Kliens szerver alkalmazások automatikus optimalizálása bonyolultsági mértékek alapján

Király Roland
Eszterházy Károly Főiskola - Matematikai és Informatikai Intézet
kiraly.roland@ektf.hu

2014. március 24.

Kivonat

A programok méretének és bonyolultságának növekedésével a fejlesztés egyre nagyobb részét képezi a tesztelés és a tesztelés során felmerülő problémák megoldása.

A különböző program elemzésekkel a forrásszöveg azon tulajdonságait mérjük, amelyek segítségével képet kaphatunk annak struktúrájáról, karakterisztikájáról, és bonyolultságáról.

Az így kapott eredmények alapján becsléseket adhatunk a programszöveg tesztelési, fejlesztési, valamint átalakítási költségeire. A funkcionális programozási nyelvek, így az Erlang nyelv is számos olyan különleges programkonstrukciót tartalmaz, amelyeket az Objektum Orientált, és az imperatív nyelveknél nem találhatunk meg. A különleges nyelvi elemek teszik a funkcionális nyelveket mássá, és ezektől a tulajdonságoktól válnak érdekessé, vagy különlegessé, de szintén ezek miatt az ismert bonyolultsági mértékek egy része nem, vagy csak átalakítással használható a programkódjuk mérésére. Ebben a környezetben felmerült az igény egy olyan összetett, és sokoldalú eszköz elkészítésére, amely képes a funkcionális programok bonyolultságát mérni, a mért értékek alapján lokalizálni a kezelhetetlenül bonyolult részeket, valamint alkalmas ezeknek a programrészeknek az automatikus, vagy félautomatikus javítására.

[0]A kutatási tevékenység a TÁMOP 4.2.4.A/2-11-1-2012-0001 azonosító számú Nemzeti Kiválóság Program – Hazai hallgatói, illetve kutatói személyi támogatást biztosító rendszer kidolgozása és működtetése országos program című kiemelt projekt keretében zajlik. A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

A probléma megoldásához, vagyis a bonyolultsági mértékek méréséhez késztettünk egy magas szintű, strukturált lekérdező nyelvet és egy elemző programot, amelyet kibővítettünk transzformációs szkriptek írásának a lehetőségével. Elkészítettük azt a transzformációs programozási nyelvet, amely segítségével lehetőségünk nyílik szkriptek írására, és a programszövegek automatikus javítására. Az elemző és hibajavító eszközünket teszteltük nagyobb forrásszövegeken és jelen cikkben szeretnénk megvizsgálni a kliens szerver alapú programok szerveroldali részének a tesztelését és átalakításának lehetőségeit.

1. Automatikus programtranszformációk

1. Probléma (Automatikus transzformációk). *Az értekezésben azt vizsgáljuk meg, hogy képesek vagyunk-e funkcionális (Erlang [8, 9]) nyelven készült alkalmazások forrásszövegének automatikus transzformációjára a szoftver bonyolultsági mértékek alapján, és hogy van-e lehetőségünk kódminőség javító transzformációs sémák kidolgozására a mértékek eredményeire alapozva. Mindezek mellett a kliens szerver alapú programokon eszközölt transzformációkat végre lehet-e hajtani úgy, hogy azok futását nem kell megszakítani.*

A probléma megválaszolása érdekében létrehoztunk egy a programok bonyolultsági mértékeit mérő, és a mértékeken alapuló automatikus transzformációkat végző algoritmust, és definiáltuk azt a szkript nyelvet, amely lehetővé teszi a programok átalakítását szolgáló transzformációs lépések leírását.

Véleményünk szerint a bonyolultsági mértékek mérése a forrásszövegből épített szintaxisfán, valamint az abból létrehozott, szemantikus gráfon [6, 10] lehetővé teszi a forrásszöveg minőségének automatikus javítását.

A létrehozott rendszer használata mellet az alábbi kérdésekre keressük a választ:

- Hogyan javíthatjuk a programok bonyolultságát és olvashatóságát, valamint milyen átalakításokat kell elvégezni ahhoz a programon, hogy annak lexikális felépítése, vagyis a programozói stílus is javuljon, de a szemantikája ne változzon meg?
- Tudjuk-e automatizálni a bonyolultság alapú programtranszformációkat az általunk készített programozási nyelv segítségével?
- Hogyan lehet a kódjavítást végző szkripteket alkalmazni a programokon úgy, hogy azok futását ne szakítsuk meg?

2. A méréshez használt bonyolultsági mértékek

Ebben a fejezetben, az egyértelműség kedvéért definiálunk két ismert bonyolultsági mértéket, amelyet a szkriptek alkalmazása során a transzformációk irányítására használhatunk, majd megmutatjuk a szkriptjeinkben használható összes mértéket táblázatos formában.

Az általunk készített elemző algoritmusban alkalmazható mértékek közül jelen írásban a *McCabe* ciklomatikus számot, valamint a *case* kifejezések maximális beágyazottságát mérő mértékeket mutatjuk be részletesen, mivel az optimalizálásra készített szkriptekben is ezeket a mértékeket használtuk fel:

A *McCabe* bonyolultság mérték értéke a *Thomas J. McCabe* által konstruált vezérlési gráfban [1] definiált alapvető útvonalak számával azonos, vagyis azzal, hogy hányféle kimenete lehet egy függvénynek nem számítva a benne alkalmazott további függvények bejárási útvonalainak a számát.

Thomas J. McCabe a programok ciklomatikus számát a következőképpen definiálja:

1. Definíció. *A $G = (v, e)$ vezérlési gráf $V(G)$ ciklomatikus száma $V(G) = e - v + 2p$, ahol p a gráf komponenseinek a számát jelöli, ami megegyezik az erősen összefüggő gráfban található lineárisan összefüggő körök számával [3].*

A *McCabe* szám az Erlang programok függvényeinek méréséhez a következő módon adható meg:

2. Definíció. *Az f_i függvény ágait (overload változatait) jelölje $fc(f_i)$, és az ágakban található if, valamint case kifejezések ágait jelölje $if_{cl}(f_i)$, és $case_{cl}(f_i)$. Ekkor a McCabe ciklomatikus szám függvényekre mért eredménye $MCB(f_i) = |fc(f_i)| + |case_{cl}(f_i)| + |if_{cl}(f_i)|$.*

A következő általunk használt bonyolultsági mérték a *case* kifejezések maximális beágyazottságát méri a függvényekben.

c_0:
  case e of
    p_1 [when g_1] $\rightarrow e_1^1, \ldots, e_{l_1}^1$;
    $\vdots$
    p_n [when g_n] $\rightarrow e_1^n, \ldots, e_{l_n}^n$
  end

case kifejezést jelöl, ahol e, és $e_i \in E$ kifejezések, $p \in P$ minták $g_i \in G$

őr feltételek az ágakban. A *case* kifejezések ágaiban az e_j^i kifejezések tartalmazhatnak beágyazott vezérlő szerkezeteket, többek között újabb *case* kifejezéseket is.

3. Definíció. *A beágyazottság mérése érdekében jelöljük $T(f_i)$-vel az f_i függvényben található összes case kifejezés halmazát. Jelölje $t(c_1, c_2)$ azt, ha c_1 case kifejezés valamely ága tartalmazza c_2 case kifejezést, és $\nexists c_3$ case kifejezés, hogy $t(c_1, c_3) \wedge t(c_3, c_2)$. Jelölje $t_s(c, c_x)$ azt az esetet, hogy a c case kifejezés valamely ágában valamely mélységben tartalmazza a c_x case kifejezést, vagyis $\exists$ $c_1, ..., c_n$ case kifejezések, hogy*

$$t(c, c_1), t(c_1, c_2), ..., t(c_{n-1}, c_n), t(c_n, c_x).$$

A $|t_s(c, c_x)|$ beágyazás mélysége ez esetben $n + 1$. Legyen $T_0(f_i)$ azon case kifejezések halmaza, amelyeket egyetlen $T(f_i)$ halmazbeli case kifejezés sem tartalmaz (felső szintű case kifejezések). Ekkor az

$$MDC(f_i) = max\{|t_s(c, c_x)| \quad |c \in T_0(f_i), c_x \in T(f_i)\}.$$

Ahogy azt korábban említettük, az itt bemutatott mértékeken kívül az elemző algoritmus számos más bonyolultsági mértéket is képes mérni, és alkalmazni a mérési eredményeket különböző transzformációk kivitelezésére. Ezeket a 1. táblázatban láthatjuk.

Mérték megnevezése	**Mérhető program-konstrukciók**
module_sum	module
line_of_code	module/function
char_of_code	module/function
number_of_fun	module
number_of_macros	module
number_of_records	module
included_files	module
imported_modules	module
internal_cohesion	module
function_calls_in	module
function_calls_out	module
cohesion	module
function_sum	module/function
otp_used	module
max_depth_of_calling	function/module
max_depth_of_cases	function/module
max_depth_of_structs	function/module
number_of_funclauses	function/module
number_of_exceptions	function/module
branches_of_recursion	function/module
mcCabe	module/function
calls_for_function	function
calls_from_function	function
number_of_funexpr	function/module
number_of_messpass	function/module
fun_return_points	function/module
average_size	module/function
average_length_of_line	module/function
max_length_of_line	module/function
extarnal_calls	function
internal_calls	function
extarnal_calls_from	function
internal_calls_from	function
number_of_variables	function/module
length_of_name	function/module

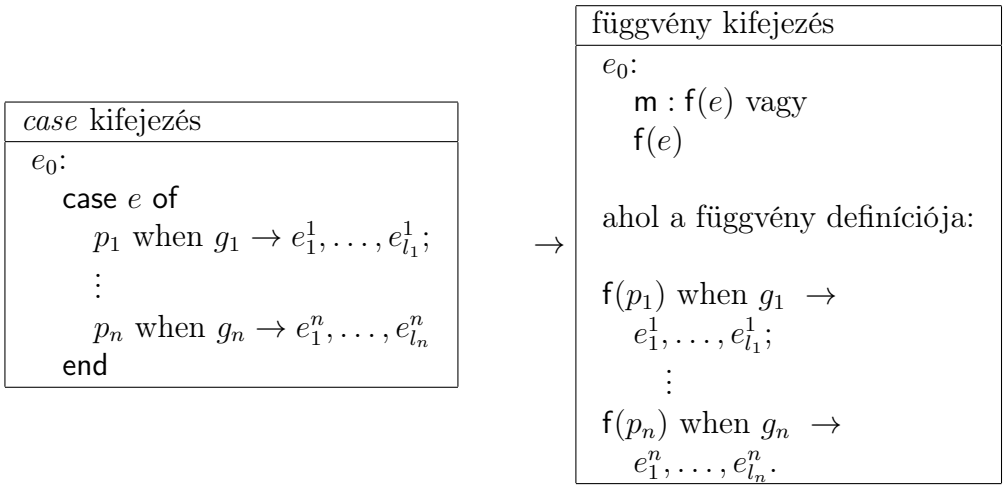
1. táblázat. Implementált szoftver bonyolultsági mértékek

3. A kódminőség javítására használt transzformációk

Ebben a fejezetben ismertetjük a forrásszöveg minőségének javítására használt, szkriptekben alkalmazott transzformációs lépések működését. A szkriptek a 2. fejezetben bemutatott bonyolultsági mértékek alapján automatikusan átalakítják a forrásszövegben található programkonstrukciókat.

A *McCabe* ciklomatikus szám, valamint a programozói stílus javítására alkalmazzuk a mélyen beágyazott case kifejezések kiemelését, és néhány esetben, ahol a transzformációk hatására a függvények száma túlságosan megnő, a függvény mozgatását végző transzformációs lépéseket.

Case kifejezés függvénnyé alakítása nevű transzformációs lépés a kiemelésre kijelölt *case* kifejezést függvénnyé alakítja, majd az új függvényre egy hívást helyez el a kifejezés eredeti helyén úgy, hogy a benne kötött változókat paraméterré alakítja (lásd.: az 1. ábrán).



1. ábra. Case kifejezés kiemelése

A függvények modulok közötti mozgatására használható transzformáció a kijelölt függvényeket egy másik modulba helyezi át. Természetesen ez a transzformációs lépés (az előre és jól meghatározott szabályok betartásával) elvégzi a szükséges kompenzációs lépéseket, mint a kapcsolódó rekordok, és makrók elérhetőségének a biztosítása, a függvényre irányuló, és az onnan kiinduló minősített hívások kezelése, vagy cseréje.

4. Transzformációs szkriptek

A fejezetben bemutatjuk azt az általunk kifejlesztett és implementált, automatikus program transzformációkra alkalmas nyelvet, amely segítségével a bonyolultsági mértékek javítására irányuló szkripteket készíthetünk.

A transzformációs nyelv alkalmas arra, hogy használatával a mérések alapján, előre definiált feltételek figyelembe vételével a programszövegből épített szemantikus gráfban [6, 10] tárolt forráskódot automatikusan át lehessen alakítani, majd a gráfból az átalakítást követően visszaállítani azt.

Query → **MetricQuery** | **OptQuery**
OptQuery → **Opti Where Limit**
Opti → **optimize Transformation**
Transformation → *TransformationName*
| *TransformationName* **Params**
Params → (*Attr*, *ValueList*)
Where → **where Cond**
Cond → *Metric Rel CondValue*
| **Cond** *LogCon* **Cond**
Limit → **limit** *Int*

2. ábra. A transzformációs szkriptek nyelve.

A szintaxis leírásban a *Transformation* transzformációt (pl. `extract_fun`), a *Rel* relációt és egyéb operátort (pl. $<, <=, >=, >, like$), a *LogCon* logikai operátort (pl. „és”, „vagy”), a *CondValue* egész számot vagy lexikális elemet jelölhet (pl. egy modul neve a *like* használata mellett). Az *Int* helyére a *limit* szakaszban pozitív egész szám helyettesíthető. (A nemterminális elemek nagy kezdőbetűsek, a nyelv kulcsszavai kis kezdőbetűvel lettek szedve.)

Az *optimize* szakaszban megadhatjuk az átalakításokhoz alkalmazandó transzformációs lépést, és annak paramétereit. A *where* kulcsszó után definiálható összetett feltétel az, amely a mérések elvégzését kezdeményezi, és a transzformációk lefutását vezérli, vagyis azt, hogy milyen feltételek teljesülése mellett kell az adott transzformációs lépést újra és újra lefuttatni, és azt is, hogy mely szemantikus gráf csomópontokat kell transzformálni. Az így megadható "bázis-feltételben", ami egy logikai kifejezés, szerepelnie kell legalább egy olyan rész-kifejezésnek, ami a transzformációra kijelölt modulokon, vagy függvényeken mérhető bonyolultsági mértéket, egy aritmetikai operátort, valamint egy konstans értéket tartalmaz.

A kijelölt, vagyis a transzformáció tárgyát képező elemek így már nem direkt definiált programkonstrukciók, vagy kifejezések, hanem a feltételek

alapján automatikusan kiválasztásra kerülő programszeletek. Ezzel a módszerrel a transzformálandó programrészek kijelölése a lexikális szintről átkerül a szemantikus elemzések szintjére [5].

A transzformációs nyelven leírt szkript futása során az elemző megkeresi a feltételének megfelelő programrészeket, majd végrehajtja a *optimize* szakaszban kijelölt transzformációt, ezután megméri a feltételben adott bonyolultsági mértékeket az összes szemantikus gráf csomópontra. Amely csomópontok az operátor, és a konstans alapján nem kell, hogy szerepeljenek a további transzformációkban kiesnek a szkript hatásköréből. Amennyiben nincs olyan csomópont, amelyen a transzformációt újra le kell futtatni, a szkript futása megáll.

Ezen feltételek mellett előfordulhatnának olyan esetek, mikor a program futása nem áll meg. A probléma elkerülésére a *limit* kulcsszó után leírt konstans segítségével definiálható a maximálisan futtatható ismétlések száma. Amennyiben tehát a transzformációs lépés nem hozza meg a kívánt eredményt, a *limit* konstansa adott lépésszám után biztosan megállítja a futását.

5. Szkriptek alkalmazása programkód javítására

Az előző fejezetben ismertetett szkriptnyelv használatával képesek vagyunk olyan bonyolultság alapú transzformációs folyamatokat leírni, amelyek automatikusan kijavítják a programok forrásszövegét a szkriptekben leírtak alapján.

Az alábbi szkriptet tesztelési célokkal lefuttattuk az Erlang disztribúcióba épített Dialyzer [9] nevű szoftveren, és a 3., valamint a 4. ábrákon látható eredményeket kaptuk.

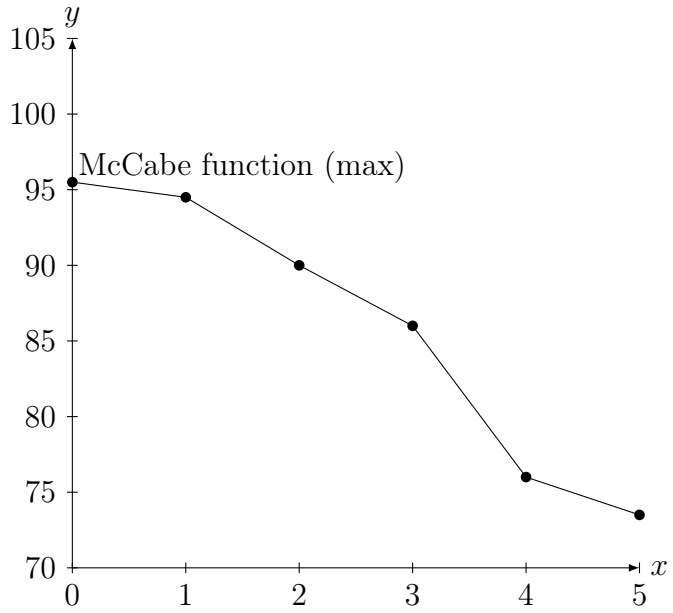
```
optimze
   extract_function (exprtype, case_expr)
where
   f_mcCabe > 6
    and
   f_max_depth_of_structures > 2
    and
   f_max_depth_of_cases > 1
limit
   7;

optimize
   move_fun (targetmod, 'funlib')
```

```
where
  f_external_calls > 1,
  f_internal_calls < 1,
  f_external_calls_from > 1,
  f_internal_calls_from < 1,
  m_number_of_fun > 1
limit 2;
```

A szkript célja, hogy a mérés tárgyát képező szoftver moduljaiban a McCabe értéket javítsa úgy, hogy a két szintnél mélyebben beágyazott case kifejezéseket emelje ki, majd ezekből hozzon létre új függvényeket mindaddig, amíg a McCabe szám nem javul, vagy a limit feltétel meg nem állítja a program futását.

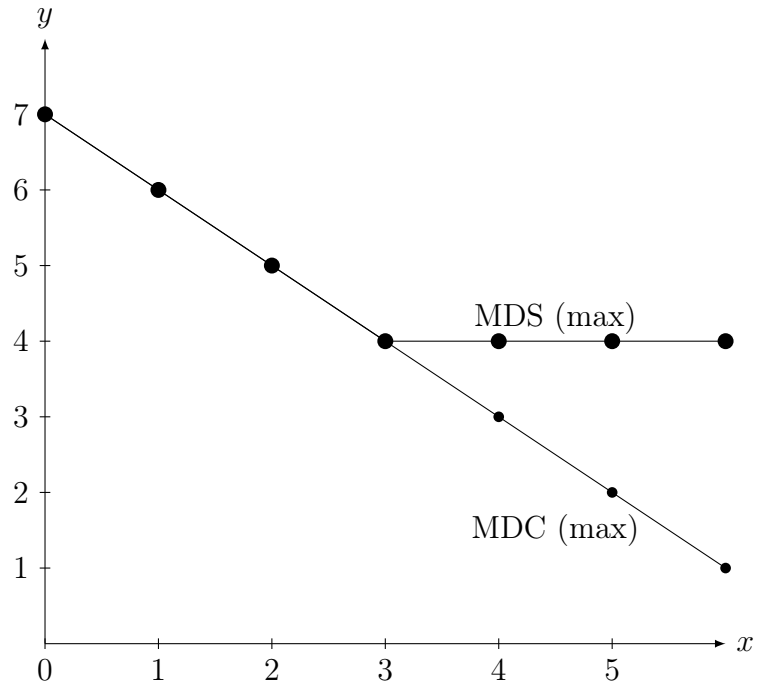
Az optimalizálási folyamatot kiegészítettük egy második *optimize* szakasszal is, amelynek a célja, hogy az olyan függvényeket, amelyekre csak hívás érkezik más modulok függvényeiből, de ezek nem hívnak függvényeket, kerüljenek bele egy új, *library* modulnak tekinthető programrészbe. Ez a lépés kicsit finomítja a szkript futása után kapott eredményeket.



3. ábra. A függvények McCabe számának maximuma (y tengelx) az egyes transzformációs lépéseket követően (y tengely)

Az optimalizálást követően két bonyolultsági mértéket vizsgáltunk meg az egyik a már említett McCabe szám, amelynek a függvényekre mért eredménye, vagyis a mérték változása a transzformációk során a 3. ábrán látható.

A másik mérték a *case* kifejezések, és az egyéb struktúrák beágyazottságát mérő mérték, amelynek eredményét a 4. ábrán láthatjuk.



4. ábra. A struktúrák (MDS) és a *case* kifejezések(MDC) beágyazottságának alakulása (y tengely) a transzformációs lépések során (x tengely)

Az eredmények javulást mutatnak a forrásszöveg struktúráját, és annak bonyolultságát tekintve, de jelen esetben nem kimondottan az eredmények javulására vagyunk kíváncsiak, hanem arra, hogy transzformációs szkripteket hogyan tudjuk alkalmazni kliens-szerver alapú programok transzformációjához, ráadásul úgy, hogy azok futását ne szakítsuk meg.

Az Erlang programozási nyelvnek van egy kiterjesztése Open Telecom Platform néven. Ez a kiterjesztése a nyelvnek lehetővé teszi az elosztott programok írását, és a párhuzamosítást közös memória használata nélkül. Ugyanez a kiterjesztés definiálja azt a kliens szerver modellt, amely segítségével a korábban bemutatott optimalizálási problémát meg tudjuk oldani.

Az OTP alapú szervereket processzusokba szervezik, és ezek a processzusok hierarchikus rendben, egy meghatározott újraindítási stratégia alapján indulnak el, és hiba esetén ugyanezen stratégia alapján indulnak újra.

A hierarchia élén, vagyis a processzusokból létrehozott gráf gyökér elemében van a szülő csúcs, amely az alatta lévő csúcsok működésért felelős (ezek a csúcsok önálló programok). Ha azok hiba esetén leállnak, egy ún.: EXIT

szignált küldenek a szülő csomópontnak, amely így értesül a hibáról, és az újraindítási stratégia alapján ismét elindítják az adott csomóponthoz tartozó programot.

Ezt a mechanizmust ki tudjuk használni az optimalizálási folyamat kivitelezésére az OTP alapú kliens szerver programok esetén. Egy adott optimalizáló lépés a futó alkalmazás kódját kijavítja, ezzel hibát okozva annak futásában. Hiba esetén a csúcshoz tartozó program leáll, a szülője pedig újraindítja, de már a javított forrásszöveggel (ezt a folyamat is benne van az OTP alkalmazásokban "Code Change" néven).

Az ilyen jellegű kódcserék kivitelezéséhez módosítani kell az optimalizáló szkripteket úgy, hogy azok minden javítást követően megvárják az adott processzus újraindítását, és az újraindítási stratégiát is úgy kell módosítani, hogy az ne csak néhányszor, hanem akárhányszor újra tudja indítani a csúcsokat.

Természetesen a kivitelezés során számos szinkronizációs problémával is szembesültünk. Ezen problémák megoldását nem ismertetjük ebben a cikkben, mivel azok még a kutatás egy kezdeti szakaszában vannak, és konkrét eredményeket nem tudunk szolgáltatni. Az újraindítással történő optimalizálást a prototípus szoftverünk már képes alkalmazni teszt környezetben, és hasonló eredményeket lehet elérni a segítségével, mint amilyeneket a nem kliens-szerver alapú programszövegek esetében is kaptunk.

6. Összegzés

Ismertettük az általunk kifejlesztett optimalizáló nyelvet és elemző algoritmusának működését. A nyelv segítségével a bonyolultsági mértékek mérésén alapuló, automatikus program transzformációs szkripteket tudunk készíteni. A 2. fejezetben bemutattuk azokat a strukturális bonyolultsági mértékek, amelyeket az Erlang forrásszövegek bonyolultságának méréséhez használtunk.

Az 3. és a 4. fejezetekben megvizsgáltuk annak a lehetőségét, hogy a bonyolultsági mértékek mérésével, és elemzésével hogyan lehet automatizált javítását célzó programtranszformációkat megvalósítani. Definiáltuk a szoftver bonyolultsági mértékeken alapuló automatikus transzformációs lépéssorozatok leírására és futtatására alkalmas nyelv szintaxisát, és ismertettük a nyelvhez konstruált elemző és futtatást végző algoritmus működési elvét.

A 5. fejezetben egy egyszerű példaprogrammal, és annak futási eredményeivel igazoltuk az automatikus kódminőség javítás működőképességét.

A szintaxis, és használati esetek ismertetése mellett megmutattuk azt, hogy milyen eredményeket érhetünk el egy egyszerű, néhány soros szkript felhasználásával.

Szintén ebben a fejezetben megvizsgáltuk annak a lehetőségét, hogyan lehet a kliens-szerver alapú Erlang programok automatikus javítását elvégezni az Erlang Open Telecom Platform környezet újraindítási stratégiájára alapozva úgy, hogy a programok futását ne szakítsuk meg.

Összefoglalva, a bonyolultsági mértékeken alapuló elemző, és optimalizáló algoritmus, amelyet Erlang nyelvű forrásszövegek készítése során, valamint korábban készült, de átalakításra váró programszövegek automatikus, vagy fél-automatikus javítására használhatunk megfelelően működött kliens-szerver alapú szoftverek transzformációja során.

A transzformációs lépéssorozatok javítottak azokon a bonyolultsági mértékeken, amelyeket optimalizálásra kijelöltünk. A transzformációk során a forrásszöveg jelentése nem változott meg, és a program újraindítását követően az elvárt módon működött.

A továbbiakban az itt bemutatott eredmények felhasználásával az elemzőt, és a hozzá konstruált transzformációs nyelvet ki szeretnénk egészíteni a szinkronizációs folyamatok vezérlését végző résszel, és tesztelni azt ipari környezetben működő programok forrásszövegére. Továbbá kísérletet teszünk arra, hogy a transzformációs szkriptek lefutását követően, az átalakított forrásszöveg jelentés megőrző tulajdonságát, valamint helyességét matematikai módszerek segítségével bizonyítsuk.

Hivatkozások

[1] McCabe T. J. A Complexity Measure, *IEE Trans. Software Engineering, SE-2(4), pp.308-320 (1976)*

[2] Zoltán Porkoláb, Ádám Sipos, Norbert Pataki, Structural Complexity Metrics on SDL Programs. *Computer Science, CSCS 2006, Volume of extended abstracts, (2006)*

[3] Zoltán Porkoláb Programok Strukturális Bonyolultsági Mérőszámai. *PhD thesis Dr Töke Pál, ELTE Hungary, (2002)*

[4] Zoltán Horváth, Zoltán Csörnyei, Roland Király, Róbert Kitlei, Tamás Kozsik, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víg.: Use cases *for refactoring in Erlang, To appear in Lecture Notes in Computer Science, (2008)*

[5] Csörnyei Zoltán *Fordítóprogramok* Typotex Kiadó, Budapest, 2006.

[6] R. Kitlei, L. Lövei, M Tóth, Z. Horváth, T. Kozsik, T. Kozsik, R. Király, I. Bozó, Cs. Hoch, D. Horpácsi.: Automated Syntax Manipulation in RefactorErl. *14th International Erlang/OTP User Conference. Stockholm, (2008)*

[7] Lövei, L., Hoch, C., Köllö, H., Nagy, T., Nagyné-Víg, A., Kitlei, R., and Király, R.: Refactoring Module Structure *In 7th ACM SIGPLAN Erlang Workshop, (2008)*

[8] Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víg, A., and Nagy, T.: Refactoring in Erlang, a Dynamic Functional Language *In Proceedings of the 1st Workshop on Refactoring Tools, pages 45–46, Berlin, Germany, extended abstract, poster (2007)*

[9] Erlang - Dynamic Functional Language
http://www.erlang.org

[10] T. Kozsik, Z. Horváth, L. Lövei, T. Nagy, Z. Csörnyei, A. Víg, R. Király, M. Tóth, R. Kitlei.. Refactoring Erlang programs. *CEFP'07, Kolozsvár (2007)*

[11] Thanassis Avgerinos, Konstantinos F. Sagonas *Cleaning up Erlang code is a dirty job but somebody's gotta do it.* Erlang Workshop 2009: 1-10

[12] Konstantinos F. Sagonas, Thanassis Avgerinos *Automatic refactoring of Erlang programs.* PPDP '09 Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming 2009: 13-24

[13] Király, R., Kitlei R.: *Complexity measurments for functional code* 8th Joint Conference on Mathematics and Computer Science (MaCS 2010) refereed, and the proceedings will have ISBN classification July 14-17, 2010

[14] Király, R., Kitlei R.: *Implementing structural complexity metrics in Erlang.* '10 ICAI 2010 – 8th International Conference on Applied Informatics to be held in Eger, Hungary January 27-30, 2010

[15] Király, R. and Kitlei, R.: *Implementing structural complexity metrics for Erlang* Poster on the 8th International Conference on Applied Informatics, ICAI 2010, 2010