ELŐADÁS:

KLIENS-SZERVER ALAPÚ ERLANG PROGRAMOK TRANSZFORMÁCIÓJA ERLANG OTP SÉMÁRA

Király Roland, serial@aries.ektf.hu

EKF Információtechnológiai Tanszék - Matematikai és Informatikai Intézet, Eger

Az Erlang nyelv

Az Erlang egy funkcionális programozási nyelv, amely konkurens, valós idejű, elosztott, nagy hibatűrő képességű rendszerek készítését teszi lehetővé. Az Ericsson és az Ellemtel Computer Science Laboratories készítette. Kidolgozását indokolta, hogy a 90-es évek elején nem volt a fejlesztők kezében az igényeiknek megfelelő programozási nyelv.

A szekvenciális programok írása mellett szükség volt funkcionális nyelvi elemekre, melyek az elosztottságot támogatják. A konkurencia bevezetését elsősorban a magas szintű párhuzamossági problémák megoldásának lehetősége motiválta (pl. valós idejű irányító rendszerek készítése).

A nyelv legfontosabb jellemzői:

- Funkcionális nyelvi elemek
- Folyamatokon alapuló konkurencia modell
- Valós idejű alkalmazások készítése kis válaszidővel
- Magas hibatűrő képesség
- Működés közbeni kódcsere lehetősége
- Robosztusság
- Elosztottság, és közös memória nélküli kommunikáció
- Más nyelven írt programkomponensek használatának vagy hívásának támogatása

Az Erlang/OTP (Open Telecom Platform) az Ericson által használt nyelv, mely alkalmas elosztott és párhuzamos működésű programok írására. A rendszerben készült, és futtatott programok hibatűrő képessége és robosztussága igen magas. Az Erlang rendelkezik általános jellegű funkcionális nyelvi elemekkel, kiterjesztve az üzenetküldés lehetőségével az elosztott programok megvalósításához.

Refaktorálás

A refaktorálás olyan program-transzformációs technika, amely segít megváltoztatni a programok kódját a viselkedési formájuk megváltoztatása nélkül. Segítségével tisztább, átláthatóbb kódokhoz juthatunk úgy, hogy a program a transzformáció előtti módon működik.

Támogatók:
GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK és Ericsson Hungary

A legáltalánosabb refaktorálási lépések a változókkal és a függvényekkel kapcsolatosak, mivel a programok számottevő részét ezen alkotóelemek adják. A változók, függvények és kifejezések mellett azok környezetének a vizsgálata is fontos feladat. A változók hatáskörének, kifejezések mellékhatásainak vizsgálata, szemantikai és szintaktikai elemzések nélkül a transzformációs lépések implementálása nem valósítható meg.

Refaktorálási példa - Merge Subexpression Duplicates

Refaktorálást sok nyelven, sokféle módon végeznek. Az alábbi példa a többször előforduló kifejezések helyettesítését mutatja be Erlang nyelvű környezetben.

A Merge Subexpression Duplicates [1][3][4] refaktor lépésben a kijelölt kifejezés összes előfordulását helyettesítjük egy változóval, melybe előzőleg kötjük a kifejezést. A transzformáció után a kód, vagyis a program viselkedése nem változik meg, de olvashatóbbá, egyszerűbbé válik.

Az alábbi példa [8] azt mutatja be, hogyan transzformálható a bal oldali, többször szereplő kifejezést tartalmazó forráskód a jobboldali, összevont kifejezést tartalmazó kóddá.

```
-module(refac).
-export([foo/2]).

foo(A,B) ->
  peer ! {note, A+B},
  A+B.
```

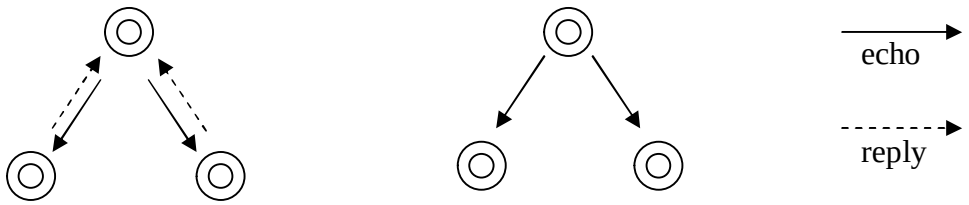
```
-module(refac).
-export([foo/2]).

foo(A,B) ->
  V = A+B,
  peer ! {note, V},
  V.
```

Kliens-szerver alapú Erlang programok

Az erlang programok minden esetben futó processzek, melyek nem használnak közös memóriát. A kommunikációt üzenetek küldésével oldják meg. Az üzenetküldés lehet szinkron, vagy aszinkron működésű. Az aszinkronról szinkron-működésűvé transzformálás is lehetséges transzformációs lépés [1].

Szinkron és aszinkron kommunikáció:



Egy tipikus, erlang, kliens-szerver alapú alkalmazás egy szerver és egy- vagy több kliens processzből épül fel.

A szerver processzt regisztrálják a rendszerben, így megszólíthatóvá válik a kliens-alkalmazások számára. A regisztráció történhet lokálisan, vagy globálisan, hálózati eléréssel.

Szerver alkalmazás forráskódja Erlang nyelven:

```
1. -module(server).
2. -export([start/0,init/1]).

3. start() ->
4.    register(server_,spawn(server, init, [])).

5. init(State)->
6.     loop(State).


7.  loop(State) ->
8.    receive
9.     {client_1, A} -> State1=foo(State, A), loop(State1);
10.    {client_2, B} -> State1=foo(State, B), loop(State1);
11.    {client_3, C, Pid} ->
12.           Pid ! {server_, ok},
13.           State1=foo(State, C, loop(State1).
14.    end,

16. foo(State, Var) -> do_something.
```

A sorok számozása nem része a forráskódnak. Az 1. sorban a modul neve, majd a 2. sorban a külvilág felé exportált függvényei láthatóak (függvénynév aritás párral definiálva). A 3. sor a szerver elindítására szolgáló függvényt tartalmazza, mely a kezdőállapotot a szerver processz ID értékkel tölti fel, majd a 4. sorban a függvény törzse foglal helyet, mely regisztrálja a szervert server_ néven. A processz ID az Erlang nyelvben egy külön típus, mely a processzeket azonosítja. Az 5. sor init/1 függvény inicializálja a szervert úgy, hogy elindítja a rekurzív loop/1 függvényt, mely maga a szerveralkalmazás. A szerver háromféle üzenetet fogad (9-10-11. sorok) mintaillesztéssel kiválasztva az üzenetekhez tartozó klózt, melyet végre kell hajtania.

Az üzenetek a kliens oldali függvényekben leírt üzenetküldő rutinoktól kerülnek a szerverhez. Az ágak végrehajtása után a szerver állapotot vált, vagyis újra meghívja önmagát a State1 változó értékével (a foo/2 függvény a példa szempontjából érdektelen).

A {client_3, C, Pid} mintára illeszkedő üzenet hatására lefutó klóz szinkron, a 9-10. sorban található klózok aszinkron működésűek.

A szerver kezdő állapotként a saját processz ID-jét állítja be, majd minden beérkező üzenet hatására állapotot vált. Az új állapotok a kliens processzek üzeneteiből jönnek létre.

A szerverhez tartozó kliensek forráskódja:

```
1. -module(clients).

2. -export([client_1/0,client_2/1,client_3/0]).

3. client_1() ->
4.      server_ ! {client_1, message1}.

5. client_2(B) ->
6.      server_ ! {client_2, B}.

7. client_3() ->
8.    server_ ! {client_3, message3, self()},
9.    receive
10.       {server_, ok} -> ok;
11.   end.
```

A 3-5-7. sorokban kezdődnek a kliens oldalról hívható függvények, melyek üzenetként egy rendezett n-est, vagyis tuple-t küldenek a szervernek. A 7. sorban található függvény különlegessége abban rejlik, hogy szinkron kommunikációt használ, vagyis megvárja a szerver válaszát, ahogy ez a 9-10. sorokban a receive kezdetű részben látható.

A három függvény egymástól teljesen független, a közös bennük az, hogy egyazon szerverhez kapcsolódnak, de az első kettő más szerverekkel is képes megfelelően kommunikálni.

A kliens-szerver alkalmazások alapvető hiányosságai

A szinkron kommunikációt használó, rekurzívan működő szerver alkalmazások átalakítása szinkron kommunikációt használó, iteratívan működő alkalmazásokká lényegesen javíthatja azok hatékonyságát [1].

Erlang nyelven írt szerver alkalmazások esetén állapotátmenetekkel oldják meg az aktuális állapotok megőrzését, mikor a szerver rekurzívan meghívja önmagát.

A rosszul megírt rekurzív hívások hamar a programverem telítődését okozzák. Ilyen esetben célszerű a rekurziót átalakítani iteratívvá, ami azt jelenti, hogy az adott függvényben az utolsó hívás más, külső függvény meghívása lesz.

Az egyszerűség kedvéért a jól ismert faktoriális problémát használjuk szemléltetésre.

Rekurzív megoldás:

```
fact(0) -> 1.

fact(N) -> N*fact(N1).
```

Ez esetben az `N*fact(N-1)` résszel van a probléma, mivel minden egyes futáskor adatokat kell elhelyezni a verembe.

Iteratív megoldás:

```
fact(N) ->  fact_x(N,1).

fact_x(0,X) -> X;
fact_x(N,X) -> fact(N1, N*X).
```

Legtöbbször a rekurzív probléma egy új függvény és egy változó bevezetésével kiküszöbölhető, de nem minden rekurzív megoldáshoz létezik vele egyenértékű iteratív. A processzek szinkronizálására pedig nehéz lenne találni általánosítható refaktorizációs megoldást, mivel még az is nehéz feladat, hogy felismerjük, melyik processz melyik másikkal áll kapcsolatban.

Erlang Open Telecom Platform

Az Erlang OTP lehetővé teszi nagy hibatűrésű, elosztott programok készítését. Az OTP kiterjesztés gen-server, gen-fsm, gen-event, és supervisor moduljai tartalmazzák azokat az előre elkészített viselkedési formákat, melyeket az ún. callback modulokban lehet felhasználni az üzenetküldések és állapotátmenetek hatékonyabb implementálása érdekében. Az említett elemek mellett az Erlang programozási nyelvben lehetőség van saját viselkedési formák megvalósítására is.

Az Erlang folyamatok nem használnak közös memóriát, így az üzenetküldés nagyon fontos eleme az elosztottság kezelésének.

Az OTP modulok használata lehetővé teszi a kód szétválasztását generikus és specifikus részekre.

A generikus rész speciálisan képzett programozók által elkészített viselkedési formákból áll (melyek leírják a kommunikációs formákat az elosztott, vagy kliens - szerver alapú programok esetén), a specifikus rész, vagyis a callback modul a generikus részek felhasználásával készül modul, mely a generikus részben leírt függvényeket implementálja.

Supervisor Tree

Az Erlang OTP központi ötlete a Supervisor Tree, mely lehetővé teszi a processzek hierarchikus elrendezését. A hierarchia élén, a gyökér elem minden esetben egy supervisor processz, melyhez worker processzek kapcsolódnak. A worker processzek maguk is lehetnek supervisor-ok.

A worker processzek végzik a tényleges számításokat, a supervisor processzek felügyelik a worker-ek működését. A hierarchikus elrendezése a processzeknek nagy hibatűrő képességgel ruházza fel az alkalmazásokat.

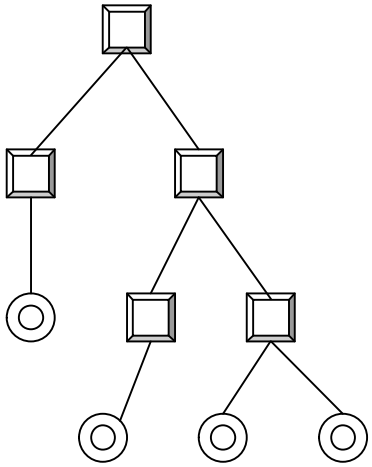
Amint valamelyik worker hibás működést eredményez, magáll, exit signal-t küld a felügyelőjének, ezzel értesítve azt a hibáról. A felügyelő processz újraindítja a workert és a hozzá tartozó részfát is, ha az nem levél eleme a fának. Az újraindítás módját a programozó szabályozhatja az aktuális processz forráskódjában.

A supervisor viselkedési formák implementálása viszonylag egyszerű, mivel az Erlang OTP Supervisor modulja tartalmazza az előre programozott funkcionalitást.

A supervisor-worker működési elv lehetővé teszi azt, hogy az Erlang nyelven írt programok kis százalékban, vagy egyáltalán ne tartalmazzanak hibakezelő és javító

kódokat (kivételeket, azok feldobását és kezelését), így csökkentve a forráskódok méretét akár 20-30%-kal.

Supervisor tree



A képen a dobozok reprezentálják a supervisor, a kettős körök a worker processzeket.

Az Erlang OTP az alábbi négy alapvető behaviour modullal rendelkezik:

- gen_server – Generic Server a kliens-szerver alkalmazásokhoz
- gen_fsm - Állapotátvitelekhez
- gen_event – Eseménykezelésre, pl.: Error Handling
- supervisor – A supervisor tree implementálásához

Az Erlang nyelv támogatja a magasabb rendű függvények használatát így adva lehetőséget a szerver-alkalmazások függvényekkel történő paraméterezésére.

A szerver processz funkcionalitását egyszerűen megvalósíthatjuk egy paraméterként átadott függvénnyel, melyet a szerver inicializálásakor adunk át paraméterként. Ez lehetővé teszi azt, hogy a szerver-alkalmazást általánosan írjuk meg, nem specifikálva a tényleges funkciót. A szerver kódja csak az alapvető viselkedési formákat írja le, alkalmassá téve azt más-más szerverek implementálására. (a példában szereplő gen_server implementáció nem használja ki ezt a lehetőséget.)

Transzformációk Erlang OTP mintára

A transzformáció lényege, hogy a hagyományos kliens-szerver kódokat Erlang OTP Generic Server mintájúra alakítsuk. A transzformáció során létrejött kód tartalmazza a szerver és a kliens függvények, vagyis a processzek kódját. Ez gyakori megoldás Erlang programok esetén.

A gen_server implementációja során választható a szinkron és az aszinkron működés, és a gen_server teljesen elrejti a processzek közti kommunikációt, csökkentve ezzel a hibák lehetőségét.

A fentebb ismertetett szerver OTP Generic Server segítségével implementált változata:

```
1. -module(server).
2. -behaviour(gen_server).
3. -export([client_1/0, client_2/1, client_3/0]).
4. -export([start/0, init/1, handle_call/3,handle_cast/2]).

5.  start() ->
6.   gen_server:start_link({local,server}, server, [], []).

7.  init(State) -> {ok, server}.

9.  handle_cast(_Request, State) ->
       State1=foo(_Request), {noreply, State1}.
10. handle_call(_Request, _From, State) ->
11.    State1=foo(_Request),{reply, State1, State1}.

13. client_1() ->
14.      gen_server:cast(server, 10. {client_1,message1}).
15. client_2(B) ->
16.      gen_server:cast(server, {client_2, B}).
17. client_3() ->
18.      gen_server:call(server, {client_3,message3}).
19. foo(Request) -> do_something.
```

A transzformáció után a szerver és a kliens kódja lényegesen egyszerűbbé válik úgy, hogy a funkcionalitás nem változik meg. A szerver receive része, mely az üzeneteket feldolgozza, szükségtelenné válik, mivel az állapotátmeneteket és az üzenetküldést a gen_server függvényei teljes egészében megvalósítják, s a callback modul csak implementálja azokat.

A megvalósítás lépései

A bemutatott transzformációs lépés meghatározott felépítésű kódokra alkalmazható. Jelenleg tervezési fázisban van és nincs minden általános megoldása. Eset-elemzések és megvalósíthatósági tanulmány készül hozzá, melyek az implementáláshoz szükséges szabályok megalkotását teszik lehetővé [5] [6] [7].

A transzformáció elvégzéséhez szükséges a jelenleg működő refaktor lépésekhez használt, szintaktikai és szemantikai analízist ki kell terjeszteni több modulra, mivel a kliens és szerver oldali kódok általában külön fájlokban foglalnak helyet [9].

Egyes esetekben szükséges lehet adatfolyam analízis és függőségi vizsgálatok elvégzésére a függvények és változók hatókörének megállapításához [4].

A jelenleg implementált és implementálás alatt álló transzformációs lépések mellet remélhetőleg ez az új is megjelenik a közeljövőben. Kedvező esetben a transzformáció elvégezhető, alapvető refaktorálási lépések egymás utáni alkalmazásával, vagy azok kis mértékű átdolgozásával [1] [10].

Siker esetén ez a művelet nagyon megkönnyítené a hagyományos kliens-szerver alapú programok átalakítását, modernizálását.

Támogatók:
GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK és Ericsson Hungary

Irodalomjegyzék

[1] Huiqing Li, Simon Thompson University of Kent, UK, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, Tamás Nagy Eőtvős Loránd University, Hungary: **Refactoring Erlang Programs**
Erlang User Conference 2006 Stockholm, November 9-10, 2006

[2] Horváth, Z., Lövei, L., Kozsik, T., Víg, A., Nagy, T.: **Refactoring Erlang Programs**
Submitted to The Fifth Conference of PhD Students in Computer Science, CSCS 2006

[3] Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: **Refactoring Erlang Programs**
Conference poster at High Speed Networking Workshop, HSN 2006

[4] Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: **Erlang programok refaktorálása**
Presentation at IKKK conference, Nov. 23, 2006

[5] Nagy, T., Víg, A.: **Storing Erlang source code in database**
ELTE 2006

[6] Nagy, T., Víg, A.: **An Erlang refactor step: tuple function arguments**
Scientific Students' Associations Conference, ELTE, Budapest, Dec. 2006

[7] Nagy, T., Víg, A.: **Erlang refactoring with relational database**
Presentation in University of Kent Functional Program Group Meeting (Canterbury), 14. December 2006.

[8] ELTE IK **Refactoring Erlang Programs** http://plc.inf.elte.hu/erlang

[9] Lövei, L., Horváth, Z., Kozsik, T., Király, R.: **Static rules for variable scoping in Erlang**
Submitted to the 7th International Conference on Applied Informatics, ICAI 2007

[10] Nagy, T., Víg, A.: **Erlang refactoring with relational database**
Presentation for the Erlang community in the office of Erlang Training and Consulting Ltd. (London), 8. February 2007.