UDC: 004.4'2 INFO M: str. 14-20

РЕГРЕСИОНА ВЕРИФИКАЦИЈА СОФТВЕРА КОРИШЋЕЊЕМ СИСТЕМА LAV
REGRESSION VERIFICATION BY SYSTEM LAV

Милена Вујошевић Јаничић

РЕЗИМЕ: Алати за аутоматску верификацију софтвера постигли су огроман напредак и постају све шире практично употребљиви. Однедавно се практично разматра и један нови вид верификације софтвера — регресиона верификација. У регресионој верификацији, слично као у регресионом тестирању, функционална исправност програма проверава се тако што се испитује еквивалентност са програмом за који се претпоставља да је исправан. Наравно, регресиона верификација, онда када је применљива, даје много поузданије и дубље закључке него регресионо тестирање. У овом раду описаћемо укратко основне појмове верификације софтвера и показати да верификацијски систем LAV може успешно да се користи за регресиону верификацију. Могуће примене су у развоју софтвера (као алтернатива регресионом тестирању) али и у образовању, за аутоматску евалуацију студентских програма.

КЉУЧНЕ РЕЧИ: верификација софтвера, регресиона верификација, симболичко извршавање, проверавање ограничених модела, аутоматска евалуација студентских радова

ABSTRACT: Software verification tools have achieved enormous progress and are becoming practically usable to a broad set of users. Over the last years, regression verification, a special sort of software verification, is also practically considered. In regression verification, similarly as in regression testing, functional correctness of a program is checked by analyzing its equivalence with the program which is assumed to be correct. Of course, regression verification, when applicable, provides much more reliable and deeper conclusions than regression testing. In this paper we briefly describe basic concepts of software verification and show that the verification system LAV can be successfully used for regression verification. Possible applications are in software development (as an alternative to regression testing), but also in education, for automatic evaluation of students' programs.

KEY WORDS: software verification, regression verification, symbolic execution, bounded model checking, automated evaluation of students' programs

1 УВОД

Грешке у софтверу коштају светску економију милијарде долара годишње [27]. Неке софтверске грешке могу имати и материјално немерљиве последице као, на пример, грешке у функционисању софтвера нуклеарних електрана или грешке у функционисању софтвера уређаја који се користе у здравству. Због свега тога, испитивање исправности софтвера тј. верификција софтвера један је од кључних задатака у развоју софтвера.

Постоје различити приступи верификацији софтвера, од динамичких приступа (испитивања исправности софтвера приликом његовог извршавања, најчешће путем тестирања) до строго формалних математичких приступа засновних на статичком испитивању исправности софтвера (без извршавања кода). Приликом развоја софтвера, избор и примена одговарајућег приступа испитивању исправности софтвера зависи од великог броја фактора, а најчешће се своди на тестирање као на најједноставнији и најприступачнији метод испитивања исправности. Међутим, тестирањем се не може утврдити да грешака у софтверу нема, што значи да чак и темељно тестирани софтвер и даље у себи може садржати фаталне грешке. С друге стране, употреба строго формалних математичких метода у верификацији омогућава велики број предности по питању поузданости софтвера, али често захтева значајне финансијске и временске ресурсе, као и високостручне експерте.

Да би се омогућила широка примена формалних математичких метода у испитивању исправности софтвера, неопходно је поједноставити употребу и аутоматизовати процес формалне верификације. Аутоматизација процеса верификације обухвата прецизно математичко моделовање основних градивних јединица програма и тока контроле програма, аутоматско конструисање формула које одговарају условима исправности програма, као и аутоматско доказивање својстава конструисаних формула. Аутоматизација процеса верификације садржи мноштво изазова и проблема, како теоријске тако и практичне природе. Пре свега, због неодлучивости проблема заустављања, не постоји општи аутоматизован начин за проверавање да ли је нека наредба програма достижна, па тиме ни да ли је исправна, односно да ли је сам програм исправан. То значи да није могуће направити програм који би потпуно аутоматски, у коначном времену, користећи коначне ресурсе, могао да утврди исправност прозвољног програма потпуно прецизно, тј. без лажних упозорења или пропуштених грешака.

Упркос свим препрекама и теоријским ограничењима, током последњих година направљен је огроман прогрес у области аутоматске верификације софтвера и нови верификацијски алати користе се све чешће и све шире. У то значајне ресурсе улажу и велике ИТ компаније као што су Мајкрософт, ИБМ и Интел. У годинама које долазе, примена верификацијских алата ће бити све више и вероватно ће их рутински користити већина програмера. Верификација ће се примењивати у разноврсним формама, пре свега у најчешћим – верификацији у односу на расположиву спецификацију и у откривању багова, али у специфичним као што је регресиона верификација о којој ће бити речи у овом чланку.

2 МЕЂУЈЕЗИЦИ, ТЕОРИЈЕ И РЕШАВАЧИ У ВЕРИФИКАЦИЈИ

Формална верификација програма често се ослања на међујезике који олакшавају анализу исправности програма трансформисањем изворног кода у одговарајући погодан облик, као и на рад решавача који утврђују ваљаност формула које се генеришу приликом анализе исправности програма.

Међујезици у верификацији. Анализа изворног кода програма написаног на вишем програмском језику отежана је разним факторима, на пример, присуством сложених израза, имплицитних конверзија типова и бочних ефеката. Због тога се анализа програма обично изводи над кодом који се добија трансформацијом оригиналног кода у погодни облик [22].

LLVM платформа је популаран софтверски алат, слободно доступан и отвореног кода, који се може користити за трансформацију програма у међујезик погодан за анализу. Почетна верзија LLVM пројекта имала је за циљ да обезбеди подршку за компилацију произвољног програмског језика [23, 24]. Временом је тај пројекат прерастао у низ различитих потпројеката који се широко користе у производњи и верификацији софтвера, као и у истраживачке сврхе[1]. Основни LLVM потпројекат чине библиотеке које омогућавају разне врсте трансформација, анализе и оптимизација програма независно од изворног кода и циљне архитектуре, као и библиотеке које омогућавају генерисање извршног кода за многе популарне архитектуре микропроцесора. Ове библиотеке раде над језиком који се зове LLVM међујезик. LLVM има развијене приступне компоненте за програмске језике C, C++, Ada и Fortran. Постоје разни спољни пројекти који преводе друге програмске језике (на пример, програмске језике Python, Ruby, Haskell, Java, D, Pure, Scala и Lua) у LLVM међујезик. У верификацији софтвера, LLVM се користи, на пример, у оквиру алата KLEE [8], LLBMC [25] и SAFECode [15].

Теорије за моделовање програма и решавачи. У аутоматској верификацији софтвера, потребно је задати опис услова исправности програма у терминима неке одлучиве теорије за коју постоје ефикасне процедуре одлучивања. Неке од теорија које се често користе за моделовање понашања и услова исправности програма су исказна логика, линеарна аритметика (теорија логике првог реда у којој се користе функцијски симболи: 0 и 1 арности 0, − арности 1, + арности 2 и предикатски симбол ≤ арности 2), аритметика бит-вектора (покрива целобројне операције које може да врши процесор над низовима битова) и теорија неинтерпретираних функција (теорија која не задаје никаква ограничења за функцијске симболе и која користи само аксиоме једнакости). Детаљне дефиниције и описи теорија могу се наћи у литератури [4].

[1] LLVM пројекат је добио престижну награду ACM System Software Award за 2012. годину.

Приликом аутоматске анализе исправности програма генеришу се услови исправности чију је ваљаност, у оквиру изабране теорије, потребно проверити адекватним системом. Систем за проверавање ваљаности може бити саставни део верификацијског алата, тј. може бити специјализован за услове исправности које верификацијски алат генерише или може бити независан од верификацијског алата. Као екстерни системи за проверавање ваљаности често се користе SMT решавачи. Задовољивост у односу на теорију (енг. Satisfiability Modulo Theory, скраћено SMT) проблем је одлучивања задовољивости у односу на основну теорију T описану у класичној логици првог реда са једнакошћу. Решавачи за овај проблем називају се SMT решавачи. SMT решавачи имају многе примене, посебно у верификацији хардвера и софтвера. Неки од најпознатијих SMT решавача су Z3 [13], Yices [16], MathSAT [7], STP [18] и Boolector [6].

3 МЕТОДИ И АЛАТИ ЗА ВЕРИФИКАЦИЈУ СОФТВЕРА

Основни методи статичке анализе исправности софтвера су проверавање модела, апстрактна интерпретација и симболичко извршавање.

Проверавање модела. Метод верификације у којем се систем (хардверски или софтверски), који је потребно верификовати, описује коначним аутоматом, а спецификација се задаје у терминима темпоралне логике назива се проверавање модела (енг. model checking) [10]. Доступна стања модела се затим систематски обилазе са циљем да се докажу услови задати спецификацијом. У случају да доказ не успе, генерише се контрапример који нарушава услове спецификације. Основни проблем метода проверавања модела је његова скалабилност — скуп могућих стања експоненцијално расте са повећањем броја променљивих. Метод *проверавања ограничених модела* (енг. bounded model checking) заснива се на ограничавању дужине путање стања која се проверава [3]. Уколико се грешка не пронађе за фиксирану вредност дужине путање стања k, тада се k повећава све док се не пронађе грешка, док модел не постане превелик за анализу или док се не достигне вредност која је горња граница дужине могућих путања кроз систем (чиме је модел верификован). Неки од верификацијских алата који се заснивају на проверавању ограничених модела су CBMC [9], ESBMC [11] и LLBMC [25]. Ови алати су првенствено намењени анализи ANSI-C и C++ програма, при чему су алати CBMC и ESBMC јавно доступни и користе се за анализу програма за уграђене системе (енг. embedded software).

Симболичко извршавање. Техника симболичког извршавања (енг. symbolic execution) одавно је позната [21], али постала је популарна тек последњих година. Симболичким извршавањем се конструишу симболички изрази који описују одређене путање кроз програм. Симболички изрази омогућавају испитивање исправности програма за све могуће вредности на путањи која се анализира, за

разлику од тестрирања, где се једним тест примером проверава исправност програма само за фиксиране вредности. Симболичко извршавање је веома прецизна анализа и њоме се остварује велика предност у односу на технике динамичког тестирања. Међутим, број могућих путања кроз програм често је превелик да би се могао систематски потпуно претражити. Због тога се симболичко извршавање најчешће користи за проналажење грешака у програму, а не и за потпуну верификацију програма. Најпопуларнији верификацијски алати који се заснивају на симболичком извршавању су KLEE [8] и PEX [28]. KLEE је јавно доступан алат отвореног кода за симболичко извршавање C програма и за аутоматско генерисање тест примера. PEX је Мајкрософтов јавно доступан алат за аутоматско генерисање тест примера и проналажење грешака у програмима, а може се користити за све језике .NET платформе.

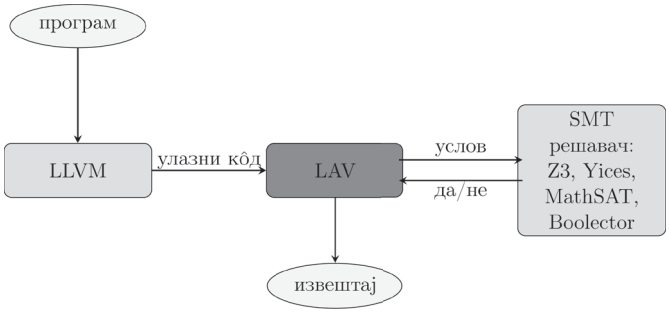
Апстрактна интерпретација. Теорија апстрактне интерпретације (енг. abstract interpretation) је теорија апроксимације формалне семантике програма, тј. математичког модела могућих понашања програма [12]. Апстрактна интерпретација дефинише формалне методе апроксимације скупова и апроксимације операција над скуповима. Апроксимације се заснивају на коришћењу монотоних функција на парцијално уређеним скуповима и посебно на аутоматском рачунању фиксних тачака програма, које одговарају петљама или рекурзивним позивима функција у оквиру програма. Апстрактна интерпретација има низ могућих примена, а одликује је изузетна склабилност: технике апстрактне интерпретације могу се применити и на програме од неколико стотина хиљада линија кода [17]. Скалабилност ових техника последица је губљења прецизности приликом моделовања, што често резултује великим бројем лажних упозорења. Неки од најпопуларнијих верификацијских алата који се заснивају на апстрактној интерпретацији су комерцијални алати Astree [5], Polyspace [14] и Coverity SAVE [2].

4 СИСТЕМ LAV

LAV[2] је систем за статичку анализу, генерисање и проверу услова исправности императивних програма [31, 30]. Комуникације система LAV са окружењем приказана је на слици 1. LAV је намењен првенствено анализи програма написаних на програмском језику C, али, пошто за трансформацију улазних програма у форму која је погодна за анализу користи LLVM палтформу, може се применити и за анализу програма написаних у другим процедуралним програмским језицима. За испитивање ваљаности формула, LAV има подршку за рад са неколико SMT решавача (Boolector, MathSAT, Yices и Z3). Сам систем обухвата моделовање понашања програма, конструкцију услова исправности програма, трансформисање конструисаних услова у формуле одговарајућих теорија, комуникацију са решавачем и генерисање одговарајућег извештаја за корисника.

LAV комбинује описивање понашања програма исказним променљивама, симболичко извршавање и проверавање ограничених модела. Индивидуални блокови LLVM међукода моделују се формулама логике првог реда које се конструишу симболичким извршавањем. Релације између блокова се моделују исказним променљивама. Формуле које описују понашања блокова кода заједно са релацијама између блокова комбинују се и на основу њих конструишу се формуле које описују понашање програма. Те формуле користе се за формирање услова исправности појединачних наредби програма. Услови исправности шаљу се на проверу SMT решавачу који покрива одговарајућу комбинацију теорија. Подржане теорије, у којима је могуће моделовати услове исправности програма, су теорија линеарне аритметике, теорија бит вектора, теорија неинтерпретираних функција и теорија низова. Наредба може имати статус безбедне (извршавање наредбе не доводи до грешке), неисправне (извршавање наредбе сигурно доводи до грешке), небезбедне (извршавање наредбе може да доведе до грешке) и недостижне (до извршавања наредбе никада неће доћи).

LAV је имплементиран у програмском језику C++, јавно је доступан и отвореног кода [29].



Слика 1: *Комуникација система LAV са окружењем.*

5 ФУНКЦИОНАЛНА ЕКВИВАЛЕНТНОСТ ПРОГРАМА И ПРИМЕНЕ

Функционална исправност једног програма може бити, и најчешће јесте, формулисана у терминима прецизне спецификације. Насупрот томе, функционална исправност једног програма може бити формулисана у терминима понашања другог програма. На овај начин, појам исправности је слабији (јер, на пример, не укључује предуслове и постуслове функције), али зато постоје следећи позитивни аспекти: није потребно формулисати спецификацију и, генерално, испитивање еквивалентности требало би да буде рачунски мање захтевно од функционалне верификације (јер би комплексност анализе могла да зависи само од разлика између два програма, а не и од њихових величина).

[2] LAV је акроним за LLVM Automated Verifier.

Идеја испитивања да ли су два програма еквивалентна се широко и интензивно користи у *регресионом тестирању*. Регресионо тестирање је тестирање које има за циљ да открије да ли су уведене нове грешке у програм након прављења измена које се односе на побољшања текућег система, „закрпе“ или промену конфигурације система. У оквиру регресионог тестирања, врши се поређење функционалности система пре и након уведених измена и разматра се еквивалентност првобитног и новонасталог програма.

Проверавање еквивалентности је најкоришћенији приступ у верификацији хардвера. Формално, аксиоматски засновано, испитивање еквивалентности програма разматрао је Тони Хор још током 1960-их [20], али прогрес на овом плану је био ограничен и постојеће методе често нису биле практично применљиве. Скорашњи приступи проблему формалног испитивања еквивалентности програма пружају нове могућности [26, 19].

Појам еквивалентности програма се може описати помоћу појма парцијалне еквивалентности и појма узајамног заустављања. Програми P_1 и P_2 су *парцијално еквивалентни* ако враћају исти резултат за једнаке улазне вредности за које се оба заустављају, док су *узајамно заустављајући* ако се за једнаке улазне вредности или оба заустављају или се ниједан не зауставља. Програми P_1 и P_2 су *тотално еквивалентни* ако су парцијално еквивалентни и узајамно заустављајући. Проблеми да ли су два програма парцијално еквивалентна, узајамно заустављајућа или тотално еквивалентна су неодлучиви.

Формално испитивање еквивалентности програма, тј. *регресиона верификација* може се користити у развоју софтвера у индустрији, аналогно као регресионо тестирање, али са далеко већим степеном поузданости. Регресиона верификација може се применити и у образовању са циљем прецизне и поуздане аутоматске евалуације студентских радова, при чему се врши проверавање еквивалентности студентског и наставничког решења.

5. 1 *Регресиона верификација у аутоматској евалуацији студентских програма*

Аутоматска евалуација програма важна је и за наставнике и за студенте [32]. За наставнике, аутоматско оцењивање тестова и испита корисно је јер им омогућава да имају више времена за остале активности са студентима. За студенте, аутоматска евалуација омогућава брзу повратну информацију о квалитету програма, што је веома важно за процес учења. Брзо добијање повратних информације је нарочито корисно студентима уводних курсева програмирања. На овим курсевима, програмирање представља посебно тежак изазов. Студенти имају мало или нимало знања о основним алгоритамским и програмским темама, а често имају и дубоке заблуде [33].

Користи од аутоматске евалуације програма су још значајнији у контексту учења уз помоћ рачунара. Велики број водећих светских универзитета нуди бројне курсеве који се могу пратити преко Интернета. Број студената који прате ове курсеве је у порасту и мери се милионима [1]. У оваквим курсевима, процес подучавања преузима рачунар, контакт са наставником је минималан, па су брзи и квалитетни подаци о резултатима рада студента посебно пожељни.

Верификацијски алати могу се успешно користити за откривање грешака у студентским радовима, па тиме и у побољшању стандардних метода за квалитетну аутоматску евалуацију студентских радова [32]. Примена регресионе верификације софтвера у овом контексту може да додатно повећа прецизност и унапреди аутоматску евалуацију студентских програма. Регресиона верификација се, у овом контексту, односи на утврђивање функционалне еквивалентности решења које даје наставник и решења које даје студент. Наредним примерима илустроваћемо како се систем LAV може користити за испитивање еквивалентности функција.

5. 2 *Функције без петљи и рекурзија*

Регресиона верификација функција које не садрже петље ни рекрузивне позиве релативно је једноставна и даје веома прецизне резултате. Размотримо, као пример, две имплементације алгоритма за одређивање минимума три цела броја које су приказане на слици 2.

```
int minA(int x, int y, int z)
{
   int m = x;
   if (y < m) m = y;
   if (z < m) m = z;
   return m;
}
```

```
int minB(int a, int b, int c)
{
   if ( a <= b ) {
      if ( a <= c )
         return a;
      else
         return c;
   }
   else {
      if ( b <= c )
         return b;
      else
         return c;
   }
}
```

Слика 2: *Две имплементације функције која одређује минимум три цела броја.*

Да би се утврдила функционална еквивалентност функција `minA` и `minB` у оквиру система LAV, потребно је верификовати програм који је приказан на слици 3.

```
#include <stdio.h>
int main()
{
   int p, q, r;
   scanf("%d%d%d", &p, &q, &r);
   assert(minA(p,q,r) == minB(p,q,r));
}
```

Слика 3: *Програм који је потребно верификовати да би се утврдила функционална еквивалентност за* `minA` *и* `minB`.

Функцијски позив `assert`, програма који је приказан на слици 3, односи се на проверу једнакости повратних вредности предложене две функције (за произвољне улазне вредности). За сваку функцију, систем LAV генерише формулу која (симболички) описује сва њена могућа извршавања. Ова формула садржи информације о свим мо-

гућим путањама кроз функције, као и о односима између промењљивих на тим путањама. Илустрације ради, ове формуле за функције `minA` и `minB` приказане су на сликама 4 и 5 (обе формуле су поједностављене ради лакше читљивости). У оквиру формула, променљива TR_y^x означава прелазак из блока инструкција *x* у блок инструкција *y*, променљива A_x означава да се блок инструкција *x* извршава, a_0^x означава вредност променљиве *a* у блоку *x* на уласку у блок, док a_1^x означава вредност променљиве *a* на изласку из блока. Блокове инструкција на основу С кода одређује LLVM трансформатор, а детаљан опис генерисања формула може се наћи у оквиру описа система LAV [30]. Формула чију ваљаност LAV испитује и која одговара услову еквивалентности функција `minA` и `minB`, генерише се од формула које описују ове функције и од једнакости њихових повратних вредности.

$$A_0 \wedge (m_0^0 = x_0^0) \wedge ((A_0 \wedge (y_0^0 < x_0^0)) \Leftrightarrow TR_1^0) \wedge ((A_0 \wedge (\neg(y_0 < x_0))) \Leftrightarrow TR_2^0) \wedge ((TR_1^0 \vee TR_2^0) \Leftrightarrow A_0) \wedge (m_1^1 = y_0^0) \wedge (TR_1^0 \Leftrightarrow A_1) \wedge (A_1 \Leftrightarrow T R_2^1) \wedge (m_0^2 = m_0^2) \wedge (TR_2^1 \Rightarrow (m_0^2 = m_1^1)) \wedge (TR_2^0 \Rightarrow (m_0^2 = m_1^0)) \wedge ((TR_2^1 \vee TR_2^0) \Leftrightarrow A_2) \wedge ((A_2 \wedge (z_0^0 < m_0^2)) \Leftrightarrow TR_3^2) \wedge ((A_2 \wedge (\vee(z_0^0 < m_0^2))) \Leftrightarrow TR_4^2) \wedge ((TR_3^2 \vee TR_4^2) \Leftrightarrow A_2) \wedge (m_1^3 = z_0^0) \wedge (TR_3^2 \Leftrightarrow A_3) \wedge (A_3 \Leftrightarrow TR_4^3) \wedge (m_1^4 = m_0^4) \wedge (x_1^4 = x_0^0) \wedge (y_1^4 = y_0^0) \wedge (z_1^4 = z_0^0) \wedge (TR_4^3 \Rightarrow (m_0^4 = m_1^3)) \wedge (TR_4^2 \Rightarrow (m_0^4 = m_1^2)) \wedge ((TR_4^3 \vee TR_4^2) \Leftrightarrow A_4)$$

Слика 4: *Формула која описује могућа извршавања функције* `minA`.

$$A_0 \wedge ((A_0 \wedge (a_0^0 \leq b_0^0)) \Leftrightarrow TR_1^0) \wedge ((A_0 \wedge (\neg(a_0^0 \leq b_0^0))) \Leftrightarrow TR_4^0) \wedge ((TR_1^0 \vee TR_4^0) \Leftrightarrow A_0) \wedge (TR_1^0 \Leftrightarrow A_1) \wedge ((A_1 \wedge (a_0^0 \leq c_0^0)) \Leftrightarrow TR_2^1) \wedge ((A_1 \wedge (\neg(a_0^0 \leq c_0^0))) \Leftrightarrow TR_3^1) \wedge ((TR_2^1 \vee TR_3^1) \Leftrightarrow A_1) \wedge (retval_1^2 = a_0^0) \wedge (TR_2^1 \Leftrightarrow A_2) \wedge (A_2 \Leftrightarrow TR_7^2) \wedge (retval_1^3 = c_0^0) \wedge (TR_3^1 \Leftrightarrow A_3) \wedge (A_3 \Leftrightarrow TR_7^3) \wedge (TR_4^0 \Leftrightarrow A_4) \wedge ((A_4 \wedge (b_0^0 \leq c_0^0)) \Leftrightarrow TR_5^4) \wedge ((A_4 \wedge (\neg(b_0^0 \leq c_0^0))) \Leftrightarrow TR_6^4) \wedge ((TR_5^4 \vee TR_6^4) \Leftrightarrow A_4) \wedge (retval_1^5 = b_0^0) \wedge (TR_5^4 \Leftrightarrow A_5) \wedge (A_5 \Leftrightarrow TR_7^5) \wedge (retval_1^6 = c_0^0) \wedge (TR_6^4 \Leftrightarrow A_6) \wedge (A_6 \Leftrightarrow TR_7^6) \wedge (a_1^7 = a_0^0) \wedge (b_1^7 = b_0^0) \wedge (c_1^7 = c_0^0) \wedge (retval_1^7 = retval_0^7) \wedge (TR_7^6 \Rightarrow (retval_0^7 = retval_1^6)) \wedge (TR_7^5 \Rightarrow (retval_0^7 = retval_1^5)) \wedge (TR_7^3 \Rightarrow (retval_0^7 = retval_1^3)) \wedge (TR_7^2 \Rightarrow (retval_0^7 = retval_1^2)) \wedge ((TR_7^6 \vee TR_7^5 \vee TR_7^3 \vee TR_7^2) \Leftrightarrow A_7))$$

Слика 5: *Формула која описује могућа извршавања функције* `minB`.

LAV генерише формуле и доказује еквивалентност ове две функције коришћењем теорије линеарне аритметике и доказивача Z3 за 0.008 секунди на рачунару са процесором Intel Core i3-2350M на 2.30GHz и са 3.8GB меморије. Уколико се користи теорија бит-вектора, време потребно да се докаже еквивалентност претходних функција је 0.16 секунди.

Претходни приступ може се применити на све програме који не садрже петље, али и на програме који садрже петље за које постоји горња граница броја могућих извршавања. У случају да у програму постоји петља за коју постоји горње ограничење броја извршавања, тада се та петља може размотати максималан број пута и описани приступ се може применити на програм након елиминације петљи размотавањем.

Да би се детаљније анализирала практична употребљивост система LAV у аутоматском испитивању еквивалентности студентских и наставничких решења, коришћен је корпус студентских програма са испита и тестова универзитетског уводног курса програмирања који су прошли аутоматско тестирање спроведено у оквиру оцењивања радова [32]. Из овог корпуса, који садржи 266 студентских радова, издвојено је 145 радова који не садрже рекурзивне позиве функција, као ни петље за које не постоји горња граница размотавања. Да би се извршила провера еквивалентности наставничких и студентских решења, потребно је извршити једноставне трансформације студентских програма у облик који је погодан за верификицију (на пример, функцију `main` треба преименовати, да не би постојале две функције са истим именом, док је резултате који се штампају потребно превести у повратне вредности функција). Такође, потребно је и написати програме са одговарајућим позивима функције `assert` (налик на програм приказан на слици 3) које LAV треба да верификује. Анализирани корпус као и систем LAV јавно су доступни [29].

LAV успешно доказује еквивалентност за 143 студентска програма са одговарајућим наставничким решењима. За 2 програма LAV је утврдио да нису еквивалентни са наставничким решењима и ти студентски програми заиста садрже грешке. Ове грешке промакле су аутоматском тестирању, што показује да испитивање функционалне еквивалентности може да побољша квалитет и прецизност аутоматске евалуације програма. Време потребно за анализу свих 145 програма укупно је 4 минута и 50 секунди.

5. 3 *Функције са рекурзијом или петљама*

Као што је већ речено, функције које садрже петље за које постоји горња граница броја могућих извршавања могу се, размотавањем петљи, трансформисати у функције које не садрже петље и у том случају могу бити аутоматски верификоване регресионом верификацијом или верфикацијом у односу на спецификацију. Међутим, аутоматско утврђивање еквивалентности функција са произвољним петљама представља велики изазов и у општем случају није решив проблем. Постоји веза између петљи и рекурзивних позива, па се разматрање петљи може свести на разматрање рекурзивних функцијских позива.

```
unsigned EuklidA(unsigned a,
unsigned b)
{
   unsigned g;
   if (b == 0)
      g = a;
   else  {
      a = a % b;
      g = EuklidA(b, a);
   }
   return g;
}
```

```
unsigned EuklidB(unsigned x,
unsigned y)
{
   unsigned z;
   z = x;
   if (y > 0)
      z = EuklidB(y, z % y);
   return z;
 }
```

Слика 6: *Две имплементације Еуклидовог алгоритма за одређивање највећег заједничког делиоца два природна броја.*

У одређеним ситуацијама, регресиона верификација рекурзивних функција може да се заснива на коришћењу неинтерпретираних функција [26, 19]. Ову технику илустроваћемо на примеру Еуклидовог алгоритма за одређивање највећег заједничког делиоца два природна броја. Две рекурзивне имплементације овог алгоритма приказане су на слици 6. Математичком индукцијом, по другом аргументу функције, може се доказати да су те две функције еквивалентне:

Индуктивна основа. Уколико је други аргумент једнак 0, прва функција враћа *g*, где је $g = a$, а друга *z*, где је $z = x$. Дакле, када је $a = x$ важи *EuklidA(a, 0) = EuklidB(x, 0)*.

Индуктивни корак. Претпоставимо да функције враћају исте вредности за све вредности другог аргумента мање од *n* и докажимо да функције враћају исте вреднсоти и за *n*, уколико је $a = x$. У том случају, прва функција враћа $g = EuklidA(b, a')$ где је $b = n$, а $a' = a\%b$. Друга функција враћа вредност $z = EuklidB(y, z\%y)$ где је $z = x$ и $y = n$. Дакле, из $a = x$, индуктивне хипотезе, $g = EuklidA(b, a')$, $b = n$, $a' = a\%b$, $z = EuklidB(y, z\%y)$, $z = x$ и $y = n$, потребно је доказати $g = z$. Важи $b = y$ и $a' = z\%y$. Пошто је a' остатак при дељењу са *n*, сигурно важи $a' < n$, па на основу индуктивне хипотезе, важи $EuklidA(b, a') = EuklidB(y, z\%y)$, па одатле и $g = z$, што је и требало доказати.

Наведени индуктивни доказ може се симулирати доказом који не користи индукцију већ неинтерпретиране функције. Индуктивна претпоставка у наведеном доказу може бити опонашана тако што ће рекурзивни позиви у обе функције да буду замењени позивом треће функције, рецимо *u*. Уколико су вредности аргумената почетне две функције једнаке, онда ће и вредности функције u свакако бити једнаке, при чему није битно која је то тачно вредност. Зато ова функција *u* може бити моделована неинтерпретираном функцијом. Аутоматски доказ еквивалентности заснивање се за индуктивни корак на описаној модификацији, при чему ће у услов еквивалентности, поред услова који одговара индуктивном кораку, ући и опис базних случајева који покривају индуктивну основу.

Према томе, да би се доказало да су функције са слике 6 еквивалентне, довољно је доказати да су еквивалентне верзије у којима су рекурзивни позиви замењени позивом истоимене функције `u`, на следећи начин:

- рекурзивни позив

  `g = EuklidA(b, a);`

  у функцији `EuklidA` замењује се са

  `g = u(b, a)`

- рекурзивни позив

  `z = EuklidB(y, z%y);`

  у функцији `EuklidB` замењује се са

  `z = u(y, z%y);`

при чему новоуведена функција `u` треба да се третира као неинтерпретирана функција. Еквивалентност ових функција LAV потврђује за 0.008 секунди коришћењем доказивача Z3 и теорије линеарне аритметике. Нагласимо да се остатак при дељењу у оквиру лиенарне аритметике третира такође као неинтерпретирана функција.

Претходни приступ се не може увек применити. Наиме, он претпоставља да у рекурзивним функцијама које се испитују нема утицаја и промена глобалних променљивих (јер се претпоставља да помоћна функција увек има исту вредност за исте вредности аргумената). Постоје и друга ограничења. На слици 7 приказане су четири сличне рекурзивне имплементације функције факторијел. За прве две се може доказати еквивалентност коришћењем описаног метода. Међутим, уместо линеарне аритметике у којој се множење мора моделовати неинтерпретираном функцијом, потребно је користити теорију која располаже знањем да је множење комутативна операција, на пример, теорија аритметике бит-вектора (LAV доказује ову еквивалентност за 0.06 секунди). Еквивалентност са трећом функцијом не може се доказати на овај начин. Наиме, услови изласка из рекурзије (тј. базни случајеви) су различити и не може се утврдити еквивалентност функција за `n=1`: на основу имплементација прве две функције, познато је да је за `n=1` вредност ових функција 1, док је за трећу функцију та вредност `1*u(0)`. Такође, еквивалентност се не може утврдити ни са четвртом имплементацијом: иако су услови изласка из рекурзије исти, рекурзивни позив има другачији аргумент од прве три функције. Због ових ограничења није могуће применити ни форму доказа која је приказана у првом примеру.

```
unsigned
faktorijelA(unsigned n)
{
if(n<=1) return 1;
return faktorijelA(n-1)*n;
}
```

```
unsigned
faktorijelB(unsigned n)
{
if(n<=1) return 1;
else return
n*faktorijelB(n-1);
}
```

```
unsigned
faktorijelC(unsigned n)
{
if(n<=0) return 1;
else return
n*faktorijelC(n-1);
}
```

```
unsigned
faktorijelD(unsigned n)
{
if(n<=1) return 1;
else return n*(n-1)
*faktorijelD(n-2);
}
```

Слика 7: *Имплементације алгоритма за одређивање факторијела природног броја.*

На основу наведених примера показали смо да описани метод не може да се примени на све парове еквивалентних рекурзивних функција. Међутим, у пракси веома често може, због природе уобичајних модификација током развоја програма (имплементација рекурзивне функције може да се мења, али се често не мења њен базни случај и сам рекурзивни позив). Из сличних разлога, описани метод може да буде применљив у многим случајевима у аутоматској евалуацији студентских програма. За практичне примене у програмима са петљама остаје изазов креирања општег механизма за трансформирање петље у рекурзивне функције или модификације метода за непосредни рад са петљама, што превазилази обим овог чланка.

6 ЗАКЉУЧЦИ И ДАЉИ РАД

Верификација је последњих година постигла огроман напредак. Алати и методи верификације софтвера су били ограничени на академску заједницу, али то полако почиње да се мења. Очекујемо да у наредном периоду верификацијски алати имају већу улогу у свим аспектима и сферама у развоју софтвера.

Регресиона верификација се, иако је помињана одавно, може сматрати једним од најновијих верификацијских приступа јер су значајни помаци на овом пољу начињени тек недавно. Регресиона верификација се у развоју софтвера може користити по аналогији са регресионим тестирањем али, наравно, са много већим степеном поузданости.

У овом раду показано је да се регресиона верификација може веома успешно применити у аутоматској евалуацију

студентских програма, што може да буде изузетно корисно и за наставнике и за студенте. Поред тога, показано је да овај верификацијски приступ може успешно да спроведе систем LAV. Са подршком за различите теорије које описују понашање програма и пратеће решаваче, систем LAV нуди висок степен флексибилности.

За даљи рад планирамо развој нових метода за регресиону верификацију функција са петљама и рекурзивним позивима. Поред тога, као конкретну примену, планирамо да интегришемо регресиону верификацију са другим нашим техникама за евалуацију студентских програма.

ЛИТЕРАТУРА

[1] I. E. Allen J. Seaman. Learning on demand: Online education in the United States, 2009. Technical report, The Sloan Consortium, 2010.

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, February 2010.

[3] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[4] A. Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation*., volume 2566 of LNCS, pages 85–108. Springer-Verlag, October 2002.

[6] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2009.

[7] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *Computer-Aided Verification*, volume 5123 of *Lecture Notes in Computer Science (LNCS)*, pages 299–303. Springer, 2008.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, pages 209–224. USENIX Association Berkeley, 2008.

[9] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176. Springer, 2004.

[10] E. M. Clarke. *25 Years of Model Checking — The Birth of Model Checking*. Lecture Notes in Computer Science (LNCS). Springer, 2008.

[11] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *International Conference on Automated Software Engineering (ASE)*, pages 137–148, 2009.

[12] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.

[13] L. De Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[14] A. Deutsch. Static Verification of Dynamic Properties, 2003. White paper, PolySpace Technologies Inc.

[15] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 144–157, New York, NY, USA, 2006. ACM.

[16] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[17] P. Emanuelsson and U. Nilsson. A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, July 2008.

[18] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer-Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Softw. Test., Verif. Reliab.*, 23(3):241–258, 2013.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of th ACM*, 12(10):576–580, October 1969.

[21] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[22] J. Laski and W. Stanley. *Software Verification and Analysis*. Springer-Verlag, London, 2009.

[23] C. Lattner. The LLVM Compiler Infrastructure, 2012. on-line at: http://llvm.org/.

[24] C. Lattner and V. Adve. The LLVM Instruction Set and Compilation Strategy. Technical Report UIUCDCS-R-2002-2292, CS Department, University of Illinois at Urbana-Champaign, Aug 2002.

[25] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, Lecture Notes in Computer Science (LNCS), pages 146–161. Springer, 2012.

[26] Ofer Strichman and Benny Godlin. Regression verification - a practical way to verify programs. In Bertrand Meyer and Jim Woodcock, editors, Verified Software: Theories, Tools and Experiments *(VSTTE)*, volume 4171 of *Lecture Notes in Computer Science*, pages 496–501. Springer, 2005.

[27] G. Tassey. The Economic Impacts of Inadequate Infrastructure For Software Testing. Technical report, National Institute of Standards and Technology, 2002.

[28] N. Tillmann and J. Halleux. Pex – White Box Test Generation for .NET. In *Proceedings of TAP 2008, the 2nd International Conference on Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science (LNCS)*, pages 134–153. Springer, 2008.

[29] M. Vujošević Janičić. LAV page, 2009–2014. on-line at: http://argo.matf.bg.ac.rs/?content=lav.

[30] M. Vujošević Janičić. Automatsko generisanje i proveravanje uslova ispravnosti programa, Decembar 2013. Doktorska disertacija, Matematički fakultet, Univerzitet u Beogradu.

[31] M. Vujošević Janičić and V. Kuncak. Development and Evaluation of LAV: An SMT-Based Error Finding Platform. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, Lecture Notes in Computer Science (LNCS), pages 98–113. Springer, 2012.

[32] M. Vujošević Janičić, M. Nikolić, D. Tošić, and V. Kuncak. Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology*, 55(6):1004 – 1016, 2013.

[33] M. Vujošević Janičić and D. Tošić. The Role of Programming Paradigms in the First Programming Courses. *The Teaching of Mathematics,* XI(2):63–83, 2008.

dr Milena Vujošević Janičić
Matematički fakultet, Univerzitet u Beogradu
Kontakt: milena@matf.bg.ac.rs
Oblasti interesovanja: analiza i verifikacija softvera, automatsko otkrivanje grešaka u softveru, semantika programskih jezika