Universitatea “Politehnica” din Timisoara

SINTEZA LUCRARILOR
la contractul de cercetare ID_1215 nr. 357/01.01.2007
Metode si instrumente pentru asigurarea continua a calitatii in sisteme software complexe
pentru anul 2009

Cercetari stiintifice

Obiectivul 1/2009*: *Evaluare intermediara a metodelor si instrumentelor de analiza dezvoltate

1.1 Evaluarea infrastructurii de testare si analiza statica pe proiecte de dimensiuni mari in Java si C++.

In fazele anterioare ale proiectului, unul din obiective a fost realizarea unui analizor static pentru programe in C/C++ care sa fie capabil de a analiza programe de dimensiuni mari. Pentru acesta, a fost selectata ca baza infrastructura de analiza statica oferita de cadrul de compilare LLVM (http://www.llvm.org), care ofera un cadru de analiza si optimizare a codului performant, comparabil si chiar depasind rezultatele standardului de facto care este compilatorul gcc.

Rezultate intermediare au fost prezentate la simpozionul de cercetare Alpine Verification Meeting 2008, in care participa prin invitatie cercetatori din tari europene activi in domeniul analizei si verificarii programelor, si au demonstrat ca sistemul este performant, reusind sa analizeze cod de dimensiunea serverului X din Gnu/Linux, dar necesita cresterea preciziei analizei. In faza curenta, a fost extinsa, rafinata si optimizata implementarea analizorului static, in conjunctie cu lucrarea de diploma [Tor09] a studentului Edwin Torok implicat in proiect, prin:

- efectuarea de analize interprocedurale. Aceasta se efectueaza prin inlining selectiv al codului, limitata la o dimensiune prestabilita pentru a evita cresterea exponentiala
- optimizarea prin slicing pentru focalizarea analizei asupra codului direct relevant pentru proprietatea care se doreste analizata. Experimentele au demonstrat utilitatea simplificarii prin slicing **inainte** de expandarea prin inlining, aceasta strategie fiind adoptata in proiect
- optimizarea analizei de constrangeri, prin compararea a trei solutii: calcularea de constrangeri si apelarea ca rezolvitor a bibliotecilor SMT-LIB (satisfiability modulo theories), implementarea directa prin constrangeri poliedrala cu PPL (Parma Polyhedral Library), si realizarea unui rezolvitor propriu pentru constrangerile produse de modulul SCEV (scalar evolution) al LLVM, ultima solutie dovedindu-se cea mai performanta (vezi si activitatea 2.2, Rafinarea analizei statice folosind proceduri de decizie).

Sistemul realizat a fost testat pe programe C/C++ open-source, cele mai mari fiind antivirusul ClamAV (110.000 linii de cod) si serverul Xorg (480.000 linii de cod). Sistemul are performante distinctive prin viteza sa deosebita, prelucrand peste 100.000 linii de cod si 240.000 de constrangeri pe minut, si fiind astfel integrabil chiar in procesul de compilare, pentru detectarea imediata a erorilor in procesul de dezvoltare. Sistemul a descoperit erori reale in aceste programe (rapoartele #15964 in Xorg, #1639 in ClamAV), iar module de analiza realizate in acest sistem sunt integrate deja in codul public disponibil al infrastructurii LLVM (http://llvm.org)

[Tor09] Edwin Torok, Interprocedural bounds checker for C programs using symbolic constraints and slicing, lucrare de diploma, Univ. Politehnica Timisoara, 2009. Disponibila la http://llvm.org/

In ceea ce priveste instrumentul ECHOS pentru analiza integrata in Eclipse a programelor Java, permitand analiza carentelor privind substituibilitatea obiectelor si generarea de teste, acesta a fost evaluat utilizand portiuni de cod ce reproduc probleme de proiectele de tip “violari ale principiului de proiectere LSP” regasite in cod real (ex. in biblioteca standard Java).

Evaluarea a urmarit mai muste aspecte. Din punctul de vedere al utilizabilitatii s-a analizat timpul de executie necesar generarii de teste utilizand diferite strategii de generare a acestora (ex. teste cu un apel de metoda, teste cu multiple apeluri de metoda, etc.). Tabelul de mai jos, un sumar al rezultatelor obtinute, suprinde media timpului necesar generarii de teste functie de strategia utilizata. Se poate observa faptul ca odata cu cresterea complexitatii strategie de testare timpul de executie creste destul de puternic. Cu toate acestea, acesti timpi de executie se incadreaza in valori acceptabile din punctul de vedere al utilizabilitatii (ex. cea mai complexa strategie de testare necesita un timp de executie mediu de aproximativ 2 minute).

Strategie	**Timp (s)**
Teste cu un apel de metoda (OneCallGenerator)	46.4
Teste cu doua apeluri de metoda (TwoCallGenerator)	95.9
Teste cu apeluri multiple de metode (MultipleCallGenerator)i	127.6

Un alt aspect de evaluare a constat in estimarea modului in care testele generate surprind suficient de bine comportamentul clasei testate pentru a fi suficiente pentru detectia de violari ale principilui de proiectare LSP. In acest scop am aplicat in izolare si in combinatie diferite strategii de generare de teste. Diferitele suite de teste obtinute astfel au fost apoi utilizate individual pentru inferarea dinamica de invarianti (utilizand instrumentul Daikon integrat in instrumentul ECHOS), invarianti ce au fost apoi utilizati in detectia de violari ale principiului de proiectare LSP. In urma acestui experiment se pare ca generarea de teste exlusiv pe baza strategiei “TwoCallGenerator” este suficienta in vederea detectiei violarilor LSP in toate cazurile avute in vedere. Evident, si generatoarele mai complexe (ex, MultipleCallGenerator) sunt utile, insa sunt mai costisitoare din punctul de vedere al timpului de executie (a se vedea sectiunea anterioara).

Un aspect diferit avut in vedere la evaluarea instrumentului ECHOS a constat in analiza exactitatii de filtrare utilizand analiza statica a invariantilor inferati dinamic. In acest scop, au fost analizati manual un numar de 111 postconditii rezultate in urma rularii instrumentului Daikon, in vederea identificarii corectitudinii lor in raport cu realitatea din codul sursa pentru care au fost inferatii. In continuare, am aplicat filtrul de invarianti din cadrul instrumentului ECHOS. Ca rezultat am determinat precizia respectiv completitudinea (recall-ul) pentru filtrarea postconditiilor, acestea fiind de 95% respectiv 96%. Aceste rezultate sunt foarte bune si dovedesc eficienta filtrarii utilizad analiza statica a invariantilor determinati dinamic.

Pasul de adaptare intre codul preluat direct dintr-un mediu real si ECHOS folosit in cadrul acestei evaluari e necesar din cauza instrumentrului de analiza jCute folosit ca fundatie si integrat in ECHOS (acesta nu poate crea instante ale unor clase ce nu au constructor no-arg). Pentru eliminarea acestei limitari va trebui extins modul de generare a codului “stub” necesar pentru rularea instrumentului jCute astfel incat acesta sa fie degrevat de responsabilitatea creerii obiectelor.

1.2 Evaluarea scalabilitatii si utilizabilitatii modulului CoDePro pe o suita de proiecte industriale in Java

Ca parte a Activitatii 1.2 s-a efectuat un studiu pentru a evalua daca modul in care au fost implementate analizele in CodePro scaleaza odata cu cresterea in dimensiune a sistemelor. Rezultatele acestui studiu sunt sintetizate in Tabelul 1.2

Sistemul	Dimensiune (in linii de cod)	Timp de rulare "System Overview" (in secunde)	Memoria Alocata (in MB)
HTML Parser 2.0	62.889	15,7	45
jHotDraw 6.0	91.473	40,3	48
jFreeChart 1.0.13	217.354	45,5	59
ArgoUML 0.24	247.965	83,1	62
Vuze 4.2.0.8	647.419	178	82
Subsistemul Eclipse org.eclipse.jdt.ui	1.413.344	1208	88

Tabel 1.2 – Sinteza rezultatelor studiului de scalabilitate pentru CodePro. Toate masuratorile au fost efectuate pe un sistem MacBook Pro, procesor Intel Core 2 Duo 2.33 GHz, 2 GB RAM avand 1 GB RAM memorie alocata pentru procesul Java

Din tabel se observa ca au fost analizate 6 sisteme Java open-source, foarte cunoscute, dintre care 4 avand peste 200.000 LOC, iar unul avand chiar aproape 1.5 milioane de LOC! Astfel, am putut verifica daca optimizarile de performanta pe care le-am efectuat pentru CodePro. Astfel se observa ca toate analizele de carente de proiectare pe un sistem de 1.5 MLOC dureaza doar 20 minute. Deasemene trebuie remarcat faptul ca pentru sistemul Vuze care are peste 500 KLOC de cod timpul de rulare este doar 3 minute. In privinta memoriei alocate, consumul este extrem de redus. Faptul ca se pot analiza sisteme de 1.5 MLOC fiind ocupati doar 88 MB de RAM este graitor in acest sens. Consumul de redus de memorie este datorat optimizarilor pe care le-am efectuat in aceste sens, astfel la orice moment de timp memoria este ocupata doar cu obiectele necesare precum si cu acele obiecte care sunt comune multor analize, astfel incat in "lupta" dintre memorie si timpul de executie sa se asigure un balans optim. Toate aceste masuratori indica fara drept de apel faptul ca instrumentul software dezvoltat scaleaza satisfacator, si este utilizabil pentru sisteme de mari dimensiuni.

1.3 Initierea unui proiect cu o firma pentru evaluarea metodologiei agile de asigurarea calitatii propuse

Evaluarea directa intr-un cadru industrial a metodologiei propuse de asigurare a calitatii a fost initiata cu *Océ Software SRL*, o diviziune a companiei multinationale *Océ*, avand in vedere multiplele colaborari cu aceasta firma existente deja. *Océ Software SRL* are ca principal obiectiv de activitate dezvoltarea de software, iar in acest context testarea si asigurarea calitatii sistemelor software rezultate reprezinta un angajament important, pentru care aplicarea de tehnici si metode inovatoare este o necesitate bine inteleasa la *Océ Software*
Cele doua parti vor desfasura in comun activitati de colaborare prin care metodele si tehnicile dezvoltate de in acest proiect vor aplicate pe proiecte software aflate in faza de dezvoltare sau de mentenanta la *Océ Software SRL*, rezultând:

– o analiza in privinta calitatii proiectarii pe un set de proiecte software puse la dispozitie de catre *Océ Software*, analiza ce va prezenta si concluzii privind modalitati de imbunatatire a sistemelor analizate.

– evaluarea instrumentelor de analiza folosite, in vederea dezvoltarii si optimizarii functionalitatii acestora.

***Obiectivul 2/2009:* Rafinarea si extinderea analizelor statice**

2.1 Integrarea tehnicilor de detectare a invariantilor algebrici

Aceasta activitate s-a desfasurat in doua contexte distincte. In primul rand, a fost investigata si evaluata folosirea invariantilor algebrici integrat in analizorul static pentru programe C/C++ descris in activitatea 1.1. Infrastructura de analiza LLVM prezinta cadrul SCEV (scalar evolution) care permite solutionarea unor recurente matematice folosind functii larg intalnire (polinomiale, exponentiale, etc). In acest mod se pot obtine invarianti algebrici pentru ciclurile din program care respecta aceste clase de recurenta si expresii matematice care reprezinta valorile variabilelor la iesirea din aceste cicluri.
In aceasta activitate, au fost extinse capabilitatile modulului SCEV pentru a trata cazuri tipice care apar la detectarea erorilor de memorie prin depasirea indicilor de tablou. In acest sens, pentru a determina corectitudinea unui acces la memorie, trebuie cunosctul *offset*-ul pointerului in cadrul zonei accesate, *dimensiunea* zonei de memorie valide, si *lungimea* accesului. Programul manipuleaza o reprezentare si constrangere de validitate pentru fiecare pointer bazate pe formatul SCEV si foloseste invariantii algebrici detectati de infrastructura. Rezultatele sunt combinate cu doua tehnici de optimizare, global value numbering, si scalar replacement of aggregates pentru a realiza un rezolvitor dedicat pentru determinarea validitatii constrangerilor de acces. In prezent, acesta poate determina automat validitatea a cca 40% din accesele de memorie din cadrul unui program, eliminand astfel necesitatea altor verificari.

Al doilea aspect tratat in cadrul acestei activitati a fost adnotarea automata a programelor Java cu invarianti pentru cicluri. Aceasta reprezinta un pas important in vederea verificarii acestora. In cadrul unui proiect de diploma, a fost integrat generatorul de invarianti algebrici Valigator (Kovacs, 2009) bazat pe sistemul simbolic Mathematica intr-un plugin Eclipse pentru anotarea programelor. S-a realizat o analiza preliminara care identifica ciclurile de interes din program, le converteste in formatul necesar detectorului de invarianti algebrici, si foloseste rezultatele generate de acesta pentru anotarea programului cu preconditii si postconditii in format JML. Prin aceasta integrare, specificatiile locale de invarianti generate pentru fiecare ciclu pot fi inlantuite pentru a demonstra proprietati mai complexe despre programe.

2.2 Rafinarea preciziei analizei statice folosind proceduri de decizie

O prima instanta de rafinare a analizei statie, prin folosirea de proceduri de decizie a fost realizata in contextul depasirii de memorie in programe C/C++ (a carei evaluare a fost descrisa la activitatea 1.1). In acest scop, mediul de analiza identifica toate utilizarile de pointeri (incluzand referinte de tablou cu indici), calculeaza constrangerile corespunzatoare unui aces valid si le propaga inapoi pe graful de flux de control al programului pana la punctul de definitie a pointerului, unde prin rezolvarea constrangerii se determina daca accesul e valid sau nu. Pentru obtinerea unor rezultate eficiente si precire au fost comparate trei solutii: calcularea de constrangeri si apelarea ca rezolvitor a bibliotecilor SMT-LIB (satisfiability modulo theories), implementarea directa prin constrangeri poliedrala cu PPL (Parma Polyhedral Library), si realizarea unui rezolvitor propriu pentru constrangerile produse de modulul SCEV (scalar evolution) al LLVM. Desi primele doua solutii, modulare, folosesc biblioteci clasice din literatura, pentru scopul specific si bine definit in aceasta problema, ultima solutie dovedindu-se cea mai performanta, oferind o viteza mare de analiza, ceea ce a constituit unul din primele criterii de proiectare initiala a aplicatiei. Metoda folosita reuseste sa demonstreze complet automat corectitudinea a cca 40% din accesele de memorie din programele de test.

O a doua directie de rafinare a analizelor statice e legata de detectia pierderilor de memorie.
Biblioteca prototip ce implementeaza analiza de identificare a erorilor de alocare a memoriei (implementata in urma activitatii 3.3 din 2008) a fost extinsa cu algoritmi suplimentari care imbunatatesc precizia analizei. Astfel daca implementarea initiala nu lua in considerare apelurile

intre functiile programului (fiind intraprocedurala), versiunea actuala remediaza acest neajuns, analiza devenind interprocedurala. Pentru o mai buna precizie am implementat un algoritm de anliza a pointerilor care ia in considerare contextul de executie (context-sensitive).

Analiza interprocedurala extinde analiza intraprocedurala prin urmarirea propagarii referintelor la memorie in cadrul apelurilor intre functii. Astfel la intalnirea unui apel, algoritmul initiaza analiza functiei apelate pastrand contextul apelului (parametrii actuali). La finalizarea analizei unei functii se identifica referintele care pot parasi contextul functiei fie prin returnarea ca rezultat al functiei sau stocarea lor in variabile globale sau in memorie la locatii primite ca parametru (side-effects). Urmarirea referintelor se realizeaza prin intermediul unui algoritm dependent de context, care identifica referintele echivalente (pointer aliases) in fiecare punct al programului.
Analizele implementate in cadrul acestei activitati sunt relativ costisitoare din punct de vedere computational dar necesare pentru a obtine o precizie acceptabila. In acest sens ordinea in care se aplica diversi algoritmi de analiza devine foarte importanta pentru scalabilitatea globala a instrumentului dezvoltat. Astfel, algoritmul de analiza intraprocedural ce foloseste analiza de pointeri independenta de context se aplica primul datorita performantei ridicate. Pe instantele de folosire a memoriei pentru care acest algoritm nu este suficient de precis, se aplica algoritmul interprocedural cu o analiza a pointerilor mai precisa.

Nr. Fisier	Numar linii de cod	Timp de executie (prima rulare)	Timp de executie (rulare ulterioare)
1	20	2.750	0.269
2	40	2.870	0.253
3	70	4.062	1.580
4	95	3.982	1.066
5	115	3.310	0.440
7	170	6.620	2.140
8	190	3.144	0.502
9	212	3.890	1.310
10	230	3.750	0.792
11	253	4.240	1.614
12	260	4.270	1.737
13	405	3.456	0.474
14	480	5.774	2.977
Proiect	2540	17.956	16.42

Tabelul alaturat prezinta timpii de executie (in secunde) a analizei de pierdere a memoriei (apelata din mediul de dezvoltare Eclipse) masurati de studentul Borca Alexandru in cadrul lucrarii sale de diploma prezentate in Iulie 2009. Programul analizat compara de 4 algortimi euristici de cautare si genereaza grafice si fisiere cu masuratori.

Prima rulare include trei operatii: compilare a surselor, analiza dependentelor si analiza erorilor de alocare. Rularile ulterioare includ doar analiza dependentelor si analiza erorilor de alocare. Analiza fisierului 7 este costisitoare (desi numarul de linii de cod este redus) deoarece fisierul contine functia *main* a carei analiza conduce la analiza intregului program. Fisierul 13, desi are o dimensiune rezonabila, este analizat rapid deoarece contine implementarea algortmilor de trasare a graficelor generate de programul analizat.

Folosind biblioteca de analiza a pierderilor de memorie, am identificat cateva tipuri de erori de alocare pentru care se pot genera si sugestii de corectie (prin introducerea de instructiuni de eliberare a memoriei in fisierele sursa). Aceasta abordare are la baza o analiza statistica a cailor de execute prin program in urma careia se determina cu o anumita probabilitate daca eroarea de alocare va fi corectata prin adaugarea unei instructiuni de eliberare a memoriei la caile de executie care nu elibereaza memoria. Aceasta abordare este inca in stare de prototip iar rezultatele sunt discutabile deoarece analiza modifica semantica programului.

```
#include <stdlib.h>

int sum()
{
    int *ipa=(int *) malloc(2);
Insert a free statement
```

***<u>Obiectivul 3/2009</u>*: Detectia de carente de proiectare in C++**

3.1 Analiza elementelor specifice ale limbajului C++ sub aspectul carentelor de proiectare

Spre deosebire de Java limbajul C++ prezinta cateva elemente care fac analiza programelor scrise in acest limbaj un capitol distinct, si nu doar distincti, ci mai ales complex. In cadrul proiectului am facut o analiza elementelor de specificitate care maresc gradul de complexitate a analizei.

Primul element critic in programele scrise in C++ este dat de faptul ca partea de declaratie a unei clase este separata fizic de implementarea clasei. Astfel, declaratiile sunt plasate in fisierele header (ex fisiere cu extensia *.h, *.hpp) in timp ce implementariile efective se gasesc in alte fisierele (ex fisiere cu extensia *.cc, *.cpp). Aceasta delocalizare intre partea de interfata si cea de implementare adauga un plus de complexitate in analiza pentru ca trebuie compuse fragmentele de informatie procesate din cele doua parti.

Al doilea element pe care l-am identificat ca problematic pentru analiza sistemelor C++ este utilizarea elementelor rezolvate de catre preprocesor, si anume include-urile conditionale si *macro-urile*. Exista o multitudinea de variatii (flavors, dialecte) chiar ale limbajului in sine functie de set-ul de macro-uri folosite; mai mult, exista aplicatii in care programatorii implementeaza chiar logica a aplicatiei prin functii *macro*. Toate acestea, impreuna cu include-urile conditionale creaza una dintre cele mai critice probleme de rezolvat pentru acest tip de sisteme. Astfel, trebuie construit un dictionar de macro-uri pentru ca altfel la analiza codului anumite (in anumite cazuri substantial de multe) simboluri (adica numele macrourilor) nu vor putea fi rezolvate si astfel analiza tinde sa devina incompleta.

In context de mai sus o problema oarecum corelata este cea a tipurilor generice. Intrucat spre deosebire de Java, in C++ tipurile generice (templates) nu sunt constructii propriu-zise de limbaj ci mai degraba sunt extensii de limbaj rezolva tot la nivel de pre-procesor, ele prezinta particularitati si moduri de utilizare aparte fata de modul in care acestea sunt utilizate in Java.

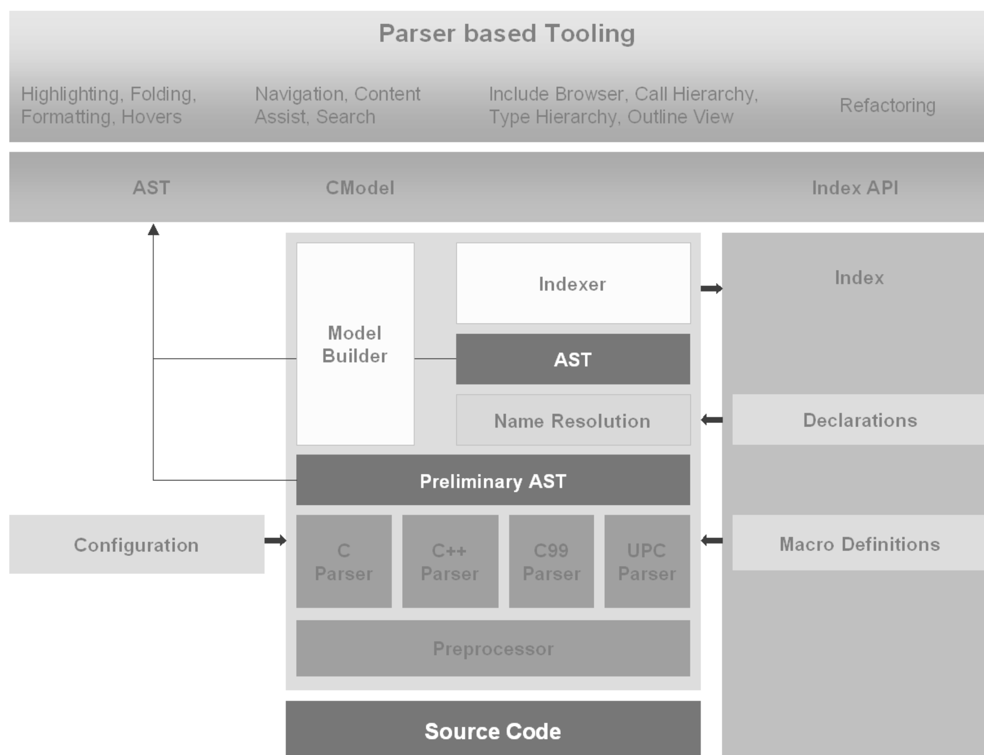
Nu in ultimul rand, la nivelul constructiilor de limbaj se stie ca C++ permite, in plus fata de Java definirea de functii si variabile globale. Acest aspect a implicat nu doar o extindere a meta-modelului folosit de CodePro, ci si o problema mai serioasa, si anume mixtura de code procedural (scris in C) cu cod orientat pe obiecte. Practic, intrucat orice program scris in C este compilabil si ca program C++, granita intre programarea orientata pe obiecte si cea procedurala este destul de greu de trasat in practica, mai ales la sisteme de mari dimensiuni unde cele 2 paradigme de proiectare se amesteca.

3.2 Adaptarea modulului CodePro la analiza proiectelor C++ in Eclipse

CDT sau C/C++ Development tools este un framework pentru dezvoltarea de aplicatii C/C++ in mediul Eclipse. CDT pune la dispozitie programatorilor de plugin-uri Eclipse un model al sistemului care nu ocupa multa memorie si este foarte rapid (vezi Fig 3.2.1). De asemenea CDT ofera posibilitatea de a crea Arbori Abstracti de Sintaxta (AST) din codul sursa. Modelul CDT contine cate un obiect pentru fiecare entitate a codului sursa, pana la nivel de metoda si atribut. Nu contine, de exemplu, cate un obiect pentru fiecare variablia locala, ar fi prea costisitor in termeni de memorie ocupata. Obiectele din modelul CDT sunt wrappuite cu ajutorul clasei CWrapper din CoDePro. Aceasta clasa este centrala in cadrul modelului CoDePro deoarece reprezinta orice entitate a codului sursa : metoda, clasa, atribut, etc. Aici se face legatura intre o entitate a modelului CDT si o entitate a modelului CodePro, fiecare entitate a modelului CoDePro avand o referinta catre o entitate a modelului CDT.

In cadrul sistemului CoDePro facilitatile CDT de model si AST au fost utilizate pentru a extrage informatiile necesare din codul sursa. Pentru aceasta am implementat un set de clase cu ajutorul carora este construit modelul CoDePro. Aceste clase functioneaza in felul urmator : pornind de la un obiect de tipul CWrapper, se obtine obiectul echivalent din modelul CDT, apoi, daca este nevoie, se creeaza AST-ul iar apoi se folosesc de mecanismul de Binding-uri pentru a se ajunge la un element din modelul CDT. De exemplu, pentru a afla toate metodele apelate de metoda curenta, este construit arborele abstract al metodei, sunt vizitate toate nodurile care reprezinta apeluri, iar apoi utilizand mecanismul de Binding se ajunge de la nodurile din arbore la reprezentarea, in modelul CDT, a declaratiei metodei apelate . Declaratia metodei este in final wrapp-uita intr-un obiect de tipul CWrapper. Exista si analize care nu au nevoie de AST, ele isi obtin datele folosind exclusiv modelul CDT.

O problema importanta a fost de a stabili ordinea de parcurgere a fisierelor. Asta deoarece o parcurgere in ordine aleatoare (secventiala) nu ar fi foarte eficienta. Pentru a ne asigura de eficienta maxima, am utilizat PDOM (Persistent Document Object Model). PDOM este un index stocat pe disc creat in mod automat de Eclipse. O alta problema majora ce a trebuit rezolvata a fost aceea a referintelor ambigue. In faza de parsing daca nu sunt gasite referintele (sau daca sunt gasite mai multe referinte pentru acelasi nod) CDT creeaza ProblemBindings. Acest ProblemBinding nu contine nici o legatura catre alt element in cazul in care nu se gasesc referintele cerute, sau contine doua sau mai multe referinte pentru cel de-al doilea caz (cand sunt



gasite mai multe referinte pentru acelasi nod).

Figura 3.2.1 – Arhitectura generala a CDT (Sursa: http://wiki.eclipse.org)

Un alt aspect care a trebuit luat in considerare au fost limitarile specifice CDT. Acestea sunt datorate in mare parte caracteristicilor limbajului C++, care il fac foarte dificil de analizat. Doua

dintre aceste caracteristici care au pus probleme au fost macro-urile si constructiile contiditionale de compilare. In cazul macro-urilor, pot sa apara diferente mari intre ceea ce vede utilizatorul si ceea ce vede parserul. Iar in cazul constructiilor conditionale de compilare, acestea trebuiesc evaluate, ceea ce inseamna ca regiuni inactive din cod sunt ignorate de catre unele facilitati ale CDT.

In cazul in care codul sursa contine erori de compilare, parserul produce un "nod problema". Aceste noduri problema pot fi de patru tipuri: IASTProblemDeclaration, IASTProblemExpression, IASTProblemStatement, IASTProblemTypeId. Parserul CDT este capabil sa continue operatia de parse chiar daca a intalnit o eroare de compilare prin crearea acestor noduri problema. Un exemplu de astfel de problema este cel in care un macro nu este utilizat corect sau un fisier .h nu poate fi gasit.

***Obiectivul 4/2009:* Suport integrat pentru depanarea automata**

4.1 Izolarea atributelor unei intrari de test care exercita o eroare data

Eforturile privind aceasta directie de cercetare au fost concentrate pe suportul pentru depanarea programelor scrise in limbajul C. S-a urmarit automatizarea procesului de depanare, prin oferirea de informatii cat mai precise privind executia eronata a programului si motivele posibile care au dus la aceasta. Ca prim element, aceasta implica instrumentarea programului pentru extragerea informatiilor relevante despre executia care a dus la eroare. Intrucat abundenta detaliilor irelevante este practic la fel de daunatoare ca si lipsa completa de informatie, s-a procedat la o abordare ierahica, evidentiindu-se apelurile de functii ca puncte relevante in program. Codul a fost instrumentat folosind infrastructura CIL de analiza de programe de la UC Berkeley. Programul instrumentat produce o urma de executie care contine toate apelurile de functie, cu valorile pentru parametri si rezultate; un pas suplimentar creeaza valori simbolice pentru valorile de tip adresa care variaza de la o rulare la alta a programului. Aceasta instrumentare e baza pentru o comparatie automata de tip delta debugginng a unei executii de program eronate cu o executie corecta, identificandu-se automat diferentele. Continuand aceasta abordare, intr-un proiect de diploma realizat de Carmen Trutia, prin implementarea unei prelucrari automate a urmelor de executie inregistrate, s-a realizat detectarea primului punct de program (identificat printr-un apel de functie sau de intrare/iesire) care prezinta o manifestare a erorii fata de executia corecta. Abordarea a fost evaluata experimental pe programe (versiuni corecte si cu diferite erori injectate) din suita de teste Siemens, larg folosita pentru raportarea de rezultate in lucrarile de depanare automata.

4.2 Analiza inapoi a caii de eroare pentru identificarea locatiei si a cauzei

In continuarea abordarii descrisa la punctul anterior, identificarea mai precisa a defectului din program implica nu doar detectarea punctului de manifestare a erorii, ci si stabilirea cauzei (instructiunii de program) care a produs acest efect, chiar daca el se manifesteaza mai tarziu.
In acest scop e necesara simplificarea portiunii analizate de program pentru filtrarea instructiunilor relevante (prin efecte datorate fluxului de control si de date) asupra punctului de manifestare a erorilor.
In acest scop, instrumentul de detectare a punctului de manifestare a erorii a fost combinat cu o prelucrare prin *slicing* si o integrare in mediul de analiza statica Codesurfer al firmei Grammatech, unul din cele mai performante din domeniu, pentru care s-a obtinut o licenta academica. A fost implementata o filtrare a programului prin slicing inapoi, pornind de la punctul de manifestare a erorii, care produce un fragment de program continand toate instructiunile care afecteaza in mod potential punctul de eroare. Deoarece procedeul de slicing este unul static, independent de executia programului, fragmentul (slice) relevant obtinut e intersectat apoi cu urma de executie a programului, pentru a rafina prin reducere multimea instructiunilor care ar fi putut in mod real sa cauzeze eroarea. Rezultatele experimentale initiale au aratat ca multimea

de candidati de eroare e redusa cu cca 50%, factor care ar putea fi ameliorat prin rafinarea procedeului de slicing. Se estimeaza ca aceasta directie de cercetare va fi continuata in directia depanarii sistemelor formate din componente, prin metode ierarhice si compozitionale.

***Obiectivul 5/2009*: Generarea de planuri de corectie pentru carente de proiectare**

5.1 Studierea elementelor relevante de context si corelare pentru un set de 6 carente de proiectare

Cele sase carente de proiectare pentru care s-a facut analiza contextului sunt: *Feature Envy*, *Data Class*, *Code Duplication*, *Collapsed Type Hierarchy*, *Explicit State Checks* si *Embedded Strategy*. In urma analizei au rezultat urmatoarele elemente de context relevante: pentru Feature Envy, decizia majora din punct de vedere a restructurarii se refera la masura in care atributele accesate mai sunt utilizate si de catre alte metode, si mai ales alte clase, in afara metodei care prezinta simptomele de *Feature Envy*. In cazul in care metoda respectiva este unica functie ce acceseaza acele date, sau daca este dominanta ca si utilizator, solutia de restructurare va inclina inspre mutarea respectivelor date in clasa ce contine metoda care le acceseaza (cel mai mult). Un alt caz este acela in care metoda *Feature Envy* nu foloseste (aproape) nici un fel de date din clasa in care este definita (aici trebuie inclusa si o analiza a claselor din care acea clasa este eventual derivata). In aceasta situatie, si mai ales daca clasa ce expune datele este *Data Class*, solutia de restructurare implica mutarea metodei in clasa in care sunt declarate datele.

Pentru cazul unei clase *Data Class*, analiza vizeaza deasemenea contextul de utilizare al atributelor neincapsulate, intr-o maniera similara celei descrise pentru *Feature Envy*. Functie de localizarea utilizatorilor atributelele pot fi pur si simplu incapsulate prin modificarea specificatorului de acces (daca sunt folosite doar din acea clasa, sau din interiorul ierarhiei), sau pot fi mutate in clasa unde sunt folosite cel mai intens, sau se pot extrage intr-o noua clasa impreuna cu metodele (sau fragmentele de metode) care le folosesc si pot fi incapsulate in respectiva clasa.
In cazul duplicarii de cod (*Code Duplication*) analiza contextului vizeaza in principal gradul de apropiere intre clasele ce contin metodele cu cod duplicat. Ca urmare distingem trei cazuri: *duplicare intraclass* (strict cu metode apartinand aceleeasi clase), *duplicare sibling* (metodele apartin unor clase apartinand aceleeasi ierarhii de clase), *duplicare external* (intre metode ce apartin la doua sau mai multe clase oarecare). Pe baza acestei distinctii solutia de restructurare poate implica extragerea codului duplicat intr-o metoda comuna a aceleeasi clase, rezolvarea duplicarii de cod prin introducere unui *Template Method* (in cazul duplicarii sibling) si respectiv definirea unei clase in care sa se mute codul din metodele duplicate (in cazul in care clasele nu sunt in nici o relatie speciala).

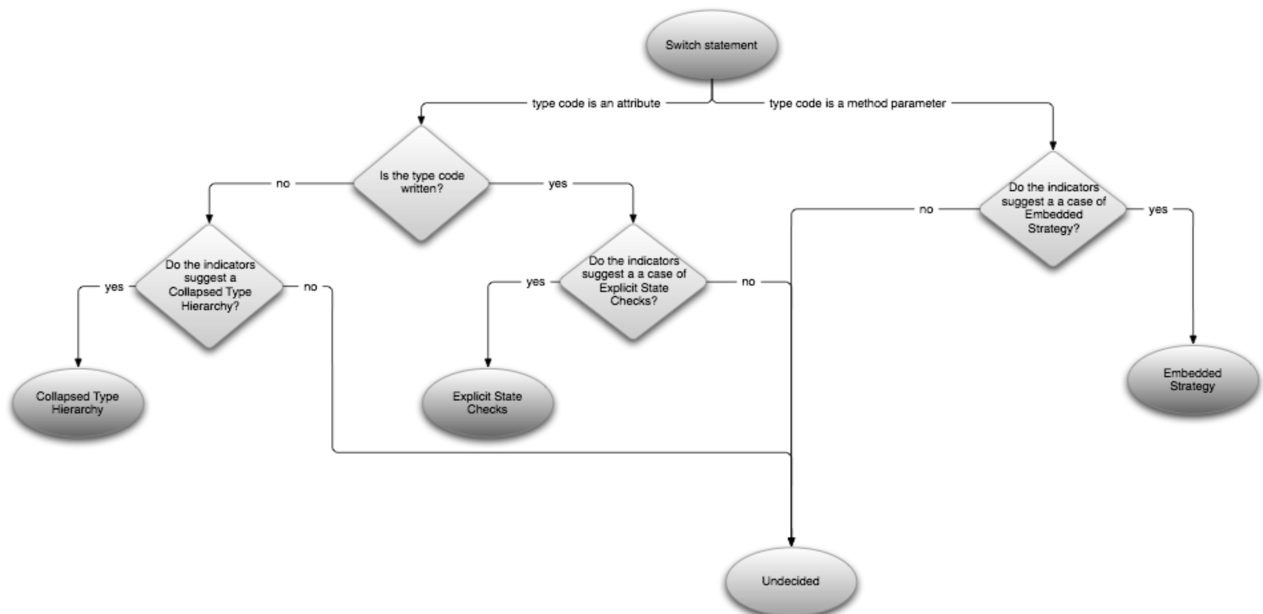


Figura 5.1.1 – Structura de decizie bazata pe informatii contextuale pentru diferitele solutii de restructurare in cazul problemei switch-urilor

Ultimele trei probleme mentionate se refera toate la cazul mai general al problemei switch-urilor (*Switch Statements*) iar informatiile de context sunt cele care releva care din solutii ar trebui aplicata. Figura 5.1.1 evidentiaza arborele de decizie in care informatia contextuala indentifica situatia de restructurarea optima.

5.2 Definirea si implementarea unui generator de planuri de corectie in Eclipse pentru Java

Pentru a putea implementa planurile de corectie pentru diferitele carente de proiectare meta-modelul lui CodePro a trebuit exstins dupa cum rezulta din Figura 5.2.1. Astfel, am extins meta-modelul lui CodePro cu o serie de entitati, proprietati si relatii care opereaza la nivel de metoda. Astfel noul meta-model este compus din entitatile care apar in figura urmatoare, fiind reprezentata de asemenea si relatia de continere intre entitati.

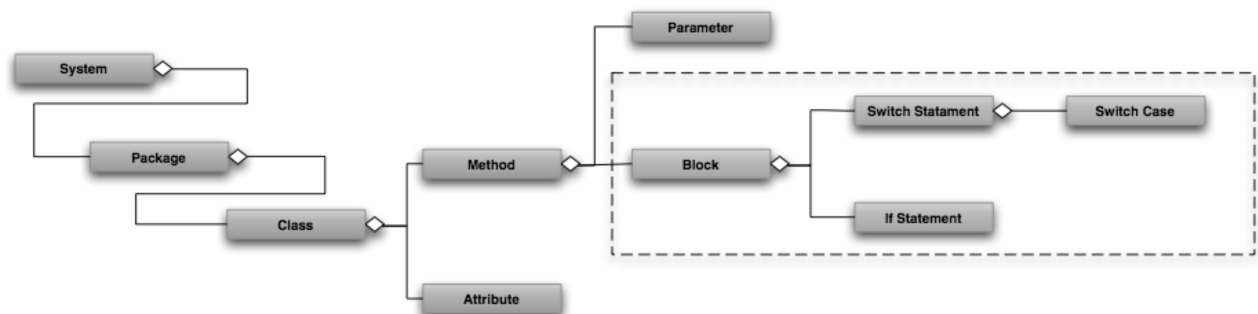


Figura 5.2.1 – Entitatile meta-modelului folosit de componenta de restructurare (cu verde) extind meta-modelul lui CodePro (cu mov)

In termenii acestui meta-model au fost definite strategiile de detectie pentru problema , precum si verificarea preconditiilor pentru o refactorizare. Vom exemplifica in continuare strategia de corectie si cea de detectie pentru una dintre cele patru carente de proiectare si anume *Collapsed Type Hierarchy*. O ierarhie colapsata este o situatie in care o abstractie ”absoarbe” propriile specializari, emuland ierarhia prin verificarea explicita a valorii atribuite variabilei care reprezinta ”tipul” obiectului, si pe care in continuare o vom referi ca fiind selector.

Strategia de detectie in acest caz presupune existenta in cod a cel putin unei constructii switch sau if-else echivalente, care nu foloseste verificare de tip, selectorul constructiei switch fiind un

atribut al clasei, care este comparat cu o serie de constante simbolice. Un exemplu pentru aceasta structura patologica il reprezinta cel din Figura 5.2.2.

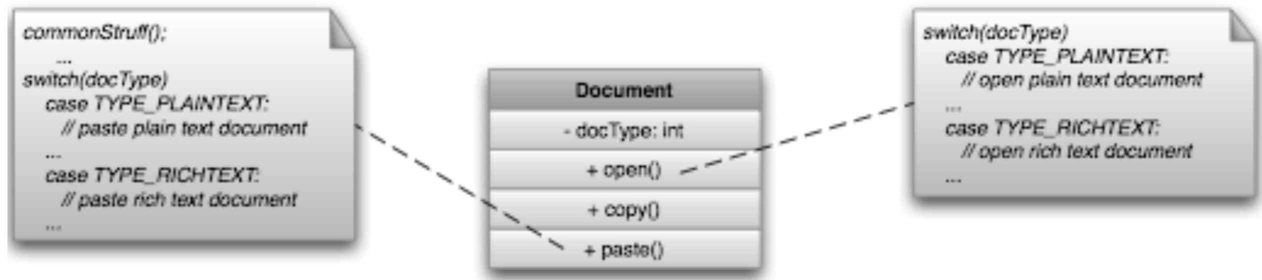


Figura 5.2.2 – Exemplu tipic pentru Collapsed Type Hierarchy

In termenii exemplului din figura putem exemplifica pasii efectuati de CodePro pentru a exemplifica strategia de detectie si cea de corectie. Cele doua refactorizari pe care le vom efectua pentru corectia carentei de proiectare sunt:

- Inlocuirea selectorului cu subclase (*Replace Type Code with Subclasses*)
- Inlocuirea expresiilor conditionale cu polimorfism (*Replace Conditional With Polymorphism*)

Fiecare din aceste refactorizari sunt compuse dintr-o serie de transformari atomice. De exemplu primul refactorizare presupune parcurgerea a patru etape: (i) incapsularea selectorului, (ii)crearea unei metode creationale (*factory method*), (iii) crearea unei subclase pentru fiecare dintre valorile pe care le ia selectorul si intr-un final (iv) eliminarea selectorului si orice referinta la acesta.

In fiecare dintre subclasele nou create va exista o metoda getter care suprascrie metoda devenita abstracta din clasa initiala si care returneaza valoarea corespunzatoare fiecarei subclase (adica constanta mentionata anterior).

Pe langa cele doua refactorizari care a fost exemplificata in aceasta sectiune, CodePro detecteaza situatiile in care poate sa aplice o serie de alte trei strategii de corectie, acestea fiind: inlocuirea expresiilor conditionale cu tiparul Strategie (*Strategy*), inlocuirea expresiilor conditionale cu tiparul Stare (*State*) si mutarea unei metode (*Move Method*). Deasemenea sunt adresate integral cazurile de restructurare pentru carenta de proiectare *Feature Envy*.

Recunoasterea rezultatelor stiintifice

Pentru realizarile sale, Dr. Radu Marinescu a fost premiat in 2009 cu IBM John Backus Award, acordat cercetatorului la nivel mondial care „a facut cel mai mult pentru a ameliora productivitatea programatorilor, prin instrumente, metodologii sau alte tehnici.” Acest premiu, cel mai important acordat de IBM, este la prima editie in 2009, si jurizarea e facuta de

Vizite si colaborari de cercetare

Grupul de cercetare si-a continuat colaborarea cu alte colective internationale, astfel, dr. Marius Minea a fost propus in comitetul de management pentru doua actiuni COST: Formal verification of object-oriented software (IC0701) si Rich Model Toolkit – an infrastructure for reliable computer systems (IC0901). De asemenea, dr. Minea a intrat intr-un grup de actiune pentru scrierea unei proiect de european, cu parteneri din Belgia, Germania si Spania privind metode si tehnici pentru dezvoltarea de sisteme distribuite adaptive si predictibile, care implica o componenta de proiectare si verificare cu metode formale. Dl. Minea a participat la prima intalnire de scriere a proiectului la Aachen/Leuven, urmand a doua, la Karlsruhe in Germania.

Cercetatorii cu experienta din proiect au participat ca membri in comitete de program la conferinte internationale, dupa cum urmeaza:
dr. Radu Marinescu:
WCRE 2009 - 16th Working Conference on Reverse Engineering
MODELS/UML 2009 - ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems
ICCP 2009 - IEEE 5th International Conference on Intelligent Computer Communication and Processing
ICSM 2009 - 25th IEEE International Conference on Software Maintenance
ICPC 2009 - IEEE 17th International Conference on Program Comprehension
ENASE 2009 - 4th Working Conference on Evaluation of Novel Approaches to Software Engineering
CSMR 2009 - 13th European Conference on Software Maintenance and Reengineering

dr. Marius Minea:
SAVBCS 2008: 7th International Workshop on Specification and Verification of Component-Based Systems
FM 2009: 16th International Symposium on Formal Methods
SYNASC 2009: 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing

Diseminarea Rezultatelor

Au fost redactate o serie de articole care au fost acceptate la o serie de conferinte internationale in domeniu (cotate ISI Proceedings si/sau INSPEC), precum si la o serie de worksop-uri internationale co-locate cu aceste conferinte, dupa cum urmeaza:

[1] Petru Florin Mihancea. Towards a Reverse Engineering Dataflow Analysis Framework for Java and C++, Proceedings of 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2008)- Tool Demonstration Section, Timisoara, Romania, IEEE Computer Society Press, 2009

[2] Petru Florin Mihancea, Radu Marinescu. Discovering Comprehension Pitfalls in Class Hierarchies, Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), IEEE Computer Society Press, ISBN 978-07695-3589-0, pag. 7-16, 2009

[3] Peter Bulychev, Marius Minea. An evaluation of duplicate code detection using anti-unification. Proceedings of the 3rd International Workshop on Software Clones, held as workshop of IEEE CSMR 2009

[4] Dan Cosma, Radu Marinescu. Restructuring Object-Oriented Distributed Systems: An Impact-Driven Approach. Proceedings of the 5th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP 2009). IEEE Computer Society Press, 2009

[5] Dan Cosma, Radu Marinescu. Representing Object-Oriented Distributed Systems to Focus the Process of Reverse Engineering. Proceedings of the 5th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP 2009). IEEE Computer Society Press, 2009

[6] Daniel Ratiu, Radu Marinescu, Jan Juerjens. The Logical Modularity of Programs. Proceedings of the 16th IEEE Working Conference on Reverse Engineering (WCRE 2009), IEEE Computer Society Press, 2009.

Un articol suplimentar va aparea in lucrarile unei conferinte internationale in octombrie 2009:

[7] D. Dig, M. Tarce, C. Radoi, M. Minea, R. Johnson. ReLooper: Refactoring for Loop Parallelism in Java. Companion to the 24rth Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, 2009

Au fost publicate de asemenea doua carti cu rezultate de cercetare pe tematica proiectului:

[1] Radu Marinescu. Software Metrics for Quality Assessment. Editura Politehnica, Timisoara, 184 pagini, ISBN 978-973-625-885-5, 2009

[2] Cristina Marinescu. Towards Understanding and Quality Assessment of Enterprise Software Systems. Editura Politehnica, Timisoara, 139 pagini, 2009. ISSN: 1842-7707 ISBN: 978-973-625-847-3

7 septembrie 2009 **conf. dr. ing. Marius Minea**