UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Tine Kavčič

Implementacija sporočilnega nivoja Sensor Network Protocol v Javi

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Borut Robič

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01757/2011
Datum: 04.04.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **TINE KAVČIČ**

Naslov: **IMPLEMENTACIJA SPOROČILNEGA NIVOJA SENSOR NETWORK PROTOCOL V JAVI**

SENSOR NETWORK PROTOCOL MESSAGE LAYER IMPLEMENTATION IN JAVA

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Predstavite glavne lastnosti sporočilnega nivoja Sensor Network Protocol in možna področja njegove uporabe. Implementirajte sporočilni nivo protokola v programskem jeziku Java in ga uporabite v aplikaciji za zajem in izvoz podatkov iz SNP naprav. Učinkovitost implementacije preverite na realnem sistemu.

Mentor:

prof. dr. Borut Robič

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko
UNIVERZA V LJUBLJANI
2

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Tine Kavčič,

z vpisno številko 63030110,

sem avtor diplomskega dela z naslovom:

Implementacija sporočilnega nivoja Sensor Network Protokola v Javi

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom (naziv, ime in priimek) prof. dr. Boruta Robiča
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne _____Podpis avtorja: _____

Zahvala

Za pomoč in nasvete pri izdelavi diplomske naloge se najprej zahvaljujem mentorju prof. dr. Borutu Robiču.

Za sodelovanje in pomoč se zahvaljujem Urošu Platiše in podjetju Energijski konduktorji d.o.o.

Rad bi se zahvalil tudi moji družini, ki mi je tekom študija vedno stala ob strani.

Kazalo

Povzetek **1**

Abstract **2**

1 Uvod **3**

2 Sensor Network Protokol **5**
2.1 Splošno 5
2.2 Sporočilni nivo 6
2.2.1 Format sporočila 6
2.2.2 Opis sporočila 7
2.2.3 Terminator 11
2.2.4 Identifikacija(ID) 11
2.2.5 Argumenti 13

3 Zasnova sporočilnega nivoja in teoretični principi **14**
3.1 Cilji 14
3.2 Analiza sporočil 15
3.2.1 Leksikalna analiza 15
3.2.2 Sintaksna analiza 17
3.2.3 Semantična analiza 20
3.2.4 Priprava izpisa 22
3.3 Ogrodje sporočilnega nivoja 22
3.3.1 SNP Seja 23
3.3.2 Okvir MSG nivoja 24

4 Implementacija **25**
4.1 Uporabljena programska orodja 25
4.1.1 Eclipse 25
4.1.2 JavaCC 25

4.1.3 JJTree . . . 25
4.2 Generiranje analizatorjev in sintaksnih dreves . . . 26
4.2.1 Leksikalne specifikacije . . . 26
4.2.2 Gramatične specifikacije . . . 27
4.2.3 Izgradnja dreves . . . 32
4.3 Interpretacija sporočil . . . 33
4.3.1 Povezovanje argumentov . . . 34
4.3.2 Razreševanje aritmetičnih izrazov . . . 35
4.3.3 Interpretacija ostalih konstruktov . . . 36
4.4 Ogrodje . . . 37
4.4.1 Seja . . . 38
4.4.2 Okvir . . . 39

5 Testiranje **41**
5.1 Testna aplikacija . . . 41
5.1.1 Komunikacija . . . 41
5.1.2 Nižjenivojska obdelava . . . 42
5.1.3 Urejanje in izvoz podatkov . . . 42
5.2 Testni sistem . . . 43
5.3 Test aplikacije na sistemu . . . 44
5.4 Test zmogljivosti . . . 46
5.5 Možne nadgradnje . . . 47

6 Sklepne ugotovitve **49**

Literatura **51**

Slike **52**

Tabele **53**

Seznam uporabljenih kratic in simbolov

- SNP - Sensor Network Protocol
- MSG nivo - angl. Message; sporočilni nivo
- HTML - angl. HyperText Markup Language; označevalni jezik
- LL analiza - angl. Left to Right, Leftmost Derivation; tip sintaksne analize
- LR analiza - angl. Left to Right, Rightmost Derivation, tip sintaksne analize
- ASCII - angl. American Standard Code for Information Interchange; način kodiranja znakov
- RS-232 - angl. Recommended Standard 232; standard za serijsko komunikacijo

Povzetek

Sensor Network Protokol definira zmogljiv in nezahteven protokol za komunikacijo med različnimi napravami. Protokol je nastal kot alternativa bolj zahtevnim standardom, ki so neprimerni za uporabo v napravah majhne kompleksnosti. Cilj diplomske naloge je implementacija sporočilnega nivoja SNP v programskem jeziku Java. Želimo realizirati knjižnico za splošno uporabo v aplikacijah na različnih platformah.

V okviru diplomskega dela so predstavljene poglavitne značilnosti Sensor network Protokola, s poudarkom na sporočilnem nivoju, ki je najbolj bistven za našo realizacijo. Predstavili smo teoretične principe, na katerih temelji realizacija, ter defnirali potrebne komponente in strukture. Pozornost moramo nameniti tudi modularnosti in nadgradljivosti, saj želimo omogočiti čim bolj preprosto dodajanje novih funkcij. V nadaljevanju predstavimo uporabljena orodja in podrobnosti implementacije.

Rešitev uporabimo v aplikaciji za zajem in izvoz podatkov iz naprav, ki implementirajo SNP protokol. Aplikacijo preizkusimo na realnem sistemu. Na koncu predstavimo tudi možne nadgradnje in izboljšave.

Ključne besede:

Sensor Network Protokol, analiza sporočil, sintaksno drevo, Java

Abstract

Sensor Network Protocol defines a capable and lightweight protocol for communication between different systems. It was developed as an alternative to more complex and demanding standards, which can be unsuitable for use in low-complexity devices. Implementation of SNP message layer is the primary goal of this thesis. We want to create a library for use across various different platforms.

In the first part of thesis we describe the most important characteristics of Sensor Network Protocol, with an emphasis on message layer, which is the most crucial part regarding our implementation. We explore various theoretical principles upon which our implementation is based and define the necessary components and structures. We also put an ephasis on the modularity of design in order to ease adding new functionality to the design. Next, we present various necessary tools and details of the implementation.

Once finished, we use our implementation in a sample application for gathering and exporting data from SNP systems. We perform testing on a working system. Finally, we also present possible upgrades and improvements.

Keywords:

Sensor Network Protocol, message analysis, syntax tree, Java

Poglavje 1

Uvod

Danes se pogosto srečujemo s problemi povezovanja in komunikacije med napravami različnih namembnosti in kompleksnosti. Za namen izmenjave podatkov med napravami obstaja več različnih protokolov, ki pa zaradi relativne zahtevnosti povečini niso primerni za uporabo v napravah majhne kompleksnosti. Zaradi tega se pojavlja potreba po protokolu, ki bi čim bolj minimiziral porabo resursov pri napravah nizke zmogljivosti. V ta namen je bil razvit Sensor Network Protocol, ki predstavlja preprosto in zmogljivo alternativo obstoječim standardom.

V diplomski nalogi se bomo posvečali predvsem sporočilnemu nivoju SNP, ki skrbi za generiranje in kodiranje sporočil ter interpretacijo dobljenih podatkov. Naš glavni cilj je implementacija osnovnih specifikacij sporočilnega dela SNP v obliki knjižnice za splošno uporabo v Javi. Želeno je, da bi bilo knjižnico možno uporabiti na čim več različnih platformah (osebni računalniki, mobilne naprave, vgrajeni sistemi). Najpomembnejši del predstavlja razpoznavanje in analiza opisov sporočil, potrebovali pa bomo tudi mehanizme za razvrščanje sporočil in povezavo z ostalimi nivoji. Našo implementacijo bomo preizkusili na realni napravi s pomočjo preproste aplikacije za zajem in prikaz podatkov.

V naslednjem poglavju so predstavljene poglavitne značilnosti MSG nivoja Sensor Network Protokola, s poudarkom na formatu sporočil.

Tretje poglavje opisuje splošno zasnovo implementacije, teoretične principe in bistvene sestavne dele.

Četrto poglavje govori o konkretni realizaciji programskega dela. Opisana so uporabljena orodja in postopki, skupaj s primeri kode.

V petem poglavju si ogledamo še primer uporabe v aplikaciji in test na realni napravi.

Poglavje 2

Sensor Network Protokol

2.1 Splošno

Sensor Network Protocol oziroma krajše SNP definira protokol za povezovanje in komunikacijo med napravami v porazdeljenih sistemih. V prvi vrsti je namenjen za komunikacijo z napravami majhne kompleksnosti, za mobilne naprave in naprave z majhno porabo energije. V tovrstnih napravah lahko ob implementaciji bolj razširjenih samoopisljivih standardov (npr. XML) naletimo na omejitve in probleme s performansami, zato se pojavlja potreba po bolj kompaktni alternativi.

Avtor protokola je Uroš Platiše iz podjetja Isotel d. o. o. Protokol je že uporabljen v nekaterih konkretnih produktih, ki nam lahko služijo kot platforma za testiranje naše implementacije.

Osnovne značilnosti:

- Neodvisnost od gonilnikov naprave
- Enostavna in poceni implementacija
- Kompaktna struktura
- Berljiv format
- Zagotavljanje zanesljivosti prenosa
- Podpora za napredne matematične operacije
- Direktna SCPI podpora

2.2 Sporočilni nivo

Sporočilni nivo SNP je zasnovan kot enostaven samoopisljiv protokol za uporabo na enostavnih kot tudi na kompleksnejših napravah, z minimalnimi pomnilniškimi zahtevami. Prenos podatkov se vrši preko sporočil, katerih format ne potrebuje pretvorbe števil v nize, ter lahko prenaša tudi strukturirane izraze, enačbe, vektorje in matrike. Takšen format tudi ne potrebuje obsežnega razpoznavanja (v nasprotju z npr. XML) s strani vseh naprav, saj prenese obdelavo kompleksnejše aritmetike in logike k bolj zmogljivim napravam, ki so namenjene interpretiranju danih podatkov. Dodeljevanje vrednosti spremenljivkam je glede parsinga zelo nezahtevno in ga lahko implementirajo tudi najenostavnejše naprave.

Vsako sporočilo je avtonomna celota, vendar lahko več sporočil skupaj tvori večjo strukturo in s tem tudi kompleksnejšo vsebino. Vsaka naprava s SNP sporočilnim nivojem mora implementirati vsaj dve sporočili:

- **INIT MSG** - Sporočilo z ID številom 0, ki definira napravo z enolično določenim nizom v človeku berljivem formatu. Navadno se to sporočilo prenese le enkrat.
- **STATUS MSG** - Sporočilo z ID številom 1, ki posreduje trenutni status naprave.

2.2.1 Format sporočila

Format sporočila je določen s štirimi polji, kot prikazuje tabela 2.2.1

OPIS	TERMINATOR	ID	ARGUMENTI
t bajtov	1 bajt	2 bajta	a bajtov

Tabela 2.1: Format sporočila

OPIS: Tekstovni niz, ki določa splošno vsebino sporočila v berljivi obliki. Navadno je opis potrebno prenesti samo enkrat, zato ga lahko ob nadaljnjih prenosih sporočila izpustimo.

TERMINATOR: Znak za končanje niza. Določa konec opisa.

IDENTIFIKACIJA(ID): 16 bitna vrednost, ki vsebuje kontrolne bite in

10 bitov za ID število sporočila. Kadar se ne pošilja opisa sporočila, lahko prejemnik preko tega polja ugotovi, za katero sporočilo gre in poišče ustrezen opis v pomnilniku.

ARGUMENTI: Argumenti v svoji naravni (binarni) obliki.

Sporočila so navadno poslana v enem kosu, če je sporočilo predolgo obstajata dve opciji za pošiljanje:

- Pošiljatelj razdeli sporočilo na dva dela, v prvem pošlje opis, terminator in ID, v drugem pa terminator, ID in argumente.
- Sporočilo se razdeli in ponovno sestavi na nižjem nivoju

2.2.2 Opis sporočila

Opis sporočila v osnovi sledi pravilom standardne printf() funkcije iz programskega jezika C. V nadaljevanju so opisane najbolj bistvene dodatne specifikacije. Celotne specifikacije so podane v [3].

2.2.2.1 Izrazi

Izraz omogoča višjenivojske operacije nad števili in nizi. Splošna oblika izraza je:

"...(D0xD1x...xDn){izraz0,izraz1,izraz2,...,{gnezden izraz}}[enota(e)]..."

Izraz je definiran v zavitih oklepajih *{...}* in lahko vsebuje poljubno število matematičnih izjav ali gnezdenih izrazov, ločenih z vejico ali podpičjem. Pred izrazom so lahko podane dimenzije, za njim pa je lahko podana še enota. Oklepaji pri izpisu niso vidni. Vsak odprti oklepaj pomeni povečanje globine, kot je opisano v razdelku 2.2.2.3

Podprti so vsi standardni matematični simboli in funkcije ki so vsebovani v glibc[4] knjižnici (*/-+%^|&...,log,exp,sin,cos,...). Za obdelavo kompleksnejših izrazov je možna uporaba zunanjih orodij kot so matlab, octave ali druga orodja za obdelavo vektorskih in matričnih operacij.

Če ima izraz predpono, le-ta definira dimenzije izraza. Predpona je lahko konstanta ali argument. Iz predpone je razvidno, kolikokrat se izraz ponovi z

novimi argumenti. Primer: *"3x3{%i}"* definira matriko devetih celih števil v polju za argumente. Medtem ko npr. *%hu{0.024*%i}[V]* podaja vektor celih števil spremenljive dolžine od *0* do *255*, ki se nato po podanem izrazu transformirajo v vrednosti v voltih.

Enota je podana takoj po izrazu (brez presledkov) v oglatih oklepajih *"[...]"*. Vektorskim in matričnim izrazom se lahko dodeli množica enot. Podajanje enot ni obvezno.

2.2.2.2 Enumeracije

Izraz lahko vsebuje poljuben niz takoj po odprtem oklepaju in s predpono ':'. Enumeracija je podana v obliki:

"...{%?:elem0,elem1,eleml2,...}..."

Argument *?* definira, kateri element se izpiše, lahko pa je katerega koli tipa, ki predstavlja pozitivno celo število (hu, u, ...). Vejice ali podpičja služijo kot ločila med posameznimi elementi. Če to ni zaželeno, lahko nize vstavimo s pomočjo gnezdenih izrazov. Tako pridemo do razširjenega formata enumeracije:

"...{%?:elem0{:elem00, elem01,elem02},{:elem1},{:elem2},...}..."

Tukaj *elem0* določa ime skupine s tremi podopcijami, *elem1* in *elem2* pa sta gnezdena zaradi podpore vejicam in podpičjem v nizih.

Vsak element sovpada z vrednostjo argumenta od 0 dalje. Možna je tudi eksplicitna določitev intervalov ali vrednosti, na primer:

"...{%u:MIN=1,2..49,max}..."

MIN in *MAX* imata v tem primeru določene vrednosti 1 in 50, ostale vrednosti pa se izpustijo.

2.2.2.3 Podatkovne strukture

Podatkovne strukture lahko opišemo s pomočjo splošne oblike:

"... ime0{ ime1{ ... } ... } ..."

Vsak odprti oklepaj predstavlja nov podrazred sporočil, spremenljivk in izrazov, ki pripadajo razredu z imenom, podanim direktno pred odprtim oklepajem. Ime se ne sme začeti s številom, sicer lahko pride do dvoumnosti glede imena in podajanja dimenzij. Imena razredov so v izpisu vidna, medtem ko oklepaji niso.

Dimenzije podatkovnih struktur so podane z vejicami in podpičji. Vejica ',' predstavlja stolpec, medtem ko podpičje ';' predstavlja vrstico. Primer deklaracije:

"... A{b;c;d},B{b;c;d} ..."

Spremenljivke so lahko deklarirane tudi kot interval:

"A(1..3)"

ali po posameznih elementih:

"A(1)" "A(2)"

2.2.2.4 Spremenljivke in vrednosti

Spremenljivke lahko povežemo z vrednostmi v sledeči obliki:

"...{:spremenljivka}=(D0x...xDn){izraz}[enota]..."

Spremenljivko določa izraz oblike *{:niz}*, ki mu sledi enačaj '=', v sledečem izrazu pa je določena njena vrednost. Spremenljivka je lahko definirana tudi kot enumeracija *"{%?:var1,var2,var3}="* ali v obliki argumenta tipa *"{:%s}="*.

Globalne spremenljivke so definirane s predpono *"*"* pred nizom za ime spremenljivke.

2.2.2.5 Enote

Enote določajo relativen pomen izraza. Predstavljajo lahko neko fizikalno količino, lahko pa izražajo tudi kvalitativno stanje. Naprava lahko recimo pošlje: *"{:obvestilo}[error]"*, kar nakazuje, da je na nižjem nivoju prišlo do napake, medtem ko *{:obvestilo}[info]* samo sporoča neko spremembo stanja. Enota tako lahko predstavlja različne prioritete/stopnje. Enota je lahko predstavljena tudi kot vektor, npr. za tri argumente, ki predstavljajo napetost, tok in čas:

"...{%f,%f,%Lt}[V,A,s]..."

2.2.2.6 Posebna polja in urejanje izpisa

Posebna polja imajo obliko:

"...%<določilo>{polje}..."

Ta polja vključujejo tipične možnosti za urejanje besedil, kot so skrita polja, oblika črk, poravnave, itd. Določilo lahko vsebuje tudi argument, na primer:

"...%t%c{ text formatting }..."

Tu argument tipa *%c* podaja 8 bitni znak, recimo *b, i, r, f, h* za nastavitev teksta na *bold, italic, roman, fogged* ali *hidden* stil.

Segmentacija teksta se lahko opravlja s standardnimi sekvencami *"\n"* in *"\r"*, vendar je bolj priporočljiva uporaba "mehke" segmentacije s pomočjo podpičja *";"*. V tem primeru se nova vrstica uporabi le takrat, ko ni možno celotnega segmenta prikazati v trenutni vrstici. Podpičje v izpisu ni prikazano.

2.2.2.7 Argumenti

Opis sporočila določa seznam argumentov, ki so v svoji naravni obliki pripeti na koncu sporočila. Argumenti se v opisu lahko nahajajo praktično povsod, med nizi, v izrazih, enumeracijah itd.

Glede na dostop lahko ločimo tri skupine argumentov:

- bralni (read only)

- pisalni (write only)
- bralno-pisalni (read-write)

Bralno-pisalni argumenti so privzeta oblika s predpono "%". Pisalni argumenti so označeni z ">%" predpono, nastavitev potrdijo tako da vrnejo neko vrednost. Bralni argumenti imajo predpono "<%" in ne omogočajo zunanjega spreminjanja vrednosti.

Nastavljanje argumentov dosežemo tako, da pošljemo napravi sporočilo s terminatorjem, ustreznim ID-jem in argumenti v pravilnem formatu. Če je katera od vrednosti globalna, moramo poslati tudi opis sporočila.

%b bit	**%hx** unsign 8 bit hex	**%lu** unsign 32 bit	**%hf** float 16 bit
%c znak	**%i** int 16 bit	**%lx** unsign 32 bit hex	**%f** float 32 bit
%s niz	**%u** unsign 16 bit	**%Li** int 64 bit	**%lf** float 64 bit
%hi int 8 bit	**%x** unsign 16 bit hex	**%Lu** unsign 64 bit	**%Lf** float 80 bit
%hu int 8 bit	**%li** int 32 bit	**%Lx** unsign 64 bit hex	**%w** odmik

Tabela 2.2: Tipi argumentov

V tabeli 2.2 so argumenti specifirani v "little endian" predstavitvi. Pri "big endian" predstavitvi se pri opisu uporabljajo velike črke (I,U,X,F).

2.2.3 Terminator

Predstavlja standardni C-terminator (0x00) za končanje ASCII niza. Na tej točki se konča opis sporočila. Terminator se navadno pošlje tudi, če opis spustimo.

2.2.4 Identifikacija(ID)

16-bitno ID polje je sestavljeno kot prikazuje tabela 2.3.

OPIS	**Način Poizvedbe**	**Tip Opisa**	**Globina/Funkcija**	**Enoličen ID**
BITI	15:14	13	12:10	9:0

Tabela 2.3: Format ID-ja

2.2.4.1 Način poizvedbe(Query Mode)

Dva bita za način poizvedbe določata splošni status sporočila:

- **0b11** Nastavljanje vsebine sporočila v obe smeri
- **0b10** Poizvedba po sporočilu glede na funkcijske bite (celo ali samo deli sporočila)
- **0b01** Vnos ali ukaz s strani naprave
- **0b00** Posebna funkcija

2.2.4.2 Tip opisa (Variant)

Bit za tip opisa je nastavljen samo kadar opis sporočila ni konstanten. S tem sporoča prejemniku, da lahko opis shrani lokalno in ga v nadaljevanju izpusti.

2.2.4.3 Globina/Funkcija

Trije biti za globino in funkcijo imajo več funkcionalnosti glede na to, kateri način poizvedbe je nastavljen.

Pri nastavljanju vsebine v obe smeri:

- **0bXX1** Sporočilo je poslano iz originalnega izvora
- **0bX1X** Sporočilo je poslano preko usmerjevalnika
- **0b1XX** Sporočilo je poslano iz zunanjega izvora

Pri poizvedbi po sporočilu:

- **0bDxx** Poizvedba po opisu
- **0bxEx** Poizvedba po razširjenem opisu
- **0bxxA** Poizvedba po argumentih

Pri vnosu/ukazu:

- V tem primeru se biti za globino/funkcijo ne uporabljajo. Tu naprava pričakuje le argumente, kar je uporabno pri inicializaciji naprave, kalibriranju senzorjev itd.

Posebne funkcije:

- **0b000** Zadnje sporočilo
- **0b001** Zasedeno/Čakanje
- **0b010** Napaka
- **ostalo** Rezervirano

2.2.4.4 Enoličen ID

Preko desetih bitov se vsakemu sporočilu določi enolično ID število za identifikacijo pri vhodu/izhodu iz naprave.

2.2.5 Argumenti

Argumenti so pripeti na koncu sporočila, v enakem vrstnem redu, kot se pojavijo v opisu. Njihova dolžina in tip sta razvidna iz opisa sporočila.

Poglavje 3

Zasnova sporočilnega nivoja in teoretični principi

3.1 Cilji

Glavni cilj dela je napraviti učinkovito in čim bolj optimalno knjižnico za dostop, analizo in obdelavo sporočil po specifikacijah SNP. Knjižnico bomo napisali v programskem jeziku Java, želimo, da bi jo bilo mogoče uporabiti na čim več različnih platformah. Trenutno se uporablja več različic za mobilne telefone in za osebne računalnike, zato si želimo bolj splošne in prenosljive implementacije. Smotrno je uporabiti tudi modularen, objektno orientiran pristop, za kar je Java zelo primerna.

Ker želimo realizirati čim bolj kompaktno implementacijo, ki je primerna tudi za uporabo v vgrajenih sistemih, v obsegu diplomske naloge ne bomo realizirali vseh funkcionalnosti, ki jih v teoriji podpira sporočilni nivo SNP. Poleg osnovnih gradnikov moramo vgraditi podporo za izraze, enumeracije, spremenljivke, enote in formatiranje tekstovnih nizov. Zaenkrat ne bomo vdelali podpore za podatkovne strukture in ostale kompleksnejše funkcionalnosti. Kljub tem omejitvam bi morala naša implementacija zadostovati za večino primerov uporabe, posebej na napravah manjše kompleksnosti. Izdelek moramo zasnovati tako, da omogoča čim lažje dodajanje novih funkcionalnosti. Tako se lahko knjižnico prilagodi glede na dane potrebe in strojne omejitve.

3.2 Analiza sporočil

Analiza sporočil predstavlja najbolj bistveno komponento pri implementaciji sporočilnega nivoja. Če hočemo dostopati do podatkov v sporočilu na smiseln način, mora komponenta za analizo najprej iz opisa razbrati strukturo in pomen sporočila ter ga povezati z ustreznimi argumenti.

Pri načrtovanja analize sporočil si lahko pomagamo s teorijo načrtovanja prevajalnikov. Čeprav sami ne izdelujemo prevajalnika, lahko pri analizi sporočil potegnemo številne vzporednice z analizo vhodnega programa pri prevajalnikih. Analiza je navadno razdeljena na tri glavne faze [1]:

- **Leksikalna analiza:** Delitev vhodnega niza na smiselne besede oz. lekseme.
- **Sintaksna analiza:** Analiza strukture vhodnih besed glede na podana pravila
- **Semantična analiza:** Ugotavljanje pomena vhodnega niza

V nadaljevanju so podrobneje predstavljene vse tri faze, skupaj z okvirnimi specifikacijami za našo implementacijo.

3.2.1 Leksikalna analiza

Naloga leksikalnega analizatorja je delitev niza znakov v smiselne celote (žetone) in izločitev nerelevantnih znakov. Zaporedje žetonov nato preda sintaksnemu analizatorju v nadaljnjo obdelavo.

Beseda ali žeton je zaporedje znakov, ki jih lahko obravnavamo kot enoto v gramatiki našega jezika. Leksikalni žetoni so razvrščeni v različne skupine glede na njihov pomen in obliko. Nekaj primerov žetonov, ki jih bomo potrebovali za analizo SNP sporočil:

- **Argumenti:** *%hi, %c, %li, %u ...*
- **Operatorji:** *+, - , *, /, >>, | ...*
- **Separatorji:** *{}, [], \n ...*
- **Posebna polja:** *%tb, %t1 ...*
- **Številske konstante:** *12, 0.0032, 18e-5 ...*

- **Tekstovni nizi:** *prvi, drugi, spr1 ...*

Leksikalni analizator je mogoče dokaj enostavno implementirati praktično v kateremkoli programskem jeziku z uporabo zank in pogojnih stavkov, vendar je pametno uporabiti bolj formalen pristop. Za definiranje žetonov se navadno uporablja regularne izraze, za implementacijo pa se uporabi končne avtomate.

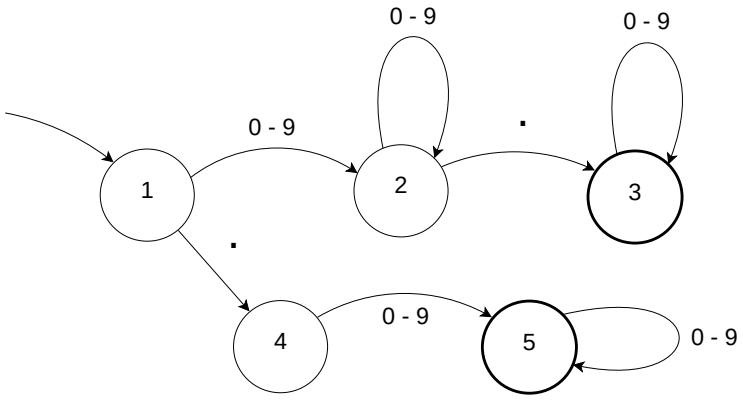
Regularni izraz je formalizem, ki opiše neko množico nizov. Osnovni elementi zapisa regularnih izrazov so naslednji:

- **Simbol:** Za vsak posamezen simbol a regularni izraz a določa jezik, ki vsebuje le niz *"a"*.
- **Alternacija:** Če imamo podana regularna izraza M in N, alternacijski operator $|$ podaja nov regularni izraz $M \mid N$. Niz pripada jeziku $M \mid N$, če je bodisi v jeziku M bodisi v jeziku N. Jezik $a \mid b$ torej vsebuje niza *"a"* in *"b"*.
- **Združitev:** Če imamo podana regularna izraza M in N, združevalni operator $\cdot$ podaja nov regularni izraz $M \cdot N$. Niz pripada jeziku $M \cdot N$, če predstavlja združitev a in b, kjer a pripada M in b pripada N. Regularni izraz $(a \mid b) \cdot a$ tako definira jezik, ki vsebuje niza *"aa"* in *"ba"*.
- **Epsilon:** Regularni izraz ε predstavlja jezik, katerega edini niz je prazen niz. Izraz $(a \cdot b) \mid \varepsilon$ torej določa jezik, ki vsebuje niza *""* in *"ab"*.
- **Ponovitev:** Izraz za ponovitev M^* predstavlja združitev nič ali več nizov, od katerih vsi pripadajo M. Izraz $(a \mid b)^*$ tako določa neskončno množico nizov tipa *"aa"*, *"bababa"*, *"bbbbaaa"*, itd.

S pomočjo teh izrazov lahko izpeljemo množico ASCII znakov, ki predstavljajo leksikalne žetone v našem jeziku.

Regularni izrazi so priročen način za definiranje leksikalnih žetonov, vendar potrebujemo formalizem, ki ga lahko implementiramo v obliki računalniškega programa. V ta namen lahko uporabimo končni avtomat. Takšen avtomat vsebuje neko končno število *stanj*, med sabo povezanih s *prehodi*. Stanja lahko razdelomo na začetna (samo eno), vmesna in končna. Vsak prehod je označen s *simbolom*. Pri determinističnem avtomatu morajo vsi prehodi, ki vodijo iz določenega stanja, imeti različne simbole. Za vsak znak v vhodnem nizu avtomat sledi točno določenemu prehodu v naslednje stanje. Po

n-prehodih za vhodni niz dolžine n mora biti avtomat v končnem stanju, kar pomeni da je niz sprejet. Če avtomat ni v končnem stanju, se niz zavrne. iz tega sledi, da avtomat prepoznava nek *jezik*, ki predstavlja množico nizov.



Slika 3.1: Primer končnega avtomata za predstavitev realnih števil

Izdelava končnih avtomatov je relativno preprost postopek, ki ga lahko avtomatsko in učinkovito opravi računalnik. Zato je za pretvorbo regularnih izrazov v končni avtomat smotrno uporabiti enega od generatorjev leksikalnih analizatorjev.

3.2.2 Sintaksna analiza

Sintaksna analiza ali parsing je proces analiziranja zaporedja žetonov z namenom ugotavljanja njihove gramatične strukture. To navadno poteka z uporabo formalnih gramatik, ki rekurzivno definirajo možne komponente izrazov in njihov vrstni red. Sintaksni analizator oz. parser nam producira izhod v obliki sintaksnega drevesa, ki predstavlja gramatično strukturo vhodnega niza. Dobljeno drevesno strukturo se nato uporabi v postopku semantične analize.

Tudi izdelavo sintaksnih analizatorjev je možno avtomatizirati, zato je podobno kot pri leksikalni analizi možna uporaba namenskih generatorjev.

3.2.2.1 Gramatika

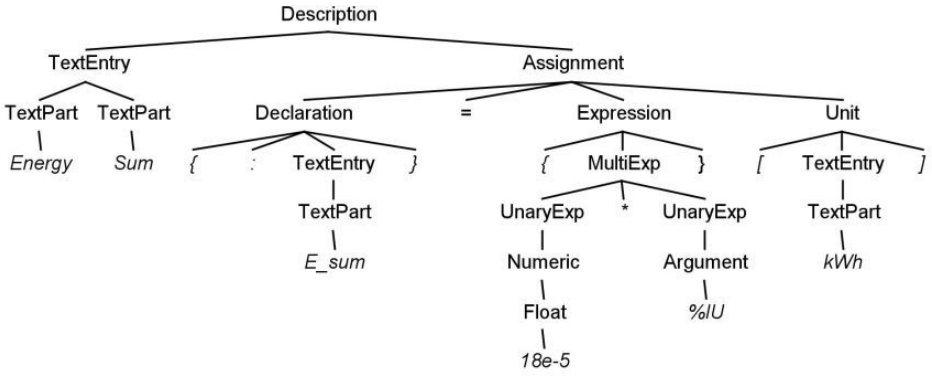
Kontekstno neodvisna gramatika je zapis pravil, ki opisujejo nek jezik. V našem primeru je to jezik opisov SNP sporočil. Jezik je množica nizov, vsak

niz pa je končna množica simbolov vzetih iz končne abecede. V našem primeru je niz opis sporočila, simboli so leksikalni žetoni, abeceda pa je množica različnih tipov žetonov, ki jih vrne leksikalni analizator.

Gramatika vsebuje množico produkcij tipa $X \rightarrow y$, kjer je X pravilo, y pa zaporedje pravil ali končnih simbolov. Končni simbol predstavlja leksikalni žeton. Pravilo je izpeljan simbol, kar pomeni, da se pojavi na levi strani neke produkcije.

Če hočemo pokazati, da niz pripada jeziku gramatike, lahko izvedemo izpeljavo: pričnemo z začetnim pravilom, nato pa postopoma nadomestimo vsako pravilo z njegovo desno stranjo, dokler nam ne ostanejo samo končni simboli.

Primer izpeljave za opis sporočila *"Energy Sum {:E_sum}={18e-5*%lU}[kWh]"* si lahko ogledamo v 3.2.



Slika 3.2: Prikaz možne izpeljave opisa SNP sporočila *"Energy Sum {:E_sum}={18e-5*%lU}[kWh]"*. V tem primeru nastopajo končni simboli *{, }, [,], :, *, =, Energy, Sum, E_sum, 18e-5* in *%lU*. Ostale oznake predstavljajo različna gramatična pravila.

Za vsak niz obstaja več različnih izpeljav. Pri *levi izpeljavi* (leftmost derivation) se vedno naslednji razširi prvi nekončni simbol z leve strani. Pri *desni izpeljavi*(rightmost derivation) se razširi prvi nekončni simbol z desne.

Nekatere gramatike je možno analizirati z uporabo *rekurzivnega spusta*. Vsaka produkcija se spremeni v en razdelek v rekurzivni funkciji. Takšen algoritem deluje samo pri gramatikah, kjer prvi končni simbol v vsakem podizrazu vsebuje dovolj informacije za izbor primerne produkcije. Za

analizo kompleksnejših gramatik obstajata dve poglavitni kategoriji analize:

- **LL analiza:** Vhodni niz bere od leve proti desni in izgradi *levo izpeljavo*. Oznaka LL(k) pomeni, da pri analizi pregleduje k končnih simbolov vnaprej. V ta namen je potrebno definirati sintaksno tabelo, s pomočjo katere se izbere ustrezno pravilo(če takšno pravilo obstaja) glede na trenutne vhodne parametre. Gramatike, ki jih lahko analiziramo na takšen način se imenujejo *LL gramatike*. Potencisalen problem predstavlja leva rekurzija, pri kateri LL analizatorji v veliko primerih odpovejo.

- **LR analiza:** Slabost LL(k) analize je ugotavljanje ustrezne produkcije samo iz prvih k žetonov na desni. Naprednejši postopek se imenuje LR(k) analiza, pri kateri se lahko odločitev preloži, dokler niso obdelani vsi vhodni žetoni, ki predstavljajo dano produkcijo. LR analizator bere vhodni niz od leve proti desni in izgradi *desno izpeljavo*. Analizator potrebuje sintaksno tabelo in sklad, na katerega hrani svoja prejšnja stanja. Glede na vsebino sklada in k žetonov na desni se lahko trenutni žeton potisne na vrh sklada ali pa se uporabi neko gramatično pravilo, ki se nato prav tako potisne na sklad. Postopek se ponavlja dokler ni niz sprejet ali zavrnjen.

3.2.2.2 Sintaksna drevesa

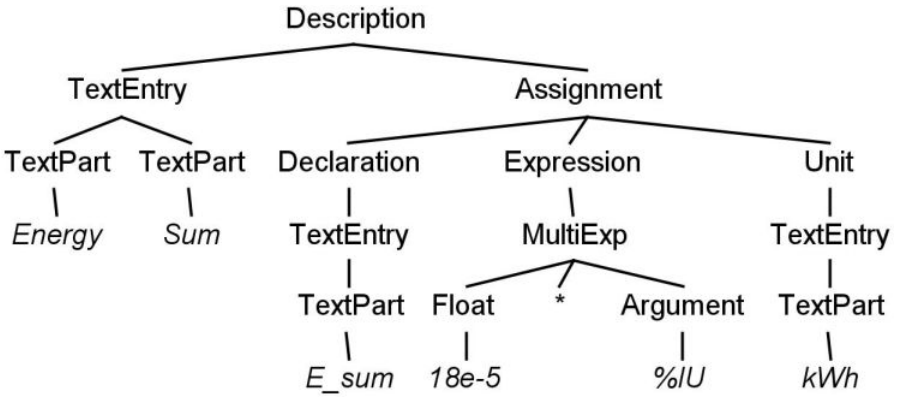
Načeloma bi lahko analizirali pomen vhodnega niza med opravljanjem sintaksne analize, vendar bi bil takšen analizator zelo nepregleden in težaven za vzdrževanje in nadgrajevanje. Za izboljšanje modularnosti je zato bolje ločiti obdelavo na sintaksno in semantično fazo. Najpogostejši način za dosego tega je, da med sintaksno analizo zgradimosintaksno drevo, ki ga uporabljajo naslednje faze analize. Takšno drevo ima en list za vsak končni simbol in eno notranje vozlišče za vsako gramatično pravilo, ki je uporabljeno med izpeljavo.

Toda takšno sintaksno drevo je lahko neprimerno za direktno uporabo. Veliko žetonov je redundantnih in ne nosijo uporabnih informacij. Uporabni so v vhodnem nizu, toda ko je enkrat zgrajeno sintaksno drevo lahko tovrstne informacije izvemo iz strukture drevesa. Poleg tega lahko struktura drevesa preveč zavisi od same gramatike. Gramatike pogosto uvajajo številna dodatna pravila zaradi tehničnih razlogov, vendar nam ta pravila izven sintaksne faze ne koristijo, zato se jih je pametno znebiti pred semantično

fazo.

Potrebujemo torej boljši vmesnik med sintaksno fazo in poznejšimi fazami, zato uvedemo pojem abstraktno sintaksnega drevesa. Abstraktno drevo nam podaja strukturo vhodnega niza z razrešenimi sintaksnimi problemi, vendar brez semantične interpretacije.

Primer abstraktnega drevesa za opis sporočila *"Energy Sum {:E_sum}={18e-5*%lU}[kWh]"* si lahko ogledamo v 3.3. Vidimo da smo se v primerjavi z izpeljavo v 3.2 znebili precej elementov. Vendar pa pri tem nismo izgubili nobenih relevantnih informacij za semantično fazo obdelave. Na primer shranjevanje oklepajev bi bilo nesmiselno, saj so relacije med različnimi strukturami v opisu zadostno podane s tipi vozlišč in njihovo razporeditvijo.



Slika 3.3: Prikaz možne strukture abstraktnega sintaksnega drevesa SNP sporočila *"Energy Sum {:E_sum}={18e-5*%lU}[kWh]"*. V tem primeru nastopajo končni simboli **, Energy, Sum, E_sum, 18e-5* in *%lU*. Takšno drevo je primerno za uporabo v sledečih fazah.

3.2.3 Semantična analiza

V teoriji prevajalnikov semantična analiza navadno vključuje opravila kot so povezovanje definicij spremenljivk z njihovo uporabo, preverjanje tipov spremenljivk in prevajanje abstraktne sintakse v enostavnejšo vmesno kodo. Pri analizi SNP sporočil moramo v tej fazi povezati polje argumentov z njihovo uporabo, preveriti tipe argumentov, razrešiti izraze in prevesti strukturo v obliko, primerno za izpis.

3.2.3.1 Interpretacija dreves

V opisu faze sintaksne analize smo si ogledali nekaj konceptov gradnje abstraktnih sintaksnih dreves. Poglavitna naloga semantične faze pa je interpretacija dobljenega drevesa. Za dosego tega obstaja več različnih pristopov. Najbolj preprost način je, da metode za evaluiranje vgradimo kar v sintaksne razrede, ki predstavljajo vozlišča v drevesu. Tako bi lahko npr. v vozlišče *MultiExp* vgradili metodo, ki bi med sabo zmnožila numerične vrednosti iz potomcev in vrnila dobljen rezultat. Takšnemu načinu lahko rečemo *objektno-orientiran* način. Pri tem pristopu je enostavno dodati nov *tip* vozlišča, saj moramo dodati le nov razred skupaj z metodo za evaluacijo. Vendar se pojavi problem, če hočemo dodajati nove *interpretacije*, saj moramo za vsako novo interpretacijo popraviti vse razrede. Za nas je takšen pristop neroden, saj si želimo večje fleksibilnosti pri interpretiranju .

Alternativni pristop temelji na ločitvi sintaksnih razredov in kode za interpretacijo. V tem primeru lahko potujemo po drevesu s pomočjo *instanceof* ukaza in glede na to izberemo ustrezno evaluacijo. Takšen pristop lahko imenujemo *ločena sintaksa od interpretacije*. Pri takšnem pristopu se da enostavno in modularno dodajati nove interpretacije: Moramo le napisati funkcijo z razdelki za vsak tip vozlišča. Pomanjkljivost tega pristopa se pokaže pri dodajanju novih tipov vozlišč, saj moramo dodati nov razdelek v vse interpretacije.

Za analizo SNP sporočil nam bolj odgovarja pristop ločene sintakse od interpretacije, saj je možno informacije iz sporočil predstaviti in uporabiti na mnogo različnih načinov, zato si želimo čim bolj enostavno in modularno dodajanje interpretacij. Na srečo imamo poleg nerodnega pristopa z *instanceof* na voljo bolj elegantno rešitev; implementacijo *obiskovalca* (ang. Visitor). Obiskovalec je objekt, ki vsebuje metode za "obisk" vsakega sintaksnega razreda. Poleg tega morajo sintaksni razredi vsebovati preprosto metodo za sprejem obiskovalca, ki preda kontrolo ustrezni metodi obiskovalca. Na takšen način lahko enostavno dodajamo poljubne interpretacije, brez popravljanja in ponovnega prevajanja obstoječih razredov, če le-ti vsebujejo metodo za sprejem obiskovalca.

3.2.3.2 Uporaba argumentov

Poleg opisa sporočila moramo med semantično fazo sprocesirati tudi argumente, ki nastopajo v sporočilu. V prejšnjem razdelku smo se odločili za

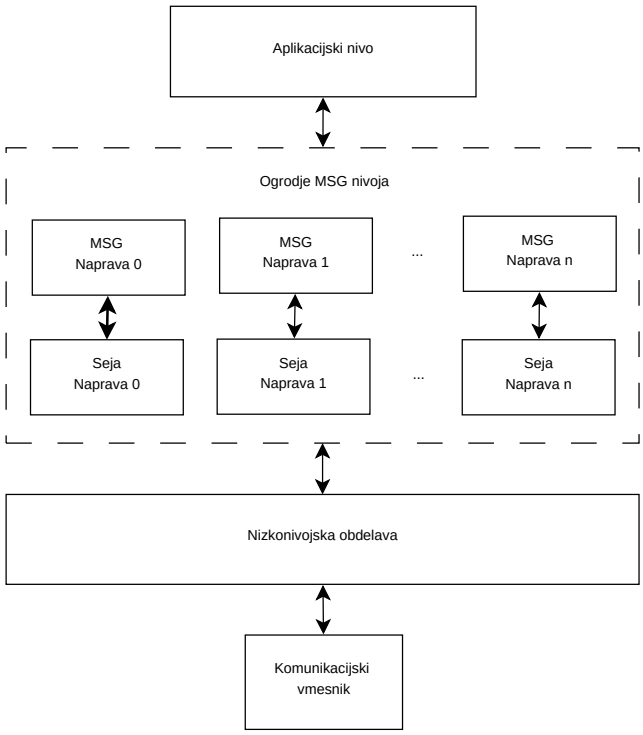
uporabo obiskovalca, torej lahko definiramo funkcije za razreševanje argumentov kar v obiskovalcu. To pomeni, da moramo obiskovalcu pred uporabo posredovati ustrezno polje argumentov. Poleg tega moramo definirati še indeks, ki med premikanjem po drevesu prdstavlja trenutno pozicijo v polju argumentov. Če se po drevesu premikamo z leve proti desni, bomo na argumente naleteli v pravilnem vrstnem redu, kot so definirani v sporočilu. Ko v drevesu naletimo na argument najprej izračunamo njegovo dolžino, nato pa iz polja argumentov preberemo ustrezno število bajtov od indeksa dalje. Prebrane podatke nato pretvorimo v ustrezen podatkovni tip, povečamo indeks in vrnemo vrednost argumenta.

3.2.4 Priprava izpisa

Med interpretiranjem drevesa lahko poskrbimo tudi za formatiranje izpisa. V obiskovalca lahko vgradimo razširitev, ki izhodnim nizom doda poljubne elemente. Tako lahko nizom dodamo npr. HTML oznake za izvoz v obliki spletne strani ali poljuben drug jezik za urejanje tekstovnega izpisa. Tak pristop je marsikdaj lahko učinkovitejši kot formatiranje celotnega izpisa šele na aplikacijskem nivoju. Seveda lahko tako spreminjamo le izpis iz vozlišč, nad katerimi se ne bodo več izvajale operacije, npr. do konca razrešen izraz.

3.3 Ogrodje sporočilnega nivoja

Poleg komponente za analizo sporočil moramo implementirati tudi ogrodje za dostop do samih sporočil in komunikacijo z ostalimi nivoji. Definirati moramo vmesnik, s katerim lahko aplikacijski nivo dostopa do sporočilnega nivoja in podaja zahteve. Poskrbeti moramo tudi za shranjevanje in ažuriranje spremenljivk in sintaksnih dreves. Poleg tega moramo definirati vmesni nivo med sporočilnim nivojem in nižjimi nivoji. Sporočilni nivo mora biti neodvisen od implementacije nižjih nivojev, ki se lahko med sabo zelo razlikujejo. Struktura nivojev je prikazana na sliki 3.4.



Slika 3.4: Umestitev in struktura sporočilnega nivoja

3.3.1 SNP Seja

Seja (ang. Session) predstavlja vmesni nivo med MSG in nižjimi nivoji. Vsaki SNP entiteti (navadno naprava) se določi svojo sejo. Entiteta je enolično določena z 32-bitnim naslovom, ki se ga shrani ob kreiranju seje. Seja služi kot medpomnilnik med nivoji in hrani sporočila v njihovi binarni obliki. Ko nižji nivoji prejemajo sporočila z entitet jih posredujejo ustrezni seji, ki sproti ažurira podatke in jih po potrebi posreduje sporočilnemu nivoju. Poleg tega lahko sprejema tudi zaheve za pošiljanje sporočil entiteti, jih obdela in posreduje nižjim nivojem.

Sejni razred mora med drugim implementirati naslednje metode:

- **void updateData(byte[] data)** - Metoda za ažuriranje podatkov, klicana iz nižjega nivoja ob prihodu sporočila z ustreznim naslovom.
- **void sendRequest(int id, byte[] args)** - To metodo kliče sporočilni nivo ob zahtevo po pošiljanju sporočila entiteti. Seja zahtevo oblikuje v

sporočilo in ga posreduje nižjemu nivoju v obdelavo in pošiljanje entiteti.

- **byte[] getArguments(int id)** - Zahteva po polju za argumente sporočila, določenega z *ID* številom v argumentu. Uporabno je, kadar že imamo zgrajena sintaksna drevesa in potrebujemo samo najnovejše parametre za prikaz.

- **Object getMessageData()** - Zahteva po celotni množici sporočil. Uporabno je, če želimo sporočila sprocesirati le enkrat ali če opisi sporočil niso konstantni.

Poleg teh metod mora seja implementirati tudi primerno podatkovno strukturo za hranjenje sporočil.

3.3.2 Okvir MSG nivoja

Okvir predstavlja glavni razred za koordinacijo sporočilnega nivoja. Zadolžen je za koordiniranje obdelave sporočil, hranjenje sintaksnih dreves in obdelavo zahtev iz višjih nivojev. Tesno je povezan s sejnim nivojem, s katerim si izmenjujeta podatke v obe smeri. Ob prejemu novih sporočil inicializira izgradnjo sintaksnih dreves in jih v primeru konstantnih opisov shrani v ustrezno podatkovno strukturo. Ob zahtevi za izpis inicializira semantično fazo obdelave in zbere podatke za posredovanje višjim nivojem.

Okvir med drugim implementira sledeče metode:

- **void updateTree(byte[] msg)** - Ta metoda izvede analizo opisa in zgradi sintaksno drevo. Kliče jo pripadajoča seja, ko prejme nov opis sporočila. V argumentu je podano celotno sporočilo.

- **void print(OutputStream out, <options>)** - Zahteva za izpis podatkov, podana s strani višjega nivoja. V nastavitvah(options) lahko uporabnik specifira katera sporočila naj izpis obsega, lahko pa tudi poda svoj lasten objekt za interpretacijo. Tako zagotovimo veliko fleksibilnost in modularnost uporabe sporočilnega nivoja.

Poglavje 4

Implementacija

4.1 Uporabljena programska orodja

4.1.1 Eclipse

Eclipse je zmogljivo odprtokodno razvojno okolje, primerno za izdelavo programske opreme v Javi. Izkaže se kot zelo primerno okolje za naše potrebe, saj lahko preko vtičnikov vanj integriramo vsa potrebna programska orodja.

4.1.2 JavaCC

Java Compiler Compiler (JavaCC) je popularno orodje za generiranje sintaksnih analizatorjev v Javi. Generator prebere slovnične specifikacije in jih pretvori v Java program za prepoznavanje nizov, ki ustrezajo podani gramatiki [5]. Orodje se lahko uporablja samostojno, vendar smo ga rajši uporabili v obliki vtiča (ang. plugin) za okolje Eclipse.

4.1.3 JJTree

JJTree je predprocesor za uporabo skupaj z JavaCC, ki v izvorno JavaCC kodo na različnih mestih vstavi funkcije za generiranje sintaksnih dreves. V privzetem načinu JJTree zgradi vozlišče za vsako gramatično pravilo, ki nastopa v vhodnem nizu. Takšno delovanje pa je mogoče modificirati, tako da lahko izgradnjo vozlišč za določena pravila izpustimo ali jih ustvarimo le pod določenimi pogoji. JJTree definira vmesnik *Node*, ki ga morajo

implementirati vsa vozlišča v drevesu. Vmesnik določa metode za operacije kot so določitev starša, dodajanje potomcev itd.

4.2 Generiranje analizatorjev in sintaksnih dreves

Za implementacijo leksikalnih in gramatičnih specifikacij smo uporabili orodje JavaCC. Vse specifikacije lahko podamo skupaj v datoteki s končnico .jjt, kjer lahko na ustreznih mestih vstavimo tudi ukaze za izgradnjo sintaksih dreves. JavaCC skupaj s predprocesorjem JJTree nato generira ustrezne razrede.

JavaCC specifikacija se začne s seznamom opcij, ki mu sledi deklaracija analizatorja v sledeči obliki:

PARSER_BEGIN(SNParse)
class SNParse{
...
}
PARSER_END (SNParse)

Temu sledijo specifikacije leksikalnih žetonov in gramatičnih pravil.

4.2.1 Leksikalne specifikacije

Podajanje leksikalnih specifikacij je v JavaCC preprosto. Podprto je več različnih tipov žetonov, pri naši implementaciji smo uporabili le dva; *SKIP* in *TOKEN*. Žetoni tipa *SKIP* določajo zaporedja, ki se med leksikalno analizo preskočijo in jih ne upoštevamo. V našem primeru so to le presledki. Žetoni tipa *TOKEN* predstavljajo končne simbole v naši gramatiki. Podani so v naslednji obliki:

TOKEN: {
< SIMBOL_1: [regularni izraz 1] >
| < SIMBOL_2: [regularni izraz 2] >
...
}

pri čemer SIMBOL_1, SIMBOL_2,.. predstavljajo imena simbolov. Kot primer uporabe si lahko pogledamo definicijo simbolov, ki predstavljajo cela

števila:

```
TOKEN: {
< DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
| < HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
}
```

4.2.2 Gramatične specifikacije

V privzetem načinu JavaCC gradi LL(1) sintaksni analizator, torej pri odločanju preverja en žeton vnaprej. Če nam to ne zadostuje lahko podamo večje število žetonov, torej gradimo LL(k) analizator, kjer je k število žetonov. Ker pa večji k upočasni analizator, ni smiselno uporabiti večjega k za celotni analizator. Za večino pravil namreč zadostuje LL(1) analizator. Zato lahko v JavaCC povečamo število preverjenih žetonov na ključnih točkah z uporabo ukaza *LOOKAHEAD(k)*. To nam omogoča uporabo bolj kompleksnih gramatik, hkrati pa lahko ohranimo dobre performanse analizatorja.

4.2.2.1 Osnovna pravila

Gramatična pravila si lahko ogledamo po pristopu od spodaj navzgor. Najprej definiramo pravila za sledeče osnovne gradnike:

Numerične konstante - Integer(), Float(): Ločimo celoštevilske (integer) in realne (float) številske konstante. Zapisane so lahko v različnih oblikah (desetiški, šestnajstiški, eksponentni,...). Upoštevamo tudi morebiten negativen predznak.

Argumenti - Argument(): Rezervirani nizi, ki predstavljajo argumente.

Segmentacija teksta - TextSegment(): V to kategorijo spadajo npr. znaki za prelom vrstice

Besede - TextPart(): Poljubni ASCII nizi, ki ne spadajo v ostale kategorije. Ločeni so s presledki.

Pravila za osnovne gradnike so preprosta, navadno so sestavljena le iz enega ali zaporedja dveh leksikalnih žetonov. Kot primer si lahko ogledamo pravilo,

ki obravnava realne številske konstante:

```
void Float(): { }
{
    (<MINUS> ) ? (<FLOATING_POINT_LITERAL>)
}
```

Iz teh pravil lahko izpeljemo pravila za obdelavo tekstovnih vnosov in aritmetičnih izrazov.

Tekstovni vnos -TextEntry(): je sestavljen iz besed, ločenih s presledki. Poleg besed lahko vsebuje tudi številske konstante:

```
void TextEntry(): { }
{
    (TextPart()) (TextPart() | Numeric())*
}
```

Obdelava matematičnih izrazov je podrobneje predstavljena v razdelku 4.2.2.2.

4.2.2.2 Aritmetični izrazi in vrstni red operacij

V okviru tega dela bomo implementirali zgolj osnovne operacije nad števili, kot so +,-,*,/,%,<<,>>,^,| in &. Pri pisanju gramatike moramo posebno pozornost nameniti vrstnemu redu operacij, sicer lahko pride do napak pri razreševanju matematičnih izrazov.

Vrstni red smo povzeli po specifikacijah programskega jezika C, kot je razvidno iz preglednice 4.1. Za vsako prioritetno stopnjo potrebujemo svoje gramatično pravilo.

OPERATOR	OPIS	PRIORITETA	PRAVILO
( )	oklepaji	0	UnaryExp()
* / %	množenje, deljenje, modul	1	MultiExp()
+ -	seštevanje, odštevanje	2	AddExp()
<< >>	pomik levo, pomik desno	3	ShiftExp()
&	bitni IN	4	AndExp()
^	bitni ekskluzivni ALI	5	XorExp()
|	bitni ALI	6	OrExp()

Tabela 4.1: Vrstni red operacij, od najvišje prioritete do najnižje

Pravila definiramo v rekurzivni hierarhiji na sledeč način:

```
void OrExp() : {}
{
     XorExp() ( <BIT_OR> XorExp() )*
}

void XorExp() : {}
{
    AndExp() ( <XOR> AndExp() )*
}

void AndExp() : {}
{
     ShiftExp() ( <BIT_AND> ShiftExp() )*
}

void ShiftExp() : { }
{
    AddExp() ( ( <SHIFTL> | <SHIFTR> ) AddExp() )*
}

void AddExp() : { }
{
    MultiExp() ( ( t=<PLUS> | t=<MINUS>) MultiExp() )*
}

void MultiExp() : { }
{
    UnaryExp()( ( t=<MULTIPLY> | t=<DIVIDE> | t=<REM> ) UnaryExp() )*
}

void UnaryExp() : {}
```

```
{
    ( <LPAREN> OrExp() <RPAREN> | Argument() | Numeric() )
}
```

Pri nekaterih pravilih se lahko pojavlja več različnih operatorjev (množenje, deljenje). V tem primeru moramo shraniti zaporedje operatorjev, kot se pojavijo v izrazu, da zagotovimo pravilnost interpretacije.

4.2.2.3 Izpeljana pravila

S pomočjo osnovnih pravil lahko opišemo naprednejše formalizme, kot jih določa sporočilni nivo SNP. Nekatere smo za potrebe osnovne implementacije nekoliko poenostavili glede na uradne specifikacije.

Izraz - Expression(): Predstavlja opis aritmetičnega izraza, podanega v zavitih oklepajih. Je eno najbolj bistvenih pravil sporočilnega nivoja.

```
void Expression() :{ }
{
    <LBRACE> OrExp() <RBRACE>
}
```

Deklaracija - Declaration(): posebna vrsta izraza, katerega vrednost predstavlja niz, v nasprotju z numerično vrednostjo navadnega izraza. Navadno se uporablja za deklaracijo spremenljivk.

```
void Declaration() : { }
{
    <LBRACE><COLON>TextEntry()<RBRACE>
}
```

Urejanje niza - TextFormat(): Polje za urejanje tekstovnega izpisa. Oblika je podana z določilom, ki mu sledi tekstovni vnos v zavitih oklepajih.

```
void TextFormat() : { }
{
    <T_FORM><LBRACE>TextEntry()<RBRACE>
}
```

Enumeracija - Enumeration(): Predstavitev enumeracije. Vsebuje najmanj en celoštevilski argument in seznam pripadajočih tekstovnih

vrednosti.

```
void Enumeration() : { }
{
    <LBRACE>Argument()<COLON>TextPart()
    ((<COMMA> |<SEMICOLON>)TextPart())*<RBRACE>
}
```

Enota - Unit(): Predstavitev enote. Sestoji iz tekstovnega niza v oglatih oklepajih.

```
void Unit() : { }
{
    (<LBRACKET> (TextEntry())? <RBRACKET>)
}
```

Dodeljevanje spremenljivke - Assignment(): Povezovanje deklarirane spremenljivke z izrazom ali enumeracijo in enoto. Deklaracija in vrednost sta povezani z enačajem.

```
void Assignment() { }
{
    (Declaration())(<ASSIGN>)(SNStruct())(Unit())?
}
```

Pri tem SNStruct() predstavlja pomožno pravilo, uvedeno zaradi boljše preglednosti:

```
void SNStruct() #void: {}
{
    LOOKAHEAD(3) Enumeration() | LOOKAHEAD(3) Expression()
    | LOOKAHEAD(3) Declaration()
}
```

Ko imamo definirana pravila za vse potrebne formalizme, potrebujemo še začetno pravilo, ki vse elemente sestavi skupaj.

Opis sporočila - Description(): Predstavlja začetno pravilo, iz katerega se lahko izpeljejo vsi možni nizi v jeziku opisov SNP sporočil.

```
SNTRDescription Description(int id) : { }
{
    ( Argument() | LOOKAHEAD(2) Assignment() | SNStruct()
```

```
    | TextSegment() |TextEntry() |Unit())*
}
```

4.2.3 Izgradnja dreves

Implementacijo sintaksnih dreves smo lahko v veliki meri avtomatizirali s pomočjo orodja JJTree. Uporabili smo napredni način, kjer lahko uporabljamo različne tipe vozlišč v drevesu. JJTree sam generira osnovno implementacijo vsakega tipa vozlišča, ki ga potem lahko modificiramo za lastne potrebe. V privzetem načinu se vozlišča avtomatsko generirajo za vsako pravilo, kar pa ni vedno zaželeno. Če hočemo to preprečiti, lahko k gramatičnemu pravilu dodamo dve vrsti določil:

- **Prazno vozlišče** ***#void*** - To določilo prepreči kreiranje vozliča. Nekatera pravila nam koristijo samo v sintaksni fazi in ni potrebe, da bi zaradi njih vstavljali vozliča v abstrakstno sintaksno drevo, saj bi to le otežilo interpretacijo.
- **Pogojno vozlišče** ***#ime_vozlišča(pogoj)*** - To določilo dovoli kreiranje vozlišča le, če je izpolnjen pogoj v oklepaju. Pogost pogoj je recimo *#ime_vozlišča(>1)*, kar pomeni, da se vozlišče kreira, če je število potomcev na skladu večje od 1. To nam pride zelo prav pri implementaciji razreševanja matematičnih izrazov in vrstnega reda operacij.

Poleg določil lahko med specifikacije pravil vstavimo tudi uporabniške ukaze v Javi. Pri tem lahko ukazi dostopajo do trenutnega vozlišča v izgradnji preko posebnega identifikatorja *jjtThis.* Na ta način lahko dostopamo do vseh metod, ki so definirane v vozlišču. To lastnost smo uporabili predvsem za shranjevanje relevantnih informacij v vozlišča. Kot primer si lahko ogledamo shranjevanje določila pri gramatičnem pravilu za formatiranje nizov:

```
void TextFormat() :
{
    Token t;
}
{
    t=<T_FORM><LBRACE>TextEntry()<RBRACE>
    { jjtThis.setEntry(t.image);}
}
```

Vrednost določila smo shranili v novonastalo vozlišče v obliki niza s pomočjo

metode *setEntry(String n)*. Takšno metodo vsebujejo vsa vozlišča, pri katerih je potrebno direktno shranjevanje informacij. Podatki iz sintaksne faze so tako vedno shranjeni v obliki niza, do njih pa lahko dostopamo kar z metodo *toString()*. Izjema je le korensko vozlišče *Description*, kjer ob kreiranju podamo in shranimo še celoštevilsko ID konstanto. Do nje lahko dostopamo z metodo *getID()*.

4.3 Interpretacija sporočil

Pri interpretaciji sintaksnega drevesa smo se odločili za uporabo pristopa z *obiskovalcem*. V ta namen lahko v nastavitvah orodja JJTree nastavimo opcijo *VISITOR*. V tem primeru nam JJTree v vsako vozlišče avtomatsko vstavi metodo za sprejem obiskovalca. Poleg tega nam generira tudi vmesnik, ki ga morajo implementirati interpretacije. Pri naši implementaciji sporočilnega nivoja vmesnik vsebuje naslednje metode:

```
public interface SNParseVisitor
{
  public Object visit(SimpleNode node, Object data);
  public Object visit(SNTRDescription node, Object data);
  public Object visit(SNTRAssignment node, Object data);
  public Object visit(SNTRUnit node, Object data);
  public Object visit(SNTRDeclaration node, Object data);
  public Object visit(SNTRExpression node, Object data);
  public Object visit(SNTRTextFormat node, Object data);
  public Object visit(SNTRTextSegment node, Object data);
  public Object visit(SNTREnumeration node, Object data);
  public Object visit(SNTROR node, Object data);
  public Object visit(SNTRXOR node, Object data);
  public Object visit(SNTRAND node, Object data);
  public Object visit(SNTRShift node, Object data);
  public Object visit(SNTRAdd node, Object data);
  public Object visit(SNTRMultiply node, Object data);
  public Object visit(SNTRTextEntry node, Object data);
  public Object visit(SNTRTextPart node, Object data);
  public Object visit(SNTRArgument node, Object data);
  public Object visit(SNTRInteger node, Object data);
  public Object visit(SNTRFloat node, Object data);
}
```

Program 4.1: SNParseVisitor.java

Za naše potrebe smo generiran vmesnik nadgradili še z metodo *ReInit*, kjer obiskovalca reinicializiramo z novimi argumenti, izhodnim tokom in po potrebi s komponento za urejanje izpisa:

```
public interface SNExtendVisitor extends SNParseVisitor {
```

```
    public void ReInit(byte[] args, OutputStream out);
    public void ReInit(byte[] args, OutputStream out, SNFormat form);
}
```

Program 4.2: SNExtendVisitor.java

4.3.1 Povezovanje argumentov

Obiskovalcu moramo ob inicializaciji podati pripadajoče polje argumentov. Poleg tega objekt vsebuje globalno spremenljivko *arg_index,* ki hrani trenutno pozicijo v polju argumentov. Ko pri prehajanju drevesa naletimo na argument, preberemo shranjeno oznako in izračunamo dolžino v bajtih:

```
private int argLength(String arg) {
    if(arg.charAt(0)=='%') {
        if(arg.charAt(1)=='i'|| arg.charAt(1)=='u' ||arg.charAt(1)=='x'
        ||arg.charAt(1)=='I'|| arg.charAt(1)=='U'|| arg.charAt(1)=='X'
        || arg.equalsIgnoreCase("%hf")|| arg.equalsIgnoreCase("%hF"))
            return 2;

        else if(arg.charAt(1)=='h'||arg.charAt(1)=='c'
        ||arg.charAt(1)=='b')
            return 1;

        else if(arg.equals("%Lf") || arg.equals("%LF"))
            return 10;

        else if(arg.charAt(1)=='L' || arg.equals("%lf")
        || arg.equals("%lF"))
            return 8;

        else if(arg.charAt(1)=='l'||arg.charAt(1)=='d'
        ||arg.charAt(1)=='f'||arg.charAt(1)=='F')
            return 4;

    }
    return 0;
}
```

Program 4.3: ArgumentLength.java

Nato iz polja preberemo ustrezno število bajtov, jih shranimo v ustrezen podatkovni tip, povečamo *arg_index* in vrnemo vrednost argumenta. Metoda za obisk argumenta tako lahko vrača različne podatkovne tipe (Integer, Long, Double, Character).

4.3.2 Razreševanje aritmetičnih izrazov

Razreševanje aritmetičnih izrazov je razmeroma preprosto. Ob obisku aritmetičnega vozlišča se izračunajo numerične vrednosti vseh potomcev, na njih se opravijo podane operacije in vrne rezultat. Postopek si lahko ogledamo na primeru pomika (shift):

```
public Object visit(SNTRShift node, Object data) {

   long sum=0;
   for (int i=0; i<node.jjtGetNumChildren(); i++) {
      Object d=node.jjtGetChild(i).jjtAccept(this, data);
      sum=bitCalc(i, sum, d, opCalc(i, node.toString()));
   }
   return sum;
}
```

Program 4.4: ShiftVisitor.java

Ker imamo tukaj dva možna operatorja (pomik levo in desno), je potrebno v vozlišču shraniti zaporedje operatorjev v izrazu. Zaporedje je shranjeno v obliki niza dolžine *število_potomcev-1*. S pomočjo funkcije *opCalc()* pregledamo niz in dobimo trenutni operator:

```
private int opCalc(int inx, String operators) {
   if(inx<1) return -1;
   if(inx<=operators.length()) {
      char c=operators.charAt(inx-1);
      if(c=='+') return ADD;
      else if(c=='-') return SUB;
      else if(c=='*') return MUL;
      else if(c=='/') return DIV;
      else if(c=='%') return REM;
      else if(c=='<') return SLEF;
      else if(c=='>') return SRIG;
      }
      return -1;
   }
}
```

Program 4.5: OpCalc.java

Operator in vrednosti nato posredujemo funkciji za računanje vmesnega rezultata:

```
private long bitCalc(int inx, long sum, Object val, int operator) {
   long temp_sum=sum;
   if(val instanceof java.lang.Integer) {
      if(inx==0) return ((Integer)val).longValue();
      switch(operator) {
```

```
            case SLEF: return temp_sum<<=(Integer)val;
            case SRIG: return temp_sum>>=(Integer)val;
            case AND: return temp_sum&=(Integer)val;
            case OR: return temp_sum|=(Integer)val;
            case XOR: return temp_sum^=(Integer)val;
        }
    }
    return temp_sum;
}
```

Program 4.6: BitCalc.java

Operacijo ponavljamo za vse potomce vozliča in vrnemo dobljen rezultat.

4.3.3 Interpretacija ostalih konstruktov

Pri interpretaciji SNP formalizmov običajno združimo več osnovnih konstruktov v smiselno, berljivo obliko. Kot primer si lahko ogledamo interpretacijo enumeracije:

```
public Object visit(SNTREnumeration node, Object data) {
    int inx=0;
    if(node.jjtGetNumChildren()>1) {
        Object d=node.jjtGetChild(0).jjtAccept(this, data);
        if(d instanceof java.lang.Integer){
            inx=(Integer)d;
            if(inx<node.jjtGetNumChildren()-1 && inx>=0) {
                Object e=node.jjtGetChild((inx+1)).jjtAccept(this, data);
                if(e instanceof java.lang.String) {
                    if(form!=null) e=form.format(node, (String)e);
                    return (String)e;
                }
            }
        }
    }
    return null;
}
```

Program 4.7: Enumeration.java

Glede na vrednost celoštevilskega argumenta izberemo ustrezen niz. Če imamo definiran poseben objekt za urejanje izpisa, mu lahko dobljen niz pošljemo v obdelavo z metodo *format()*. Če objekt nima definirane metode za urejanje izhoda iz enumeracije se vrne osnovni niz. Na podoben način so realizirane interpretacije tudi za ostale formalizme.

Interpretacija osnovnega vozlišča (Description) ostale elemente združi v enoten niz, primeren za izpis:

```
public Object visit(SNTRDescription node, Object data) {
   String s="";
   for (int i=0; i<node.jjtGetNumChildren(); i++) {
      Object d=node.jjtGetChild(i).jjtAccept(this, data);
      if(d instanceof java.lang.String ) {
         s+=d;
      }
      else if(d instanceof java.lang.Integer ) {
         s+=((Integer)d).toString();
      }
      else if(d instanceof java.lang.Long ) {
         s+=((Long)d).toString();
      }
      else if(d instanceof java.lang.Double ) {
         s+=((Double)d).toString();
      }
      else if(d instanceof java.lang.Character ) {
         s+=(Character)d;
      }
      if(i<node.jjtGetNumChildren()-1)
         s+=" ";
   }
   try {
      if(form!=null) s=form.format(node, s);
      out_str.write(s.getBytes());
   } catch(Exception e) { return null; }
   return s;
}
```

Program 4.8: Description.java

4.4 Ogrodje

Za izdelavo ogrodja sporočilnega nivoja smo najprej definirali dva abstraktna razreda, iz katerih lahko potem izpeljemo bolj konkretno implementacijo. Razred *SNSession* nam definira SNP sejo, medtem ko nam razred *SNMessageIO* podaja okvir sporočilnega nivoja. Razreda podajata zgolj osnovne specifikacije, vse podrobnosti implementacije se realizirajo v izpeljanih razredih.

```
package snp.msg.box;

public abstract class SNSession {
   protected long addr;

   public SNSession(long addr) {
      this.addr=addr;
   }
   public long getAddress() {
      return addr;
   }
```

```
    public abstract int updateData(byte[] args);
    public abstract int sendRequest(int id, byte[] args);
    public abstract byte[] getArguments(int id);
    public abstract Object getMessageData();

}
```

Program 4.9: SNSession.java

```
package snp.msg.io;
import snp.msg.box.SNSession;

public abstract class SNMessageIO {
    protected SNSession parent;
    protected String prefix;

    public SNMessageIO(SNSession parent, String prefix)
    {
        this.parent=parent;
        this.prefix=prefix;
    }

    public abstract void print(OutputStream out);
    public abstract void print(OutputStream out, int start, int end);
    public abstract void print(OutputStream out, SNFormat form,
    SNExtendVisitor visitor, int start, int end);
    public abstract void updateTree(byte[] msg);
    public abstract void setParameter(String var, Object value);

}
```

Program 4.10: SNMessageIO.java

4.4.1 Seja

V izpeljanem sejnem razredu sta definirana dva vektorja. Prvi služi za shranjevanje dobljenih sporočil, v drugem pa se shranjujejo odhodna sporočila. Ko seja prejme sporočilo lahko doda nov vnos v vektor ali pa popravi obstoječega z novimi vrednostmi:

```
public int updateData(int size, byte[] data) {

        int id=getMsgID(data);
    if(id==update_status && update_status<MAX_MSG) {
        update_status++;
        msg_fifo.add(new byte[]{0,0x60,update_status});
    }
    for(int i=0; i<msg_vec.size(); i++) {
        byte[] m=msg_vec.elementAt(i);
        if(id==getMsgID(m)) {
            updateEntry(m, data, i);
            return 1;
        }
```

```
    }
    //new msg ID, new entry
    msg_vec.add(data);
    String desc =new String(data);
    if(desc.indexOf(0)>1)
       msgIO.updateTree(data);

    return 0;
}
```

Program 4.11: UpdateData.java

Sejni objekt po kreiranju navadno ne dobi vseh sporočil in njihovih opisov, zato mora seja poslati poizvedbe, preko katerih dobi celoten opis entitete, ki jo predstavlja. Poizvedbe se vstavijo v izhodni vektor v obliki izhodnih sporočil. Ker nočemo naenkrat preveč obremeniti sistema, jih pošiljamo posamezno - naslednjo zahtevo podamo šele, ko dobimo odgovor na prejšnjo.

Ko seja prejme nov opis sporočila, ga navadno pošlje objektu za obdelavo sporočil. Prav tako mu lahko posreduje zahteve za izpis. Na zahtevo seja posreduje tudi polje argumentov za dano sporočilo.

4.4.2 Okvir

V okvirnem razredu je definiran vektor, v katerega se shranjujejo sintaksna drevesa. Drevesa se navadno gradijo sproti, ko objekt dobi nove opise s strani sejnega objekta:

```
public void updateTree(byte[] msg) {
   String desc =new String(msg);
   int clen = desc.indexOf(0);
   if (clen > msg.length || msg.length - clen < 3 || clen<2) {return; }

   int msg_id=(int)((((0x000000FF &(int)msg[clen+1]) << 8)
   | (0x000000FF &(int)msg[clen+2]))&0x3FF);

   //node already present
   if(contains(msg_id)>=0) return;
   try{
      InputStream io=new ByteArrayInputStream(desc.substring(0, clen).getBytes
          ());
      SNParse parser=new SNParse(io);
      try {
          SNTRDescription n = parser.Description(msg_id);
          desc_vec.add(n);
          this.sortVec();
      } catch (ParseException e) {
          System.out.println("Parse error: "+desc.substring(0, clen));
          System.out.println(e.getMessage());
        }
```

```
    } catch(Exception e) {
        System.out.println("Exception: "+e.getMessage());
    }
}
```

Program 4.12: UpdateTree.java

Iz sporočila se izloči polje za opis in ID, ki se jih posreduje analizatorju. Analizator zgradi drevo, ki se ga nato vstavi v vektor. Vektor se pri vsakem dodajanju razvršča po ID številih sporočil, kar omogoča lažje iskanje in izpis.

Ob zahtevi za izpis sporočil lahko poleg izhodnega toka podamo več različnih parametrov:

- **Začetni in končni ID:** V primeru, ča uporabnik želi izpisati le določeno podmnožico sporočil.
- **Komponenta za urejanje izpisa:** Omogoča dodatno oblikovanje izpisa po željah uporabnika.
- **Obiskovalec:** Omogoča uporabniku, da definira svojo interpretacijo sintaksnega drevesa. V kolikor ta parameter ni podan se uporabi privzeta interpretacija, kot je opisana v 4.3.

Komponenta za urejanje izpisa implementira vmesnik *SNFormat* s sledečimi metodami:

```
public interface SNFormat {

    public String format(SimpleNode node, String entry);
    public String format(SNTRTextSegment node, String entry);
    public String format(SNTRTextFormat node, String entry);
    public String format(SNTRDescription node, String entry);
    public String format(SNTRExpression node, String entry);
    public String format(SNTREnumeration node, String entry);
    public String format(SNTRUnit node, String entry);
    public String format(SNTRDeclaration node, String entry);
    public void prepend(OutputStream out, String entry, int option);
    public void append(OutputStream out, String entry, int option);
}
```

Program 4.13: SNFormat.java

Poglavje 5

Testiranje

5.1 Testna aplikacija

Za praktično testiranje naše implementacije smo izdelali aplikacijo za zajem in izvoz podatkov iz naprav, ki implementirajo SNP protokol. Podatke aplikacija izvozi v HTML datoteko v obliki, primerni za prikaz v spletnih brskalnikih. Aplikacijo sestavlja gonilnik za komunikacijo preko serijskega vmesnika, komponenta za nizkonivojska obdelavo SNP paketov, sejni nivo, sporočilni nivo in komponenta za izvoz podatkov v datoteko.

5.1.1 Komunikacija

Povezovanje in komunikacija med našo aplikacijo in SNP napravami poteka preko serijskega vmesnika. Navadno uporabimo RS-232 standard ali njegovo brezžično emulacijo preko Bluetooth SPP profila. Pri realizaciji komponente za serijsko komunikacijo smo uporabili knjižnico RXTX [7]. Komponento predstavlja relativno preprost razred, katerega metode omogočajo vzpostavljanje povezave s podanim priključkom, sprejemanje in pošiljanje podatkov ter prekinitev povezave. Željen priključek se poda kot argument pri zagonu programa. Po uspešni vzpostavitvi povezave si aplikacija z napravo periodično izmenjuje pakete. Ti lahko predstavljajo le "ping" pakete ali pa vsebujejo dejanska sporočila. Prejeti paketi se shranjujejo v medpomnilnik za nadaljnjo obdelavo.

5.1.2 Nižjenivojska obdelava

Na najnižjem SNP nivoju aplikacija skrbi za vzdrževanje povezave, pripravo sporočil na pošiljanje prek serijskega vmesnika in osnovno obdelavo prejetih sporočil. Ob sprejemu sporočila se odstrani okvir in naredi preverjanje za napake. Pravilno prenešeno sporočilo se zapiše v medpomnilnik, od kjer ga prevzame vmesni nivo.

Vmesni nivo periodično dostopa do medpomnilnika nižjega nivoja in prebere prejeta sporočila. Vsako sporočilo odpakira, ugotovi njegov izvor in ga posreduje ustrezni seji. Iz sej zbira tudi odhodna sporočila, jih ustrezno obdela in preda nižjemu nivoju za pošiljanje. V enem ciklu se pošlje največ eno sporočilo.

5.1.3 Urejanje in izvoz podatkov

V naši aplikaciji želimo podatke izvoziti v obliki HTML datoteke. Ime datoteke je podano kot argument programa. Ob zagonu se ustvari izhodni tok za pisanje v datoteko, ki se ga posreduje objektom za obdelavo sporočil.

Podatke iz sistema želimo organizirati v pregledni in človeku lahko berljivi obliki, za kar je HTML format zelo primeren. Izvožene podatke tako lahko prikazujejo spletni brskalniki na praktično vseh obstoječih platformah. Z uporabo strežnika lahko na enostaven način omogočimo tudi vpogled v sistem na daljavo.

Urejanje podatkov lahko enostavno realiziramo s pomočjo vmesnika *SNFormat*, opisanega v 4.4.2. V metodi *prepend()* definiramo glavo dokumenta, z metodo *append()* pa dokument zaključimo. Ostale metode definirajo obliko ostalih elementov. Za naše potrebe ne potrebujemo posebej kompleksnih elementov, v glavi definiramo nekaj stilov za označevanje različni elementov sporočil:

```
<html>
<head>
<style type="text/css">
ex {color:MediumBlue}
un {color:MidnightBlue;font-weight: bold;}
de {color:ForestGreen ;font-style:italic;font-weight: bold; }
</style>
</head>
```

<body>

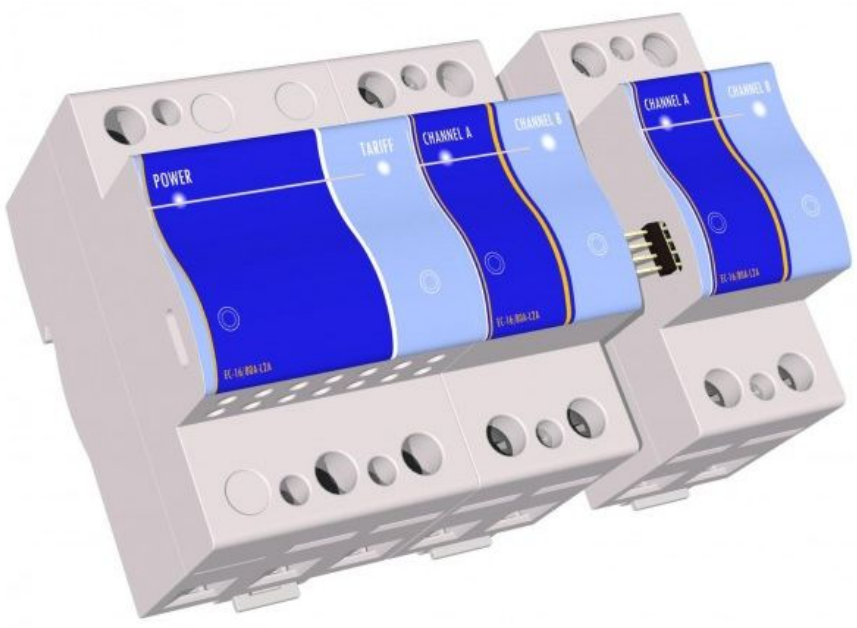
S temi stili in osnovnimi HTML oznakami za velikost in obliko teksta lahko uredimo pregleden izpis SNP elementov. Vsak element opremimo z ustreznimi HTML oznakami in dobljen niz vrnemo obiskovalcu. Na koncu izpisa vsake entitete dodamo še čas zapisa podatkov. Primer uporabe se nahaja v 5.3

5.2 Testni sistem

Kot primer realnega sistema za testiranje naše aplikacije smo uporabili sistem za kontrolo energije podjetja Energijski Konduktorji. Gre za enega prvih produktov, ki implementira SNP protokol. Sistem je sestavljen iz povezovalnega modula in večih modulov za kontrolo energije.

Povezovalni modul (Energy Control Link) v prvi vrsti skrbi za dovod napajanja ostalim modulom in vsebuje vmesnike za komunikacijo z zunanjim svetom. Poleg tega omogoča tudi izračun skupne porabe in nastavljanje globalnih omejitev. Za zunanjo komunikacijo vsebuje več različnih vmesnikov (M-Bus, RS-232,...), za nas pa je najprimernejša uporaba vgrajene Bluetooth komponente.

Modul za kontrolo energije (Energy Control Module) je naprava, ki lahko nadzoruje in upravlja z največ dvema električnima priključkoma. Navadno se module namesti v električno omarico in skupaj s povezovalnim modulom poveže med sabo s 4-pinskim priključkom [2]. Sistem omogoča pregled različnih parametrov in količin v sistemu, nastavljanje omejitev in prioritet itd.



Slika 5.1: Modul za povezovanje in dva modula za kontrolo energije. Med sabo so povezani s 4-pinskimi konektorji.

Za povezavo aplikacije s sistemom za nadzor energije smo uporabili Bluetooth komponento. Serijski Bluetooth profil (SPP) se obnaša kot navaden serijski vmesnik, zato ga lahko brez težav uporabimo v naši aplikaciji.

5.3 Test aplikacije na sistemu

Testiranje aplikacije smo opravili na sistemu, sestavljenem iz dveh različnih naprav, kot je opisano v 5.2. V nadaljevanju so predstavljeni opisi sporočil modula za kontrolo energije in dobljen HTML izpis.

Opisi sporočil iz modula za kontrolo energije so zbrani v tabeli 5.1.

ID	OPIS SPOROČILA
0	%t1{EC-16/20AHR-L2A-1.2};Rev:1628/1646; ec{
1	Energy {:E}={18e-5*((%U<<8)+%hu)}[kWh] in {:dt}={0.0417*%U}[min];Channel {:ChA}={%hu} {:ChB}={%hu}
2	Distribution {:Ea}={18e-5*((%U<<8)+%hu)}[kWh], {:Eb}={18e-5*((%U<<8)+%hu)}[kWh]
3	Energy Sum {:E_sum}={18e-5*%lU}[kWh];%t2{Load Tracking}
4	T_on:%lU[s] N:%U {:I_max}={%hu*70}[mA] {:I_min}={%hu*70}[mA] {:T_max}={%U}[s] {:T_min}={%U}[s]
5	Power {:P}={0.5*%U}[W] {:U_avg}={%U*(%U/299827)}[V] {:I_rms}={0.002174*%U}[A] {:f}={1.4*%hi}[o]
6	%t2{Energy Limits};{:E_lim1}={47e-3*%U}[kWh/day/tariff1];{:E_lim2}={47e-3*%U}[kWh/day/tariff2]
7	%t2{Demand};Maximum {:I_max}={.002174*%U}[A];Typical {:I_typ}={2.174*%U}[mA]
8	Standby Thresholds {:I_stdbyA}={2.174*%hu}[mA] {:I_stdbyB}={2.174*%hu}[mA]
9	Shedded time {:ASt}={21.84*%U}[min] {:BSt}={21.84*%U}[min]
10	%t2{Groups};Mask bits %hx %hx %hx %hx %hx %hx %hx %hx {:Inv}={%hx} {:Tariff}={%hx}
11	External input {:Xfn}={%hx}{:Xtr}={%hx};%t2{General Settings}
12	Time {:t}={%lU}[UTC];Phase {:phase}={%hu>>5};Opts. {:optAB}={%hu} {:optA}={%hu} {:optB}={%hu}

Tabela 5.1: Opisi sporočil modula za kontrolo energije

V opisih lahko najdemo večino konstruktov, ki smo jih implementirali v sklopu analize sporočil, torej lahko sistem služi kot zadovoljiva testna platforma. Dobljen izpis enega od kontrolnih modulov je predstavljen v 5.2. Vidimo, da izpis služi svojemu namenu in učinkovito predstavi parametre naprave uporabniku. Na enak način je v dokumentu predstavljen tudi povezovalni modul.

Device1

EC-16/20AHR-L2A-1.2
Rev:1628/1646

Energy *E*=0**[kWh]** in *dt*=1324.183**[min]**
Channel *ChA*=1 *ChB*=0
Distribution *Ea*=0**[kWh]** , *Eb*=0**[kWh]**
Energy Sum *E_sum*=4.25**[kWh]**
Load Tracking
T_on: 137367329 **[s]** N: 0 *I_max*=0**[mA]** *I_min*=0**[mA]** *T_max*=15**[s]** *T_min*=46336**[s]**
Power *P*=0**[W]** *U_avg*=240.546**[V]** *I_rms*=0**[A]** *f*=0**[o]**
Energy Limits
E_lim1=60.0**[kWh/day/tariff1]**
E_lim2=50.0**[kWh/day/tariff2]**
Demand
Maximum *I_max*=0**[A]**
Typical *I_typ*=0**[mA]**
Standby Thresholds *I_stdbyA*=10.87**[mA]** *I_stdbyB*=0**[mA]**
Shedded time *ASt*=617198.4**[min]** *BSt*=550564.559**[min]**
Groups
Mask bits 0 0 0 0 32 116 105 108 *Inv*=101 *Tariff*=32
External input *Xfn*=123 *Xtr*=71
General Settings
Time *t*=3068530**[UTC]**
Phase *phase*=1
Opts. *optAB*=0 *optA*=9 *optB*=9

Exported: Sun Jun 12 15:50:12 CEST 2011

Slika 5.2: Parametri modula za kontrolo energije v HTML obliki

5.4 Test zmogljivosti

Poleg praktičnega testa na napravi smo opravili tudi nekaj meritev hitrosti izvajanja in porabe pomnilnika pri obdelavi na sporočilnem nivoju. Za test smo uporabili različno število generiranih sporočil podobnega tipa, kot so uporabljeni v testnem sistemu 5.1. Takšen test je uporaben za analizo delovanja posameznih komponent in njihove zahtevnosti. Iz meritev porabe pomnilnika lahko dobimo približno predstavo o tem, kakšne velikosti sistemov lahko obdelujemo na posameznih platformah. Tovrstni testi nam pomagajo tudi pri dodatnih optimizacijah in nadgradnjah implementacije.

Meritve smo opravljali na osebnem računalniku sledečih specifikacij:

- CPE: Intel E8400 3.0 GHz
- Pomnilnik: 2.0 Gb DDR2
- OS: Windows XP SP3

Pri meritvah smo uporabljali naslednje pripomočke:

- Eclipse Test and Performance Tools Platform (TPTP)
- Funkcija *System.currentTimeMillis()* v Javi.

Pri merjenju porabe pomnilnika nas zanima predvsem količina pomnilnika, uporabljena za shranjevanje sporočil in sintaksnih dreves. Pri merjenju časa izvajanja nas zanimata dva parametra; čas izgradnje sintaksnih dreves iz opisov sporočil in čas za interpretacijo in izpis podatkov.

SEJE x MSG	**Skupaj MSG**	**Pomnilnik (kB)**	**Izgradnja (ms)**	**Interpretacija (ms)**
5 x 20	100	125.82	48	16
5 x 50	250	311.64	79	31
10 x 50	500	628.58	93	42
10 x 100	1000	1219.02	172	78
20 x 100	2000	2438.89	281	110
50 x 100	5000	6101.45	562	250
100x100	10000	12196.32	1063	468

Tabela 5.2: Poraba pomnilnika in časi izvajanja

V tabeli 5.2 so predstavljene meritve izvajalnega časa in porabe pomnilnika za različno število sej in sporočil. Čas in poraba po pričakovanjih naraščata bolj ali manj linearno. Ugotovimo da čas, potreben za izgradnjo dreves, več kot dvakrat presega čas interpretacije sporočil, zato je smotrno graditi sintaksna drevesa le enkrat in jih shranjevati v pomnilniku. To je posebej pomembno, kadar aplikacija zahteva pogosto interpretacijo podatkov. Pri takšnem pristopu moramo sicer plačati ceno pri porabi pomnilnika, vendar nam omogoča boljšo odzivnost in manjšo obremenitev procesorja.

5.5 Možne nadgradnje

V okviru tega dela smo izdelali osnovno implementacijo, namenjeno kot ogrodje za nadaljnje delo. Tako obstaja še veliko prostora za nadgradnje in izboljšave na naslednjih področjih:

- **Podatkovne strukture:** Potrebno je definirati elemente za predstavitev podatkovnih struktur, vektorjev in matrik, kot so opisane v specifikacijah sporočilnega nivoja.
- **Naprednejši elementi:** Tukaj obravnavamo koncepte, kot so globalne in lokalne spremenljike, atributi spremenljivk itd.
- **Vgnezdeni izrazi:** Obsega izdelavo podpore za gnezdenje izrazov, kar skupaj z uporabo podatkovnih struktur omogoča predstavitev kompleksnejših konstruktov.
- **Interaktivnost:** Izboljšati je potrebno podporo za nastavljanje parametrov in interpretacijo sporočil za potrebe interaktivnih aplikacij.
- **Obdelava izrazov:** Potrebno je omogočiti uporabo naprednejših matematičnih funkcij, skupaj s podporo vektorskim in matričnim operacijam. V ta namen je potrebno vdelati tudi podporo zunanjim orodjem za naprednejšo obdelavo. Trenutno je predvidena uporaba JMathLib[8]. Tako se omogoči bolj fleksibilna uporaba, saj lahko uporabnik definira svoje skripte za obdelavo.
- **Predstavitev podatkov:** V osnovni verziji smo že realizirali osnovne objekte za predstavitev podatkov, ki pa jih je potrebno ustrezno nadgraditi in prilagoditi ob dodajanju novih zmogljivosti.

Poglavje 6

Sklepne ugotovitve

V okviru diplomskega dela je predstavljena implementacija sporočilnega nivoja Sensor Network Protokola v programskem jeziku Java. Želena je bila izdelava kompaktne implementacije, primerne za splošno uporabo v različnih sistemih in aplikacijah. Za izdelavo modularne, objektno orientirane implementacije se Java izkaže zelo dobro. Pomemben kriterij pri izbiri jezika je bila tudi kompatibilnost s čim več različnimi platformami in operacijskimi sistemi. Pred konkretno izdelavo programskega dela je bilo potrebno preučiti teoretične principe in narediti načrt komponent. Poglavitni del predstavlja komponenta za analizo SNP sporočil, ki smo ji tudi namenili največ pozornosti. Pri načrtovanju sporočilnega nivoja smo si pomagali s teorijo iz področja leksikalne, sintaksne in semantične analize. Pomemben del predstavljajo tudi komponente za hranjenje podatkov in komunikacijo z ostalimi nivoji.

V veliko pomoč pri sami realizaciji so bila orodja za generiranje kode po leksikalnih in sintaksnih specifikacijah. Ročno pisanje celotne implementacije bi bilo namreč mnogo bolj zamudno, poleg tega pa je z uporabo generatorjev olajšano tudi vzdrževanje in nadgrajevanje.

Za ilustracijo uporabe in test na realnem sistemu sem izdelal preprosto aplikacijo za zajem in izvoz podatkov iz SNP sistemov. Aplikacija omogoča povezovanje in komunikacijo s SNP sistemom preko Bluetootha ali navadnega serijskega vmesnika. V ta namen je bilo potrebno izdelati vmesnik za serijsko komunikacijo in realizirati nižjenivojske funkcije SNP protokola. Nižji nivoji skrbijo za komunikacijo s sistemom in izmenjavo sporočil. Iz sistema se zbere podatke iz naprav in jih pošlje sporočilnemu nivoju v obdelavo, nato pa se

jih ustrezno formatirane izvozi v HTML datoteko. Na tak način se lahko omogoči vpogled v sistem preko spletnega brskalnika. Aplikacija je uporabna tudi za preverjanje pravilnosti sestave sporočil pri razvoju SNP sistemov.

Testiranje na realnem sistemu se je izkazalo za smotrn korak v razvoju, saj se je med njim razkrilo še nekaj manjših pomanjkljivosti v implementaciji sporočilnega nivoja. Te napake so bile hitro odpravljene in v splošnem se je implementacija izkazala za uspešno.

V tem delu opisana implementacija bo služila tudi kot ogrodje za nadaljnji razvoj, saj pušča še veliko prostora za nadgradnje in izboljšave. Dodajanje novih komponent je zaradi učinkovite zasnove dokaj enostavno opravilo. Naslednji korak v razvoju predstavlja izboljšanje podpore za interaktivne aplikacije in omogočanje uporabe zunanjih orodij za obdelavo podatkov, kot je recimo JMathLib. Za naprednejšo uporabo bo potrebno implementirati tudi kompleksnejše strukture, kot so definirane v SNP specifikacijah. Zaradi želje po čim večji kompaktnosti v osnovni različici niso podprte podatkovne strukture in ostali naprednejši elementi. Toda osnovna verzija je kljub omejitvam vseeno primerna za uporabo, posebej pri napravah manjše kompleksnosti, katerim je SNP protokol tudi namenjen.

Literatura

[1] Andrew W. Appel, *Modern Compiler Implementation in Java*, Second edition, Cambridge University Press, 2002

[2] Uroš Platiše, Mihael Mohorčič, *SIB: Sensor Instrumentation Bus for power and energy control units*, MIDEM, vol. 133, no. 1, 2010.

[3] Uroš Platiše, *SNP Message preliminary specifications*, Dostopno na http://www.isotel.org/Home/

[4] *http://www.gnu.org/s/libc/* (zadnji obisk: 19.5.2011)

[5] *http://javacc.java.net/* (zadnji obisk: 30.5.2011)

[6] *http://www.difranco.net/cop2220/op-prec.htm* (zadnji obisk: 26.5.2011)

[7] *http://rxtx.qbang.org/wiki/index.php/Main_Page* (zadnji obisk: 14.6.2011)

[8] *http://www.jmathlib.de/index.php* (zadnji obisk: 16.6.2011)

Slike

3.1 Primer končnega avtomata za predstavitev realnih števil 17
3.2 Prikaz možne izpeljave opisa SNP sporočila 18
3.3 Prikaz strukture abstraktnega sintaksnega drevesa 20
3.4 Umestitev in struktura sporočilnega nivoja 23

5.1 Moduli za povezovanje in kontrolo energije 44
5.2 Parametri modula za kontrolo energije v HTML obliki 46

Tabele

2.1 Format sporočila 6
2.2 Tipi argumentov 11
2.3 Format ID-ja 11

4.1 Vrstni red operacij 29

5.1 Opisi sporočil modula za kontrolo energije 45
5.2 Poraba pomnilnika in časi izvajanja 47