Провера програмских преводилаца помоћу улаза једнаких по модулу

Семинарски рад у оквиру курса
Верификација софтвера
Математички факултет

Лазар Васовић
mi16099@alas.matf.bg.ac.rs

3. децембар 2020.

Сажетак

У тексту је приказан рад *Compiler validation via equivalence modulo inputs* (2014) аутора Вуа Леа, Мердада Афшарија и Џендунга Суа са Одсека за рачунарске науке Универзитета Калифорније у Дејвису. Уз теоријско проширење излагања из рада, објашњени су појмови метаморфног тестирања, улаза једнаких по модулу и *EMI* варијанти програма. Током истраживања је откривено 147 грешака у популарним компилаторима *GCC*-у и *LLVM*-овом Клангу, али је тај број временом порастао на 1634, од чега је 1076 поправљено. Систематично је представљен и начин реализације, те резултати и примери примене.

Кључне речи — метаморфно тестирање, погрешна компилација, метаморфне варијанте програма, аутоматска провера компилатора

Садржај

1 Увод

Тестирање је најпознатији и најкоришћенији начин провере исправности, односно уопштено **верификације софтвера**. Иако је овај приступ у начелу једноставан, често га спутавају два важна проблема – проблем пророчишта и проблем поузданог скупа тестова [1].

1.1 Проблеми тестирања

Проблем пророчишта чине случајеви када није могуће непосредно проверити исправност резултата теста, пошто није доступна информација о томе шта је очекивани излаз. **Проблем поузданог скупа тестова** чине особине које треба да задовољавају тест примери како би, уколико прођу, могло да се закључи да је стечен задовољавајући степен поверења у програм који се тестира. Када се ова два проблема удруже, поставља се питање како проверити оно што је добијено и има ли онда сврхе тестирати нешто што је на први поглед нетестабилно.

Наведене потешкоће су изражене у многим важним областима, као што су уграђени системи, нумеричка израчунавања, рачунарска графика, машинско учење [2, 3]. У питању су свакодневно коришћене класе програма, тако да развијање посебног начина тестирања није нешто са ограниченим доменом. Штавише, то би помогло и у ситуацијама када је пророчиште доступно, али је скупо користити га. Још једна врло значајна област у којој није могуће лако тестирати резултате јесте **конструкција компилатора**. Није сигурно користити програм који је преведен непоузданим преводиоцем (он може бити бескористан), тако да је тачност компајлера од суштинске важности, и неопходно је спровести посебне мере за бригу о квалитету и исправности [4].

Како је то често у рачунарству, решење лежи у олакшавању услова – кад то није могуће, одустаје се од захтева за **непосредном** провером резултата, те допушта **посредна** провера. Генерише се скуп **изворних** тест примера, након чега се на посебан начин генеришу **накнадни** тест примери. Уместо провере поклапања резултата између добијеног и очекиваног излаза, сада се тестира поклапање два добијена излаза – једног на основу полазног скупа тест примера, другог на основу додатног скупа – која су у посебној **вези**. За овакав приступ очигледно није потребно пророчиште, већ само улази и везе.

1.2 Метаморфно тестирање

Добар пример описаног типа везе јесу **метаморфне релације**, а приступ тестирању заснован на њиховом коришћењу назива се **метаморфно тестирање** [1, 3, 5]. Често својство програма је да неке промене улаза могу да предодреде поједине карактеристике новог излаза у односу на стари. Управо овакве везе називају се метаморфним и могу се употребити за дефинисање посебног начина тестирања.

Примера ради, проста метаморфна релација над низовима природних бројева је да проширење низа (низ је улаз) новим елементом повећава укупан збир његових елемената (сума је излаз). У том случају, изворни тест пример је случајно генерисан низ бројева. Накнадни тест пример одговара изворном, са додатим једним или више бројева. Неопходна провера следи из предложене релације тоталног уређења – да ли је сума елемената накнадног низа већа од суме полазног тј. изворног низа. Одговарајући тест случај дат је псеудокодом 1.

Тест случај 1: Метаморфна релација уређења

```
1 niz := generisiSlucajanNiz();
2 stariZbir := sumiraj(niz);
3 dodajSlucajanBroj(niz);
4 noviZbir := sumiraj(niz);
5 assertGT(noviZbir, stariZbir);
```

Ипак, метаморфне релације **уређења** су често превише опште, па лако могу нематиˇ велики значај у тестирању. У претходном примеру нпр. постоји могућност да сумирање уместо збира бројева враћа дупло већу вредност, и тај пропуст не би био примећен предложеним тестом. Сигурније би било проверити да ли је збир повећан баш за вредност новог елемента. Ово чак и уопштава тест са низова природних бројева на низ било каквих бројева, чак и реалних, не рачунајући евентуалну грешку у рачунским операцијама у покретном зарезу (ово се може поправити приближном провером, са одређеним степеном толеранције). Нови тест случај дат је псеудокодом 2. Иако је овај пример тривијалан, јасно указује на то да су метаморфне релације **еквиваленције** јаче [3, 5], и оне посебно добијају на значају у сложенијим применама.

Тест случај 2: Метаморфна еквиваленција

```
1 niz := generisiSlucajanNiz();
2 stariZbir := sumiraj(niz);
3 noviBroj := dodajPaVratiSlucajanBroj(niz);
4 noviZbir := sumiraj(niz);
5 assertEQ(noviZbir, noviBroj + stariZbir);
```

Најјаче метаморфне релације, за које је лако **аутоматизовати** генерисање накнадних тест примера и само тестирање, јесу релације еквиваленције засноване на улазима једнаким по модулу. О њима ће касније бити више речи, а засад је довољно рећи да је у тој парадигми излаз **једнак**, па би тест случај могао бити као у псеудокоду 3.

Тест случај 3: Улази једнаки по модулу

```
1 ulaz1 := generisiIzvorniUlaz();
2 izlaz1 := primeniAlgoritam(ulaz1);
3 ulaz2 := generisiNaknadniUlaz(ulaz1, izlaz1);
4 izlaz2 := primeniAlgoritam(ulaz2);
5 assertEQ(izlaz1, izlaz2);
```

1.3 Le et al. (2014)

Главна тема овог текста јесте приказ рада ***Compiler validation via equivalence modulo inputs*** **(2014)** аутора Вуа Леа (*Vu Le*), Мердада Афшарија (*Mehrdad Afshari*) и Џендунга Суа (*Zhendong Su*) са Одсека за рачунарске науке Универзитета Калифорније у Дејвису, САД (*Department of Computer Science, University of California, Davis,*

USA) [6]. Аутори предлажу метод провере исправности програмских преводилаца заснован на метаморфном тестирању. Притом користе јаке метаморфне релације еквиваленције засноване на улазима једнаким по модулу. Већ су наглашени тежина и важност тестирања компилатора, па су резултати изложени у раду утолико значајнији за целу област. Поред **теоријског** значаја, аутори су дошли и до великог **практичног** успеха – током истраживања је откривено **147 грешака** у популарним компилаторима *GCC*-у и *LLVM*-овом Клангу, а број је временом порастао на **1634**, од чега је **1076** поправљено. Претходни сличан покушај резултовао је откривањем само две грешке [1, 7].

Уз саме улазе једнаке по модулу (енгл. *equivalence modulo inputs, EMI*) у смислу појма, дефинишу се *EMI* варијанте програма (накнадни улази), као и стохастички алгоритам за њихово проналажење на основу тока извршавања тј. профила изворних тест примера. Идеја је, дакле, у **комбиновању** динамичког извршавања тест програма (изворног улаза за компилатор) и статичке анализе његовог синтаксног стабла ради генерисања метаморфно еквивалентних улаза. Алгоритам је заснован на стратегији проучи и измени (енгл. *profile and mutate*). Очекивано, захтева се једнакост у понашању извршивог кода (излаза компилатора) свих *EMI* варијанти улазног програма.

Рад *Compiler validation via equivalence modulo inputs* написан је као пројекат у оквиру Групе са посебним интересовањем за програмске језике (*Special Interest Group on Programming Languages, SIGPLAN*) Удружења за рачунарске машине (*Association for Computing Machinery, ACM*) и као такав је представљен у јуну 2014. на годишњој конференцији *PLDI*. Услед значаја и успеха у проналажењу грешака награђен је као **изузетан** (*Distinguished Paper Award*) [8]. Заправо, од 1998. године, када је метаморфно тестирање први пут предложено, па све до данас, ово је једна од **најуспешнијих** примена те стратегије [1]. Истраживање је делимично стипендирано грантовима америчке Националне фондације за науку, која ради под окриљем Владе САД.

2 Садржај рада

У наставку је систематично изложен садржај рада аутора Леа и осталих. По увођењу кључних идеја и појмова, приказан је начин реализације предложеног алгоритма тестирања помоћу алата Орион, насталог у оквиру истраживања. Напослетку су издвојени најважнији резултати и примери примене идеје. Подразумевано, поглавље је у целости засновано на приказаном раду, без употребе других извора, уз напомену да је тек **накнадно** схваћено да су улази једнаки по модулу и варијанте према њима заправо метаморфне релације, односно варијанте. У време писања рада, аутори нису уочили ову везу, па из тог разлога, примера ради, у чланку ни у једном тренутку није поменут појам метаморфне релације или метаморфног тестирања [1].

2.1 Кључне идеје и појмови

Централни појам приказаног рада јесу **улази једнаки по модулу** (енгл. *equivalence modulo inputs, EMI*). У питању је једноставан концепт за проверу исправности компилатора, са широком могућношћу примене. Према идеји метаморфног тестирања, неопходно је на основу неког изворног тест примера генерисати накнадни. Притом се захтева

да стари и нови програм – тест примери за компилатор су програми – буду у метаморфној релацији, и то у јакој релацији еквиваленције. То значи да излаз компилатора (извршиви код) у неку руку мора бити једнак, при чему је неопходно осмислити паметан критеријум.

Формалније гледано, задатак је на основу програма P (изворног тест примера) генерисати програм Q (накнадни тест пример). За метаморфну релацију еквиваленције узима се једнакост излаза компилатора – извршивих кодова. Проблем је, међутим, што није лако упоредити два извршива програма, тако да је то неопходно урадити на неком ограниченом домену. За те потребе дефинише се коначан скуп улаза I (модуло скуп), уз напомену да се овде не ради о улазима за тестирани преводилац, већ за програме који служе као тест примери. За програме P и Q каже се да су једнаки по модулу I уколико важи $(\forall i \in I)P(i) = Q(i)$, где је са $P(i)$ означен излаз настао извршавањем програма P, преведеног тестираним преводиоцем, над улазом $i \in I$. Уколико за било који улаз из модуло скупа једнакост не важи, метаморфна релација је **нарушена**, те постоји **грешка у превођењу**.

Подразумева се да је I подскуп пресека домена програма P и Q, како би извршавање имало смисла. Специјално, уколико програм не прихвата улаз, једнакост по модулу своди се на потпуну семантичку једнакост. Још један посебан случај су недетерминистички програми, који при различитим извршавањима не дају исте излазе за исте улазе, па их не би требало разматрати без претходне детерминизације.

Накнадни улази једнаки по модулу називају се ***EMI* варијантама** изворног тест програма. Њихова снага лежи у томе што могу да **циљају** на специфичне фазе анализе и оптимизације разматраног преводиоца и тиме га интензивно тестирају (енгл. *stress-test*) у циљу откривања скривених грешака. Иако се на први поглед може чинити да идеја не обећава превише, пошто еквиваленција важи само на врло ограниченом скупу улаза, ипак није тако. Компилатор свакако мора да прође кроз целокупну статичку анализу и оптимизације приликом прављења извршивог кода за *EMI* варијанте, јер преведени програм мора да ради нешто над свим улазима. Упркос семантичкој еквивалентности на модуло скупу, то се не захтева за углавном већи број других улаза, па овакве варијанте могу да имају знатно различит статички **ток података**. Како су ток података и уопште контрола тока главни чиниоци који одређују које ће тачно оптимизације и на који начин бити примењене, *EMI* варијанте форсирају **различит** правац извршавања унутар компилатора, а самим тим и емитовање различитог кода. Када се узме у обзир да захтевају излаз који се исто понаша на модуло скупу улаза, омогућавају лаку проверу једнакости различитих стратегија оптимизације, простим тестирањем једнакости излаза.

Идеја *EMI* варијанти је кључна за оправдавање употребе метаморфног тестирања програмских преводилаца, а опет довољно једноставна да би се релативно брзо имплементирала и аутоматизовала. Притом ју је, осим на компилаторе, могуће применити и на друге статичке анализаторе и трансформаторе кода написаног у било ком програмском језику. Добра страна је и што ју је могуће применити на било какав код, па је тако вероватно пожељно подесити домен над којим се преводилац тестира. Проширењем ове замисли може се добити начин откривања грешака насталих приликом компилације у готовом софтверу, што је од посебног значаја за све критичне примене.

2.2 Реализација алгоритма

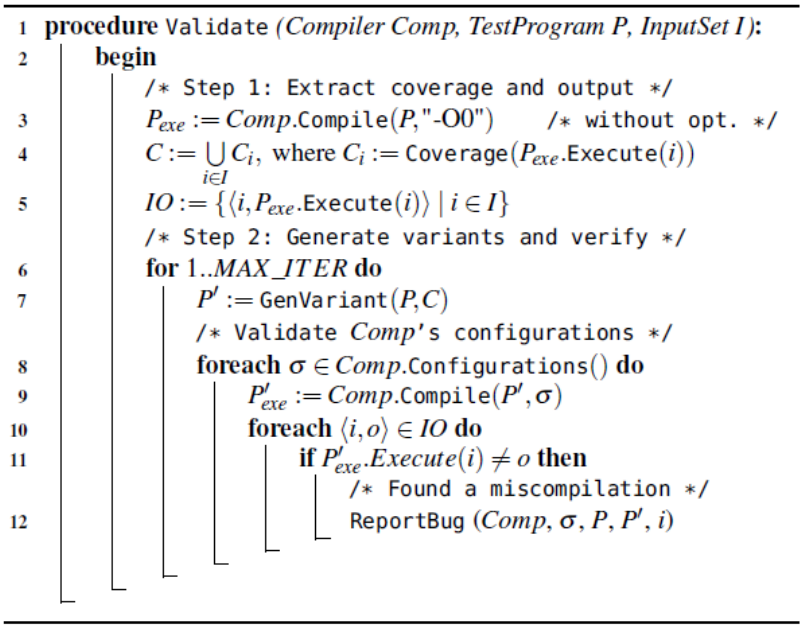
Аутори су предложени облик тестирања имплементирали кроз алат **Орион**. Овај алат замишљен је као ловац на компајлерске багове, па је зато назван по џину-ловцу из грчке митологије. Како је простор *EMI* варијанти неког програма огроман, било је неопходно осмислити реалистичан начин за њихово генерисање. Управо је ово главни задатак Ориона, док је провера поклапања излаза лакши део посла. Псеудокод алгоритма за генерисање варијанти дат је на слици 1.

```
1  function GenVariant (TestProgram P, Coverage C): Variant P′:
2      begin
3          P′ := P
4          foreach s ∈ P′.Statements() do
5              PruneVisit (P′, s, C)
6          return P′
7  procedure PruneVisit (TestProgram P′, Statement s, Coverage C):
8      begin
           /* Delete this statement when applicable */
9          if s ∉ C and FlipCoin(s) then
10             P′.Delete (s)
11             return
           /* Otherwise, traverse s's children */
12         foreach s′ ∈ s.ChildStatements() do
13             PruneVisit (P′, s′, C)
```

Слика 1: Генерисање *EMI* варијанти

Главнина овакве реализације метаморфног тестирања програмских преводилаца јесте стратегија **проучи и измени** (енгл. *profile and mutate*). Како сам назив сугерише, идеја је проучити изворни програм (параметар P типа $TestProgram$) на начин који за резултат има скуп покривених наредби које треба задржати неизмењеним (параметар C типа $Coverage$), а затим произвољно изменити непокривене наредбе. Прецизније, пролази се кроз сваку наредбу програма редом (променљива s типа $Statement$), пратећи апстрактно синтаксно стабло (AST) програма. Уколико наредба није покривена, брише се са неком вероватноћом (одређеном функцијом $FlipCoin$), док се у супротом наставља низ стабло. Резултат овога је накнадни програм. Како се у истраживању ограничило на измену брисања, односно поткресивања стабла, без додавања или другог начина мутације кода, алтернативни назив ове стратегије је **проучи и поткреши** (енгл. *profile and prune*).

Остаје питање на који се тачно начин одређује које наредбе треба задржати неизмењеним. Одговор је веома једноставан – оне које су извршене при покретању над неким од улаза из модуло скупа. Притом је начин генерисања модуло скупа, као и самог изворног програма, произвољан, о чему ће конкретније бити говорено у оквиру коментара резултата и примера. Псеудокод Орионове главне процедуре дат је на слици 2, а може се уочити да концептуално одговара тест случају 3.



Слика 2: Орионова главна процедура

Укратко, може се поделити на три корака. Први се састоји од превођења тестираним преводиоцем (параметар $Comp$ типа $Compiler$) и извршавања изворног програма (параметар P типа $TestProgram$) над модуло скупом улаза (параметар I типа $InputSet$), са циљем прикупљања података о покривености и очекиваног излаза (линије 3–5). Други корак у петљи генерише накнадне тест примере помоћу претходно изложене процедуре (линије 6–7), док трећи преводи и покретањем проверава резултате EMI варијанти, уз евентуално понављање процеса за различита подешавања преводиоца (линије 8–12).

Што се тиче првог корака, неопходно је одредити тачан начин за одређивање покривености кода, односно скупа наредби које су извршене приликом покретања изворног програма над улазима из модуло скупа. Ово је најбоље урадити профајлирањем [9], па би трећи и можда најпрецизнији назив за описану стратегију генерисања могао бити **профајлирај и поткреши**. Конкретно, техником профајлирања могуће је пребројати колико пута је извршена свака наредба програма, па тако оне чији је бројач на крају рада већи од нуле упадају у скуп покривених, док остале спадају у „мртав код“, који се може уклонити без промене понашања над критичним улазима из модуло скупа. Орион овај корак имплементира позивањем алата *gcov*, доступног у оквиру *GCC*-а. Како би код успешно био инструментализован (допуњен) бројачима, неопходно га је превести без оптимизација. Након извршавања, доступне су датотеке са срачунатом статистиком.

Када су у питању детаљи другог корака, који је већ концептуално описан, његова имплементација заснована је на *LLVM*-овој библиотеци *LibTooling*, која је део Кланга у ужем смислу. Уз њену помоћ лако се формира AST изворног програма, коме се придружују подаци доби-

јени профајлирањем. Стабло у чворовима садржи наредбе, док свака наредба као децу може имати зависне наредбе или изразе. Примера ради, наредба условног гранања типа *IfStmt* има три детета – услов као израз, грану када је услов испуњен као наредбу и грану када услов није испуњен као наредбу. Сложена наредба типа *CompoundStmt* може имати произвољан број деце, док нпр. празна наредба типа *NullStmt* нема ниједно. Наредба се сматра извршеном уколико је *gcov* линију у којој се налази њен први токен прогласио за покривену или уколико је неко њено дете извршено. Брисање наредбе брише цело подстабло, па резултујући накнадни програм остаје синтаксно исправан.

Сама главна процедура имплементирана је као спој неколико скриптова љуске (енгл. *shell script*), који редом сакупљају покривеност и излазе, генеришу варијанте, упоређују резултате извршавања и пописују евентуална непоклапања. Језик другог корака је *C++*. Свеукупно, Орион се састоји од око петсто линија скрипт кода и хиљаду линија *C++* кода. Ово га издваја у односу на друге генераторе програма за тестирање преводилаца за језик *C*, међу којима је најпознатији *Csmith*, који је написан у позамашних тридесетак хиљада линија *C++* кода.

2.3 Резултати и примери

Грешке до којих долази у раду програмских преводилаца најчешће се деле у две групе различитог значаја. Мање проблематичан је **пад преводиоца** (енгл. *compiler crash*), који се догоди када компилатор није у стању да преведе неки програм, те стане са радом и не генерише излаз. Значајнију класу чине **грешке у превођењу** (енгл. *miscompilation*), које настају када компајлер произведе погрешан извршиви код, који не одговара изворном, извршући тако намеру програмера. Овакви проблеми тихо настају, па их је тешко приметити, и та чињеница може бити обесхрабрујућа по питању провере исправности.

Пример важне карактеристике грешака у превођењу, која их издваја од других типова багова у софтверу, јесте да оне воде до пропуста у другим програмима. У уобичајеним условима, грешка у неком програму је локалног типа. Ипак, у случају преводиоца, грешка која изазива генерисање неадекватног извршивог програма практично се **пропагира** на тај програм, изазивајући неочекивано понашање у њему. Ова непријатност је тешка за локализацију, пошто ретко који програмер кривца тражи у преводиоцу, што је очекивано, јер су и такве грешке ретке. И саму грешку у превођењу и њен узрок често је тешко пронаћи, јер у доста ситуација она погађа неки редак пут извршавања, за који се деси да не буде посећен приликом развоја. Наравно, све ово највише погађа критичне системе, за које је неопходна апсолутна поузданост. По наведеним карактеристикама могло би се рећи да грешке у превођењу више личе на хардверске багове него на софтверске.

Алат Орион управо се показао као врло ефикасан у откривању погрешне компилације, и то је његова основна примена и резултат. Као конкретне примере, аутори наводе два илустративна, по један за *LLVM* и *GCC*. Једна грешка у **Клангу** откривена је радом са програмом 1, који је заправо део стандарне свите тестова за *GCC*. Програм илуструје рад са малим структурама, функцијом која преноси те структуре и системском функцијом *abort* за насилан прекид рада.

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
long l) {
  if (x.c != 10) abort();
  if (x.d != 20) abort();
  if (x.e != 30) abort();
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) abort();
  if (z.c != 12) abort();
  if (z.d != 22) abort();
  if (z.e != 32) abort();
  if (l != 123) abort();
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
}
```

Тест пример 1: Програм који се исправно преводи

Овај програм исправно преводе оба компилатора и он овде представља изворни тест пример. Очекивано понашање је нормалан прекид рада, пошто није испуњен ниједан услов у функцији. Међутим, помоћу Ориона је малим изменама генерисана *EMI* варијанта која је била окидач за баг у Клангу. Наиме, приликом профајлирања изворног програма, закључено је да ниједан *abort* није покривен, што значи да је било који од њих могуће уклонити. Орион је то урадио на следећи начин, генеришући варијантни програм 2 као накнадни тест пример.

```
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y, struct tiny z,
long l) {
  if (x.c != 10);
  if (x.d != 20) abort();
  if (x.e != 30);
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31);
  if (z.c != 12) abort();
  if (z.d != 22);
  if (z.e != 32) abort();
  if (l != 123);
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
```

```
22    x[2].d = 22;
23    x[0].e = 30;
24    x[1].e = 31;
25    x[2].e = 32;
26    f(3, x[0], x[1], x[2], (long)123);
27    exit(0);
28  }
```

Тест пример 2: Програм који *LLVM* погрешно преводи

Очигледно, ова два програма су значењски једнака. Ипак, у време писања приказаног рада постојао је баг у Клангу који је доводио до тога да извршива верзија другог програма, преведена на 32-битној архитектури x86 уз укључене оптимизације, увек позива *abort*. Ради лакшег описа, откривања тачног узрока и исправљања проблема, аутори су развијаоцима *LLVM*-а уз пријаву грешке приложили упрошћену верзију примера 3, која помоћу мање кода показује исто понашање.

```
1  struct tiny { char c; char d; char e; };
2  void foo(struct tiny x) {
3    if (x.c != 1) abort();
4    if (x.e != 1) abort();
5  }
6  int main() {
7    struct tiny s;
8    s.c = 1; s.d = 1; s.e = 1;
9    foo(s);
10   return 0;
11 }
```

Тест пример 3: Упрошћени програм који *LLVM* погрешно преводи

Убрзо је откривено да је узрок овоме био пропуст у *LLVM*-овом **оптимизатору**. Испоставило се да је дошло до сукоба две оптимизације, што је условило неисправну иницијализацију структуре. У оригиналном програму није било проблема пошто је Кланг за функцију *f* закључио да је довољно велика да нема потребе инлајновати је. Пошто се структура преноси као аргумент функције, њена иницијализација се не оптимизује. Супротно томе, мање кода у поткресаној *EMI* варијанти учинило је функцију довољно малом за инлајновање, што је компилатор навело да сада оптимизује рад са структуром.

Конкретно, откривено је да код проблематичног накнадног тест примера оптимизатор прво извршава оптимизацију глобалног бројања вредности (енгл. *Global Value Numbering, GVN*), која између осталог иницијализације структура преводи у једно 32-битно пуњење (учитавање, енгл. *load*). Након тога, примењена је оптимизација скаларне замене агрегата (енгл. *Scalar Replacement of Aggregates, SRoA*), која закључује да је претходно наметнуто 32-битно пуњење недефинисано понашање, пошто оперише изван граница мале структуре, што је збуњује и она затим емитује погрешан код. Конкретно, иницијализација у потпуности бива одбачена, па није неочекивано што поља неиницијализоване структуре не испуњавају услове наметнуте у функцији.

С друге стране, програм 4 издвојен је као накнадни тест пример који је изазвао грешку у компилацији код преводиоца ***GCC 4.8***, како у 32-битној, тако и у 64-битној верзији, док код Кланга није. Изворни тест пример ове *EMI* варијанте направљен је помоћу аутоматског генератора *C* кода *Csmith*, али није приказан због комплексности.

```
int a, b, c, d, e;
int main() {
  for (b = 4; b > -30; b--)
    for (; c;)
      for (;;) {
        e = a > 2147483647 - b;
        if (d) break;
      }
  return 0;
}
```

Тест пример 4: Програм који *GCC 4.8* погрешно преводи

Очекивано је да се извршива верзија овог програма заврши без икаквог рада, пошто се услов друге петље увек евалуира као нетачан. Наиме, у услову је статичка целобројна променљива c, која је на почетку нужно иницијализована на нулу. У складу са тим, ову петљу је оптимизацијом могуће уклонити, а затим је исто могуће урадити и са првом петљом, чије тело остаје празно, без даљег реферисања на променљиву b. Овај пример није правио проблем Клангу, али га је *GCC* са укљученим оптимизацијама преводио у бесконачну петљу.

Испоставља се да је и овде проблем био у лошој оптимизацији. Наиме, оптимизација уклањања делимичних редундантности (енгл. *Partial Redundancy Elimination, PRE*) прогласила је подизраз наредбе доделе $2147483647 - b$ за инваријанту друге петље. Приликом наивног покушаја издизања тог израза изнад петље дошло је до недефинисаног понашања. Како се тад тај израз нашао на месту на ком ће се стварно извршити, оптимизатор је приметио да ће у једном тренутку доћи до прекорачења у означеним целобројним операцијама, што га је збунило и довело до емитовања кода који се не завршава.

Занимљивост је да на нивоу оптимизација $-O2$ сам *GCC* унапред закључује да је дошло до грешке и то јавља кориснику следећом поруком, односно упозорењем – *warning: iteration 5u invokes undefined behavior [-Waggressive-loop-optimizations]*. Наиме, ово експлицитно указује на то да због агресивне оптимизације петље у шестој итерацији долази до недефинисаног понашања. Конкретно, тада ће важити $b = -1$, што стварно доводи до прекорачења у одузимању. Иако проблем постоји и на нивоу оптимизација $-O3$, тада нема упозорења. Наравно, ово није грешка у самом програму тј. *EMI* варијанти, јер је критична операција семантички гледано мртав код, па је сасвим валидна и коректна. Грешка је у оптимизатору, који није смео тај мртав код да оживи издизањем изнад петље чији услов никада није испуњен.

Изворни програм, који није приказан јер има око три хиљаде линија кода, није окидач за овај баг, пошто су у унутрашњој петљи постојале додатне наредбе које су мењале вредност променљиве b, па самим тим оптимизатор није могао да пронађе ниједну инваријанту. Ово је добар пример како предложени алгоритам генерисања *EMI* варијанти може довести до сасвим другачијег тока процеса компилације.

Изложена стратегија примењена је и на алат *CompCert*, формално верификовани оптимизујући преводилац за широк подскуп језика *C99*, слободно доступан за употребу у некомерцијалне сврхе. Донекле очекивано, резултати су изостали, те није пронађен ниједан пример погрешног превођења. Ово потврђује значај **формалне потврде** исправности преводилаца, али ипак треба имати у виду да су такви алати врло скупи за израду, те да подржавају знатно мањи скуп мо-

гућности и оптимитација од популарних и широко коришћених *GCC*-а и *LLVM*-а. Самим тим је и извршиви код слабијих перформанси.

3 Закључак

Програмски преводиоци спадају у најважнији, најсложенији и најшире коришћени софтвер. Управо због тога је провера њихове исправности од изузетног значаја. Ипак, обично је веома тешко открити неке грешке, поготову у домену оптимизација, које сваки напреднији компилатор спроводи. Ово може довести до неисправног излаза у виду извршивог програма који не одговара изворном коду, па самим тим није прихватљиво за критичне програме. Због тога су поједини преводиоци у потпуности формално верификовани. Тај процес је, међутим, веома скуп, тако да се чешће прибегава некој врсти тестирања.

Испоставља се да је провера помоћу улаза једнаких по модулу, предложена у приказаном раду, добар начин за откривање већег броја прикривених грешака. Тај принцип, у суштини заснован на метаморфном тестирају, једноставан је за разумевање и реализацију, и лак за аутоматизацију, а може се уопштити и на друге језике (аутори су се ограничили на *C*), па чак и друге, шире области примене – базе података, интерпретаторе, анализу програма и уопштено трансформационе системе. Додатни напор могао би се уложити у рад са реалним бројевима (аутори разматрају само целе), пошто код њих због грешке у прецизности рачуна у покретном зарезу важе другачије варијанте.

Изложени примери откривених багова приказују како је грешке у превођењу могуће открити на различитим преводиоцима, и то подједнако ефикасно на малим кодовима и онима са хиљадама линија. Ови пропусти наглашавају важност исправне оптимизације, као главног и највећег узрока компајлерских грешака, али и лоше стране недефинисаног понашања, као нечег што по стандарду језика *C* преводиоцу дозвољава да емитује произвољан код. Показују да скривене грешке настају како при појединачним оптимизационим пролазима (пример *PRE*), тако и у њиховој интеракцији (пример комбинације *GVN* и *SRoA*). *EMI* варијанте лако могу открити оба типа проблема.

За крај ваља напоменути следеће – иако је у овом тексту у потпуности приказана суштина рада *Compiler validation via equivalence modulo inputs*, ипак постоји још детаља за чије се откривање читаоцу препоручује анализа самог рада. Пример додатних информација до којих се може доћи увидом у чланак – где пронаћи изворне тестове, како упростити накнадне тестове а успут задржати грешку, колико варијанти генерисати. Дати су и многи квантитативни резултати – статистичка расподела пронађених грешака по преводиоцу, типу и конфигурацији превођења (заставицама – типу архитектуре, нивоу оптимизација, верзији преводиоца, компоненти преводиоца која изазива баг), утицај грешака на брзину компилације... Пример запажања је да су брже исправљање грешке у *GCC*-у од оних у *LLVM*-у. Пример занимљивости је да је за један од откривених багова у *GCC*-у накнадно утврђено да је погрешним превођењем изазвао дотад необјашњени проблем у апликацији *MPlayer* за репродукцију видео-снимака. Ово је уједно и добра потврда да стварно није бесмислено вештачким тест примерима откривати и исправљати ситне пропусте у превођењу – никад се унапред не може знати када ће и како направити проблем у некој реалној апликацији, сасвим могуће много значајнијој од видео-плејера.

Литература

[1] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys, Association for Computing Machinery, New York*, 51(1), January 2018. https://www.cs.hku.hk/data/techreps/document/TR-2017-04.pdf.

[2] Растко Ђорђевић. Метаморфно тестирање – семинарски рад у овкиру курса Верификација софтвера. Математички факултет, Универзитет у Београду, доступно на интернет страници: http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/02_testiranje/31_RastkoDjordjevic_MetamorfnoTestiranje.pdf.

[3] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016. доступно на интернет страници: http://www.cs.ecu.edu/reu/reufiles/read/metamorphicTesting-16.pdf.

[4] Милена Вујошевић Јаничић. Конструкција компилатора – основно о компајлерима – слајдови. Математички факултет, Универзитет у Београду, доступно на страници: http://www.prevodioci.matf.bg.ac.rs/kk/2020/predavanja/01_uvod_slajdovi.pdf.

[5] Милена Вујошевић Јаничић. Верификација софтвера – технике тестирања – слајдови. Математички факултет, Универзитет у Београду, доступно на интернет страници: http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/02_testiranje/02_testiranje_slajdovi.pdf.

[6] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *SIGPLAN Notices, Association for Computing Machinery, New York*, 49(6):216–226, June 2014. доступно на: https://web.cs.ucdavis.edu/~su/publications/emi.pdf.

[7] Q. Tao, W. Wu, C. Zhao, and W. Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *2010 Asia Pacific Software Engineering Conference*, pages 270–279, 2010. делом доступно на: https://ieeexplore.ieee.org/document/5693203.

[8] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs, 2014. презентација рада: https://people.inf.ethz.ch/suz/publications/emi-talk.pdf.

[9] Милена Вујошевић Јаничић. Верификација софтвера – дебаговање, профајлирање – слајдови. Математички факултет, Универзитет у Београду, доступно на интернет страници: http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/03_dinamicka_analiza_slajdovi.pdf.