УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Милена М. Вујошевић Јаничић

АУТОМАТСКО ГЕНЕРИСАЊЕ И ПРОВЕРАВАЊЕ УСЛОВА ИСПРАВНОСТИ ПРОГРАМА

докторска дисертација

Београд, 2013.

UNIVERSITY OF BELGRADE
FACULTY OF MATHEMATICS

Milena M. Vujošević Janičić

AUTOMATED GENERATION AND CHECKING OF VERIFICATION CONDITIONS

Doctoral Dissertation

Belgrade, 2013.

Ментор:

др Душан Тошић, редован професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Виктор Кунчак, ванредни професор
EPFL, Лозана, Швајцарска

др Филип Марић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: _____

Мами

Наслов дисертације: Аутоматско генерисање и проверавање услова исправности програма

Резиме: LAV је систем за аутоматско генерисање и проверавање услова исправности програма. Систем је намењен првенствено анализи програма написаних у програмском језику C, али, пошто користи LLVM међујезик, може се применити и за анализу програма написаних у другим процедуралним програмским језицима.

Систем комбинује симболичко извршавање, описивање понашања програма исказним променљивама 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 праве се формуле које описују понашање програма. Те формуле користе се за формирање услова исправности појединачних команди 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. Команда може имати статус безбедне (извршавање команде не доводи до грешке), неисправне (извршавање команде сигурно доводи до грешке), небезбедне (извршавање команде може да доведе до грешке) и недостижне (до извршавања команде никада неће доћи).

Предложени систем је имплементиран у програмском језику C++ као алат LAV који је јавно доступан и отвореног кода. У оквиру алата постоји подршка за рад са неколико SMT решавача (Boolector, MathSAT, Yices и Z3). Експериментални резултати на корпусу C програма, који служе за утврђивање могућности верификацијских алата, показују да је представљен приступ упоредив са постојећим сродним алатима. Такође, експериментални резултати показују предност предложеног система у односу на симболичко извршавање у програмима у којима постоји велики број могућих путања. Компактно моделовање могућих путања кроз програм омогућава утврђивање исправности команди у програмима који су ван домашаја алата за симболичко извршавање.

LAV је успешно примењен и у контексту образовања где је коришћен за откривање грешака у студентским радовима. Овом применом показано је да у студентским радовима уводног курса програмирања постоји велики број грешака које верификацијски алат може ефикасно да пронађе. Експериментални резултати показали су да је верификација ефикаснија од тренутно доминантно коришћених техника аутоматског тестирања јер може да открије грешке у студентским програмима које су ван домашаја ових техника. Такође, показано је како LAV може да унапреди аутоматску евалуацију студентских радова на пољу генерисања квалитетних, разумљивих и поузданих података о резултатима рада студента, као и на пољу аутоматског оцењивања.

Кључне речи: верификација софтвера, аутоматско проналажење грешака у програму, симболичко израчунавање, проверавање ограничених модела

Научна област: рачунарство

Ужа научна област: верификација софтвера

УДК број: 004.415.5(043.3)

Dissertation title: Automated Generation and Checking of Verification Conditions

Abstract: LAV is a system for statically verifying program assertions and locating bugs such as buffer overflows, pointer errors and division by zero. LAV is primarily aimed at analyzing programs written in the programming language C. Since LAV uses the popular LLVM intermediate code representation, it can also analyze programs written in other procedural languages. Also, the proposed approach can be used with any other similar intermediate low level code representation.

System combines symbolic execution, SAT encoding of program's control-flow, and elements of bounded model checking. LAV represents the program meaning using first-order logic (FOL) formulas and generates final verification conditions as FOL formulas. Each block of the code (blocks have no internal branchings and no loops) is represented by a FOL formula obtained through symbolic execution. Symbolic execution, however, is not performed between different blocks. Instead, relationships between blocks are modeled by propositional variables encoding transitions between blocks. LAV constructs formulas that encode block semantics once for each block. Then, it combines these formulas with propositional formulas encoding the transitions between the blocks. The resulting compound FOL formulas describe correctness and incorrectness conditions of individual instructions. These formulas are checked by an SMT solver which covers suitable combination of theories. Theories that can be used for modeling correctness conditions are: theory of linear arithmetic, theory of bit-vectors, theory of uninterpreted functions, and theory of arrays. Based on the results obtained from the solver, the analyzed command may be given the status safe (the command does not lead to an error), flawed (the command always leads to an error), unsafe (the command may lead to an error) or unreachable (the command will never be executed). If a command cannot be proved to be safe, LAV translates a potential counterexample from the solver into a program trace that exhibits this error. It also extracts the values of relevant program variables along this trace.

The proposed system is implemented in the programming language C++, as a publicly available and open source tool named LAV. LAV has support for several SMT solvers (Boolector, MathSAT, Yices, and Z3). Experimental evaluation on a corpus of C programs, which are designed to demonstrate areas of strengths and weaknesses of different verification techniques, suggests that LAV is competitive with related tools. Also, experimental results show a big advantage of the proposed system compared to symbolic execution applied to programs containing a big number of possible execution paths. The proposed approach allows determining status of commands in programs which are beyond the scope of analysis that can be done by symbolic execution tools.

LAV is successfully applied in educational context where it was used for finding bugs in programs written by students at introductory programming course. This application showed that in these programs there is a large number of bugs that a verification tool can efficiently find. Experimental results on a corpus of students' programs showed that LAV can find bugs that cannot be found by commonly used automated testing techniques. Also, it is shown that LAV can improve evaluation of students's assignments: (i) by providing useful and helpful feedback to students, which is important in the learning process, and (ii) by improving automated grading process, which is especially important to teachers.

Keywords: software verification, automated bug finding, symbolic execution, bounded model checking

Research area: Computer Science

Research sub-area: Software Verification

UDC number: 004.415.5(043.3)

Предговор

Присутан свуда око нас, софтвер представља важан саставни део свакодневног живота. Софтвер нам пружа могућност удобног руковања различитим уређајима, олакшава нам и поједностављује различите животне ситуације. Међутим, понекад осетимо фрустрацију зато што наш мобилни телефон не функционише како треба, или чујемо катастрофалну вест о паду авиона. Ове ситуације често су последице грешака у софтверу.

Неисправан софтвер кошта светску економију милијарде долара годишње. Међутим, суштинска важност исправности софтвера не мери се новцем јер неисправан софтвер може да узрокује материјално несагледиве последице. То чини питање испитивања исправности софтвера, односно верификације софтвера, посебно важним и мотивисало ме је за рад и истраживање у овој области. У томе ми је изузетно помогао ментор, професор Душан Тошић, коме овом приликом желим посебно да захвалим. Поред драгоценог научног усмеравања, многих коментара и савета захваљујући којима је теза попримила постојећи облик, неизмерно сам му захвална и на подршци и разумевању које ми је указао у току рада на тези. Такође, захвална сам му и на свему што сам од њега научила током многих година: у гимназији, на основним студијама, на последипломским студијама, током рада на магистратури и током рада на овој тези. Велику захвалност дугујем и професору Виктору Кунчаку који је активно учествовао у истраживањима у оквиру тезе и дао многе кључне идеје, савете и увиде засноване на најновијим научним резултатима. Веома сам захвална и доценту Филипу Марићу на многим корисним саветима и дискусијама, пажљивом читању прелиминарних верзија радова и детаљним коментарима на тезу. Захвална сам и колеги Младену Николићу на сарадњи у истраживањима на тему аутоматске евалуације студентских радова.

Својој породици дугујем захвалност на љубави, разумевању и подршци која ми је пружена у току рада на тези. Највећу захвалност, свакако, дугујем својој мајци, којој и посвећујем саму тезу.

Милена Вујошевић Јаничић

Београд, август 2013.

Садржај

1

Увод

Паралелно са развојом нових технологија, а у циљу пружања нових квалитета и удобности у раду, програми који пружају корисницима различите услуге, као и програми који контролишу уређаје присутне свуда око нас, постају све сложенији. Сложеност програма неминовно повлачи и већи простор за прављење грешака. Ове грешке, у зависности од намене самог програма, могу да буду непријатне по појединца, али могу да имају и фаталне последице по животе и здравље људи. На пример, грешке у софтверу мобилних телефона, Интернет прегледача и музичких уређаја немају исту тежину као грешке у софтверу аутомобила, банака, авиона, свемирских летилица, апарата у здравству и нуклеарних електрана. Због тога се у процесу развоја софтвера посебна пажња поклања *верификацији софтвера*, односно испитивању исправности софтвера.

Исправност софтвера може се испитивати приликом његовог извршавања. За то се користе технике динамичке анализе, најчешће технике тестирања. С друге стране, исправност софтвера је могуће проверавати и без његовог извршавања, само на основу анализе изворног кода, коришћењем техника статичке анализе. Анализа се може извршити прегледом кода од стране програмера, може бити полу-аутоматизована (програмер уз помоћ специјализованих алата проверава исправност софтвера) или може бити потпуно аутоматизована. Сваки од претходних приступа има своје предности, али и своје мане. Динамичком анализом може се утврдити исправност софтвера само за оне параметре за које је програм покренут. Прегледање кода најчешће није довољно поуздан начин откривања грешака. Полу-аутоматски приступи могу да буду веома временски захтевни и исцрпљујући по програмера.

За аутоматске приступе постоји важно теоријско ограничење: неодлучивост. То значи да није могуће конструисати алгоритам који у коначном времену аутоматски утврђује исправност прозвољног програма потпуно прецизно. Поред

овог фундаменталног ограничења, постоје и практична ограничења која се односе на постојање коначних ресурса: коначне меморије и ограниченог времена за анализу исправности. То значи да, чак и за програме који имају коначан простор могућих стања и за које је проблем испитивања исправности теоријски одлучив, овај простор може да буде превелик и да, услед ограничености ресурса, теоријско решење проблема није практично употребљиво.

Мотивација и циљ тезе

Због важности проблема верификације, а услед поменутих теоријских и практичних ограничења, развијају се разнородне технике и приступи засновани на статичком аутоматском испитавању исправности софтвера. Ове технике разликују се по ефикасности и прецизности, а њихова практична примена зависи од програмског језика и врсте грешака које имају за циљ да открију.

Основне технике статичке анализе су проверавање модела, апстрактна интерпретација и симболичко извршавање (о овим техникама биће више речи у одељку 2.1.2). Техника симболичког извршавања одавно је позната, а последњих година постаје све популарнија. Симболичким извшавањем 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. Ова анализа веома је прецизна, а њоме се остварује велика предност у односу на технике динамичког тестирања, где се једним тест примером проверава само исправност програма за фиксиране вредности. Основни проблем овог приступа јавља се у програмима у којима постоји огроман број путања при извршавању, као што је то илустровано у одељку 4.2.1. С друге стране, техника ограниченог проверавања модела је такође веома прецизна, а компактније моделује могуће путање кроз програм. Међутим, овом техником могу да се анализирају само они програми у којима постоји ограничен број могућих стања, што се првенствено односи на постојање горњих ограничења за број пролазака кроз петље у програму. То значи да је на овај начин могуће верификовати програме у којима постоји горње ограничење за петље у програму. Уколико такво ограничење не постоји, програм је могуће анализирати само парцијално. Парцијална анализа оставља простор за неоткривене грешке. Анализа програма, који садржи петље за које не постоји горње ограничење, могућа је техникама апстрактне интерпретације, али са значајно смањеном прецизношћу.

Различити програмски језици омогућавају прављење различитих врста грешака у програмима. Због тога, анализа исправности значајно зависи од језика

на којем је софтвер написан и постоји велики број алата који су специјализовани за проналажење грешака у програмима одабраног програмског језика. Међутим, постоје и грешке које су заједничке за више програмских језика, као и грешке које су заједничке за језике који припадају истој програмској парадигми. Постојање засебних алата за проналажење ових заједничких грешака последица је тога што је сваки програм потребно најпре трансформисати у облик погодан за анализу, а ова трансформација зависи од синтаксе и могућности програмског језика. Превазилажење проблема трансформације кода, развојем система који анализира кôд међујезика ниског нивоа, омогућило би примену верификацијског система на разнородне програмске језике.

Основни циљ ове тезе је *развој хибридног система за верификацију и аутоматско откривање грешака у софтверу процедуралних програмских језика, који је у стању да превазиђе неке од слабости појединачних метода које комбинује.*

Доприноси тезе

У оквиру рада на тези, осмишљен је нови верификацијски систем, реализована је његова имплементација, експериментално је потврђена ефикасност предложеног система, а систем је и успешно примењен за проналажење грешака у студентским програмима. У наставку текста набројани су основни доприноси тезе.

- Развијен је систем LAV који на нов начин комбинује постојеће технике верификације и који уводи новине у моделовање контроле тока програма (глава 3). Систем се заснива на анализи кода ниског нивоа па га је могуће применити на различите програмске језике.

- Развијена је имплементација предложеног система (глава 4). Експериментални резултати поређења имплементације система и сродних алата показују да је предложени систем упоредив по ефикасности са постојећим сродним техникама, као и да је примењив на класу програма који су ван домашаја техника симболичког извршавања. Развијен систем, имплементација и експериментална евалуација представљени су на конфернецији *Verified Software: Theories, Tools, and Experiments*[1] [168].

[1] Ову конференцију је засновао Тони Хор (енг. Tony Hoare) 2005. године у оквиру пројекта *Grand Challenge*. Овај пројекат има за циљ обједињавање напора истраживача из целог света како би у догледно време алати за верификацију софтвера били широко и практично употребљиви.

- Имплементација предложеног система, алат LAV, отвореног је кода и јавно је доступна са адресе: `http://argo.matf.bg.ac.rs/?content=lav`.
- Алат LAV успешно је примењен у домену образовања (глава 5). Овом применом показано је да у студентским радовима постоји велики број грешака које верификацијски алат може ефикасно да пронађе, да верификацијски алати могу да допринесу проналажењу грешака које су ван домашаја техника које се тренутно доминантно користе у овом контексту, као и да верификацијски алат може значајно да унапреди аутоматску евалуацију студентских радова. Ови резултати објављени су у часопису *Information and Software Technologies* [170].

Организација тезе

У оквиру тезе приказани су основни појмови који су потребни за разумевање приступа верификацији (глава 2), описан је предложен систем (глава 3), његова имплементација и експериментална евалуација (глава 4), једна практична примена (глава 5), закључци и могући правци даљег рада (глава 6), као и употреба самог система (додатак A). У наставку текста дат је прецизнији опис организације тезе.

Основе (глава 2) — У оквиру ове главе приказани су најважнији појмови и технике које се користе у остатку тезе. Статичке технике верификације софтвера (одељак 2.1) могу да користе различите теорије за моделовање понашања програма (одељак 2.2). Анализа програма обично се врши над кодом који је претходно трансформисан (одељак 2.3).

Опис система LAV (глава 3) — У оквиру ове главе дат је детаљан опис предложеног система LAV. Систем LAV комуницира са окружењем (одељак 3.1), комбинује различите технике моделовања основних градивних јединица програма (одељак 3.2) и контроле тока програма (одељак 3.3). Да би проверио да ли нека наредба може да доведе до грешке у програму, систем гради формуле које представљају услове исправности наредби (одељак 3.4) и које припадају изабраној комбинацији предложених теорија (одељак 3.5). Услови исправности се проверавају одговарајућим решавачем на основу чијих резултата се генерише извештај о статусу наредби у програму (одељак 3.6). Апроксимације које се уводе приликом прављења модела програма или приликом трансформације модела програма у формулу на језику изабране теорије, доводе до могућих неоткривених грешака и/или лажних упозорења (одељак 3.7).

Имплементација и евалуација система LAV (глава 4) — У оквиру ове главе детаљно је приказана имплементација и евалуација предложеног система. Имплементација алата (одељак 4.1) прати општи опис система. Резултати експерименталног поређења показују да је алат LAV упоредив са сродним системима, као и да за неке примере даје значајно боље резултате (одељак 4.2).

Примена система LAV (глава 5) — У оквиру ове глава описана је примена система LAV у аутоматској евалуацији студентских радова. Постоје многе користи од аутоматске евалуације студентских програма (одељак 5.1). Већина постојећих алата за аутоматску евалуацију студентских програма се заснива на аутоматском тестирању (одељак 5.2). Верификацијски алати могу да пронађу мноштво грешака у студентским програмима (одељак 5.3). Увођење софтверске верификације у аутоматску евалуацију студентских програма, поставља нове изазове пред верификацијске алате (одељак 5.4), али такође значајно доприноси квалитету генерисаних података о резултатима рада студента (одељак 5.5). Резултати верификације могу се користити за прецизно аутоматско оцењивање студентских радова уводних курсева програмирања (одељак 5.6). Постоје различити сродни алати за аутоматску евалуацију студентских радова (одељак 5.7).

Закључци и даљи рад (глава 6) — У оквиру ове главе сумирају се резултати тезе и дискутује се о могућим правцима даљих истраживања.

Употреба система LAV — (додатак A) У оквиру ове главе описана је употреба система LAV, тј. инсталација система (одељак A.1), подешавање улазних параметара који одређују начин моделовања програма, прецизност анализе и избор и начин коришћења решавача (одељак A.2), као и извештај који систем LAV генерише (одељак A.3).

2

Основе

У овој глави укратко су описани појмови и технике који се користе у остатку тезе: општи приступи верификацији софтвера, најчешће коришћене теорије за моделовање понашања и услова исправности програма, решавачи који се за ове теорије користе и међујезици који се користе за трансформацију и анализу програма.

2.1 Верификација софтвера

Испитивање исправности софтвера један је од кључних проблема у развоју софтвера. Грешке у софтверу коштају светску економију милијарде долара годишње [158]. 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.

Да би се испитивала исправност софтвера, најпре је потребно прецизно дефинисати његово жељено понашање [99]. Жељено понашање софтвера задаје се *спецификацијом*. Спецификација даје опис шта софтвер треба да ради. На пример, спецификација функције најчешће садржи аргументе и повратну вредност функције, спољашње променљиве којима функција приступа, *предуслов* и *постуслов* функције. Предуслов функције (енг. precondition) чини скуп захтева које улазне вредности функције треба да задовољавају. Улаз је *исправан* (или *валидан*) уколико задовољава задате предуслове. Постуслов функције (енг. postcondition) представља жељену везу између исправних улазних вредности и излазних вредности. Уколико улазне вредности функције нису исправне, онда то резултује *недефинисаним понашањем* (енг. undefined behaviour) функције. Недефинисано понашање функције, у зависности од програмског језика,

може да доведе до генерисања изузетка, до краха програма или до израчунавања бесмислених резултата.

Верификација је поступак доказивања да је програм исправан, односно да задовољава спецификацију [99]. Спецификација није испуњена уколико постоје грешке у програму које нарушавају *функционалне* и *нефункционалне* жељене особине софтвера [7]. Функционалне особине софтвера дефинишу жељене излазе за задате улазе, док нефункционалне особине софтвера дефинишу време одзива, перформансе и ефикасност. Посебну класу грешака које могу да наруше функционалне особине софтвера чине грешке које су независне од намене програма, а које се односе на испуњавање *безбедносних захтева* (енг. safety requirements), тј. да програм не сме да крахира, да се блокира, успорава рад током времена и слично [7]. Грешке које могу да наруше безбедносне захтеве су, на пример, покушај дељења нулом, читање садржаја ван граница резервисане меморије, покушај дереференцирања NULL показивача и присуство кружних блокада (енг. dead-lock). Неке грешке које нарушавају безбедносне захтеве су специфичне за програмски језик у којем је софтвер имплементиран. Тако, на пример, за програмске језике C и C++ постоје специфичне грешаке које су веома важне због високе поплуларности ових програмских језика [143, 77, 30]. То су грешке цурења меморије, које могу значајно да утичу на перформансе програма који се извршава, као и грешке дереференцирања неиницијализованог показивача или показивача на већ ослобођену меморију, које могу да доведу до краха програма, али и до неочекиваног и неисправног тока извршавања програма (уписивањем или читањем садржаја из непредвидивог дела меморије). Писање ван граница резервисане меморије у овим језицима посебно је честа и опасна грешка и назива се *прекорачење* или *препуњење бафера* (енг. buffer overflow, buffer overrun) [138, 166, 167]. Под бафером се, у овом контексту, подразумева блок меморије резервисан статички или динамички за привремено складиштење података.[1] Прекорачење бафера може да има за последицу неочекивано и неисправно понашање програма, може да доведе до краха програма, али додатно је посебно опасно зато што постоји велики број начина за сигурносне нападе и злоупотребе програма са овом грешком [164, 4, 95, 166, 167].

Два основна приступа верификацији су *динамички* и *статички* [99], и они ће бити детаљније описани у наставку текста. За утврђивање исправности софтвера и за проналажење (неких од) претходно описаних грешака у софтверу постоје разни алати који се заснивају на овим приступима.

[1] Термин *прихватник* веома је добар превод за енглеску реч *buffer*, али како још није зажив‌ео у нашем језику, у наставку тезе углавном ће се користити опште прихваћен англицизам *бафер*.

2.1.1 Динамичка верификација

Динамичка верификација софтвера односи се на проверу исправности софтвера у фази извршавања, најчешће путем тестирања. Тестирањем се не може доказати одсуство грешака у програму, али се може показати њихово присуство. Због тога, циљ тестирања заправо није доказ исправности већ проналажење грешака у програму [122, 23].

Простор могућих улаза програма обично је превелик па није могуће покретање и провера програма са свим могућим улазима. Тестирање се састоји од покретања програма са репрезентативним скупом улазних података и поређење добијених резултата са очекиваним резултатима. Метод одређивања репрезентативног скупа података, над којим ће се вршити тестирање, назива се *стратегија тестирања* (енг. testing strategy) [99]. Стратегија тестирања треба да води избору скупа података који: (1) има висок потенцијал откривања грешака, (2) реалтивно је мале величине, (3) води до високог поверења у поузданост софтвера након што софтвер успешно прође све тест примере. Постоје два основна извора тестова: спецификација програма и кôд програма.

Стратегије генерисања тестова на основу спецификације, односно без разматрања интерне структуре кода, називају се стратегијама тестирања *црне кутије* (енг. black-box testing) [24]. У оквиру ових стратегија, издваја се тестирање функционалних и нефункционалних особина софтвера, као и *регресионо тестирање* [96, 55] — тестирање да ли софтвер задржава своје понашање и након извесних измена.

Стратегије генерисања тестова на основу кода програма називају се стратегијама структурног тестирања или тестирања *беле кутије* (енг. structural testing, white-box testing) [122, 136, 92, 55]. Најчешћи тип структурног тестирања су *јединични тестови* (енг. unit tests) којима се проверава исправност кода приликом писања сваке основне градивне јединице програма (на пример, функције или класе). Структурно тестирање користи се и за проналажење безбедносних пропуста у софтверу [74]. Основне стратегије структурног тестирања заснивају се на критеријуму *покривености кода* (енг. code coverage). Покривеност кода може се рачунати према броју извршених путања кроз програм (енг. path coverage), према броју извршених инструкција (енг. instruction coverage) или према броју извршених грана након сваке инструкције гранања (енг. branch coverage) или комбинацијом претходних метода [99, 128].

Мешовите стратегије генерисања тестова, које укључују и коришћење спецификације и увид у кôд програма, називају се стратегије тестирања *сиве кутије* (енг. gray-box testing) [136, 55].

Аутоматизација процеса генерисања тест примера и провере резултата тестирања посебно је важна јер олакшава и убрзава процес тестирања [136, 75]. Пример технике коју је могуће потпуно аутоматизовати је *расплинуто тестирање* (енг. fuzz testing) [157, 155]. Овом техником генеришу се неисправни, неочекивани и случајни улази за које се затим прати ток извршавања програма са циљем детектовања неочекиваних крахова, подизања изузетака, цурења меморије и других безбедносних слабости. Тест пример са којим се започиње тестирање може бити случајно генерисан или улаз обезбеђен од стране програмера. Сваки наредни тест пример генерише се на основу претходно генерисаних тест примера, праћења тока извршавања програма и утицаја промене улазних параметара на извршавање нових, неистражених путања кроз програм. Расплинуто тестирање је широко распрострањено [59]. Може се користити и као стратегија структурног тестирања [74, 73, 150] и као стратегија тестирања црне кутије [118].

2.1.2 Статичка верификација

Статичка верификација је испитивање исправности програма без његовог извршавања, анализом кода. Анализа кода може бити ручна, и односи се на људске провере и прегледе кода, или може бити аутоматизована. У оквиру аутоматизоване провере исправности издвајају се формалне методе верификације у којима се услови исправности софтвера исказују у терминима математичких тврђења на стриктно дефинисаном формалном језику изабране математичке теорије [7].

Због неодлучивости халтинг проблема (енг. halting problem), не постоји општи аутоматизован начин за проверавање да ли је нека наредба програма достижна [162], па тиме ни да ли је исправна, односно да ли је сâм 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* (енг. false positives) или *пропуштених грешака* (енг. false negatives). Лажно упозорење је извештај да програм има грешку која заправо не постоји. Пропуштена грешка је грешка која није откривена приликом испитивања исправности програма.

У складу са неодлучивошћу проблема испитивања исправности програма, особине полазног система могу се верно описати адекватним неодлучивим теоријама, али онда, због неодлучивости изабраних теорија, процес доказивања није могуће потпуно аутоматизовати. Зато се за доказивање комплексних особина софтвера дефинисаних у изражајним неодлучивим логикама користе *ин-*

терактивни доказивачи теорема (енг. interactive theorem provers) [175]. Ови системи служе за поуздано интерактивно доказивање исправности програма. Провером доказа обезбеђује се највиши степен поузданости и сигурности у доказ исправности, па тиме и у кôд који се верификује, јер се провером доказа искључује могућност постојања грешке у самом доказу исправности. Такви системи су, на пример, системи Isabelle [132] и Coq [25].

С друге стране, описивање полазног система одлучивим теоријама омогућава потпуну аутоматизацију процеса верификације, али само под претпоставком неограниченог времена, што у пракси често није случај. Поред тога, у општем случају, сваки коначан опис у оквиру одлучиве теорије нужно је само апроксимација понашања полазног система, што доводи до [47]:

- непрецизне анализе, односно увођења разних апроксимација због којих опис полазног система није веран, па направљене апроксимације могу да доведу до лажних упозорења или да неке грешке не буду уочене;
- анализе коју, за неке програме, није могуће завршити услед недостатка потребних ресурса или није могуће завршити у коначном времену.

Ради добијања прецизнијих резултата у коначном времену и са коначним ресурсима, квалитет анализе се може побољшати додавањем *обележја* (енг. annotations) [22]. У програм који се анализира, обележјима се додају информације које је тешко или немогуће аутоматски извести. Обележја се додају и за информације које се могу аутоматски извести уколико се додавањем обележја може уштедети време потребно за анализу. Обележјима се најчешће задају инваријанте петљи и предуслови и постуслови функција, јер су анализа петљи, анализа ефекта позива функције за коју није доступан изворни кôд и анализа рекурзивних позива функција најчешћи узроци прављења апроксимација приликом анализе. Обележја која се односе на ефекат позива функције називају се и *уговори* функција (енг. contracts).

У потпуно аутоматизоване приступе верификацији програма спадају методе *проверавања модела*, *апстрактне интерпретације* и *симболичко извршавање*.

Проверавање модела

Проверавање модела (енг. model checking) метод је верификације у којем се систем (хардверски или софтверски), који је потребно верификовати, описује коначним аутоматом (енг. finite state machine), а спецификација се задаје у терминима темпоралне логике [44]. 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 [28]. 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 путањи кроз систем (чиме је модел верификован). Проверавање модела за расуђивање о испуњавању услова исправности обично користи дијаграме бинарних одлука (енг. binary decision diagrams) док проверавање ограничених модела обично користи SAT или SMT решаваче (који су описани у секцији 2.2.2) [28, 8]. У контексту софтвера, због великог простора стања који је потребно систематски претражити, проверавање (ограничених) модела се најчешће користи за проналажење грешака у систему, као допуна тестирању програма, а не као метод потпуне верификације програма. Софтверске грешке које се најчешће траже проверавањем модела су: присуство кружних блокада, присуство неадекватног рада са меморијом и показивачима, односно присуство стања која доводе до краха система.

Неки од верификацијских алата који се заснивају на проверавању ограничених модела су CBMC, ESBMC и LLBMC.

CBMC [43] је јавно доступан алат првенствено намењен анализи ANSI-C и C++ програма за уграђене системе (енг. embedded software). Овај алат врши провере прекорачења бафера, безбедности у раду са показивачима и изузецима, као и услова задатих од стране корисника. CBMC користи компилатор `goto-cc` [133] који преводи C или C++ програм у GOTO програм (тј. у граф контроле тока) над којима врши анализу. Анализа програма се врши размотавањем петљи у програму и прослеђивањем формула које одговарају условима исправности одговарајућој процедури одлучивања. Основна верзија алата користи исказну логику и SAT решавач MiniSAT2, али постоји и подршка за рад са SMT решавачима Boolector [36], MathSAT [38] и Z3 [54][2].

ESBMC [46] је јавно доступан алат намењен анализи ANSI-C и C++ програма за уграђене системе. Овај алат проверава присуство поткорачења

[2]SMT подршка је још увек у експерименталној фази.

и прекорачења у аритметичким изразима, безбедност у раду са показивачима, цурење меморије, прекорачење бафера и грешке специфичне за вишенитни софтвер (енг. multi-threaded software), као што су присуство кружних блокада и истовремени приступи истој меморији од стране више нѝти, при чему бар једна врши уписивање новог садржаја у меморију (енг. data race). ESBMC користи приступну компоненту (енг. front-end) алата CBMC, тако да врши анализу над GOTO програмима. ESBMC формулише услове исправности користећи теорију линеарне аритметике, теорију неинтерпретираних функција, теорију низова и теорију бит-вектора (о овим теоријама ће бити више речи у секцији 2.2.1), и за проверавање услова исправности користи SMT решаваче Z3 [54] и Boolector [36].

LLBMC [117] је алат за проверавање ограничених модела C и C++ програма. Овај алат врши провере грешака дељења нулом, прекорачења у аритметичким изразима, грешке прекорачења бафера, неисправног ослобађања меморије, двоструког ослобађања меморије, као и услова задатих од стране корисника. LLBMC врши анализу над LLVM кодом (о LLVM систему ће бити више речи у секцији 2.3.2). За проверавање услова исправности алат може да користи SMT решаваче STP [71] и Boolector [36]. Алат је јавно доступан само за академске сврхе.

Апстрактна интерпретација

Апстрактна интерпретација (енг. abstract interpretation) техника је апроксимације формалне семантике програма, тј. математичког модела могућих понашања програма [48]. Семантика програма може се описати конкретним доменом D_c и релацијама над овим доменом. Ове релације се могу мењати током извршавања наредби програма. За велике програме, испитивање да ли неко својство важи над доменом D_c отежано је због величине самог домена, због великог броја могућих путања кроз програм као и због неодлучивости која се јавља у разним контекстима. Један начин превазилажења ових потешкоћа је апроксимација конкретног домена D_c апстрактним доменом D_a, односно конкретан домен вредности замењује се апстрактним доменом описа ових вредности. На пример, бесконачни домен скупа целих бројева може се заменити апстрактим доменом који садржи вредности знакова бројева, тј. скупом $\{+, -, 0\}$. Иако апстрактни домен није тако прецизан као конкретан домен, он може у неким ситуацијама да дâ одговоре о важењу неких својства. На пример, за рачунање знака резултата множења, довољно је знати само знаке операнада. Апстраховањем се могу изгубити важне информације, тако, на пример, за рачунање

знака резултата сабирања, знаци операнада нису увек довољни. Бирање адекватне апроксимације домена ослања се на теорију мрежа (енг. lattice theory), на рачунање фиксних тачака парцијално уређених скупова и на Галоаове везе (енг. Galois connections).

Апстрактна интерпретација има низ могућих примена, али се најчешће користи за анализу програма у оквиру компилатора како би се одлучило да ли је могуће применити неке оптимизације или трансформације програма, као и са циљем проналажења извесних класа грешака у програму (нпр. дељење нулом или дереферцирање NULL показивача) [47]. Апстрактна интерпретација се може користити и као техника аутоматског одређивања инваријанти петљи у програму [102].

Технике апстрактне интерпретације могу се применити и на велике програме (реда величине и од неколико стотина хиљада линија кода) [47, 65]. Скалабилност ових техника последица је губљења прецизности приликом моделовања, што често резултује великим бројем лажних упозорења.

Неки од верификацијских алата који се заснивају на апстрактној интерпретацији су Astrée, Polyspace, Coverity SAVE и PAGAI.

Astrée [32] је комерцијални алат намењен првенствено анализи програма за софтвер у уграђеним системима чија је безбедност критична (енг. safety-critical embedded software). Алат анализира програме написане у подјезику програмског језика C који не садржи динамичку алокацију меморије и рекурзивне позиве функција. Овај алат проверава услове задате од стране корисника и грешке које могу да доведу до краха софтвера.

Polyspace [56] је комерцијални алат за статичку анализу програма написаних у програмским језицима C, C++ и Ada. Алат проналази грешке као што су аритметичка прекорачења, прекорачења бафера и дељење нулом. Алат такође користи метрике за оцењивање квалитета кода и проверава да ли је кôд писан по одговарајућим стандардима.

Coverity SAVE [26] је комерцијални алат за анализу C, C++, C# и Java програма. Поред апстрактне интерпретације, алат користи и друге технике статичке анализе како би анализа кода била што прецизнија. Овај алат карактерише изузетна скалабилност: може да анализира кôд од више милиона линија.

PAGAI [79] је јавно доступан алат отвореног кода који се дистрибуира под мањом GNU општом јавном лиценцом (енг. GNU Lesser General Public

License) [106]. PAGAI аутоматски генерише нумеричке инваријанте петљи LLVM кода коришћењем техника апстрактне интерпретације. Генерисане инваријанте алат уписује у виду коментара у анализирани кôд. Алат може да проверава и кориснички задате услове.

Симболичко извршавање

Симболичко извршавање се односи на анализу програма праћењем симболичких уместо конкретних вредности променљивих [93]. На овај начин конструишу се симболички изрази који описују одређене путање кроз програм и омогућава се истовремено расуђивање о свим улазима који прате исту путању кроз програм.

Уколико се симболичким извршавањем одређене путање у програму покаже да она задовољава услове исправности програма, онда се тај закључак може пренети на све могуће улазе који прате ту путању кроз програм. Тиме се значајно добија на ефикасности у односу на технике динамичке анализе програма где се анализа врши само за конкретне вредности улазних параметара. 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.

Време потребно за проналажење грешке у програму може значајно да зависи од путање у којој се дата грешка испољава [168]. На пример, уколико се грешка испољава у путањи програма која се симболичким извршавањем прва анализира, онда је време проналажења грешке минимално. Међутим, уколико се грешка испољава у путањи програма која се симболичким извршавањем последња анализира, тада, уколико постоји велики број могућих путања кроз програм, постоји опасност да се грешка у програму не пронађе (услед истека времена предвиђеног за анализу). Овај проблем ће бити детаљније анализиран у одељку 4.2.1.

Неки од верификацијских алата који се заснивају на симболичком извршавању су KLEE, PEX и Calysto.

KLEE [39] је јавно доступан алат који се дистрибуира под лиценцом NCSA отвореног кода Универзитета Илиноис (енг. The University of Illinois/NCSA Open Source License) [107], а служи за симболичко извршавање C програма и за аутоматско генерисање тест примера. Алат проверава грешке прекорачења бафера, грешке у раду са показивачима и грешке дељења нулом. KLEE врши анализу над LLVM кодом и користи SMT решавач STP [71]

за проверавање услова исправности које генерише. KLEE користи неколико оптимизација и хеуристика за побољшавање покривености кода приликом симболичког извршавања. Овај алат користи се и као саставни део система за верификацију, на пример у оквиру S2E платформе [42] за развијање нових алата за анализу програма.

PEX [160] је Мајкрософтов јавно доступан алат за аутоматско генерисање тест примера и проналажење грешака у програмима. PEX се може користити за све језике .NET платформе. За проверавање услова исправности PEX користи SMT решавач Z3 [54].

Calysto [11] је истраживачки прототип алата који ради над LLVM кодом. Овај алат проверава безбедност рада са показивачима и услове задате од стране корисника. Алат користи решавач за теорију бит-вектора Spear [12]. Алат није јавно доступан.

2.2 Моделовање понашања и услова исправности програма

У оквиру тезе, подразумеваће се познавање основних логичких појмова [116, 97, 114].

Да би се процес верификације потпуно аутоматизовао, потребно је задати опис услова исправности програма у оквиру неке одлучиве теорије за коју постоје ефикасне процедуре одлучивања. У наставку текста, термин моделовање програма обухватаће и моделовање понашања програма и моделовање услова исправности програма.

2.2.1 Теорије за моделовање програма

У овој секцији наведене су неке од теорија које се често користе за моделовање понашања и услова исправности програма [21]. Ове теорије су, погодности ради, задате семантички (навођењем модела), сем теорије низова која је задата синтаксно (аксиоматски). Теорије су описане у оквиру вишесортне логике првог реда са једнакошћу[3]. Поред ових теорија, за моделовање програма може се користити и исказна логика.

[3]Сва ограничења које сорте (типови) уводе могу се представити одговарајућим унарним предикатима у оквиру логике првог реда, тако да се за дефинисање ових теорија може користити и логика првог реда без сорти.

Линеарна аритметика

Целобројна линеарна аритметика (енг. Linear Integer Arithmetic, скраћено LIA) теорија је логике првог реда у којој се користе функцијски симболи:

- 0 и 1, арности 0,
- $-$, арности 1,
- $+$, арности 2

и предикатски симбол $\leq$ арности 2. Додавањем функцијских симбола за нове нумерале или за симболе $<$, $>$, $\geq$ и $\neq$, не мења се изражајна моћ теорије. Ради једноставности, ови симболи нису укључени у саму дефиницију теорије.

Теорија целобројне линеарне аритметике се може семантички описати на следећи начин: скуп њених теорема је скуп свих формула без слободних променљивих које су тачне у природној интерпретацији у структури $(\mathbb{Z}, +)$. Универзално квантификовани фрагмент ове теорије је одлучив, при чему је проблем задовољивости у универзално квантификованом фрагменту ове теорије NP-комплетан [135].

Реална линеарна аритметика (енг. Linear Real Arithmetic, скраћено LRA) се описује на аналогни начин: скуп њених теорема је скуп свих формула без слободних променљивих које су тачне у природној интерпретацији у структури $(\mathbb{R}, +)$. Универзално квантификовани фрагмент ове теорије је одлучив, и за овај фрагмент постоји процедура одлучивања чија је временска сложеност полиномска [91]. У пракси се, због добрих резултата, најчешће користе симплекс и Фурије-Моцкинов алгоритм [60, 149], иако они у најгорем случају имају експоненцијалну сложеност.

Скуп теорема *рационалне линеарне аритметике* (енг. Linear Rational Arithmetic, скраћено LQA) скуп је свих формула без слободних променљивих које су тачне у природној интерпретацији у структури $(\mathbb{Q}, +)$. Ова теорија је одлучива [114]. Уколико се утврди да је формула задовољива у оквиру рационалне линеарне аритметике, онда се тај закључак може пренети и на задовољивост у оквиру реалне линеарне аритметике јер је сваки модел над рационалним доменом истовремено и модел над реалним доменом.

У оквиру линеарне аритметике често се користе разне синтаксне скраћенице у запису формула (на пример, израз $3x$ је скраћеница за израз $x + x + x$) тако да су терми линеарне аритметике линеарни полиноми облика:

$$a_1x_1 + a_2x_2 + ... + a_nx_n \bowtie b,$$

при чему су $a_1, a_2, ..., a_n$ и b цели (или рационални) бројеви, $x_1, x_2, ..., x_n$ целобројне, рационалне или реалне променљиве, а $\bowtie$ неки од постојећих предикатских симбола.

Пример 2.1 *Пример формуле линеарне аритметике је:*

$$(x + x + x + y + y + z \leq 1) \ \wedge \ (x + (-y) \leq 0)$$

Претходна формула, коришћењем скраћеница, записује се на следећи начин:

$$(3x + 2y + z \leq 1) \ \wedge \ (x - y \leq 0)$$

Ова формула је задовољива у оквиру целобројне линеарне аритметике (па тиме и у оквиру реалне и рационалне линеарне аритметике). Један њен модел је $x = -1$, $y = 0$ *и* $z = 0$.

Аритметика бит-вектора за целе бројеве и бројеве у покретном зарезу

Различите дефиниције *теорије бит-вектора за целе бројеве* (енг. Bit-vector Arithmetic, скраћено BVA) су присутне у литератури [49, 119, 20, 31, 85]. Ова теорија мотивисана је правилима која су присутна у целобројним израчунавањима које врши процесор над низовима битова. Обично се подразумева да су бит-вектори коначни низови нула и јединица (при чему се најчешће користе дужине бит-вектора 8, 16, 32 и 64) и да се користи запис означених бројева у потпуном комплементу. Теорија аритметике бит-вектора за целе бројеве може се семантички описати на следећи начин: скуп њених теорема је скуп свих формула без слободних променљивих које су тачне у природној интерпретацији у структури бит-вектора са уобичајним операцијама и релацијама над њима. Ова теорија је одлучива, а проблем задовољивости у универзално квантификованом фрагменту ове теорије је NP-комплетан [21].

Аритметика бит-вектора за бројеве у покретном зарезу (енг. Floating-Point Arithmetic, скраћено FPA) мотивисана је правилима која су присутна у израчунавања која врши процесор над бројевима у покретном зарезу по стандарду IEEE 754 [144]. Ова теорија се може семантички описати на аналогни начин: скуп њених теорема је скуп свих формула без слободних променљивих које су тачне у природној интерпретацији у структури бит-вектора у покретном зарезу са уобичајним операцијама и релацијама над њима. Ова теорија је одлучива.

Пример 2.2 *Пример формуле теорије бит-вектора је:*

$$((x/_s 2) <_s (x + y)) \ \wedge \ (y = (x \ll 2))$$

при чему су $+$, $/_s$ *и* $\ll$ *редом ознаке за операције сабирања, означеног дељења и померања у лево у складу са одговарајућим операцијама које врши процесор над низовима битова, док је* $<_s$ *оператор означеног поређења у складу са поређењем означених бројева које врши процесор. Ова формула је задовољива и један њен модел је* $x = 2$ *и* $y = 8$.

Теорија неинтерпретираних функција

Теорије обично ограничавају интерпретацију постојећих функцијских и предикатских симбола. Теорија која не задаје никаква ограничења за функцијске симболе (и код које је скуп аксиома празан) назива се *теорија неинтерпретираних функција* (енг. theory of Equality with Uninterpreted Functions, скраћено EUF). Универзално квантификовани фрагмент ове теорије је одлучив и за овај фрагмент постоји процедура одлучивања чија је временска сложеност полиномска [125, 13]. Једна процедура одлучивања за ову теорију заснива се на конструкцији константног конгруентног затворења алгоритмом Нелсон-Опена (енг. Nelson-Oppen algorithm) [126].

Пример 2.3 *Наредне две формуле су формуле теорије неинтерпретираних функција (за функцијске симболе* f *и* g*):*

$$f(a) = a \ \wedge \ \neg(f(f(a)) = f(a))$$

$$g(a) = c \ \wedge \ (\neg((f(g(a)) = f(c)) \vee g(a) = d) \ \wedge \ \neg(c = d)$$

Обе формуле су незадовољиве.

Теорија низова

Теорија низова (енг. Theory of Arrays, скраћено ARR) уводи тернарни функцијски симбол *store* и бинарни функцијски симбол *select*.

Функција *store* врши измену вредности на одговарајућем месту у низу и има три аргумента:

(i) низ,

(ii) позицију (индекс) у низу на коју се смешта вредност,

(iii) вредност која се смешта у низ.

За задати низ a, целобројну вредност i и вредност v типа над којим је низ дефинисан, терм $store(a, i, v)$ означава низ који је идентичан са низом a, осим што је вредност на позицији i једнака v.

Функција *select* врши читање вредности са одговарајућег места у низу и има два аргумента:

(i) низ,

(ii) позицију (индекс) у низу са које се чита вредност.

За задати низ a и целобројну вредност i, терм $select(a, i)$ означава вредност на позицији i низа a.

Теорија низова је теорија која има следеће две аксиоме:

$$\forall a\ \forall i\ \forall v\ (select(store(a, i, v), i) = v) \tag{A1}$$

$$\forall a\ \forall i\ \forall j\ \forall v\ (i \neq j \Rightarrow select(store(a, i, v), j) = select(a, j)) \tag{A2}$$

Универзално кватнификовани фрагмент ове теорије је одлучив и проблем задовољивости у овом фрагменту теорије је NP-комплетан [156, 154, 35, 134]. Поред претходне две аксиоме, понекад се додаје и аксиома *проширивања* (енг. extensionality)

$$\forall a\ \forall b\ ((\forall i\ (read(a, i) = read(b, i)) \Rightarrow a = b) \tag{A3}$$

Пример 2.4 *Наредна формула је пример задовољиве формуле теорије низова:*

$$b = store(a, 5, select(a, 3))\ \wedge\ select(a, 3) = select(b, 5).$$

Пример 2.5 *Наредна формула је пример незадовољиве формуле теорије низова:*

$$\begin{aligned} store(a, i, x) \neq b \quad &\wedge \quad select(b, i) = y\ \wedge\ select(store(b, i, x), j) = y \\ &\wedge \quad a = b\ \wedge\ i = j. \end{aligned}$$

Да би се доказала незадовољивост ове формуле, потребно је користити и аксиому (A3).

2.2.2 Решавачи

Приликом аутоматске анализе исправности програма обично се генеришу услови исправности чију је ваљаност, у оквиру изабране теорије, потребно проверити адекватним системом. У наставку текста, под појмом ваљаност подразумеваће се ваљаност у оквиру изабране теорије. Систем за проверавање ваљаности може бити саставни део верификацијског алата, тј. специјализован за услове исправности које верификацијски алат генерише или независан од верификацијског алата. Као екстерни системи за проверавање ваљаности често се користе SAT и SMT решавачи. Испитивање ваљаности тада се своди на дуалан проблем испитивања задовољивости негиране формуле.

SAT решавачи

Проблем задовољивости (енг. Satisfiability problem, скраћено SAT) проблем је одлучивања да ли за исказну формулу у конјунктивној нормалној форми постоји валуација у којој су све њене клаузе тачне. Постоји велики број SAT решавача и велики број проблема који се могу решити свођењем на SAT [29]. За ефикасно утврђивање задовољивости исказних формула најчешће се користе варијације Дејвис-Патнам-Лонгман-Ловеландовог алгортима (енг. Davis-Putnam-Longman-Loveland algorithm, скраћено DPLL) [52, 53].

У верификацији, програм и услови исправности програма, могу се моделовати исказним формулама за чије се проверавање користи SAT решавач. Овај приступ се користи, на пример, у техникама проверавања ограничених модела [28]. Познати SAT решавачи су zChaff [112], MiniSAT [63] и PicoSAT [27].

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 [19]. Теорије које су од посебног интереса су теорије описане у секцији 2.2.1. Решавачи за овај проблем називају се SMT решавачи.

Модерни SMT решавачи заснивају се на DPLL(T) архитектури [129, 169] и у великој мери користе исказно расуђивање и технике уведене модерним SAT решавачима [140]. SMT-lib иницијатива има за циљ стварање библиотеке SMT тест примера и свих пратећих стандарда и нотацијских конвенција [139, 18].

SMT решавачи имају многе примене, посебно у верификацији хардвера и софтвера. Неки од најпознатијих SMT решавача су Z3 [54], Yices [61], MathSAT [38], STP [71] и Boolector [36].

2.3 Међујезици у верификацији

Анализа изворног кода програма написаног на вишем програмском језику додатно је отежана 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 облик који је погодан за анализу [99].

2.3.1 Трансформација кода

Анализа кода се најчешће врши над апстрактним синтаксним стаблом (енг. abstract syntax tree), обележеним графом контроле тока (енг. labeled flowgraphs), упрошћеним кодом на истом програмском језику, или над кодом међујезика који одваја концепте и семантику вишег програмског језика од језика који одговара конкретној машини на којој се програм извршава.

Примери платформи које врше трансформацију програма у међујезик погодан за анализу су LLVM и .NET. Ове платформе имају приступне компоненте (енг. front-ends) за различите програмске језике.

У оквиру .NET платформе развијена је библиотека Code Contracts. Ова библиотека пружа подршку јединственом начину назначавања спецификације програма у оквиру кода [15]. Писање спецификације у оквиру програма омогућава аутоматску проверу да ли је спецификација задовољена. За реализацију ове провере користи се верификацијски међујезик и алат Boogie [16]. Верификацијски међујезик има улогу да раздвоји семантику дефинисану програмским језиком од генерисања формула за решавач. Boogie повезује јединствени верфикацјски алат у оквиру платформе са библитеком Code Contracts тако што трансформише компилирани кôд у верификацијски међујезик и на основу међујезика гради услове исправности програма. Постоје и други верификацијски алати који користе архитектуру у чијој се основи налази верификацијски међујезик [67, 40].

Трансформација кода у облик погодан за верификацију посебно је важна за програме написане на програмском језику C, па постоје многи специјализовани трансформатори за C програме [124]. У наставку текста биће детаљније приказана LLVM платформа и LLVM репрезентација кода.

2.3.2 LLVM

Почетна верзија LLVM пројекта имала је за циљ да обезбеди подршку за компилацију произвољног програмског језика [100, 101]. Временом је тај пројекат прерастао у низ различитих потпројеката који се широко користе у оквиру комерцијалне производње софтвера, пројеката отвореног кода, као и у истраживачке сврхе[4]. У верификацији софтвера, LLVM се користи, на пример, у оквиру алата KLEE [39], Calysto [11], S2E [42], LLBMC [117] и SAFECode [57]. Кôд LLVM пројекта дистрибуира се под лиценцом NCSA отвореног кода Универзитета Илиноис [107].

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* (енг. LLVM intermediate representation). 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 [111], Ruby [14], Haskell [78], Java [165], D [51], Pure [108], Scala [141] и Lua [110]) у LLVM међујезик.

LLVM међујезик има једноставне инструкције које су налик на асемблерске RISC инструкције али, за разлику од асемблерских инструкција, садрже и информације о типовима података операнада инструкција као и о њиховој употреби у програму. Инструкције LLVM међујезика су у форми *статички јединствене доделе* (енг. Static Single Assignment, скраћено SSA) [50]. Информације о типовима, SSA форма и постојећа библиотечка подршка чине LLVM међујезик погодним за развијање нових врста трансформација, анализа и оптимизација.

Свака функција програма на LLVM међујезику састоји се од *блокова*. Сваки блок је низ инструкција за који извршавање може да почне само у улазној тачки блока, односно кроз прву инструкцију блока, а извршавање блока може да се заврши само кроз излазну тачку блока, односно кроз последњу инструкцију. За сваки блок, познати су његови претходници и следбеници, и за сваку функцију познат је граф контроле тока функције. Пример 2.6 илуструје структуру и неке основне инструкције LLVM кода.

[4]LLVM пројекат је добио престижну награду *ACM System Software Award* за 2012. годину. Ова награда се сваке године додељује једном изузетном софтверском систему, а ранијих година, добили су је и системи *Eclipse*, *VMware*, *Make*, *Java*, *Apache*, *TCP/IP*, *PostScript*, *SMALLTALK*, *TeX* и *UNIX*.

Пример 2.6 *За програм који рачуна минимум два унета броја:*

```
#include <stdio.h>

int main()
{
  int a, b, min;
  printf("Unesi dva broja\n");
  scanf("%d%d", &a, &b);
  min = a;
  if(b<min) min = b;
  printf("Najmanji broj je %d\n", min);
  return 0;
}
```

генерише се LLVM кôд приказан на слици 2.1. Следи кратко објашњење основних елемената LLVM кода који су присутни у овом примеру.

Коментари се у оквиру LLVM кода записују иза ознаке `;` *и садрже додатне информације о самим LLVM наредбама, као што су тип промењљиве над којом је извршена додела вредности, број коришћења промењљиве у наставку кода, или имена блокова претходника датог блока. У оквиру* `main` *функције постоје четири блока:* `entry`, `bb`, `bb1` *и* `return`. *На почетку улазног блока, тј. блока* `entry`, *алоцирана је меморија за све локалне промењљиве функције* `main` *инструкцијом* `alloca`. *Локалне промењљиве функције садрже префикс* `%`. *Промењљиве које служе за складиштење привремених резултата (да би програм био у форми статички јединствене доделе) уместо именима означавају се бројевима. Кôд садржи инструкције за учитавање и уписивање садржаја у меморију, тј. инструкције* `load` *и* `store`, *инструкцију целобројног аритметичког поређења* `icmp slt` *и инструкцију позива функције* `call`. *Сваки блок, сем излазног блока* `return`, *завршава се инструкцијом* `br` *условног (блок* `entry`*) или безусловног скока (блокови* `bb` *и* `bb1`*) која одређује којим се блоком наставља извршавање програма.*

```
; ModuleID = 'minimum.o'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-
                     f32:32:32-f64:32:64-v64:64:64-v128:128:128-a0:0:64-f80:32:32"
target triple = "i386-pc-linux-gnu"

@.str = internal constant [16 x i8] c"Unesi dva broja\00" ; <[16 x i8]*> [#uses=1]
@.str1 = internal constant [5 x i8] c"%d%d\00"    ; <[5 x i8]*> [#uses=1]
@.str2 = internal constant [21 x i8] c"Najmanji broj je %d\0A\00" ; <[21 x i8]*> [#uses=1]

define i32 @main() nounwind {
entry:
  %retval = alloca i32                            ; <i32*> [#uses=2]
  %min = alloca i32                               ; <i32*> [#uses=4]
  %b = alloca i32                                 ; <i32*> [#uses=3]
  %a = alloca i32                                 ; <i32*> [#uses=2]
  %0 = alloca i32                                 ; <i32*> [#uses=2]
  %"alloca point" = bitcast i32 0 to i32          ; <i32> [#uses=0]
  %1 = call i32 @puts(i8* getelementptr ([16 x i8]* @.str, i32 0, i32 0)) nounwind
                                                  ; <i32> [#uses=0]
  %2 = call i32 (i8*, ...)* @scanf(i8* noalias getelementptr ([5 x i8]* @.str1,
                                  i32 0, i32 0), i32* %a, i32* %b) nounwind
                                                  ; <i32> [#uses=0]
  %3 = load i32* %a, align 4                      ; <i32> [#uses=1]
  store i32 %3, i32* %min, align 4
  %4 = load i32* %b, align 4                      ; <i32> [#uses=1]
  %5 = load i32* %min, align 4                    ; <i32> [#uses=1]
  %6 = icmp slt i32 %4, %5                        ; <i1> [#uses=1]
  br i1 %6, label %bb, label %bb1

bb:                                               ; preds = %entry
  %7 = load i32* %b, align 4                      ; <i32> [#uses=1]
  store i32 %7, i32* %min, align 4
  br label %bb1

bb1:                                              ; preds = %bb, %entry
  %8 = load i32* %min, align 4                    ; <i32> [#uses=1]
  %9 = call i32 (i8*, ...)* @printf(i8* noalias getelementptr ([21 x i8]* @.str2,
                                   i32 0, i32 0), i32 %8) nounwind
                                                  ; <i32> [#uses=0]
  store i32 0, i32* %0, align 4
  %10 = load i32* %0, align 4                     ; <i32> [#uses=1]
  store i32 %10, i32* %retval, align 4
  br label %return

return:                                           ; preds = %bb1
  %retval2 = load i32* %retval                    ; <i32> [#uses=1]
  ret i32 %retval2
}

declare i32 @puts(i8*)

declare i32 @scanf(i8* noalias, ...) nounwind

declare i32 @printf(i8* noalias, ...) nounwind
```

Слика 2.1: *LLVM кôд генерисан на основу C програма који рачуна минимум два унета броја.*

3

Опис система LAV

У овој глави биће описан систем LAV — систем за статичку анализу, генерисање и проверу услова исправности императивних програма [168]. Систем комуницира са другим системима у окружењу ради трансформисања кода у форму која је погодна за анализу и ради добијања информација о ваљаности генерисаних формула. Систем обухвата моделовање понашања програма, конструкцију услова исправности програма, трансформисање конструисаних услова у формуле одговарајућих теорија, испитивање ваљаности формула у оквиру изабране теорије и генерисање одговарајућег извештаја за корисника.

У оквиру система LAV, испитивање исправности програма своди се на испитивање да ли у задатој тачки програма:

(1) неки израз може да има неку (недозвољену) вредност;

(2) нека адреса не припада границама резервисане меморије придружене одговарајућем показивачу.

Према томе, грешке које могу да буду обухваћене конкретном имплементацијом система су грешке које се могу дефинисати на један од претходна два начина. Такве су, на пример, грешке дељења нулом, дереференцирања NULL показивача, прекорачење бафера и неиспуњавање предуслова функције за коју је познат уговор у виду аритметичко-логичког израза.

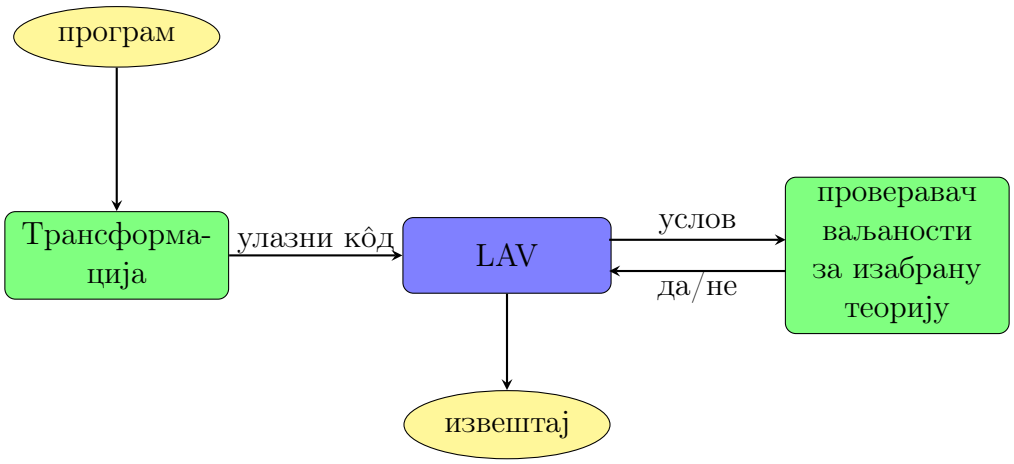
Испитивање функционалне исправности програма може бити вршено на описани начин. То се може постићи додавањем позива специјалне функције `assert` у тачки изворног програма за коју желимо да проверимо важење неког услова. Аргумент ове функције треба да одговара услову за који се врши провера. Проверавање исправности програма може се олакшати додавањем услова за који се претпоставља да важи у некој тачки изворног програма, навођењем жељеног услова у оквиру позива специјалне функције `assume`.

3.1 Окружење

Систем LAV анализира програм за који се претпоставља да се састоји од блокова инструкција, при чему извршавање блока може да почне само у улазној тачки блока, а може да се заврши само кроз последњу инструкцију блока. Такође, претпоставља се да инструкције користе само један оператор или позив функције (поред, евентуално, оператора доделе), као и да се на основу последње инструкције блока може закључити којим блоковима у програму је могуће настављање извршавања програма. Овакав облик улазног програма може да се добије, на пример, трансформисањем програма на императивном програмском језику у LLVM кôд и имплементација система LAV као улазне програме користи управо LLVM програме. У наставку ће, за илустрацију примене система, поред LLVM кода, једноставности ради, бити коришћен кôд на програмском језику C и, тамо где је то погодно, поједностављени LLVM кôд. Поједностављени LLVM кôд, који је уведен за потребе овог текста, неформално апстрахује неке детаље језика LLVM како би се олакшало праћење кода који се анализира. На пример, уместо префиксних аритметичко-логичких инструкција користе се одговарајући инфиксни C оператори, подаци о типовима операнада се не наводе, а уместо LLVM инструкција `load` и `store` учитавање и уписивање у меморију се врши наредбама доделе.

Систем LAV гради модел извршавања програма и генерише услове исправности програма. Модел извршавања програма и услови исправности се исказују на језику вишесортне логике првог реда теорије која одговара семантици програмског језка. Овај језик и одговарајућа теорија неће бити експлицитно описивани. LAV трансформише изграђене формуле у формуле изабране одлучиве теорије логике првог реда за које постоји проверавач ваљаности. Избор теорије у коју LAV трансформише изграђене формуле може се извршити на основу анализираног кода, али и на основу жељених критеријума за прецизност резултата и ефикасност расуђивања. На основу комуникације са проверавачем ваљаности, LAV генерише извештај о безбедности наредби у оригиналном програму. Утицај избора теорије на генерисање резултата анализе детаљно је описан у одељку 3.5.

Комуникације система LAV са окружењем приказана је на слици 3.1.



Слика 3.1: *Комуникација система LAV са окружењем.*

3.2 Моделовање основних градивних јединица програма

Да би се конструисале формуле које одговарају току извршавања програма и условима исправности програма, неопходно је најпре моделовати основне градивне јединице програма: промењиве (локалне, показивачке и глобалне), типове података (основне и сложене), инструкције, блокове инструкција и позиве функција. Моделовање променљивих, типова података, инструкција и блокова инструкција описано је у овом одељку, док је ефекат позива функције описан у одељку о моделовању контроле тока (секција о интерпроцедуралној анализи 3.3.2).

3.2.1 Промењиве и основни типови података

Промењиве основних типова података програма моделују се промењивама вишесортне логике првог реда. Различитим типовима података програмског језика (целобројном, реалном и показивачком типу) придружују се различите сорте у логици. И истоврсним типовима података (на пример целобројним) за чије се складиштење користе различите дужине у бајтовима, придружују се различите сорте. Промењиве сложених типова података, као што су структуре и уније, дискутоване су у секцији 3.2.4.

Да би се омогућило превођење типова података у одговарајуће сорте, претпоставља се да трансформатор може да разреши недорећености стандарда језика C по питању дужина целобројних типова у складу са машином на којој се извршава или на неки други начин. LLVM ово питање разрешава у складу са

архитектуром рачунара на којој је програм трансформисан у LLVM међујезик. На овај начин, анализа програма се спроводи само за ту конкретну архитектуру и постоји могућност да се нека грешка, која се на овој архитектури не испољава, може испољити на некој другој архитектури.

Пример 3.1 *Једнобајтном целобројном типу* `char` *придружује се једна сорта, назовимо је* τ_1*, а четворобајтном целобројном типу* `int` *придружује се друга сорта, назовимо је* τ_4*. Једнобајтним целобројним променљивама* `a` *и* `b`*, које су декларисане у програму са* `char a, b;` *придружују се променљиве* a *и* b *сорте* τ_1*. Четворобајтним целобројним променљивама* `c` *и* `d` *које су декларисане у програму са* `unsigned int c; signed int d;` *придружују се променљиве* c *и* d *сорте* τ_4*. Приметимо да је променљивама* c *и* d *придружена иста сорта* τ_4 *иако је променљива* `c` *неозначеног, а променљива* `d` *означеног целобројног типа. Својства оваквог моделовања биће дискутована у одељку 3.5.*

У даљем тексту ће се подразумевати да се под логиком првог реда мисли на вишесортну логику првог реда. Такође, сорту променљиве у формули која моделује програм зваћемо и типом променљиве. Ради једноставнијег записа, у наставку текста, у формулама неће бити навођене сорте променљивих.

3.2.2 Основне инструкције и блокови

Систем LAV користи *складиште* за праћење вредности променљивих у оквиру блокова програма. Складиште (енг. store) скуп је променљивих блока b којем је придружено пресликавање *складиштење* S_b. Ово пресликавање додељује вредност свакој променљивој складишта, при чему су могуће вредности променљиве одређене типом променљиве. Променљиве чије су вредности релевантне у различитим блоковима функције припадају сваком складишту блока функције и називају се *сталне* променљиве. Променљиве чије су вредности релевантне искључиво у оквиру једног блока, припадају само складишту блока у којем су настале и називају се *привремене* променљиве.

Пример 3.2 *У једноставном програму који је приказан на слици 3.2, издваја се пет блокова. Први блок (entry) чине инструкције које су настале превођењем наредби C програма у линијама 3, 4, 5*[1]*; други блок (*`bb`*) чине инструкције које су настале превођењем наредбе C програма у линији 6; трећи блок (*`bb1`*) чине инструкције које су настале превођењем наредбе C програма у линији 8;*

[1] Линије 5, 6, 7 и 8 заправо чине једну C наредбу коју LLVM разбија на неколико инструкција које припадају различитим блоковима.

четврти блок (`bb2`*) чине инструкције које су настале превођењем наредбе C програма у линији 9; последњи блок (return) завршава рад функције. Складиштима сваког блока припадају променљиве* `a`*,* `b` *и* `c` *које су целобројног четворобајтног типа. Складишту блока* `bb` *припадају и две привремене променљиве (*`r5` *и* `r6`*), чије се вредности користе само у оквиру овог блока.*

Свака инструкција блока може да утиче на складиште блока и може да дода ограничења над скупом променљивих. Симболичким извршавањем, које је описано у секцији 2.1.2 се на основу инструкције кода конструише израз логике првог реда над текућим вредностима променљивих у складишту. Овај израз постаје нова вредност одговарајуће променљиве складишта (уколико је присутна инструкција доделе) или се додаје у скуп додатних ограничења (о чему ће бити више речи у наставку текста).

Пример 3.3 *За наредбу сабирања* `c=a+b;` *и текуће вредности одговарајућих променљивих у складишту* a*,* b *и* c*, симболичким извршавањем конструише се израз* $a + b$ *који, због оператора доделе, постаје нова вредност променљиве* `c`*.*

Означимо са $s_b(v)$ вредност променљиве v у улазној тачки блока b, а са $e_b(v)$ вредност променљиве v у излазној тачки[2]. Све инструкције блока се редом симболички извршавају.

Пример 3.4 *У табели 3.1 приказана је измена функције* S_{bb} *складиштења блока* `bb` *са слике 3.2 након симболичког извршавања сваке инструкције. На уласку у блок, сталне променљиве складишта пресликавају се у своје иницијалне вредности. Вредности привремених променљивих у овом тренутку нису дефинисане. Након симболичког извршавања прве инструкције блока* `bb`*, вредност променљиве* `r5` *поставља се на текућу вредност променљиве* `a`*. Након симболичког извршавања друге инструкције, дефинише се вредност за променљиву* `r6`*, а након симболичког извршавања треће инструкције, мења се вредност променљиве* `a`*.*

Табела 3.1: *Вредности променљивих складишта блока* `bb` *са слике 3.2 након симболичког извршавања сваке инструкције.*

наредба	`a`	`b`	`c`	`r5`	`r6`
`bb:`	$s_{bb}(a)$	$s_{bb}(b)$	$s_{bb}(c)$	—	—
`r5 = a;`	$s_{bb}(a)$	$s_{bb}(b)$	$s_{bb}(c)$	$s_{bb}(a)$	—
`r6 = r5+1;`	$s_{bb}(a)$	$s_{bb}(b)$	$s_{bb}(c)$	$s_{bb}(a)$	$s_{bb}(a)+1$
`a = r6;`	$s_{bb}(a)+1$	$s_{bb}(b)$	$s_{bb}(c)$	$s_{bb}(a)$	$s_{bb}(a)+1$

[2] $s_b(v)$ је скраћеница за $start_b(v)$, док је $e_b(v)$ скраћеница за $end_b(v)$.

```
1. int main()
2. {
3.  int a,b,c;
4.  scanf("%d%d", &b, &c);
5.  if(b>c)
6.    a++;
7.  else
8.    a--;
9.  printf("%d\n", b/a);
10. return 0;
11. }
```

```
int main() {
entry:
  int a, b, c;
  scanf("%d%d", &b, &c);
  r2 = b;
  r3 = c;
  r4 = (b>c) ? 1:0;
  br r4 ? bb : bb1;

bb:
  r5 = a;
  r6 = r5 + 1;
  a = r6;
  br bb2;

bb1:
  r7 = a;
  r8 = r7 - 1;
  a = r8;
  br bb2;

bb2:
  r9 = b;
  r10 = a;
  r11 = r9/r10;
  printf("%d\n", r11);
  br return;

return:
  return 0;
}
```

```
; ModuleID = '1.c'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8
       -i16:16:16-i32:32:32-i64:32:64-f32:32:32
       -f64:32:64-v64:64:64-v128:128:128-a0:0:64-f80:32:32"
target triple = "i386-pc-linux-gnu"
@.str = internal constant [5 x i8] c"%d%d\00"
; <[5 x i8]*> [#uses=1]
@.str1 = internal constant [4 x i8] c"%d\0A\00"
; <[4 x i8]*> [#uses=1]

define i32 @main() nounwind {
entry:
%retval = alloca i32 ; <i32*> [#uses=2]
%c = alloca i32 ; <i32*> [#uses=2]
%b = alloca i32 ; <i32*> [#uses=3]
%a = alloca i32 ; <i32*> [#uses=5]
%0 = alloca i32 ; <i32*> [#uses=2]
%"alloca point" = bitcast i32 0 to i32 ; <i32> [#uses=0]
%1 = call i32 (i8*, ...)* @scanf(i8* noalias
     getelementptr ([5 x i8]* @.str, i32 0, i32 0),
     i32* %b, i32* %c) nounwind ; <i32> [#uses=0]
%2 = load i32* %b, align 4 ; <i32> [#uses=1]
%3 = load i32* %c, align 4 ; <i32> [#uses=1]
%4 = icmp sgt i32 %2, %3 ; <i1> [#uses=1]
br i1 %4, label %bb, label %bb1

bb: ; preds = %entry
%5 = load i32* %a, align 4 ; <i32> [#uses=1]
%6 = add i32 %5, 1 ; <i32> [#uses=1]
store i32 %6, i32* %a, align 4
br label %bb2

bb1: ; preds = %entry
%7 = load i32* %a, align 4 ; <i32> [#uses=1]
%8 = sub i32 %7, 1 ; <i32> [#uses=1]
store i32 %8, i32* %a, align 4
br label %bb2

bb2: ; preds = %bb1, %bb
%9 = load i32* %b, align 4 ; <i32> [#uses=1]
%10 = load i32* %a, align 4 ; <i32> [#uses=1]
%11 = sdiv i32 %9, %10 ; <i32> [#uses=1]
%12 = call i32 (i8*, ...)* @printf(i8* noalias
      getelementptr ([4 x i8]* @.str1, i32 0, i32 0),
      i32 %11) nounwind ; <i32> [#uses=0]
store i32 0, i32* %0, align 4
%13 = load i32* %0, align 4 ; <i32> [#uses=1]
store i32 %13, i32* %retval, align 4
br label %return

return: ; preds = %bb2
%retval3 = load i32* %retval ; <i32> [#uses=1]
ret i32 %retval3
}

declare i32 @scanf(i8* noalias, ...) nounwind
declare i32 @printf(i8* noalias, ...) nounwind
```

Слика 3.2: *Једноставан C кôд (са леве стране, горе), одговарајући LLVM кôд (са десне стране), поједностављени LLVM кôд (са леве стране, доле).*

Након што се свака инструкција блока функције f симболички изврши, конструише се формула $Transformation(b)$. Ова формула конструише се користећи вредности сталних промењљивих и описује како инструкције блока b утичу на складиште:

$$Transformation(b) = \bigwedge_{v \in V_f} (e_b(v) = e_v) \ \wedge \ \bigwedge AdditionalConstraints(b)$$

при чему је:

- V_f скуп сталних промењљивих (промењљивих заједничких за сва складишта функције f);
- e_v вредност промењљиве v на излазу из блока, ова вредност је изражена у терминима:
  - иницијалних вредности $s_b(v)$, за $v \in V_f$,
  - константи,
  - новоуведених промењљивих (које су настале као резултат апроксимације симболичког извршавања) и над којима су евентуално задата некаква ограничења;
- $AdditionalConstraints(b)$ формула која садржи додатна ограничења која се користе за моделовање појединих операција (као што ће бити описано у наставку ове главе).

Формула $Transformation(b)$ је имплицитно универзално квантификована.

Пример 3.5 *За пример са слике 3.2, симболичко извршавање блока* bb *приказано је у табели 3.1. Трансформација за блок* `bb` *је задата формулом:*

$$\begin{aligned} Transformation(bb) = (e_{bb}(a) = s_{bb}(a) + 1) \\ \wedge (e_{bb}(b) = s_{bb}(b)) \\ \wedge (e_{bb}(c) = s_{bb}(c)) \end{aligned}$$

У овом примеру, скуп додатних ограничења је празан.

Формула $Transformation(b, i)$ се дефинише аналогно, али узимајући у обзир само првих i инструкција блока b.

Табела 3.2: *Вредности промењљивих складишта блока* `entry` *са слике 3.2 након извршавања сваке наредбе.*

наредба	a	b	c	r2	r3	r4
`entry:`	$s_{entry}(a)$	$s_{entry}(b)$	$s_{entry}(c)$	—	—	—
`scanf("%d%d", &b,&c)`	$s_{entry}(a)$	b_1	c_1	—	—	—
`r2=b;`	$s_{entry}(a)$	b_1	c_1	b_1	—	—
`r3=c;`	$s_{entry}(a)$	b_1	c_1	b_1	c_1	—
`r4=(b>c);`	$s_{entry}(a)$	b_1	c_1	b_1	c_1	$(b_1 > c_1)$

Пример 3.6 *За пример са слике 3.2, симболичко извршавање блока* bb *приказано је у табели 3.1. Формула* $Transformation(bb, 2)$ *описује трансформацију за блок* `bb` *узимајући у обзир само прве* 2 *инструкције:*

$$\begin{aligned} Transformation(bb, 2) = (e_{bb}(a) = s_{bb}(a)) \\ \wedge (e_{bb}(b) = s_{bb}(b)) \\ \wedge (e_{bb}(c) = s_{bb}(c)) \end{aligned}$$

И у овом примеру, скуп додатних ограничења је празан.

3.2.3 Инструкција гранања

Инструкција гранања задаје *излазна правила* за блок b. Излазна правила (c_1, b_1), (c_2, b_2), ..., (c_k, b_k) одређују којим блоком ће се наставити извршавање програма након блока b. Уколико је испуњен *излазни услов* c_i, тада је следећи блок који ће се извршити блок b_i. Ови услови могу да се изразе у терминима почетних вредности промењљивих блока, константи и новоуведених промењљивих, на аналоган начин као што се изражавају услови у израчунавању вредности промењљивих које учествују у формули $Transformation(b)$. Истинитосне вредности излазних услова блока b означаваћемо са $e_b(c_i)$.

Пример 3.7 *За пример са слике 3.2, симболичко извршавање блока* entry *приказано је у табели 3.2.*

Трансформација за блок `entry` *је задата формулом:*

$$\begin{aligned} Transformation(entry) = (e_{entry}(a) = s_{entry}(a)) \\ \wedge (e_{entry}(b) = b_1) \\ \wedge (e_{entry}(c) = c_1) \end{aligned}$$

при чему су b_1 *и* c_1 *новоуведене промењљиве над којима нису задата никаква ограничења и стога представљају произвољне вредности (што одговара ефекту*

позива функције `scanf`*). Ефекат позива произвољне функције биће разјашњен у секцији 3.3.2. Излазна правила су*

- $(r4, bb)$ — *уколико је испуњен излазни услов* $r4$*, извршавање програма се наставља у блоку* bb*;*
- $(\neg r4, bb1)$ — *уколико је испуњен излазни услов* $\neg r4$*, извршавање програма се наставља у блоку* $bb1$*.*

Вредности излазних услова су

$$e_{entry}(r4) = (b_1 > c_1)$$

$$e_{entry}(\neg r4) = \neg(b_1 > c_1)$$

3.2.4 Показивачи, бафери и сложени типови података

Прекорачење бафера и дереференцирање неисправних показивача су важни верификацијски проблеми, као што је то описано у одељку 2.1. Да би се омогућило детектовање ових грешака, посебна пажња у моделовању програма се поклања показивачима и простору који је за њих алоциран. Приметимо да се сложени типови података виших програмских језика, као што су то, на пример, структуре и уније у програмском језику C, приликом трансформације кода, могу превести у бафере којима се такође приступа уз помоћ вредности показивача и помераја у односу на тај показивач. Према томе, третирање сложених типова података може да се сведе на моделовање и рад са баферима.

Да би се пратила безбедност приступа меморији у оквиру бафера и открило потенцијално прекорачење бафера, вредностима показивача, који се третирају као и остале промењљиве и прате у оквиру складишта података, придружене су и одговарајуће величине алоциране меморије. За праћење величине алоцираног простора којем се приступа уз помоћ показивача p, уводе се две функције: $left(p)$ и $right(p)$. Вредност n функције $left(p)$ казује да се први резервисан бајт за показивач p налази на месту у меморији $p+n$, док вредност функције $right(p)$ казује да се последњи резервисан бајт за показивач p налази на месту у меморији $p + n$. Подразумева се да је сав простор између крајњег левог и крајњег десног алоцираног бајта такође алоциран. Вредности ових функција могу бити и негативни бројеви и нула. Специјално, уколико су обе ове вредности једнаке нули, онда то значи да је реч о показивачу за који није алоцирана меморија. То може да буде случај за неиницијализовани показивач, NULL показивач или показивач за који је меморија већ ослобођена.

Пример 3.8 *За низ* `a`*, за који је алоцирана меморија наредбом* `char a[10];` *функција* $left(a)$ *има вредност* 0*, а функција* $right(a)$ *има вредност* 10*. Након наредбе* `char* p=a+3;` *важи да је* $left(p) = -3$ *и* $right(p) = 7$*. За* `char* p1=a+20;` *важи да је* $left(p1) = -20$ *и* $right(p1) = -10$*. За* `char* p2=a-10;` *важи да је* $left(p2) = 10$ *и* $right(p2) = 20$*.*

Пример 3.9 *Наредба*

```
*(p+i)
```

где је `p` *показивач на целобројну вредност, а* `i` *целобројни померај, доводи до прекорачења бафера ако и само ако наредни услов није задовољен:*

$$left(p) \leq i \cdot sizeof(int) < right(p)$$

Функција sizeof (која је уведена по угледу на оператор `sizeof` *програмског језика C), у наведеним примерима, разрешава проблем недоречености C стандарда по питању величина основних типова података. Ову недореченост у оквиру самог система LAV разрешава трансформатор кода, односно LLVM, о чему је било речи у секцији 3.2.1.*

Приликом сваке алокације меморије (било да је у питању статичка или динамичка алокација меморије) у скуп додатних ограничења за блок b којем наредба алокације припада, тј. у формулу $AdditionalConstraints(b)$, додаје се конјункција услова за алоцирани простор у терминима функција $left$ и $right$. Прецизније, вредност функције $left$ за показивач за који се алоцира меморија се поставља на 0, док се вредност функције $right$ поставља на одговарајућу позитивну вредност броја резервисаних бајтова. За показивач за који није алоцирана меморија, оба ова броја се постављају на нулу.

Пример 3.10 *Алокацијом меморије наредбом* `char a[10];` *у формулу AdditionalConstraints додаје се конјункција* $left(a) = 0 \wedge right(a) = 10$*.*

Пример 3.11 *Декларацијом показивача* `char* p;` *не алоцира се простор за показивач* p *тако да се у формулу AdditionalConstraints додаје конјункција* $left(p) = 0 \wedge right(p) = 0$*.*

Особине показивача дозвољавају да се вредност показивача p_1 дефинише у терминима другог показивача p и целобројног помераја n у односу на тај показивач (целобројни померај може да буде константа или произвољан целобројни израз). Тада је, на основу расположивих информација за показивач p,

потребно одредити алоциран простор за показивач $p_1 = p + n$. Односно, потребно је одредити вредности функција $left(p_1)$ и $right(p_1)$, како би се могла проверити безбедност приступа меморији преко овог показивача. Треба уочити да увек важи:

$$left(p + n) = left(p) - n \tag{a1}$$

$$right(p + n) = right(p) - n \tag{a2}$$

Ове једнакости могу се сматрати аксиомама. Уместо увођења аксиома (као универзално квантификованих формула) у генерисану формулу, додају се само *релевантне* инстанце ових формула у скуп додатних ограничења блока. Прецизније, за сваку формулу у којој учествује израз облика $left(p + n)$, додаје се одговарајућа једнакост $left(p+n) = left(p)-n$, а за сваку формулу у којој учествује израз облика $right(p+n)$, додаје се једнакост $right(p+n) = right(p)-n$.

Пример 3.12 *За наредни кôд*

```
char a[10];
char* p;
p=a+3;
```

вредности промењљивих складишта и генерисана додатна ограничења дата су у наредној табели:

наредба	a	p	*AdditionaConstraints*
`char a[10];`	a_1	–	$left(a_1) = 0 \wedge right(a_1) = 10$
`char* p;`	a_1	p_1	$left(p_1) = 0 \wedge right(p_1) = 0$
`p=a+3;`	a_1	$a_1 + 3$	–

Након претходне три наредбе, броју бајтова резервисане меморије показивача p са његове десне стране одговара формула $right(a_1+3)$, *јер је* a_1+3 *текућа вредност показивача p. Коришћењем аксиоме (a2) и ограничења додатог приликом резервисања меморије за показивач a, може се израчунати ова вредност:*

$$right(p) = right(a_1 + 3) = right(a_1) - 3 = 10 - 3 = 7$$

Слично се може израчунати и број бајтова резервисане меморије показивача p са његове леве стране:

$$left(p) = left(a_1 + 3) = left(a_1) - 3 = 0 - 3 = -3$$

На овај начин се за сваки показивач може одредити број бајтова резервисане меморије бафера на који показивач указује, што се користи код проверавања

услова исправности наредби (услови исправности наредби описани су у одељку 3.4).

3.2.5 Меморијске локације на које указују показивачи

Систем LAV прати вредности променљивих које се чувају на стеку коришћењем складишта, осим за оне променљиве над којима се користи оператор референцирања и чијим вредностима је могућ приступ и уз помоћ неког показивача. Такве променљиве, као и локације којима се приступа искључиво уз помоћ показивача (динамички алоцирана меморија на хипу), прате се коришћењем посебног меморијског модела.

Ради једноставности и прецизности (а по цену ефикасности), систем LAV користи *раван меморијски модел* (енг. flat memory model). У оквиру овог меморијског модела, читава меморија се третира као јединствени низ меморијских локација mem. Вредност променљиве mem прати се такође кроз складиште и она се мења приликом симболичког извршавања, изменом меморијских локација у низу. За моделовање наредби, које приступају меморији преко показивача или приступају вредности променљиве која се не прати преко складишта, тј. за приступ елементима низа mem, користе се функције *store* и *select*. Функција *store* користи се за уписивање вредности на одговарајуће место у низ, а функција *select* користи се за читање вредности са одговарајућег места у низу. Семантика ових функција дефинисана је у духу стандардне теорије низова (описане у секцији 2.2.1), узимајући у обзир да ће се у фази решавања користити баш теорија низова.

Пример 3.13 *Након наредбе* `p[i+3]=5;` *где је* `p` *показивач на целобројну вредност, а* `i` *целобројни померај, вредност низа који моделује меморију, измењена је и број* `5` *је уписан на позицију која се израчунава као збир вредности показивача* `p` *и помераја* `i+3`*. Формула која ово описује је:*

$$mem' = store(mem, p + (i + 3) \cdot sizeof(int), 5)$$

при чему је mem' нова, измењена вредност променљиве која моделује меморију, а која је пре наредбе доделе имала вредност mem. У складишту се измена меморије може пратити на два начина, коришћењем нове променљиве mem' или коришћењем израза $store(mem, p + (i + 3) \cdot sizeof(int), 5)$, о чему ће бити више речи у наставку текста.

Пример 3.14 *Вредност која се додељује променљивој* `v` *целобројног типа након доделе* `v=p[i+3];` *где је* `p` *показивач на целобројну вредност, а* `i` *целобројни*

померај, вредност је у меморији која се налази на позицији која се израчунава као збир вредности показивача `p` *и помераја* `i+3`*. Формула која ово описује је:*

$$v = select(mem, p + (i + 3) \cdot sizeof(int)),$$

при чему променљива mem *означава тренутну вредност низа који моделује меморију.*

Пример 3.15 *Претпоставимо да је извршавање наредби* `p[i+3]=5; v=p[i+3];` *безбедно (односно да је за показивач* `p` *приступ* `p[i+3]` *у оквиру резервисане меморије бафера на који показивач указује). Извршавање ових наредби описује се формулама (c1) и (c2):*

`p[i+3]=5;` $\quad mem' = store(mem, p + (i + 3) \cdot sizeof(int), 5) \quad$ (c1)

`v=p[i+3];` $\quad v = select(mem', p + (i + 3) \cdot sizeof(int)) \quad$ (c2)

при чему је вредност променљиве, која моделује меморију пре извршавања прве наредбе, једнака mem*.*

Заменом вредности променљиве mem' *на основу формуле (c1) у формули (c2) добија се формула*

$$v = select(store(mem, p + (i + 3) \cdot sizeof(int), 5), p + (i + 3) \cdot sizeof(int))$$

из које, на основу аксиоме (A1) теорије низова (секција 2.2.1), закључујемо да је $v = 5$*, што одговара семантици извршавања ове две наредбе.*

Вредност променљиве mem у складишту може се пратити на два начина.

1. Нова вредност променљиве mem формира се применом функције *store* на текућу вредност променљиве mem. На овај начин се вредност меморије прати кроз формулу која садржи узастопну примену *store* функција које дефинишу тренутни садржај меморије.

2. Нова вредност променљиве mem обележава се увођењем нове променљиве mem' чија се вредност прецизира додавањем израза који у себи садржи примену функције *store* на текућу вредност променљиве mem у скуп додатних ограничења. На овај начин се вредност меморије у складишту увек описује променљивом над којом су ограничења задата у скупу додатних ограничења.

Оба ова присупа могу водити до ефикаснијег рада система у различитим околностима. Добра страна првог приступа је што у неким ситуацијама омогућава

ефикасније одређивање садржаја меморије, док је добра страна другог приступа што избегава формирање гломазних израза који описују меморију и који могу значајно да успоре рад система.

Као што је већ поменуто, уколико се користи оператор референцирања над неком локалном променљивом у оквиру функције, онда се вредност те променљиве не прати кроз складиште, као за остале променљиве, већ се прати кроз садржај меморије. Вредност адресе променљиве `a` (која је резултат референцирања променљиве `a`, тј. вредност `&a`), се описује променљивом $addr\_a$.

Пример 3.16 *Ако се над променљивом* `a` *користи оператор референцирања, када се променљивој* `a` *додељује нека вредност, рецимо* `v`*, то се моделује тако што се са* $store(mem, addr\_a, v)$*, замењује вредност променљиве која моделује меморију и која је до тог тренутка имала вредност* mem*. С друге стране, ако се чита вредност променљиве* `a`*, тада се чита вредност из меморије са* $select(mem, addr\_a)$ *(уколико претпоставимо да је* mem *тренутна вредност променљиве која моделује меморију).*

За време реалног извршавања програма, свим активним променљивама и динамичким објектима додељују се непреклапајућа места у меморији. То се у оквиру система LAV може моделовати додавањем услова облика $p \neq q$ за сваки пар (p, q) адреса променљивих или динамичких објеката. Уколико су p и q бафери, тада је за њихово непреклапање потребно додати и услове који гарантују да се адресе разликују за одговарајући простор потребан за складиштење ових бафера. Како би на овај начин било потребно додати велики број ограничења (квадратног реда по броју променљивих), користи се ефикаснији приступ: свакој адреси p се додељује јединствен *фиксиран* број (или *магични број*) [176]. На овај начин се имплицитно моделује правило да се динамичким објектима додељују непреклапајућа места у меморији, а избегава се додавање великог броја ограничења у формулу. Да би ови бројеви били јединствени, они се додељују тек у оквиру формула које одговарају условима исправности (које су дефинисане у одељку 3.4). Овакво моделовање добро описује реално извршавање програма. Апроксимација се врши само након динамичког резервисања меморије за коју није позната величина резервисаног простора. Тада се наредна потребна адреса поставља на број који се у односу на претходну адресу разликује за некакву унапред дефинисану фиксирану вредност, што у специјалним ситуацијама може да доведе до преклапања простора потребних за складиштење бафера. Могућност преклапања стека и хипа се не моделује јер се претпоставља да до овакве грешке у раду програма неће доћи.

3.2.6 Глобалне променљиве

Глобалне променљиве су променљиве за које се простор резервише унутар сегмента података и којима се може приступати у оквиру свих функција програма (и, према томе, у оквиру сваког блока). За моделовање глобалних променљивих, тј. за праћење њихових вредности у оквиру система LAV, могу се користити складиште блокова и меморијски низ.

Складишта блокова — овакво моделовање се може користити у ситуацијама када се над глобалним променљивама не користи оператор референцирања и погодно је онда када постоји пуно измена глобалних променљивих. У овом случају, глобалне променљиве припадају складишту сваког блока у свакој функцији и потребно је пратити њихове вредности кроз блокове (о чему ће бити речи у секцији о интрапроцедуралној анализи 3.3.1), а потребно је и пратити њихове вредности приликом позива функција (о чему ће бити речи у секцији о интерпроцедуралној анализи 3.3.2).

Меморијски низ mem — овакво моделовање је опште моделовање које се може увек применити. Расуђивање које укључује теорију низова може да буде временски захтевно, али уколико нема много измена глобалних променљивих, овај модел може да буде подеснији јер се складишта блокова не оптерећују додавањем нових променљивих.

Пример 3.17 *Уколико се глобалне променљиве прате преко низа* mem*, тада се за глобалну променљиву* `int a;` *додела* `a=v;` *моделује формулом*

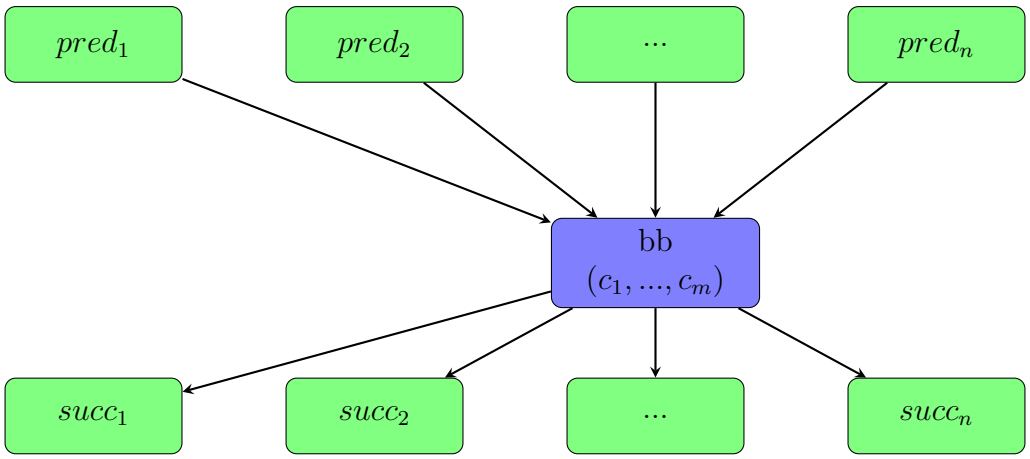
$$mem' = store(mem, addr\_a, v)$$

при чему важи: (i) mem' *је нова вредност за променљиву која означава меморију, а која је до тог тренутка имала вредност* mem*, (ii)* $addr\_a$ *је адреса променљиве* `a`*.*

3.3 Моделовање контроле тока

Блокови програма се представљају формулама $Transformation(b)$ логике првог реда које су конструисане коришћењем симболичког израчунавања (као што је описано у секцији 3.2.2). Везе између блокова описују контролу тока програма и описују се формулама исказне логике[3], што је описано у наставку одељка.

[3]Прецизније, уместо исказних променљивих користе се предикатски симболи арности нула, али ћемо једноставности ради њих у даљем тексту звати исказним променљивама.



Слика 3.3: *Блок са својим претходницима и следбеницима — излазни услови* $(c_1, c_2, \ldots, c_m)$ *одређују којим блоком ће се наставити извршавање програма након блока bb (ако је испуњен услов* c_i*, онда је следећи блок који ће се извршити блок* $succ_i$*).*

3.3.1 Интрапроцедурална контрола тока

У општем случају, блок може да буде доступан из више других блокова и, такође, може да води, у зависности од испуњења различитих услова, у више других блокова, као што је то приказано на слици 3.3. Блокови и прелази између њих чине усмерени граф контроле тока функције: блокови су чворови графа, а прелази су гране графа. Постоји тачно један улазни блок функције (блок којим почиње извршавање функције и који нема улазних грана) и један излазни блок функције (блок којим се завршава рад функције и који нема излазних грана).[4]

Претпоставимо, за почетак, да граф контроле тока функције у себи не садржи петље. Путања кроз овај граф је онда одређена низом чворова и грана (и нема циклуса). За сваки блок и за сваки прелаз уводи се по једна исказна променљива која је тачна уколико су одговарајући чвор и прелаз у путањи. Формула F, која обједињује описе свих блокова генерисане на овај начин, описује истовремено све могуће проласке кроз функцију, при чему ће сваки њен модел одговарати неком извршавању функције. Подразумева се да је током сваког извршавања функције у сваком тренутку активан неки блок, да се до њега дошло из неког блока (сем ако је он почетни) и да се из њега прешло у неки други (сем уколико је он завршни). Услови који морају бити задовољени у тренутку када неки блок постаје активан и услови који морају бити задовољени у тренутку када неки блок престаје бити активан описују се формулама које улазе у састав описа блока.

[4] Ово се може обезбедити трансформацијом кода.

Активни блокови и преласци између блокова

Исказне промењљиве које описују активне блокове и преласке између блокова обележавамо са $active_b$ и $transition^b_{b'}$. Оне имају наредно значење.

- $active_b$ означава да је чвор b у путањи, или, интуитивно, да је блок b достигнут (активан). Ова исказна променљива ће бити тачна у свим моделима формуле F који одговарају извршавањима програма у којима се долази до блока b.

- $transition^b_{b'}$ означава да је грана из чвора b до чвора b' у путањи или, интуитивно, да се у блок b' дошло из блока b. Ова исказна променљива ће бити тачна у свим моделима формуле F који одговарају извршавањима програма у којима се из блока b прелази у блок b'.

Улазни и излазни услови

Претпоставимо да је у програму који се анализира блок b директно достижан из блокова $pred_1$, $pred_2$, ..., $pred_n$ и да су блокови $succ_1$, $succ_2$, ..., $succ_m$[5] директно достижни из блока b (слика 3.3).

Могуће путање кроз граф контоле тока се описују коришћењем улазних услова за дати блок, који су представљени формулом $EntryCond(b)$ и коришћењем излазних услова који су представљени формулом $ExitCond(b)$. Улазни услови морају да важе у улазним тачкама блока b, а излазни услови морају да важе у излазним тачкама блока b. Наредне формуле прецизно описују ове услове.

EntryCond(b) се дефинише као:

$$activating(b) \wedge initialize(b)$$

Формула $activating(b)$ указује да постоји прелаз из неког претходника блока b до блока b ако и само ако је блок b био активан. Ова формула се дефинише на следећи начин:

$$\left(\bigvee_{pred \in \{pred_1, pred_2, \ldots, pred_n\}} transition^{pred}_{bb2} \right) \Leftrightarrow active_b$$

[5] Може се претпоставити да су сви $succ_i$ различити блокови и да су сви $pred_i$ различити блокови, јер се то може обезбедити трансформацијом програма.

Ако блок нема предходнике (на пример ако је у питању улазни блок функције), онда је $activating(b)$ дефинисана као $active_b$. На овај начин се функција анализира претпостављајући да она може да буде достижна из неке тачке у програму.

Ако је блок b достигнут из блока $pred$, тада су иницијалне вредности променљивих блока b вредности променљивих на излазу из блока $pred$. Ово се дефинише формулом $initialize(b)$ на следећи начин:

$$\bigwedge_{pred \in \{pred_1, ..., pred_n\}} \left(transition_{bb2}^{pred} \Rightarrow \bigwedge_{v \in V_f} e_{pred}(v) = s_b(v) \right)$$

Ако блок нема предходника, онда је $initialize(b)$ дефинисано као $\top$.

Приметимо да формула $activating(b)$ садржи дисјункцију могућих прелаза из претходника блока b у блок b, иако се, реалним извршавањем програма, у блок b може стићи само из тачно једног претходника, а не и из више њих истовремено (што дисјункција дозвољава). Коришћење ексклузивне дисјункције у овој формули дефинисало би прецизније односе између блокова, али би и значајно повећало сложеност формуле. Додатно, коришћење ексклузивне дисјункције није неопходно јер формула $initialize(b)$ регулише да не може да буде присутно више од једног прелаза из претходника блока b у блок b, осим у ретким ситуацијама када су све излазне вредности свих променљивих блокова претходника $pred_{k_1}, ..., pred_{k_m}$ једнаке. Дакле, само уколико су све излазне вредности свих променљивих блокова претходника $pred_{k_1}, ..., pred_{k_m}$ једнаке, формула допушта да је више од једног прелаза могуће, при чему то значи да су, заправо, улазне вредности за блок b исте из ма ког од блокова претходника $pred_{k_1}, ..., pred_{k_m}$ да се стигло у овај блок. Моделу формуле у којој је више од једног прелаза могуће, одговарало би неколико (у зависности од броја допуштених прелаза) различитих извршавања програма. Међутим, излазни услови (који се описују у наставку текста) обезбеђују да за сваки блок постоји само један следбеник што, уколико се анализа врши од једне полазне тачке, гарантује да модел описује само једну путању од полазне тачке до тачке која се анализира.

ExitCond(b) се дефинише као

$$jump(b) \wedge leaving(b)$$

Ако је блок b био активан и ако је услов изласка c_i блока b био задово-

љен, тада је контрола тока пребачена на блок $succ_i$, и обратно. Ово се дефинише формулом $jump(b)$ на следећи начин:

$$\bigwedge_{succ_i \in \{succ_1, succ_2, \ldots, succ_m\}} \left((active_b \wedge e_b(c_i)) \Leftrightarrow transition^b_{succ_i} \right)$$

Ако блок има само једног следбеника, онда се $jump(b)$ дефинише на следећи начин:

$$active_b \Leftrightarrow transition^b_{succ}$$

Ако блок нема следбеника, онда се $jump(b)$ дефинише као $\top$.

Формула $leaving(b)$ описује да је блок b био активан ако и само ако је водио до неког другог блока (или до излаза из функције). Овај услов се описује на следећи начин:

$$active_b \Leftrightarrow \bigvee_{succ \in \{succ_1, succ_2, \ldots, succ_m\}} transition^b_{succ}$$

Ако блок нема следбеника, тада се $leaving(b)$ дефинише као $\top$.

Приметимо да формула $leaving(b)$ садржи дисјункцију могућих прелаза из блока b у следбенике блока b иако се, реалним извршавањем програма, из блока b може стићи само до једног следбеника, а не и у више њих истовремено (што дисјункција дозвољава). Коришћење ексклузивне дисјункције у овој формули би, као и код формуле $activating(b)$, дефинисало прецизније односе између блокова, али би такође и значајно повећало сложеност формуле. И у овом случају, коришћење ексклузивне дисјункције није неопходно: формула $jump(b)$ регулише да је само један прелаз могућ тиме што за сваки блок увек може да буде испуњено само једно излазно правило.

Коначно, формула $Description(b)$ комплетно описује блок b и дефинише се као конјункција улазних услова, трансформације складишта и излазних услова блока:

$$EntryCond(b) \ \wedge \ Transformation(b) \ \wedge \ ExitCond(b)$$

Пример 3.18 *За пример са слике 3.2, симболичко извршавање блока* bb2 *приказано је у табели 3.3.*

Табела 3.3: *Вредности промењљивих складишта блока* `bb2` *са слике 3.2 након сваке наредбе.*

наредба	`a`	`b`	`c`	`r9`	`r10`	`r11`
bb2:	$s_{bb2}(a)$	$s_{bb2}(b)$	$s_{bb2}(c)$	—	—	—
`r9=b;`	$s_{bb2}(a)$	$s_{bb2}(b)$	$s_{bb2}(c)$	$s_{bb2}(b)$	—	—
`r10=a;`	$s_{bb2}(a)$	$s_{bb2}(b)$	$s_{bb2}(c)$	$s_{bb2}(b)$	$s_{bb2}(a)$	—
`r11=r9/r10;`	$s_{bb2}(a)$	$s_{bb2}(b)$	$s_{bb2}(c)$	$s_{bb2}(b)$	$s_{bb2}(a)$	$s_{bb2}(b)/$ $s_{bb2}(a)$
`printf("%d\n", r11);`	$s_{bb2}(a)$	$s_{bb2}(b)$	$s_{bb2}(c)$	$s_{bb2}(b)$	$s_{bb2}(a)$	$s_{bb2}(b)/$ $s_{bb2}(a)$

Трансформација за блок `bb2` *је задата формулом:*

$$\begin{aligned} Transformation(bb2) = (e_{bb2}(a) = s_{bb2}(a)) \\ \wedge (e_{bb2}(b) = s_{bb2}(b)) \\ \wedge (e_{bb2}(c) = s_{bb2}(c)) \end{aligned}$$

док су улазни и излазни услов задати са:

$$\begin{aligned} EntryCond(bb2) &= \left(\bigvee_{pred \in \{bb, bb1\}} transition_{bb2}^{pred} \right) \Leftrightarrow active_{bb2} \bigwedge \\ &\quad \bigwedge_{pred \in \{bb, bb1\}} \left(transition_{bb2}^{pred} \Rightarrow \bigwedge_{v \in \{a,b,c\}} e_{pred}(v) = s_{bb2}(v) \right) \\ ExitCond(bb2) &= active_{bb2} \Leftrightarrow transition_{return}^{bb2} \end{aligned}$$

Опис блока `bb2` *задаје се формулом:*

$$Description(bb2) = EntryCond(bb2) \ \wedge \ Transformation(bb2) \ \wedge \ ExitCond(bb2)$$

Петље

Претходно описан поступак моделовања тока извршавања може се применити на произвољан кôд који у себи не садржи петље. Моделовање петљи представља посебан изазов у верификацији. У оквиру статичке анализе, није могуће направити модел који је потпуно прецизан и заустављајући. Стога, моделовање петљи захтева некакву апроксимацију полазног кода.

У оквиру система LAV могуће је применити две врсте апроксимације полазног кода, у зависности од начина моделовања петљи:

- *сужавање опсега полазног кода* (енг. underapproximation) и
- *уопштење полазног кода* (енг. overapproximation).

У оба случаја, петље се елиминишу *размотавањем* (енг. unwinding, unrolling) али на различите начине. Апроксимирани кôд након примене размотавања у

```
for(i=0;i<k;i++)
  {...}
```

```
i=0;
if(i<k) {
  {...}     //prvi prolazak kroz petlju
  i++;
  if(i<k) {
    {...}   //drugi prolazak kroz petlju
    i++;
    if(i<k) {
      {...} //treci prolazak kroz petlju
      i++;
    }
  }
}
```

Слика 3.4: *Размотавањем петље, приказане са леве стране, три пута, добија се кôд приказан са десне стране.*

себи не садржи петље, па се може применити основни механизам генерисања формула које описују контролу тока програма.

У случају сужавања опсега полазног кода, петље се размотавају фиксиран број пута n, као у техникама проверавања ограничених модела (као што је то описано у секцији 2.1.2). Ако се у коду у којем су петље размотане на овај начин не пронађу грешке, то значи да и оригинални кôд нема грешака за n или мање пролазака кроз петљу. Међутим, постоји могућност да грешка у програму није откривена јер је за њено испољавање потребан већи број пролазака кроз петљу. С друге стране, ако се у коду пронађе грешка, пошто је за употребљени број размотавања моделовање прецизно, то значи да та грешка није лажно упозорење.

Пример 3.19 *Пример бројачке петље и њено размотавање фиксиран број пута дат је на слици 3.4.*

У случају уопштења полазног кода, размотани кôд симулара првих n и последњих m пролазака кроз петљу, при чему су n и m параметри који се могу конфигурисати. Да би се реализовало овакво размотавање петље, потребно је уметнути блок кода који симулира извршавање произвољне итерације петље. Произвољна итерација петље се симулира изменом свих вредности којима се приступа у оквиру петље (постоје и други приступи који користе сличне технике [17]). Измена ових вредности може да проузрокује губљење прецизности и да због тога уведе лажна упозорења.[6] Уколико се у коду, у којем су петље размотане на овај начин, не пронађу грешке, онда то значи да ни оригинални кôд нема грешака, односно, грешка не може да прође неопажено.

[6] Да би се избегао овај проблем, потребно је имати инваријанте петљи обележене у оквиру кода програма, или користити технике које аутоматски изводе инваријанте петљи у неким случајевима (као што је то описано у секцији 2.1.2).

Пример 3.20 *Пример бројачке петље и њено размотавање симулацијом првих n и последњих m пролазака кроз петљу, за n = 2 и m = 2 дати су на слици 3.5.*

```
for(i=0;i<k;i++)
  {...}
```

```
i=0;
if(i<k) {
  {...}     //prvi prolazak kroz petlju
  i++;
  if(i<k) {
    {...}   //drugi prolazak kroz petlju
    i++;

    priozovljan_prolazak();//simulacija proizvoljnog
                           //izvrsavanja instrukcija u
                           //petlji obuhvata:
                           //(i) izmenu svih promenljivih
                           //    kojima se pristupa u
                           //    okviru petlje
                           //(ii)izmenu promenljive koja
                           //    predstavlja memoriju
    if(i<k) {
      {...} //pretposlednji prolazak kroz petlju
      i++;
      if(i<k) {
        {...} //poslednji prolazak kroz petlju
        i++;
      }
    }
  }
}
```

Слика 3.5: *Размотавањем петље, приказане са леве стране, симулацијом првих n и последњих m пролазака кроз петљу, за n = 2 и m = 2, добија се кôд приказан са десне стране.*

3.3.2 Интерпроцедурална контрола тока

Интерпроцедурална контрола тока обухвата моделовање ефекта позива функције у зависности од расположивих информација о функцији. Могућа су три случаја:

- позната је дефиниција функције (нпр. кориснички дефинисана функција за коју је расположив изворни кôд);
- познат је уговор функције (није расположив изворни кôд али су расположиве извесне информације о функцији у форми уговора, нпр. функција стандардне библиотеке);
- за функцију нема расположивих информација (нпр. библиотечка функција за коју није расположив изворни кôд и нису расположиве информације у форми уговора).

У наредном тексту описује се моделовање ефекта позива функције за сваки од претходна три случаја.

Моделовање функција

Уколико је за функцију позната њена дефиниција (тј. у питању је кориснички дефинисана функција чији је кôд доступан у оквиру програма који се анализира) тада се дефиниција функције користи за конструисање формуле која описује рад функције.

Формула помоћу које се описује извршавање функције креира се коришћењем формула које описују блокове. Формуле које описују блокове су основне градивне јединице сложенијих формула. Формула која описује блок у себи садржи информације о трансформацији коју блок извршава, али и о повезаности блока са осталим блоковима у функцији. Због тога, конјункција свих формула које одговарају блоковима једне функције садржи информације о трансформацијама и о свим могућим путањама кроз функцију и представља компактан и прецизан опис функције.

Рекурзивни позиви функција се могу разматрати на сличан начин као и петље, размотавањем позива до одређене дубине или општијим апроксимирањем позива функције.

Конструисана формула којом се описује извршавање функције се користи за моделовање ефекта позива функције на сваком месту у програму где постоји наредба којом се ова функција позива. Позив функције се моделује коришћењем формуле описа функције која се умеће у скуп додатних ограничења за блок у којем је функција позвана. Да би се формула, која одговара опису функције, повезала са контекстом у којем је функција позвана, потребно је повезати *улазне* и *излазне параметре функције* са формулом која описује дату функцију. Под улазним параметрима функције подразумевамо вредности аргумената функције, вредност низа mem и вредности глобалних променљивих у улазном блоку функције. Повезивање са улазним параметрима се постиже додавањем низа једнакости у формулу $AdditionalConstraints$:

- вредности аргумената функције постављају се на конкретне вредности из контекста у којем је функција позвана;
- вредност низа mem за позвану функцију поставља се на вредност из контекста у којем је функција позвана;
- вредности глобалних променљивих које позвана функција користи постављају се на вредности глобалних променљивих из контекста у којем је

функција позвана (само уколико се глобалне променљиве прате преко складишта).

Под излазним параметрима функције подразумевамо повратну вредност функције (уколико постоји), вредност низа mem и вредност глобалних променљивих у излазном блоку функције. Повезивање са излазним параметрима функције се постиже изменама складишта блока у којем је наредба позива функције:

- повратна вредност функције користи се за ажурирање одговрајуће променљиве складишта;
- вредност меморије у излазном блоку функције користи се за ажурирање меморијске променљиве складишта;
- вредности глобалних променљивих у излазном блоку функције користе се за ажурирање вредности глобалних променљивих у складишту (само уколико се глобалне променљиве прате преко складишта).

Пример 3.21 *Опис следеће једноставне функције*

```
int kvadrat(int n) {
return n*n;
}
```

је формула

$$return\_value = n \cdot n,$$

при чему ћемо, у овом примеру, једноставности ради, занемарити садржај меморије и глобалних променљивих.

Уколико у коду постоји позив ове функције, на пример,

```
a=kvadrat(3);
```

тада се у скуп додатних ограничења додаје формула која описује ефекат позива функције `kvadrat`*. Под претпоставком да је ово први позив функције* `kvadrat`*, контекст у којем се функција позива може се означити бројем 1,*[7] *па се у скуп додатних ограничења блока додаје формула:*

$$return\_value_{context_1} = n_{context_1} \cdot n_{context_1}$$

[7] Додавање редног броја контекста је важно јер у оквиру једног блока може постојати више позива исте функције.

Такође, додаје се и формула која врши повезивање вредности аргумената функције са конкретним вредностима контекста у којем је функција позвана:

$$n_{context_1} = 3$$

Услед постојања доделе, вредност промењљиве `a` *у складишту се ажурира, тј. за нову вредност промењљиве* `a` *поставља се повратна вредност функције са редним бројем контекста у којем је функција позвана:* $return\_value_{context_1}$.

Уговори

Уговор функције садржи опис ефекта функције на промељиве у програму (постуслов функције) и опис услова који морају да буду испуњени да би функција могла да се безбедно изврши (предуслов функције). Уговори се могу користити за моделовање ефекта функција стандардне библиотеке или било које функције за коју изворни кôд није доступан.

Пример 3.22 *Уговор C функције стандардне библиотеке*

```
void * malloc ( size_t size );
```

може да се зада наредном формулом:

$$\begin{gathered} left(return\_value) = 0 \\ \wedge \\ ((return\_value = 0 \wedge right(return\_value) = 0) \vee \\ (return\_value \neq 0 \wedge right(return\_value) = size)) \end{gathered}$$

која описује постуслов функције, при чему $return\_value$ *означава повратну вредност фукнције* `malloc`*. Овако дефинисан уговор прецизно моделује понашање функције* `malloc`*: претходна формула покрива случај када је алокација меморије успела, али и случај када алокација меморије није успела и када је повратна вредност функције 0. Како се функција* `malloc` *увек може безбедно позвати, то се предуслов ове функције може дефинисати са* $\top$.

Ако је познат уговор функције, онда се вредности променљивих у складишту ажурирају и додају се додатна ограничења у формулу $AdditionalConstraints$ у складу са уговором функције.

Пример 3.23 *Уколико у програму постоји наредба:*

```
p = (int*)malloc(sizeof(int));
```

тада се у складиште за променљиву `p` *поставља њена нова вредност* p' *која се описује додавањем додатних ограничења у формулу* $AdditionalConstraints$ *генерисаних на основу уговора функције:*

$$left(p') = 0$$
$$\wedge$$
$$((p' = 0 \wedge right(p') = 0) \vee (p' \neq 0 \wedge right(p') = sizeof(int)))$$

Непознате функције

Када се користи позив функције из неке библиотеке чији изворни кôд није доступан (те није могућа интерпроцедурална анализа) и за коју не постоји унапред дефинисан уговор у оквиру система LAV (те није могућа симулација позива функције њеним уговором), тада за анализу програма ефекат функције није познат и потребно га је моделовати на начин који осликава позив произвољне функције.

У реалном окружењу, позивом произвољне функције не могу се изменити локалне вредности променљивих функције позиваоца, осим уз помоћ показивача или преко повратне вредности функције. Уколико се то преслика на моделовање које користи систем LAV, то значи да вредности које чува складиште програма не могу бити измењене, сем преко повратне вредности функције, док се вредности које се прате преко меморијске променљиве могу изменити на непознати начин. Зато се позив функције у овом случају моделује комплетном променом садржаја меморије, односно тренутни низ mem се поставља на новоуведену променљиву за коју се не додају никаква ограничења, чиме новоуведена променљива симулира стање меморије о којој не постоје никакве информације. Такође, повратна вредност функције се моделује као новоуведена променљива о којој не постоје никакве информације. Ова новоуведена променљива се уписује као нова вредност у складишту за одговарајућу променљиву. Оваквим моделовањем се губи на прецизности, што може да буде узрок лажних упозорења, али се обезбеђује да не може бити пропуштених грешака.

Пример 3.24 *Уколико у програму постоји позив непознате функције, односно функције за коју није познат ни њен уговор, нити постоји дефиниција функције, нпр.* `p=unknown(a,b);` *тада је ефекат позива ове функције измена вредности променљиве* `p` *новом вредношћу* p' *као и измена вредности променљиве које означава меморију новом вредношћу* mem'.

3.4 Услови исправности

Да би се проверило да ли нека наредба може да доведе до грешке, граде се две формуле: прва одговара услову исправности наредбе, а друга одговара услову неисправности наредбе. Тиме се испитивање да ли може доћи до грешке своди се на испитивање ваљаности генерисаних формула. Ваљаност генерисаних формула проверава се у оквиру изабране теорије. Избор могућих теорија и утицај избора теорије на ефикасност и прецизност резултата анализе описан је у одељку 3.5.

3.4.1 Генерисање формула исправности

Услову исправности одговара формула која је универзално квантификована (при чему се квантификатори подразумевају и нећемо их писати)

$$K \Rightarrow safe(c)$$

док услову неисправности одговара наредна формула (која је такође универзално квантификована)

$$K \Rightarrow \neg safe(c)$$

У овим формулама, K зовемо *контекст* у којем се услов (не)исправности разматра, а $safe(c)$ зовемо *услов безбедности* наредбе.

Контекст

Контекст дефинише информације из околине наредбе које ће бити узете у разматрање приликом расуђивања о наредби. Постоје различити могући контексти.

- *Празан контекст* — у овом случају испитује се сама наредба и K има вредност $\top$.
- *Контекст блока* — у овом случају испитује се наредба укључујући информације о вредностима променљивих које су доступне у оквиру блока којем она припада; K тада има вредност:

  $$Transformation(b, i-1)$$

  при чему је наредба која се испитује i-та наредба у блоку b.

- *Контекст функције* — у овом случају испитује се наредба укључујући све информације о вредностима променљивих које су доступне у оквиру функције у којој се наредба налази; K тада има вредност:

$$\bigwedge_{b' \in F} Description(b') \bigwedge EntryCond(b) \bigwedge Transformation(b, i-1)$$

при чему је наредба која се испитује i-та наредба у блоку b, а скуп F садржи све блокове b' који у датој функцији претходе блоку b.

- *Контекст у којем је функција позвана* — у овом случају контекст укључује информације контекста функције, али и шире информације, односно информације о контексту у којем је дата функција позвана. На пример, уколико је функција f позвана као i-та наредба блока $b_{f'}$ функције f', и ако се испитује услов исправности за наредбу c за коју је контекст функције задат формулом K_f, контекст K тада има вредност:

$$\bigwedge_{b'_{f'} \in F'} Description(b'_{f'}) \bigwedge EntryCond(b_{f'}) \bigwedge Transformation(b_{f'}, i-1) \bigwedge K_f$$

при чему F' садржи све блокове $b'_{f'}$ који у функцији f' претходе блоку $b_{f'}$.

Овај контекст није увек применљив — није применљив за наредбе које се налазе у функцији од које почиње извршавање програма. С друге стране, уколико имамо већи број позива функција унутар других функција, на овај начин контекст се може и даље проширивати.

Испитивање исправности наредбе увек се започиње са испитивањем у празном контексту, а шири[8] контексти се укључују у зависности од добијених резултата, што ће бити детаљније описано у секцији 3.4.2.

Услов безбедности наредбе

Услов безбедности наредбе, $safe(c)$, одређен је наредбом за коју се (не)исправност испитује. За њено дефинисање може се користити једно од следећег:

- дефиниција грешке — дељење нулом, прекорачење бафера, дереференцирање *null* показивача и слично;

[8] Шири контекст у односу на K је сваки контекст K' који укључује информације контекста K, али и неке додатне информације. На пример, контекст блока је шири од празног контекста, док је контекст функције шири и од контекста блока и од празног контекста.

- логички израз у оквиру `assert` наредбе;
- предуслов функције (за позив функције за коју је познат уговор).

Пример 3.25 *За наредбу* `x/y`*, формула* $safe(x/y)$ *може да се дефинише са* $y \neq 0$.

Пример 3.26 *За наредбу* `a[i] = x`*, формула* $safe(a[i] = x)$ *може да се дефинише са* $(left(a) \leq i) \wedge (i < right(a))$.

Пример 3.27 *За позив функције* `assert(a!=b)`*, формула услова безбедности,* $safe(assert(a! = b))$*, може да се дефинише са* $a \neq b$.

Пример 3.28 *Уколико је за функцију* `int f(int n)` *познат предуслов, на пример* $n > 0$*, тада за позив функције* `f(a)` *формула* $safe(f(a))$ *може да се дефинише са* $a > 0$.

Пример 3.29 *Предуслов C функције стандардне библиотеке*

```
char * strcpy ( char * destination, const char * source );
```

не може се једноставно и прецизно описати коришћењем постојећег моделовања. Наиме, да би позив ове функције био безбедан, потребно је да је искоришћеност бафера `source` *мања од алоцираног простора за бафер* `destination`*, при чему је за искоришћеност бафера потребно пратити стање меморије и позицију терминирајуће нуле. Међутим, уместо прецизног моделовања, предуслов функције може се апроксимирати формулом:*

$$\begin{aligned} right(destination) \geq 0 \quad & \wedge \\ right(source) \geq 0 \quad & \wedge \\ right(destination) \geq right(source) & \end{aligned}$$

Јасно је да претходна формула може да уведе лажна упозорења у ситуацији када је алоциран простор за изворну ниску већи него за одредишну ниску, али је искоришћени простор изворне ниске мањи од алоцираног простора одредишне ниске. Такође, претходна формула може да доведе и до пропуштених грешака у ситуацији када су њени услови испуњени, али када је изворна ниска иницијализована тако да не садржи терминирајућу нулу у оквиру алоцираног простора. Па ипак, и овако једноставно моделовања може да буде корисно за проналажење грешака у програму.

Према томе, за позив функције `strcpy(dest, src)` *формула*

$$safe(strcpy(dest, src))$$

може да се дефинише са

$$right(dest) \geq 0 \quad \wedge \quad right(src) \geq 0 \quad \wedge \quad right(dest) \geq right(src).$$

3.4.2 Статус наредбе

Сваки услов исправности разматра се у неком контексту. За фиксиран контекст, наредба може да има статус:

- *безбедна* (енг. safe) — наредба приликом извршавања програма никада неће довести до грешке;
- *неисправна* (енг. flawed) — наредба приликом извршавања програма увек ће довести до грешке;
- *небезбедна* (енг. unsafe) — наредба приликом извршавања програма може, али и не мора да доведе до грешке (у зависности од путање и вредности улазних променљивих);
- *недостижна* (енг. unreachable) — наредба неће никада бити извршена.

LAV придружује статус наредби у зависности од ваљаности (у оквиру изабране теорије) формула исправности и неисправности за разматрани контекст. Прецизније:

- уколико је формула услова исправности ваљана, а формула услова неисправности није ваљана, тада се наредби придружује статус безбедне наредбе у контексту за који је разматрана;
- уколико је формула услова неисправности ваљана, а формула услова исправности није ваљана, тада се наредби придружује статус неисправне наредбе у контексту за који је разматрана;
- уколико формуле услова исправности и услова несправности нису ваљане, тада се наредби придружује статус небезбедне наредбе у контексту за који је разматрана;
- уколико су формуле услова исправности и несправности ваљане, тада је контекст у којем су услови разматрани неконзистентан и наредби се придружује статус недостижне наредбе у контексту за који је разматрана.

У даљем тексту ће се, ради краткоће, под тврдњом да је нека наредба исправна, мислити на то да LAV у разматраном контексту тој наредби придружује статус исправне наредбе, и слично за остале статусе наредби.

Приметимо да чак и ако је нека наредба безбедна, неисправна или небезбедна у неком контексту, то и даље не значи да је та наредба достижна у ширем контексту, јер додавање нових информација може да уведе неконзистентност у контекст. Следи однос закључака донетих за један контекст са одговарајућим ширим контекстима.

- Уколико се за неку наредбу покаже да је безбедна у неком контексту K, тада је та наредба у ширем контексту и даље безбедна, или је недостижна. На основу дефиниције безбедне наредбе, формула услова исправности $K \Rightarrow safe(c)$ је ваљана, а формула услова неисправности $K \Rightarrow \neg safe(c)$ то није. Како је формула $K \Rightarrow safe(c)$ ваљана формула, онда је ваљана и формула $(K_0 \wedge K) \Rightarrow safe(c)$ (при чему је K_0 формула која проширује дати контекст K), што је последица својства монотоности класичне логике првог реда. С друге стране, како формула $K \Rightarrow \neg safe(c)$ није ваљана, то значи да проширивањем контекста K формула $(K_0 \wedge K) \Rightarrow \neg safe(c)$ може и не мора да буде ваљана формула, што зависи од формуле K_0. На основу дефиниција статуса наредби, следи да је у ширем контексту наредба безбедна (уколико формула $(K_0 \wedge K) \Rightarrow \neg safe(c)$ није ваљана) или недостижна (уколико је формула $(K_0 \wedge K) \Rightarrow \neg safe(c)$ ваљана).

- Уколико се за неку наредбу покаже да је неисправна у неком контексту K, тада је та наредба у ширем контексту и даље неисправна, или је недостижна. На основу дефиниције неисправне наредбе, формула услова исправности $K \Rightarrow safe(c)$ није ваљана формула, а формула услова неисправности $K \Rightarrow \neg safe(c)$ је ваљана формула. Како формула $K \Rightarrow safe(c)$ није ваљана, то значи да проширивањем контекста K формулом K_0, формула $(K_0 \wedge K) \Rightarrow safe(c)$ може и не мора да буде ваљана формула, што зависи од саме формуле K_0. С друге стране, како је формула $K \Rightarrow \neg safe(c)$ ваљана формула, онда је ваљана и формула $(K_0 \wedge K) \Rightarrow \neg safe(c)$, што је последица својства монотоности класичне логике првог реда. Према томе, на основу дефиниција статуса наредби, следи да је у ширем контексту наредба неисправна (уколико формула $(K_0 \wedge K) \Rightarrow safe(c)$ није ваљана) или недостижна (уколико је формула $(K_0 \wedge K) \Rightarrow safe(c)$ ваљана).

- Уколико се за неку наредбу покаже да је недостижна у неком контексту K, тада је та наредба у произвољном ширем контексту такође недости-

жна. На основу дефиниције недостижне наредбе, и формула услова исправности $K \Rightarrow safe(c)$ и формула услова неисправности $K \Rightarrow \neg safe(c)$ су ваљане формуле. Проширивањем контекста K формулом K_0, формуле $(K_0 \wedge K) \Rightarrow safe(c)$ и $(K_0 \wedge K) \Rightarrow \neg safe(c)$ су такође ваљане формуле уоквиру изабране теорије (што је последица својства монотоности логике првог реда), па је, на основу дефиниције недостижне наредбе, ова наредба недостижна и у ширем контексту.

- За наредбу за коју се покаже да је у неком контексту K небезбедна, додавање ширег контекста (односно додавање нових информација) може да промени статус наредбе. На основу дефиниције небезбедне наредбе, обе формуле, тј. формула услова исправности $K \Rightarrow safe(c)$ и формула услова неисправности $K \Rightarrow \neg safe(c)$, нису ваљане формуле. Проширивањем контекста K формулом K_0, формула $(K_0 \wedge K) \Rightarrow safe(c)$ може и не мора да буде ваљана формула, а исто важи и за формулу $(K_0 \wedge K) \Rightarrow \neg safe(c)$. Уколико проширивањем контекста обе ове формуле нису ваљане, статус наредбе у проширеном контексту остаје исти. Уколико је само формула услова исправности проширеног контекста ваљана, тада је статус наредбе у проширеном контексту безбедан. Уколико је само формула услова неисправности проширеног контекста ваљана, тада је статус наредбе у проширеном контексту неисправан. Уколико су обе формуле ваљане, тада је статус наредбе у проширеном контексту недостижан.

Закључивање на основу најширег могућег контекста је најпрецизније, али обухвата конструисање формуле са највећим бројем информација чија је провера ваљаности и најскупља. Такође, уколико је за наредбу `c` функције `f` потребно проверити услов исправности, при чему је функција `f` позвана n пута, тада је, укључивањем најширег контекста, потребно бар n провера за ову наредбу. Ако се за наредбу већ на основу празног контекста, контекста блока или контекста функције може закључити да је безбедна, значајно се добија на ефикасности јер се тиме елиминише потреба за n скупих и непотребних провера.

Да би се број скупих и непотребних провера што више смањио, за сваку наредбу најпре се разматра празан контекст, а шири контексти се разматрају по потреби. Статуси се одређују на следећи начин:

- уколико се за наредбу покаже да је безбедна или недостижна у оквиру неког контекста, онда је тај статус коначан (тј. не испитује се статус наредбе у ширем контексту) јер, у оба случаја, наредба не представља претњу исправности програма;

- уколико се за наредбу покаже да је неисправна, онда се проверава њена достижност у оквиру контекста функције (достижност се проверава ради искључивања могућности пријављивања лажних упозорења) и уколико је наредба достижна, статус неисправности је коначан, уколико је наредба недостижна, онда је статус недостижности коначан;

- уколико се за наредбу покаже да је небезбедна у оквиру неког контекста, тада се та наредба даље разматра у оквиру ширег контекста (уколико такав контекст постоји).

Овакав приступ може да обезбеди ефикаснију анализу и мањи број скупих провера, али може да се деси да има за последицу и низ неуспешних провера (провера у којима се не може одредити коначни статус) све до крајње провере која укључује и најшири могући контекст.

Пример 3.30 *Нека је дат наредни кôд:*

```
1.  a = b/3;
2.  c = 5;
3.  a = b/c;
4.  if(a>0)
5.    a = b/c;
6.  a = b/d;
```

Разматрајмо формуле исправности које се генеришу за претходни кôд у теорији бит-векторске аритметике. Да би се проверио услов исправности за наредбу у линији број 1, довољан је празан контекст, јер је формула $\top \Rightarrow 3 \neq 0$ *ваљана у теорији бит-векторске аритметике, док то не важи за формулу* $\top \Rightarrow \neg(3 \neq 0)$ *која одговара услову неисправности наредбе.*

За проверу услова исправности наредбе у линији број 3, није довољан празан контекст јер формула $\top \Rightarrow c \neq 0$ *није ваљана, а исто важи и за формулу* $\top \Rightarrow \neg(c \neq 0)$ *која одговара услову неисправности наредбе. Према томе, потребно је разматрати шири контекст. Уколико се користи контекст блока у којем се наредба налази, тада услов исправности укључује и вредност промењиве* `c`

$$(a = b/3 \wedge c = 5) \Rightarrow c \neq 0$$

што је ваљана формула, па уз разматрање одговарајуће формуле неисправности наредбе може да се закључи да је формула у овом контексту безбедна.

За утврђивање безбедности наредбе у линији број 5, нису довољни ни празан контекст ни контекст блока јер је вредност промењиве `c` *у оба ова контекста непозната. Уколико се у формули користи контекст функције, тада се може утврдити да је и ова наредба безбедна.*

За наредбу у линији број 6, није могуће доказати безбедност на нивоу контекста функције у којој се наредба налази, под претпоставком да су `a`, `b` *и* `c` *локалне промењиве, а да је* `d` *аргумент функције* f *у којој се овај кôд налази. Безбедност ове наредбе неопходно је разматрати у ширем контексту, односно у контексту позива функције* f*.*

Пример 3.31 *За кôд приказан на слици 3.2 услов безбедности наредбе у линији број 9 одговара услову безбедности за наредбу* `r11=r9/r10` *поједностављеног LLVM кôда:*

$$safe(r11 = r9/r10) = (r10 \neq 0)$$

Формула која одговара услову исправности ове наредбе, у контексту функције у којој се налази, гради се на основу формула које описују блокове и које су дате на слици 3.6 и формуле $Transformation(bb2, 3)$*:*

$$\begin{aligned} Transformation(bb2,3) \;=\; & (e_{bb2}(a) = s_{bb2}(a)) \wedge (e_{bb2}(b) = s_{bb2}(b)) \\ & \wedge (e_{bb2}(c) = s_{bb2}(c)) \wedge (e_{bb2}(r9) = s_{bb2}(b)) \\ & \wedge (e_{bb2}(r10) = s_{bb2}(a)) \end{aligned}$$

На основу тога, формула која одговара услову исправности наредбе је (у контексту функције у којој се наредба налази):

$$\begin{aligned} & (Description(entry) \bigwedge Description(bb) \bigwedge Description(bb1) \\ & \bigwedge EntryCond(bb2) \bigwedge Transformation(bb2,3)) \Rightarrow (e_{bb2}(r10) \neq 0). \end{aligned}$$

Расписивањем вредности за формуле $Description(entry)$*,* $Description(bb)$*,* $Description(bb1)$*,* $EntryCond(bb2)$ *и* $Transformation(bb2, 3)$ *добија се формула приказана на слици 3.7. Ова формула није ваљана у оквиру теорије бит-вектора, што значи да је, у зависности од ваљаности формуле која одговара услову неисправности, ова наредба небезбедна или неисправна.*

На основу овог једноставног примера може се уочити да се у оквиру формула појављује велики број једнакости облика $e_b(v) = s_{b'}(v)$*. Међутим, присуство ових једнакости у формули није увек неопходно. На пример, уколико блок има само једног претходника, или уколико се може закључити да је вред-*

$$
\begin{aligned}
EntryCond(entry) &= active_{entry} \\
Transformation(entry) &= (e_{entry}(a) = s_{entry}(a)) \ \wedge \ (e_{entry}(b) = b_1) \ \wedge \ (e_{entry}(c) = c_1) \\
ExitCond(entry) &= \left((acitve_{entry} \wedge (b_1 > c_1)) \Leftrightarrow transition_{bb}^{entry}\right) \\
&\quad \bigwedge \left((acitve_{entry} \wedge \neg(b_1 > c_1)) \Leftrightarrow transition_{bb1}^{entry}\right) \\
Description(entry) &= EntryCond(entry) \ \wedge \ Transformation(entry) \\
&\quad \wedge \ ExitCond(entry)
\end{aligned}
$$

$$
\begin{aligned}
EntryCond(bb) &= (transition_{bb}^{entry} \Leftrightarrow active_{bb}) \wedge \\
&\quad (transition_{bb}^{entry} \Rightarrow \\
&\quad ((e_{entry}(a) = s_{bb}(a)) \ \wedge \ (e_{entry}(b) = s_{bb}(b)) \\
&\quad \wedge (e_{entry}(c) = s_{bb}(c)))) \\
Transformation(bb) &= (e_{bb}(a) = s_{bb}(a) + 1) \wedge (e_{bb}(b) = s_{bb}(b)) \wedge (e_{bb}(c) = s_{bb}(c)) \\
ExitCond(bb) &= active_{bb} \Leftrightarrow transition_{bb2}^{bb} \\
Description(bb) &= EntryCond(bb) \ \wedge \ Transformation(bb) \ \wedge \ ExitCond(bb)
\end{aligned}
$$

$$
\begin{aligned}
EntryCond(bb1) &= (transition_{bb1}^{entry} \Leftrightarrow active_{bb1}) \wedge \\
&\quad (transition_{bb1}^{entry} \Rightarrow \\
&\quad ((e_{entry}(a) = s_{bb1}(a)) \ \wedge \ (e_{entry}(b) = s_{bb1}(b)) \\
&\quad \wedge (e_{entry}(c) = s_{bb1}(c)))) \\
Transformation(bb1) &= (e_{bb1}(a) = s_{bb1}(a) - 1) \wedge (e_{bb1}(b) = s_{bb1}(b)) \wedge (e_{bb1}(c) = s_{bb1}(c)) \\
ExitCond(bb1) &= active_{bb1} \Leftrightarrow transition_{bb2}^{bb1} \\
Description(bb1) &= EntryCond(bb1) \ \wedge \ Transformation(bb1) \ \wedge \ ExitCond(bb1)
\end{aligned}
$$

$$
\begin{aligned}
EntryCond(bb2) &= \left(\bigvee_{pred \in \{bb, bb1\}} transition_{bb2}^{pred}\right) \Leftrightarrow active_{bb2} \bigwedge \\
&\quad \bigwedge_{pred \in \{bb, bb1\}} \left(transition_{bb2}^{pred} \Rightarrow \bigwedge_{v \in \{a,b,c\}} e_{pred}(v) = s_{bb2}(v)\right) \\
Transformation(bb2) &= (e_{bb2}(a) = s_{bb2}(a)) \wedge (e_{bb2}(b) = s_{bb2}(b)) \wedge (e_{bb2}(c) = s_{bb2}(c)) \\
ExitCond(bb2) &= active_{bb2} \Leftrightarrow transition_{return}^{bb2} \\
Description(bb2) &= EntryCond(bb2) \ \wedge \ Transformation(bb2) \ \wedge \ ExitCond(bb2)
\end{aligned}
$$

$$
\begin{aligned}
EntryCond(return) &= (transition_{return}^{bb2} \Leftrightarrow active_{return}) \wedge \\
&\quad (transition_{return}^{bb2} \Rightarrow \\
&\quad ((e_{bb2}(a) = s_{return}(a)) \ \wedge \ (e_{bb2}(b) = s_{return}(b)) \\
&\quad \wedge (e_{bb2}(c) = s_{return}(c)))) \\
Transformation(return) &= (e_{return}(a) = s_{return}(a)) \wedge (e_{return}(b) = s_{return}(b)) \\
&\quad \wedge (e_{return}(c) = s_{return}(c)) \\
ExitCond(return) &= \top \\
Description(return) &= EntryCond(return) \ \wedge \ Transformation(return) \\
&\quad \wedge \ ExitCond(return)
\end{aligned}
$$

Слика 3.6: *Описи блокова за кôд приказан на слици 3.2.*

$$
\begin{aligned}
& (active_{entry} \\
\bigwedge\ & (e_{entry}(a) = s_{entry}(a)) \ \wedge \ (e_{entry}(b) = b_1) \ \wedge \ (e_{entry}(c) = c_1) \\
\bigwedge\ & \Big((acitve_{entry} \wedge (b_1 > c_1)) \Leftrightarrow transition_{bb}^{entry}\Big) \\
& \wedge \Big((acitve_{entry} \wedge \neg(b_1 > c_1)) \Leftrightarrow transition_{bb1}^{entry}\Big) \\
\bigwedge\ & (transition_{bb}^{entry} \Leftrightarrow active_{bb}) \wedge \\
& (transition_{bb}^{entry} \Rightarrow \\
& ((e_{entry}(a) = s_{bb}(a)) \ \wedge \ (e_{entry}(b) = s_{bb}(b)) \wedge \ (e_{entry}(c) = s_{bb}(c)))) \\
\bigwedge\ & (e_{bb}(a) = s_{bb}(a) + 1) \wedge (e_{bb}(b) = s_{bb}(b)) \wedge (e_{bb}(c) = s_{bb}(c)) \\
\bigwedge\ & active_{bb} \Leftrightarrow transition_{bb2}^{bb} \\
\bigwedge\ & (transition_{bb1}^{entry} \Leftrightarrow active_{bb1}) \wedge \\
& (transition_{bb1}^{entry} \Rightarrow \\
& ((e_{entry}(a) = s_{bb1}(a)) \ \wedge \ (e_{entry}(b) = s_{bb1}(b)) \wedge \ (e_{entry}(c) = s_{bb1}(c)))) \\
\bigwedge\ & (e_{bb1}(a) = s_{bb1}(a) - 1) \wedge (e_{bb1}(b) = s_{bb1}(b)) \wedge (e_{bb1}(c) = s_{bb1}(c)) \\
\bigwedge\ & active_{bb1} \Leftrightarrow transition_{bb2}^{bb1} \\
\bigwedge\ & \left(\bigvee_{pred \in \{bb, bb1\}} transition_{bb2}^{pred}\right) \Leftrightarrow active_{bb2} \bigwedge \\
& \bigwedge_{pred \in \{bb, bb1\}} \left(transition_{bb2}^{pred} \Rightarrow \bigwedge_{v \in \{a,b,c\}} e_{pred}(v) = s_{bb2}(v)\right) \\
\bigwedge\ & (e_{bb2}(a) = s_{bb2}(a)) \wedge (e_{bb2}(b) = s_{bb2}(b)) \wedge (e_{bb2}(c) = s_{bb2}(c)) \\
& \wedge (e_{bb2}(r9) = s_{bb2}(b)) \wedge (e_{bb2}(r10) = s_{bb2}(a))) \\
\Rightarrow\ & e_{bb2}(r10) \neq 0
\end{aligned}
$$

Слика 3.7: *Формула која представља услов исправности наредбе у линији број 9 кода приказаног на слици 3.2.*

ност неке променљиве једнака без обзира на блок претходник из којег се дошло, тада се уместо додавања једнакости може извршити иницијализација почетне вредности променљиве блока крајњом вредношћу променљиве блока претходника. Тиме се значајно смањује формула и поправља ефикасност. Ова оптимизација детаљно је описана и разматрано у оквиру секције 4.1.2.

3.5 Теорије за моделовање програма

Универзално квантификована формула вишесортне логике првог реда, која моделује програм, обично укључује аритметику, логичке и релационе операторе, али и функције као што су $left$, $right$, $select$ и $store$. Генерисану формулу потребно је трансформисати у формулу на језику изабране теорије (или комбинације теорија) како би се омогућило коришћење решавача за дату теорију (комбинацију теорија). Избор теорија може да буде вођен прецизношћу жељених резултата или ефикасношћу расуђивања. Трансформација формуле

у формулу на језику изабраних теорија, у општем случају, не мора верно да сачува својства полазне формуле. У овом одељку биће описане комбинације теорија које се могу користити за моделовање програма као и последице њиховог коришћења по ефикасност и прецизност анализе.

3.5.1 Променљиве и операције над њима

Целобројне вредности и операције над њима се могу моделовати бројевима коначне прецизности (енг. finite-precision numbers) и одговарајућим операцијама коришћењем аритметике бит-вектора или се могу моделовати бројевима произвољне прецизности (енг. arbitrary-precision numbers) и одговарајућим операцијама коришћењем целобројне или реалне линеарне аритметике. Моделовање коришћењем аритметике бит-вектора у потпуности верно осликава аритметику која се користи у реалним извршавањима програма, али расуђивање у оквиру теорије бит-вектора може бити временски захтевно. С друге стране, расуђивање у оквиру теорија реалне или целобројне линеарне аритметике је значајно ефикасније, али не осликава верно аритметику која се користи у реалним извршавањима програма. Наиме, ове теорије имају подршку само за сабирање, одузимање, множење константном вредношћу и релационе операторе, док је за моделовање других оператора који се користе у програму (на пример, за моделовање множења, дељења и битских оператора) потребно користити додатне технике које уносе непрецизности. Такође, коришћење бројева произвољне прецизности не одговара реалним извршавањима програма, па је чак и за подржане операторе и познате вредности које учествују у аритметичком изразу могуће добити неодговарајући резултат (на пример, број који је превелик да би могао да стане у одговарајући тип података предвиђен за дату променљиву). Расуђивање у оквиру реалне линеарне аритметике је ефикасније од расуђивања у оквриру целобројне линеарне аритметике, али за моделовање целобројних вредности уводи додатне непрецизности допуштањем модела који имају реалне уместо целобројне вредности. Уколико у коду не постоји информација о томе да ли је нека променљива означеног или неозначеног типа (као што је то случај са кодом који генерише LLVM), онда се информација о томе не може пренети формули у линерној аритметици, што представља још један извор непрецизности. Претходно описане непрецизности, које се јављају услед моделовања теоријом која не осликава верно аритметику реалних извршавања програма, могу да доведу до појаве лажних упозорења и пропуштених грешака.

Вредности у покретном зарезу и операције над њима могу се моделовати апроксимацијом одговарајућим целобројним вредностима и коришћењем цело-

бројне аритметике бит-вектора или коришћењем бројева произвољне прецизности и одговарајућим операцијама у оквиру реалне линеарне аритметике. Оба ова моделовања уводе непрецизности које могу да доведу до лажних упозорења или до непронађених грешака, слично као код моделовања целобројних вредности теоријом линеарне аритметике. Моделовање вредности у покретном зарезу и операција над њима бројевима коначне прецизности и одговарајућим операцијама коришћењем аритметике бит-вектора за бројеве у покретном зарезу, једино је потпуно прецизно моделовање бројева у покретном зарезу. Међутим, расуђивање у оквиру аритметике бит-вектора за бројеве у покретном зарезу је веома захтевно и још увек не постоје ефикасни имплементирани решавачи за ову теорију.

3.5.2 Функције *left* и *right*

За функције $left$ и $right$ може се користити теорија неинтерпретираних функција при чему се све релевантне инстанце њихових специфичних особина ($left(p+n) = left(p)-n$ и $right(p+n) = right(p)-n$) додатно наводе у условима исправности.

Уместо теорије неинтерпретираних функција може се користити и *акерманизација*[9] [1]. Акерманизацијом се, на основу формуле ϕ, формира нова формула ϕ' која је еквизадовољива са формулом ϕ, а која не садржи симболе неинтерпретираних функција. Елиминација симбола неинтерпретираних фунцкија из формуле ϕ, врши се увођењем нових променљивих тако што се свака примена функције $f(x_1, ..., x_n)$ замењује новом променљивом $v_{f(x_1,...,x_n)}$. Додатно, за сваки пар различитих примена $f(x_1, ..., x_n)$ и $f(y_1, ..., y_n)$ исте функције, у формулу се додају ограничења у чијем формирању учествује фунцкија ack. Ова функција пресликава сваку примену функције $g(z_1, ..., z_m)$ у одговарајућу променљиву $v_{g(z_1,...,z_m)}$, сваку променљиву у саму себе, а хомоморфна је у односу на интерпретиране симболе. Ограничења имају следећи облик

$$(\bigwedge_{i=1}^{n} ack(x_i) = ack(y_i)) \Rightarrow v_{f(x_1,...,x_n)} = v_{f(y_1,...,y_n)}$$

при чему се атом $ack(x_i) = ack(y_i)$ не додаје уколико су обе стране једнакости синтаксно идентичне.

[9] Термин акерманизација изведен је из имена математичара Вилхелма Акермана (енг. Wilhelm Ackermann).

Пример 3.32 *Акерманизацијом формуле*

$$right(a) + 3 = n \wedge right(b) - 2 = m \wedge left(a) + 3 = 0 \wedge left(b) = 0$$

добија се формула

$$(r_a + 3 = n) \ \wedge \ (r_b - 2 = m) \ \wedge \ (l_a + 3 = 0) \ \wedge \ (l_b = 0)$$
$$\wedge \ (a = b \Rightarrow r_a = r_b) \ \wedge \ (a = b \Rightarrow l_a = l_b)$$

Оба ова приступа (неинтерпретиране функције и акерманизација) једнако прецизно моделују програм, али ефикасност расуђивања може да буде различита, у зависности од конкретних формула [37].

3.5.3 Функције *select* и *store*

Функције *select* и *store*, могу природно да се моделују теоријом низова. Моделовање садржаја меморије коришћењем теорије низова може да буде временски захтевно. Једна алтернатива моделовању теоријом низова може да буде и игнорисање садржаја меморије. Сваки пут када је потребно прочитати садржај меморије са неке локације, уместо повратне вредности функције *select*, може се користити новоуведена променљива (променљива за коју заправо немамо никакве информације о њеном садржају), док се упис садржаја у меморију може у потпуности игнорисати. На тај начин се заправо губе све информације о вредностима променљивих које се не прате преко складишта и тиме формула губи на прецизности. То може да доведе до временски ефикаснијег расуђивања, али и до лажних упозорења.

3.5.4 Могуће комбинације теорија

На основу претходних разматрања, модел програма у оквиру система LAV обухвата:

- аритметику бит-вектора или линеарну аритметику;
- теорију неинтерпретираних функција или акерманизацију;
- теорију низова (опционо).

Постоје разноврсни системи који обезбеђују подршку за испитивање ваљаности формула ових теорија и њихових комбинација. Неки од њих су описани у секцији 2.2.2.

3.6 Генерисање извештаја

Уколико формула која одговара услову исправности за неку наредбу није ваљана, тада је та наредба небезбедна или неисправна и постоји контрамодел у којем услов исправности није тачан. Уколико систем за проверавање ваљаности за формуле које нису ваљане може да генерише такав контрамодел, онда се он може искористити за реконструисање вредности промењљивих и путање кроз програм које воде до грешке. Ове информације корисне су за лакше разумевање узрока пронађене грешке у програму.

За реконструкцију путање, за коју услов исправности није тачан, користе се исказне промењљиве $active_b$ и $transition_{b_j}^{b_i}$. Сви блокови b за које је $active_b$ тачно у контрамоделу, припадају траженој путањи, као и сви прелази између блокова b_i и b_j за које је $transition_{b_j}^{b_i}$ тачно у контрамоделу. Како анализа програма почиње од једне почетне тачке (од улазног блока функције која се анализира), то контрамодел који би укључио више различитих путања кроз програм није могућ, што је дискутовано у секцији 3.3.2.

За разумевање узрока пронађене грешке у програму, посебно су важне вредности промењљивих у тренутку извршавања наредбе за коју се испитује услов исправности, као и вредности промењљивих блока од којег почиње извршавање за задати контекст. Вредности промењљивих на изласку из сваког блока, који припада путањи, могу послужити као додатне информације за разумевање и праћење тока извршавања програма. Све ове вредности могу се једноставно генерисати, на основу контрамодела, коришћењем излазних вредности промењљивих за сваки активни блок.

$$
\begin{aligned}
&active_{entry} = \top,\\
&active_{bb} = \top,\\
&active_{bb1} = \bot,\\
&active_{bb2} = \top,\\
&transition_{bb}^{entry} = \top,\\
&transition_{bb1}^{entry} = \bot,\\
&transition_{bb2}^{bb} = \top,\\
&transition_{bb2}^{bb1} = \bot,\\
&e_{entry}(a) = s_{entry}(a) = s_{bb}(a) = s_{bb1}(a) = -1,\\
&e_{entry}(b) = b_1 = s_{bb}(b) = e_{bb}(b) = s_{bb1}(b) = e_{bb1}(b) = s_{bb2}(b) = e_{bb2}(b) = e_{bb2}(r9) = 5,\\
&e_{entry}(c) = c_1 = s_{bb}(c) = e_{bb}(c) = s_{bb1}(c) = e_{bb1}(c) = s_{bb2}(c) = e_{bb2}(c) = 3,\\
&e_{bb}(a) = s_{bb2}(a) = e_{bb2}(a) = e_{bb2}(r10) = 0,\\
&e_{bb1}(a) = -2
\end{aligned}
$$

Слика 3.8: *Један контрамодел за коју формула са слике 3.7 није тачна.*

Пример 3.33 *Формула која одговара услову исправности из примера 3.31 није ваљана у теорији бит-векторске аритметике, што значи да у наредби линије 9 програма са слике 3.2 може да дође до дељења нулом. Један контрамодел за формулу која одговара услову исправности приказан је на слици 3.8. На основу овог контрамодела можемо да генеришемо извештај приказан на слици 3.9.*

Извештај се генерише коришћењем наредних запажања. Блокови који се извршавају су блокови `entry`, `bb` *и* `bb2`, *јер су вредности исказник променљивих* $active_{entry}$, $active_{bb}$ *и* $active_{bb2}$ *у котнрамоделу* $\top$. *Прелаз из блока* `entry` *у блок* `bb` *одређен је тиме што је вредност* $transition_{bb}^{entry}$ *у контрамоделу једнака* $\top$, *а прелаз из блока* `bb` *у блок* `bb2` *одређен је тиме што је вредност* $transition_{bb2}^{bb}$ *у контрамоделу једнака* $\top$. *Вредности променљивих на крају почетног блока су вредности променљивих* $e_{entry}(a)$, $e_{entry}(b)$ *и* $e_{entry}(c)$ *које су у контрамоделу, редом, једнаке -1, 5 и 3. Вредности променљивих на крају блока bb су вредности променљивих* $e_{bb}(a)$, $e_{bb}(b)$ *и* $e_{bb}(c)$ *које су у контрамоделу, редом, једнаке 0, 5 и 3. Вредности променљивих у тренутку извршавања наредбе су вредности променљивих* $e_{bb2}(a)$, $e_{bb2}(b)$ *и* $e_{bb2}(c)$ *које су у контрамоделу, редом, једнаке 0, 5 и 3.*

Грешка дељење нулом може да настане за наредно извршавање програма:

1. Извршавају се блокови entry, bb и bb2, при чему се из блока entry прелази у блок bb, а из блока bb прелази у блок bb2.
2. Вредности променљивих на крају почетног блока су $a = -1$, $b = 5$ и $c = 3$.
3. Вредности променљивих на крају блока *bb* су $a = 0$, $b = 5$ и $c = 3$.
4. Вредности променљивих у тренутку извршавања наредбе су $a = 0$, $b = 5$ и $c = 3$.

Слика 3.9: *Извештај генерисан на основу једног контрамоделу формуле која одговара услову исправности из примера 3.31.*

Везе између инструкција у блоковима и оригиналних наредби могу се искористити за генерисање извештаја у којем се користе бројеви линија оригиналног кода уместо имена блокова међукода.

Пример 3.34 *Како су познате наредбе оригиналног кода од којих су настали блокови трансформисаног кода, то се уместо извештаја са слике 3.9 може ге-*

нерисати извештај, приказан на слици 3.10, који садржи информације у терминима оригиналног кода.

Грешка дељење нулом може да настане у линији 9, за извршавање програма у којем:

1. Након извршавања линије 5, вредности променљивих су $a = -1$, $b = 5$ и $c = 3$.
2. Након извршавања линије 6, вредности променљивих су $a = 0$, $b = 5$ и $c = 3$.
3. У линији 9, вредности променљивих су $a = 0$, $b = 5$ и $c = 3$.

Слика 3.10: *Извештај генерисан на основу једног контрамодела формуле која одговара услову исправности из примера 3.31, формулисан у терминима оригиналног кода.*

Извештај који садржи линије оригиналног кода може се употпунити навођењем LLVM инструкције која доводи до грешке. Ово може да буде корисно у ситуацијама када је у оквиру једне линије кода задато више наредби или ако, на пример, у оквиру једног сложеног израза није очигледно који део израза води до грешке.

3.7 Својства генерисаних формула

Комбиновање формула које одговарају описима блокова води на интуитиван начин до формула које осликавају различита извршавања програма. Међутим, ове формуле не осликавају у потпуности верно сва могућа извршавања програма. Може се десити да нека могућа извршавања програма нису описана формулом, или се може десити да су нека немогућа извршавања програма допуштена формулом. Разлог за ово су апроксимације које су увођене приликом прављења модела програма или приликом трансформације модела програма у формулу на језику изабране теорије.

Елиминација петљи и рекурзивних позива — Трансформацијом програма у програм без петљи новодобијени кôд је уопштење полазног кода или сужење опсега полазног кода, у зависности од апроксимације која се примењује приликом трансформације. Ове апроксимације су детаљно описане у секцији 3.3.1. На сличан начин, програм се трансформише у програм без рекурзивних позива, као што је описано у секцији 3.3.2.

Позиви функција — Позиви функција за које нису познате дефиниције и нису расположиви уговори, уводе непрецизност у праћењу садржаја меморије. Ова апроксимација је описана у секцији 3.3.2.

Апроксимације неодговарајућом теоријом — Елементарни типови података и елементарне операције могу бити моделоване неодговарајућом теоријом (на пример, ради добијања на ефикасности, линеарном аритметиком), или се праћење вредности променљивих преко меморијског низа може игнорисати (такође зарад добијања на ефикасности). Ове апроксимације су описане у одељку 3.5.

Наведена ограничења, или макар нека од њих, су заједничка за све приступе верификацији софтвера. Наиме, питање исправности програма укључује питање заустављања па, из чињенице да је халтинг проблем неодлучив, произилази да не постоји општи начин за проверавање да ли је нека наредба програма достижна, па тиме ни да ли је исправна. Ипак, и поред наведених ограничења, може се доћи до практично употребљивог верификацијског алата који може да открије нетривијалне грешке или да утврди њихово одсуство у многим ситуацијама.

У случају система LAV, генерисана формула верно описује израчунавања у програму и вредности променљивих у случајевима када није потребно применити наведене апроксимације. Прецизније, уколико се LAV користи над програмом који:

1. не садржи променљиве реалног типа,
2. не садржи динамичку алокацију меморије,
3. не садржи петље (или, аналогно, програмом у којем су све петље размотане одговарајући максимални број пута),
4. не садржи рекурзивне позиве функција (или, аналогно, програмом у којем су сви рекурзивни позиви размотани одговарајући максимални број пута),
5. не садржи позив функције за коју није позната дефиниција,

и уколико се за моделовање програма користе теорија бит-вектора, теорија неинтерпретираних функција и теорија низова, и уколико се разматра само најшири могући контекст, тада су извештаји које LAV генерише несумњиви: грешке које LAV пријављује сигурно јесу грешке, а наредбе за које LAV не пријављује да могу да доведу до грешке сигурно то и не могу (у смислу грешака

које је LAV у могућности да пронађе). Ово је последица прецизног моделовања могућих путања кроз програм који не садржи петље (исказним променљивама *transition* и *active*), прецизног моделовања резултата рада инструкција симболичким извршавањем, прецизног генерисања услова исправности програма (услови обухватају најшири могући контекст) и прецизног трансформирања услова исправности програма у одговарајуће теорије. У овом тексту неће бити строго формулисано нити доказивано претходно својство. Исправност моделовања које LAV користи биће потврђена само емпиријски, експерименталним путем.

У случају да је примењена нека од наведених апроксимација, могуће су неоткривене грешке и/или лажна упозорења, као што је то већ дискутовано на одговарајућим местима у претходном тексту. Као и сви други верификацијски приступи, и овај приступ је дизајниран са циљем да се смањи број неоткривених грешака и број лажних упозорења. То је и постигнуто у одређеној мери, што ће показати експериментални резултати као и резултати поређења са другим савременим системима (глава 4).

4

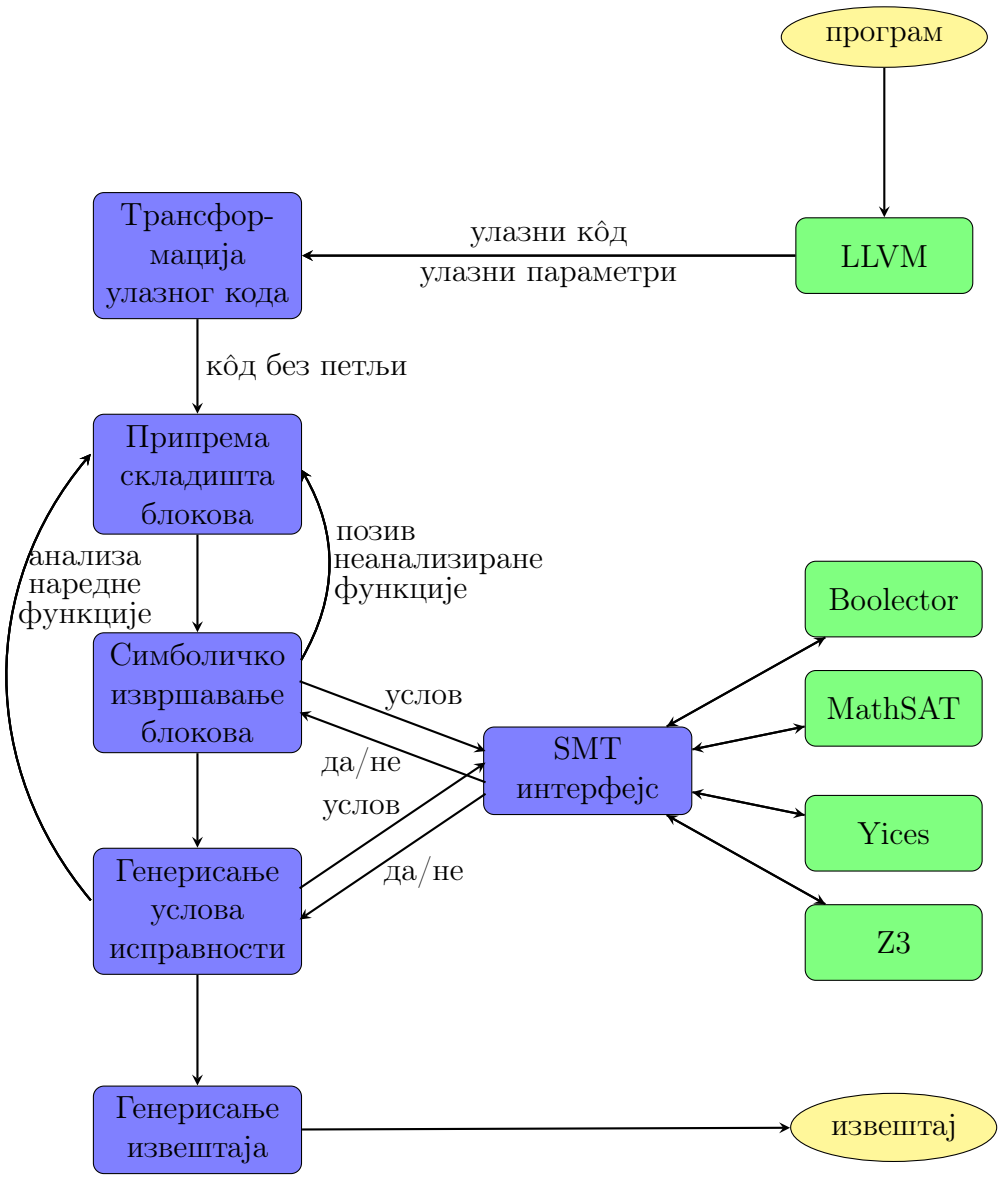
Имплементација и евалуација система LAV

Систем LAV је имплементиран у програмском језику C++, отвореног је кода и јавно је доступан са адресе: http://argo.matf.bg.ac.rs/?content=lav. Кôд се дистрибуира под лиценцом NCSA отвореног кода Универзитета Илиноис [107]. Имплементација, без интерфејса ка решавачима и помоћних класа за рад са дељеним изразима, садржи око 10 000 линија кода. У наставку текста описана је укратко имплементација система и приказани су експериментални резултати поређења система LAV са сродним алатима.

4.1 Конкретизација општег алгоритма

Имплементација алата прати општи опис система LAV (глава 3). Глобални алгоритам имплементације илустрован је на слици 4.1, док је детаљнији алгоритам имплементације приказан на слици 4.2. Основне фазе алгоритма су фаза трансформације изворног кода, фаза анализе кода и комуникација са решавачем. Као улазни кôд, LAV очекује датотеку добијену превођењем изворног улазног програма вишег програмског језика у LLVM међујезик. LAV трансформише овај кôд и врши анализу над њим користећи функције из LLVM интерфејса за програмирање (енг. application programming interface, API). Током анализе кода, LAV генерише услове исправности које проверава путем интерфејса за програмирање одабраног спољашњег SMT решавача.

У наредном тексту описане су специфичности имплементације (тј. конкретизација општег описа из главе 3), као и најважније класе и функције које чине имплементацију система.



Слика 4.1: *Имплементација система LAV.*

4.1.1 Трансформација кода

LAV је развијан првенствено за анализу програма добијених трансформацијом из програма написаних у програмском језику C, али, захваљујући универзалности LLVM система, може да се примени и за анализу програма добијених трансформацијом из других процедуралних програмских језика (на пример, програмског језика Fortran). Анализа програма добијених трансформацијом из програма написаних на објектно оријентисаним програмским језицима захтева извесне додатке тренутној имплеметацији система.

Превођење изворног кода у кôд LLVM међујезика, коришћењем одговарајуће приступне компоненте, није интегрисано у сам систем, већ је ово превођење потребно урадити независно. Превођење у LLVM међукод може се извршити у режиму за проналажење грешака (енг. debug mode) и у режиму за извршавање програма (енг. release mode). Превођење у LLVM међукод у режиму за проналажење грешака омогућава успостављање везе између пронађених грешака LLVM међукода и изворног кода. Захваљујући томе, извештај који LAV производи садржи детаљне информације у терминима бројева линија и наредби изворног кода. У случају да је превођење у LLVM међукод остварено у режиму за извршавање програма, тада извештај који LAV генерише садржи информације само у терминима инструкција LLVM међукода.

Пре анализе кода врши се трансформација LLVM међукода тако да се добије LLVM међукод који не садржи петље (моделовање петљи описано је у секцији 3.3.1). Избор врсте и параметара трансформације петљи задају се приликом покретања алата (употреба система описана је у одељку A.2). Да би се добио кôд који не садржи петље, пре размотавања петљи, за које је одговорна класа LAV алата `LLoopUnroll`[1], врши се трансформација петљи у њихову канонску форму, за шта се користе основне трансформације LLVM система (користе се трансформације `LoopSimplifyPass`, `LoopRotatePass` и `LCSSAPass`). Након размотавања петљи, елиминишу се и неки делови LLVM кода који нису релевантни за даљу верификацијску анализу. За ову трансформацију одговорна је класа `LCFGSimplifyPass`.

4.1.2 Анализа кода

У првој фази анализе кода врши се одређивање сталних променљивих складишта одабране функције улазног програма[2], као и њихова иницијали-

[1] Префикс `L` у именима класа означава да је класа из система LAV.

[2] Ова функција се задаје као аргумент приликом покретања алата или, уколико се функција не зада, подразумева се да је то *main* функција.

зација вредностима над којима нису задата ограничења. Складиште (које је описано у секцији 3.2.2) представљено је класом `LBlockStore` која омогућава приступ, иницијализацију, измену и читање симболичких или конкретних вредности складишта. Промењљиве су унутар складишта описане класом `LVariableInfo`, која за сваку промењљиву памти релевантне неоходне податке. Вредности промењљивих у оквиру класе `LVariableInfo` описују се изразима класе `Expression`[3]. Над изразима је дефинисан низ функција које врше упрошћавања и израчунавања вредности симболичких израза, којем је централна функција `LSimplifyExpression` (нека имплементирана упрошћавања израза описана су у наставку текста). Класа `LVariableInfo` обезбеђује да су вредности промењљивих у складишту увек упрошћени изрази.

Симболичко извршавање започиње извршавањем улазног блока функције, након чега се симболички извршавају други блокови функције пратећи граф контроле тока функције. Основна класа за рад са блоковима је класа `LBlock`. Задатак ове класе је:

- конструисање формуле која представља компактан опис блока (на начин иписан у секцији 3.3.1) и
- формирање услова безбедности за наредбе које припадају блоку (услови безедности дефинисани су у секцији 3.4.1).

За конструисање описа блока, ова класа користи класу `LState`, као и информације о блоковима претходницима и блоковима следбеницима (услов преласка у нови блок и одговарајући блок следбеник описују се посебном класом `LJump`). Класа `LState` је одговорна за симболичко извршавање инструкција блока, односно, ова класа:

- дефинише ефекте појединачних инструкција на складиште блока,
- чува информције о додатним ограничењима над промењљивама складишта (која су описана класом `LConstraints`),
- формира услове безбедности за оне инструкције за које је то потребно
- генерише формулу која описује трансформацију промењљивих складишта извршеним инструкцијама блока.

Класа `LBlock` за појединачне инструкције, које припадају блоку, користи класу `LInstruction`, која за приступ информацијама везаним за изворни

[3]Класе за рад са дељеним изразима делом су преузете од Филипа Марића из система ArgoLib и прилагођене су за потребе система LAV.

кôд користи класе `InstructionInfo`, `InstructionInfoTable` и `InstructionToLineAnnotator`[4].

Услови безбедности блока описују се класом `LLocalCondition` која, осим о самом услову безбедности, води рачуна и о статусу услова, инструкцији којој услов одговара, као и о врсти грешке до које ће доћи уколико одговарајући услов исправности није испуњен. За генерисане услове безбедности класа `LBlock` врши позиве решавача за празан контекст и контекст блока.

Приликом симболичког извршавања блокова врши се више оптимизација које доприносе генерисању формула мање сложености и које омогућавају прецизније закључивање коришћењем ужег контекста.

- Пре почетка симболичког извршавања блока, проверавају се излазне вредности складишта блокова претходника. Уколико блок има само једног претходника, онда се излазне вредности променљивих складишта блока претходника постављају као почетне вредности променљивих текућег блока. Уколико блок има више претходника, онда се проверава да ли постоје променљиве које у сваком блоку претходнику имају исту излазну вредност. За такве променљиве, излазна вредност у блоковима претходницима уписује се као почетна вредност променљиве текућег блока. Овим се елиминише потреба за додавањем једнакости за вредности тих променљивих у формули $initalize(b)$, а постављене вредности променљивих на уласку у блок могу да омогуће и да се услови безбедности наредби могу закључити коришћењем празног или блоковског контекста (тј. без коришћења ширег контекста).

- Уколико блок има више од једног следбеника, проверавају се услови изласка из блока. Ови услови се проверавају позивом решавача коришћењем информација које су доступне у блоку. Уколико се за неки услов закључи да не може бити испуњен, онда се ажурира списак могућих блокова следбеника текућег блока, а за блок следбеник, који се овом приликом искључује, ажурира се списак могућих блокова претходника. На овај начин се смањује резултујућа формула која описује извршавање функције смањивањем броја могућих путања кроз програм. Такође, на овај начин може се закључити да су неки блокови недостижни и тиме се смањити број блокова који се симболички извршавају и за чије се инструкције проверавају услови исправности. Уколико на овај начин настану узастопни блокови

[4]Ове три класе су делом преузете из кода алата KLEE и прилагођене за потребе система LAV.

који имају само једног претходника и само једног следбеника, онда се њихово извршавање врши коришћењем истог складишта (за шта је одговорна класа `LMerge`).

- Приликом симболичког извршавања врше се упрошћавања израза како би резултујуће формуле биле мање сложености. На пример, изрази чије су вредности константе се одмах израчунавају (на пример, 5+3 се замењује са 8), логичке конјункције у којима учествује константа $\bot$ и логичке дисјункције у којима учествује константа $\top$ се замењују константама $\bot$ и $\top$, негације релацијских оператора замењују се одговарајућим релацијским операторима (на пример, $\neg(a > b)$ се замењује са $a \leq b$), асоцијативност и комутативност оператора + се примењује за груписање константних вредности и израчунавање вредности подизраза (на пример, израз $(a + 1) + 1$ се замењује са $a + 2$, а израз $(a + 2) + (b - 1)$ са $(a + b) + 1$).

- Приликом провера исправности наредби у оквиру блоковског контекста, додају се и информације о статички алоцираном простору за локалне променљивие функције и глобалне променљиве програма. Додавање ових информација може да буде довољно за прецизно одређивање статуса исправности наредбе у многим ситуацијама. На пример, за утврђивање исправности приступа у оквиру низа за који је простор алоциран статички наредбом `int a[10];` потребно је користити само информацију о статички алоцираном простору за овај низ. Ова информација се, у LLVM коду, налази у почетном блоку функције. Издвајањем и додавањем информација о статички алоцираном простору се избегава скупа провера услова исправности која укључује целокупни контекст функције у ситуацијама када су потребне само информације о статички алоцираном простору за показиваче.

- Користи се комбиновани приступ за праћење стања меморије. Теорија низова је за проверавање ваљаности временски захтевна, па се уместо два основна начина праћења стања меморије (која су описана у секцији 3.2.5), тренутно стање меморије прати комбинацијом која користи добре стране оба приступа. Први приступ користи се приликом уписа садржаја меморије на познату локацију (локацију која се задаје константном вредношћу, на пример, за наредбу `a[i]=j;` при чему је вредност променљиве `i` позната и једнака некој константи), а други приступ користи се приликом уписа садржаја меморије на непознату локацију (локацију која се задаје променљивом вредношћу, на пример, за наредбу `a[i]=j;` при чему

вредност променљиве `i` није позната у тренутку симболичког извршавања инструкције).

Редослед симболичког извршавања блокова прати граф контроле тока функције. Уколико је у блоку присутна инструкција позива функције `f`, за коју постоји дефиниција у програму, али за коју до тог тренутка није извршена анализа исправности, онда се симболичко извршавање текућег блока привремено прекида и започиње се анализа ове функције. Тек након комплетне анализе функције `f`, наставља се симболичко извршавање текућег блока. Овакав ток анализе је могућ само уколико у оквиру програма нема посредно или непосредно рекурзивних функција. Рекурзивни позиви функција могу се елиминисати у фази трансформације кода (што је описано у секцији 3.3.2), али ова трансформација у текућој имплементацији није подржана.

Завршетак симболичког извршавања свих блокова функције омогућава проверу услова исправности у контексту функције (што је описано у секцији 3.4.1) и конструисање формуле која представља опис функције (на начин описан у секцији 3.3.2). Ове задатке обавља класа `LFunction`. Анализа програма, односно анализа различитих функција пратећи граф контроле тока програма врши се у оквиру класа `LModule`.

4.1.3 Комуникација са решавачима

Ваљаност услова исправности наредбе и ваљаност услова изласка из блока проверава се SMT решавачем (који су описани у секцији 2.2.2). Провера ваљаности формуле (у оквиру изабране теорије) SMT решавачем врши се испитивањем задовољивости негиране формуле: уколико је негација формуле незадовољива, формула је ваљана; уколико је негација формуле задовољива, формула није ваљана. Избор теорије у којој се моделује програм (одељак 3.5) и избор одговарајућег конкретног решавача за проверавање услова исправности, врши се приликом покретања система LAV (покретање и улазни параметри система описани су у одељку A.2).

Позиви решавача врше се коришћењем класе `LSolver`. Ова класа обезбеђује и припрему формуле превођењем у формулу одговарајуће теорије, а по потреби комуницира са класом `LAckermannization` која омогућава ефикасну акермanизацију формуле (која је описана у секцији 3.5.2). Такође, ова класа је одговорна и за ефикасно коришћење инкременталног приступа позива решавача (инкрементални приступ описан је у наставку текста). Приступ конкретном решавачу класа `LSolver` остварује путем интерфејса који апстрахује приступ подржаним

1. Трансформисати LLVM кôд у кôд без петљи.
2. Елиминисати делове LLVM кода који нису релевантни за анализу.
3. Одредити глобалне промењиве чије се вредности прате коришћењем складишта блокова.
4. Одабрати функцију од које почиње анализа.
5. Припремити складишта блокова за ову функцију:
   (5.1) одредити локалне промењиве које припадају складиштима блокова;
   (5.2) додати складиштима блокова одговарајуће глобалне промењиве;
   (5.3) додати складиштима блокова промењиву *mem* која служи за праћење стања меморије;
   (5.4) извршити иницијализацију промењивих за свако складиште.
6. Изабрати улазни блок функције.
7. Симболички извршити изабрани блок:
   (7.1) пре почетка симболичког извршавања блока, проверити излазне вредности промењивих блокова претходника и по потреби ажурирати иницијализоване вредности;
   (7.2) симболички извршавати редом инструкције блока:
      (7.2.1) уколико је инструкција позив функције за коју постоји дефиниција али за коју још увек није извршена анализа, вратити се на корак 5, тј. започети анализу ове функције;
      (7.2.2) у зависности од наредбе, формирати одговарајући услов безбедности и проверити га у празном контексту и контексту блока (у зависности од параметра покретања система, детектовање неисправне наредбе може да води директно на корак 12);
      (7.2.3) формирати одговарајући симболички израз и приликом формирања извршити његово упрошћавање;
   (7.3) уколико блок има више од једног следбеника, проверити услове изласка из блока и по потреби ажурирати списак блокова следбеника односно списак њихових могућих претходника.
8. На основу графа контроле тока функције, изабрати блок функције који није симболички извршен и вратити се на корак 7. Уколико су сви блокови функције симболички извршени, прећи на корак 9.
9. За сваки блок за који постоји наредба за коју није утврђен коначан статус исправности:
   (9.1) за сваки блок претходник направити формулу која одговара опису блока (уколико таква формула није већ направљена);
   (9.2) послати решавачу оне описе блокова претходника који нису били раније послати решавачу;
   (9.3) коришћењем инкременталног приступа проверавати исправност наредби (у зависности од параметра покретања система, детектовање неисправне наредбе може да води директно на корак 12).
10. Конструисати ефекат позива фунцкије.
11. У зависности од параметра покретања система, изабрати функцију која није анализирана и вратити се на корак 5 или прећи на корак 12.
12. Генерисати текстуални и по потреби HTML извештај.

Слика 4.2: *Алгоритам који се користи у имплементацији система LAV.*

појединачним SMT решавачима[5]. Приступ конкретном решавачу остварује се кроз његов интерфејс за програмирање. LAV има подршку за рад са наредним SMT решавачима и теоријама:

- Boolector — за теорију бит-вектора и теорију низова;
- Mathsat — за теорију линеране аритметике, теорију бит-вектора и теорију неинтерпретираних функција;
- Yices — за теорију линеране аритметике, теорију бит-вектора и теорију неинтерпретираних функција;
- Z3 — за теорију линеране аритметике, теорију бит-вектора, теорију неинтерпретираних функција и теорију низова.

Време потребно решавачу да утврди задовољивост формуле зависи од комплексности формуле и начина коришћења решавача. Ово време значајно утиче на ефикасност целог система. Због тога се врше наредне оптимизације у коришћењу решавача.

- Оптимизује се број провера које решавач извршава — најпре се утврђује задовољивост негираног услова исправности наредбе. Уколико је ова формула незадовољива, то значи да је услов исправности наредбе ваљана формула па је према правилима утврђивања статуса наредбе (која су описана у секцији 3.4.2) наредба безбедна или недостижна. У оба случаја, та наредба није критична за безбедност кода па даље провере нису неопходне. Уколико је негиран услов исправности наредбе задовољива формула то значи да је наредба небезбедна или неисправна. У зависности од прецизности анализе која се изводи (прецизност анализе се задаје приликом покретања система), LAV проверава и услов неисправности наредбе чиме се прецизно утврђује да ли је наредба небезбедна или неисправна.
- Користи се инкременталан приступ — када се испитују различити услови (не)исправности у оквиру једне функције, формуле које одговарају непотребним, али већ разматраним блоковима, могу се задржати у оквиру формуле која се испитује захваљујући својству дедуктивне монотоности (енг. deductive monotonicity). Ово омогућава инкрементални приступ погодан за коришћење SMT решавача који су обично у могућности да искористе резултате које су научили из претходних покушаја решавања [19].

[5]Хијерархија класа за коришћење различитих решавача преузета је од Филипа Марића из система UrsaMajor и прилагођена за потребе система. Такође, додата је комплетна подршка за коришћење решавача Z3.

4.1.4 Тестови

Паралелно са развојем самог алата, развијен је и скуп тестова који проверавају различите функционалне и нефункционалне аспекте извршавања система. Тестирање је аутоматизовано и подељено у скупове тестова различите прецизности и временске захтевности. Ови тестови су саставни део дистрибуције система LAV.

4.2 Евалуација система LAV

У наставку текста приказани су резултати поређења система LAV са сродним софтверским алатима. Симболичко извршавање и проверавање ограничених модела су технике које LAV користи па је експериментално поређење извршено са алатима за класично симболичко извршавање (алат KLEE[6]) и са алатима за проверавање ограничених модела (алати CMBC и ESBMC).

Табела 4.1: *Технике које алат користи (ПОМ — проверавање ограничених модела, СИ — симболичко извршавање), приступне компоненте које алат користи за трансформацију улазног кода, подржане теорије и решавачи које алат користи.*

Алат	LAV	CBMC	ESBMC	LLBMC	KLEE	Calysto	PEX
Технике	ПОМ СИ	ПОМ -	ПОМ -	ПОМ -	- СИ	- СИ	- СИ
Трансформација улазног кода	LLVM	goto-cc	goto-cc	LLVM	LLVM	LLVM	.NET
Теорије	- LA BV EUF ARRAYS	исказна логика - BV EUF ARRAYS	- LA BV EUF ARRAYS	- - BV - ARRAYS	- - BV - ARRAYS	- - BV - -	- LA BV EUF ARRAYS
Решавачи	- Yices - Boolector Z3 MathSAT	MiniSAT2 - - Boolector Z3 MathSAT	- CVC - Boolector Z3 -	- - STP Boolector Z3 -	- - STP - - -	- Spear - - - -	- - - - Z3 -

[6]Само један алат заснован на симболичком извршавању је укључен у поређење јер алат PEX може да анализира само језике .NET платформе (и не може да анализира програме написане у програмском језику C), док алат Calysto није јавно доступан.

У табели 4.1 упоредно су приказани алати који се заснивају на техникама проверавања ограничених модела и на техникама симболичког извршавања, приступне компоненте која ови алати користе за трансформацију улазног кода, теорије које користе за моделовање и решавачи за које имају подршку. На основу табеле можемо да приметимо да су најпопуларнији решавачи Boolector и Z3 са теоријама бит-вектора и низова.

4.2.1 Однос са симболичким извршавањем

Симболичко извршавање програма (које је описано у секцији 2.1.2) обухвата анализу могућих путања кроз програм појединачно. Проналажење грешке симболичким извршавањем зависи од путање у којој се грешка испољава. На пример, симболичко извршавање кода који је приказан на слици 4.3 обухвата анализу четири могуће путање кроз програм.

```
0:  int main()
1:  {
2:  int a, b, k, div = 1;
...
3:  if(a>0)
4:      a = 1;
5:  else
6:       a = -1;
7:  if(b>0)
8:      b = 1;
9:  else
10:     b = -1;
11: div = a + b + 2;
12: k = 1/div;
13: return 0;
14: }
```

Слика 4.3: *Пример C програма са четири различите путање.*

- Прва путања:
  - услов $a > 0$ важи у линији број 3, након чега се извршава наредба `a=1;` у линији број 4,
  - услов $b > 0$ важи у линији број 7, након чега се извршава наредба `b=1;` у линији број 8,
  - променљива div добија вредност $a + b + 2$ у линији број 11, а променљива k добија вредност $1/div$ у линији број 12.

  Дељење у линији број 12 за ову путању је безбедно јер се, на основу информација генерисаних приликом симболичког извршавања ове путање, може закључити да је вредност променљиве div, приликом извршавања ове наредбе, једнака 4.

- Друга путања:
  - услов $a < 0$ важи у линији број 3, након чега се извршава наредба `a=-1;` у линији број 6,
  - услов $b > 0$ важи у линији број 7, након чега се извршава наредба `b=1;` у линији број 8,
  - променљива div добија вредност $a + b + 2$ у линији број 11, а променљива k добија вредност $1/div$ у линији број 12.

  Дељење у линији број 12 за ову путању је безбедно јер се, на основу информација генерисаних приликом симболичког извршавања ове путање, може закључити да је вредност променљиве div, приликом извршавања ове наредбе, једнака 2.

- Трећа путања:
  - услов $a > 0$ важи у линији број 3, након чега се извршава наредба `a=1;` у линији број 4,
  - услов $b < 0$ важи у линији број 7, након чега се извршава наредба `b=-1;` у линији број 10,
  - променљива div добија вредност $a + b + 2$ у линији број 11, а променљива k добија вредност $1/div$ у линији број 12.

  Дељење у линији број 12 за ову путању је безбедно јер се, на основу информација генерисаних приликом симболичког извршавања ове путање,

може закључити да је вредност променљиве *div*, приликом извршавања ове наредбе, једнака 2.

- Четврта путања:
  - услов $a < 0$ важи у линији број 3, након чега се извршава наредба `a=-1;` у линији број 6,
  - услов $b < 0$ важи у линији број 7, након чега се извршава наредба `b=-1;` у линији број 10,
  - променљива *div* добија вредност $a + b + 2$ у линији број 11, а променљива k добија вредност $1/div$ у линији број 12.

  Дељење у линији број 12 за ову путању није безбедно јер се, на основу информација генерисаних приликом симболичког извршавања ове путање, може закључити да је вредност променљиве *div*, приликом извршавања ове наредбе, једнака 0.

На основу претходне анализе, може се закључити да постоји путања која води до грешке у извршавању програма. За овај кôд и овакав редослед претраживања путањи, грешка се открива тек анализом последње путање кроз програм, мада, за нешто другачији кôд или за другачији редослед претраживања путањи, грешка може да буде откривена и у првој путањи за коју се врши анализа.

Аналогни програм, који има 20 променљивих и 20 `if` наредби, садржи $2^{20} = 1\,048\,576$ различитих путањи кроз програм и јасно је да у овом случају обилазак свих путања кроз програм захтева незанемарљиво време. На пример, ако претпоставимо да је за анализу једне путање потребна једна милисекунда, онда је за анализу свих путања у овом случају потребно око 18 минута. Према томе, време проналажења грешке у таквом програму може да варира од једне милисекунде до 18 минута, у зависности од редоследа анализе путањи.

Повећавањем броја променљивих и броја `if` наредби, број путања експоненцијално расте. Тако већ за 30 променљивих и 30 `if` наредби програм садржи 1 073 741 824 различитих путања за чију би комплетну анализу било потребно око две године (под истом претпоставком да се анализа једне путање може урадити за једну милисекунду). Очигледно је да је анализа аналогног програма са 100 променљивих и 100 `if` наредби ван домашаја алата за симболичко извршавање.

Програми са оваквом контролном структуром нису нереални. Многе функције из стварних програма, као што су лексички анализатори и парсери, садрже велики број наредби гранања [68].

Резултати поређења система LAV са алатом KLEE, који врши симболичку анализу кода, на претходно описаним програмима различитих величина (тј. са различитим бројем променљивих и различитим бројем `if` наредби), приказани су у табели 4.2. Резултати показују да LAV може ефикасно да анализира програме који су ван домашаја симболичког извршавања. Такође, резултати показују да код система LAV време потребно за анализу програма не зависи значајно од путање у којој се грешка налази.

Табела 4.2: *Приказан је број* `if` *наредби и број променљивих, број путања кроз програм и време потребно систему LAV и симболичком извршавању алатом KLEE уколико је грешка у првој путањи, последњој путањи или уколико нема грешке. Времена су дата у секундама. Стрелица ↗ означава да је максимално време предвиђено за анализу (10 минута) истекло.*

број `if` наредби и број променљивих	број путања	LAV			KLEE		
		грешка у првој путањи	грешка у последњој путањи	без грешака	грешка у првој путањи	грешка у последњој путањи	без грешака
2	4	0.07	0.07	0.07	< 1	0.05	0.05
5	32	0.18	0.19	0.18	< 1	0.55	0.55
10	1 024	0.41	0.46	0.38	< 1	45.00	45.00
11	2 048	0.42	0.54	0.43	< 1	107.00	107.00
12	4 096	0.50	0.67	0.50	< 1	268.00	268.00
20	1 048 576	0.73	1.82	0.72	< 1	↗	↗
30	1 073 741 824	1.09	2.95	1.10	< 1	↗	↗
40	1 099 511 627 776	1.64	10.00	1.70	< 1	↗	↗
50	2^{50}	23.00	26.00	2.68	≈ 1	↗	↗
60	2^{60}	25.00	39.00	4.18	≈ 1	↗	↗
70	2^{70}	69.00	210.00	6.92	≈ 1	↗	↗
80	2^{80}	70.00	88.00	8.10	≈ 1	↗	↗
90	2^{90}	154.00	108.00	11.00	≈ 1	↗	↗
100	2^{100}	153.00	111.00	15.00	≈ 1	↗	↗

4.2.2 Експериментално поређење

Експериментално поређење система LAV извршено је са алатима CBMC и ESBMC који се заснивају на техникама проверавања ограничених модела и са алатом за симболичко извршавање KLEE. Поређење је извршено на корпусу који је специјализован за евалуацију алата који врше статичку анализу програма. Резултати поређења показују да LAV и по прецизности и по ефикасности даје сличне, а у неким ситуацијама и боље резултате у односу на сродне алате.

Опис корпуса

Америчка лабораторија NEC (енг. NEC Laboratories America, Inc.) развила је различите јавно доступне корпусе који се могу користити за евалуацију алата за верификацију хардвера и софтвера [147]. У оквиру ових корпуса налази се и корпус специјализован за евалуацију алата за статичку анализу програма: NECLA корпус за статичку анализу (енг. NECLA static analysis benchmarks). Овај корпус се састоји од малих C програма који садрже уобичајне ситуације и грешке које се јављају у реалним програмима. Неки програми из корпуса садрже грешке док неки не садрже грешке. Могућност алата да покаже да су одговарајући програми исправни (односно неисправни) демонстрира предности и мане алата.

У евалуацију је укључено 44 од 57 програма из корпуса. Укључени су сви ANSI C програми осим програма који садрже рекурзивне позиве функција, програма који користе функције из библиотеке за рад са нискама и програма који садрже израчунавања са бројевима у покретном зарезу.[7]

Опис експеримента

Алати CBMC, ESBMC, KLEE и LAV су у програмима из корпуса проверавали:

- присуство грешака у раду са показвачима,
- прекорачења бафера,
- дељење нулом,
- кориснички задате услове.

Рад са показивачима присутан је у 80% програма, при чему је у 63% програма меморија за показиваче резервисана статички, а у 25% програма меморија за показиваче резервисана је динамички (у неким програмима присутна је и статичка и динамичка алокација меморије). Оператор дељења се користи у свега 7% програма, а кориснички задати услови у 32% програма. Кориснички задати услови најчешће су нумеричке релације које укључују односе вредности промењивих у програму. Сваки од алата је исправно завршавао рад генерисањем одговарајућег извештаја када би пронашао прву грешку у програму или када би утврдио да грешака у програму нема. Под нерегуларним завршетком рада сматра се генерисање неисправног излаза као што је:

- порука да је дошло до грешке приликом извршавања алата,

[7] Поред алата LAV, грешке у овим програмима не могу да пронађу ни други разматрани алати.

- лажно упозорење,
- истек времена предвиђеног за анализу.

Алат CBMC, иако има подршку за SMT решаваче, најстаблиније ради са SAT решавачем, па је на тај начин алат и коришћен у експерименту. Алат KLEE за моделовање програма користи само теорију бит-вектора и теорију низова, па су зато и LAV и ESBMC подешени да користе ове теорије. Алати ESBMC и LAV имају подршку за рад са решавачима Z3 и Boolector, па су оба алата покретана по два пута, први пут тако да користе Boolector решавач, а други пут тако да користе Z3 решавач. Алат KLEE је покретан само са подразумеваним параметрима, јер због природе алата њему није могуће ограничити број размотавања петљи. Остали алати су покретани на следећи начин.

1. Сваки алат је најпре покретан коришћењем својих подразумеваних параметара.

2. Уколико би неки алат за своје подразумеване вредности произвео неисправан излаз, онда су алати покретани са постављеном горњом границом за размотавање петљи (уколико таква граница постоји).

3. Уколико покретање алата са прецизираном горњом границом размотавања петљи произведе неисправан излаз за неки алат или уколико горња граница размотавања петље не постоји, тада је коришћена мања горња граница за размотавање петљи (обично три размотавања).

Експеримент је извршен на Ubuntu систему са Intel процесором на 1.6GHz и са 1GB RAM меморије. Максимално време извршавања алата постављено је на десет минута. Алати су покретани из командне линије, а време је мерено системским програмом *time*.

Резултати

Резултати експеримента су приказани у табели 4.3. Табела садржи име одговарајућег програма из корпуса, број линија кода у програму, границу размотавања петљи, да ли програм има грешку, име алата и, за алате LAV и ESBMC, који је решавач коришћен (уколико је неки решавач коришћен — постоје ситуације у којима алати упрошћавањем формула исправности одреде статус без коришћења решавача). У оквиру табеле, најбоља времена су дата подебљано, а користе се наредне ознаке:

! — програм садржи грешку,

¡ — програм садржи грешку која је недостижна,
∼! — лажно упозорење,
✓ — програм не садржи грешке,
∼✓ — грешка није пронађена,
def — подразумеване вредности приликом покретања алата,
↓ — крахирање алата или пријава грешке у раду,
↗ — време предвиђено за анализу је истекло,
— — није применљиво и
* — решавач није коришћен.

Табела 4.3: *Експериментални резултати алата LAV, CBMC, ESBMC и KLEE на NECLA корпусу за статичку анализу.*

име	број линија	број размотавања	Резултат	LAV Boolector	LAV Z3	CBMC	ESBMC Boolector	ESBMC Z3	KLEE
ex1	21	*def*	✓	∼!	∼!	**5.02***	**5.02***	**5.02***	**0.19**
		513	✓	↗	↗	**5.02***	**5.02***	**5.02***	—
		3	✓	0.90	0.35	**0.14***	**0.14***	**0.14***	—
ex2	40	*def*	✓	0.63	**0.54**	↗	↗	↗	↓
		1024	✓	↗	↗	↓	Z3	**67.48**	—
		3	✓	1.03	0.47	↓	Z3	**0.27**	—
ex3	24	*def*	!	**0.04**	0.06	0.08	0.09	0.09	**0.04**
ex4	16	*def*	!	0.13	0.24	0.14	0.15	0.18	**0.02**
ex5	18	-	✓	**0.02**	**0.02**	0.06*	0.06*	0.06*	**0.02**
ex6	21	-	✓	0.07	0.11	0.07	Z3	0.07	**0.03**
ex7	28	*def*	✓	**0.22**	**0.22**	↗	↗	↗	↓
		3	✓	0.21	**0.15**	↓	Z3-∼!	∼!	—
ex8	20	*def*	!	**0.13**	0.15	↗	↗	↗	↓
		3	!	**0.14**	**0.14**	∼!	↓	↓	—
ex9	43	*def*	✓	1.34	**0.85**	↗	↗	↗	↓
		1024	✓	↗	↗	↓	Z3-↗	↗	—
		3	✓	2.93	**0.62**	↓	Z3-∼!	∼!	—
ex10	72	*def*	!	1.32	0.59	↗	↗	↗	**0.03**
		17	!	↗	10.47	**0.31**	∼✓	∼✓	—
		3	!	4.02	1.14	**0.13**	∼✓	∼✓	—
ex11	25	*def*	✓	∼!	∼!	↗	↗	↗	↗
		3	✓	**0.05**	0.08	0.06*	0.06*	0.06*	—
ex12	24	*def*	!	0.12	0.16	0.12	0.10	0.10	**0.03**
ex13	10	-	!	**0.03**	0.44	0.07	0.06	0.13	↗
ex14	16	*def*	✓	0.10	0.13	0.08*	0.08*	0.08*	**0.03**
ex15	35	-	✓	0.56	0.34	∼!	Z3	0.09	**0.03**
ex16	35	*def*	¡	**0.09!**	0.10!	↗	↗	↗	↗
		2	¡	**0.08!**	0.09!	**0.08*✓**	**0.08*✓**	**0.08*✓**	—
ex17	45	*def*	✓	1.56	0.68	0.34*	0.24*	0.24*	**0.02**
ex18	30	*def*	✓	∼!	∼!	↗	↗	↗	↓
		100	✓	↗	↗	↓	↓	↓	—
		10	✓	**1.30**	3.0	↓	↓	↓	—

Наставак на наредној страни

Табела 4.3 – наставак са претходне стране

име	број линија	број размотавања	Резултат	LAV Bool-ector	LAV Z3	CBMC	ESBMC Bool-ector	ESBMC Z3	KLEE
ex19	29	*def*	✓	∼!	∼!	↗	↗	↗	↗
		3	✓	0.14	**0.08**	0.10	**0.08**	0.09	—
ex20	33	*def*	!	∼!	∼!	↗	↗	↗	**0.12**
		1024	!	455	↗	40.98	**40.0**	206	—
		3	!	0.21	0.32	0.25	**0.09**	0.11	—
ex21	26	*def*	✓	0.45	0.36	∼!	Z3	1.68	**0.02**
ex22	39	*def*	✓	12.22	4.1	0.64	Z3	0.81	**0.06**
ex23	26	*def*	✓	∼!	∼!	16.49	**0.16**	0.18	0.69
		36	✓	25.14	6.46	16.49	**0.16**	0.18	—
ex25	27	*def*	✓	**0.26**	0.27	↗	↗	↗	↗
		3	✓	0.21	0.20	0.10*	**0.08***	**0.08***	—
ex26	30	*def*	!	1.88	**0.62**	6.42	Z3	4.79	∼✓
ex27	40	*def*	!	25.34	5.28	3.40	Z3	3.24	**0.09**
ex30	44	*def*	!	**0.15**	0.24	↗	↗	↗	↓
		100	!	↗	↗	↓	Z3-∼✓	∼✓	—
ex31	14	*def*	✓	∼!	∼!	↗	0.08*	0.08*	**0.02**
		7	✓	1.38	5.62	0.57	**0.08***	**0.08***	—
ex32	27	*def*	✓	0.78	0.5	2.30*	Z3	4.11	**0.18**
ex34	25	-	✓	**0.08**	0.24	0.10	0.12	0.14	0.16
ex37	30	-	!	**0.16**	0.20	∼!	Z3-∼✓	∼✓	↓
ex39	27	*def*	!	**0.06**	0.08	↗	↗	↗	↓
		3	!	**0.07**	**0.07**	∼✓	0.09	0.09	—
ex40	20	*def*	✓	0.09	0.12	↗	↗	↗	**0.02**
		3	✓	**0.08**	0.10	0.12	**0.08**	**0.08**	—
ex41	23	*def*	!	0.25	**0.25**	↗	↗	↗	17
		3	!	0.49	0.44	0.25	**0.07**	0.10	—
ex43	113	*def*	!	28.56	17.91	∼!	Z3	25.15	**0.06**
ex46	38	*def*	!	6.57	**5.75**	↗	↗	↗	↓
		2	!	**37.43**	↗	∼!	Z3-∼!	∼!	—
ex47	35	*def*	!	3.71	**2.32**	↗	↗	↗	↓
		2	!	4.40	**1.38**	∼!	Z3-∼!	∼!	—
ex49	16	*def*	!	0.18	**0.11**	↗	↗	↗	↗
		3	✓	**0.06**	0.08	0.08	0.07	0.08	—
inf1	36	-	!	**0.12**	0.22	0.18	∼✓	0.15	0.13
inf2	63	-	!	4.84	**1.25**	∼!	Z3-∼!	∼!	∼✓
inf4	62	-	!	0.23	0.38	**0.11**	0.12	0.19	0.40
inf5	62	-	!	0.09	0.15	0.11	0.12	0.15	**0.06**
inf6	43	-	✓	0.29	0.12	0.41	Z3	0.40	**0.06**
inf8	44	-	✓	0.16	0.19	0.11	∼!	0.12	**0.06**

Збирни резултати експеримента, који укључују проценте најбољих времена за различите параметре покретања алата као и проценте временских прекорачења, лажних упозорења, неоткривених грешака и крахирања за сваки од алата, дати су у табели 4.4.

Табела 4.4: *Збирни резултати. За неке програме из корпуса ни један алат није дао одговарајуће резултате за параметре из прве и друге врсте, па је збир у овим врстама мањи од 100%. За неке програме из корпуса више алата је дало одговарајуће резултате за исто време за параметре из треће врсте, па је збир у овој врсти већи од 100%. Проценти у последње четири врсте рачунати су у односу на број покретања алата.*

	Алат	LAV	CBMC	ESBMC	KLEE
1.	Најбоља времена са подразумеваним параметрима	45%	2%	0	**47%**
2.	Најбоља времена за горњу границу размотавања	0%	22%	**56%**	—
3.	Најбоља времена за нижу границу размотавања	**66%**	17%	44%	—
4.	Временска прекорачења	**11%**	26%	26%	13%
5.	Лажна упозорења	9%	11%	8%	**0%**
6.	Неоткривене грешке	**0%**	1%	7%	4%
7.	Грешке у раду алата	**0%**	11%	4%	23%

Анализа резултата

У оквиру анализе резултата биће приказане предности и мане алата који су учествовали у евалуацији, као и њихов однос са системом LAV.

CBMC Лажна упозорења и грешке које CBMC није открио могу се објаснити начином на који овај алат моделује меморију и контролу тока програма.[8] На пример, CBMC претпоставља да ће свака динамичка алокација успети. Како ово није ваљана претпоставка, то резултира неким непронађеним грешкама (примери ex8, ex37, ex43, ex46, ex47, inf2). Такође, CBMC не моделује прецизно меморију, која се додељује глобалним низовима и показивачима на показиваче, што доводи до неких лажних упозорења (примери ex15, ex21). LAV прецизно моделује меморију и нема непронађених грешака и лажних упозорења ове врсте.

Размотавање петљи представља основну непрецизност у моделовању контроле тока програма алата CBMC. Ако алат не може да утврди да је петља размотана одговарајући број пута, онда одбацује све информације о претходном извршавању програма, укључујући и информације о меморији која је алоцирана

[8] У примерима ex8, ex37, ex43, ex46, ex47, inf2 алат CBMC пријављује места у програму која нису погрешна и не пријављује праве и очекиване грешке. Зато су ови примери означени у табели као лажна упозорења. Ови примери се могу сврстати и у лажна упозорења и у непронађене грешке, али су у збирној табели бројани само у категорији лажних упозорења.

(статички или динамички) за низове у програму. Како CBMC пријављује само оне грешке за које је сигуран да су грешке (не и наредбе за које не може да утврди да ли су безбедне), ово може да води до неоткривених грешака (на пример, уколико би пример ex17 био незнатно измењен да садржи грешку, CBMC не би био у стању ту грешку да открије). LAV, за разлику од алата CBMC, не одбацује информације о претходном извршавању програма и пријављује и погрешне и небезбедне наредбе програма. Добра страна и предност алата CBMC је што у неким случајевима може сâм, аутоматски, да одреди максимални број размотавања петљи у програму, што LAV не може. CBMC не проверава кôд који је недостижан па према томе не може да пријави ни грешке које се у таквом коду могу наћи. То може да буде проблем у ситуацији када је грешка достижна за велики број размотавања, а није достижна уколико се користи мањи број размотавања који је алат у могућности да провери (пример ex39.c). LAV овакве грешке може да пронађе коришћењем уопштавања полазног кода.

ESBMC Алат ESBMC користи приступну компоненту алата CBMC за препроцесирање и почетну анализу програма. Тако се за анализу неких примера (примери ex1, ex5, ex11 и ex14) овај алат у потпуности ослања на приступну компоненту алата CBMC, која, захваљујући разним поједностављивањима, омогућава утврђивање исправности програма без позива решавача. Због тога, у овим примерима, алати CBMC и ESBMC дају исте резултате. ESBMC моделује програме на сличан начин као алат CBMC, али уводи и нека побољшања. ESBMC моделује меморију прецизније и има мањи број лажних упозорења у односу на CBMC. ESBMC, као и LAV, врши моделовање предвиђајући да динамичка алокација меморије не мора да успе, али и поред тога, за разлику од алата LAV, може да не пронађе неке грешке дереференцирања неисправног или NULL показивача (примери ex10, ex30 и ex37). ESBMC има подршку за рад са решавачима Z3 и Boolector. Међутим, овај алат користи Z3 решавач увек када је потребно да користи теорију неинтерпретираних функција (примери ex2, ex7, ex9, ex21, ex22, ex30, ex37, ex43, ex44, ex47, inf2 и inf6). То је зато што решавач Boolector нема подршку за теорију неинтерпретираних функција, а ESBMC, за разлику од алата LAV, нема уграђену подршку за акерманизацију. Такође, ESBMC има извесну грешку у коришћењу решавача, јер даје различите резултате када се користе различити решавачи за исту теорију и исти проблем (пример inf1). На основу табеле 4.4, може се закључити да ESBMC има највећи број временских прекорачења и непронађених грешака, као и да са подразумеваним параметрима, ESBMC није био најефикаснији ни за један програм из

корпуса, али са горњом границом размотавања петљи ESBMC има највећи број најбољих времена.

KLEE Употреба алата KLEE се разликује од употребе алата LAV, CBMC и ESBMC. У програме које KLEE анализира потребно је додати обележја која говоре које променљиве у програму треба да се прате као симболичке, док се код осталих алата подразумева да се све вредности прате симболички. Такође, KLEE, за разлику од осталих алата, не дозвољава динамичку алокацију меморије у величини која је симболичка вредност, тако да KLEE не може да анализира неке програме из корпуса (примери ex2, ex7, ex8, ex9, ex18, ex30, ex37, ex46, ex47). Алат KLEE не може да анализира ни програм у којем се симулира недетерминистички избор као услов уласка у петљу (пример ex39). KLEE не може да се подеси да се заустави када пронађе прву грешку (као што је то могуће са другим алатима). Ипак, ово понашање не утиче на најбоља времена која се пријављују у табели. Пошто је у питању алат за симболичко извршавање, број размотавања петљи не може да се зада као параметар алату, али може број различитих стања које алат разматра. Како ове две мере нису упоредиве, KLEE је поређен са осталим алатима само за подразумеване параметриме. KLEE је шест пута прекорачио време предвиђено за анализу, десет пута је пријавио грешаке у функционисању, два пута није успео да пронађе грешку, није имао лажних упозорења и имао је највећи број најбољих времена. KLEE је био најефикаснији на примерима где постоји само једна могућа путања кроз програм. То је зато што алат успе ефикасно да пронађе ту путању и симболичко извршавање у овим случајевима траје тек нешто више од правог извршавања. Остали алати, због другачије природе моделовања програма, не могу да искористе то што постоји само једна путања кроз кôд. Ипак, у практичним апликацијама, постојање само једне могуће путање кроз програм је редак случај.

LAV Највећи број најбољих времена са подразумеваним параметрима остварио је алат KLEE, док LAV има приближно исти број најбољих времена за подразумеване параметре. LAV има највећи број најбољих времена за нижу границу размотавања петљи и најмањи број временских прекорачења. LAV нема временска прекорачења када се програми корпуса покрену са подразумеваним параметрима. Међутим, како подразумевани параметри за LAV значе уопштавање полазног кода, сва лажна упозорења која је LAV генерисао резултат су рада алата са подразумеваним параметрима. За више од пола програма за које је LAV пријавио лажна упозорења, остали алати су имали прекорачење

времена, па, према томе, ниједан алат није показао жељено понашање. Што се тиче прекорачења времена, LAV има прекорачења времена најчешће у ситуацијама када је кôд било потребно размотати велики број пута. У већини случајева, покретање са подразумеваним параметрима је већ дало добре резултате тако да покретање са горњом границом размотавања петљи не би ни било неопходно. LAV нема неоткривених грешака и нема лажних упозорења за фиксиран број размотавања. Такође, приликом анализе примера у корпусу LAV, за разлику од осталих алата, ниједном није крахирао. Уколико поредимо рад алата са различитим решавачима, видимо да је ефикасност алата LAV са решавачем Boolector веома слична ефикасности која се остварује са решавачем Z3. Суштинске разлике у ефикасности јављају се само у примерима ex20, ex46 (у овим примерима је Z3 прекорачио дозвољено време извршавања, а Boolector није) и ex10 (у овом примеру је Boolector прекорачио дозвољено време извршавања, а Z3 није).

Закључак Експериментални резултати показују да LAV даје добре резултате у поређењу са сродним алатима, посебно када се користи са подразумеваним вредностима параметара и када се користи са малим бројем размотавања петљи у програму. Такође, LAV нема ни једну непронађену грешку што показују да LAV веома прецизно моделује услове исправности и понашање програма.

5

Примена система LAV

Верификација софтвера може да има различите области примене. Типична примена верификацијских алата је у оквиру провера исправности програма код којих је безбедност изразито важна (енг. safety-critical computer programs). С друге стране, верификацијски алати могу да буду корисни и за провере програма код којих безбедност није изразито важна, на пример, за провере програма које развијају студенти у оквиру курсева програмирања. У овом контексту, верификацијски алат би могао да помогне студентима указујући на грешке у програму у ситуацијама када наставник није у близини (што је најчешће случај), док би за наставнике верификацијски алат могао да буде корисан у оцењивању студентских радова. У овој глави, описују се доприноси примене верификацијског алата у аутоматској евалуацији студентских радова.

5.1 Предности и недостаци аутоматске евалуације студентских програма

Аутоматска евалуација програма важна је и за наставнике и за студенте [137, 170]. 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 брзо добијање података о резултатима рада, што је веома важно за процес учења. Постоје студије о различитим приступима аутоматској евалуацији студентских програма [3, 88].

Брзо добијање података о резултатима рада је нарочито корисно студентима уводних курсева програмирања. На овим курсевима, програмирање представља посебно тежак изазов [131]. Студенти имају мало или нимало знања о основним алгоритамским и програмским темама, а често имају и дубоке заблуде

[171]. С друге стране, аутоматско оцењивање студентских радова уводних курсева програмирања је веома важно за наставнике због великог броја студената који ове курсеве похађају.

Користи од аутоматске евалуације програма су још значајнији у контексту учења уз помоћ рачунара. Велики број светских, водећих универзитета нуди бројне курсеве који се могу пратити преко Интернета. Број студената који прате ове курсеве је у порасту и мери се у милионима [6]. У оваквим курсевима, процес подучавања преузима рачунар, контакт са наставником је минималан, па су брзи и квалитетни подаци о резултатима рада студента посебно пожељни.

Аутоматска евалуација студентских радова, поред поменутих предности, може имати и мане. Мане се, пре свега, односе на ситуације у којима би систем оценио рад неадекватно, давањем мањег или већег броја поена у односу на заслужени број поена. Због тога је важно да, у ситуацијама када је лична комуникација са настaвником могућа, правила аутоматског оцењивања студентима буду јасна, како би могли да уоче и пријаве евентуалне неправилнсоти. Међутим, давање неадекватног броја поена могуће је и у наставничкм оцењивању.

5.2 Аутоматско тестирање студентских радова

Постојећи алати за аутоматску евалуацију студентских програма најчешће се заснивају на аутоматском тестирању [58]. Код ових алата, тестирање се користи за проверавање да ли студентски програми показују жељено понашање на изабраном скупу улаза. Постоје и приступи који користе тестирање за анализу других особина софтвера [2]. Најинтересантније такве особине, у образовном контексту, су:

- ефикасност (обично се разматра профилирањем на изабраном скупу тест примера) и
- присуство безбедносних грешака (тј. пропуста у кодирању који могу да доведу до неочекиваног тока извршавања програма или до краха програма).

На евалуацију, засновану на аутоматском тестирању, значајно утиче избор тест примера. Тест примери могу да буду осмишљени од стране наставника и/или аутоматски генерисани. За проверавање функционалне исправности програма најчешће се користи комбинација наставничких тест примера и аутоматски генерисаних тест примера техником случајног тестирања (eng. random testing) [113]. Наставничким тест примерима се проверава да ли студентски

програми показују жељено понашање на изабраном скупу улаза, док се случајно генерисаним тест примерима проверава одсуство безбедносних грешака у програму. Тестирање, без обзира на начин генерисања тест примера, не може да гарантује функционалну исправност програма.

Наставнички тест примери смишљају се према очекиваном решењу и наставник не може да предвиди све важне путање кроз студентско решење. Поред тога, за тестирање није довољно да тест примери покрију све важне путање кроз програм. Неопходно је и пажљиво одабрати вредности променљивих за сваку путању: за неке вредности студентски пропуст може да буде откривен, док за неке друге вредности, иако на истој путањи, студентски пропуст може да остане неоткривен.

Случајно генерисани тест примери могу да открију једноставне, пре свега безбедносне, грешке веома ефикасно. Међутим, за дубље грешке, односно грешке које су у компликованим путањама програма за које су потребне специфичне вредности улазних параметара, постоји веома мала вероватноћа да ће бити откривене на овај начин [109, 74]. Чак и ако неки тест пример успе да изазове грешку у понашању студентског решења (на пример, прекорачење бафера у C програму), ово не мора нужно да води до нежељеног понашања програма. То значи да грешка (тј. пропуст у кодирању) у оваквој ситуацији може да остане неоткривена, што је илустровано примером 5.1.

Пример 5.1 *Функција приказана на слици 5.1 са леве стране је издвојена из студентског програма написаног на испиту. Приказана функција рачуна максималне вредности врста матрице и те вредности уписује у низ. Ова функција се користи у контексту где је меморија за матрицу статички алоцирана и број врста и колона је мањи или једнак алоцираном простору за матрицу. Ипак, у линији* 11 *постоји могуће прекорачење бафера, јер* $i + 1$ *може да премаши алоциран простор за врсте матрице. Могуће је да ова грешка неће утицати на излаз програма и да неће уништити никакве податке, али у мало измењеном и другачијем контексту ова грешка може да буде опасна. Зато студент треба да буде упозорен на ову грешку. Оваква грешка не може лако да се открије тестирањем, али се може открити верификацијским алатом као што је LAV.*

Када се са тест примером открије грешка у програму, уколико та грешка узрокује крах програма (на пример, у програмском језику C са поруком

```
0:  #define max_size 50
1:  void matrix_maximum(int a[][max_size], int rows, int columns, int b[])
2:  {
3:      int i, j, max=a[0][0];                int i, j, max;
4:      for(i=0; i<rows; i++)                 for(i=0; i<rows; i++)
5:      {                                     {
6:                                                max = a[i][0];
7:          for(j=0; j<columns; j++)              for(j=0; j<columns; j++)
8:              if(max < a[i][j])                     if(max < a[i][j])
9:                  max = a[i][j];                        max = a[i][j];
10:         b[i] = max;                           b[i] = max;
11:         max=a[i+1][0];
12:     }                                     }
13:     return;                               return;
14: }
```

Слика 5.1: *Прекорачење бафера у коду са леве стране (који рачуна максималну вредност сваке врсте матрице) не може да се открије тестирањем. Исправно решење, без прекорачења бафера, приказано је са десне стране.*

Segmentation fault), то није ситуација коју студент, а посебно неко ко је нов у програмирању, може лако да разуме и користи за отклањање узрока грешке. Додатно, у контексту аутоматског оцењивања, овај резултат не може једноставно да се користи јер може имати различите узроке.

5.3 Студентски програми и аутоматско откривање грешака

Да би се утврдило да ли верификацијски алат може ефикасно да пронађе грешке у студентским програмима, која је природа тих грешака и колико их има, за анализу корпуса студентских програма употребљен је систем LAV. Овај корпус се састоји од 157 студентских програма[1] који су писани на тестовима и завршним испитима уводног курса програмирања у програмском језику C [168]. Број линија кода по програму из корпуса варира од 30 до 60, при чему је просечан број линија кода 42. У оквиру корпуса налазе се решења различитих задатака који се могу поделити у три групе:

1. решења проблема која укључују целобројна израчунавања и рад са аргументима командне линије;
2. решења проблема који укључују рад са низовима и матрицама;
3. решења проблема који укључују рад са нискама и слоговима.

[1] Овај корпус је јавно доступан са LAV стране: http://argo.matf.bg.ac.rs/?content=lav.

Корпус укључује сва студентска решења која не садрже синтаксне грешке, при чему неки програми у оквиру корпуса не задовољавају спецификацију задатог проблема. У оквиру овог експеримента, разматрано је само постојање безбедносних грешака[2], не и функционална коректност програма. LAV је коришћен са својим подразумеваним параметрима и време које је LAV провео анализирајући ове програме је занемарљиво.

LAV је у оквиру корпуса пронашао 423 грешке у 121 програму и имао је 32 лажна упозорења у 8 програма. Просечан број грешака које је LAV пронашао по решењу је 2.69, а просечан број лажних упозорења по решењу је 0.20. Табела 5.1 садржи податке о анализираном корпусу и податке о броју пријављених грешака и лажних упозорења.

Табела 5.1: *Број решења која су разматрана за дати проблем, просечан број линија кода по решењу, просечан број пријављених грешака по решењу и просечан број лажних упозорења по решењу.*

Група проблема	Број решења	Просечан број линија	Просечан број грешака	Просечан број лажних упозорења
израчунавања и аргументи ком. линије	60	30	0.82	0.05
низови и матрице	71	46	4.20	0
ниске и слогови	26	60	2.92	1.11
Укупно	157	42	2.69	0.20

Табела 5.2 садржи податке о најчешћим грешкама у корпусу: могуће прекорачење бафера је најчешћа грешка у програмима (240 грешака у 111 програма). У прве две групе, највећи број грешака су могућа прекорачења бафера (225 грешака је откривено у 81 програму), док је друга најчешћа грешка дељење нулом (22 грешке у 22 програма). У трећој групи, највећи број грешака су могућа дереференцирања NULL показивача (46 грешака у 15 програма) док је друга најчешћа грешка прекорачење бафера (30 грешака у 15 програма).

[2]Разматране су грешке у раду са меморијом и показивачима, као и грешке дељења нулом.

Табела 5.2: *Најчешће грешке у програмима.*

	Израчунавања Низови и матрице	Ниске и слогови
Најчешћа грешка	Прекорачење бафера	Дереференцирање NULL показивача
Број програма са грешкама	81	46
Број грешака	225	15
Друга најчешћа грешка	Дељење нулом	Прекорачење бафера
Број програма са грешкама	22	15
Број грешака	22	30

Детаљним прегледом и анализом студентских радова, може се уочити да је највећи број студентских грешака (90%) последица погрешних очекивања, на пример:

- улазни параметри програма задовољавају некаква ограничења (71%);
- програм ће увек бити покренут са одговарајућим бројем аргумената командне линије (10%);
- алокација меморије ће увек успети (8%).

Један студентски превид често изазива неколико грешака у остатку програма — у 73% програма са грешкама, недостатак једне провере у програму узрокује две до десет грешака у остатку програма. На пример, недостатак адекватне провере да ли је програм покренут са одговарајућим бројем аргумената командне линије, доводи до две или три грешке прекорачења бафера у разматраном коду (на сваком месту где се аргументи командне линије користе). Други пример је недостатак провере да ли је алокација меморије успела — овај пропуст води до могућег дереференцирања NULL показивача на сваком месту где се показивач користи и уводи четири до десет пријављених грешака по решењу. Ово делом објашњава велики број грешака пронађених у корпусу — додавањем једне

неопходне провере се обично елиминиште неколико грешака у програму. Ове резултате можемо тумачити и на следећи начин: исти студентски превид може да резултира различитим бројем грешака у програму (на пример, недостатак провере да ли је алокација меморије успела у једном решењу узрокује четири, а у другом десет пријављених грешака). Због тога број пронађених грешака није поуздан индикатор квалитета програма.

Поред поменутих извора грешака, у корпусу постоји свега неколико грешака које су имале другачије узроке (као што је коришћење неиницијализованих променљивих, или приступ меморији која није алоцирана). Што се тиче лажних упозорења, она су последица уопштавања кода апроксимацијом петљи или одсуства подршке у оквиру алата за неке функције стандардне библиотеке. У корпусу нису ручно анотиране све грешке тако да није познат број грешака које LAV није успео да пронађе.

Поједностављен студентски програм из корпуса је приказан на слици 5.2. Овај пример илуструје неколико студентских грешака: линије 18 и 19 — два могућа прекорачења бафера (услед неодстатка провере да ли је програм покренут са одговарајућим бројем аргумената командне линије) и линија 12 — могуће дељење нулом. Излаз такође показује да је линија 12 небезбедна и у контексту позива функције `get_digit` из линије 20. Излаз који LAV генерише (приказан са десне стране слике) може да помогне у исправљању ових грешака јер приказује вредности релевантних променљивих које доводе до грешке. Приметимо да би у овом примеру и свака вредност променљиве `d` већа од 32 довела до исте грешке дељења нулом (на системима код којих су целобројне вредности четворобајтне), али да је генерисан излаз у којем је ова вредност 1073741824. Такође, у оквиру излаза приказана је и потенцијална вредност показивача `argv` која у овом случају није релевантна.

Претходни резултати показују да постоји велики број грешака у студентским радовима које верификацијски алат може ефикасно да пронађе. Постојање великог броја грешака значи да студентима треба помоћ да би писали исправне програме. Предсулов писању исправних програма је свест студента о грешкама које могу да се направе. Ова свест се може подстаћи путем прецизно дефинисаних излаза за различите улазе (који помажу студентима да боље разумеју шта се од њих очекује), као и прецизно дефинисаним задацима (који дефинишу понашање програма и за граничне случајеве). Могућност система LAV да ефикасно пронађе грешке у студентским програмима показује да верификацијски алати могу да помогну студентима да пишу исправне програме, јер ови алати могу да скрену пажњу на проблематичне делове кода.

```
1:  #include<stdio.h>
2:  #include<stdlib.h>
3:  int power(int n)
4:  {
5:  int i, pow;
6:  for(i=0, pow=1; i<n; i++, pow*=10);
7:  return pow;
8:  }
9:
10: int get_digit(int n, int d)
11: {
12: return (n/power(d))%10;
13: }
14:
15: int main(int argc, char** argv)
16: {
17: int n, d;
18: n = atoi(argv[1]);
19: d = atoi(argv[2]);
20: printf("%d\n", get_digit(n, d));
21: }
```

```
line 12: UNSAFE
line 18: UNSAFE
line 19: UNSAFE
line 20: 12: UNSAFE

function: get_digit
error: division_by_zero
line 12: d == 1073741824,

function: main
error: buffer_overflow
line 18: argc == 1, argv == 1

function: main
error: buffer_overflow
line 19: argc == 2, argv == 1

function: main
error: division_by_zero
line 20: 12: argc == 512,
             argv == 1,
             d == 1073741824, n == 0
```

Слика 5.2: *Поједностављен пример студентског програма из корпуса (приказан са леве стране) и излаз који LAV генерише (приказан са десне стране).*

5.4 Верификација софтвера у образовном контексту

Грешке у студентским радовима, описане у одељку 5.3, сугеришу потребу коришћења верификацијских алата у аутоматској евалуацији студентских радова. Контекст образовања намеће нове изазове верификацијским алатима.

5.4.1 Избор верификацијских параметара

Ниједан алат за верификацију софтвера не може да у коначном времену пријави све грешке у програму без лажних упозорења (као што је то објашњено у секцији 2.1.2). Лажна упозорења се јављају као последице апроксимација које су неопходне у моделовању програма. Избор одговарајућих апроксимација утиче на ефикасност и прецизност верификацијског система. Без обзира на ефикасност изабране апроксимације, често је потребно користити и додатна временска ограничења.

Апроксимације

Најважнија апроксимација у моделовању програма се односи на апроксимацију петљи. Апроксимација петљи утиче на ефикасност и прецизност анализе. Потпуно прецизна анализа петљи може да се не зауставља у коначном времену. Сужавање полазног кода веома је прецизна техника која није ефикасна за ве-

лике програме и за програме у којима је потребно петље размотавати велики број пута. Уопштавање петље није прецизна техника али је ефикасна и може се применити и на веће програме. У образовном контексту, потребно је пажљиво направити одговарајући компромис између ефикасности и прецизности. Пожељно је да систем буде што ефикаснији, али ипак не и по цену увођења лажних упозорења.

У развоју софтвера, лажна упозорења су непожељна, али ипак нису критична — програмер може да поправи проблем или да утврди да пријављени проблем није права грешка (обе ове ситуације програмер може да очекује и разуме). Међутим, лажна упозорења у оцењивању студентских радова су критична и морају бити елиминисана. За наставнике, лажна упозорења је потребно елиминисати јер процес евалуације треба да буде потпуно аутоматизован и поуздан. За студенте, лажна упозорења је потребно елиминисати, јер би студенти били збуњени извештајем да је грешка нешто што заправо није грешка, и изгубили би поверење у систем који врши евалуацију. Елиминација лажних упозорења значи да, у неким ситуацијама, може да се деси да систем не зауставља рад у коначном времену или да систем пропусти да пронађе неке грешке. У евалуацији студентских радова избор другог приступа има више смисла: алат мора да се зауставља у коначном времену и не сме да уводи лажна упозорења, макар и по цену неких не пронађених грешака.

Временска ограничења

И поред великог напретка у развоју технологије софтверске верификације, верификацијски алати и даље могу да захтевају више времена него што је то адекватно за удобан интерактивни рад. Због тога, у реалним применама у образовању, морају да се користе временска ограничења. Могу да постоје различите политике за коришћење временских ограничења. На пример, уколико верификацијски алат достигне временско ограничење:

- пријавити да програм има грешке (у циљу избегавања непронађених грешака);
- пријавити да програм нема грешке (у циљу избегавања лажних упозорења);
- уколико алат има подршку за рад са више различитих решавача, може се програм проверити користећи исте параметре, али са другим решавачем (са којим, потенцијално, алат неће достигнути временско ограничење). На

овај начин се, у неким ситуацијама, може побољшати прецизност резултата рада алата.

У општем случају, могу се користити различита временска ограничења за наставнике и студенте: дужи рад за наставнике, када се врши неинтерактивно аутоматско оцењивање студентских радова, и краћи рад приликом интерактивног коришћња алата од стране студената.

5.4.2 Употреба система LAV

LAV има велики број опција које се могу прилагодити потребама тренутне примене. За примене у образовању, већина ових опција може да буде постављена на подразумеване вредности.

Најважнији избор у коришћењу система је начин на који LAV размотава петље. LAV има подршку и за сужавање и за уопштавање модела полазног кода (ове две технике су ретко присутне у истом алату). Уколико се користи сужавање модела полазног кода, онда постављање горње границе размотавања петљи зависи од проблема који се решава. Потребно је да наставник ову границу постави за сваки проблем независно.

Пратећи претходно описане принципе компромиса ефикасности и прецизности, LAV се може користити на следећи начин. Прво покретање система је са подразумеваним параметрима, односно са размотавањем петљи уопштавањем модела програма. Ова техника је ефикасна и, уколико нема пронађених грешака на овај начин, то значи да у програму нема грешака и да даље анализе нису потребне. Међутим, ова техника може да доведе до лажних упозорења. Због тога, уколико се потенцијална грешка пронађе у програму, алат се поново покреће, али овога пута са фиксираним параметром размотавања. Ако је грешка и даље присутна, онда се она пријављује; уколико није, претходно откривена потенцијална грешка се сматра лажним упозорењем и не пријављује се.

У развоју софтвера, свака грешка која се открије верификацијским алатом је важна и треба да буде пријављена. Међутим, неке грешке могу да збуне студенте који тек уче да програмирају, на пример, грешка приказана на слици 5.3. У овом коду, у линији `11`, може да дође до прекорачења бафера. На пример, за $n = 0x80000001$ и $sizeof(unsigned) = 4$ само 4 бајта ће бити алоцирана за показивач `array` због прекорачења у рачунању целобројне вредности израза $n * sizeof(unsigned)$[3]. Ово је безбедносна грешка коју ће LAV у нормалним

[3]Постоје аналогни примери и за остале могуће вредности величине типа *unsigned*.

```
1:  unsigned i, n;
2:  unsigned *arr;
3:  scanf("%u", &n);
4:  array = malloc(n*sizeof(unsigned));
5:  if(array == NULL)
6:  {
7:      fprintf(stderr, "Unsuccessful allocation\n");
8:      exit(EXIT_FAILURE);
9:  }
10: for(i=0; i<n; i++)
11:    array[i] = i;
```

Слика 5.3: *Прекорачење бафера у овом коду је безбедносна грешка, али наставник може да одлучи да се ова врста грешке не разматра у аутоматској евалуацији.*

околностима да пријави, али наставник може да одлучи да не разматра такве врсте грешака. За ову намену LAV може да буде покренут у режиму за студенте (тако да се грешка као што је ова, која обухвата целобројно прекорачење у алокацији меморије, не пријављује). У овом режиму, поред стандардних порука о грешци, генеришу се и путања у програму која води до грешке и упутства студенту о врсти и потенцијалном начину исправљања грешке.

5.4.3 Експериментална евалуација

Тестирање не може да гарантује функционалну коректност програма, тј, програми који успешно прођу тестирање могу и даље садржати грешке. Међутим, поставља се питање да ли је овај проблем и практично важан за студентске програме уводног курса програмирања, с обзиром на мали обим и комплексност ових програма. Да би се добио одговор на претходно питање, LAV је анализирао нови корпус студентских радова.

Корпус

За експерименталну евалуацију, у наставку овог одељка, коришћен је корпус студентских програма писаних на испитима и тестовима уводног курса програмирања. Програми су написани у програмском језику C. Корпус се састоји само од програма који су успешно прошли тестирање тест примерима које је осмислио наставник курса. Корпус укључује 266 решења 15 различитих проблема и јавно је доступан[4]. Ови проблеми укључују целобројна израчунавања, рад са низовима и матрицама, рад са нискама и слоговима. Следе кратки описи

[4] http://argo.matf.bg.ac.rs/?content=lav

проблема.

1. (a) Написати програм који проверава да ли су цифре датог четвороцифреног броја уређене у растућем претку.
   (b) Написати програм који рачуна производ парних цифара четвороцифреног броја.
2. (a) Написати функцију која рачуна максималну вредност датог низа. Написати функцју која рачуна аритметичку средину датог низа. Написати програм који користи претходне две функције да одреди да ли је максимална вредност низа бар два пута већа од аритметичке средине низа.
   (b) Написати функцију која израчунава индекс минималног елемента у низу. Написати функцију која израчунава индекс максималног елемента у низу. Написати програм који користи претходне две функције и израчунава да ли је индекс максималног елемента у низу већи од индекса минималног елемента у низу.
3. (a) Написати функцију која конвертује сва мала слова, која су на парним позицијама у нисци, у одговарајућа велика слова, а сва велика слова на непарним позицијама дате ниске, у одговарајућа мала слова. Написати програм који користи ову функцију. Улазна ниска није дужа од 20 карактера.
   (b) Написати функцију која конвертује сва мала слова задате ниске, која су на позицијама које су дељиве са три, у одговарајућа велика слова, а сва велика слова која су на позицијама које при дељењу са три дају остатак један, у одговарајућа мала слова. Написати програм који користи ову функцију. Улазна ниска није дужа од 20 карактера.
4. Написати функцију која рачуна низ максималних елемената врста дате матрице. Написати програм који користи ову функцију.
5. (a) Написати функцју која брише карактере са позиције k у задатој нисци. Написати програм који користи ову функцију. Улазна ниска није дужа од 20 карактера.
   (b) Написати функцију која дуплира карактер на позицији k задате ниске. Написати програм који користи ову функцију. Улазна ниска није дужа од 20 карактера.
6. (a) Написати функцију која рачуна суму свих елемената које су изнад споредне дијагонале дате матрице. Написати програм који користи ову функцију.
   (b) Написати функцију која рачуна суму елемената које су испод споредне дијагонале дате матрице. Написати програм који користи ову функцију.
7. Написати програм који рачуна максимум два реална броја.
8. Написати функцију int `strcspn(char* s, char* t)` која рачуна позицију првог појављивања карактера ниске t у нисци s. Написати програм који користи ову функцију. Улазна ниска није дужа од 20 карактера.
9. Дефинисати структуру података `razlomak`. Написати функцију која пореди два разломка. Написати функцију која рачуна минимални разломак у задатом низу. Написати програм који користи ове функције.
10. Написати програм који штампа „машнице” величине n. На пример, за $n = 5$ излаз треба да буде

```
xxxxx
.xxx.
..x..
.xxx.
xxxxx
```

11. Написати програм који рачуна детерминанту матрице димензије 2×2.
12. Написати програм који рачуна максималну вредност три задата броја.
13. Написати програм који штампа вредности косинуса функције у десет еквидистантних тачака интервала $[a, b]$.
14. Написати програм који рачуна број секунди до наредног поднева.
15. Написати програм који за број n штампа бројеве из интервала $[1, n-1]$, затим бројеве из интервала $[2, n-2]$, $[3, n-3]$ и тако редом.

Експериментални резултати

За сваки проблем, LAV је покретан са подразумеваним вредностима. Анализа је прекидана проналажењем прве потенцијалне грешке у програму. Програми код којих је уочена потенцијална грешка проверавани су додатно са фиксираним параметром размотавања петљи[5]. Резултати су приказани у табели 5.3. На систему са Intel процесором i7 са 8 GB RAM меморије, који користи оперативни систем Ubuntu, LAV је у просеку потрошио 2.8 секунди за анализу једног програма.

LAV је открио 35 грешака у 35 програма који су успешно прошли тестирање. При томе, за један програм LAV није успео да пронађе грешку која је откривена ручним прегледањем, а за један програм LAV је пронашао грешку која није била откривена ручним прегледањем. Грешка, која није пронађена ручним прегледањем, налази се у програму који је приказан на слици 5.1. Грешка коју LAV није успео да пронађе последица је ниске горње границе параметра размотавања петљи. Коришћењем подразумеваних вредности, LAV је имао само два лажна упозорења која су успешно елиминисана наредним покретањем алата са фиксираним бројем размотавања петљи. Према томе, у завршном извештају LAV није имао лажних упозорења.

Представљени резултати показују да и у корпусу студентских радова, који су успешно прошли тестирање, постоје грешке, као и да те грешке LAV може ефикасно да пронађе. То значо да верификацијски алат може да унапреди аутоматску евалуацију студентских радова која се заснива на тестирању тест примерима које је осмислио наставник.

[5]За анализу решења три проблема (3, 5 и 8), само фиксиран број размотавања петљи је коришћен. Ово је последица формулације задатка која је дата студентима. Наиме, формулација проблема садржи додатне претпоставке о улазним параметрима које повлаче да неке потенцијалне грешке не треба да се разматрају.

Табела 5.3: *Пронађене грешке у корпусу: друга колона приказује број студентских решења за дати проблем, трећа колона приказује просечан број линија кода по решењу, четврта и пета колона приказују број грешака детектованих ручним прегледањем и алатом LAV, шеста колона приказује број програма за које је LAV показао да су без грешке коришћењем уопштавања петљи полазног кода (подразумевани параметри алата, обележени са 1.) и, када је то било неопходно, коришћењем фиксираног размотавања петљи (обележено са 2.), седма колана приказује бројеве лажних упозорења за подразумеване параметре (1.) и, када је примењиво, за коришћење фиксираног размотавања петљи (2.).*

проблем	број решења	просечан број линија	програми са грешкама ручна провера	програми са грешкама LAV	програми без грешака LAV		лажна упозорења LAV	
					1.	2.	1.	2.
1.	44	29	0	0	44	-	0	-
2.	32	55	11	11	20	1	1	0
3.	7	30	2	2	-	5	-	0
4.	5	43	0	1	3	1	1	0
5.	12	39	3	2	-	10	-	0
6.	7	35	0	0	6	1	1	0
7.	33	14	0	0	33	-	0	-
8.	31	29	11	11	-	20	-	0
9.	10	83	6	6	4	0	0	0
10.	14	36	2	2	12	0	0	0
11.	31	13	0	0	31	-	0	-
12.	18	16	0	0	18	-	0	-
13.	3	20	0	0	3	-	0	-
14.	7	28	0	0	7	-	0	-
15.	12	21	0	0	12	-	0	-
укупно	266	30	35	35	193	38	2	0

5.5 Однос верификације и расплинутог тестирања у образовном контексту

У евалуацији студентских радова могу се користити различите врсте тест примера. Поред наставнички осмишљених тест примера, за оцењивање студентских радова најчешће се користе алати за генерисање случајних тест примера [113]. Ови алати имају за циљ проналажење истих класа грешака које проналазе и верификацијски алати. Зато је потребно размотрити да ли верификацијски алати могу да допринесу прецизнијој и квалитетнијој аутоматској евалуацији студентских програма и да ли постоје предности које верификацијски алат може да понуди у овом контексту у односу на алате за генерисање случајних тест примера.

Алат Bunny [178] је алат за случајно тестирање C програма заснован на стратегијама расплинутог тестирања црне кутије. Алат је јавно доступан, отвореног кода и дистрибуира се под лиценцом Апач 2.0 (енг. Apache License) [105]. Bunny прати процес извршавања програма и податке о извршавању користи

за измену улазних података на начин који омогућава високу покривеност кода. Као и остали алати за расплинуто тестирање, овај алат користи се за откривање грешака које доводе до краха или блокирања програма и не може се користити за откривање других врста грешака у програму.

5.5.1 Експериментални резултати

Да би се извршило поређење са резултатима које је остварио LAV, Bunny је анализирао програме из истог корпуса. Сви тест примери које је алат генерисао су затим ручно проверавани. Тест примери које је Bunny генерисао указују на свега 12 од 35 грешака које је LAV пронашао у програмима који су прошли основно тестирање. За анализу корпуса, Bunny је потрошио значајно више времена у односу на LAV.

Bunny није открио 16 грешака прекорачења бафера (укључујући и грешку из примера 5.1). Како су ове грешке углавном прекорачења бафера за једно место у меморији, а с обзиром на то да не узрокују крах програма, њих вероватно не би могао да пронађе ни један алат који се заснива на расплинутом тестирању стратегијама црне кутије. Да би се пронашле овакве грешке динамичком анализом кода неопходно је користити технике које прецизно прате стање меморије. На пример, алати CRED [145], CCured [45] и Valgrind [127] детаљно прате стање меморије са циљем детектовања прекорачења бафера. Међутим, ови алати имају значајно дуже време извршавања [177, 145, 166] и не генеришу тест примере аутоматски. Преосталих 7 грешака, које Bunny није пронашао, су такве да постоји мала вероватноћа да би биле откривене неким другим алатом који врши случајно тестирање, што је чест проблем са овом врстом алата [74].

За решења проблема 3, 5 и 8 (укупно 49 програма) Bunny је произвео тест примере који указују на нерелевантне грешке. Ове грешке нису релевантне због ограничења задатих формулацијама задатка која дефинишу могуће вредности улазних параметара програма. Ова ограничења не могу да се задају алату Bunny, јер је он неосетљив на ограничења улазних параметара (енг. protocol-blind fuzzer). За 26 програма, Bunny је генерисао лажна упозорења: Bunny је пријављивао да се програм заглавио у ситуацијама када је програм био заузет штампањем великих количина коректних података на стандардни излаз. За 5 програма Bunny је генерисао тест примере који нису представљали усправне улазне податке (на пример, први број на улазу одређује број улазних података које програм треба да прочита, док се тест примером не обезбеђује довољна количина улазних података). Ови тест примери откривају безбедносне пропусте (недостатак провере да ли је функција `scanf` успешно извршила учитавање са

стандардног улаза). Међутим, у контексту студентских задатака овакви пропусти нису релевантни.

Као и код алата за верификацију, где број пронађених грешака није добар индикатор квалитета програма, тако и код алата као што је Bunny, број генерисаних тест примера не може да се користи као мера квалитета програма који се анализира:

- број генерисаних тест примера може драстично да се разликује за програме који представљају решења истог проблема и садрже исте грешке, на пример, овај број на анализираном корпусу варира од 1 до 228 за решења истог проблема;
- велики број тест примера генерисаних за један програм може да одговара истој грешци у програму;
- неки тест примери могу да одговарају лажним упозорењима.

Коришћење само првог генерисаног тест примера (као што је LAV коришћен да извештава само за прву пронађену грешку) није добар избор јер овај први тест пример може да буде лажно упозорење.

Слични резултати могу се очекивати и од других алата за расплинуто тестирање стратегијом црне кутије, јер ови алати не могу да пронађу грешке које не узрокују крах програма и нису добри за проналажење дубоких грешака у коду. Случајно тестирање унапређује резултате тестирања тест примерима које састави наставник, јер може да пронађе нове грешке у програмима. Међутим, у поређењу са верифкацијским алатима, случајно тестирање даје лошије резултате: мањи број пронађених грешака, већи број лажних упозорења и мању временску ефикасност.

5.5.2 Информације о грешкама у програму

Тест пример, који се генерише алатом за расплинуто тестирање, садржи улазне вредности са којима је потребно покренути програм да би дошло до краха програма. Тест пример не садржи информације о врсти грешке и линији кода у којој се грешка налази, нити о вредностима променљивих у тренутку када дође до грешке. С друге стране, коришћење алата LAV омогућава добијање смислених и разумљивих информација о грешци у програму. Ове информације укључују линију кода, врсту грешке, путању која доводи до грешке и вредности променљивих на тој путањи. То може да буде од користи студентима да унапреде своје решење. Такође, на основу врсте грешке, LAV генерише и поруку

```
1:  #include<stdio.h>
2:  #include<stdlib.h>
3:  int get_digit(int n, int d);
4:  int main(int argc, char** argv)
5:  {
6:      int n, d;
7:      n = atoi(argv[1]);
8:      d = atoi(argv[2]);
9:      printf("%d\n", get_digit(n, d));
10:     return 0;
11: }
```

```
verification failed:
line 7: UNSAFE

function: main
error: buffer_overflow
in line 7: counterexample:
argc == 1, argv == 1

HINT:
A buffer overflow error occurs when
trying to read or write outside the
reserved memory for a buffer/array.
Check the boundaries of the array!
```

Слика 5.4: *Поједностављен студентски програм писан на испиту (са леве стране) и излаз који LAV генерише (са десне стране).*

која подсећа студента да дода у програм одговарајуће провере које недостају. Пример приказан на слици 5.4 је поједностављен пример студентског кода писаног на испиту. Овај пример показује пронађену грешку и излаз који LAV за њу генерише.

Са наставничке стране, поузданост информације коју LAV генерише о постојању грешке у програму, омогућава коришћење алата LAV у оцењивању програма. Такође, ова информација може да се узме у обзир у оквиру ширег интегрисаног оквира за аутоматско оцењивање, који је описан у одељку 5.6.

5.6 Аутоматско оцењивање

Аутоматско оцењивање студентских програма је посебно битно за уводне курсеве програмирања јер ове курсеве похађа највећи број студената. Аутоматским оцењивањем се смањују наставничке обавезе прегледања студентских радова, чиме се ствара више времена за активности са студентима које рачунар не може да одмени.

Функционална коректност програма, о чијем је утврђивању до сада било речи, важна је компонента оцене студентског програма. Програм који није функционално коректан свакако не може да освоји максималан предвиђен број поена. Међутим, и функционално коректан програм не мора обавезно да освоји максималан предвиђен број поена. На пример, максималан број поена не може се остварити уколико програм не задовољава друге важне аспекте оцењивања као што су одговарајући дизајн и модуларност (адекватна декомпозиција кода на функције). Слика 5.5 показује фрагменте студентских програма који представљају решења истих проблема, али који су различите модуларности и структуре. Тестирање и верификација не могу да се користе да би се оценили ови

Проблем	Прво решење	Друго решење

1.	`if(a<b) n = a;`
	`else n = b;`
	`if(c<d) m = c;`
	`else m = d;`
		`n = min(a, b);`
		
		`m = min(c, d);`

2.	`for(i=0; i<n; i++)`
	`  for(j=0; j<n; j++)`
	`  if(i==j)`
	`  m[i][j] = 1;`
		`for(i=0; i<n; i++)`
		`  m[i][i] = 1;`

Слика 5.5: *Примери издвојени из студентских радова који илуструју структурне разлике решења истог проблема.*

аспекти програма.

Обично се дизајн и модуларност аутоматски процењују поређењем сличности студентског програма са решењем које је дао наставник. Да би се проверила сличност, могу се анализирати фреквенције кључних речи, број линија кода и број променљивих. Нешто прецизнији приступ евалуацији студентских програма може се реализовати мерењем сличности графова који одговарају програмима [173, 123].

У овој секцији биће приказан оквир за аутоматску евалуацију студентсих програма. Циљ је аутоматско оцењивање малих студентских програма уводних курсева програмирања [170]. Приступ се заснива на комбиновању информација из три различите методе евалуације студентских програма:

1. аутоматско тестирање (наставничким тест примерима);
2. верификација софтвера (тј. аутоматско проналажење грешака);
3. мерење сличности графа контроле тока програма.

Заједничка употреба претходне три методе побољшава квалитет и прецизност аутоматске евалуације превазилажењем индивидуалних слабости сваког од приступа, а мотивисана је следећим запажањима.

- Аутоматским тестирањем не могу се пронаћи све грешке у програму. То је илустровано примером 5.1, а експериментима на корпусу студентских радова који су успешно прошли тестирање показано је да је овај проблем и практично важан. Утврђивање сличности са очекиваним решењем не

може да помогне у проналажењу грешака у програму јер исправна и неисправна решења могу да буду веома слична (што се може илустровати примером са слике 5.1).

- Тестирањем и верификацијом не могу се проценити аспекти модуларности и дизајна програма, што је илустровано примерима кода са слике 5.5.
- Утврђивањем сличности студентског решења са наставничким програмом, без коришћења тестирања и верификације [173, 123], не може се одредити функционална коректност програма, што је илустровано примером 5.2.

Пример 5.2 *Једноставан пример студентског кода, приказан на слици 5.6, покушај је рачунања максимума низа бројева. Ово решење је веома слично очекиваном решењу: не садржи грешаке у раду са меморијом нити грешке које би могле да доведу до краха програма. Међутим, решење није функционално коректано, јер не даје коректан резултат уколико су сви бројеви низа негативни. Овакав пропуст се може утврдити тестирањем.*

```
max = 0;
for(i=0; i<n; i++)
   if(a[i] > max)
       max = a[i];
```

```
max = a[0];
for(i=1; i<n; i++)
   if(a[i] > max)
       max = a[i];
```

Слика 5.6: *Студентски кôд (са леве стране) и очекивано решење (са десне стране). Студентско решење не садржи безбедносне грешке, веома је слично очекиваном решењу, али не приказује жељено понашање што се може лако утврдити тестирањем.*

Како су у претходним секцијама детаљно разматрани аспекти тестирања и верификације у аутоматској евалуацији студентских програма, у наставку текста биће укратко описана техника мерења сличности програма и експериментални резултати који показују да предложени приступ води оцењивању које је у високој корелацији са ручним оцењивањем.

5.6.1 Мерење сличности програма

Да би се евалуирале структурне особине програма, студентско решење се пореди са решењем које предлаже наставник. Студентско решење се сматра добрим уколико је слично неком од предложених наставничких решења [159, 173, 123]. Ова претпоставка не може да се направи за програме великих

студентских пројеката у којима постоји пуно различитих начина решавања проблема који не могу да се унапред предвиде. Међутим, за програме који се пишу на уводним курсевима програмирања, не постоји много смислених решења која су суштински различита или која имају суштински различиту структуру. Нова и непредвидива решења су увек могућа, али у овом контексту ретка. Према томе, реалан ризик је да студент напише програм који је комплекснији него што је потребно, што се поређењем са наставничким решењем може утврдити. Ова претпоставка је разумна и оправдана добрим експерименталним резултатима (који су дати у секцији 5.6.2).

Мерење сличности програма може да се врши на основу графа контроле тока програма (енг. control flow graph). Граф контроле тока програма је усмерени граф који осликава све могуће путање програма за време његовог извршавања [5]. Сваки чвор овог графа је основни блок који се састоји од низа наредби које у себи не садрже скокове, петље или условне изразе. Граф контроле тока раздваја структуру програма и његов садржај и представља погодно средство за структурно поређење програма. Граф контроле тока може се направити коришћењем различитих програма, укључујући и систем LLVM. Да би се мерила сличност програма, потребно је узети у обзир и структуру графа и сличност самих чворова.

Постоје различите мере сличности графова [94, 81, 33, 130]. Мера сличности графова која се користи у предложеном приступу зове се *упаривање суседа* и заснива се на интуитивној идеји сличности: два чвора i и j графова A и B се сматрају сличним уколико се суседни чворови чвора i могу упарити са сличним суседима чвора j [130]. Ова мера, која мери структурну сличност програма, допуњена је тако да се узима у обзир и сличност самих чворова [170]. Сличност чворова, чији је садржај низ наредби, рачуна се на основу едит растојања[6] ових наредби [103]. Мера сличности графова, која се користи у предложеном приступу, може да узима вредности из интервала $[0, 1]$ и експериментално је показано да добро осликава интуитивну сличност програма [170].

Сличност графова се може користити за аутоматско генерисање коначне оцене, али и као самосталан податак о томе колико је студентско решење блиско очекиваном решењу. Ово може бити корисно студентима док пишу своје програме.

[6]Едит растојање је најмањи број операција убацивања, брисања и замена које је потребно урадити над елементима једног низа тако да се он трансформише у други низ.

5.6.2 Оцењивање

Могу постојати различити типови оцењивања у зависности од курса и циљева наставника. Једна од стандардних поставки оцењивања се користи у оквиру уводног курса програмског језика C на Математичком факултету Универзитета у Београду. У оквиру овог курса испит се полаже за рачунаром и очекује се од студената да напишу исправне програме који су решења задатих проблема. Да би се помогло студентима да остваре овај циљ, уз поставку задатка студент добија и неколико тест примера који илуструју жељено понашање очекиваног решења проблема. Студенти имају одговарајуће ограничено време за решавање добијених проблема. Уколико студент не успе да напише програм који ради исправно за виђене тест примере, њено/његово решење се не бодује. У супротном, програм се тестира додатним тест примерима (које студенти нису били у могућности да виде и који су генерисани од стране наставника). У зависности од броја тест примера за које програм исправно ради, програму се додељује одговарајући (пропорционалан) број поена. У ситуацији када сви тест примери успешно прођу тестирање, програм се даље ручно проверава и додатни поени се додају у зависности од осталих особина решења (ефикасност, модуларност, једноставност, одсуство грешака у раду са меморијом и слично).

Пратећи претходно описану поставку рада и оцењивања, настао је корпус студенских радова који је описан и коришћен у секцији 5.4.3. Корпус се састоји од програма који су успешно прошли тестирање јер су такви програми и основни циљ аутоматске евалуције (ручно оцењивање је примењивано само у овом случају). Ови програми освојили су барем 80% максималне оцене (јер су успешно прошли све тест примере). У зависности од квалитета програма који је установљен ручном провером, овим програмима је додат одговарајући удео од преосталих 20%. Оцењивање је вршено на скали од 0 до 10. Оцене придружене програмима корпуса су јавно доступне заједно са самим корпусом[7].

5.6.3 Експериментални резултати

Аутоматски генерисана оцена може се описати једноставним моделом који укључује линеарну комбинацију предложених фактора (тестирање, верификација и сличност) са одговарајућим тежинским коефицијентима:

$$\hat{y} = \alpha_1 \cdot x_1 + \alpha_2 \cdot x_2 + \alpha_3 \cdot x_3$$

при чему важи

[7] http://argo.matf.bg.ac.rs/?content=lav

- $\hat{y}$ је аутоматски генерисана оцена,
- x_1 је резултат који се добија аутоматским тестирањем и чија је вредност у интервалу $[0, 1]$,[8]
- x_2 је 1 уколико је студентско решење коректно на основу верификацијског алата, 0 иначе,[9]
- x_3 је максимална вредност сличности између студентског решења и предложених наставничких решења (вредност из интервала $[0, 1]$).

Могући су различити избори коефицијената α_i, за $i = 1, 2, 3$. У случају описаног корпуса, једноставан избор би могао да буде $\alpha_1 = 8$, $\alpha_2 = 1$ и $\alpha_3 = 1$, јер су сви програми из корпуса освојили 80% укупне оцене захваљујући успеху у фази тестирања.

Међутим, није увек јасно како наставнички интуитивни критеријум може бити преточен у аутоматски мерљиве величине. Наставници не морају да имају интуитивни осећај за величине које су укључене у формули за аутоматско рачунање оцене. На пример, понашање сваке од постојећих мера сличности (укључујући и упаривање суседа) [173, 123, 130] није очигледно из њихових дефиниција. Због тога, може бити нејасно како треба изабрати тежинске факторе за различите променљиве или да ли нека од ових променљивих треба да буде нелинеарно трансформисана са циљем да буде корисна за оцењивање.

Природан избор вредности тежинских фактора је аутоматско подешавање коефицијената α_i, за $i = 1, 2, 3$, тако да понашање модела за предвиђање одговара наставничком стилу оцењивања. У ту сврху, коефицијенти се могу аутоматски одредити коришћењем линеарне регресије (техником најмањих квадрата) [76] на основу корпуса наставнички оцењених програма.

Ради подешавања тежинских фактора и провере квалитета добијеног модела, програми из корпуса су подељени у два скупа.

1. На основу програма из *тренинг скупа* (енг. training set) одређују се коефицијенти линеарног модела. Овај скуп се састоји од две трећине програма из корпуса који су решења осам различитих проблема.
2. На основу програма из *тест скупа* (енг. test set) вршена је евалуација добијеног модела. Овај скуп се састоји од једне трећине програма корпуса

[8] За све програме из корпуса важи да је $x_1 = 1$, јер су сви програми успешно прошли тестирање.

[9] Број пронађених грешака се не користи као параметар у оцењивању, већ само да ли грешка у програму постоји или не. То је зато што број грешака у програму не одговара квалитету решења (што је дистукутовано у одељку 5.3).

који су решења преосталих седам различитих проблема. Тренинг и тест скуп немају заједничких елемената.

Оптималне вредности коефицијената α_i, $i = 1, 2, 3$, су одређене коришћењем метода најмањих квадрата линеарне регресије на основу тренинг скупа. Добијена је формула

$$\hat{y} = 6.058 \cdot x_1 + 1.014 \cdot x_2 + 2.919 \cdot x_3$$

Ова формула за аутоматско предвиђање оцене на први поглед не изгледа интуитивно, с обзиром да је минимална оцена у корпусу 8 и да важи да је $x_1 = 1$ за све инстанце (па би било логично очекивати да је $\alpha_1 \approx 8$). Овај раскорак је последица чињенице да је минимална вредност за x_3 за сва решења у корпусу једнака 0.68. У корпусу не постоји програм са ниском вредношћу сличности јер су сва решења у корпусу релативно добра (сва су успешно прошла тестирање). Уколико се ово узме у обзир, претходна формула може да се запише на следећи начин

$$\hat{y} = 8.043 \cdot x_1 + 1.014 \cdot x_2 + 0.934 \cdot x_3'$$

при чему је $x_3' = \frac{x_3 - 0.68}{1 - 0.68}$, тако да променљива x_3' узима вредности из интервала $[0, 1]$. Ово значи да када се опсег променљивих x_2 и x_3 скалира на интервал $[0, 1]$, њихов допринос оцени на посматраном корпусу је веома сличан.

Модел $\hat{y}$, који је комбинација евалуационих техника подешених на тренинг скупу у односу на наставнички стил оцењивања, упоређен је са моделом којем су унапред дефинисане вредности тежинских фактора аутоматског оцењивања и са техникама индивидуалне евалуације програма (модели који не обухватају све поменуте параметре):

- модел $\hat{y}_1 = 8 \cdot x_1 + x_2 + x_3$ има унапред дефинисане параметре,
- модел $\hat{y}_2$ има унапред дефинисан параметар за тестирање ($\alpha_1 = 8$, јер су сви програми из корпуса добили најмање 8 поена на основу резултата тестирања), а трениран је са информацијама о резултатима верификације и сличности,
- модел $\hat{y}_3$ је трениран само са информацијама о резултату верификације x_2 (без коришћења мере сличности),
- модел $\hat{y}_4$ је трениран само на основу сличности x_3 (без коришћења информација о верификацији).

Резултати поређења модела дати су у табели 5.4. Ова табела приказује коефицијент корелације наставничке оцене и аутоматски предвиђене оцене на програмима из тест скупа, p-вредност као индикатор статистичке значајности резултата[10] и релативну грешку аутоматски предвиђене оцене у односу на наставничку оцену[11]. Резултати показују да модел $\hat{y}$ има најбоље перформансе: највиши степен корелације са наставничким оценама (0.842), најмању релативну грешку (10.1%), као и да су добијени резултати статистички значајни (p-вредност<0.001). Нешто слабији резултати добијени су за моделе $\hat{y}_1$ и $\hat{y}_2$, док модели који не укључују све три компоненте евалуације дају значајно лошије резултате.

Табела 5.4: *Особине модела предвиђања оцена на тест скупу. За сваки модел се наводи да ли узима у обзир све понуђене променљиве и да ли су коефицијенти унапред дефинисани или су рачунати прилагођавањем у о односу на тренинг скуп. У табели су приказани: коефицијент корелације, p-вредност и релативна грешка.*

	Параметри/ Одређивање	Коефицијент кореалције	p-вредност	Релативна грешка
$\hat{y}$	сви/ прилагођени	0.842	<0.001	10.1%
$\hat{y}_1$	сви/ предефинисани	0.730	<0.001	12.8%
$\hat{y}_2$	сви/ прилагођени (предефинисано α_1)	0.727	<0.001	13.9%
$\hat{y}_3$	без сличности/ прилагођени	0.620	<0.001	16.7%
$\hat{y}_4$	без верификације/ прилагођени	0.457	<0.001	17.7%

Резултати показују да су приступи, који користе информације о тестирању, верификацији и сличности, квалитетнији у односу на приступе који уз тести-

[10] p-вредност је резултат статистичког теста у односу на нулту хипотезу, тј. да не постоји корелација између предвиђене оцене и оцене коју је дао наставник. Ниска p-вреднсот означава да су добијени резултати статистички значајни.

[11] Релативна грешка је просечна грешка подељена дужином интервала на којем су даване оцене (који је од 8 до 10 у случају разматраног корпуса).

рање користе само једну од претходне две информације, као и да аутоматски подешени коефицијенти модела обезбеђују боље предвиђање оцена, него уколико се коефицијенти унапред (у потпуности или делимично) одреде. Такође, на основу резултата може да се закључи да постоји јака и стабилна веза између оцена које је дао наставник и променљивих x_i, што значи да модел $\hat{y}$ омогућава поуздано предвиђање оцене.

5.7 Сродни приступи и алати

Постоје различити приступи и алати за евалуацију студентских програма. Основна и најстарија евалуација је наставничко оцењивање. Модерни приступи користе алате за аутоматско тестирање, алате који процењују дизајн програма и алате који користе верификацијске технике за проналажење грешака у студентским програмима. У наставку текста биће укратко описани алати који одговарају овим приступима. Алати који се описују се не користе у експерименталном поређењу са предложеним моделом оцењивања јер:

- неки од поменутих алата не генеришу аутоматски оцену већ само пружају помоћ у аутоматизацији процеса евалуације;
- неки од поменутих алата за евалуацију користе само аутоматско тестирање (па поређење са таквим алатима нема много смисла);
- већина поменутих алата није јавно доступна нити је отвореног кода;
- ниједан јавно доступан алат не генерише аутоматски оцене коришћењем комбинације различитих техника аутоматске евалуације.

5.7.1 Наставничко оцењивање

Током година подучавања програмирања, развиле су се различите технике и приступи оцењивању студентских радова. У овим приступима, основна ствар, која обично није формализована већ се прати интуитивно, приписивање је различитих тежинских фактора одређеним аспектима студентских решења (као што су ефикасност, дизајн, коректност и стил). На пример, постоји систем оцењивања у којем се 25% максималне оцене додељује на основу дизајна програма, 20% за испуњавање спецификације програма, 20% за могућност извршавања програма, 15% за стил програмирања, 10% за постојање одговарајућих коментара и 10% за креативност [84]. Оваква расподела је често индивидуална или зависи од курса, тако да је веома корисно да систем, који врши аутоматизацију

процеса евалуације, може да прихвати стил оцењивања конкретног наставника. Предложени модел оцењивања дозвољава коришћење тежинских фактра за различите аспекте оцењивања, а описани приступ дозвољава и рачунање специфичних фактора који одговарају стилу оцењивања натавника (на основу већ оцењених примера).

5.7.2 Алати засновани на аутоматском тестирању

Аутоматско тестирање је најчешћи начин еваулације студентски програма [58]. Тест примере обично саставља наставник и/или се аутоматски генеришу техникама случајног тестирања [113]. Основни проблем аутоматског тестирања је то што систем може да погреши у давању оцене уколико студент не произведе жељени излаз у очекиваном формату.

Постоји велики број алата заснованих на аутоматском тестирању програма различитих програмских језика. То су, на пример, алати PSGE [80], Automark [69] (за fortran 77), Kassandra [115] (за Maple и Matlab кôд, за курсеве научног израчунавања), Schemerobo [146] (функционално програмирање у оквиру програмског језика Scheme), TRY [142] (Pascal), HoGG [120] (Java), BAGS [121], JEWL [66] (аутоматско оцењивање програма са графичким интерфејсом у програмском језику JAVA) и JUnit [174] (тестирање модула у програмском језику Java). Сви претходни алати користе само аутоматско тестирање и не узимају у обзир дизајн програма и алгоритам који се користи (за разлику од предложеног модела оцењивања који то узима у обзир индиректно, преко мере сличности). Због овог ограничења, ради прецизнијих резултата евалуације, аутоматско тестирање се често комбинује са наставничким оцењивањем.

Аутоматско тестирање се користи као компонента бројних система за предавање и евалуацију радова преко мреже (енг. web-based system). Примери оваквих система су систем Online Judge [41] за тестирање програма на такмичењима у програмирању, комерцијални алат WebToTeach [9] са подршком за различите врсте вежбања програмирања, сервер Quiver [64] за прављење, одржавање и администрирање квизова, систем Praktomat [180] који дозвољава студентима да читају, прегледају и приступају међусобним решењима истог проблема (са циљем побољшавања квалитета и стила програмирања), систем Web-CAT [62] који охрабрује студенте да пишу своје сопствене тест примере (са циљем да науче важност тестирања и утицај тестирања на квалитет софтвера [86]) и систем Marmoset [153] за предавање и евалуацију радова преко мреже, са подршком за аутоматско тестирање и скупљање различитих верзија кода. У оквиру система Marmoset, који је јавно доступан и отвореног кода, а дистри-

буира се под лиценцом Апач 2.0 (енг. Apache License) [105], подаци о резултату рада се заснивају само на тестирању, док коначно оцењивање ради инструктор након истека одговарајућег рока за писање задатка. За разлику од овог система, предложени приступ има за циљ аутоматско оцењивање које омогућава тренутно добијање завршне оцене, што је веома важно за интерактивно учење. Marmoset је систем који аутоматизује процес тестирања пројеката различитих величина и комплексности, док се предложени приступ оцењивања фокусира на мале проблеме типичне за уводне курсеве програмирања.

Поред поменутих система, постоје и алати за праћење курса (енг. course management tools). Ови алати омогућавају наставнику да прати освојене поене студената у претходним активностима и помажу у оцењивању текућих радова подршком за аутоматско тестирање. Такви су, на пример, алати Assyst [89], BOSS [90], CourseMarker [82] и GAME [34]. Алати BOSS, CourseMarker и GAME, у оквиру механизама за подршку оцењивању, поред аутоматског тестирања користе и опште метрике које процењују квалитет и стил програма који се оцењује. Ове метрике не могу да процене начин на који је проблем решен, тј. дизајн решења и алгоритам који се користи [173]. Предложени модел оцењивања би могао да буде корисна компонента оваквих система.

5.7.3 Верификација у аутоматском оцењивању

Верификацијски алати нису често заступљени у аутоматској евалуацији програма. Постоје кратки прикази о коришћењу алата Java PathFinder за аутоматско генерисање тест примера у образовању [87]. Алат Ceasar [72] има интегрисану подршку за аутоматско тестирање и верификацију, али није намењен применама у образовању. За програме писане у програмском језику Java, систем Marmoset користи алат FindBugs [10] који статичком анализом кода покушава да нађе грешке у студентским програмима. Marmoset не пријављује све грешке које овај алат генерише, али и поред тога може да пријави лажна упозорења.

5.7.4 Оцењивање дизајна студентског решења

Да би се оценио дизајн решења, неопходно је извршити поређење студентског решења са наставничким решењем (или решењима). Алат PASS [159] тестирањем проналази еквивалентне функције студентског и наставничког решења и на основу еквивалентних функција процењује дизајн програма. Алат MOSS [148] одређује сличност два програма упаривањем подниски. На овај начин је могуће пронаћи сличне делове кода, али није могуће одредити сличност повезаности

тих делова (што се може постићи када се за поређење користе графови контроле тока). Постоје и приступи који пореде два програма коришћењем различитих метрика, као, на пример, алати ELP [161] и WAGS [179]. Ови приступи имају за циљ евалуацију задатака који се заснивају на допуњавању кратких недостајућих делова кода (енг. fill in the gap exercises) [172].

Новији алати [173, 123, 104] уводе аутоматско оцењивање C програма које се заснива на сличности графова зависности (eng. dependence graph) [83]. Формирању графова зависности претходе трансоффрмације кода које имају за циљ стандардизацију репрезентације C програма. Алат AutoLEP [172] има подршку за предавање радова, за добијање података о грешкама приликом компилирања и тестирања, као и за добијање информација о сличности студентског и наставничког решења. Ниједан од претходних алата није јавно доступан. Детаљније поређење ових приступа са предложеним начином оцењивања се може наћи у литератури [170].

6

Закључци и даљи рад

У оквиру тезе, представљен је нови верификацијски систем LAV за проналажење грешака и проверавање услова исправности програма. Приказана је имплементација и експериментална евалуација система као и примена система за унапређивање аутоматске евалуације студентских програма. Даљи рад може укључивати унапређивање прецизности моделовања и ефикасности самог система, као и развој одговарајућег окружења за аутоматску евалуацију студентских програма.

Моделовање програма

Систем LAV генерише компактан и прецизан опис услова исправости програма. Формула која описује понашање програма садржи информације о свим могућим путањама кроз програм, чиме се омогућава истовремено расуђивање о свим путањама. Тиме се постиже значајна предност у односу на алате који се заснивају на симболичком извршавању. LAV може успешно да анализира и програме који су у потпуности ван домета ових алата.

Могућа су разноврсна унапређења моделовања програма у оквиру система LAV. Ова унапређења би се заснивала на закључцима донетим на основу остварених резултата.

- За испитивање услова исправности, LAV користи празан контекст, контекст блока, контекст функције и шире контексте. Коришћење различитих контекаста доприноси ефикасности система јер је често закључке о безбедности наредби могуће донети већ на основу празног контекста или контекста блока. Додавање новог контекста (или више њих), који би укључио само неколико блокова претходника, могло би позитивно да утиче на ефикасност система. Ово је мотивисано идејом да се важне провере, које

утичу на исправност неке наредбе, налазе најчешће у близини те наредбе, па није потребно узимати у обзир све блокове који претходе датом блоку у функцији, већ је довољно разматрати само неколико њих.

- Уопштавање модела полазног кода омогућава ефикасно расуђивање о безбедности наредби у програму, али омогућава и увођење лажних упозорења. Унапређење прецизности расуђивања алата уопштавањем модела полазног кода, може се постићи коришћењем инваријанти петљи. На пример, систем PAGAI [79] аутоматски изводи инваријанте петљи у LLVM коду користећи технике апстрактне интерпретације, па је могуће повезивање овог система и система LAV.

- Ручно задавање горње границе размотавања петљи може да буде напорно и оптерећујуће. Постоје ситуације када је горњу границу размотавања могуће аутоматски одредити. Додавање подршке за аутоматско одређивање горње границе размотавања петљи (онда када је то могуће) олакшало би употребу система. Оваква подршка је присутна код алата који су засновани на техникама проверавања ограничених модела.

- У процесу испитивања исправности програма, највише времена троши SMT решавач. То значи да приликом генерисања формула исправности треба радити на њиховом поједностављивању, како би одређивање задовољивости формуле одузимало мање времена.

- У оквиру система није развијено прецизно моделовање функција библиотеке за рад са нискама. Ове функције су чести узроци грешака у програмима. Због тога је важан развој одговарајућег моделовања (или интеграција неког постојећег система) које би омогућило праћење дужине ниски и тиме прецизно дефинисање предуслова и постуслова функција библиотеке за рад са нискама.

Имплементација система

Поред унапређења која се заснивају на изменама или допунама основног приступа моделовању програма, постоји простор и за унапређења имплементације система. Ова унапређења утицала би на ефикасност и употребљивост алата. Идеје за унапређивање имплементације система проистичу из сагледаних могућности система и практичних ограничења која не зависе од начина моделовања програма.

- Алгоритам, који LAV имплементира, заснива се на генерисању формула које описују услове исправности и на испитивању задовољивости ових формула. Испитивање задовољивости формуле најчешће траје дуже од генерисања, при чему време испитивања задовољивости формуле може веома да варира. На пример, могуће је да испитивање задовољивости првих неколико формула траје кратко, да затим испитивање задовољивости неке наредне формуле траје значајно дуже, а да након ње поново следе формуле за које испитивање задовољивости траје кратко. У оваквој ситуацији, тешка формула, за чије је испитивање задовољивости потребно више времена, блокира цео систем, тј. систем наставља са радом тек након што се разреши статус те формуле. Да би се добило на ефикасности система, могу се искористи могућности вишепроцесорског хардвера тако што би један процес (који би се извршавао на једном процесору) генерисао формуле које описују услове исправности, док би други процеси (на другим процесорима) паралелно испитивали задовољивост ових формула. Паралелним испитивањем задовољивости формула избегла би се ситуација у којој, због једне формуле за коју је тешко испитати задовољивост, стоји читав систем. Оваква имплементација би довела до тога да се грешке у програму не откривају редом у којем се појављују током извршавања програма.

- У оквиру имплементације система, постоји простор за побољшавање робустности самог система и корисничког интерфејса. Ова два фактора су кључна за практичну употребљивост система.

- Пожељна је и експериментална евалуација на другим корпусима (на пример, [181] и [98]), како би се боље сагледале могућности система и правци даљег развоја. Такође, укључивање неких постојећих LLVM трансформација, које би утицале на поједностављивање кода пре почетка саме анализе, могло би да допринесе ефикасности система.

Примена у образовању

Коришћење верификацијског алата у аутоматској евалуацији студентских програма доприноси прецизнијем и ефикаснијем проналажењу грешака у студентским радовима. Такође, верификација доноси и могућност генерисања квалитетних и разумљивих података о резултатима рада студента, што је кључно за помоћ и подршку у учењу програмирања.

Да би се омогућила свакодневна примена система LAV у образовању, потребно је интегрисати модул за рачунање сличности графова контроле тока програма и LAV у систем који подржава предавање радова преко мреже, компилирање, аутоматско тестирање, профилирање и откривање плагијаторства студентских програма. Поред тога, могућа су и додатна унапређења предложеног система за евалуацију.

- У оквиру подршке аутоматском тестирању, као допуна наставничким тест примерима, може се користити и алат KLEE [39] за аутоматско генерисање тест примера симболичким извршавањем. Тест примере овим алатом могуће је генерисати на основу наставничког решења. Ти тест примери би били додати у скуп тест примера са којима се тестира свако студентско решење и употпунили би скуп тест примера које је осмислио наставник и на основу којих се проверава да ли студентски програм даје жељене излазе за задате улазе. Поред тога, могуће је генерисати и додатне тест примере на основу студентског решења и проверавати еквивалентност наставничког и студентског решења. На овај начин би тестирањем биле покривене, не само важне путање кроз наставничко решење, већ и важне путање кроз студентско решење. Ови тест примери би се генерисали посебно за свако студентско решење и не би улазили у општи скуп тест примера. Тестирање на овај начин би требало да буде свеобухватније од техника аутоматског тестирања које се доминантно примењују.

- Тестирањем се не може доказати функционална коректност програма. Због важности функционалне коректности програма за примене у образовању, планира се истраживање коришћења система LAV за доказивање функционалне коректности студентских програма. Овај задатак поставља нове изазове. Тестирање, профилирање, проналажење грешака и мерење сличности се користе над оригиналним студентским програмима, па је аутоматизација ових процеса једноставна. За верификацију функционалне коректности, наставник би морао да дефинише услове исправности (обично у терминима имплементираних функција) и да убаци одговарајуће услове на права места у студентским програмима. То се може аутоматизовати у неким случајевима, али није тривијално у општем случају.

- Како укупан број пронађених грешака у раду није добра мера квалитета програма, за аутоматско оцењивање коришћена је само чињеница да ли програм садржи грешку или не. Прецизност оцењивања могла би се побољшати откривањем независних грешака у програму. Ово се може

остварити додавањем нових ограничења у формуле које описују понашање програма. Прецизније, након откривања наредбе која доводи до грешке у програму, у формулу која описује блок којој неисправна наредба припада, може се додати ограничење које гарантује да је услов безбедности наредбе испуњен. На овај начин може се делимично елиминисати пријављивање грешака које имају исти основни узрок.

- Генерисање квалитетног извештаја о грешци веома је важно за студенте јер им омогућава да лакше сагледају пропусте у својим програмима. Извештај о грешци може се, на основу генерисаног контрамодела, додатно прилагодити тако да прецизније указује на могући начин исправљања грешке.

Литература

[1] W. Ackermann. *Solvable cases of the decision problem.* Studies in logic and the foundations of mathematics. North Holland, 1954.

[2] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, June 2009.

[3] K. M. Ala-Mutka. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15:83–102, June 2005.

[4] AlephOne. Smashing the stack for fun and profit. *Phrack Magazine*, 7(47), 1998.

[5] F. E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

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

[7] J.B. et al. Almeida. *Rigorous Software Development.* Undergraduate Topics in Computer Science. Springer, London, 2011.

[8] R. Arm, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *SPIN*, volume 3925 of *Lecture Notes in Computer Science (LNCS)*, pages 146–162. Springer, 2006.

[9] D. Arnow and O. Barshay. WebToTeach: An Interactive Focused Programming Exercise System. *Frontiers in Education, Annual*, 1:12A9/39–12A9/44, 1999.

[10] N Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 1–8. ACM, 2007.

[11] D. Babić and A. J. Hu. Calysto: Scalable and Precise Extended Static Checking. In *ICSE'08*, pages 211–220. ACM, May 2008.

[12] D. Babić and F. Hutter. Spear Theorem Prover. In *SAT'08: Proceedings of the SAT 2008 Race*, 2008.

[13] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract Congruence Closure. *Journal of Automated Reasoning*, 31(2):129–168, December 2003.

[14] T. Bagby. LLVM Ruby, 2012. on-line at: http://llvmruby.org/.

[15] M. Barnett, B. E. Chang, R. Deline, B. J., and K. R. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science (LNCS)*, pages 364–387. Springer, 2006.

[16] M. Barnett and K. Leino. To Goto Where No Statement Has Gone Before. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, volume 6217 of *Lecture Notes in Computer Science (LNCS)*, pages 157–168. Springer, 2010.

[17] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. *ACM SIGSOFT Software Engineering Notes*, 31:82–87, 2006.

[18] C. Barrett, S. Ranise, C. Tinelli, and A. Stump. The SMT-LIB web site, 2008. on-line at: http://www.smt-lib.org/.

[19] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[20] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35TH Design Automation Conference*, pages 522–527, 1998.

[21] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[22] B. Beckert, T. Bormer, and V. Klebanov. On Essential Program Annotations and Completeness of Verifying Compilers. In *Verified Software: Theory, Tools, and Experiments (VSTTE)*, 2009.

[23] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[24] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[25] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[26] 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.

[27] A. Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 2008.

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

[29] Armin 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.

[30] D. W. Binkley. C++ in safety critical systems. *Annals of Software Engineering*, 4:223–234, January 1997.

[31] N. S. BjØrner and M. C. Pichora. Deciding fixed and non-fixed size bit-vectors. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science (LNCS)*, pages 376–392. Springer, 1998.

[32] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, 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 T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science (LNCS)*, pages 85–108. Springer-Verlag, October 2002.

[33] V. D. Blondel, A. Gajardo, M. Heymans, P. Snellart, and P. van Dooren. A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching. *SIAM Review*, 46(4):647—666, April 2004.

[34] M. Blumenstein, S. Green, S. Fogelman, A. Nguyen, and V. Muthukkumarasamy. Performance analysis of GAME: A generic automated marking environment. *Computers & Education*, 50(4):1203–1216, May 2008.

[35] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag.

[36] 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.

[37] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, A. Santuari, and R. Sebastiani. To Ackermannize or Not to Ackermannize? On Efficiently Handling Uninterpreted Function Symbols in $SMT(EUF \cup T)$. In *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2006*, volume 4246 of *Lecture Notes in Computer Science (LNCS)*, pages 557–571. Springer, 2006.

[38] R. Bruttomesso, A. Cimatti, A. Franzén, 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.

[39] 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.

[40] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, TACAS'07, pages 19–33, Berlin, Heidelberg, 2007. Springer-Verlag.

[41] B. Cheang, A. Kurnia, A. Lim, and W. Oon. On Automated Grading of Programming Assignments in an Academic Institution. *Computers & Education*, 41(2):121–131, September 2003.

[42] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform For In-vivo Multi-path Analysis of Software Systems. *ACM SIGARCH Computer Architecture News*, 39:265–278, March 2011.

[43] 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.

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

[45] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 232–244. ACM, 2003.

[46] 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.

[47] P. Cousot. Abstract Interpretation Based Formal Methods and Future Challenges. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 138–156, London, UK, UK, 2001. Springer-Verlag.

[48] 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.

[49] D. Cyrluk, O. Möller, and H. Rueß. An Efficient Decision Procedure for a Theory of Fixed-Sized Bitvectors with Composition and Extraction. In *Computer-Aided Verification*, pages 60–71. Springer, 1997.

[50] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[51] LLVM D Compiler, 2012. on-line at: http://www.ohloh.net/p/ldc.

[52] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[53] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[54] 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.

[55] S. Desikan and G. Ramesh. *Software Testing: Principles and Practice*. Pearson Education Canada, 2006.

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

[57] 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.

[58] C. Douce, D. Livingstone, and J. Orwell. Automatic Test-based Assessment of Programming: A Review. *Journal on Educational Resources in Computing*, 5(3), September 2005.

[59] J. W. Duran and S. C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions of Software Engineering*, 10(4):438–444, July 1984.

[60] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer-Aided Verification*, CAV'06, pages 81–94, Berlin, Heidelberg, 2006. Springer-Verlag.

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

[62] S. H. Edwards. Rethinking Computer Science Education from a Test-First Perspective. In *Companion of the 2003 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 148–155, 2003.

[63] N. Een and N. Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science (LNCS)*, chapter 37, pages 333–336. Springer, Berlin, Heidelberg, 2004.

[64] C. C. Ellsworth, J. Fenwick, James B., and B. L. Kurtz. The Quiver System. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 205–209. ACM, 2004.

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

[66] J. English. Automated Assessment of GUI Programs Using JEWL. *ACM SIGCSE Bulletin*, 36(3):137–141, June 2004.

[67] J. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer-Aided Verification*, pages 173–177, 2007.

[68] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 193–205, New York, NY, USA, 2001. ACM.

[69] W. H. Fleming, K. A. Redish, and W. F. Smyth. Comparison of manual and automated marking of student programs. *Information and Software Technology*, 30(9):547–552, 1988.

[70] Free Software Foundation. GNU gcc, 2013. on-line at: http://gcc.gnu.org/.

[71] 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.

[72] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science (LNCS)*, pages 68–84. Springer, March 1998.

[73] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, New York, NY, USA, 2005. ACM.

[74] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, January 2012.

[75] D. Graham and M. Fewster. *Experiences of Test Automation: Case Studies of Software Test Automation*. Addison Wesley, 2012.

[76] J. Gross. *Linear Regression*. Springer, 2003.

[77] D. J. Grossman. Safe Programming at the C Level of Abstraction. Technical report, Cornell University, 2003.

[78] Haskell LLVM, 2012. on-line at: http://www.haskell.org/haskellwiki/LLVM.

[79] J. Henry, D. Monniaux, and M. Moy. PAGAI: A Path Sensitive Static Analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, December 2012.

[80] J. B. Hext and J. W. Winings. An Automatic Grading Scheme for Simple Programming Exercises. *Communications of the ACM*, 12(5):272–275, May 1969.

[81] M. Heymans and A. Singh. Deriving Phylogenetic Trees from the Similarity Analysis of Metabolic Pathways. *Bioinformatics*, 19:138—146, 2003.

[82] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas. Automated assessment and experiences of teaching programming. *Journal on Educational Resources in Computing*, 5(3), September 2005.

[83] S. Horwitz and T. Reps. The Use of Program Dependence Graphs in Software Engineering. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 392–411. ACM, 1992.

[84] J. W. Howatt. On criteria for grading student programs. *ACM SIGCSE Bulletin*, 26(3):3–7, September 1994.

[85] Alan J. Hu and Moshe Y. Vardi, editors. *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1998.

[86] L. Huang and M. Holcombe. Empirical investigation towards the effectiveness of Test First programming. *Information and Software Technology*, 51(1):182–194, January 2009.

[87] P. Ihantola. Creating and Visualizing Test Data From Programming Exercises. *Informatics in education*, 6(1):81–102, January 2007.

[88] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93. ACM, 2010.

[89] D. Jackson and M. Usher. Grading student programs using ASSYST. *ACM SIGCSE Bulletin*, 29(1):335–339, March 1997.

[90] M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. *Journal of Educational Resources in Computing*, 5(3), September 2005.

[91] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, December 1984.

[92] M. E. Khan. Different Approaches to White Box Testing Technique for Finding Errors. *International Journal of Software Engineering and Its Applications*, 5(3):1–6, July 2011.

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

[94] J. M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46:604 — 632, 1999.

[95] Klog. The Frame Pointer Overwrite. *Phrack Magazine*, 9(55), September 1999.

[96] B. Korel and A. M. Al-Yami. Automated regression test generation. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, pages 143–152, New York, NY, USA, 1998. ACM.

[97] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 1 edition, 2008.

[98] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of ASE '07*. ACM, 2007.

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

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

[101] 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.

[102] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 119–134, Berlin, Heidelberg, 2005. Springer-Verlag.

[103] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[104] J. Li, W. Pan, R. Zhang, F. Chen, S. Nie, and X. He. Design and Implementation of Semantic Matching Based Automatic Scoring System for C Programming Language. In *Proceedings of the Entertainment for education, and 5th international conference on E-learning and games*, pages 247–257. Springer, 2010.

[105] Apache License. on-line at: http://www.apache.org/licenses/LICENSE-2.0.

[106] GNU Lesser General Public Licence. on-line at: http://www.gnu.org/licenses/lgpl-2.1.html.

[107] The University of Illinois/NCSA Open Source License. on-line at: http://perfsuite.ncsa.illinois.edu/LICENSE.html.

[108] The Pure Programming Language, 2012. on-line at: http://code.google.com/p/pure-lang/.

[109] P. Loo and W. Tsai. Random testing revisited. *Information and Software Technology*, 30(7):402–417, 1988.

[110] JIT/Static compiler for Lua using LLVM on the backend, 2012. on-line at: http://code.google.com/p/llvm-lua/.

[111] R. Mahadevan. LLVM-py: Python Bindings for LLVM, 2012. on-line at: http://www.mdevan.org/llvm-py/.

[112] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: an efficient SAT solver. In *Proceedings of the 7th international conference on Theory and Applications*

of Satisfiability Testing, SAT'04, pages 360–375, Berlin, Heidelberg, 2005. Springer-Verlag.

[113] A. K. Mandal, C. A. Mandal, and C. Reade. A System for Automatic Evaluation of C Programs: Features and Interfaces. *International Journal of Web-Based Learning and Teaching Technologies*, 2(4):24–39, 2007.

[114] F. Marić. Implementacija shema za ugradnju procedura odlučivanja u dokazivače teorema. Master's thesis, University of Belgrade, Serbia, 2005.

[115] U. Von Matt. Kassandra: The Automatic Grading System. *ACM SIGCUE Outlook*, 22:22–26, 1994.

[116] E. Mendelson. *Introduction to mathematical logic; (3rd ed.)*. Wadsworth and Brooks/Cole Advanced Books & Software, Monterey, CA, USA, 1987.

[117] 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.

[118] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[119] M. O. Möller and H. Rueß. Solving Bit-Vector Equations. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, FMCAD '98, pages 36–48, London, UK, UK, 1998. Springer-Verlag.

[120] D. S. Morris. Automatically Grading Java Programming Assignments Via Reflection, Inheritance, and Regular Expressions. *Frontiers in Education Conference*, 1:T3G–22, 2002.

[121] D. S. Morris. Automatic Grading of Student's Programming Assignments: An Interactive Process and Suit of Programs. In *Proceedings of the Frontiers in Education Conference 3*, volume 3, pages 1–6, 2003.

[122] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[123] K. A. Naudé, J. H. Greyling, and D. Vogts. Marking Student Programs Using Graph Similarity. *Computers & Education*, 54(2):545–561, 2010.

[124] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[125] G. Nelson and D. C. Oppen. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[126] G. Nelson and D. C. Oppen. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM*, 27(2):356–364, 1980.

[127] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notes*, 42(6):89–100, June 2007.

[128] S. Nidhra and J. Dondeti. Black Box and White Box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, June 2012.

[129] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the Association for Computing Machinery*, 53(6):937–977, November 2006.

[130] M. Nikolić. Measuring Similarity of Graph Nodes by Neighbor Matching. *Intelligent Data Analysis*, 16(6):865–878, 2013.

[131] T. Nipkow. Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, VMCAI'12, pages 24–38, Berlin, Heidelberg, 2012. Springer-Verlag.

[132] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2002.

[133] University of Oxford. goto-cc. on-line at: http://www.cprover.org/goto-cc/.

[134] D. C. Oppen. Reasoning About Recursively Defined Data Structures. *Journal of the ACM*, 27(3):403–411, July 1980.

[135] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, October 1981.

[136] R. Patton. *Software Testing (2nd Edition)*. Sams, Indianapolis, IN, USA, 2005.

[137] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A Survey of Literature on the Teaching of Introductory Programming. In *Working group reports on ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '07, pages 204–223. ACM, 2007.

[138] K. Piromsopa and R. J. Enbody. Buffer Overflow: the Fundamentals. Technical Report MSU-CSE-04-47, Department of Computer Science, Michigan State University, East Lansing, Michigan, November 2004.

[139] S. Ranise and C. Tinelli. The SMT-LIB Format: An Initial Proposal, 2003. on-line at: http://goedel.cs.uiowa.edu/smt-lib/.

[140] S. Ranise and C. Tinelli. Satisfiability Modulo Theories. *Trends and Controversies - IEEE Intelligent Systems Magazine*, 21(6):71–81, 2006.

[141] G. Reedy. Compiling Scala to LLVM, 2012. on-line at: http://greedy.github.com/scala-llvm/.

[142] K. A. Reek. The TRY system -or- how to avoid testing student programs. *ACM SIGCSE Bulletin*, 21(1):112–116, February 1989.

[143] D. W. Reinhardt. Use of the C++ Programming Language in Safety Critical Systems. Technical report, University of New York, 2004.

[144] P. Rümmer and T. Wahl. An SMT-LIB Theory of Binary Floating-Point Arithmetic. In *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*, 2010.

[145] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, feb 2004.

[146] R. Saikkonen, L. Malmi, and A. Korhonen. Fully Automatic Assessment of Programming Exercises. *ACM Sigcse Bulletin*, 33:133–136, 2001.

[147] S. Sankaranarrayanan. NECLA Static Analysis Benchmarks, 2009. on-line at: http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php.

[148] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[149] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[150] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[151] GNU Operating Sistems. GNU Make, 2010. on-line at: http://www.gnu.org/software/make/.

[152] GNU Operating Sistems. GNU Autoconf, 2011. on-line at: http://www.gnu.org/software/autoconf/.

[153] J. Spacco, D. Hovemeyer, W. Pugh, J. Hollingsworth, N. Padua-Perez, and F. Emad. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In *Proceedings of the 11th annual conference on Innovation and technology in computer science education (ITiCSE)*, pages 13–17. ACM Press, 2006.

[154] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *LICS*, pages 29–37, 2001.

[155] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007.

[156] N. Suzuki and D. Jefferson. Verification Decidability of Presburger Array Programs. *Journal of the ACM*, 27(1):191–205, January 1980.

[157] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

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

[159] D. G. Thorburn and G. W. A. Rowe. PASS: An Automated System for Program Assessment. *Computers & Education*, 29(4):195–206, 1997.

[160] 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.

[161] N. Truong, P. Roe, and P. Bancroft. Automated feedback for "fill in the gap" programming exercises. In *Proceedings of the 7th Australasian conference on Computing education*, volume 42 of *ACE '05*, pages 117–126. Australian Computer Society, Inc., 2005.

[162] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[163] D. van Heesch. Doxygen, 1997-2013. on-line at: http://www.doxygen.org.

[164] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[165] A substrate for virtual machines, 2012. on-line at: http://vmkit.llvm.org/.

[166] M. Vujošević-Janičić. Automated Detection of Buffer Overflows in Programming Language C. Master's thesis, University of Belgrade, Serbia, June 2008.

[167] M. Vujošević-Janičić. Ensuring Safe Usage of Buffers in Programming Language C. In J. Cordeiro, B. Shishkov, A. Ranchordas, and M. Helfert, editors, *Proceedings of Third International Conference on Software and Data Technologies — ICSOFT 2008*, volume PL-DPS-KE, pages 29–36. Institute for Systems and Technologies of Information, Control and Communication — INSTICC, 2008.

[168] 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.

[169] M. Vujošević-Janičić, F. Marić, and D. Tošić. Using Simplex Method in Verifying Software Safety. *Yugoslav Journal of Operations Research*, 19(1):133–148, June 2009.

[170] 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.

[171] 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.

[172] T. Wang, X. Su, P. Ma, Y. Wang, and K. Wang. Ability-training-oriented automated assessment in introductory programming course. *Computers & Education*, 56(1):220–226, 2011.

[173] T. Wang, X. Su, Y. Wang, and P. Ma. Semantic Similarity-based Grading of Student Programs. *Information and Software Technology*, 49(2):99–107, 2007.

[174] M. Wick, D. Stevenson, and P. Wagner. Using Testing and JUnit Across the Curriculum. *ACM SIGCSE Bulletin*, 37(1):236–240, February 2005.

[175] F. Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott.* Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence (LNCS/LNAI). Springer-Verlag, Secaucus, NJ, USA, 2006.

[176] Z. Yang, C. Wang, A. Gupta, and F. Ivanvčić. Model checking sequential software programs via mixed symbolic analysis. *ACM Transactions on Programming Languages and Systems*, 14(1):10:1–10:26, January 2009.

[177] S. H. Yong and S. Horwitz. Using Static Analysis to Reduce Dynamic Analysis Overhead. *Formal Methods in System Design*, 27(3):313–334, November 2005.

[178] M. Zalewski. Bunny the Fuzzer, 2008. on-line at: http://code.google.com/p/bunny-the-fuzzer/.

[179] N. Zamin, E. E. Mustapha, S. K. Sugathan, M. Mehat, and E. Anuar. Development of a Web-based Automated Grading System for Programming Assignments using Static Analysis Approach, 2006. International Conference on Technology and Operations Management (Institute Technology Bandung).

[180] A. Zeller. Making Students Read and Review Code. In *ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 89–92. ACM Press, July 2000.

[181] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *ACM SIGSOFT Software Engineering Notes*, 29, 2004.

Додатак А

Употреба система LAV

У наредном тексту биће укратко описана инсталација система, подешавања улазних параметара који одређују начин моделовања програма, прецизност анализе и избор и начин коришћења решавача, као и извештај који систем LAV генерише.

А.1 Инсталација

LAV је развијан у стандардном C++ језику као апликација командне линије. Иако је превођење алата независно од оперативног система, LAV је развијан и тестиран под Linux оперативним системом коришћењем компилатора *GNU gcc* [70]. Компилација система LAV захтева претходно инсталирање LLVM алата као и бар једног од подржаних SMT решавача. LAV користи систем за аутоматско конфигурисање и подешавање параметара пројекта *GNU Autoconf* [152], као и систем за превођење *GNU Make* [151], чиме се максимално олакшава и аутоматизује конфигурисање и превођење система LAV.

У оквиру дистрибуције система LAV дата су детаљна упутства за његову инсталацију. Приликом инсталације система постоји могућност генерисања одговарајуће документације система коришћењем алата *Doxygen* [163].

А.2 Улазни параметри

LAV се покреће из командне линије. Приликом његовог покретања, могу се задати различити параметри којима се прецизирају начин размотавања петљи, провере које LAV извршава, рад решавача, форма и начин генерисања извештаја и општи аспекти система.

- Начин размотавања петљи:

-loop-unroll-begin=<uint> — број размотавања у односу на почетак петље (подразумевана вредност је 2);

-loop-unroll-end=<uint> — број размотавања у односу на крај петље (подразумевана вредност је 1, уколико се ова вредност постави на 0, онда то одговара размотавању петље фиксиран број пута).

- Провере које LAV извршава:

  -check-assert — проверавати *assert* услове задате у оквиру програма који се анализира;

  -check-div-zero — проверавати грешке дељења нулом;

  -check-pointers — проверавати исправност рада са показивачима;

  -skip-inside-loop — провере услова исправности вршити само за први и последњи пролазак кроз петљу; ова опција може се користити само када је извршен фиксиран број размотавања петље (односно када је задато -loop-unroll-end=0).

  LAV подразумевано извршава све наведене провере (да би се искључила нека провера потребно је навести да је вредност одговарајуће опције једнака `false`).

- Рад решавача:

  -solver=<string> — могуће су наредне вредности параметра које се односе на избор решавача и теорија:

  Boolector-BV-ARR-ACK — избор решавача Boolector, теорије бит-вектора, теорије низова и акерманизације;

  Z3-LA-ARR-EUF — избор решавача Z3, теорије линеарне аритметике, теорије низова и теорије неинтерпретираних функција;

  Z3-LA-ARR-ACK — избор решавача Z3, теорије линеарне аритметике, теорије низова и акерманизације;

  Z3-BV-ARR-EUF — избор решавача Z3, теорије бит-вектора, теорије низова и теорије неинтерпретираних функција;

  Z3-BV-ARR-ACK — избор решавача Z3, теорије бит-вектора, теорије низова и акерманизације;

  MS-LA-EUF — избор решавача MathSAT, теорије линеарне аритметике и теорије неинтерпретираних функција;

MS-LA-ACK — избор решавача MathSAT, теорије линеарне аритметике и акерманизације;

MS-BV-EUF — избор решавача MathSAT, теорије бит-вектора и теорије неинтерпретираних функција;

MS-BV-ACK — избор решавача MathSAT, теорије бит-вектора и акерманизације;

Yices-LA-EUF — избор решавача Yices, теорије линеарне аритметике и теорије неинтерпретираних функција;

Yices-LA-ACK — избор решавача Yices, теорије линеарне аритметике и акерманизације;

Yices-BV-EUF — избор решавача Yices, теорије бит-вектора и теорије неинтерпретираних функција;

Yices-BV-ACK — избор решавача Yices, теорије бит-вектора и акерманизације;

-light — одређивање статуса наредби свести само на утврђивање безбедних и небезбедних наредби (тј. не правити разлику између безбедних и недоступних наредби, нити између небезбедних и неисправних наредби);

-track-unreachable — приликом одређивања статуса наредби, утврдити да ли је наредба безбедна или недоступна;

LAV подразумевано користи решавач Boolector, теорију бит-вектора, теорију низова и акерманизацију, прави разлику између небезбедних и неисправних наредби, али не и између безбедних и недоступних наредби.

- Форма и начин генерисања извештаја:

-model — генерисање извештаја који укључује вредности променљивих и путању кроз програм која доводи до грешке;

-find-first-flawed — генерисање извештаја и прекид анализе након проналаске прве сигурне грешке у програму;

-output-folder=<string> — задавање имена излазног директоријума у који ће бити уписан извештај;

-print-html — генерисање извештаја у HTML формату.

LAV подразумевано не генерише модел, врши анализу целокупног кода (тј. не зауставља се када пронађе прву грешку), име излазног директоријума је `Output`, и генерише се само текстуални излаз.

- Општи аспекти:

  -help — одштампати све могуће параметре покретања алата и објашњења њихових значења;

  input — име датотеке која садржи улазни кôд који се анализира, уколико се не наведе, очекује се унос кода са стандардног улаза;

  -memory-flag — пратити стање меморије коришћењем оптимизованог комбинованог приступа описаног у секцији 4.1.2; уколико је ова вредност постављена на `false` за праћење меморије се користи други приступ описан у секцији 3.2.5;

  -timeout=<int> — зауставити рад система LAV након одговарајућег броја секунди;

  -starting-function=<string> — експлицитно задавање имена функције од које се почиње анализа;

  -track-pointers — пратити стање меморије којој се приступа уз помоћ показивача;

  -students-mode — односи се на непријављивање неких грешака и на форму извештаја (о чему ће бити више речи у глави 5).

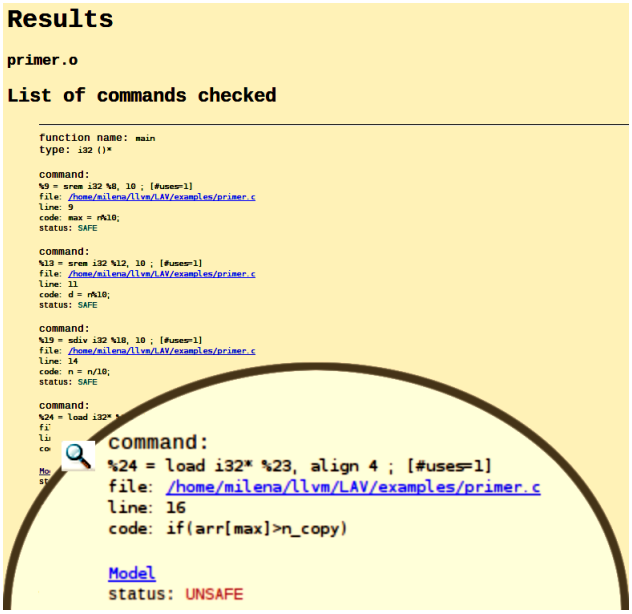
  LAV подразумевано пратити стање меморије коришћењем оптимизованог комбинованог приступа, нема ограничено време извршавања, анализу кода почиње од функције `main`, уколико функција са тим именом постоји, иначе почиње од произвољне функције, прати стање меморије којој се приступа уз помоћ показивача и не користи студентски режим.

A.3 Извештај

У зависности од опција које се задају приликом покретања система, извештај који LAV генерише (одељак 3.6) може бити текстуални и/или HTML документ. Пример кода са одговарајућим текстуалним и HTML извештајем дат је на слици A.1.

```
1: #include <stdio.h>
2: #define MAX 10
3: int main()
4: {
5: int arr[MAX],d,n,n_copy,max;
6: get_array(arr, MAX);
7: scanf("%d", &n);
8: n_copy = n;
9: max = n%10;
10:while(n){
11:  d = n%10;
12:  if(max<d)
13:      max=d;
14:  n = n/10;
15:}
16:if(arr[max]>n_copy)
17:  printf("it is bigger\n");
18:else printf("it is not bigger\n");
19:}
```

```
verification failed:
line: 16   UNSAFE
function: main
error: buffer_overflow
16: d = -3, max = -3, n = 0, n_copy = -34859,
10: d = -3, max = -3, n = 0, n_copy = -34859,
10: d = -3, max = -3, n = 0, n_copy = -34859,
14: d = -3, max = -3, n = 0, n_copy = -34859,
13: d = -3, max = -3, n = -3, n_copy = -34859,
12: d = -3, max = -4, n = -3, n_copy = -34859,
10: d = -4, max = -4, n = -3, n_copy = -34859,
14: d = -4, max = -4, n = -3, n_copy = -34859,
13: d = -4, max = -4, n = -34, n_copy = -34859,
12: d = -4, max = -5, n = -34, n_copy = -34859,
10: d = -8, max = -5, n = -34, n_copy = -34859,
14: d = -8, max = -5, n = -34, n_copy = -34859,
12: d = -8, max = -5, n = -348, n_copy = -34859,
10: d = -5, max = -5, n = -348, n_copy = -34859,
14: d = -5, max = -5, n = -348, n_copy = -34859,
13: d = -5, max = -5, n = -3485, n_copy = -34859,
12: d = -5, max = -9, n = -3485, n_copy = -34859,
10: d = -9, max = -9, n = -3485, n_copy = -34859,
14: d = -9, max = -9, n = -3485, n_copy = -34859,
12: d = -9, max = -9, n = -34859, n_copy = -34859,
10: d = -9, max = -9, n = -3485, n_copy = -34859,
10: d = 0, max = -9, n = -34859, n_copy = -34859,
```



Слика А.1: *За кôд приказан горе лево, текстуални извештај са путањом програма која доводи до грешке, као и вредности променљивих на тој путањи приказан је горе десно. Одговарајући HTML извештај са увећаним делом који указује на грешку, приказан је у доњем делу слике.*

Биографија аутора

Милена Вујошевић Јаничић рођена је 3. јуна 1980. године у Београду, где је завршила основну школу Иван Горан Ковачић и Математичку гимназију. Студије на смеру Рачунарство и информатика на Математичком факултету у Београду уписала је 1999. године. Током студија боравила је на две стручне праксе, на универзитету државе Ајова (САД) и на Политехничком универзитету у Хонг Конгу (Кина), и била је добитник стипендије Краљевског дома Карађорђевића, стипендије Владе Србије, стипендије краљевине Норвешке и стипендије Републичке фондације за развој научног и уметничког подмлатка.

Дипломирала је 2004. године са просечном оценом 9.86 (од 10.00) и уписала магистарске студије на истом смеру. Исте године била је изабрана у звање асистента приправника на Математичком факултету. Магистарски рад под називом „Аутоматско откривање прекорачења бафера у програмском језику C" одбранила је 2008. године. Изабрана је у звање асистента на Математичком факултету 2009. године. Током досадашњег рада на Математичком факулету држала је вежбе из шест предмета.

Бави се истраживањима у области верификације софтвера, пре свега у подобласти аутоматског откривање грешака техникама симболичког извршавања и проверавања модела, као и применама верификацијских техника у образовању. Учествовала је у раду више међународних радионица и летњих школа, и боравила је у истраживачкој посети универзитету EPFL у Лозани (Швајцарска). Објавила је радове на водећим међународним форумима из области верификације софтвера.

Прилог 1.

Изјава о ауторству

Потписани-а Милена Вујошевић Јаничић

број индекса _____

Изјављујем

да је докторска дисертација под насловом

Аутоматско генерисање и проверавање услова исправности програма

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

Потпис докторанда

У Београду, 1. 9. 2013.

Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора Милена Вујошевић Јаничић

Број индекса _____

Студијски програм рачунарство и информатика

Наслов рада Аутоматско генерисање и проверавање услова исправности програма

Ментор проф. Душан Тошић

Потписани/а Милена Вујошевић Јаничић

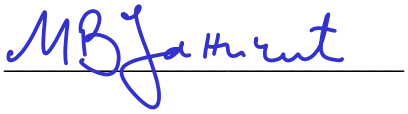
Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис докторанда

У Београду, 1. 9. 2013.



Прилог 3.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Аутоматско генерисање и проверавање услова исправности програма

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ла.

1. Ауторство

2. Ауторство - некомерцијално

3. Ауторство – некомерцијално – без прераде

4. Ауторство – некомерцијално – делити под истим условима

5. Ауторство – без прераде

6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

Потпис докторанда

У Београду, 1. 9. 2013.

МВЈаничић

1. Ауторство - Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. Ауторство – некомерцијално. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. Ауторство - некомерцијално – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

4. Ауторство - некомерцијално – делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. Ауторство – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. Ауторство - делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.