UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Davor Ocelić

Uporabniški vmesnik za mikroskop s-SNOM

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

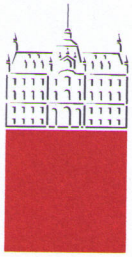
Mentor: doc. dr. Patricio Bulić

Ljubljana, 2009

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: 00435/2009
Datum: 05.04.2009

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

Kandidat: **DAVOR OCELIĆ**

Naslov: **UPORABNIŠKI VMESNIK ZA MIKROSKOP S-SNOM**
A USER INTERFACE FOR A S-SNOM MICROSCOPE

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

Izdelajte grafični uporabniški vmesnik, ki bo omogočal krmiljenje s-SNOM mikroskopa. Vmesnik naj se na kontrolni strežnik mikroskopa poveže po protokolu TCP/IP. Omogočen naj bo priklop več oddaljenih uporabnikov hkrati. Za implementacijo vmesnika uporabite programski jezik Ruby in grafično knjižnico Qt.

Mentor:
doc. dr. Patricio Bulić

Dekan:
prof. dr. Franc Solina

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Davor Ocelić,

z vpisno številko 63000357,

sem avtor diplomskega dela z naslovom:

"Uporabniški vmesnik za mikroskop s-SNOM"

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  doc. dr. Patricia Bulića
- 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 30.09.2009 Podpis avtorja:

Zahvala

Zahvaljujem se celotnem osebju Fakultete za računalništvo in informatiko za podporo in korekten odnos do vseh študentov ter posebno doc. dr. Patriciu Buliću za mentorstvo pri izdelavi diplomske naloge.

Zahvaljujem se svojim staršem za potrpežljivost ter moralno in materialno podporo tekom študija.

Kazalo

Seznam uporabljenih kratic in simbolov

AD Analogno-digitalni

API Programski vmesnik (angl. application programming interface)

DA Digitalno-analogni

FT Fourierova transformacija (angl. Fourier transform)

GPL Licenca GNU/GPL (GNU General Public License)

PC Osebni računalnik (angl. personal computer)

RAD Hitri razvoj programov (angl. rapid application development)

RPC Klic oddaljenih funkcij (angl. remote procedure call)

s-SNOM Blizkopoljna optična mikroskopija (angl. scattering-type scanning near-field optical microscopy)

TCP/IP Protokol Interneta (angl. transmission control protocol / Internet protocol)

Povzetek

Diplomska naloga obsega razvoj praktičnega uporabniškega vmesnika za specializirano digitalno napravo, mikroskop s-SNOM, v programskem jeziku Ruby in grafični knjižnici Qt.

Cilj naloge je pokazati, kako je s pravilno izbranimi tehnologijami mogoče rešiti tipične programerske izzive, uresničiti zahtevo po veliki hitrosti izvajanja in vizualni prijaznosti do uporabnika ter celotno nalogo končati v času, predvidenem za opravljanje diplomskega dela.

Razlaga problemov in priložena izvirna koda bralcu lahko služita kot napredni uvod v jezik Ruby in programiranje v grafični knjižnici Ruby/Qt ter kot splošna receptura pri izdelavi uporabniških vmesnikov in pri reševanju programskih problemov.

Ključne besede:
Ruby, Qt, uporabniški vmesnik, digitalna naprava, mikroskop, komunikacijski protokol

Abstract

The thesis describes implementation of a practical user interface for a specialized digital device, an s-SNOM microscope, using the Ruby programming language and the Qt graphics library.

The aim is to show how correctly chosen technologies can solve typical programming challenges, achieve good execution speed, satisfy modern GUI requirements and produce a usable interface within time intended for students' diploma thesis.

Problem discussion and the program source code may serve as an advanced introduction to programming in the Ruby/Qt language and as a general help in design and development of user interfaces.

Key words:

Ruby, Qt, user interface, digital device, microscope, communication protocol

Poglavje 1

Uvod

1.1 Predstavitev

Ljudje smo danes obkroženi s številnimi naravnimi in umetnimi objekti, ki kažejo zanimive strukturne vsebine v nanometerski skali oziroma v področju 1nm do 100nm. Primeri vključujejo nanokompozitne materiale, elektronske komponente in tudi biološke oblike, kot so molekule, proteinske verige in virusi.

Zaradi inherentne zgornje meje ločljivosti optičnih mikroskopov, ki izhaja iz valovne narave svetlobe, takšnih nanostruktur ni mogoče opazovati na navaden optični način. Difrakcijska limita (angl. diffraction barrier) namreč omejuje najmanjšo ločljivo razdaljo med točkama na okrog 1/2 valovne dolžine svetlobe, oziroma 200nm.

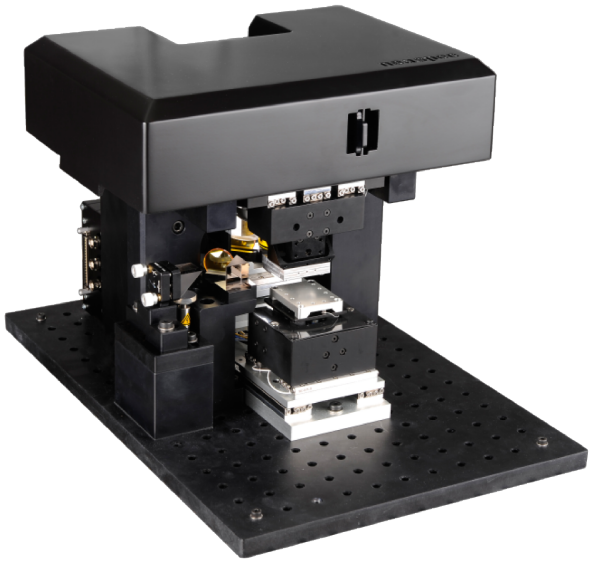
Zahteve po večji ločljivosti so usmerjale razvoj različnih drugih tehnik, med katerimi ene danes lahko dosežejo in tudi presežejo atomsko ločljivost.

Mikroskopija s-SNOM[1] je ena od novih, nedestruktivnih metod, ki se uspešno izogiba difrakcijskemu limitu s pomočjo ostre sonde neposredno nad opazovanim vzorcem. Sonda je osvetljena z laserskim žarkom, a svetloba, ki se razprši okrog sonde, je opazovana z ustreznimi detektorji. Amplituda in faza razpršene svetlobe sta odvisni od blizkopoljne (angl. near-field) interakcije med sondo in vzorcem. Pri tem je največja resolucija določena samo z ostrino sonde in ni odvisna od valovne dolžine svetlobe.

Cilj diplomske naloge je bil izdelati uporaben uporabniški vmesnik za specializirano napravo, mikroskop s-SNOM, in pokazati, kako je s pravilno izbranimi tehnologijami mogoče rešiti tipične programerske izzive, uresničiti zahtevo po veliki hitrosti izvajanja in vizualni prijaznosti do uporabnika, ter celotno nalogo končati v času, predvidenem za opravljanje diplomskega dela.

1.2 Mikroskop s-SNOM

Izbrani model mikroskopa s-SNOM je bil NeaSNOM, proizvod nemškega podjetja Neaspec GmbH.



Mikroskop NeaSNOM, dimenzij 45cm x 30cm x 29cm, vsebuje 11 motorjev, 4 leče in apokromatični objektiv. V ozadju se nahajajo: 2 digitalna in 3 analogna vhoda, 2 digitalna in 5 analognih izhodov ter 4 multipolna konektorja za vhodno/izhodne kartice na strani krmilnika.

Digitalne povezave so namenjene sinhronizaciji, analogne pa interoperabilnosti z drugimi mernimi instrumenti.

Za popolno delovanje so mikroskopu potrebni še:

- Laboratorijska merilna oprema (stabilizirajoča miza, laser s pripadajočimi lečami, detektor)
- Kontrolni strežnik
- Uporabniški vmesnik (TCP/IP in RPC zasnovana programska rešitev)

1.2.1 Kontrolni strežnik

Del celotne opreme mikroskopa podjetja Neaspec GmbH je kontrolni strežnik, ki ima dve nalogi - krmiliti mikroskop in omogočiti uporabnikom dostop do

podatkov po protokolu TCP/IP.

V času delovanja, strežnik podatke pobira z vodila, s pomočjo AD konverterjev pretvarja v digitalno obliko, obdeluje in shranjuje v glavnem pomnilniku. Med tem se izvede tudi osnovna dodatna obdelava podatkov, kot je Fourierova transformacija. Zaradi hitrosti in lažjega zagotavljanja brezhibnega delovanja, je količina operacij na strežniku zmanjšana na minimum ter je pričakovano, da bo uporabniški vmesnik pred prikazom podatkov opravil vse ostale potrebne obdelave in računanja.

Kontrolni strežnik ima dve omrežni povezavi. Ena povezava je prilagojena za direktno povezavo med strežnikom in uporabnikovo delovno postajo, druga pa za priključitev strežnika in več delovnih postaj v neko skupno omrežje.

Programski dostop se izvede s pomočjo standardnega omrežnega protokola TCP/IP, kar v primeru skupnega omrežja omogoča priklop več oddaljenih uporabnikov na strežnik, tudi istočasno. Zaradi narave samega mikroskopa in fizičnih omejitev, mikroskop ne omogoča dobesednega večuporabniškega dostopa - opravlja se lahko samo ena funkcija istočasno. Večuporabniški dostop je torej mišljen kot primaren enega uporabnika z več lokacij (npr. iz laboratorija in kabineta) in ne kot več neodvisnih uporabnikov.

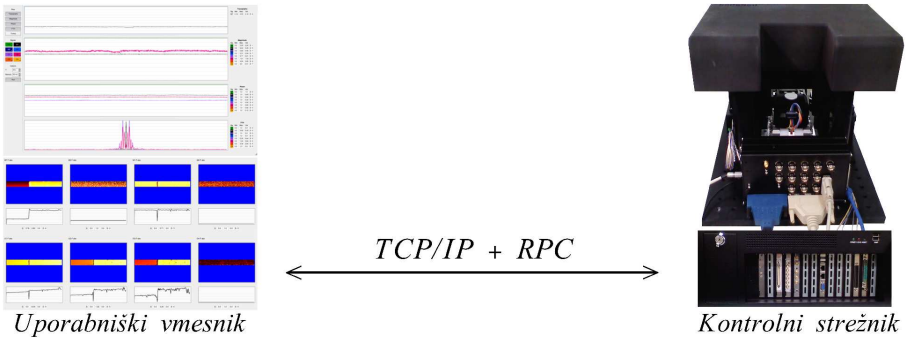
Dostop je iz fizičnih varnosnih razlogov zaščiten tudi s kratkim geslom. Če je klijent v času pristopa edini uporabnik, kontrolni strežnik dovoli dostop in s slučajno izbiro kreira kratko geslo. Vsi klijenti, ki sledijo prvemu, nujno rabijo pravilno geslo za uspešno povezavo. Ko vsi uporabniki končajo svoje delo, prvi novi uporabnik spet dobi novo, slučajno izbrano geslo.

Strežnik za uporabnika konstantno generira 9 kompleksnih vrednosti vsakih 6.5ms oz. 22kB/s in okvirno še dodatnih 15kB/s v času skeniranja vzorca.

Strežnik implementira vmesnik API, ki uporabniškim programom omogoča spremembo parametrov, izvajanje skeniranja in zbiranje dobljenih rezultatov. Uporabnik do podatkov lahko pride s pomočjo oddaljenega, poljubnega računalniškega programa na osnovi TCP/IP in RPC protokola za izmenjavo podatkov.

1.3 Uporabniški vmesnik

Kontrolni strežnik mikroskopa ponuja standardni TCP/IP- in RPC-zasnovani programski vmesnik API z dostopno dokumentacijo, kar uporabnikom omogoča poljubno implementacijo grafičnih vmesnikov.



Poskušal sem izdelati praktičen uporabniški vmesnik, ki naj bi omogočil popolno uporabo mikroskopa - nadzor stanja, postavljanje in spremembo ključnih parametrov ter skeniranje vzorca.

1.3.1 Zahteve

Na področju splošnih, in posebej na področju specializiranih računalniških programov smo priča številnim različnim izvedbam, ki se med sabo, neodvisno od namena, močno razlikujejo po vseh karakteristikah, vključno z vizualno podobo in kvaliteto. Razlika je pogosto prisotna tudi v programski opremi istega proizvajalca in istega namena, za različne računalniške platforme (npr. Microsoft Windows in GNU/Linux).

Pri izdelavi diplomske naloge je bil poudarek pri izbiri vseh komponent vmesnika na enostavnosti in hitrosti programiranja (angl. rapid application development), veliki hitrosti izvajanja in uporabi dobro znane grafične knjižnice, ki naj bi ponujala moderno in "familiarno" vizualno podobo na vseh platformah.

1.3.2 Izbira programskega jezika

Pri izdelavi uporabniškega vmesnika je bila izbira programskega jezika določena z zahtevami po:

- Sintaksi in funkcijah visokega nivoja, ki naj bi omogočile optimalno, ciljno-orientirano programiranje, brez zadrževanja na zapleteni sintaksi, preveč nizkem nivoju implementiranih funkcij ali neustreznimi dodatnimi knjižnicami
- Dostopnosti ene od modernih in standardnih grafičnih knjižnic za implementacijo vseh uporabniško-vidljivih elementov
- Hitrosti izvajanja, ki naj bi bila čim bolj podobna hitrosti izvajanja v sistemskem jeziku C in tako omogočala dovolj hitro opravljanje vseh zahtev (omrežnega prenosa, obdelave podatkov in grafičnega prikaza) v približno realnem času
- Podprtosti na platformi GNU/Linux
- Kompatibilnosti s strežnikom NeaSNOM, v katerem je del uporabniške funkcionalnosti že implementiran v jeziku Ruby

1.3.3 Programski jezik Ruby

Pri navedenih pogojih je kot programski jezik bil izbran Ruby.

Ruby je dinamičen, objektno-orientiran skriptni jezik splošnega namena, s sintakso inspirirano jezikoma Perl in Smalltalk ter prvič objavljen na Japonskem leta 1995, z licenco GNU GPL.

Jezik podpira več programskih pristopov, vključno s funkcionalnim, objektnim, imperativnim in refleksivnim. Vsebuje tudi dinamične tipe (angl. dynamic type system) in avtomatsko upravljanje z memorijo. Na različne načine je torej podoben tudi jezikom Python, Lisp, Dylan ter CLU.[2]

Avtor jezika Ruby je poskušal ustvariti programski jezik, ki naj bi izpolnjeval zahteve uporabnikov in ne računalnikov ter ki bi sledil intuitivnemu načinu programerskega razmišljanja (angl. principle of least surprise)[3].

V tem smislu jezik vsebuje tudi nekoliko zanimivih, čeprav nenavadnih lastnosti. Na primer: vrednost 0 (nula) ima logično vrednost "da" (angl. true). Za logično vrednost "ne" (angl. false) je treba uporabiti vrednosti "false" ali "nil".

1.3.4 Grafična knjižnica Qt

Kot moderna grafična knjižnica na vrhuncu današnjega razvoja grafičnih toolkitov in s podporo za strojno pospešen grafični prikaz je bila izbrana knjižnica Qt podjetja Trolltech oziroma Nokia.

1.3.5 Povezava med programskim jezikom in grafično knjižnico

Ruby ne vsebuje direktne povezave med programskim jezikom in grafično knjižnico Qt. Zasebna skupina ljudi, organizirana v projektu KDE, spremlja razvoj Qt-ja, KDE-ja in povezanih komponent ter usklajuje najnovejše spremembe s paketom KDE Bindings, ki potem ponuja povezavo s širokim spektrom različnih programskih jezikov, tudi Rubyjem.

Oglejmo si primer enostavnega Ruby/Qt programa, ki inicializira aplikacijo Qt, kreira in prikaže eno okno:

```
require ‘‘Qt4’’

app= Qt::Application.new ARGV.count, ARGV

window= Qt::MainWindow.new
window.show

button= Qt::PushButton.new "Button"
window.setCentralWidget button

app.exec
```

Pri programiranju v qtruby kot programsko dokumentacijo lahko uporabljamo standardna navodila na web strani knjižnice Qt, pripravljena za jezik C[4]. Razlike obstajajo samo v osnovni sintaksi jezika in inicializaciji spremenljivk. Če se programer teh osnovnih razlik zaveda, lahko vsa ostala navodila direktno prenese na delo v Ruby/Qt.

Poglavje 2

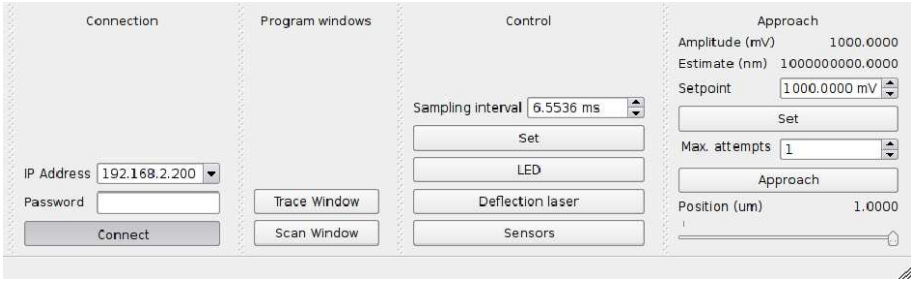
Vizualni prikaz uporabniškega vmesnika

Zaradi lažjega razumevanja celotnega diplomskega dela in kasneje opisane izvedbe bo najprej predstavljen uporabniški vmesnik v svoji končni grafični obliki z glavnimi deli in v praktičnem primeru uporabe.

Celoten postopek izvajanja skeniranja in prikaza dobljenih rezultatov je sestavljen iz naslednjih korakov:

1. Zagona kontrolnega strežnika, ki avtomatsko inicializira mikroskop s-SNOM
2. Zagona kontrolnega okna uporabniškega vmesnika
3. Vzpostavljanja povezave s strežnikom
4. Nameščanja stanja LED diode, referenčnega laserskega žarka in senzorjev pozicije motorjev
5. Odpiranja okna "Trace" in vizualne analize signalov
6. Grobega približevanja sonde vzorcu, s pomočjo signalov v oknu "Trace"
7. Odpiranja okna "Scan"
8. Nameščanja parametrov skeniranja v oknu "Scan"
9. Izvajanja skeniranja in analize rezultatov v oknu "Scan" ali zunanjih programih

2.1 Kontrolno okno

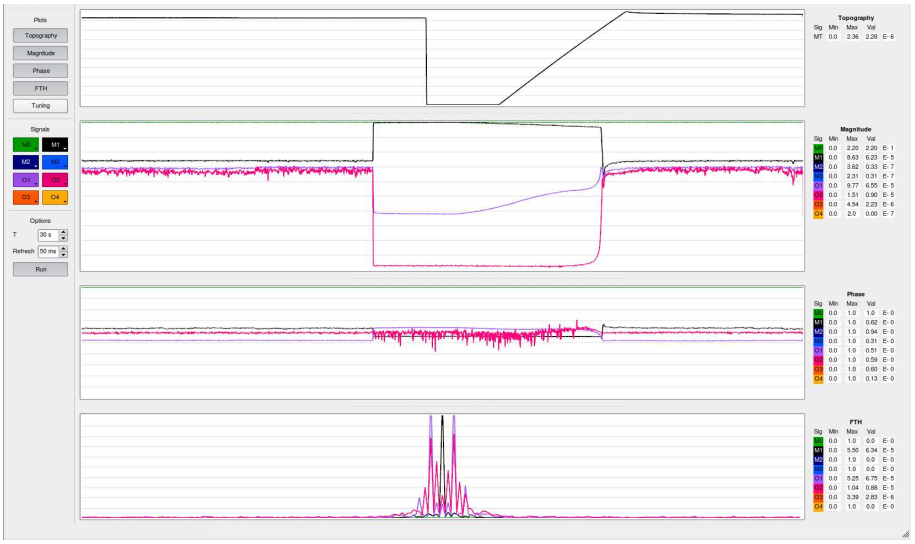


Kontrolno okno razdeljuje opcije na 4 glavne kategorije:

1. Povezavo s kontrolnim strežnikom (naslov IP s pripadajočim geslom)
2. Odpiranje dodatnih oken (za nadzor signalov in izvajanje skeniranja)
3. Kontrolo glavnih elementov mikroskopa (interval vzorčenja, stanje LED osvetlitve, referenčnega laserskega žarka in senzorjev motorjev)
4. Grobo približevanje sonde vzorcu

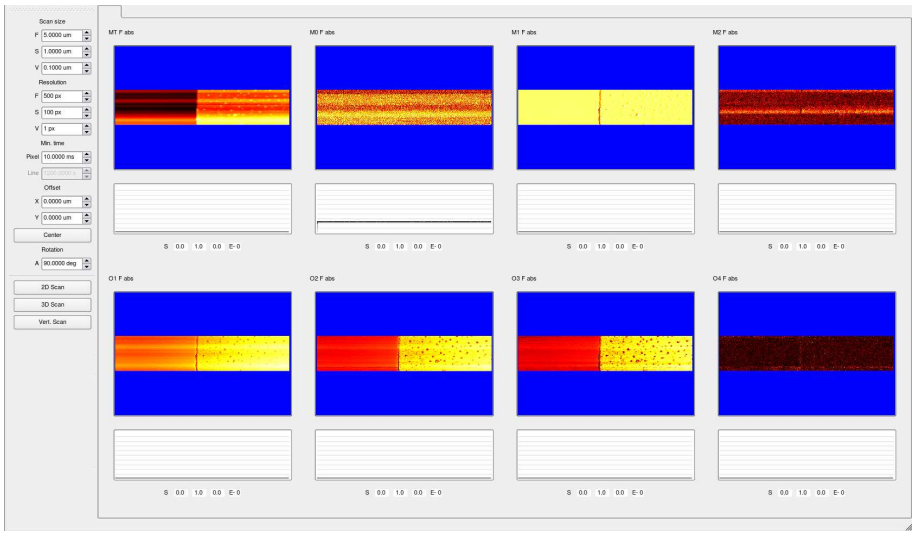
2.2 Pregled signalov mikroskopa

Pregled vhodnih signalov mikroskopa opravimo v oknu “Trace”.



2.3 Izvajanje skeniranja

Skeniranje vzorca opravimo v oknu “Scan”.



Poglavje 3

Izvedba

V tem poglavju bomo opisali najbolj zanimive probleme, ki smo jih srečali in so bili rešeni v procesu izdelave diplomske naloge, to je grafičnega uporabniškega vmesnika za specializirano digitalno napravo, mikroskop s-SNOM. Seznam bralec lahko direktno izkoristi pri izdelavi lastnih programov v jeziku Ruby/Qt ali pri izdelavi podobnih programov za druge specializirane naprave.

3.1 Programski dostop do kontrolnega strežnika

Za pričetek dela je najprej treba vzpostaviti povezavo s kontrolnim strežnikom. Vmesnik na strani strežnika omogoča programski dostop do krmilnih funkcij in podatkov mikroskopa po protokolu TCP/IP + RPC.

3.1.1 Rešitev

Za osnovno implementacijo mehanizma RPC lahko uporabimo že pripravljen Ruby razred RO4R (angl. Remote Objects For Ruby), del programskega paketa mikroskopa s-SNOM.

Implementiramo dodaten razred, ki izvede funkciji za avtomatski prehod uporabniškega vmesnika iz stanja "neaktivno" (angl. offline) v "aktivno" (angl. online) in obratno.

Funkcija online navadno poskrbi za prehod v novo stanje in izvede vse potrebne spremembe v izgledu uporabniškega vmesnika. V času prve spremembe stanja oz. v času zagona programa okna uporabniškega vmesnika še niso kreirana in je celotne funkcionalnosti potrebno doseči v dveh ločenih korakih.

3.1.2 Primer uporabe

```
require RO4R::Connection
require Connection

# Korak 1: kreiranje povezave (argument ‘‘false’’ prepreči
# avtomatski klic funkcije go_state)
status= Connection.online $opts['autoconnect'], false

# Kreiranje oken
restore

# Korak 2: vsklajevanje oken s stanjem povezave
Connection.go_state status
```

V ozadju, funkcija za nas pripravi Ruby objekt “root”, oz. spremenljivko $c.r, ki enostavnim klicem metod nad objektom omogoča popolnoma transparenten klic oddaljenih funkcij (angl. RPC).

3.2 Pošiljanje ukazov

Pri pošiljanju ukazov strežniku želimo biti obveščeni o povratni vrednosti funkcije ali o napaki pri izvajanju.

Če napake ni, strežnik pošlje povratno vrednost. Če do napake pride, strežnik pošlje objekt razreda Exception.

3.2.1 Rešitev

Za efikasnost pošiljanja ukazov kreiramo splošno funkcijo safe_send, ki enostavno pokliče željeno metodo na oddaljenem objektu. Prava vrednost te dodatne funkcije se pokaže v primeru napake - v tem slučaju funkcija avtomatsko izvede dani blok za reševanje napak. Teoretično, vsak klic zahteva posebno kodo za reševanje specifičnih napak. V praksi se vendar izkaže, da skoraj vedno zadostujeta samo dve funkciji - “soft” ki sporočilo o napaki izpiše v vidnem oknu in nadaljuje izvajanje, in “hard”, ki v primeru kritičnih napak prekine izvajanje celotnega programa.

3.2.2 Povzetek kode

```
def safe_send *arg
```

```
  ret= nil

  begin
    ret= $c.r.__send__ *arg
    rescue Exception => e
      ret= yield e
  end

  ret
end
```

3.2.3 Primer uporabe

```
$c.safe_send 'test_deflection_laser', nil, true, &@error_soft
```

3.3 Branje podatkov

Ko je kanal za pošiljanje ukazov pripravljen, kot je opisano v poglavju 3.2, isti mehanizem lahko uporabimo za branje podatkov. Operacije branja so ukazi brez efekta spremembe s strani strežnika.

3.3.1 Primer uporabe

Primer branja trenutnega stanja skanerja mikroskopa in zadnje preskenirane točke vzorca:

```
@direction, @spos, @fpos, spt, @last_pt, nstr=
  $c.safe_send 'scan_points',
    [ @direction, @spos, @fpos, @last_pt], &@error_soft
```

3.4 Periodične vrednosti

Za pravilno delovanje vmesnika in prikaz vrednosti je v določenih primerih funkcij potrebno klicati periodično.

3.4.1 Rešitev

Na strani vmesnika uporabimo navaden programski "tajmer", ki poskrbi za branje vrednosti in osveževanje polja vmesnika.

3.4.2 Povzetek kode

Primer inicializacije "tajmerja" na 100ms in periodičen klic funkcije getData(). Funkcija getData() prebere in prikaže trenutno izproženost piezo motorjev, amplitudo valovanja sonde ter heuristično oceno oddaljenosti sonde in vzorca.

```
@timer= Qt::Timer.new
@timer.setInterval 100

connect @timer, SIGNAL( 'timeout()'), self, SLOT( 'getData()')

def getData
  pos= $c.safe_send 'control_z', &@error_soft
  @feedback_position.setText( '%.4f' % pos)
  @feedback_position_slider.setValue pos

  pos= $c.safe_send 'control_reading', &@error_soft
  @approach_amplitude.setText( '%.4f' % ( pos * 1e3))

  pos= $c.safe_send 'control_probe_estimate', &@error_soft
  @approach_estimate.setText( '%.4f' % ( pos * 1e9))

  nil
end
```

3.5 Hitrost izvajanja

Pri izdelavi vmesnika so bile uporabljane verzije jezika Ruby 1.8 in Ruby 1.9. Po lastno opravljenih testih, prevajalnik v Ruby 1.9 ("YARV") kaže 400% do 2000% boljše performanse kot prevajalnik v Ruby 1.8 ("CRuby").

Vendar so v primerjavi z jezikom C praktično vsi skriptni jeziki počasni. Glede na zahtevano hitrost omrežnega prenosa, količino vhodnih podatkov, število dodatnih operacij, ki jih je treba opraviti, in kompleksnost prikazovanja rezultatov, dovolj hitrega uporabniškega vmesnika ne bi bilo mogoče implementirati v jeziku Ruby brez uporabe bolj hitrih funkcij.

Trije glavni problemi jezika so iteracije po zankah, matematične operacije nad polji podatkov in grafično prikazovanje v realnem času.

3.5.1 Rešitev

Hitrost izvajanja lahko izboljšamo na način, da kritičnih delov kode ne implementiramo s pomočjo jezika Ruby, temveč s pomočjo funkcij iz C knjižnic (angl. library).

Na ta način potencialno kompleksne in počasne operacije opravimo s hitrostjo kode C. Edini del posla, ki ga opravi Ruby oz. qtruby, je branje Ruby programske kode, izbira točne funkcije v knjižnici C in končno klic izbrane funkcije.

Na primer: matematične operacije nad podatki se lahko izvedejo s pomočjo numerične knjižnice NArray, grafične operacije pa s pomočjo številnih strojno-pospešenih funkcij v knjižnici Qt. Pri tem je zelo pomembno, da večina funkcij že podpira obdelovanje podatkov v blokih, kar omogoča kompletno izogibanje izrecnim zančnim operacijam.

Omenimo, da qtruby implementira relativno zapleten mehanizem izbire in klica funkcij v nizkonivojski knjižnici Qt. Vendar je implementacija funkcij Qt v jeziku C toliko hitrejša, da še vedno dosežemo velike hitrosti.

3.5.2 Primer uporabe

Jezik Ruby poskrbi, da za programerja ni nobene razlike v klicu funkcij, ne glede na njihov izvor oziroma implementacijo v podlagi. Primer klica funkcij 'fill' in 'setPixel', ki sta definirani v C knjižnicah NArray oz. Qt:

```
# Numerično polje @values velikosti 1000x1000
# izpolnimo s vrednostjo 0
@values= NArray.sfloat 1000, 1000
@values.fill! 0

# Kreiramo Qt razred Image in postavimo vrednost
# piksla (50, 50) na 0
@image_array= NArray.int 1000, 1000
@image= Qt::Image.new @image_array.buffer, 1000, 1000,
  Qt::Image::Format_ARGB32
@image.setPixel 50, 50, 0
```

3.6 Komandna linija

Uporabniški vmesnik rabi podporo za branje konfiguracijskih vrednosti (argumentov) direktno s komandne linije.

Poleg tega je za določene opcije potrebno imeti standardne vrednosti, ki se bodo upoštevale, če uporabnik ne navede drugače.

3.6.1 Rešitev

Problema sta povezana in se pogosto rešujejo kot celota. Rešitev lahko implementiramo z listo standardnih vrednosti, Ruby funkcijo za branje vrednosti s komandne linije in delom kode, ki bo poskrbel, da vse opcije na komandni liniji prevzamejo prioriteto nad standardnimi. Iz kode je razvidno, da opcije s prefiksom "no-" negirajo značenje in da je v določenih primerih omogočena tudi posebna obdelava opcij ("case opt" in prikazana opcija "state").

3.6.2 Povzetek kode

```
# Definicija opcij v programu
$default_opts= {
  'state'        => true,
  'state-load'   => true,
  'state-save'   => true,
  'opengl'       => true,
  'antialiasing' => true,
  'autoconnect'  => false,
}

# Opcije, ki jih želimo prebrati s komandne linije
opts= [
  [ '--state',          '--s',          GetoptLong::NO_ARGUMENT],
  [ '--opengl',         '--gl',         GetoptLong::NO_ARGUMENT],
  [ '--autoconnect',    '--connect',    GetoptLong::NO_ARGUMENT],

  # --no- variante opcij
  [ '--no-state',       '--no-s',       GetoptLong::NO_ARGUMENT],
  [ '--no-opengl',      '--no-gl',      GetoptLong::NO_ARGUMENT],
  [ '--no-autoconnect', '--no-connect', GetoptLong::NO_ARGUMENT],
]
```

```
# Osnovno procesiranje argumentov s komandne linije
args= GetoptLong.new *opts

# Obdelava in snemanje opcij v trenutno konfiguracijo
begin
  args.each do |opt, arg|
    opt= opt.sub /^\-+/, ''
    arg= true  if arg.length== 0
    arg= false if opt.sub! /^no-/, ''

    case opt
      when 'state'
        $opts['state-load']= $opts['state-save']= arg
      else
        $opts[opt]= arg
    end
  end
rescue GetoptLong::InvalidOption
  exit 1
end
```

3.7 Osnovne funkcije v jeziku Ruby

Ruby je objektno-orientiran programski jezik, ki vse funkcije definira kot metode nad razredom (angl. class) ali objektom (angl. object). V času izdelave uporabniškega vmesnika je postalo razvidno, da bo koristno izboljšati nekoliko osnovnih funkcij.

3.7.1 Rešitev

V jeziku Ruby funkcije spreminjamo in dodajamo na isti način. Ruby podpira tudi spreminjanje "standardnih" oziroma osnovnih funkcij, ki tvorijo del osnovnih Ruby knjižnic (angl. library).

Potrebno je samo podati ime razreda in funkcije, ki jo želimo dodati ali spremeniti.

Med ostalim bomo dodali funkcije za pretvorbo prvega znaka teksta v velike črke (podobno funkciji 'ucfirst' programskega jezika Perl), za pretvorbo

poljubnih vrednosti v obliko "da/ne" ter za enostavno izstavljanje elementov iz polja vrednosti (>>).

3.7.2 Povzetek kode

Datoteka extensions.rb:

```
class String

  # ucfirst
  def ucfirst
    self.dup.ucfirst!
  end

  def ucfirst!
    self[0,1]= self[0,1].upcase
    self
  end

  # Pretvorba v vrednost Da/Ne
  def to_bool
    return false if self == false || self =~ /^false$/i || self== ''
    return true  if self == true  || self =~ /^true$/i
    raise ArgumentError.new "Invalid value for Boolean: '#{self}'"
  end
end

# Okrajšave imen obstoječih funkcij
class Object
  alias_method :self, :instance_exec
  alias_method :kind, :class
  alias_method :here, :binding
  alias_method :seti, :instance_variable_set
  alias_method :geti, :instance_variable_get
end

# Izstavljanje elementov v polja s pomočjo operatorja >>
class Array
  def >> arg
```

```
    delete arg
    self
  end
end

# Debug funkcija - izpis lokacije in danih parametrov
def pfl *arg
  print caller[0], '-> ', arg.inspect[1..-2]
  puts
end
```

3.8 Bližnjice

Pri izdelavi uporabniškega vmesnika se je iskazalo, da so bližnjice oz. kratke kombinacije tipk, ki jih uporabnik pritisne za dostop do določenih funkcij, zelo uporabne. Grafični toolkit Qt že podpira implementacijo bližnjic, vendar je mehanizem splošen in ima nekoliko pomankljivosti:

1. Ustvarjanje posameznih akcij oz. bližnjic zahteva veliko pisanja, kar podaljšuje čas izvedbe, povečuje verjetnost programskih napak in otežuje vzdrževanje

2. Bližnjice se "izvajajo" v kontekstu definicije. Na primer, če bližnjico "Zapri (eno) okno" definiramo na nivoju aplikacije, bo ta dostopna povsod, vendar bo vedno zaprla eno samo okno, nad katerim je bila definirana. Če bližnjico kreiramo na nivoju posameznega okna, bo delovala v kontekstu želenega okna, vendar je ne bo mogoče avtomatsko izkoristiti pri drugem oknu

3. Qt "akcije" (osnovni gradniki bližnjic) podpirajo samo eno bližnjico po akciji (npr. funkcije zapiranja oken ni mogoče imeti dostopne na dve kombinaciji tipk, Ctrl+Q in F10)

3.8.1 Rešitev

Uveden je bil podrazred, ki na osnovi enolinijskega poziva omogoča kreiranje standardnih objektov, baziranih na razredu Qt::Action, in avtomatsko definira množico njihovih parametrov, kar skrajšuje količino potrebne kode pri vsaki akciji in omogoča hitro kreiranje akcij po preverjenem postopku.

Paralelno je bil definiran dodaten razred, ki definira standardne bližnjice in njihove pripadajoče parametre ter enolinijskim pozivom omogoča kreiranje bližnjic v kontekstu specifičnih objektov.

3.8.2 Povzetek kode

```
class NFCAction < Qt::Action

  def initialize text, parent, shortcut, tooltip, checkable = false,
   context = :win, statustip = tooltip, whatsthis = tooltip

    super        text, parent
    setText      tr text.ucfirst

    setShortcut  Qt::KeySequence.new(tr shortcut.to_s) if shortcut
    setStatusTip tr statustip
    setToolTip   tr tooltip
    setWhatsThis tr whatsthis

    setCheckable checkable
    setShortcutContext CONTEXT[context]
  end
end

module Actions

 ACTIONS= {
    :quit => ['Quit',  'self', 'Ctrl+Q', 'Exit application',false, :win],
    :close=> ['Close', 'self', 'Ctrl+W', 'Close window',    false, :win],
  }

  SIGNALS= {
    # Default= triggered()
  }

  OBJECTS= {
    # Default= 'self'
    :quit    => '$qApp',
  }
```

```
  CONNECTS= {
    # Default= 'close()'
    :quit     => 'closeAllWindows()',
  }

  def installAction name, slot= nil, obj= nil, sig= nil

   action= NFCAction.new tpl[0], obj || eval(tpl[1]), s, *tpl[3..-1]

    shortcuts.each do |s|
      connect( action,
               SIGNAL(sig || SIGNALS[name] || 'triggered()'),
               object,
               SLOT(slot) )

      addAction a
    end
  end

  def stockAction *arg
    arg.each {|a| installAction a }
  end
end
```

3.8.3 Primer uporabe

```
stockAction :quit, :close
```

3.9 Večuporabniški dostop, povratna povezava

V svoji osnovni obliki kontrolni strežnik mikroskopa že podpira več istočasnih povezav in njihovo medsebojno obveščanje o spremembi stanja mikroskopa. Ampak ta funkcionalnost še vedno zahteva posebno podporo na strani vmesnika in je za pravilno delovanje ni mogoče zanemariti. Na primer, če uporabnik na eni lokaciji aktivira LED osvetlitev opazovanega vzorca, se ta sprememba pri njem izkaže kot gumb "LED" v stanju "aktiven". Vprašanje je, kaj se zgodi z gumbom "LED" pri ostalih uporabnikih.

3.9.1 Rešitev

Kontrolni strežnik vsem priključenim uporabnikom pošlje sporočilo RPC o spremembi stanja. Vendar je na strani vmesnika treba ustvariti mehanizem, ki ta obvestila sprejema in usklajuje lokalno stanje. Po dokumentaciji kontrolnega strežnika za vsako uspešno opravljeno zahtevo, kjer je povratno obvestilo zaželjeno, strežnik na klientu pokliče komando RPC z imenom "<zahteva>_callback".

V osnovni obliki bi lahko enostavno v vmesniku definirali funkcijo "<zahteva>_callback". Vendar bi se ta izvajala v drugi niti (angl. thread) programa, kar v kombinaciji programskega jezika Ruby in grafične knjižnice Qt ni zaželjeno.

Implementiramo torej posebno funkcijo method_missing, ki ima v jeziku Ruby poseben pomen in se pokliče, ko neka poklicana metoda ne obstaja. Znotraj funkcije ime in argumente željene funkcije dodamo v določeno vrsto (angl. queue). Osnovna nit (angl. thread) programa gre po tem periodično skozi vrsto in kliče navedene funkcije.

3.9.2 Povzetek kode

```
Control.self do

  def self.__respond_to? name, priv
    if /_callback$/=~ name.to_s
      return true
    else
      respond_to? name, priv
    end
  end

  def self.method_missing name, *arg
    unless /_callback$/=~ name.to_s
      raise NoMethodError, "No method #{name}"
    end

    $callback_queue.push [ name.to_s+ '_handler', *arg]
    nil # Important
  end

  # Implementacija test funkcije
```

```
  def self.control_led_callback_handler v
    $qApp.windows_hash['test'].wled.defaultAction.callback v \
      if $qApp.windows_hash['test']

    $qApp.windows_hash['control'].led_button.defaultAction.callback v \
      if $qApp.windows_hash['control']
  end
end
```

3.10 Kreiranje elementov v blokih

Pri izdelavi vmesnika se izkaže, da je v številnih primerih treba kreirati niz podobnih elementov, elementov istega tipa, oblike in funkcionalnosti, ki se med sabo razlikujejo samo po znanih in določljivih, tehnično nepomembnih detajlih. Na primer, gumbi za aktivacijo LED osvetlitve, laserskega žarka in senzorjev imajo različen naziv, opis, bližnjico in finalni efekt, ampak so tehnično med sabo zelo podobni. Koristno je bilo osmisliti mehanizem, ki bo omogočal kreiranje takih elementov v zanki, z minimizacijo ponavljajočih blokov.

3.10.1 Rešitev

Določeno zahtevo lahko dosežemo z uporabo standardnih funkcij jezika Ruby.

3.10.2 Povzetek kode

Primer kreiranja dveh gumbov v zanki. Začetne spremenljivke definirajo vse gumbe specifične elemente, univerzalni del kode potem kreira gumbe. Podani primer s samo dvema gumboma deluje nepotrebno zapleten. Vendar se koristnost takega pristopa še kako izkaže, ko v bloku kreiramo 4, 8 ali več elementov.

```
titles=     %w/led deflection_laser/
longtitles= [ "LED", "deflection laser"]
tooltips=   [ 'Toggle LED', 'Toggle deflection laser']
shortcuts=  %w/l d/
radio=      [ true, true]
actions=    [
  Proc.new {|b|
```

```
    $c.safe_send 'control_led', b, &@error_soft
  },
  Proc.new {|b|
    $c.safe_send 'control_deflection_laser', b, &@error_soft
    @find_button.setEnabled b
  },
]

titles.zip( longtitles, tooltips, shortcuts, radio, actions).each do
 |title, longtitle, tip, shortcut, checkable, action|

  w= frame.prepareWidget( longtitle.ucfirst, frame, shortcut, tip, checkable, &
  layout.addWidget w, -1, 0, 1, 2

  seti "@#{title}_button", w
  self.class.self { attr_reader "#{title}_button".to_sym }

  handleState "#{title}_button" if checkable
end
```

3.11 Podrazredi razredov Qt

Pri izdelavi uporabniškega vmesnika želimo čimveč obstoječih funkcij uporabljati neposredno iz implementacije C knjižnice Qt. Ampak to ni vedno mogoče in je za potrebe vmesnika določene razrede potrebno dopolniti s specifičnimi dodatnimi funkcijami v jeziku Ruby.

3.11.1 Rešitev

Objektno-orientiran jezik Ruby pozna relacije med razredi in mehanizme dedovanja. Če torej želimo spremeniti obnašanje razreda Qt::Label, ga lahko enostavno spremenimo na način, da redefiniramo obstoječe ali definiramo nove funkcije:

```
class Qt::Label
  def initialize *arg

  # Standardno izvajanje
  super
```

```
  # Naši dodatki
  puts ‘‘Kreiran nov objekt Label’’

  end
end
```

V večini primerov je še bolj zanimivo kreirati nov razred, na podlagi obstoječega, in tako imeti na voljo obe funkcionalnosti:

```
class MyLabel < Qt::Label
  def initialize *arg

  # standardno izvajanje
  super

  # Naši dodatki
  puts ‘‘Kreiran nov objekt MyLabel’’

  end
end
```

3.12 Odpiranje oken

Pri načrtovanju vmesnika je postalo jasno, da bo uporabniški del sestavljen iz več oken. Iz tega izhaja, da je v smislu optimalne programske rešitve treba izdelati "recept" za enostavno kreiranje novih oken, ki že vsebujejo standardno funkcionalnost (npr. snemanje stanja, naziv, bližnjice itn.).

3.12.1 Primer uporabe

Če želimo kreirati novo okno "Test", je dovolj kreirati datoteko TestWindow.rb z naslednjo vsebino:

```
class TestWindow < Qt::MainWindow

  def initialize *arg
    super init
```

```
      setWindowTitle tr 'Test Window'

      stockAction :quit, :close, :shell, :version, :pwdisplay

      statusBar.show
    end
  end
```

3.13 Evaluator

Izkaže se, da je za potrebe testiranja, iskanja napak in tudi preseganja limita grafičnih aplikacij zelo uporabno, če ima aplikacija vgrajen "evaluator" oziroma komandno linijo, s katero lahko dostopamo do programskega jezika in vseh elementov odprte aplikacije (vključno s celotno memorijsko strukturo in grafičnimi elementi).

Dobro bi bilo, če bi evaluator:

1. Preusmerjal izhoda STDOUT in STDERR v naše okno in ne na terminal
2. Podpiral izvajanje v trenutni ali novi programski niti
3. Enumeriral izhodne linije
4. Omogočal kontekstualno dopolnjevanje vpisanega teksta

3.13.1 Rešitev

Rešitev lahko implementiramo z enostavnim dodatnim oknom, ki ima dve glavni polji - za vnos in izpis podatkov - in ki implementira navedene zahteve.

3.13.2 Povzetek kode

Zaradi kompleksnosti kode je težko podati smiseln povzetek. Bralec naj si ogleda datoteki Evaluator.rb in RubyCompleter.rb.

3.13.3 Primer uporabe

```
[  1]> print "test"
test
```

```
[  2]> p $qApp
[  2]= #<vmesnik::vmesnikApplication:0x00000001af6aa8>
```

3.14 Instalacija programa

Uporabniški vmesnik lahko zaženemo brez posebne instalacijske procedure. Vendar je uradna instalacija programa pogosto zahtevana.

3.14.1 Rešitev

Instalacijo pripravimo z uporabo obstoječega Ruby mehanizma extconf.rb.

3.14.2 Povzetek kode

Datoteka extconf.rb:

```
require "mkmf"

$INSTALLFILES = [
  ['vmscan',   '$(bindir)'],
  ['*.rb',     '$(rubylibdir)/vmesnik'],
  ['images/*', '$(datadir)/vmesnik']
]

create_makefile('vmesnik')
```

3.14.3 Primer uporabe

```
ruby extconf.rb
make
make install
```

3.15 Popravki v paketu KDE-Bindings

Vzdrževalci paketa KDE-Bindings posvečajo veliko časa usklajevanju z ostalimi komponentami, vendar jim ne uspe vedno poskrbeti za vse različne verzije in kombinacije komponent. Kombinacija Ruby/Qt je popolnoma uporabna

in presenetljivo dobro vzdrževana, vendar relativno nova in neznana ter se je v času izdelave uporabniškega vmesnika izkazalo, da vsebuje nekoliko pomankljivosti, ki jih je bilo treba odpraviti. Oglejmo si listo najbolj zanimivih problemov, ki sem jih odpravil in ki so danes uradni del paketa KDE-Bindings:

- Kompletna podpora za verzijo Ruby 1.9 (podprta je bila samo verzija Ruby 1.8)
- Popravki pri razčlenjevalniku Smoke (podpora za pretvorbo različnih variant tipa podatkov "char*" med jezikom Ruby in knjižnico Qt)
- Izboljšava mehanizma izbire funkcij v knjižnici Qt (pri tipu podatkov "char*" in "bits")
- Popravki v algoritmu za upravljanje pomnilnikom (odpravljena segmentacijska napaka)

Poglavje 4

Zaključek

V diplomskem delu sem izdelal in opisal uporabno aplikacijo, uporabniški vmesnik, ki omogoča opravljanje vseh korakov pri skeniranju vzorca na mikroskopu s-SNOM: nameščanje osnovnih parametrov v oknu "Control", pregled vhodnih signalov v oknu "Trace" ter skeniranje vzorca v oknu "Scan".

Dobljene rezultate (slike) je potem mogoče analizirati s pomočjo drugih primernih aplikacij za analizo slik (na primer Gwyddion).

Končna hitrost izvajanja je dovolj dobra, da uporabnik sprejme, obdela in prikaže spremembe v vseh oknih s periodo 50ms - 100 ms.

V času izdelave vmesnika sem ugotovil tudi nekoliko dodatkov in popravkov, ki bi jih bilo smiselno še opraviti, ki pa zaradi kompleksnosti ali relativne nepomembnosti niso bili nujni za končanje dela:

- Kontrolni strežnik mikroskopa podpira nekoliko različnih načinov dela in množico dodatnih parametrov, za katere smo v uporabniškem vmesniku enostavno uporabili standardne vrednosti. Za popolnoma funkcionalen vmesnik bi bilo treba za vse parametre omogočiti prikaz in izbiro

- Ob zagonu drugih, nevezanih aplikacij na računalniku z uporabniškim vmesnikom, vmesnik včasih prekine izvajanje s segmentacijsko napako (angl. segmentation fault). Pogostost napake je odvisna od specifične aplikacije, oziroma od tipa dostopa, ki ga druga aplikacija opravlja na procesorju in datotečnem sistemu. Vemo, da v okolju GNU/Linux procesi nimajo neposrednega dostopa do tujih memorijskih struktur (segmentov) ter da pri skriptnih jezikih segmentacijska napaka vedno pomeni problem v nižjem nivoju, zunaj aplikacije same. Na osnovi izkušenj predpostavimo tudi, da vzroka napake ni v samem jeziku ali grafični knjižnici. Napako bi torej bilo smiselno poiskati v paketu KDE-bindings,

posebej na mestih, kjer KDE-Bindings opravlja prehod iz izvajanja v Ruby-ju na izvajanje v jeziku C in obratno ter kjer izvajanje v nitih (angl. thread execution) ni onemogočeno ali pravilno izvedeno

- Skriptni programski jezik Ruby, grafična knjižnica Qt in paket KDE-Bindings (ter na ta način tudi uporabniški vmesnik) delujejo na več operacijskih sistemih, vključno z Microsoft Windowsom. Zanimivo bi bilo komponente in celotno aplikacijo testirati tudi v tem okolju

Upam, da smo v diplomskem delu na konkretnem primeru praktičnega uporabniškega vmesnika uspeli pokazati glavne prednosti kombinacije jezika Ruby in grafične knjižnice Qt ter bralca zainteresirati za samostojno dodatno učenje in napredovanje.

Literatura

[1] N. Ocelić, "Quantitative near-field phonon-polariton spectroscopy," Ph.D. dissertation, Technical University Munich, 2007.

[2] Wikipedia, "The Ruby Programming Language."

[3] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby 1.9: The Pragmatic Programmers' Guide*, 3rd ed. Pragmatic Bookshelf, 28 Apr. 2009.

[4] Nokia Corporation, "Qt 4.5 Reference Documentation," Internet publication, 2009.