd je hierarchicky uspořádána podle vztahu dědičnosti. Jedna ze tříd je prohlášena za počáteční a jako taková je vždy instanciována při spuštění simulace. Každá z těchto tříd je tvořena:

- sítí objektu, která definuje reprezentaci instancí třídy a jejich nezávislou aktivitu
- množinou sítí metod, jenž definuje reakce na zasílané zprávy z akcí přechodů

- množinou synchronních portů, která definuje reakce na zasílané synchronní zprávy ze stráží přechodů

[1]

2.3 Systém a jazyk PNtalk

Systém a jazyk PNtalk je založen na oběktově orientovaných Petriho sítí. Jméno PNtalk je odvozeno ze slovního spojení *"Petri nets & Smalltalk"*. Implementace jazyka PNtalk přímo vyzchází z definice OOPN[1], avšak má své některé syntaktické odlišnosti inspirované jazykem Smalltalk. Definice OOPN připouští u některých skutečností určitou volnost, kterou jazyk PNtalk upřesňuje. Jedná se především o hierarchii dědičnosti tříd a primitivní objekty. Dalším rozšířením, které PNtalk přináší, je jednodušší zápis určitých složitějších struktur, jakými jsou seznamy, konstruktory a složené zprávy.

Programování v *jazyce PNtalk* je založeno na vytváření tříd neprimitivních objektů a jejich následné zařazování do hierarchie dědičnosti. Tyto třídy jsou definovány množinami Petriho sítí, které jsou složeny z míst a přechodů propojených hranami. Místa, přechody i hrany mohou být pojmenovány.

Systém PNtalk představuje počítačový nástroj, umožňující prakticky využít OOPN a jazyk PNtalk, umožňující specifikovat model, testovat jej a provádět nad ním simulační experimenty. Konkrétní implementace tohoto systému se mohou v různých detailech a případných rozšíření lišit. Obecně by však pro každou implementaci systému PNtalk mělo platit, aby umožňovala vytváření a editaci OOPN a také provádění simulací OOPN jak automaticky, tak interaktivně. [1]

Systémem a jazykem PNtalk se detailně zabývá konkrétně kapitola 6 v práci docenta Vladimíra Janouška [1].

2.4 Simulační server PNtalk 2.0

Nástroj *PNtalk 2.0* je experimentální software určený především k výuce a výzkumu. Jedná se o simulační a modelovací framework využívající formalismy Oběktově orientovaných Petriho sítí, tedy kombinaci vlastností Petriho sítí a výhod objektově orientovaného návrhu systémů. *PNtalk 2.0* je rozšířením původního nástroje *tool PNtalk*. Aplikace umožňuje provádět simulace nad již existujícími modely a sbírat nad nimy výsledky a nebo statistiky. [3]

Komunikační protokol

Server komunikuje po počítačové síti výhradně prostřednictvím jazyka PNtalk. Má definovanou sadu příkazů, kdy každý příkaz odpovídá funkci serveru. Každý z příkazů má jasně specifikované upřesňující parametry. Úplný výčet všech podporovaných příkazů se nachazí v příloze C. Příkazy jsou načítany po řádcích, což vede k zadávání příkazu a parametru separatně na jednotlivé řádky. Server neumožňuje zpracování složených příkazů, tedy skriptů.

[1] Formální definice OOPN má několik úrovní, které jsou specifikovány v kapitole 5 v práci [1].

Notace zápisu

Forma zápisu protokolu:

```
(#nazev_pozadavku, parametr1, parametr2, ...) => (status, res1, res2, ...)
```

Na levé straně se nachází požadavek zasílaný na server, na pravé straně je odpověď serveru na požadavek. Jak již bylo řečeno, jednotlivé složky jsou odděleny koncem řádku. První složka odpovědi osahuje vždy stav provedeného požadavku a to buď *OK* nebo *FAILED*. Jako ukázku, kterou lze otestovat na image[2] obsahující nástroj PNtalk 2.0, je možné uvést následující příklad příkazu *stats*, jehož funkcí je získání statistik simulace.

Příklad zadání požadavku:

```
$ telnet localhost 9999
stats
12
```

Příchozí odpověď serveru:

```
OK
('14'
  ('1@R1'
    ('1@object'
      ('p1' 0.0 0 1 0.0 0)
      ('p2' 0.0 0 1 0.0 0)
      ('p3' 0.0 0 1 0.3333333333333333 1)
      ('p4' 0.0 0 0.0 0.5 1)
    )
  )
)
```

Forma zápisu protokolu:

```
(#stats, R1) => (OK, ('14' ('1@R1' ('1@object' ('p1' 0.0 0 1 0.0 0) ('p2' 0.0
0 1 0.0 0) ('p3' 0.0 0 1 0.3333333333333333 1) ('p4' 0.0 0 0.0 0.5 1)))))
```

Odpověď serveru byla pro přehlednost naformátována. Ve skutečnosti by odpověď serveru neobsahovala konce řádků, jak je znázorněno v zápise protokolu. Veškeré příkazy ve formě protokolového zápisu se nacházejí v příloze C. Příklady použití jednotlivých příkazů s názornými odpověďmi serveru jsou popsány v příloze C.

[2]Pojmem *image* se označuje výraz pro obraz paměti, který se používá v souvislosti s virtuálními stroji.

Kapitola 3

Interpret

Tato kapitola uvádí do problematiky tvorby interpretů a jejich funkce. Budou zde představeny různé druhy přístupů k interpretaci programovacího jazyka. Také bude popsán rozdíl mezi interpretem a překladačem. A na závěr budou také zohledněny výhody používání interpretů.

Interpretem je označován obecně počítačový program, který umožňuje přímo vykonávat instrukce zapsané ve zvoleném programovacím nebo skriptovacím jazyce. Interpretace neprovádí předchozí převod zdrojového kódu do strojového jazyka procesoru, ani do jiného tvaru, který by bylo možné spustit. Místo toho se spoléhá an existenci jiného spustitelného programu, tzn. složeného výhradně ze strojových intrukcí. [4] Takový program je nazýván interpretem. Tento přístup umožňuje vytváření snadno přenositelného zdrojového kódu.

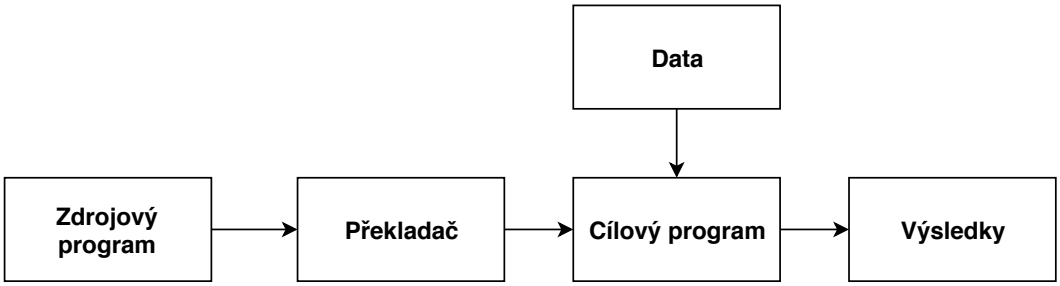
3.1 Rozdíl mezi překladačem a interpretem

Máme-li zdrojový kód zapsaný v některém z vyšších programovacích jazyků, existuje několik možných přístupů, jak jej spustit. Jednou z možností je zdrojový kód převést do odpovídajícího tvaru ve strojovém kódu počítače. V takovém případě se jedná o *kompilátory* nebo *kompilapilační překladače*. Druhou variantu pak představuje vytvoření programu, který bude interpretovat příkazy zdrojového jazyka tak, jak jsou zapsány, a přímo provádět k nim ekvivalentní operace. Programy této druhé varianty se pak nazývají *interprety* nebo také *interpretační překladače*. [6]

Kompilátor

Kompilace programu spočívá v tom, že zdrojový kód programu, který je zapsán ve vyšším jazyce, je jednorázově převeden na soubor strojových instrukcí počítače, na němž probíhá překlad. Program přeložený tímto způsobem, již následně může fungovat nezávisle na původním zdŕojovém programu. Přeložený program pak může být nadále samostatně opakovaně souštěn. V případě, kdy se má spouštět původní program popsaný vyšším programovacím jazyce, se místo něj provede spuštění již přeloženého programu. [4]

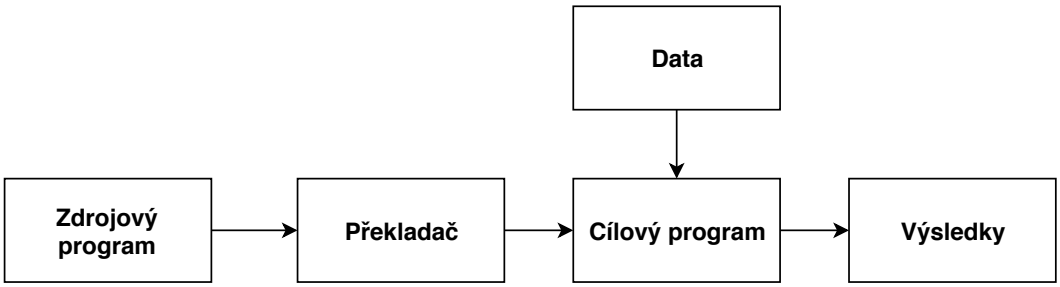
Výhodou kompilace je ta, že analýza zdrojového kódu a jeho následný překlad jsou provedeny jen jednou. Tato skutečnost je znázorněna v diagramu 3.1. Může se jednat o časově náročný proces, avšak následně se už spouští pouze odpovídající program zapsaný ve strojovém programu, který je výsledkem překladu. Nevýhodou pak může být obtížnější hledání chyb ve zdrojovém kódu, za předpokladu, že jsou dostupné informace o umístění chyby vyjádřené v pojmech strojového jazyka (např. adresy). [6]



Obrázek 3.1: Diagram kompilačního překladače.

Interpret

Oproti překladu zdrojového kódu, při interpretaci nedochází k převodu na kolekci strojových instrukcí počítače, ale příkazy zapsány ve zdrojovém programu jsou přímo vykonávány. V případě interpretace kódu se tedy neprovádí žádný převod nebo transformace zdrojového kódu. [4] Tato skutečnost je znázorněna v diagramu 3.2. Na místo toho jsou jednotlivé příkazy, které jsou zapsány ve vyšším programovacím jazyce, přímo interpretovány.



Obrázek 3.2: Diagram interpretačního překladače.

3.2 Výhody a nevýhody interpretu

Interpretace je mnohem pomalejší proces než kompilace. To je zapříčiněno převážně nutností analyzovat zdrojový příkaz pokaždé při jeho výskytu v programu. Další nevýhodou interpretu je náročnost na paměť. Ta je způsobena nutností mít stále k dispozici kompletní překladač během celého chodu programu.

Interprety však mají i své výhody. Při výskytu chyby ve zdrojovém kódu jsou k dispozici vždy přesné informace o jejím výskytu. Lze tak poměrně snadno a hlavně rychle odhalit příčinu výskytu této chyby. Takovýto přístum je vhodný zvláště v případě provádění ladění programů. Interprety některých jazyků (např. jazyk Prolog) umožňují provádění modifikací obsahu programu i v průběhu jeho chodu. U jazyků neobsahující blokové struktury kódu (např. BASIC) pak lze změnit část obsahu zdrojového kódu bez nutnosti znovu překládat zbylé části programu. Interprety jsou dále využívány v případech, v kterých je možné dynamicky měnit typy objektů v průběhu provádění programu (takovým jazykem je např. *Smalltalk-80*). Interpretační překladače jsou do značné míry strojově nezávislé, což je zapří-

činěno absencí generování strojového kódu. Pro přenos na jiný počítač obvykle stačí provést kompilaci interpretu. [6]

Přístupy uvedené výše jsou však extrémy. Mnoho překladačů využívá spíše jejich kombinace. Některé interprety nejprve převádí zdrojový kód do nějakého vnitřního stavu (např. do nějakého mezikódu). Tento vnitřní stav je následně interpretován. Takové řešení využívá kompromisu mezi časově náročným překladem kompilovaného programu a pomalým během interpretovaného programu. [6]

Kapitola 4

Návrh řešení a implementace

Tato kapitola se zaměřuje na návrh a realizaci konkrétního řešení vyvíjeného systému. Nejprve budou popsány požadavky na systém, z kterých návrh systému vyvhází. Dle počátečních specifikací, budou následně prezentovány návrhy jednotlivých částí projektu, v nichž budou na závěr rozebrány jejich implementace včetně vyskytlých problémů a jejich řešení.

4.1 Analýza požadavků

Cílem projektu je vytvoření robustního spojení s nástrojem *PNtalk 2.0* implementovaným na virtuálním stroji Pharo zajišťující odolnost vůči uživatelským chybám a zároveň také podporu jak jednoduchých textových příkazů, tak i jejich dávkových souborů. Jelikož má požadovaný systém nabízet vzdálenou správu nástroje pomocí textových příkazů, tedy jedná se o přímou interpretaci zdrojového kódu (v tomto případě příkazů), není potřeba využívat žádných podpůrných nástrojů, které by byly příliš komlikované pro dané řešnení a mohly by zpomalovat samotnou interpretaci. K návrhu projektu se dá přistupovat jako ke dvoum samostatným částem, tedy jako k síťové komunikaci a příkazovému interpretu.

Z pohledu síťové komunikace je důležité zabývat se jakým typem spojení bude komunikace procházet a dále také jakou strukturu bude mít její jazyk resp. komunikační protokol. Protokol by měl, dle specifikací, zahrnovat náležitosti položek v hlavičce k identifikaci odesilatele požadavků, kontrolní součet (*checksum*) k ověření celistvosti přijaté zprávy a časové razítko (*timestamp*).

Systém má být rozdělen na klientskou a serverovou část. Od serverové části jsou požadovány dvě základní funkcionality. První z těchto funkcionalit je komunikace s klientskou částí systému. Na straně serveru se má nacházet obsluhovaný simulační nástroj, měla by být tedy samozřejmostí schoptnost komunikace s více uživately zároveň. Druhou požadovanou funkcionalitou je schopnost spolupráce se simulačním nástrojem *PNtalk 2.0*. Tento nástroj je implementován v prostředí *Pharo 5.0*, které je konkrétní implementací prostředí a jazyka *Smalltalk*. Prostředí *Pharo 5.0* je tedy implicitně vyžadováno i pro implementaci serverové části tohoto systému.

Klientská část systému má být umístěna na zařízení uživatele a vytváří prostředníka mezi serverem a příkazovým interpretem. Implementace je požadována v programovacím jazyce C++. Ovládána má být pomocí textových příkazů.

Příkazový interpret by měl pokrývat implementovanou funkcionalitu simulačního nástroje. Z tohoto hlediska by měla být uvažována možnost budoucího rozšíření množiny

příkazů a k tomu uzpůsoben návrh. Jelikož je smyslem provadění simulací získání nových dat o řešeném problému a ne každá operace nad modelem je možná řešit triviálně jedním příkazem, je nutno uvažovat také příkazy umožňující zjednodušení práce podporováním dávkových souborů (tvorby skriptů) a také práci se vstupy a výstupy k ukládání získaných informací. Jedním z požadavků bylo také zpracování a odesílání po jednom příkaze.

4.2 Komunikační protokol

Protokol v kontextu počítačových sítí označuje konvence, resp. standard, podle kterého probíhá elektronická komunikace a přenos dat mezi dvěma koncovými body v síti. Protokol je tedy nedílnou součástí každé počítačové komunikace, jelikož každou takovou komunikaci obstarává. Před samotným návrhem protokolu byly požadavky na něj konzultovány s vedoucím práce a definovali jsme základní specifikace hlavičky zprávy.

První z nich je opatření hlavičky zprávy časovým razítkem (*timestamp*), aby bylo možné uvažovat ukončení komunikace po určité časové prodlevě mezi příchozími zprávami. Další nezbytnou součástí hlavičky zprávy je kontrolní součet (*checksum*) spočítaný na základě atributů hlavičky zprávy, čímž je zajištěna integrita přijatých dat. Poslední požadovanou součástí hlavičky jsou atributy, které umožní unikatní identifikaci konkrétní komunikace.

Součástí každé zprávy je hlavička obsahující důležité informace pro oba komunikující body. Tyto informace především slouží ke kontrole, zda zpráva došla bez chyb, nebyla podvržena nebo k identifikaci odesilatele. Formát hlavičky zprávy byl navržen následovně:

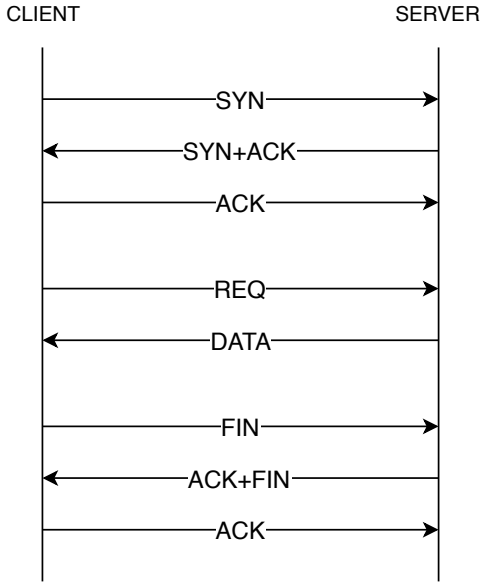
```
NAME: <jmeno_protokolu>
FLAG: <priznak_zpravy>
TIME: <casova_znamka>
CSUM: <vypocteny_checksum_z_hlavicky>
SQNB: <sekvencni_cislo_odesilatele>
ACKN: <sekvencni_cislo_prijemce>
DATA: <datovy_obsah_zpravy>
```

Specifikované položky hlavičky byly rozšířeny o další položky. Název protokolu (*NAME*) má statickou hodnotu *"PNtalkShellProtocol"*. K indentifikaci komunikace byla zvolena dvojice atributů sekvenčního (*SQNB*) a potvrzovacího čísla (*ACKN*). Příznak zprávy (*FLAG*) může nabývat až osmi různých hodnot, odpovídajícím jednotlivým zprávám poslouponosti komunikačního protokolu. Význam jednotlivých příznaků je následující:

- `SYN` – inicializační zpráva spojení, která je vytvořena při spuštění systému na straně klienta. Tato zpráva zasílá vygenerované sekvenční číslo pro klientský komunikační bod.
- `SYN+ACK` – synchronizační zpráva, kterou vytváří server. Tato zpráva zasílá vygenerované sekvenční číslo pro server.
- `ACK+REQ` – zpráva se synchronizovanými sekvenčními čísly, která obsahuje verifikovaný požadavek uživatele v datové části zprávy. Tato zprává je zasílána klientem.
- `REQ` – zpráva obsahující verifikovaný požadavek uživatele v datové části zprávy. Tuto zprávu zasílá klient.

- `DATA` – zpráva obsahující odpověď simulačního nástroje v datové části zprávy.
- `FIN` – inicializační zpráva k ukončení spojení se serverem.
- `ACK+FIN` – potvrzovací zpráva serveru o ukončení spojení.
- `ACK` – potvrzovací zpráva klienta o ukončení spojení.

Při vytváření protokolu je také důležité navrhnout určitou posloupnost zpráv. Komunikace je zahájena klientem zasláním zprávy s příznakem zprávy *SYN*. Na tuto zprávu reaguje server zasláním své synchronizační zprávy s nastaveným příznakem *SYN+ACK*. Touto sekvencí zpráv jsou nastaveny atributy hlavičky potřebné k identifikaci komunikace. Tato výměna zpráv probíhá automaticky bez zásahu uživatele. Klient poté potvrzuje nastavené hodnoty v hlavičce zprávou s příznakem *ACK*. Nyní nastává obslužná rutina zpráv, která zajišťuje výměnu dat mezi klientem a serverem. Klient zasíla zprávy s příznakem (*REQ*), obsahující požadavek v podobě textového příkazu v datové části. Reakcí serveru na tuto zprávu je odpověď zprávou s příznakem *DATA*, která v datové části zprávy obshauje výsledek požadované operace. Tuto obslužnou rutinu ukončuje klient zadáním příkazu k ukončení komunikace, zaslaném ve zprávě s příznakem *FIN*. Touto zprávou je zahájena automatická sekvence zpráv pro ukončení spojení. Server potvrzuje zprávu svou odpovědí s příznakem *ACK+FIN*, která ja nakonec potvrzena i ze strany klienta zprávou s příznakem *ACK*. Pro přehlednost je posloupnost jednotlivých zpráv, identifikovaných příznaky, znázorněna v následujícím sekvenčním diagramu komunikačního protokolu 4.1.



Obrázek 4.1: Sekvenční diagram navrženého komunikačního protokolu.

Kontrolní součet

Kontrolní součet (angl. *checksum*) je hodnota, která se zasílá společně s vlastní infomací a slouží k ověření, zda je tato informace úplná nebo jestli se při jejím přenosu nevyskytla

chyba. Kontrolní součet je výsledek předem zvolené operace provedené nad vlastní informací. Příjemcem má možnost kdykoliv provést danou operaci, aby získal kontrolní součet, a ověřil tak, zda se kontrolní součet shoduje s tím přijatým. V případná neshoda součtů znamená chybu v přenosu vlastní informace.

Pro vytvoření kontrolního součtu byla použita hashovací funkce `MD5`, která je popsána v internetovém standardu RFC 1321 [5]. Tato funkce je vhodná mimo jiné právě také pro ověřování integrity souborů.

V případě navrženého protokolu je kontrolní součet spočítán nejprve konkatenací řetězců určitých položek hlavičky společně s datovou složkou zprávy. Tato sekvence znaků je následně zahashována funkcí `MD5` a výsledná hodnota přidána do hlavičky na patřičné místo. Přesný výpočet kontrolního součtu je vyjádřen tímto výrazem, kde operátor `"+"`je operací `konkatenace`:

```
MD5(<priznak_zpravy> + <casova_znamka> + <sekvencni_cislo_odesilatele> +
<sekvencni_cislo_prijemce> + <datovy_obsah_zpravy>)
```

Identifikace spojení

Identifikace spojení je definována dvojicí sekvenčního (*SQNB*) a potvrzovacího čísla (*ACKN*). Při vytvoření inicializační zprávy komunikace *SYN* je vygenerováno číslo pro klientský komunikační bod. Po přijetí této zprávy serverem je sekvenční číslo klienta inkrementováno a použito jako potvrzovací číslo serveru. Při vytvoření potvrzovací zprávy serveru (*SYN+ACK*) je vygenerováno sekvenční číslo pro serverový komunikační bod. Následně při každé další výměně zpráv jsou tato čísla prohozena, vzhledem ke svému odesilateli. Následně jsou jejich hodnoty inkrementovány. Počáteční vygenerování sekvenčních čísel je zajištěno náhodným generátorem v intervalu $< 1; 100 >$.

Třída Protocol

Tato podkapitola je věnována popisu implementace třídy *Protocol*. Tato třída je implementována v jazyce C++. Třída představuje prostředníka mezi koncovými komunikačními body. V této třídě je definována kompletní logika navrženého komunikačního protokolu včetně operací ke správnému chodu komunikace. Třída obsahuje kompletní strukturu hlavičky zprávy protokolu:

- `protocolName` – obsahuje název protokolu, který je inicializován v konstruktoru třídy a dále se již nemění.
- `messageFlag` – obsahuje aktuální příznak přijaté, resp. odesílané zprávy. Tato hodnota je inicializována v konstruktoru na počáteční synchronizační zprávu *SYN*. Následně se tato hodnota mění v závislosti na typu přijaté zprávy podle definované posloupnosti zpráv (obr. 4.1).
- `checkSum` – obsahuje aktuální kontrolní součet zprávy vypočtený podle algoritmu uvedeném výše (4.2), který je imlementován metodou *GenerateCheckSum*.
- `sequenceNumber` – obsahuje aktuální sekvenční číslo daného koncového komunikačního bodu, jehož hodnota je inicializována v konstruktoru metodou *GenerateSequenceNumber* a dále se pak mění podle principu popsaném v podkapitole 4.2.

- `acknowledgeNumber` – obsahuje aktuální potvrzovací číslo daného koncového komunikačního bodu, jehož hodnota je inicializována v konstruktoru na hodnotu 1. Tato hodnoto je následně vygenerována na straně serveru a zaslána klientské straně v synchronizační zprávě *SYN+ACK*.

- `timeStamp` – obsahuje časové razítko, které je vygenerováno pomocí knihovny *time.h*, při vytvoření zprávy.

V třídě *Protocol* je implementováno veřejné rozhraní se základními operacemi s protokolem. Tyto metody slouží k vykonání jednotlivých kroků výměny zpráv mezi zařízeními. Toto veřejné rozhraní třídy je implementováno následujícími metodami:

- `CreateMessage` – je metoda, která slouží k sestavení zprávy z proměnných třídy, reprezendující hlavičku zprávy. Metoda přijímá jako vstupní parametr řetězec se zasílanými daty, který je vložen do datové části zprávy. Při vytvoření zprávy je vygenerováno časové razítko (*timestamp*). Zpráva je vytvořena konkatenací jednotlivých částí řetězce.

- `SendMessage` – je metoda, která se stará o odesílání zpráv.

- `ReceiveMessage` – je metoda, jejíž úkolem je mimo přijetí zprávy také kontrola atributů hlavičky přijaté zprávy. Kontrolují se sekvenční a potvrzovací čísla, která jsou následně přehozena a inkrementována, jak je popsáno výše. Také je provedena kontrola správného typu zprávy v pořadí.

- `GetDataContent` – je metoda, která vrací načtená data z přijaté zprávy.

- `GetEndCommunication` – je metoda, která vrací hodnotu příznaku značící ukončující sekvenci zpráv.

4.3 Klient

Klient představuje jeden z koncových komunikačních bodů v počítačové síti. Nejběžnějším modelem komunikace v takové síti je komunikace typu klient-server. Klient je v tomto modelu iniciátorem komunikace, kdežto server jen reaguje na požadavky klienta. Komunikační bod je indetifikován IP adresou a síťovým portem.

Při spuštění klienta jsou nejprve zpracovány spouštěcí parametry programu. Z těchto parametrů je získána IP adresa a síťový port identifikující server.

Komunikace je iniciována zasláním synchronizační zprávy s příznakem *SYN* serveru na IP adrese a portu, získaných ze zadaných parametrů programu. Touto zprávou je vygenerováno sekvenční číslo pro klienta. Po přijetí zprávy od serveru s příznakem zprávy *SYN+ACK* dostává klient i své potvrzovací číslo.

Po inicializaci těchto dvou čísel klient zavolá interpret, který provádí zpracování zadaných příkazů. Zpracovaný příkaz je předán zpět klientovi, který vytvoří zprávu s příznakem *REQ* a příkaz připojí do datové části zprávy. Klient vždy odesílá pouze jeden požadavek a čeká dokud nedostane zpět odpověď. Jelikož formát datové části v odpovědi serveru vychází ze stejného formátu, jaký využívá simulační nástroj *PNtalk 2.0*, je z přijaté zprávy od serveru oddělena její datová část, která je předána interpretu k pochopitelnějšímu naformátování. Teprve poté je výsledek prezentován uživateli.

Při zadání požadavku k ukončení spojení, je nastaven příznak *FIN* odesílané zprávy serveru. Zároveň je při této zprávě přerušena oslužná linka volající interpret a spojení je automaticky ukončeno.

V tomto návrhu klient tvoří prostředníka mezi počítačovou sítí a interpretem (kapitola 4.5). Při návrhu bylo využito rozhraní pro programování aplikaci (*API*) *Socket API*, které je obvykle poskytováno operačním systémem a umožňuje programům používat síťové schránky (*angl.* sockets). Tyto sockety vytvářejí strukturu, která identifikuje koncové zařízení v počítačové síti síťovým portem a IP adresou.

Třída Client

Tato podkapitola je věnována implementaci třídy *Client*. Třída je implementována v programovacím jazyce C++. Úkolem třídy je vytvoření síťového komunikačního bodu pomocí API *socket*. Pro tuto strukturu třída obsahuje proměnné *Host* a *Port* sloužící k identifikaci cílového zařízení. Tyto hodnoty se nemohou při probíhající komunikaci měnit (jedinou možností je restartování serveru), a proto jsou proměnné určeny uživatelem již při spuštění programu.

Metoda *ProgramArguments* slouží ke zpracování spouštěcích parametrů programu. Podporuje zadání parametrů v dlouhém (*long_option*, např. *"–port"* pro nastavení portu) i krátkém (*short_option*, např. *"-p"* pro nastavení portu) tvaru. Zároveň metoda umožňuje zadání parametrů v libovolném pořadí. Touto metodou jsou nastaveny identifikační údaje potřebné k navázání spojení se serverem.

Metoda *CreateSocket* zajišťuje inicializaci *socketu* definovaného zadanými parametry třídy. Následně je provedeno připojení k definovanému síťovému bodu. Je-li spojení úspěšné, metoda vytvoří přístupový bod k serveru.

Metoda *CloseConnection* je využívána pouze k uzavření již vytvořeného síťového komunikačního bodu se serverem.

Třída také implementuje metody umožňující přístup k chráněným třídním proměnným *Port* a *Host*, tzv. *gettery* a *settery*. Tyto metody zajišťují lepší zapouzdření objektu. Funkcionalitou metody *GetHost* je načtení hodnoty třídní proměnné *Host*. Metoda *SetHost* umožňuje nastavení hodnoty třídní proměnné *Host*. Metoda *GetPort* umožňuje načtení hodnoty třídní proměnné *Port*. Metoda *SetPort* umožňuje zápis hodnoty do třídní proměnné *Port*.

Tato třída dědí z třídy *Protocol*, jejíž funkcí a jazyka využívá. Klient těmito metodami řídí průběh komunikace se serverovou částí systému během celého chodu programu.

4.4 Server

Úkolem serverové části systému je přijímání uživatelských požadavků skrze navržený komunikační protokol a také vyvolání příslušné funkce simulačního nástroje Petriho sítí *PNtalk 2.0*. Jedinými výstupy této programové části budou výsledky požadovaných operací, které jsou okamžitě zasílány zpět na klientskou část, a nebo chybová hlášení při výskytu chyb v síťové komunikaci s klientem, kterou jsou logovány. Program by měl umožňovat změnu portu, na kterém bude naslouchat. Jelikož server bude přímo komunikovat se simulačním nástrojem, bude muset být implementována znovu logika komunikačního protokolu, tentokrát však v programovacím jazyce *Smalltalk*. Kvůli zvýšení rychlosti vyřizování požadavků, tedy úspory času vynaloženého na kontrolu přijatých dat, a snížení režie síťového přenosu by měl server dostávat přímo verifikované požadavky, jejichž funkčnost by měla být oka-

mžitě vyvolána. Přijímané požadavky budou přijímány v jednotlivých zprávách i v případě zpracování dávkového souboru (skriptu).

Třída PNtalkserver

Tato podkapitola je věnována implementaci třídy *PNtalkServer*. Serverová část systému je implementována v jazyce *Smalltalk*, konkrétně s využitím konkrétní open source implementace tohoto programovacího jazyka *Pharo 5.0*. Implementace serveru se kvůli využití tohoto prostředí nachází na tzv. *image*[1] souboru, k jehož spuštění je potřeba právě *Pharo 5.0*. Metody jsou rozděleny do dvou protokolů[2].

Protokol *internal* sdružuje metody s takzaným *private*[3] přístupem. Tyto podpůrné metody implementují kompletní logiku navrženého protokolu. Jde tedy o metody ve smyslu kontrol hlavičkových atributů, automatických přepínačů typů zpráv a sekvenčních čísel, či výpočtu kontrolního součtu.

Metoda *checkSum* slouží k výpočtu kontrolního součtu přijaté i odesílané zprávy. Jako argument přijímá zprávu, nad kterou má být kontrolní součet proveden. Výpočet kontrolního součtu je uveden v podkapitole návrhu komunikačního protokolu 4.2.

Metoda *createMessage* slouží k sestavení zprávy. Nejprve je vygenerováno časové razítko (*timestamp*). Následně je vypočítaný kontrolní součet (*checksum*). Nakonec je provedena konkatenace jednotlivých částí zprávy, reprezentované řetězci znaků.

Metoda *getMessageData* přečte obsah datové části zprávy. Tento obsah je rozdělen podle řádků do pole, v kterém je každá položka jeden řádek. V tomto poli je pak obsažen přijatý požadavek a jeho atributy.

Metoda *getMessageHeaderItemValue* umožňuje přečíst konkrétní hodnotu položky hlavičky zprávy. Vstupnímy parametry metody jsou čtená zpráva společně s pořadovým číslem atributu hlavičky. Každý atribut hlavičky je na samostatném řádku, který určuje pořadí atributu. Hodnota atributu je přečtena vyhledáním názvu atributu hlavičky, jeho přeskočením, a následného načtení až po konec řádku.

Metoda *performRequest* slouží k invokaci příslušné metody, definované přijatým požadavkem. Požadavek i s jeho parametry jsou předány v poli řetězců, do kterých byl zpracován metodou *getMessageData*. Před invokací příslušné metody je nejprve zkontrolován příznak (*FLAG*) přijaté zprávy, který umožňuje zasílání požadavků. Daná metoda je následně invokována využitím funkce *perform*.

Metoda *sequenceNumberSwitch* přečte sekvenční a potvrzovací čísla z hlavičky přijaté zprávy. Tyto hodnoty jsou následně zkontrolovány, zda odpovídají, a poté jsou prohozeny a inkrementovány (viz. kapitola 4.2). Vstupním argumentem je přijatá zpráva.

Metoda *typeControl* slouží k provedení ověření správnosti příznaku přijaté zprávy a následné srovnání s předchozím uloženým příznakem (*FLAG*), aby bylo ověřeno neporušení posloupnosti zpráv. Vstupním argumentem je přijatá zpráva.

Metoda *typeSwitch* následuje po metodě *typeControl*. Funkcí této metody je pouhé provádění změn hodnoty příznaku (*FLAG*) zprávy v reakci na zprávu přijatou.

Protokol *control* sdružuje metody implementující obsluhu požadavků a také metody, propojující server se simulačním nástrojem. Metoda *serve* tvoří jádro obsluhy požadavků. Využívá metod z protokolu *internal*, který je popsán níže, ke zpracování příchozích zpráv,

[1] V souvislosti s virtuálním prostředím se jedná o otisk paměti, který je možné samostatně spustit.

[2] Prostředí Pharo umožňuje rozdělení metod do takzvaných *protokolů*, jejichž smyslem je kategorizace metod do logických celků.

[3] V programovacím jazyce Smalltalk je ke všem metodám přístup *public*. V tomto případě jde tedy převážně o přehlednost v metodách.

tedy kontroly hlavičky zprávy a následně také parsování datové části zprávy. Dále tento protokol obsahuje metody jednotlivých funkcí nástroje.

K volání příslušných metod je zvolena funkce *perform*, která umožňuje invokaci metody podle přijatého názvu jako parametr. Při zpracování příchozího požadavku proto server spoléhá na přesný název příchozího požadavku. Tento problém řeší umístění příkazového interpretu na klientskou část systému, který zodpovídá za korektní tvar požadavku. Názvy příkazů jsou inspirovány příkazy simulačního nástroje *PNtalk 2.0* 2.4.

4.5 Příkazový interpret

Intepretem se nazývá program, jež umožňuje přímo vykonávat (interpretovat) zdrojový kód jiného programu zapsaný ve zvoleném programovacím jazyce. U interpretu tedy nevzniká nutnost převádět zdrojový kód do strojového kódu procesoru, což je případem překladače. Interpret je v praxi využit například pro rozhraní *Shell*, které vytváří příkazový řádek. Tento návrh se tedy bude snažit o vytvoření podobného rozhraní jako je příkazový řádek.

Příkazový interpret je umístěn na klientské části systému, kvůli snížení režie síťové komunikace a obejmu přenášených dat po síti mezi klientem a serverem. Díky umístění interpretu na klientskou část systému navíc dochází k minimalizaci zátěže na serverovou část systému. Interpret je spouštěn klientskou částí systému v obslužné smyčce až po ustavení spojení se serverem synchronizační sekvencí zpráv. Klient předává intepretu pouze celou datovou část zprávy. Interpret pak celé zpracování řídí sám. Následně po zpracování je výsledek intepretu předán klientovi, který jej připojí do zprávy a zašle serveru. Pokud se klientovi nepodaří navázat spojení se serverem, interpret není volán, a program je ukončen.

Součástí interpretu je jednoduchý parser vstupních dat. Jeho funkcionalita spočívá v rozdělení vstupních dat na jednotlivé příkazy. Příkazy nejsou nijak spracovávány, jsou pouze ukládány v podobě v jaké je uživatl zadal, se všemi parametry do textového řetězce. Tyto textové řetězce jsou pak za účelem efektivního zpracování skriptů uloženy v podobě vektoru řetězců.

Tyto řetězce jsou zpracovávány postupně, tedy příkaz se nedostane ke zpracování, dokud není dokončené zpracování všech příkazů, které mu předcházejí. Pokud je některý příkaz chybně zadán, je zpracování zbylých příkazů ukončeno a uživateli je hlášena chyba. V této fázi zpracování nejsou nijak ošetřeny zadání chybných parametrů požadavků nebo zadání sémanticky nesmyslného příkazu. Na tyto situace reaguje až serverová část systému.

Textový řetězec s příkazem je následně rozdělen na jednotlivé lexémy, které jsou uloženy ve formě textových řetězců do vektoru. Z tohoto vektoru je následně vybrán první prvek, který by měl vždy představovat příkaz pro simulační nástroj *PNtalk 2.0*. U zadávání příkazů uživatelem není kladen důraz na velikost písmen (*case-sensitive*). Za účelem zjištění validity příkazu jsou znaky tohoto příkazu následně převedeny na malé znaky. Takto upravený příkaz je následně zkontrolován, zda je validním příkazem. V případě, že ano, je převeden do tvaru vyžadovaného simulačním nástrojem. Pokud příkaz nebyl nalezen, je ukončeno zpracování a uživateli je hlášena chyba.

Prvky takto upraveného vektoru jsou následně spojeny do jednoho řetězce, který představuje výsledný obsah datové části zprávy. Řetězec obsahuje na každém řádku jednu část příkazu (na jednom řádku se nachází příkaz, na dalším jeho parametr). Tento výsledný řetězec je předán zpět klientovi. Klient připojí tento řetězec se zpracovaným příkazem do datové části zprávy a odešle požadavek serveru.

Po přijetí zprávy serveru je její datová část předána interpretu. Tato data jsou přijata bez jakéhokoliv formátování. Interpret taková data umožňuje formátovat do přehlednější formy,

jako je použití odřádkování pro logické celky a užití tabulátorů pro zvýraznění zanoření jednotlivých bloků kódu. Dále také umožňuje uložit výsledek volané operace serveru do předem vytvořené proměnné. Tato porměnná pak může být použita ke srovnávání výsledků různých operací. Výstupy přijatých zpráv od serveru mohou být přesměrovány do souboru, na standartní výstup a nebo kombinaci obou.

Třída Interpreter

Tato podkapitola se věnuje implementaci třídy *Interpreter*. Třída je implementována v programovacím jazyce C++. Třída představuje jádro tohoto interpretačního systému. V této třídě je definováno prostředí pro interakci s uživatelem spolu s kompletní logikou interpretace uživatelských příkazů.

Ke kontrole uživatelských příkazů jsou definovány tři konstatní vektory. Vektor *validCommands* představuje seznam příkazů přijímaný serverem, jejichž názvy jsou upraveny do tvaru přijímaným serverovou částí systému. Dalším vektorem je *acceptableCommands*. Tento vektor obsahuje všechny přípustné uživatelské příkazy, jejichž názvy jsou zapsány malými znaky (*lower-case*). Přípustnými uživatelskými příkazy jsou jak příkazy přijímané serverovou částí systému, tak příkazy operující tzv. "offline", tedy mají význam pro klientskou část systému. Posledním konstantním vektorem je vektor *parameterCount*. Tento vektor obsahuje povinný počet lexémů pro každý z podporovaných příkazů. Příkazy přijímané serverem jsou následující:

```
existModel, addClass, delClass, getClass, newSim, getState,
stepSim, simulateForSteps, registerEvent, simulateToEvent,
destroySim, stats, end
```

Dále je deklarován vektor *scriptByLanes*, který využívá *parser* víceřádkových složených příkazů, tzv. *skriptů*. Položky tohoto vektoru obsahují jednotlivé řádky načteného skriptu.

Posledním vektorem je vektor *parsedInput*, do kterého se ukládá výsledek *parseru* jednotlivých příkazů. Obsahem tohoto vektoru je pak rozdělený řetězec příkazu na jednotlivé složky, tedy na název příkazu a jeho jednotlivé parametry.

Posledními třídními proměnnými jsou *endFlag* a *standartInput*. Třídní proměnná *endFlag* je nastaven po zadání příkazu k ukončení interpretu a následně i komunikace se serverem. Třídní proměnná *standartInput* značí, zda má být čteno ze standartního vstupu nebo má být proveden dávkový soubor (skript).

V třídě *Interpreter* je implementováno veřejné rozhraní dvěma metodami. První je *getter* metoda *GetEndFlag*, která umožňuje načíst hodnotu chráněné třídní proměnné *endFlag*. Druhá metoda je klíčová pro celou třídu.

Metoda *GetCommand* implementuje textové uživatelské rozhraní systému. Metoda pracuje ve dvou režimech čtení uživatelského vstupu. Impicitním režimem je čtení ze standartního vstupu. Druhý režim je nastaven třídní proměnnou *standartInput*, která nabývá hodnotu *true* pro čtění ze standartního vstupu a nebo *false*, který značí čtení z dávkového souboru. Jedná se o řídící metodu, která po načtení vstupu volá podpůrnou metodu *InterpretCommand*, jejíž funkcí je zpracování načteného uživatelského příkazu.

Metoda *InterpretCommand* má vstupní parametr načtená uživatelský příkaz, který má být zpracován. Metoda předá nejprve příkaz parseru a následně ošetřuje stav, kdy byl příkaz chybně zadán, chybovým hlášením. Pokud byl načetný příkaz identifikován, jako příkaz pro ukončení obsluhy, metoda nastavuje příslušnou proměnnou *endFlag*.

Metoda *ReadUserInput* funguje jako *parser* jednotlivých příkazů, které jsou předány jako řetězec vstupním parametrem této metody.

Metoda *StrToLower* slouží jako konvertor znaků řetězce (v tomto případě příkazu) na malá písmena (*lower-case*). Vstupním parametrem je název příkazu získaný z *parseru*.

Metoda *CheckUserCommand* nejprve předá název příkazu obsažený ve vektoru *parsedInput* metodě *StrToLower*. Následně je provedena kontrola, zda se takto konvertovaný příkaz nachází ve vektoru přijatelných příkazů *acceptableCommands*. Pokud se v tomto vektoru nachází je uložen index nalezené shody s položkou vektoru. Tento index je následně využit ke kontrole počtu parametrů. Provede se srovnání počtu položek vektoru *parsedInput* a hodnoty položky vektoru *parameterCount* odpovídající načtenému indexu. Nyní je příkaz zkontrolován a už jen chybí rozhodout, zda bude zaslán serveru nebo se jedná o tzv. "offline"příkaz. Toto rozhodnutí je provedeno na základě vyhledání ve vektrou *validCommands*, který obsahuje příkazy podporované serverem. Pokud je takto příkaz identifikován je do výsledného řetězce vložena hodnota položky vektoru *validCommands* na odpovídajícím indexu, zjištěného z předchozí kontroly. K tomuto řetězci jsou následně připojeny parametry uloženy ve jako položky vektoru *parsedInput*. Každá položka výsledného řetězce, obsahujícího finální formát příkazu, je oddělena koncem řádku. Pokud by se v jakékoliv z předchozích kontrol vyskytla chyba, je oznámeno uživatli zadání chybného příkazu.

Metoda *SetInput* je *parserem* pro dávkové soubory. Ze zadaného souboru vtupním parametrem rozdělí jeho obsah po řádcích do vektoru *scriptByLines*.

Kapitola 5

Testování

Tato kapitola se zabývá testování vyvýjeného systému pro správu simulačního nástroje *PNtalk 2.0*. Testování při vývoji tohoto systému bylo prováděno výhradně na na lokálním serveru, který byl umístěn na virtuálním stroji *Pharo 5.0*. Budou zde popsány ukázkové testy základních operací s vyvýjeným systémem.

Serverová část systému bohužel nebyla propojena s reálným simulačním nástrojem. Toto propojení však není součástí mé práce. Pro účely testování tedy musejí být odpovědi serverové části systému simulovány statickými odpověďmi požadovaných operací.

Test jednoduchých příkazů

Výchozí stav je nově sputěný program. Ovládání probíhá zadáním příkazu do příkazové řádky.

První test existence modelu – testovaný model existuje.

> Vstup: `existModel R1`
> Výstup: `OK`
> Závěr: Testovaný model existuje. Předpokládaná odpověď serveru byla `OK`. Test je úspěšný.

Druhý test existence modelu – testovaný model neexistuje.

> Vstup: `existModel R2`
> Výstup: `FAILURE`
> Závěr: Testovaný model neexistuje. Předpokládaná odpověď serveru byla `FAILURE`. Test je úspěšný.

Test posloupnosti jednoduchých příkazů. Bude po sobě následovat několik jednoduchých příkazů.

Vstup: `newSim R1`
Výstup:
`OK`
`13`
Vstup: `stepSim 12`
Výstup: `OK`
Vstup: `existModel R2`
Výstup: `FAILURE`
Závěr: Jednotlivé příkazy reagovaly předpokládaným výstupem. Postupné zadávání jednotlivých příkazů nezapříčinilo žádnou chybu v chodu systému. Test proběhl úspěšně.

Test krokování simulace a zjištění stavu

Výchozí stav testu je nově spuštěný program. Ovládání probíhá zadávání příkazů do příkazové řádky. Cílem testu je ověření funkčnosti příkazů ke krokování simulací. Simulování kroků je provedeno postupnou inkrementací identifikátoru simulace.

Vstup: `newSim TestAckermann`
Výstup:
`OK`
`12`
Vstup: `getState 12`
Výstup:

```
((('TestAckermann' 1 (('object' 1 (('p1' ()) ('y' (2->#obj->2))
('p2' (1->#ref->2)) ('p3' ()) ('x' (1->#obj->1 1->#obj->2))) ())
) () ()) ('Ackermann' 2 (('object' 1 () ())) () ())))
```

Vstup: `stepSim 12`
Výstup: `OK`
Vstup: `getState 13`
Výstup:

```
((('TestAckermann' 1 (('object' 1 (('p1' (1->#obj->#e))
('y' (2->#obj->2)) ('p2' ()) ('p3' ()) ('x' (1->#obj->1
 1->#obj->2))) ())) () ())))
```

Závěr: Jednotlivé příkazy reagovaly předpokládaným výstupem. Ve výstupu se projevily změny vyvolané provedením kroku simulace. Test proběhl úspěšně.

Kapitola 6

Závěr

Cílem práce bylo vytvořit systém, umožňující provádět vzdálenou správu simulačního nástroje PNtalk 2.0. Tento nástroj je konkrétní implementací jazyka PNtalk, což je programovací jazyk Obejektově orientovaných Petriho sítí. Systém s nástrojem komunikuje prostřednictvím textových příkazů, o jejichž zpracování se stará příkazový interpret. Součástí řešení bylo navrhnout vhodný formát síťové komunikace. Návrh zahrnuje tvorbu klientské a serverové části systému a komunikačního protokolu, prosttřednictvím kterého budou tyto strany komunikovat. Příkazový interpret by měl pokrývat základní funkčnost práce s nástrojem. To zahrnuje příkazy pro spuštění, krokování a zobrazení stavu simulace a schopnost načtení modelů.

Výsledkem řešení je klientská strana systému spolupracující s příkazovým interpretem. Tyto části systému jsou implementovány v programovacím jazyce C++. Jejich návrh a implementace jsou popsány v kapitole 4.

Klient využívá navrženého komunikačního protokolu, který byl v průběhu vývoje otestován. Příkazový interpret podporuje všechny příkazy, které byly ještě rozšířeny o několik příkazů pouze pro ovládání nastaveni příkazového interpretu.

Serverová část systému byla implementována v programovocím jazyce Smalltalk, konkrétně v prostředí Pharo 5.0. Tento jazyk byl zvolen kvůli umístění simulačního nástroje PNtalk 2.0 v tomto virtuálním prostředí, s kterým by měl být server propojen. Bohužel propojení serveru není implementováno, jelikož se nejedná o součást mé práce. Server tedy pouze simuluje odpovědi na příkazy, za účelem otestování systému jako celku.

Literatura

[1] Janoušek, V.: *Modelování objektů Petriho sítěmi*. Disertační práce. Vysoké učení technické v Brně Fakulta informačních technologií, 1998.

[2] Jedlička, P.: *Qualifications of Petri nets for modeling of logistical systems*. Acta univ. agric. et silvic. Mendel. Brun., LIV, No. 3, pp. 131–136, 2006.

[3] Kočí, R.: *PNtalk 2.0*. 2018, [Online; navštíveno 15.3.2019].
URL `http://perchta.fit.vutbr.cz/pntalk2k/`

[4] Peterka, J.: *Compiler vs. Interpreter*. Computerworld č. 6/95, 1995.
URL `https://tools.ietf.org/html/rfc1321`

[5] Rivest, R.: *The MD5 Message-Digest Algorithm*. RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.
URL `https://tools.ietf.org/html/rfc1321`

[6] Češka, M.; Hruška, T.; Beneš, M.: *Překladače*. Vysoké učení technické v Brně Fakulta elektrotechnická, [Online; navštíveno 12.2.2019].
URL `http://www.fit.vutbr.cz/~meduna/fjp/skripta.pdf`

Příloha A

Obsah CD

```
/
├── doc/ ..............................  adresář obsahující zdrojové soubory
│                                        této písemné práce
│   ├── bib-styles/ ...................  složka obsahující styly literatury
│   ├── obrazky-figures/ ..............  složka obsahující obrázky a schémata
│   │                                    písemné práce
│   └── template-fig/ .................  složka obsahující obrázky šablony
├── src/ ..............................  složka obsahující zdrojové soubory
│                                        systému
├── README.txt ........................  README soubor obsahující návod ke
│                                        spuštění
└── xhuspe00.pdf ......................  elektronická podoba této dokumentace ve
                                         formátu pdf
```

Příloha B

Návod ke spuštění

Tento návod je k dispozici také v souboru *"README.txt"*.

1. Ke spuštění serveru je potřeba mít nainstalované virtuální prostředí Pharo 5.0 (dostupné z `http://files.pharo.org/platform/`).
2. Spuštění image se serverem v prostředí Pharo.
3. Spuštění překladu klienta příkazem v terminálu `make` .
4. Spuštění klienta v terminálu příkazem `./PNshell` – jsou nastaveny implicitní hodnoty portu a hosta, tak aby nebyl problém se připojit.
5. Obsluha programu pomocí textových příkazů v příloze C ve tvaru `> "zadany_prikaz""parametry"`
6. Ukončení programu zadáním příkazu `> end`

Příloha C

PNtalk 2.0 - protokol

Pozn.: Převzato ze sdíleného repozitáře `https://perchta.fit.vutbr.cz/svn/pntalk/PNShell/protocol/PNtalkServer-protokol.txt`. U každého příkazu je popsána i jeho význam.

```
= test existence modelu
(#existModel, 'nazev') => (true|false)

= pridani tridy
(#addClass, 'retezec s definici tridy') => ()

= smazani tridy
(#delClass, 'nazev') => ()

= ziskani zdrojoceho kodu tridy
(#getClass, 'nazev') => (source_code)

= inicializace simulace
(#newSim, 'nazev pocatecni tridy') => (id_simulace)

= ziskani stavu simulace
(#getState, id_simulace) => (state)

= krok simulace
(#stepSim, id_simulace) => ()

= simulace s nastavenym poctem kroku
(#simulateForSteps, id_simulace, n) => ()

= registrace udalosti
(#registerEvent, id_simulace, nazev_udalosti) => ()

= simulace do registrovane udalosti
(#simulateToEvent, id_simulace) => ( seznam_udalosti )
```