TAMPEREEN AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

Tutkintotyö

Esa Kulmala

LUA-OHJELMOINTIKIELEN SIIRTÄMINEN SYMBIAN-KÄYTTÖJÄRJESTELMÄÄN

Työn ohjaaja Jari Mikkolainen
Työn teettäjä Elektrobit Wireless Communications Oy, valvojana Vespe Savikko
Tampere 2007

TAMPEREEN AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka
Esa Kulmala
Lua-ohjelmointikielen siirtäminen Symbian käyttöjärjestelmään
Tutkintotyö 55 sivua
Työn ohjaaja Jari Mikkolainen
Työn teettäjä Elektrobit Wireless Communications Oy, valvojana Vespe Savikko
kesäkuu 2007
Hakusanat Lua, Symbian, ohjelmointi, C++

TIIVISTELMÄ

Lua on dynaamisesti tulkattava ohjelmointikieli, joka on suunniteltu laajentamaan ohjelmistoja. Luan on toteuttanut PUC-Rion (Pontifical Catholic University of Rio de Janeiro) yliopisto Brasiliassa. Lua on ns. vapaa ohjelmisto, eli sitä voidaan käyttää niin akateemisiin kuin kaupallisiinkin tarkoituksiin ilman käyttömaksuja.

Symbian on käyttöjärjestelmä, joka on laajasti käytössä uusimman sukupolven matkapuhelimissa. Symbian on suunniteltu toimimaan laitteistoissa, joissa on käytettävissä rajallinen määrä muistia. Symbianin kehityksestä vastaa Symbian Ltd. Suurimpana Symbianin laitevalmistajana on Nokia.

Lua eroaa käännettävistä ohjelmointikielistä, kuten esimerkiksi C++:sta, siten, että lähdekoodia ei varsinaisesti käännetä laitteeseen ajettavaan binäärimuotoon, vaan lähdekoodi tulkataan dynaamisesti eli ajonaikaisesti laitteessa olevassa ns. virtuaalikoneessa. Siis periaatteessa tuotettu Lua-lähdekoodi on laitteistosta riippumatonta, kunhan Luan virtuaalikone on sovitettu haluttuun laitteistoon.

Tämä työ perustuu Luan virtuaalikoneen sovittamiseen Symbian-käyttöjärjestelmään. Tämä työ ei tarjoa täydellisesti toimivaa Luaa Symbianissa, vaan prototyypin, jolla voidaan ajaa Lua-koodia Symbianissa. Tässä työssä on myös muutama käyttötapausesimerkki, miten Lua-ohjelmia voidaan käyttää Symbianissa.

TAMPERE POLYTECHNIC
Computer System Engineering
Software Engineering
Esa Kulmala
Porting Lua-programming language to Symbian OS
Engineering Thesis 55 pages
Thesis Supervisor Jari Mikkolainen
Comissioning Company Elektrobit Wireless Communications Oy, Supervisor Vespe Savikko
June 2007
Keywords Lua, Symbian, porting, programming, C++

ABSTRACT

Lua is dynamically interpreteted programming language, which is designed to extend applications. Lua has been designed and implemented in Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in Brazil. Lua is so called free software, which means that it can be used both academically and commercially without any license costs.

Symbian OS is an operating system, which is widely used in modern mobile phones. Symbian OS designed to be used in embedded systems with limited amount of resources, for example in devices with low memory. Symbian Ltd. has designed Symbian OS. Nokia is the largest phone manufacturer, which uses Symbian OS.

Lua differs from compiled programming languages, such as C++, so that the source code is not compiled to binary form, but the source code is dynamically interpreted by Lua virtual machine on the device. So the Lua source code is basically OS independent, as long as the Luas virtual machine has been ported to the device.

This thesis is about how to port Lua's virtual machine to Symbian OS. This thesis does not offer fully working Lua in Symbian, but a prototype, which can be used to execute Lua scripts in the device.

KÄYTETYT LYHENTEET JA TERMIT

Aktiiviolio	Active object (Symbian OS); käytetään asynkronisen ohjelmoinnin yhteydessä.
Aktiivivuoroitin	Active scheduler (Symbian OS); käytetään asynkronisen ohjelmoinnin yhteydessä.
Bytecode	Tavukoodi; Lua-koodi käännetään tähän muotoon, jotta se voidaan suorittaa virtuaali koneen ytimessä
Debug	Virheen jäljittäminen
Korutiini	Coroutine; Luan tarjoama vaihtoehto moniajoon
Lua-laajennus	Dynaamisesti ladattava Lua-skripti, tai C-kirjasto
Prosessi	Symbian OS:ssä sisältää yhden tai useamman säikeen
Skripti	Korkean tason ohjelmointikielen lähdekooditiedosto
Säije	Symbian OS:ssä suorituksen yksikkö
Userdata	Luan käyttämä tyyppi, jolla välitetään C-kielessä luotua tietorakennetta
Ydin	Kernel; sovelluksen alin kerros

SISÄLLYSLUETTELO

1 JOHDANTO

Tämän työn tarkoituksena on selvittää, miten Lua-ohjelmointikieli voidaan siirtää Symbian-käyttöjärjestelmään. Tämän työn tarkoituksena ei ole tuottaa täydellisesti toimivaa Lua-käännöstä Symbian-käyttöjärjestelmässä, vaan selvittää, mitä ongelmia liittyy käännöksen tekemiseen ja tarjota niihin ratkaisu.

Lua-ohjelmointikieltä Symbianissa käytettäisiin muun muassa ohjelmistokomponenttien testaamiseen ja integrointiin. Luasta on tehty mm. kaksi julkista Symbian-toteutusta: LuaS60 /4/ ja Lua5 /5/, mutta ne ovat joko täysin kehitysasteella tai vanhentuneita, eivätkä tarjoa ratkaisua siihen, miten voidaan käyttää Symbianin erityispiirteitä, kuten esimerkiksi aktiiviolioita.

Tässä työssä pyrin selvittämään, miksi Luan kaltaisia erittäin korkean tason ohjelmointikieliä käytetään, mitä erityispiirteitä Lua tarjoaa, minkälainen on Luan arkkitehtuuri, miten se voidaan siirtää Symbianiin ja miten Symbianin erityisominaisuuksia voidaan käyttää Lua-koodissa.

Tämä työ on toteutettu käyttäen seuraavia työkaluja ja versioita.

Lua:

- Lua 5.1.1

Kehitysympäristö:

- Käyttöjärjestelmä: Windows XP, service pack 2
- SDK: S60 3rd Edition SDK for Symbian OS, Maintenance Release
- Kääntäjä (emulaattori): CodeWarrior Development Studio for Symbian OS v3.1
- Kääntäjä (laite): GCC CSL ARM toolchain

Kohdelaite:

- Käyttöjärjestelmä: Symbian 9.1
- Puhelinmalli: Nokia N71

2 LUA-OHJELMOINTIKIELEN ERITYISPIIRTEET

Lua on tehokas ja kevytrakenteinen ohjelmointikieli, joka on suunniteltu laajentamaan ohjelmistoja. Luan kehityksestä ja ylläpidosta vastaavat Roberto Ierusalimschy, Waldemar Celes ja Luiz Henrique de Figueiredo PUC-Rion yliopistolla Brasiliassa. Lua 1.0 valmistui heinäkuussa 1993, mutta sitä ei koskaan julkistettu. Vuoden kuluttua Lua 1.1 oli ensimmäinen julkinen versio Luasta ja siitä lähtien Luaa onkin jatkuvasti kehitetty ja uudelleen kirjoitettu monta kertaa /7,8/.

Luan toteutuksen lähtökohdiksi otettiin yksinkertaisuus, tehokkuus, siirrettävyys, ja mahdollisuus ajaa Luaa laitteissa, joissa on rajalliset resurssit käytettävissä /1/. Luan siirrettävyys on saavutettu käyttämällä ”puhdasta” ANSI C-lähdekoodia, eli Luan toteutus on pyritty tekemään mahdollisimman käyttöjärjestelmistä riippumattomaksi. Symbianiin on myös toteutettu ANSI C-kirjasto siirrettävyyttä helpottamaan, mutta se ei tue kaikkia ANSI C:n funktioita. Luan toteutuksen siirrettävyyttä erilaisiin ympäristöihin helpottaa ANSI C:n käyttö Luan toteutuksessa.

Lua on saavuttanut suuren suosion peliteollisuudessa. Luaa käytetään yleisesti peliteollisuudessa mm. pelin logiikan, tekoälyn ja käyttöliittymien mallintamiseen. Luaa on käytetty esimerkiksi World of Warcraft, FarCry ja Escape From Monkey Island hittipeleissä /7,9/.

Lua tarjoaa yksinkertaisen syntaksin, hajautustaulut, laajennettavan ja monipuolisen rakenteen. Lua on dynaamisesti tyypitetty, rekisteripohjainen virtuaalikone ja siinä on automaattinen muistinhallinta, eli ns. roskien keruu. Tässä luvussa tutustutaan yleisesti Lua-ohjelmointikieleen ja sen erityispiirteisiin. Tämä luku toimii myös johdantona varsinaiseen siirtotyöhön.

2.1 Korkeamman tason ohjelmointikieli

Lua tarjoaa erilaisen tavan ohjelmoida, kuten esimerkiksi C tai Assembler. Lua on niin sanotusti erittäin korkean tason ohjelmointikieli eli skriptauskieli.

Seuraavaksi tarkastellaan kolmea eri tasoista ohjelmointikieltä: /6/

1. Konetason ohjelmointikielet, esimerkiksi Assembler.
2. Systeemitason ohjelmointikielet, esim. C ja C++.
3. Skriptauskielet, esim. Lua, Python.

Konetason ohjelmointikielet ovat kaikkein nopeimpia ja niitä käytetäänkin erittäin suoritusajasta riippuvaisissa ohjelmistoissa. Näissä kielissä yhtä koodiriviä vastaa yksi konekielen komento. Ohjelmoijan tarvitsee huolehtia erittäin matalan tason asioista, kuten rekisterien allokoinnista. Konetason ohjelmointikielet tuottavat erittäin nopeita ohjelmia, mutta vastaavasti ohjelmat ovat vaikeita ylläpitää ja hitaita kehittää.

Hitaan ja vaikean ohjelmistokehityksen vuoksi alkoivat systeemitason ohjelmointikielet kehittyä. Systeemitason ohjelmointikielet eroavat kahdella tavalla konetason ohjelmointikielistä: ne ovat korkeamman tason ohjelmointikieliä ja ne ovat vahvasti tyypitettyjä kieliä.

Systeemitason ohjelmointikielissä korkeampi taso saavutetaan erillisellä kääntäjällä. Kääntäjä tuottaa koodista binääriä eli konekäskyjä. Kääntäjän avulla voidaan automatisoida monia asioita, kuten rekisterien allokointia ja parametrien välittämistä eri aliohjelmille.

Toinen suuri ero systeemitason ohjelmointikielillä on vahva tyypitys. Tyypityksellä saavutetaan monia etuja. Tyypityksellä saadaan ohjelmistoihin abstraktioita eli yksinkertaistuksia mallinnettavasta käsitteestä. Abstaktioilla saavutetaan selkeyttä ja turvallisuutta siihen, miten asioita tulisi käsitellä. Turvallisuus saadaan jo käännösvaiheessa, sillä kääntäjät osaavat tehdä virheilmoituksia tyyppitietojen perusteella.

Systeemitason ohjelmointikielet korvasivat ajan myötä lähes täysin konetason ohjelmointikielet. Konetason kieliä tosin vielä käytetään, mutta niiden käyttö on supistunut hyvin pienten ohjelmien toteuttamiseen, joissa saadaan parempi

suoritusaika.

Siinä missä systeemitason ohjelmointikielet suunniteltiin korvaamaan konetason kielet, skriptauskielet ottavat erilaisen lähestymistavan ohjelmointiin: skiptauskielet on suunniteltu laajentamaan ohjelmistoja, ei toteuttamaan ohjelmistoja täysin tyhjästä. Siis skirptauskielien ei ole tarkoitus korvata täysin systeemitason ohjelmointikieliä, vaan tarjota vielä korkeampi taso ohjelmointiin.

Skriptauskielet saavuttavat korkeamman tason olemalla niin sanotusti heikosti tyypitettyjä ohjelmointikieliä. Heikosti tyypitetyissä kielissä muuttujilla ei ole tyyppiä, mutta muuttujien arvoilla on. Esimerkiksi muuttujassa *X* voi olla alussa merkkijono ”Hello world!”, mutta siihen voidaan ongelmitta sijoittaa myöhemmin vaikkapa numeroarvo 5.

Skriptauskielet ovat hyvin paljon hitaampia, kuin systeemitason ohjelmointikielet, koska skriptauskielet on rakennettu niiden yläpuolelle. Yleisesti skriptauskielet käyttävät virtuaalikonetta, jossa ohjelmakoodia suoritetaan. Siten yhden skriptikielen koodirivin ajaminen saattaa koostua tuhansista konekäskyistä.

Viime vuosina skriptauskielet ovat kuitenkin yleistyneet niin prosessoritehojen kasvaessa kuin skriptauskieltenkin kehittyessä. Nykyään skriptauskieliä käytetään laajasti graafisten käyttöliittymien ohjelmoinnissa, Internetsivustojen ohjelmoinnissa ja valmiiden komponenttien yhdistämiseen.

Skriptauskielien käytöllä saavutetaan nopeampi ohjelmistokehitysaika. Skirptauskielet ovat myös nopeita oppia, koska heikko tyypitys tekee niiden syntaksista paljon yksinkertaisemman kuin systeemitason ohjelmointikielissä on. Korkeammalla ohjelmointitasolla ja yksinkertaisella syntaksilla saavutetaan skriptauskielillä suurempi tuottavuus kuin systeemitason ohjelmointikielillä.

2.2 Dynaaminen tyypitys Luassa

Lua on dynaamisesti tyypitetty kieli, jossa muuttujilla ei ole tyyppiä vaan niiden arvoilla. Luassa muuttujien arvoilla on kahdeksan perustyyppiä: nil, boolean-totuusarvot, numero, funktio, merkkijono, userdata, säie ja taulukko. Tässä kappaleessa käydään myöhemmin lävitse kaikki muuttujatyypit.

Dynaaminen tyypitys tarkoittaa sitä, että muuttujien tyyppiä voidaan ajonaikaisesti muuttaa toiseksi. Alla oleva Lua-skripti havainnollistaa dynaamisen tyypityksen käyttöä Luassa. Esimerkissä myös tutustutaan yleisimpiin Lua-tyyppeihin.

```
#1      print(type(a))   --> nil   ('a' is not initialized)
#2      a = 10
#3      print(type(a))   --> number
#4      a = "a string!!"
#5      print(type(a))   --> string
#6      a = print        -- yes, this is valid!
#7      a(type(a))       --> function
```

Listaus 1 Dynaaminen tyypitys Luassa /2/.

Edellisessä esimerkissä muuttujaa *a* käytettään havainnollistamaan dynaamista tyypitystä. Luan yksi erityispiirteistä ilmenee heti esimerkin ensimmäisellä rivillä: muuttujaa *a* voidaan käyttää ennen kuin se on alustettu. Jos kuitenkin käytetään alustamatonta muuttujaa, sen tyyppi on silloin *nil*. *nil* on Luassa käytetty erikoistyyppi, joka tarkoittaa tyhjää, alustamatonta arvoa.

Esimerkissä käytetään *type*-funktiota tarkastamaan muuttajan arvon tyyppi. *type*-funktio ottaa parametrina tarkasteltavan muuttujan ja palauttaa sen tyypin. Esimerkissä tyypin tulostukseen käytetään *print*-funktiota, joka tulostaa merkkijonot standardiulostuloon.

Rivillä 2, *a*-muuttujaan alustetaan numero arvo 10. Luassa kaikki numeromuuttujat ovat reaalilukuja. Seuraavalla rivillä tulostetaan muuttujan *a* uusi tyyppi.

Rivillä 4 ilmenee varsinainen dynaaminen tyypitys: *a*-muuttujan arvo muutetaan numerosta 10 merkkijonoon ”a string!!” (ilman lainausmerkkejä). Tämä ei olisi mahdollista esimerkiksi C-kielessä ilman tyyppimuunnosta, mutta Luassa muunnoksen voi suorittaa ajonaikaisesti pelkällä uudelleen alustuksella.

Luassa voidaan myös liittää muuttujia funktioihin, kuten esimerkiksi C-kielessä

voidaan tehdä funktio-osoittimia. Esimerkissä rivillä 6, muuttuja *a* alustetaan *print*-funktioon. Seuraavalla rivillä muuttuja *a* välittää *print*-funktiolle parametrina funktion *type* paluuarvon. *Type*-funktio palauttaa *a*-muuttujan arvon tyypin, joka on funktio.

Luan muut tyypit

Edellisessä esimerkissä tutustuttiin Luan tyypeistä *nil*-tyyppiin, numeroon, merkkijoon ja funktioon. Seuraavaksi tutustutaan muihin Luan tyyppeihin.

Boolean-totuusarvot

Boolean-totuusarvoja on vain kaksi: *true* (tosi) ja *false* (epätosi). Boolean-totuusarvoja käytetään yleisesti vertailuoperaattori *if*:ssä. Luassa boolean-totuusarvot ovat hyvin samankaltaisia kuin esimerkiksi C-kielessä. Tosin Luassa mitä tahansa tyyppiä voidaan käyttää totuusarvona. Tällöin *false* ja *nil* vastaavat epätotta ja kaikki muut vastaavat totta. Luan totuusarvojen käyttö eroaa myös muista skriptauskielistä siten, että numeroarvo nolla ja tyhjä merkkijono vastaavat totta.

Userdata ja säie

Userdatalla ja light userdatalla voidaan tallettaa C- tai C++-tietorakenteita Luan käytettäväksi. Luassa ei pystytä käyttämään userdataa muuhun kuin muuttajaan sijoittamiseen ja yhtäsuuruuden testaamiseen, joten userdatan varsinainen käsittely tapahtuu C-kirjastossa. Userdataa käytetään uusien tyyppien ja tietorakenteiden luomiseen, mikä tarjoaa Lualle geneerisen tavan käsitellä kaikkia uusia C-kirjaston luomia tyyppejä. Userdataa käsitellään tarkemmin luvussa 5. Säiettä käsitellään korutiinien yhteydessä.

Taulukot

Luan taulukot eroavat suuresti C-kielen tavasta käsitellä taulukkoja. Luassa taulukot ovat niin sanottuja hajautustauluja /9/. Hajautustauluja käytetään yleisesti myös muissa skriptauskielissä, kuten esimerkiksi Pythonissa. Hajautustaulun suurin ero on taulukon indeksoinnissa: hajautustaulun indeksi voi olla mitä tahansa tyyppiä, eikä se ole siis sidottu numerointiin, kuten C-kielessä. C-kielessä taulukon ensimmäisen alkion indeksi on aina nolla, mutta Luassa ensimmäisen alkion indeksi voi olla vaikkapa merkkijono, funktio tai mikä muu tahansa tyyppi. Ainoa rajoitus indeksointiin on se, että *nil*-tyyppillä ei saa indeksoida taulukkoa.

Toinen suuri ero Luan hajautustauluilla on dynaamisesti muuttuva koko. Kun esimerkiksi C-kielessä halutaan luoda taulukko, sen koko pitää määritellä alustuksessa. Luassa sen sijaan taulukkoon voidaan lisätä ja siitä poistaa alkioita rajoituksitta. Luassa taulukot ovat ainoa tapa luoda tietorakenteita.

Lua ei ole varsinaisesti oliopohjainen ohjelmointikieli, mutta taulukot tarjoavat mekanismit luoda ja käsitellä olioita. Tätä käsittellään vielä tarkemmin tässä luvussa.

2.3 Korutiinit

Korutiini (*coroutines*, ei vakiintunutta suomennosta) on Luan todellinen erityispiirre, sillä korutiineja käytetään hyvin harvoissa ohjelmointikielissä. Esimerkiksi C-kielestä ei löydy vastinetta korutiineille.

Korutiini on vaihtoehto perinteiseen aliohjelma-ajattelumalliin ja sen käyttö voidaan rinnastaa säikeiden käyttöön. Korutiini on samankaltainen säikeiden kanssa, koska niissä ohjelmansuoritus voidaan jakaa samankaltaisesti kuin säikeiden käytössä. Korutiinifunktioita ajetaan samassa säikeessä, joten vain yhden korutiinifunktion ajaminen kerrallaan on mahdollista. Tällöin muiden korutiinifunktioiden suoritus on asetettu tauolle.

Korutiinifunktiolla on kolme tilaa:

1. Suoritus (*running*)
2. Tauko (*suspended*)
3. Kuollut (*dead*)

Seuraavassa yksinkertaisessa esimerkissä tutustutaan korutiinifunktion tiloihin.

```
#1      co = coroutine.create(function ()
#2         print("hi")
#3         coroutine.yield()
#4         print("hi again")
#5       end)
#6      print(co)   --> thread: 0x8071d98
#7      print(coroutine.status(co))   --> suspended
#8      coroutine.resume(co)   --> hi
#9      print(coroutine.status(co))   --> suspended
#10     coroutine.resume(co)   --> hi again
#11     print(coroutine.status(co))   --> dead
```

Listaus 2 Korutiinifunktion tilat /2/.

Rivillä 1 luodaan nimeämätön korutiinifunktio käyttämällä *coroutine*-kirjaston *create*-funktiota. Rivillä 6 tarkastetaan muuttujan *co*-tyyppi, joka on säie. Näin ollen korutiinifunktiot ovat Luassa tyypiltään säikeitä. Seuraavalla rivillä käytetään *coroutine*-kirjaston *status*-funktiota, jolla voidaan tarkastaa annetun korutiinifunktion tilaa. Koska korutiinifunktiota ei ole vielä kutsuttu, se on taukotilassa (*suspended*).

Rivillä 8 korutiinifunktio ajetaan käyttämällä *coroutine*-kirjaston *resume*-funktiota. Tämän jälkeen näytölle tulostuu ”hi”-merkkijono. Huomioitavaa korutiinifunktiossa on se, että sen suoritus keskeytyi *yield*-funktiokutsulla. Tämä voidaan todeta tarkastamalla korutiinifunktion tila rivillä 9. Tämän jälkeen voidaan *resume*-funktiolla vielä jatkaa korutiinifunktion suoritusta, jolloin näytölle tulostuu merkkijono ”hi again”. Rivillä 11 tarkastetaan korutiinifunktion tila, joka on kuollut (*dead*). Näin ollen korutiinin suoritus on päättynyt.

Luan käyttämät korutiinit ovat niin sanottuja asymmetrisia korutiineja, eli Luassa on toteutettu erilliset funktiot, joilla voidaan käynnistää (*resume*) tai keskeyttää (*yield*) korutiinin suoritus. Vastaavasti ohjelmointikielissä, jotka käyttävät

symmetrisiä korutiineja, voidaan vain yhdellä funktiolla siirtää suoritus korutiinilta toiselle.

2.4 Olio-ohjelmointi ja metametodit

Lua ei ole varsinaisesti olio-ohjelmointikieli, mutta Lua tarjoaa kuitenkin mekanismit olioiden käyttämiseen. Luassa taulukkotyyppiä voidaan pitää olion vastineena. Taulukolla voi olla sisäisiä jäsenmuuttujia, metodeja ja taulukot voivat periä ominaisuuksia muilta taulukoilta, kuten muissa oliopohjaisissa ohjelmointikielissä.

Luassa luokkakäsite eroaa hieman siitä, millainen se esimerkiksi C++:ssa on. C++:ssa jokainen olio on jonkin luokan ilmentymä, mutta Luassa jokainen olio määrittelee toimintansa oliokohtaisesti. Luassa voidaan kuitenkin käyttää luokkamaisuutta periyttämällä olioita yhteisestä prototyypistä.

Metametodit

Metametodeja voidaan käyttää yhdistämään erilaisia olioita, tai taulukkoja. Luassa metametodit on kapseloitu metataulukkoon. Metataulukoita voidaan käyttää tavallisissa taulukoissa (esimerkiksi *setmetatable*-funktiolla), tai niitä voidaan myös käyttää Userdata-tyyppisissä arvoissa. Metatauluihin userdatassa tutustutaan paremmin luvussa 5.

Metataulut ovat erittäin monikäyttöisiä: metatauluja voidaan tehdä mihinkä tahansa taulukkoon, yhteisiä metatauluja voidaan jakaa eri taulukkojen välillä, taulukko voi olla myös itsensä metataulu.

Seuraavassa esimerkissä tehdään yhteinen prototyyppiolio, sekä periytetään kaksi oliota tästä prototyypistä käyttämällä metatauluja.

```
#1      Account = { balance = 0 }
#2      function Account:new (o)
#3            obj = o or {}
#4            setmetatable(obj, self)
#5            self.__index = self
#6            return obj
#7      end
#8      function Account:withdraw (v)
#9          self.balance = self.balance - v
#10     end
#11     function Account:getbalance (v)
#12         return self.balance
#13     end
#14     a = Account:new()
#15     b = Account:new{balance = 10000}
#16     a:withdraw(500)
#17     b:withdraw(1500)
#18     print( a:getbalance() )
#19     print( b:getbalance() )
```

Listaus 3 Olio-ohjelmointia Luassa /2/.

Rivillä 1 luodaan *Account*-olio, jolla on balance-jäsenmuuttuja. Rivillä 2 luodaan *Account*-olioon *new*-metodi, jolla voidaan luoda uusia Account-tyyppisiä olioita. Tälle metodille voidaan myös antaa parametrina valmis olio. Rivillä 3 alustetaan *obj*-olio, joko välitetyllä *o*-parametrilla tai luodaan uusi olio, mikäli *o*-parametri on *nil*. Luassa *or*-operaattori palauttaa aina verrattavista parametreistä toden. Mikäli *o*-parametri ei ole olio, sen arvo on *nil*. Tällöin *o*-muuttuja alustetaan tyhjäksi taulukoksi {}-rakentajalla.

Luassa voidaan myös käyttää olioiden yhteydessä *self*-käsitettä, joka tarkoittaa metodin omistavaa oliota. Seuraavalla rivillä *self* vastaa *Account*-oliota, jolloin uudesta oliosta ja *Account*-oliosta tehdään metataulu. Luassa metatauluilla voidaan yhdistää ominaisuuksia erilaisten olioiden välillä. Näiden avulla voidaan esimerkiksi käyttää +,-,= operaattoreita erilaisten olioiden välillä. Tämä yhdessä seuraavan rivin metametodin *__index* kanssa tarkoittaa sitä, että *new*:ssa luotu uusi olio on perinyt *Account*-olion ominaisuudet.

Riveillä 8 – 13 luodaan metodeja, joilla voidaan vähentää pankkitilin saldoa ja hakea uusin saldo. Nämä metodit ovat jälleen oliokohtaisia, koska niissä käytetään *self*-parametriä.

Rivillä 14 luodaan *a*-olio. Tämä olio luodaan välittämättä *new*-metodille mitään parametreja, joten *a*-olion *balance*-muuttuja saa arvon nolla.

Rivillä 15 luodaan *b*-olio. Tämä olio luodaan välittämällä *new*-metodille olio, jossa *balance*-muuttujan arvo on 10000. On huomioitavaa, että

b = Account:new{balance = 10000} on vastaava kuin

b = Account:new({balance = 10000}) tai

b = Account.new(b, {balance = 10000})

Riveillä 16 ja 17 *a*- ja *b*-olioilta nostetaan erilaiset summat ja tulostetaan tilin saldo. Tällöin voidaan käyttää ko. olion omia metodeja eikä enään *Account*-olion metodeja. Lopuksi saadaan *a*-olion tilin saldoksi -500 ja *b*-olion tilin saldoksi 8500.

3 LUAN ARKKITEHTUURI

Luan toteutuksen vaatimuksena on tehokkuus, hyvä siirrettävyys, laajennettavuus, yksinkertaisuus, vakaus ja pieni koko /1/. Nämä vaatimukset on pyritty saavuttamaan toteuttamalla Lua modulaarisesti, eli jakamalla Luan toiminnallisuus selkeisiin yksilöllisiin kokonaisuuksiin.

Luan toteutus koostuu kolmesta päämoduulista:

1. Dynaaamisesta Lua-tulkista, joka ottaa vastaan Lua-skriptejä käyttäjältä dynaamisesti.
2. Lua-kääntäjästä, jolla voidaan kääntää Lua-skriptejä Lua-virtuaalikoneen ajettavaan muotoon eli niin sanottuun tavukoodiin (*bytecode*) /10/.
3. Virtuaalikoneesta, joka suorittaa Lua-skriptin.

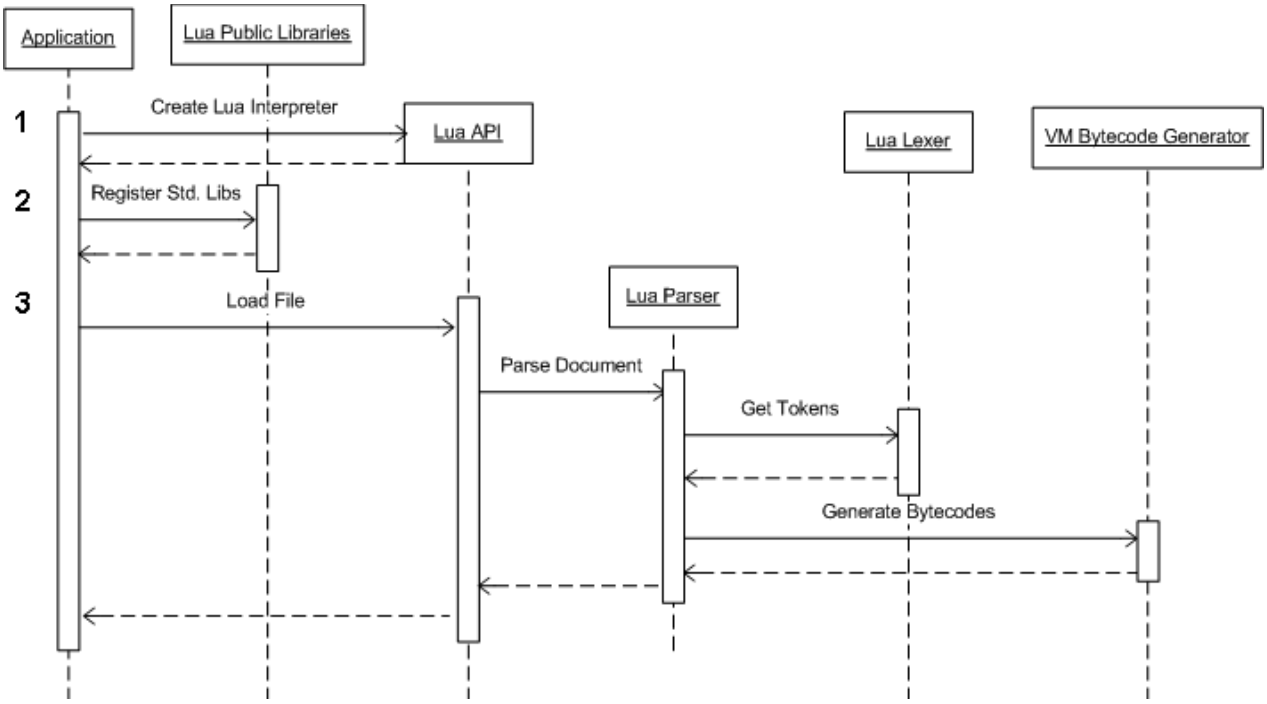
Luaa käyetään suurimmaksi osaksi kuitenkin sulautettuna suuremmissa ohjelmistoissa. Tämän vuoksi Luan virtuaalikoneeseen on kapseloitu koko Luan toiminnallisuus. Lua-tulkki onkin itseasiassa vain rajapinta käyttäjälle Luan virtuaalikoneeseen. Mikäli siis ohjelmistossa halutaan käyttää Luaa, niin Lua-tulkin

käyttö ei ole välttämätöntä.

3.1 Lua-skriptin tulkkaaminen

Lua-skriptin kääntäminen virtuaalikoneelle tavukoodiksi tapahtuu neljässä vaiheessa:

1. Lua-tulkin luominen.
2. Lua-kirjastojen lataaminen.
3. Lua-skriptin jäsentäminen tavukoodiksi.
4. Tavukoodin välittäminen virtuaalikoneelle.



Kuva 1 Luan käynnistyssekvenssi /1/

Kuvassa 1 siis ladataan Lua-skriptitiedosto ja jäsennetään tämä skripti tavukoodiksi.

Ensimmäisessä vaiheessa luodaan Lua-tulkki. Tämä tapahtuu kutsumalla Lua API:sta *lua_open*-funktioita, joka varaa muistista *lua_State*-tietorakenteen. Tätä tietorakennetta välitetään jokaiseen Lua API:n tarjoamaan funktioon, koska Luan sisäinen rakenne ei käytä globaaleja muuttujia. Näin ollen on mahdollista luoda useita instasseja Lua-tulkista.

Seuraavassa vaiheessa rekisteröidään Lua-tietorakenteeseen Lua-kirjastoja, joita Lua-skriptissä voidaan mahdollisesti käyttää. Luan tarjoamien standardikirjastojen lisäksi voidaan ladata myös omia kirjastoja rekisteröinnin yhteydessä. Luan standardikirjastoja tarkastellaan myöhemmin tässä luvussa.

Kolmannessa vaiheessa Lua-skripti ladataan ja jäsennetään tavukoodiksi. Lua API tarjoaa funktiot skriptien lataamiseen tiedostosta, tai Lua skriptin voi myös vaihtoehtoisesti antaa suoraan merkkijonona. Luan kääntäjä koostuu kolmesta alimoduulista: jäsentäjästä, lekseristä ja tavukoodigeneraattorista. Näihin moduuleihin tutustutaan paremmin myöhemmin tässä luvussa.

Kääntäjämoduuleilla on merkittävä vaikutus Luan nopeuteen, minkä vuoksi Luan toteutuksessa ei ole käytetty automaattista tavukoodigeneraattoria, vaan Luaan on toteutettu oma tavukoodigeneraattori.

Lopuksi neljännessä vaiheessa, kun kääntäjä on saanut tehtyä tavukoodin ilman syntaksivirheitä, tavukoodi voidaan välittää Luan virtuaalikoneen ytimelle. Luan virtuaalikoneen ydin lukee tavukoodit ja kääntää ne virtuaalikoneen käskyiksi.

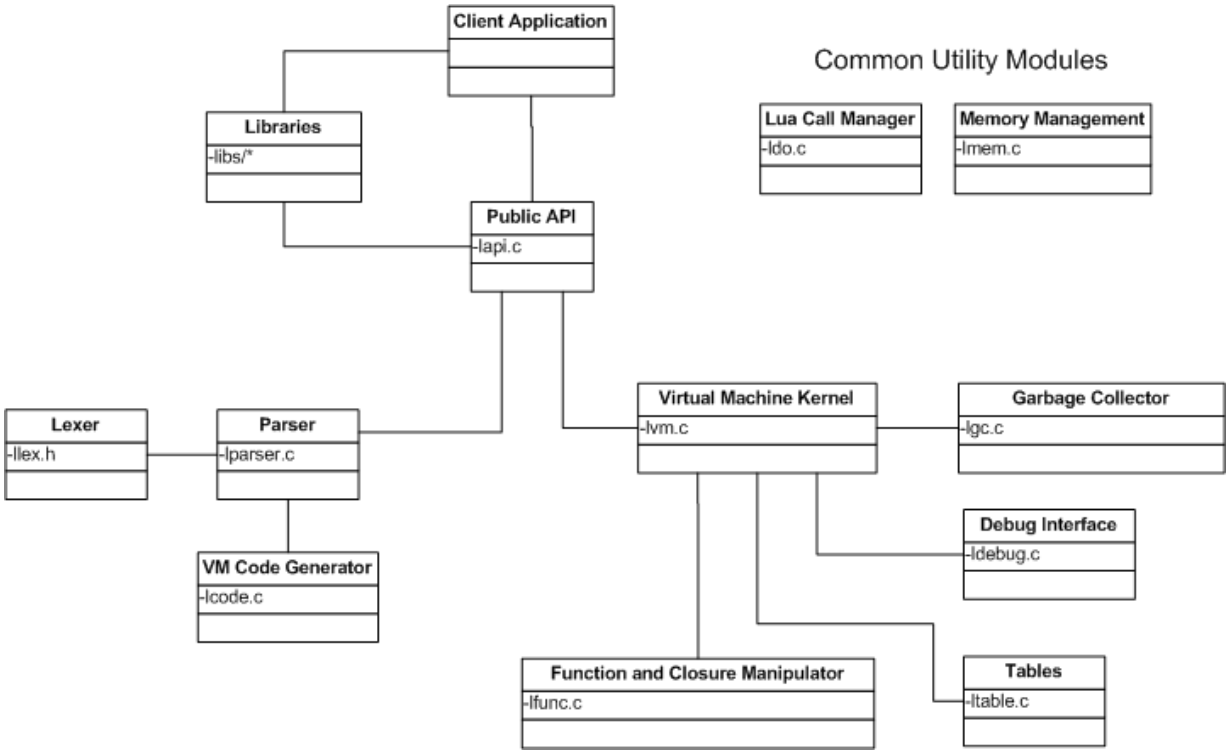
Luassa on myös mahdollista kääntää Lua-skripti tavukoodiksi erillisessä ohjelmassa. Tällöin Lua-skriptiä ei välitetä virtuaalikoneen ytimelle ajettavaksi, vaan tavukoodi talletetaan tiedostoon. Siten jos Lua-skripti esikäännetään, on mahdollista hypätä kolmas vaihe yli ja välittää Lua-skripti suoraan toisesta vaiheesta neljänteen vaiheeseen.

3.2 Luan modulaarinen rakenne

Luvun alussa todettiin, että Lua on jaettu kolmeen päämoduuliin: tulkkiin, kääntäjään ja virtuaalikoneeseen. Luan modulaarisuus mahdollistaa Luan eri moduulien irrottamisen varsinaisesta Luan toteutuksesta. Näin voidaan saada käyttäjittäin räätälöity Lua-tulkki. Luan moduulaarisuudella saadaan myös minimoitua ulkoisten sovellusten ja Luan välistä riippuvuutta. Tämä tarkoittaa sitä, että Luan kehittäjät voivat tehdä Luaan sisäisiä muutoksia, ilman että ne rikkoisivat yhteensopivuutta ulkoisen sovelluksen ja Luan API:n välillä. Näin ollen Luan API:n pitää pysyä mahdollisimman muuttumattomana.

Luan virtuaalikone jakaantuu neljään alimoduuliin:

1. Standardikirjastot.
2. Lua-API
3. Kääntäjä
4. Virtuaalikoneen ydin



Kuva 2 Luan modulaarinen rakenne /1/.

Kuvassa 2 on kuvattu Luan moduulit ja niitä vastaava lähdekooditiedosto. Lua-API toimii sidoksena eri moduulien välillä. Se sitoo Luan standardikirjastot, kääntäjän ja virtuaalikoneen ytimen yhteen. Lua-API jakaantuu vielä erilliseen apukirjastoon (*Auxiliary library*), joka tarjoaa mukavuusfunktioita Luan rajapintaan. Tämä kirjasto ei tarjoa sinällään mitään uutta toiminnallisuutta, mutta sen avulla saadaan korkeamman tason lähestyminen Luan toiminnallisuuteen.

Minimaallisessa Lua-toteutuksessa tarvitaan vain Luan virtuaalikoneen ydin ja sovelluksittain tarvittavat kirjastot. Kääntäjän voi poistaa, jos Luan virtuaalikoneen ytimelle välitetään valmiiksi käännetty tavukoodi.

3.3 Kääntäjä

Kääntäjän tarkoitus on kääntää Lua-skripti virtuaalikoneen ymmärtämään tavukoodimuotoon. Kuvan 2 mukaisesti kääntäjämoduuli jakaantuu vielä kolmeen alimoduuliin: lekseriin, jäsentäjään ja tavukoodigeneraattoriin. Lekseri hakee Lua-skriptistä varattuja sanoja, kuten esimerkiksi if, for, while. Jäsentäjä puolestaan analysoi lekserin hakemat komennot ja välittää ne tavukoodigeneraattoriin.

Kääntäjä on suunniteltu mahdollisimman tehokkaaksi, joten jäsennysvaiheessa havaitaan vain syntaksivirheitä. Luan käyttämä heikko tyypitys mahdollistaa hyvin nopean tarkistuksen, koska muuttujilla ei ole tyyppiä vaan niiden arvoilla. Tällöin muuttujien arvoissa olevat tyyppivirheet havaitaan vasta ajonaikana.

3.4 Virtuaalikoneen ydin

Luan tärkein toiminnallisuus on kapseloitu virtuaalikoneen ytimeen, eikä sitä voida poistaa muiden moduulien tapaan minimaalisesta Luan toteutuksesta. Luan ydin koostuu seuraavista alimoduuleista: roskien keruu (*garbage collector*), taulukoiden käsittely ja funktioiden ja näkyvyysalueiden hallintamoduulista (*closure*).

Luan virtuaalikoneen ydin ajaa jäsentäjän tuottamaa tavukoodia. Tavukoodin komennot ovat 32 bittiä pitkiä ja ne sisältävät yhdestä kolmeen parametriä. Seuraavaksi tutustutaan Luan ytimen toimintaan käyttäen yksinkertaista Lua-skriptiä ja siitä tuotettua tavukoodia.

```
#1      y = 5
#2      print(y)
```

Listaus 4 Yksinkertainen Lua-skripti /1/.

Listauksessa 4 on globaali muuttuja *y*, jolle alustetaan numeroarvo 5. Tämän jälkeen se tulostetaan käyttäen *print*-funktiota.

```
#1      LOADK           0 1     ; 5
#2      SETGLOBAL       0 0     ; y
#3      GETGLOBAL       0 2     ; print
#4      GETGLOBAL       1 0     ; y
#5      CALL            0 2 1
#6      RETURN          0 1 0
```

Listaus 5 Tuotettu tavukoodi /1/.

Listauksessa 5, *LOADK*-komento lataa arvon vakioarvotaulukosta. Vakioarvotaulukon tarkoitus on tallettaa mitä tahansa dataa, jotta virtuaalikoneen operaattoreiden ei tarvitse suoraan käyttää vakioarvoja. Näin ollen useat viittaukset samaan vakioarvoon haetaan samasta vakioarvotaulukon indeksistä.

SETGLOBAL-komento asettaa *y*-muuttujan arvon globaaliin taulukkoon. Globaalin taulukon tarkoitus on tallettaa arvoja, joita voidaan kutsua milloin tahansa Lua-skriptissä.

GETGLOBAL-komento lukee arvon globaalista taulukosta, jonka paluuarvo talletaan ensimmäiseksi parametriksi. Toinen parametri viittaa vakioarvotaulukon indeksiin. Näin ollen riveillä 3 ja 4 haetaan globaalista taulukosta *print*-funktio ja tämän jälkeen haetaan *y*-muuttuja. Rekisteri on siis pinon mallinen, jossa tällä hetkellä päälimmäisenä on *print*-funktio ja ja sen alapuolella *y*-muuttuja.

CALL-komento käytetään funktion suorittamiseen. Sen ensimmäinen parametri on suoritettavan funktion sijainti rekisterissä. Tällöin suoritetaan rekisterin ensimmäisessä alkiossa oleva *print*-funktio. *CALL*-komennon toinen parametri on *print*-funktion annettavien parametrien määrä lisättynä yhdellä, eli parametrina välittyy *y*-muuttuja. *CALL*-komennon kolmas parametri on kutsuttavan funktion paluuarvojen lukumäärä. Koska *print*-funktio ei palauta mitään, niin kolmanneksi parametriksi tulee nolla.

Lopuksi *RETURN*-komentoa käytetään funktiosta palaamiseen. *RETURN*-komennon ensimmäinen parametri kertoo palautettavien arvojen ensimmäisen alkion sijainnin. Muut arvot ovat ensimmäisen alkion jälkeen. Komennon toinen parametri kertoo palautettavien arvojen lukumäärän lisättynä yhdellä. Esimerkissä ei palaudu arvoja, joten toisen parametrin arvoksi asetetaan 1. Komennon kolmas parametri on palautettavien arvojen lukumäärä.

3.5 Luan standardikirjastot

Luan kirjastot jakaantuvat kahteen erilaiseen kirjastoon: peruskirjastoon ja järjestelmäkirjastoon. Peruskirjasto sisältää Luan ydinfunktioiden toteutuksen, kuten Lua-skriptitiedostojen lataamisen, dynaamisten kirjastojen lataamisen, metataulujen asettamisen jne. Järjestelmäkirjasto puolestaan tarjoaa pääsyn Luan ulkopuoliseen järjestelmään, kuten I/O-palveluihin, käyttöjärjestelmän palveluihin jne. Järjestelmäkirjastojen toteutus eroaa peruskirjastojen toteutuksesta siten, että järjestelmäkirjastojen funktiot ovat käytettävissä Lua-skripteissä olioiden metodeina. Esimerkiksi I/O-olio tarjoaa metodit lukemiseen ja kirjoittamiseen, kun peruskirjastojen funktioita voidaan kutsua suoraan.

Yhtenä Luan toteutuksen kulmakivenä on ollut hyvä siirrettävyys uusiin järjestelmiin. Tämän vuoksi Luan standardikirjastoissa on pyritty käyttämään mahdollisimman käyttöjärjestelmistä riippumatonta toteutusta käyttäen ANSI C:n tarjoamia funktioita. Kuitenkaan kaikkea Luan standardikirjastossa ei ole pystytty tekemään käyttämällä ANSI C:tä, kuten esimerkiksi dynaamisten kirjastojen (DLL) lataamista. Näin ollen kaikki ANSI C:stä poikkeavat toteutukset ovat Luan standardikirjastoissa. Seuraavassa luvussa tarkastellaan tarkemmin, millaisia ongelmia on siirtää Lua-kirjastoja Symbian-käyttöjärjestelmälle.

Luan standardikirjastot käsittävät peruskirjastojen lisäksi seuraavat kirjastot:

- Matematiikkakirjasto, tarjoaa palvelut yleisimpiin matemaattisiin funktioihin, kuten siniin, kosiniin jne.
- Taulukonkäsittelykirjasto, tarjoaa palveluita taulukonkäsittelyyn, kuten esimerkiksi alkioiden lisääminen ja poisto taulukosta.
- Merkkijonon käsittelykirjasto, tarjoaa lisäfunktioita merkkijonojen käsittelyyn.
- I/O-kirjasto, tarjoaa palveluita tiedoston käsittelyyn, kuten esimerkiksi tiedoston lukeminen ja kirjoittaminen. I/O-virtaa voidaan myös ohjata esimerkiksi näytölle.
- Käyttöjärjestelmäkirjasto, tarjoaa funktioita käyttöjärjestelmän tiedostojärjestelmään, ajanhallintaan ja ulkoisten ohjelmien ajamiseen (EXE). Tämä kirjasto sisältää paljon käyttöjärjestelmistä riippuvaista

lähdekoodia.

- Debug-kirjasto, ei ole varsinainen debuggeri, mutta tarjoaa funktiot sellaisen toteuttamiseen. Tämän kirjaston avulla voi esimerkiksi tarkkailla Luan rekistereiden sisältöä.

3.6 Luan C API

Aiemmessa kappaleessa todettiin, että Lua jakaantuu tulkkiin, kääntäjään ja virtuaalikoneeseen. Tulkki ja kääntäjä käyttävät Luan C API:a rajapintana virtuaalikoneeseen, jossa varsinaisesti suoritetaan Lua-koodin ajaminen. Lua onkin itseasiassa kirjasto, jota käsitellään C API:n kautta. Tämän vuoksi Luaa voidaan sulauttaa erilaisiin sovelluksiin mitä erilaisimpiin käyttökohteisiin. Luan C API on siis joukko funktioita, joilla voidaan kommunikoida ja vaihtaa dataa Luan ja C:n välillä.

Tässä luvussa käsitellään miten Lua ja C voivat kommunikoida: miten Luasta voi kutsua C-funktiota ja C:stä Lua-funktiota, miten voidaan välittää dataa funktioihin ja miten voidaan muodostaa C:ssä Luan ymmärtämiä tietorakenteita.

Luan pinorakenne

Kun Lua ja C kommunikoivat keskenään, ilmenee kaksi onglemaa, mitkä johtuvat kielten erilaisuudesta:

1. Lua on dynaamisesti tyypitetty kieli, kun C on staattisesti tyypitetty kieli.
2. Luassa muistinhallinta on automaattista, kun C:ssä se on ohjelmoijan vastuulla.

Näiden kahden ongelman ratkaisuun Luassa on kehitetty abstrakti pinomalli, jonka avulla voidaan vaihtaa dataa Luan ja C:n välillä. Luan pinon alkio voi sisältää minkä tahansa Lua-tyypin, kuten esimerkiksi funktion tai merkkijonon. Pinorakenne yksinkertaistaa ja vähentää funktioiden määrää Lua-API:sa, joilla voidaan vaihtaa dataa Luan ja C:n välillä. Lua on myös vastuussa pinon muistinhallinnasta.

Miltei kaikki funktiot Lua API:ssa käyttävät pinorakennetta. Luan pinorakenne toimii ns. LIFO- periaatteella (*Last In, First Out*), jossa pinon päälimmäiseksi asetetut arvot poistuvat pinosta aina ensimmäisenä.

C-funktioiden kutsuminen Luasta

Lua tarjoaa useita mekanismeja, joilla voi välittää dataa molempiin suuntiin C:n kanssa. Saumattoman kommunikoinnin kannalta onkin erittäin tärkeää, että on mahdollista suorittaa funktiokutsuja molempiin suuntiin: Lua-skriptissä pitää pystyä kutsumaan C:ssä toteutettuja funktioita ja vastaavasti C:ssä pitää pystyä kutsumaan funktioita, jotka on toteutettu Lua-skriptissä. Luan API tarjoaa keinot molempien ongelmien ratkaisuun.

Jotta Lua-skriptissä voitaisiin kutsua C-funktiota, se pitää ensin rekisteröidä Luan käytettäväksi. Tämä asettaa tiettyjä vaatimuksia kutsuttavaan C-funktioon. Tällöin kutsuttavan C-funktion pitää olla räätälöity Luaa varten. Luan toteutus ei pysty kutsumaan mitä tahansa C-funktiota, mutta on olemassa erilaisia Lua-laajennuksia, joilla tämän pystyy tekemään, kuten esimerkiksi C/Invoke /11/. Nämä toteutukset ovat tosin käyttöjärjestelmästä riippuvaisia, eivätkä siis ole helposti siirrettävissä käyttöjärjestelmästä toiseen.

Seuraavassa esimerkissä rekisteröidään C-funktio ja kutsutaan sitä Lua-funktiosta.

```
#1      static int l_sin( lua_State* L ) {
#2        double d = luaL_checknumber( L, 1 );
#3        lua_pushnumber( L, sin(d) );
#4        return 1;  /* number of results */
#5      }
```

Listaus 6 Kutsuttava C-funktio /2/.

Yllä olevassa listauksessa on C-funktio, joka laskee sinin annetusta numerosta ja palauttaa tämän vastauksen. Luasta kutsuttavan C-funktion suurin vaatimus on se, että siihen pitää mennä parametrina *lua_State*-tietorakenne ja sen pitää palauttaa *int*-tyyppinen numero. *lua_State*-tietorakenne sisältää Luan tilakoneen, jossa ovat kaikki rekisteröidyt Lua-funktiot ja -kirjastot. Luasta kutsuttavan C-funktion pitää palauttaa *int*-numero, koska siitä tiedetään, kuinka monta arvoa funktio palauttaa.

Luassa on siis mahdollista palauttaa funktiosta useita arvoja, toisin kuten C:ssä.

Rivillä 2 tarkistetaan, onko ensimmäinen parametri numero. *luaL_checknumber*-funktio aiheuttaa poikkeuksen, mikäli ensimmäinen parametri ei ole numero ja sen seurauksena suoritus päättyy *l_sin*-funktiossa. Kutsuttavassa C-funktiossa ei ole muuta keinoa tarkistaa annettuja parametrejä, joten C-funktion toteutuksessa on tarkistettava parametrien oikeellisuus käsin. Rivillä 3 lasketaan annetun numeron sini ja tämän tulos pannaan Luan pinoon päälimmäiseksi. Näin ollen rivillä 4 palautetaan numero 1, koska funktio palauttaa vain sinin annetusta numerosta.

```
#1      lua_pushcfunction( l, l_sin );
#2      lua_setglobal( l, "mysin" );
```

Listaus 7 C-funktion rekisteröinti /2/.

C-funktion rekisteröinti tapahtuu kahdessa vaiheessa yllä olevan listauksen mukaan. Ensin Lualle annetaan funktio-osoitin luotuun funktioon *lua_pushcfunction*-funktiolla. Tämän jälkeen C-funktio asetetaan globaaliin taulukkoon *lua_setglobal*-funktiolla, jolloin sitä voidaan kutsua mistä tahansa kohtaa Lua-skriptistä. Rekisteröityä C-funktiota voidaan kutsua Lua-skriptistä esimerkiksi komennolla: *mysin(1)*.

Lua-funktioiden kutsuminen C:stä

Lua-funktion kutsuminen C:stä toimii samoin periaattein kuin C-funktion kutsuminen Luasta. Ensin kutsuttava funktio on rekisteröitävä käytettäväksi funktioksi. Seuraavassa esimerkissä kutsutaan Lua-skirptistä funktiota, joka laskee kahdesta annetusta parametristä matemaattisen tuloksen.

```
#1      function f (x, y)
#2        return (x^2 * math.sin(y))/(1 - x)
#3      end
```

Listaus 8 Kutsuttava Lua funktio /2/.

Yllä olevassa listauksessa oleva Lua-funktio pitää rekisteröidä, jotta sitä voidaan kutsua C:stä. Rekisteröinti tapahtuu ”ajamalla” Lua-skriptitiedosto ensin, jolloin Lua-funktio *f* on Luan muistissa. Tämän jälkeen voidaan kutsua Lua-funktiota C:stä.

```
#1      double f ( double x, double y ) {
#2        double z;
#3        /* push functions and arguments */
#4        lua_getglobal( L, "f" );  /* function to be called */
#5        lua_pushnumber( L, x );   /* push 1st argument */
#6        lua_pushnumber( L, y );   /* push 2nd argument */
#7        /* do the call (2 arguments, 1 result) */
#8        if( lua_pcall( L, 2, 1, 0 ) != 0 )
#9          error( L, "error running function 'f': %s",
#10                  lua_tostring( L, -1 ) );
#11       /* retrieve result */
#12       if ( !lua_isnumber( L, -1 ) )
#13         error( L, "function 'f' must return a number" );
#14       z = lua_tonumber( L, -1 );
#15       lua_pop( L, 1 );  /* pop returned value */
#16       return z;
#17     }
```

Listaus 9 Lua funktion kutsu C:stä /2/.

Yllä olevassa listauksessa kutsutaan Lua-skriptistä funktiota. Rivillä 4 haetaan kutsuttava Lua-funktio globaalista taulukosta *lua_getglobal*-funktiolla, jossa annetaan parametrina *lua_State*-tietorakenne ja kutsuttavan funktion nimi merkkijonona. Kun kutsuttava Lua-funktio on pinon päälimmäisenä, voidaan antaa Lua-funktiolle tarvittavat parametrit. Nämä asetetaan myös Luan pinoon siinä järjestyksessä missä niitä kutsuttavassa Lua-funktiossa käsitellään. Eli ensin välitetään *x*-muuttujan rivillä 5 *lua_pushnumber*-funktiolla ja sen jälkeen *y*-muuttujan vastaavalla tavalla.

Staattisen datan tallettaminen Luaan

Luan C-API tarjoaa rekisterimekanismin, minne voi tallettaa dataa C-koodista. Rekisteri on erillään globaalista- ja paikallisestataulukosta, joten Lua-koodista ei voida muuttaa rekisterin sisältöä. Rekisteriin voidaan tallettaa mitä tahansa arvoja.

Luan rekisteri on taulukko, jota voidaan käyttää samoin periaattein, kuten muitakin Luan tauluja. Rekisteri muodostuu talletettavasta datasta, sekä indeksistä, joka yksilöi datan. Indeksi, eli avain, pitää valita huolellisesti, koska Luan rekisterit ovat

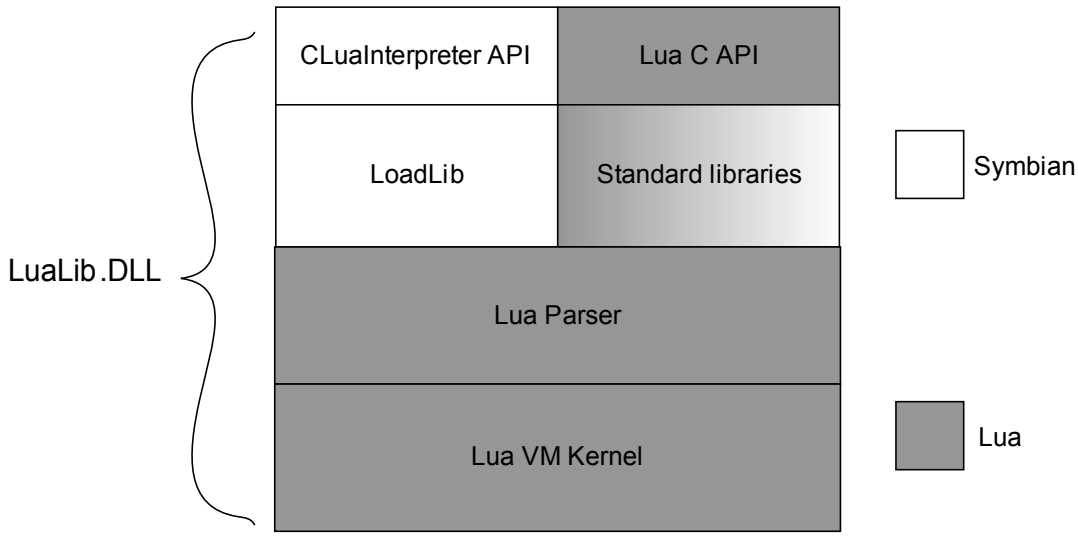
kaikkien C-kirjastojen käytettävissä, jolloin sekaannukset ovat mahdollisia.

Lua tarjoaa automatisoidun referenssijärjestelmän, jolla voidaan tallettaa dataa rekistereihin välittämättä avaimien yksilöllisyydestä. Referenssijärjestelmä tallettaa Luan pinon päälimmäisen arvon rekisteriin, sekä tarkistaa avaimen yksilöllisyyden. Tämän jälkeen referenssijärjestelmä palauttaa referenssiarvon, joka viittaa rekisteriin talletettuun dataan. Tämän jälkeen voidaan käsitellä dataa rekistereistä referenssiarvon kautta.

Lua tarjoaa myös Upvalues-mekanismin datan tallentamiseen. Upvalues-mekanismi eroaa rekisteristä lähinnä datan näkyvyyden rajoittamisella. Upvalues-mekanismissa voidaan tallettaa dataa siten, että talletettava data on näkyvissä vain tietyssä funktiossa.

4 LUAN PERUSTOTEUTUKSEN SIIRTÄMINEN

Tässä luvussa käsitellään mitä ongelmia on Luan perustoteutuksen siirtämisessä Symbian-käyttöjärjestelmään.



Kuva 3 Luan toteutuksen siirtäminen Symbian-käyttöjärjestelmään

Kuvassa 3 on havainnollistettu kuinka Lua on siirretty Symbianiin. Valkoisella kuvatut laatikot ovat moduuleja, joita on pitänyt muuttaa Luan standardi toteutuksesta. Harmaalla kuvatut laatikot ovat moduuleja, jotka ovat pysyneet muuttumattomina. Liukuvärillä kuvatuissa laatikoissa on pitänyt tehdä paikallisia muutoksia.

Luan siirtämisen periaatteena on ollut Luan lähdekoodin ja toiminnallisuuden säilyttäminen mahdollisimma muuttumattomana, jolloin olisi mahdollista kääntää sama Lua-lähdekoodi myös muille järjestelmille, kuten esimerkiksi Linuxille. Tämä on saavutettu käyttämällä ehdollista kääntämistä eli käännösmakroja, joilla käyttöjärjestelmä kohtainen koodi on eriytetty muusta toteutuksesta.

Luan virtuaalikoneen toiminnallisuus on kapseloitu LuaLib-kirjastoon. Tällöin Luan toiminnallisuutta voidaan käyttää useissa erilaisissa sovelluksissa. LuaLib-kirjastoa voidaan käyttää kahdella rajapinnalla. Sitä voidaan käyttää Luan C API:n kautta, tai Symbianille tehdyn *CLuaInterpreter*-luokan kautta. *CLuaInterpreter*-luokka tarjoaa yksinkertaistetun Symbian C++-rajapinnan Luan virtuaalikoneen alustamiseen, vapauttamiseen, sekä Lua-skripti tiedostojen ajamiseen laitteelta.

Dynaamisten Lua kirjastojen, eli Lua-laajennuksien, lataamista varten luotiin LoadLib-moduuli, johon tutustutaan tarkemmin tämän luvun aikana.

4.1 Siirtämisen ongelmakohdat

Luan perustoteutus tarjoaa toteutuksen Windowsille, Linuxille, MacOS X:lle, sekä geneerisen toteutuksen, jossa on käytetty vain ANSI C kirjaston funktioita. Symbian OS 9.1 ei tue kokonaan näiden käyttöjärjestelmien tarjoamia kirjastoa, joten siirtämisen lähtökohdaksi valittiin geneerinen toteutus.

Geneerinen toteutus tarjoaa vain rungon dynaamiseen kirjastojen ja Lua-skriptien lataamiseen. Eikä se tarjoa toteutusta myös muutamiin Luan asetustietoihin, jotka on määritelty luaconf-otsikkotiedostossa. Jotta geneeristä toteutusta voitaisiin käyttää mahdolliseti uudelleen muissa järjestelmissä, luotiin erillinen käännösmakro Symbian-keskeisiin komponentteihin.

Ongelmia myös aiheutti Luan OS-kirjasto, josta Symbianin ANSI C-kirjaston toteutus ei tukenut tiedoston poistofunktiota. Tähän versioon tukea ei toteutettu, vaan kutsuttaessa tätä funktiota aiheutuu virheilmoitus.

4.2 Käännösympäristö

Seuraavaksi käsitellään lyhyesti Luan käännösympäristön asettamista Symbianille.

Käännös tehtiin emulaattorille sekä laitetta varten. Testaamiseen luotiin myös LuaLaunch-sovellus, jolla voidaan käyttää Luan Symbian rajapintaa.

Symbian-kirjaston kääntämiseen tarvitaan projektitiedosto (Lualib.mmp), käännöstiedosto (bld.inf), sekä tarvittavat lähdekoodit. Tämän lisäksi tarvitaan asennuspakettitiedosto (Lualib.pkg), sekä sertifikaatti, jotta kirjasto voidaan asentaa ja ajaa testi puhelimessa (Nokia N71).

Yllä olevista Symbian-käännöstiedostoista mmp-tiedostossa on määritelty käännettävän komponentin ulkoiset riippuvaisuudet, käännettävät lähdekoodit, sekä käännösmakrot.

```
TARGET        lualib.dll
TARGETTYPE    dll
MACRO         __STRICT_ANSI__
MACRO         LUA_BUILD_AS_DLL
MACRO         LUA_SYMBIAN
... // lähdekooditiedostojen listaus
LIBRARY       euser.lib
LIBRARY       estlib.lib
```

Listaus 10 Lualib.mmp-tiedoston keskeinen sisältö

Lualib.mmp-tiedostossa määritellään käännettävät Symbian-rajapinnan lähdekoodit, sekä Luan lähdekoodit. Käännöksessä käytetään kolmea makroa, joilla voidaan hallita mitä osia otetaan mukaan lähdetiedostoista. Tärkein näistä on *__STRICT_ANSI__* -makro, jolla otetaan Luan toteutuksesta vain ANSI C:llä toteutettu osa. *LUA_SYMBIAN* makro luotiin siirtoa varten. Sitä käytetään eriyttämään Symbian keskeinen lähdekoodi Luan perustoteutuksesta.

Tärkeimmät ulkoiset kirjastot olivat euser-, sekä estlib-kirjastot. Euser-kirjastossa on toteutettuna Symbianin yleisimmien käytetyt komponentit. Estlib-kirjastossa on Symbianin ANSI C-kirjaston toteutus.

4.3 Luan Symbian C++-rajapinnan toteutus

CLuaInterpreter-luokka tarjoaa yksinkertaisen rajapinnan Symbian C++:lle. *CLuaInterpreter*-luokan käyttötarkoitus on luoda *lua_State*-tietorakenne, ladata Luan standardikirjastot, sekä tarjota metodit Lua-skriptien lataamiseen ja ajamiseen laitteelta.

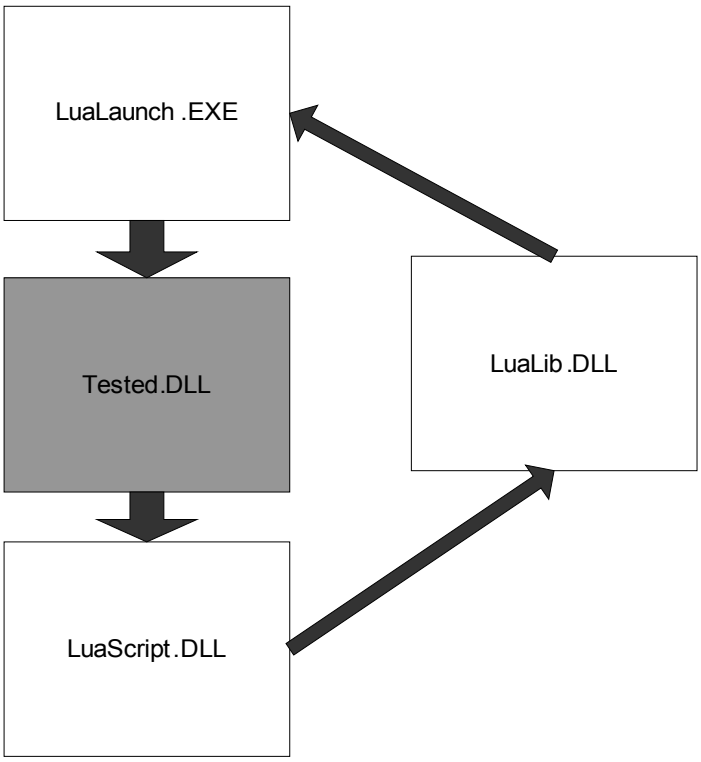
CLuaInterpreter-luokka tarjoaa rajapinnan neljälle metodille, jolla voidaan luoda *CLuaInterpreter*-tyyppinen olio, ajaa tiedostossa oleva Lua-skripti ja välittää *lua_State*-tietorakenne.

CLuaInterpreter:ssä käytetään Symbian-ohjelmoinnissa tyypillistä kaksivaiheista rakenninta. Tällöin perusrakentajassa tehdään vain turvallisia alustuksia, jotka eivät aiheuta poikkeusta. Toisen vaiheen rakentajassa alustetaan *CLuaInterpreter*-luokan muut jäsenmuuttujat, jotka saattavat aiheuttaa poikkeuksen, tai joiden alustuksen epäonnistuminen on kriittistä olion toiminnan kannalta.

Rakentaja eroaa myös siten, että että *CLuaInterpreter*-tyyppisten olioiden luomista on rajoitettu säiekohtaisesti. *CLuaInterpreter* on niin sanotusti Singleton-luokka /12/, jolloin säiettä kohden voi rakentaa vain yhden *CLuaInterpreter*-tyyppisen olion. Tällöin käytetään hyväksi Symbianin tarjoamaa TLS (*Thread Local Storage*) toiminnallisuutta /12/. Säiekohtainen rajoitus otettiin käyttöön, jotta olisi mahdollista päästä käsiksi säikeessä käytettävään *lua_State*-tietorakenteeseen myös niissä funktioissa, mihin sitä ei välitetä suoraan parametrina. Tämän käyttöön tutustutaan seuraavaksi tarkemmin.

TLS:n käyttö rakentajassa

Kuva 4 havainnollistaa miten TLS:n käyttöä voidaan hyödyntää esimerkiksi komponenttitestauksessa.



Kuva 4 TLS:n hyödyntäminen Luassa testauksessa

Kuvan esimerkissä testataan Tested-kirjastoa, joka on riippuvainen jostakin alemman tason kirjastosta. Tätä alemman tason kirjastoa ei ole esimerkissä vielä toteutettu, tai sitä ei ole muutoin saatavilla, joten sen toimintaa on pyritty simuloimaan Lua-skriptillä. Lua-skriptiin välitetään siis funktiokutsu LuaScript-kirjastosta.

Esimerkissä LuaLaunch-sovellus luo säiekohtaisen *CLuaInterpreter*-olion. Tested-kirjastossa ei ole Lua-riippuvaista ohjelmakoodia, joten *lua_State*-tietorakentta ei voida välittää Tested-kirjaston kautta alemman tason LuaScript-kirjastoon. Näin ollen kun Tested-kirjasto suorittaa kutsun LuaScript-kirjastoon, LuaScript-kirjasto ei pysty kutsumaan Lua-skriptia ilman osoitinta *lua_State*-tietorakenteeseen.

Tämän ongelman ratkaisuna käytetään Symbianin TLS:ää. Tällöin aina kun tulee Tested-kirjastosta kutsu LuaScript-kirjastoon, niin haetaan TLS:n kautta osoitin säiekohtaiseen *CluaInterpreter*-olion, jolloin päästään myös käsiksi *lua_State*-

tietorakenteeseen.

4.4 Muutokset Luan perustoteutuksessa

Aiemmin tässä luvussa todettiin, että siirtämisen kannalta suurimmat ongelmat ovat dynaamisten kirjastojen lataamisessa, joissakin Luan standardikirjastoissa, sekä Luan asetuksissa. Tässä kappaleessa tutustutaan näihin muutoksiin tarkemmin. Näiden muutosten myötä Lua on käytettävissä laitteessa ja emulaattorissa.

Lua-laajennusten lataaminen

Lua-laajennukset ovat siis kirjastoja, tai Lua-skriptitiedostoja, joita voidaan kutsua Lua-skriptistä ajonaikaisesti. Symbianissa dynaamisesti ladattavia C++-kirjastoja kutsutaan polymorfhisiksi kirjastoiksi. Tämä asettaa myös tiettyjä vaatimuksia kirjaston toteuttamiseen. Kirjastojen pitää olla dynaamisesti ladattavissa Lua-skriptissä, joten niitä ei voida staattisesti linkata LuaLib-kirjastoa vasten. Näin ollen Lua ei oikeastaan tiedä mitään ladattavista kirjastoista, mitä ne esimerkiksi sisältää ja mitä funktioita niissä on.

Lua-laajennusten käyttö toimii seuraavasti:

1. Lua-skriptistä pyydetään lataamaan laajennus esimerkiksi *require*-komennolla, jolle välitetään parametriksi kirjaston nimi.
2. Tämän jälkeen Lua lataa kirjaston, tai Lua-skriptin dynaamisesti levyltä.
3. Kun kirjasto on ladattu, ladataan kirjastosta alustusfunktio, jota kutsumalla rekisteröidään laajennuksen tarjoamat Lua-funktiot. Alustusfunktio on aina vakio kaikissa Lua-laajennuksissa.
4. Kutsutaan haluttuja funktioita laajennuksesta.
5. Kun Lua-skripti on päättynyt ja Lua on aloittanut roskien keruun, laajennus suljetaan.

```
#1      void* LoadSymbianLib( const char* path )
#2      {
#3        TInt err = 0;
#4        TPtrC8 ptr( (TText8*)path );
#5        TBuf16<KMaxLen> buf;
#6        CnvUtfConverter::ConvertToUnicodeFromUtf8( buf, ptr ) ;
#7        RLibrary* lib = new (ELeave) RLibrary();
#8        err = lib->Load( buf );
#9        if( err != KErrNone )
#10       {
#11         delete lib;
#12         lib = NULL;
#13       }
#14       return lib;
#15     }
#16     void UnLoadSymbianLib( void* lib )
#17     {
#18       RLibrary* rlib = (RLibrary*)lib;
#19       rlib->Close();
#20       delete rlib;
#21       rlib = NULL;
#22     }
#23     void* LoadSymbianLibFunction( void* lib )
#24     {
#25       RLibrary* rlib = (RLibrary*)lib;
#26       TLibraryFunction libfunc = rlib->Lookup( 1 );
#27       return libfunc;
#28     }
```

Listaus 11 Lua-kirjastojen dynaaminen lataaminen

Symbianissa voidaan käyttää dynaamisten kirjastojen lataamiseen *RLibrary*-luokkaa. *RLibrary* luodaan perinteisestä Symbian-ohjelmoinnista poiketen dynaamisesti (R-luokat luodaan staattisesti), koska muutoin sitä on hankala välittää muihin funktioihin. Yllä olevassa listauksessa Lua-laajennus ladataan *LoadSymbianLib*-funktiosta, jota kutsutaan Luan standarditoteutuksesta (Loadlib.c). Luan standarditoteutus vaatii myös pieniä muutoksia tämän suhteen.

LoadSymbianLibFunction-funktiolla ladataan Lua-laajennuksesta Luan alustusfunktio. Tämä alustusfunktio tallettaa *lua_State*-tietorakenteeseen kaikki laajennuksessa olevat Lua-funktiot. Huomattavaa *LoadSymbianLibFunction*-

funktiossa on se, että Luan alustusfunktion oletetaan olevan ensimmäinen rajapintafunktio kirjastossa, kuten rivillä 26 tehdään.

Lua-laajennus suljetaan kutsumalla *UnLoadSymbianLib*-funktiota. Tällöin *RLibrary*-olio sulkee kirjaston, sekä vapauttaa sen muistista.

Muutokset Luan lähdekoodeihin

Luan perustoteutukseen tehtiin muutamia muutoksia, jotta Lua saatiin toimimaan Symbian-käyttöjärjestelmässä. Suurimmat muutokset tulivat luaconf.h, loadlib.c, sekä loslib.c:hen. Kaikki muutokset erotettiin *LUA_SYMBIAN*-käännösmakrolla, joten Symbian-kohtaiset muutokset eivät poissulkeneet käännöstä muille käyttöjärjestelmille.

Luaconf-otsikkotiedon muutokset

Luaconf-otsikkotiedosto sisältää Luan käyttämiä vakioasetuksia, kuten muuttujien tarkemmat tyypitykset, käytettävät hakemistopolut ja niiden merkintä, sekä kaikki Luan käyttämät vakioarvot. Seuraavaksi tutustumme näihin lisäyksiin tarkemmin.

```
...
#elif defined(LUA_SYMBIAN)
#define LUA_LDIR  "e:\\Lua\\"
#define LUA_CDIR  "c:\\sys\\bin\\"
#define LUA_PATH_DEFAULT  \
            LUA_LDIR"?.lua;"
#define LUA_CPATH_DEFAULT \
        LUA_CDIR"?.dll;"
...
#elif defined(LUA_SYMBIAN)
#define LUA_API EXPORT_C
```

Listaus 12 Muutoket luaconf.h-tiedostoon

Edellisessä listauksessa on esitelty mistä hakemistopoluista Lua etsii Lua-laajennuskirjastoja, tai -skriptejä, sekä määritellään Lua käyttämään Symbianissa

käytettyä *EXPORT_C*-makroa, joka mahdollistaa funktioiden kutsumisen DLL:n ulkopuolelta. Tässä toteutuksessa siis oletetaan, että kaikki Lua-laajennukset sijaitsevat laitteen C-aseman *sys\bin*-hakemistossa, sekä Lua skriptit sijaitsevat E-aseman *Lua*-kansiossa. Näin ollen, kun Lua-skriptissä käytetään esimerkiksi *require*–komentoa, niin tälle ei tarvitse määritellä erikseen hakemistopolkua.

Loadlib.c:n muutokset

Loadlib-moduuli vastaa Lua-laajennusten ja -skriptien dynaamisesta lataamisesta. Tästä modulista suoritetaan kutsuja jo aiemmin tässä luvussa esiteltyyn Symbian-moduuliin, joka lataa laajennukset.

```
...
#elif defined(LUA_SYMBIAN)
extern void* LoadSymbianLib( const char* path );
extern void UnLoadSymbianLib( void* lib );
extern void* LoadSymbianLibFunction( void* lib );

static void ll_unloadlib( void* lib )
{
  UnLoadSymbianLib(lib);
}

static void *ll_load( lua_State*, const char* path )
{
  return LoadSymbianLib(path);
}

static lua_CFunction ll_sym( lua_State*, void* lib, const char* )
{
  lua_CFunction func =
    (lua_CFunction)LoadSymbianLibFunction(lib);
  return func;
}

...
```

```
#if defined (LUA_SYMBIAN)
static int loader_Symbian( lua_State* L )
{
  const char* func = NULL;
  const char* name = luaL_checkstring(L, 1);
  char libname[50];
  char* ptr = &libname[0];
  strcpy( ptr, name );
  if( ll_loadfunc( L, strcat( ptr, ".dll" ), func ) != 0 )
  {
    return 0;
  }
  return 1;
}
#else
...
#if defined(LUA_SYMBIAN)
static const lua_CFunction loaders[] =
  {loader_preload, loader_Lua, loader_Symbian, NULL};
#else
static const lua_CFunction loaders[] =
  {loader_preload, loader_Lua, loader_C, loader_Croot, NULL};
#endif
```

Listaus 13 Muutokset loadlib.c-tiedostoon

Yllä olevassa listauksessa on määritelty miten Lua-laajennukset ladataan. Listauksen alimmaisena luodaan *loaders*-taulukko, johon on määritelty funktio-osoittimet. Näitä funktio-osoittimia voidaan käyttää Lua-laajennusten lataamiseen. Näitä käyttämällä Lua osaa kutsua oikeita funktioita, jotta saadaan haluttun tyyppinen Lua-laajennus dynaamisesti ladattua.

loader_Symbian-funktio hakee käyttäjän antaman Lua-laajennuksen nimen, sekä lisää tämän merkkijonon perään ”.dll”-merkkijonon, jotta kirjasto voitaisiin ladata. Tämän jälkeen Lua laajennuksen nimi välitetään *ll_loadfunc*-funktioon, joka lopulta välittää kutsun Symbian-lataajaan.

loslib.c:n muutokset

loslib.c on moduuli, joka sisältää käyttöjärjestelmästä saatavaa tietoa, kuten esimerkiksi ajan ja päivämäärän, ympäristömuuttujia (mikäli käyttöjärjestelmässä niitä käytetään), sekä mahdollistaa myös sovellusten käynnistämisen laitteelta.

Symbian 9.1-versioon tulleen turvallisuusmuutoksen myötä, Symbanin ANSI C-kirjasto ei tue kaikkea OS-kirjaston toiminnallisuutta. Ongelmia aiheuttaa etenkin *remove*-funktio, jonka tarkoitus on poistaa levyltä tiedosto. Tätä funktiota ei ole tuettu Symbian 9.1 ANSI C-kirjastossa, joten se täytyy korvata. Tähän versioon ei vielä tullut kyseiseen funktioon korjausta, vaan se korvattiin virheilmoituksella, mikäli sitä kutsutaan.

4.5 Lua-siirron testaus

Luan siirron testaamiseen Symbian käyttöjärjestelmässä käytettiin LuaLaunc-konsollisovellusta, joka käytti Luan Symbian C++-rajapintaa. Konsollisovelluksen kautta käynnistettiin ylensä jokin Lua-skripti, jonka avulla pystyttiin testaamaan Luan ominaisuuksia. Tässä kappaleessa tutustumme muutamaan testattavaan kohteeseen tarkemmin.

Symbian C++-rajapinnan testaaminen

Symbian C++-rajapinnan testaamisessa käytetään LuaLaunch-konsollisovellusta. Tämä sovellus käyttää *CLuaInterpreter*-tyyppistä oliota, jonka avulla voidaan ajaa Lua-skripti.

```
#1      #include <stdio.h>
#2      #include <stdlib.h>
#3      #include <sys/reent.h>
#4      #include "LuaLaunch.h"
#5      #include "LuaInterpreter.h"
#6      LOCAL_D CConsoleBase* console; // debug messages
#7      LOCAL_C void MainL()
#8      {
#9       console->Write(_L("Lua Test\n"));
#10       CLuaInterpreter* it = CLuaInterpreter::NewL();
#11       console->Write(_L("Success, Testing...\n"));
#12       TRAPD( err,
#13       it->RunFileL( _L8("c:\\lua\\test.lua") );
#14       );
#15       if( err == KErrNotFound )
#16       console->Write(_L("test file not found\n"));
#17       else if( err == KErrNone )
#18       console->Write(_L("test file launched\n"));
#19       else
#20       console->Write(_L("error launching test file\n"));
#21       delete it;
#22     }
#23     GLDEF_C TInt E32Main()
#24     {
#25       __UHEAP_MARK;
#26       CTrapCleanup* cleanup = CTrapCleanup::New();
#27       TRAPD(createError,
#28         console = Console::NewL(KTextConsoleTitle,
#29         TSize(KConsFullScreen,KConsFullScreen))
#30       );
#31       if( createError )
#32         return createError;
#33       TRAPD( mainError, MainL() );
#34       if( mainError )
#35         console->Printf(KTextFailed, mainError);
#36       console->Printf(KTextPressAnyKey);
#37       console->Getch();
#38       delete console;
#39       delete cleanup;
#40       __UHEAP_MARKEND;
```

```
#41         return KErrNone;
#42     }
```

Listaus 14 Luan Symbian C++-rajapinnan testaaminen

Konsollisovellus on yksinkertainen ajettava sovellus, jossa on vähän riippuvaisuuksia järjestelmän muihin osiin. Näin ollen sovellus soveltuu parhaiten Luan Symbian C++-rajapinnna testaamiseen.

Sovelluksen suoritus alkaa *E32Main*-funktiosta, joka on Symbian-sovellusten pääohjelma. Testissä käytetään debug-tulostuksiin Symbianin tarjoamaa *CConsoleBase*-tyyppistä oliota, jonka avulla voidaan kirjoittaa näytölle tulostuksia ohjelman suorituksesta.

Testissä luodaan *CLuaInterpreter*-tyyppinen olio, joka kapseloi Luan rakenteen. Tämän jälkeen pyydetään *CLuaInterpreter*-oliota ajamaan *RunFileL*-metodilla testattava Lua-skripti laitteen C-levyn *Lua*-kansiosta. Mikäli *RunFileL*-metodi aiheuttaa poikkeuksen, niin *CLuaInterpreter*-olio ei pystynyt ajamaan Lua-skriptiä.

Tätä testisovellusta käytettiin yleisesti useiden Lua-skriptien ajamiseen.

Lua-laajennusten testaaminen

Lua-laajennukset ovat siis kirjastoja, jotka ladataan Luassa dynaamisesti. Tämä testi muodostuu kahdesta osasta, jossa ladataan dynaamisesti joko Lua-skripti, tai Lua C++-kirjasto laitteen levyltä. Testiskriptissä ladataan molemmat kirjastot, sekä kutsutaan funktioita niistä.

```
#1      module( "req", package.seeall )
#2      function ReqTest( val1, val2 )
#3          return val1 + val2
#4      end
```

Listaus 15 Ladattava Lua-skriptikirjasto

Ladattava Lua-skriptikirjasto on melko yksinkertainen. Skriptissä pitää vain määritellä, että kyseessä on Lua-kirjasto *module*-funktiolla. Tämän jälkeen määritellään kirjaston funktiot, joita tässä kirjastossa on vain yksi. *ReqTest*-funktio palauttaa yhteen laskettuna sille annetut kaksi parametriä. Jotta Lua osaisi ladata tämän skriptin, sen pitää sijaita E-aseman *Lua*-kansiossa.

```
#1      #include <e32base.h>
#2      #include <stdio.h>
#3      #include <stdlib.h>
#4      #include <sys/reent.h>
#5      extern "C" {
#6      #include "lua.h"
#7      #include "lauxlib.h"
#8      #include "lualib.h"
#9      }
#10     #include "LuaExtension.h"
#11
#12     static int l_dir( lua_State* L )
#13     {
#14         double d = luaL_checknumber( L, 1 );
#15         lua_pushnumber( L, d );
#16         return 1;  /* number of results */
#17     }
#18     static const struct luaL_reg mylib [] = {
#19         { "dir", l_dir },
#20         { NULL, NULL }  /* sentinel */
#21     };
#22     EXPORT_C int luaopen_mylib( lua_State* L )
#23     {
#24         luaL_openlib( L, "mylib", mylib, 0 );
#25         return 1;
#26     }
```

Listaus 16 Ladattava Lua laajennus

Yllä olevassa listauksessa on yksinkertainen Lua-laajennus, jossa on vain yksi Lua-skriptistä kutsuttava *l_dir*-funktio. Lua-laajennuksen lataaminen toimii siten, että ensin Luasta tulee kutsu *luaopen_mylib*-funktioon, joka on laajennuksen rajapintafunktio Lualle. Tämä funktio välittää Lualle *mylib*-taulukon, jossa on määritelty Luasta kutsuttavat funktiot. Esimerkissä niitä on vain *l_dir*-funktio, joka palauttaa parametrina saamansa numeron.

```
#1      require "req" -- lua script
#2      require "LuaExtension" -- lua c extension module
#3      val = req.ReqTest( 2, 3 )
#4      print( val ) -- 5
#5      val = mylib.dir( 2 )
#6      print( val ) -- 2
```

Listaus 17 Molemmat kirjastot lataava testiskripti

Testiskriptissä siis ladataan Lua-skriptitiedosto, sekä Lua C-laajennus käyttämällä *require*-komentoa. Tämän jälkeen kirjastoista voidaan kutsua niiden tarjoamia funktioita.

5 ASYNKRONINEN OHJELMOINTI LUASSA

Edellisessä luvussa käsiteltiin kuinka Luan perustoiminnallisuus saadaan toimimaan Symbian-käyttöjärjestelmässä, sekä kuinka sitä voidaan testata ajamalla Lua-skripti konsollisovelluksesta. Tässä luvussa käsitellään, miten Luan perustoteutusta voidaan laajentaa käyttämään Symbianin asynkronisia palveluita.

5.1 Asynkronisuus Symbian-OS:ssä

Kutsuttaessa järjestelmästä jotakin palvelua, sen käsittely voi tapahtua kahdella eri tavalla: joko synkronisesti, tai asynkronisesti.

Synkroonisessa ohjelmoinnissa järjestelmä käsittelee ja suorittaa saadun kutsun välittömästi. Kun järjestelmä on käsitellyt pyynnön, ohjelman suoritus palaa takaisin siihen funktioon, mistä palvelua kutsuttiin. Tällöin yleensä järjestelmä palauttaa jonkin virhekoodin, mikä ilmaisee onnistuiko operaation suoritus.

Asynkronisessa ohjelmoinnissa järjestelmä rekisteröi palvelupyynnön, muttei aloita sen suorittamista välittömästi. Tällöin ohjelman suoritus palaa välittömästi takaisin palvelua pyytäneeseen ohjelmaan. Asynkronisen ohjelmoinnin suurin ero synkroniseen ohjelmointiin on siinä, miten järjestelmä ilmaisee saaneensa tehtävän päätökseen. Asynkronisessa ohjelmoinnissa järjestelmä signaloi suorituksen päätyttyä palvelua pyytäneen ohjelman tapahtumankäsittelijäfunktiota (*Event*

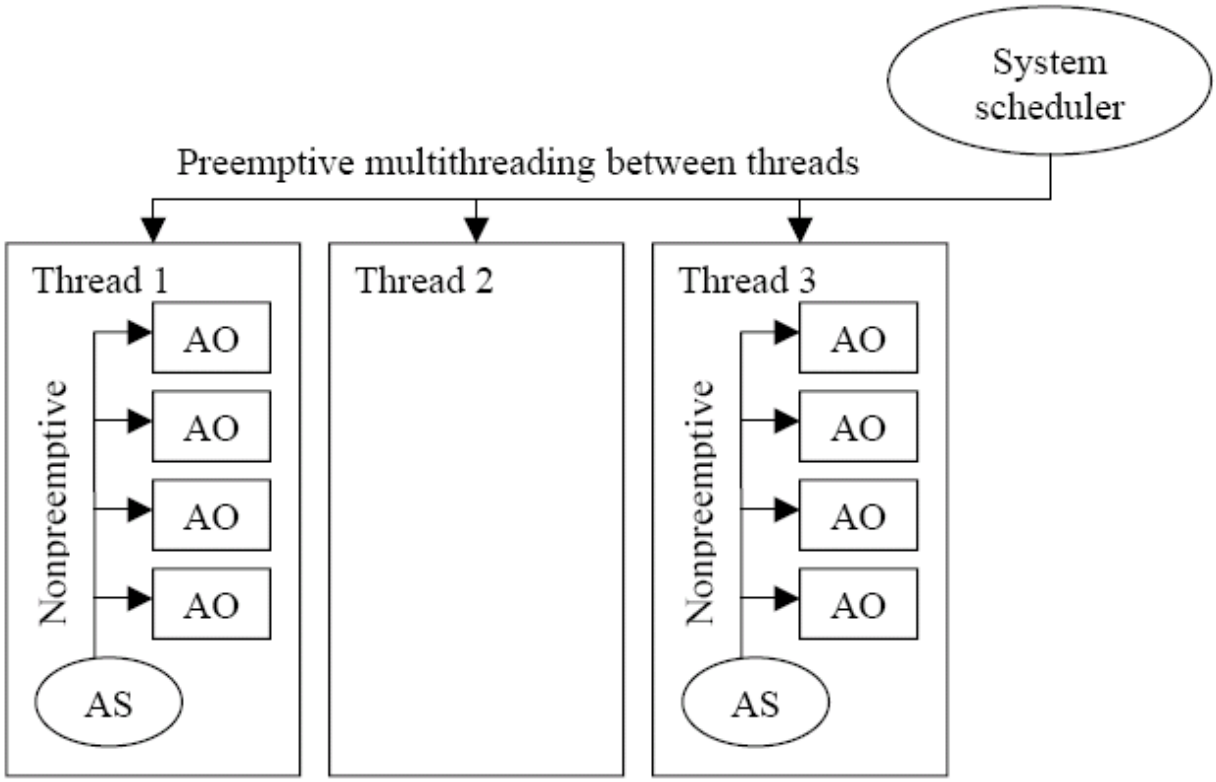
handler).

Asynkronisen ohjelmoinnin suurin etu on pavelun pyytäjän kyky toimia myös silloin, kun se on lähettänyt järjestelmälle pyyntöjä, joiden suoritus kestää kauan. Haittapuolena asynkronisessa ohjelmoinnissa on ohjelman monimutkaistuminen, sekä koodin rivien määrän moninkertaistuminen.

Usein synkronissa järjestelmissä käytetään säikeistystä suorittamaan pitkäkestoisia pyyntöjä. Tällöin ohjelman pääsäie pystyy toimimaan myös silloin, kun järjestelmä suorittaa useita raskaita operaatioita. Tosin säikeiden käyttö lisää myös ohjelman monimutkaisuutta huomattavasti.

Asynkroonisessa ohjelmoinnissa suoritetaan pitkäkestoiset pyynnöt ja niiden käsittely yhdessä säikeessä. Tällöin ohjelma asettuu odotus tilaan, kun pyynnöt on välitetty järjestelmään. Tämän jälkeen järjestelmä herättää ohjelman, kun jokin pyynnöistä on saatu päätökseen. Näin ollen asynkroonista ohjelmointitapaa suositellaan Symbian-ohjelmoinnissa käytettäväksi säikeistyksen sijaan.

Symbian OS:ssä käytetään yleisesti asiakas- ja palvelinarkkitehtuuria (*Client, Server*). Tällöin asiakas ja palvelin toimivat omissa proseississaan, jolloin käyttöjärjestelmä välittää viestejä osapuolten välillä.



Kuva 5 Aktiivioliot ja aktiivisvuorotin Symbianissa /14/

Symbianissa asiakas käyttää aktiiviolioita (kuvassa 5 AO) asynkronisiin toimintoihin. Aktiiviolio tarjoaa vähintään metodit aktivointiin, peruuttamiseen, sekä tapahtumankäsittelijän pyynnön päätyttyä. Aktiiviolio käyttää kahvoja (eli ns. R-luokkia) kommunikoidakseen palvelimen kanssa. Symbianissa käytetään aktiivivuorotinta (kuvassa 5 AS), jotta saadaan tieto palvelun käyttäjälle tapahtumankäsittelyn päätöksestä.

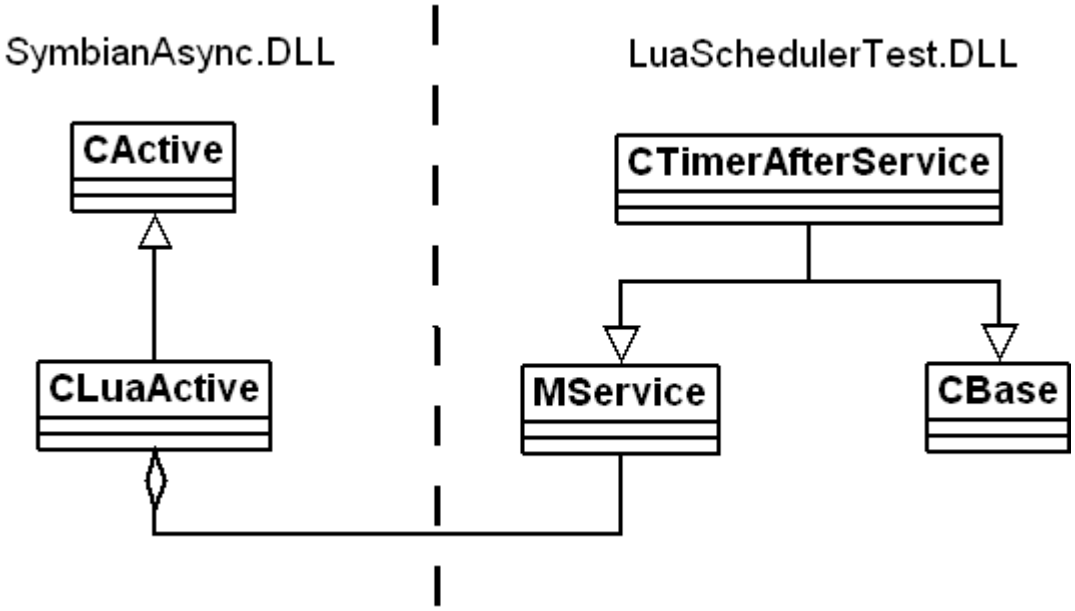
Aktiivivuorottimen tehtävä on ylläpitää tietoa, mitkä aktiivioliot ovat välittäneet pyynnön asiakkaalle, sekä mitkä pyynnöt ovat päättyneet. Aktiivivuorottimen toiminta perustuu jatkuvaan palvelun tarjoajien tilojen tarkkailuun (*Event Loop*). Kuvan 5 mukaan, aktiivivuorotin on säiekohtainen Symbianissa. Aktiivioiliot vaativat toimiakseen aktiivivuorottimen. Kuvan 5 mukaisesti kaikissa säikeissä ei ole pakko olla aktiivivuorotinta.

Lua on suunniteltu synkroniseksi ohjelmointikieleksi, joten siihen ei ole toteutettu tukea asynkroniseen ohjelmointiin. Luassa voidaan tosin käyttää aktiivioliota, mutta silloin ei voida koskaan saada takaisinkutsua aktiiviolioiden tapahtuman käsittelijään, koska Lua-skripti ei jää odottamaan pyydettävän asynkroonisen toiminnon päättymistä. Tämän ongelman ratkaisemiseen käytetään hyväksi

Symbianin aktiivivuorotinta.

5.2 Asynkronisen ohjelmointi tuen toteutus Luaan

SymbianAsync-Lua-laajennuksella voidaan luoda aktiivioloita, sekä käyttää aktiivivuorotinta Lua-skriptien kautta. Aktiivioliolla on metodit aktivointiin, peruuttamiseen, sekä takaisinkutsu funktion asettamiseen. Aktiivivuorottimen voi vain käynnistää, tai pysäyttää.



Kuva 6 Aktiivioliotuen mallintaminen Luaan

Aktiivioliotuki on mallinnettu Luaan kuvan 6 esimerkin mukaisesti, jossa SymbianAsync-kirjastossa rakennetaan *CLuaActive*-aktiiviolio. Tälle välitetään *MService*-tyyppinen olio, jossa on kahva asynkroniseen palveluun. *MService*-luokka on abstrakti kantaluokka, jossa on määriteltynä metodit vain aktivointiin, sekä peruuttamiseen. *MService*-luokka on näin ollen geneerinen rajapinta luokkaan, jossa on toteutettu kahva asynkrooniseen palveluun. Kuvan 6 esimerkissä käytetään *CTimerAfterService*-oliota kapseloimaan kahva asynkroonisen ajastimen toimintaan.

Aktiiviolioita Luassa käyttetään seuraavasti:

1. Aktiiviolio-, sekä asynkronisen kahvan toteuttavien Lua-laajennuksien lataaminen Lua-skriptissä *require*-komennolla.
2. *MService*-olion luominen. Tällöin myös alustetaan kahvat asynkroonisiin palveluihin.
3. *MService*-olion välittäminen *CLuaActive*-oliolle.
4. *CLuaActive*-olion alustaminen takaisinkutsu funktiolla.
5. *CLuaActive*-olion aktivointi.
6. Aktiivivuorottimen käynnistäminen.
7. Tapahtumankäsittely, kun pyyntö on suoritettu järjestelmässä.
8. Aktiivivuorottimen pysäyttäminen, kun kaikki pyynnöt on suoritettu.

Asynkronisen ohjelmoinnin toteutus Luassa

SymbianAsync-laajennuksessa on Lua-aktiiviolion toteutus, sekä funktiot aktiivivuorottimen käyttöön.

SymbianAsync-laajennuksen alustus

Lua-aktiivioliolioiden on oltava tyypiltään Userdataa, jotta niitä voitaisiin luoda Lua-laajennuksista ja välittää Lua-skriptin kautta. Userdatalla on oltava myös metatauluja, jotka mahdollistavat useiden Lua-olioiden luomisen.

```
#1      static const struct luaL_reg mylib [] = {
#2        { "LuaActive", LuaActive },
#3        { "Start", Start },
#4        { "Stop", Stop },
#5        {NULL, NULL}  /* sentinel */
#6      };
#7      static const struct luaL_reg LuaActiveMethods [] = {
#8        { "Callback", Callback },
```

```
#9        { "Queue", Queue },
#10       { "Cancel", Cancel },
#11       { "__gc", FreeLuaActive },
#12       { NULL, NULL }  /* sentinel */
#13     };
#14     EXPORT_C int luaopen_mylib( lua_State* L )
#15     {
#16       luaL_newmetatable( L, KObjectName );
#17       lua_pushstring( L, "__index" );
#18       lua_pushvalue( L, -2 ); /* pushes the metatable */
#19       lua_settable( L, -3 ); /* metatable.__index = metatable*/
#20       luaL_openlib( L, NULL, LuaActiveMethods, 0 );
#21       CreateScheduler( L );
#22       luaL_openlib( L, "SymbianAsync", mylib, 0 );
#23       return 1;
#24     }
```

Listaus 18 SymbianAsync-laajennuksen alustus (*SymbianAsync.cpp*)

Yllä oleva listauksessa alustetaan SymbianAsync-laajennus Lua-skriptissä käytettäväksi.

Laajennuksen lataaminen alkaa riviltä 14 *luaopen_mylib*-funktiosta, kun Lua-skriptissä pyydetään *require*-komennolla Lua-laajennusta ladattavaksi. Funktiossa rekisteröidään Lua-olion metodit *LuaActiveMethods*-taulukosta, sekä Lua-laajennuksesta normaalisti kutsuttavat funktiot *mylib*-taulukosta.

Lua-olio rekisteröidään rivillä 16 käyttämällä Luan API:n *luaL_newmetatable*-funktiota. Funktio luo metataulun Luan pinon päälimmäiseksi ja yhdistää annetun nimen (*KobjectName* = ”*CLuaActive.array*”) Luan rekistereihin. Kun metataulu on rekisterissä, siihen voidaan myöhemmin viitata *luaL_getmetatable*-funktiolla. Riveillä 18 ja 19 alustetaan metaulun indeksit.

Rivillä 20 ladataan Lua-olion metodit, jotka on määritelty LuaActiveMethods-taulukossa. Lua olio tarjoaa metodit takaisinkutsufunktion asettamiseen (*Callback*), aktiiviolion pyynnön aktivointiin (*Queue*), aktivointipyynnön peruuttamiseen (*Cancel*), sekä purkajametametodin (*__gc*). Lua kutsuu purkaja metametodia Luan roskien keruu moduulista, kun skriptin suoritus on päättynyt. Huomioitavaa *luaL_openlib*-funktio kutsussa on myös toiseksi parametriksi välitetty *NULL*-osoitin. Tällöin Lua ei luo taulukkoa, jossa Luan käyttämät funktiot ovat.

Rivillä 21 kutsutaan kirjaston sisäistä *CreateScheduler*-funktiota. Tämä funktio luo aktiivivuorottimen säikeen käyttöön.

Rivillä 22 ladataan SymbianAsync-laajennuksen tarjoamat muut funktiot *mylib*-taulukosta.

Aktiiviolion luominen

Aktiiviolion luominen on toteutettu SymbianAsync-laajennuksen perusfunktioissa. Kahva asynkrooniseen palveluun (*MService*-olio) on välitettävä aktiiviolion rakentamisen yhteydessä. *MService*-olion välittämiseen Luan kautta Userdatana käytetään *TServiceUserdata*-luokkaa. Se sisältää vain osoittimen oikeaan olioon. *CLuaActive*-olion välittämiseen käytetään *TLuaActiveUserdata*-luokkaa, jolla on sama käyttötarkoitus, kuin *TServiceUserdata*-luokalla.

```
#1      static int LuaActive( lua_State* L )
#2      {
#3        if( lua_isuserdata( L, 1 ) )
#4        {
#5          TServiceUserdata* service;
#6          service = (TServiceUserdata*)lua_touserdata( L, 1 );
#7          if( !service )
#8          {
#9            return 0;
#10         }
#11         TLuaActiveUserdata* udata;
#12         udata = (TLuaActiveUserdata*)lua_newuserdata( L,
#13           sizeof(TLuaActiveUserdata) );
#14         luaL_getmetatable( L, KObjectName );
#15         lua_setmetatable( L, -2 );
#16         udata->iLuaActive = new (ELeave)
#17           CLuaActive( service->iService );
#18         return 1;
#19       }
#20       return 0;
#21     }
```

Listaus 19 Aktiiviolion luominen (*SymbianAsync.cpp*)

Rivillä 3 tarkistetaan, onko parametriksi annettu Userdataa. Mikäli ei ole, niin aktiivioliota ei luoda ja mitään ei palauteta Lua skriptiin. Jos annettu parametri on oikein, niin rivillä 6 haetaan Luan pinosta Userdata ja tehdään tyyppimuunnos *TserviceUserdata*:ksi.

Kun osoitin asynkroniseen palveluun on saatu, voidaan luoda *TLuaActiveUserdata*-olio, sekä asettaa se metatauksi. Kun *TLuaActiveUserdata* on alustettu, voidaan dynaamisesti luoda *CLuaActive*-aktiivioliо. Tälle aktiivioliolle välitetään rakentajan yhteydessä osoitin *MService*-olioon riveillä 16 – 17.

Takaisinkutsufunktion asettaminen

Takaisinkutsufunktio Luaan pitää asettaa, jotta aktiivioliota voitaisiin hyödyntää Lua-skriptissä.

```
#1      static int Callback( lua_State* L )
#2      {
#3        TLuaActiveUserdata* udata;
#4        udata = (TLuaActiveUserdata*)luaL_checkudata( L, 1,
#5          KObjectName );
#6        lua_remove( L, 1 );
#7        if( lua_isfunction( L, 1 ) )
#8        {
#9          udata->iLuaActive->SetLuaRef( luaL_ref( L,
#10           LUA_REGISTRYINDEX ) );
#11         udata->iLuaActive->SetLuaState( L );
#12       }
#13       return 0;
#14     }
```

Listaus 20 Takaisinkutsufunktion asettaminen (*SymbianAsync.cpp*)

Takaisinkutsufunktion asettaminen aloitetaan tarkistamalla Luan API:n *luaL_checkudata*-funktiolla, onko Luasta välitetty ensimmäisenä parametri Userdataa. *luaL_checkudata*-funktio aiheuttaa suorituksen keskeytyksen ja tulostaa virheilmoituksen, mikäli parametrina ei ole oikean tyyppinen olio.

Mikäli toinen funktioon välitetty parametri on Lua-funktio (rivi 7), voidaan se asettaa Luan aktiivioliolle. Lua-funktion asettamiseen käytetään hyväksi Luan

rekistereitä. Tällöin skriptistä parametriksi välitetty Lua-funktio siirretään rekisteriin ja viite rekisteristä talletetaan aktiiviolion muistiin (rivi 9). Osoitin *lua_State*-tietorakenteeseen pitää myös asettaa, jotta takaisinkutsu voitaisiin suorittaa aktiiviolion tapahtuman käsittelyssä (rivi 11).

Asynkronisen kahvan toteuttaminen

Seuraavassa esimerkissä käytetään ajastinta (*RTimer*-luokka) kahvana asynkroniseen palveluun. Kahvan asynkroniseen palveluun kapseloi *CTimerAfterService*-luokka. Tämän toteutus on erillisessä Lua-laajennuksessa ja se pitää ladata Lua-skriptissä *require*-komennolla. *CTimerAfterService*-luokka on periytetty *MService*-luokasta, joten luokassa pitää olla toteutettuna aktivointi (*Queue*), sekä peruuttaminen (*Cancel*).

```
#1      #include <e32base.h>
#2      #include <stdio.h>
#3      #include <stdlib.h>
#4      #include <sys/reent.h>
#5      #include <string.h>
#6      #include "LuaSchedulerTest.h"
#7      extern "C" {
#8      #include "lua.h"
#9      #include "lauxlib.h"
#10     #include "lualib.h"
#11     }
#12     CTimerAfterService::CTimerAfterService()
#13     {
#14       iTimer.CreateLocal();
#15     }
#16
#17     CTimerAfterService::~CTimerAfterService()
#18     {
#19       Cancel();
#20     }
#21
#22     void CTimerAfterService::Queue( lua_State* aLuaState,
#23       TRequestStatus* aStatus )
```

```
#24     {
#25       iTimer.After( *aStatus, 3000000 );
#26     }
#27
#28     void CTimerAfterService::Cancel()
#29     {
#30       iTimer.Cancel();
#31     }
#32
#33     static int TimerAfterService( lua_State* L )
#34     {
#35       TServiceUserdata* udata;
#36       udata = (TServiceUserdata*)lua_newuserdata( L,
#37         sizeof(TServiceUserdata) );
#38       udata->iService = new (ELeave) CTimerAfterService;
#39       return 1;
#40     }
#41
#42     static const struct luaL_reg mylib [] = {
#43       {"TimerAfterService", TimerAfterService },
#44       {NULL, NULL}  /* sentinel */
#45     };
#46
#47     EXPORT_C int luaopen_mylib (lua_State *L)
#48     {
#49       luaL_openlib(L, "LuaSchedulerTest", mylib, 0);
#50       return 1;
#51     }
#52
```

Listaus 21 Asynkronisen ajastinpalvelun toteuttaminen (*LuaSchedulerTest.cpp*)

Yllä olevassa listauksessa on Lua-laajennuksen toteutus, joka tarjoaa vain funktion ajastinkahvan (*TimerAfterService*) luomiseen.

Laajennuksen lataamien alkaa riviltä 47, kun Lua-skriptissä on pyydetty *require*-komennolla LuaSchedulerTest-laajennusta. Laajennuksesta ladataan vain *TimerAfterService*-funktio Luan kutsuttavaksi (rivi 49).

TimerAfterService-funktio luo uuden *CTimerAfterService*-olion, jossa on toteutettu kahva asynkroniseen palveluun (rivit 33 – 40).

Yleensä asynkronisten palveluiden kahvoihin, eli R-luokkiin, liittyy alustustoimenpiteitä. Esimerkissä käytettävässä *RTimer*-kahvassa alustuksessa kutsutaan olion *CreateLocal*-metodia (rivi 14), jossa luodaan säiekohtainen ajastin. Tämän jälkeen kahva voitaisiin normaalisti aktivoida, mutta Luan asynkroonisessa tuessa on vielä muita alustustoimenpiteitä (esim. takaisinkutsufunktion asettaminen).

Kun kaikki alustustoimenpiteet on suoritettu, kutsutaan kahvan aktivointi funktiota (*Queue*) riveillä 22 – 26. Kahvan aktivointiin pitää yleensä välittää *TRequestStatus*-olio. Tätä oliota käytetään yksilöimään aktiiviolio, jotta aktiivivuorotin osaisi kutsua oikeata aktiivioliota pyynnön päätyttyä. Funktioon välitetään myös *lua_State*-tietorakenne, jotta voitaisiin mahdollisesti käyttää hyväksi Lua-skriptistä välitettyjä muita parametrejä. Esimerkissä ajastin on kuitenkin kovakoodattu käynnnistymään kolmen sekunnin kuluttua.

5.3 Testaus Lua-skriptissä

Asynkroninen ohjelmointi Luassa toimii samoin periaattein, kun Symbian C++:ssa. Ensimmäiseksi välitetään yksi tai useampi pyyntö järjestelmälle asynkronisesta palveluista, jonka jälkeen käynnistetään aktiivivuorotin. Aktiivivuorotin ilmoittaa koska pyyntö on saatu päätökseen, jolloin tehdään takaisinkutsu Lua-funktioon.

Alla olevassa esimerkissä käytetään ajastinta asynkroonisena palveluna.

```
#1      require "SymbianAsync"
#2      a = require "LuaSchedulerTest"
#3      function test()
#4        print "lua callback function works!"
#5        SymbianAsync.Stop()
#6      end
#7      service = a.TimerAfterService()
#8      la = SymbianAsync.LuaActive( service )
#9      la:Callback( test )
#10     la:Queue()
#11     SymbianAsync.Start()
```

Listaus 22 Asynkroonisuuden käyttö Lua skriptissä

Asynkronisuuden tuki Luaan on toteutettu SymbianAsync-Lua-laajennukseen,

jolloin Luan perustoteutus pysyy muuttumattomana. Esimerkissä käytetään LuaSchedulerTest-Lua-laajennusta, johon on toteutettu kahva käytettävään ajastimeen.

Rivillä 3 on määritelty Lua-funktio, jota kutsutaan, kun pyyntö on suoritettu.

Rivillä 7 luodaan Lua-olio, jossa on kahva ajastimeen.

Rivillä 8 luodaan Lua-aktiiviolio *la*, joka saa parametriksi *service*-olion. Tällöin aktiiviolio on alustettu kahvalla asynkroniseen palveluun. Aktiiviolio pitää myös alustaa takaisinkutsufunktiolla (*test*) ennen kuin palvelua voidaan pyytää. Tämän jälkeen voidaan kutsua tätä funktiota, kun asynkroninen operaatio on päättynyt.

Kun alustustoimenpiteet on tehty, voidaan välittää rivillä 10 aktivointikomento *la*-oliolle. Mikäli tämän jälkeen ei enään välitetä muita aktivointikomentoja, aktiivivuorotin voidaan käynnistää. Tällöin ohjelma menee odotus tilaan, kunnes järjestelmä on saanut pyynnön päätökseen.

Kun pyyntö on käsitelty ja suoritettu, niin välittyy takaisinkutsu *test*-funktioon. *Test*-funktiossa pysäytetään aktiivivuorotin, jolloin ohjelma voi päättyä.

6 YHTEENVETO

Tämän työn tarkoituksena oli selvittää, miten Lua-ohjelmointikieli voitaisiin siirtää Symbian-käyttöjärjestelmään. Työ keskittyi lähinnä Luan perustoiminnallisuuden, sekä asynkroonisen ohjelmointitavan siirtämiseen Symbianiin. Siirto ei ollut ajan puutteen vuoksi täydellinen, mutta se tukee jo monia Luan ominaisuuksia ja toimii sellaisenaan laitteessa ja emulaattorissa. Myös Luan perustoiminnallisuuden kattava testaaminen jäi vähäiseksi ja testaus keskittyi lähinnä muutamiin oleellisimpien ominaisuuksiin, kuten tiedostonkäsittelyyn, laajennusten lataamiseen ja taulukon/olioiden testaukseen.

Luassa olisi mahdollista valita poikkeuksien käsittelymakrot, mutta Luan perustoiminnallisuudesta jäi toteuttamatta tuki Symbianin poikkeuksien hallintaan.

Myös asynkronisen ohjelmoinnin tuki jäi vielä hieman kesken, vaikka tarjoaakin jo toimivan kehyksen asynkroniseen ohjelmointiin Lua-skripteissä. Tukea on vielä tarkoitus laajentaa käyttämään korutiineja. Korutiinit tulisivat nykyisen toteutuksen

yläpuolelle. Tällöin asynkroonisen operaation aktivointi keskeyttäisi korutiinin suorituksen. Korutiinin suoritus jatkuisi, kun asynkrooninen operaatio on suoritettu järjestelmässä.

Asynkronisen ohjelmoinnin tuessa virheiden käsittely on myös hieman puutteellista. Tämä johtuu osin siitä, että tukea Symbianin poikkeuskäsittelyyn ei ole.

Luaan on myös suunnitteilla parempi tuki C/C++-kirjastojen käyttämiseen. Tässä versiossa C/C++-koodia voidaan käyttää vain Lua-laajennusten kautta. On myös olemassa sovelluksia, mitkä tarjoavat rajapinnan kutsua mitä tahansa C/C++-kirjastoa Luasta, kuten esimerkiksi C/Invoke.

Kaiken kaikkiaan työ osoittaa sen, että Luan toteutus on mahdollista siirtää Symbianiin melko vähäisellä muutostyöllä. Työtä aiheuttaa ehkä eniten perehtyminen Luan arkkitehtuuriin, sekä itse ohjelmointikieleen ja sen erityispiirteisiin. Luan toteutus on kuitenkin monimutkainen ja sen omaksuminen vaatii paljon aikaa.

LÄHTEET

Sähköiset lähteet

1 The Lua Architecture, Advanced Topics in Software Engineering [www-sivu]. [viitattu 13.3.2007] Saatavissa: pgl.yoyo.org/lua/docs/luaarchitecture.doc

2 Programming in Lua [www-sivu]. [viitattu 17.03.2007] Saatavissa: http://www.lua.org/pil/

3 Lua 5.1 Reference Manual [www-sivu]. [viitattu 17.03.2007] Saatavissa: http://www.lua.org/manual/5.1/

4 Lua for Series 60 [www-sivu]. [viitattu 17.03.2007] Saatavissa: http://luaforge.net/projects/luas60/

5 Lua5 [www-sivu]. [viitattu 17.03.2007] Saatavissa: http://www.freepoc.org/viewapp.php?id=32

6 Scripting: Higher Level Programming for the 21st Century [www-sivu]. [viitattu 05.06.2007] Saatavissa: http://home.pacbell.net/ouster/scripting.html

7 Lua [www-sivu]. [viitattu 17.03.2007] Saatavissa: http://www.lua.org

8 Lua Version history [www-sivu]. [viitattu 17.03.2007] Saatavissa: http://www.lua.org/versions.html

9 Associative array [www-sivu]. [viitattu 17.03.2007] Saatavissa: http://en.wikipedia.org/wiki/Associative_array

10 Tavukoodi [www-sivu]. [viitattu 17.03.2007] Saatavissa: http://fi.wikipedia.org/wiki/Tavukoodi

11 C/Invoke [www-sivu]. [viitattu 17.03.2007] Saatavissa:
http://www.nongnu.org/cinvoke/index.html

12 Implementing_a_Singleton_Class_in_Symbian_OS_v1_1_en.pdf
[www-sivu]. [viitattu 24.04.07] Saatavissa:
http://www.forum.nokia.com/

13 S60 3rd Edition SDK Symbian OS, Using Asynchronous Programming
[www-sivu]. [viitattu 30.04.07] Saatavissa:
http://www.forum.nokia.com/

14 Symbian OS: Active Objects And The Active Scheduler
[www-sivu]. [viitattu 06.05.07] Saatavissa:
http://www.forum.nokia.com/