Matematički fakultet, Univerzitet u Beogradu

Formalizacija, implementacija i primene SAT rešavača

doktorska disertacija

Autor:
mr Filip Marić

Mentor:
dr Predrag Janičić

Članovi komisije:
dr Žarko Mijajlović,
Matematički fakultet, Beograd
dr Sava Krstić
Intel Corporation, Portland, SAD
dr Duško Vitas
Matematički fakultet, Beograd
dr Zoran Ognjanović
Matematički institut, Beograd
dr Predrag Janičić
Matematički fakultet, Beograd

oktobar 2006. - jun 2009.

PhD thesis: Formalization, Implementation and Applications of SAT Solvers
Author: Filip Marić (1978 -)

Keywords: SAT problem, SAT solvers, propositional logic, program verification, formal theorem proving, proof assistants, Isabelle, abstract state transition systems, shallow embedding into HOL, object-oriented design, timetabling, SMT problem, link grammars
Pages: 251
References: 80
Area: SAT solving, propositional logic, program verification
PhD Committee: Predrag Janičić (advisor), Žarko Mijajlović, Sava Krstić, Duško Vitas, Zoran Ognjanović
Date: June 11, 2009.
Language: Serbian

Abstract

Background - Software verification and formal theorem proving are introduced. SAT problem is informally defined and techniques used in modern SAT solvers are introduced.
I Formalization - A higher order meta-logic used for all further formalizations of SAT problem and SAT solvers is described. A formalization of propositional logic of CNF formulae is given. Several abstract state transition systems for SAT (basic search, DPLL search, backjumping, learning, restarting, and conflict analysis) are formalized. For each of these systems, soundness, termination, completeness and, therefore, total correctness have been proved.
II Implementation - The implementation of a core of a modern SAT solver is described in an imperative pseudocode whose correctness is proved by using Hoare logic. The classic DPLL procedure and the core of a modern SAT solver are embedded into higher order logic and the correctness of this form of implementation is proved. A flexible object-oriented architecture of a modern SAT solver is presented and the solver ArgoSAT is described.
III Applications - Automated timetabling based on SAT solving is described. The SMT problem is introduced and applications of SAT solvers for its solving are described. The link grammars, used for natural language syntax descriptions, are introduced and possible applications of SAT solvers within link-grammar parsers are given.
Discussion and Conclusions - Related work is discussed and comparison of the results obtained in this thesis with relevant related results is given. Possible directions of further work are given. Final conclusions are drawn.

Naslov disertacije: Formalizacija, implementacija i primene SAT rešavača
Autor: Filip Marić (1978 -)

Ključne reči: SAT problem, SAT rešavači, iskazna logika, verifikacija programa, formalno dokazivanje teorema, pomoćnici za dokazivanje, Isabelle, apstraktni sistemi promena stanja, plitko utapanje u logiku višeg reda, objektno orijentisani dizajn, raspoređivanje časova, SMT problem, link gramatike
Broj strana: 251
Reference: 80
Oblast: rešavanje SAT problema, iskazna logika, verifikacija programa
Komisija: Predrag Janičić (mentor), Žarko Mijajlović, Sava Krstić, Duško Vitas, Zoran Ognjanović
Datum: 11. jun 2009.
Jezik: Srpski

Sažetak

Osnove - Dat je pregled oblasti verifikacije softvera i pregled sistema za formalno dokazivanje teorema. Neformalno je uveden SAT problem i opisane su osnovne tehnike koje se koriste u okviru savremenih SAT rešavača.
I Formalizacija - Dat je opis meta-logike višeg reda u okviru koje je izvršena dalja formalizacija SAT problema i SAT rešavača. Izvršeno je formalno definisanje iskazne logike KNF formula. Izvršena je formalizacija nekoliko različitih apstraktnih sistema promena stanja za SAT (osnovna pretraga, DPLL pretraga, povratni skokovi, učenje, otpočinjanje iznova i analiza konflikata). Za svaki od ovih sistema dokazana je saglasnost, zaustavljanje, potpunost i time i totalna korektnost.
II Implementacija - Dat je opis implementacije jezgra savremenog SAT rešavača u obliku imperativnog pseudokoda čija je korektnost dokazana korišćenjem Horove logike. Dat je opis klasične DPLL procedure i opis jezgra savremenog SAT rešavača utapanjem u logiku višeg reda i izršen je dokaz korektnost ove implementacije. Opisana je fleksibilna objektno-orijentisana arhitektura savremenog SAT rešavača i prikazan je sistem ArgoSAT.
III Primene - Dat je prikaz tehnike automatskog generisanja rasporeda časova korišćenjem SAT rešavača. Dat je opis SMT problema i opisano je kako se savremeni SAT rešavači koriste za njegovo rešavanje. Opisan je formalizam link gramatika koje se koriste za opis sintakse prirodnih jezika i napomenuto je kako je moguće izgraditi parser za ovaj formalizam korišćenjem SAT rešavača.
Diskusija i zaključci - Dat je pregled relevantnih rezultata i izvršeno je poređenje rezultata ove teze sa njima. Navedeni su mogući pravci daljeg rada. Izvedeni su zaključci.

Sadržaj

1 Predgovor

Problem iskazne zadovoljivosti (SAT) je problem ispitivanja da li postoje vrednosti iskaznih promenljivih za koje je data iskazna formula (zadata u konjunktivnoj normalnoj formi) tačna. Na primer, formula $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$ je tačna ako je promenljiva x_1 tačna, a promenljive x_2 i x_3 netačne. SAT problem je prvi za koji je dokazan da je NP-kompletan [Coo71] i on i dalje ima centralno mesto u oblasti složenosti izračunavanja. SAT problem ima i mnoge praktične primene (na primer u verifikaciji softvera i hardvera, automatizaciji dizajna elektronskih kola, veštačkoj inteligenciji, operacionim istraživanjima). Zahvaljujući napretku u oblasti rešavanja SAT problema, SAT rešavači postaju alat koji se koristi za rešavanje sve više i više teških praktičnih problema.

Po načinu rešavanja problema, SAT rešavači se mogu podeliti na (i) potpune i (ii) nepotpune (stohastičke). Za datu iskaznu formulu (instancu SAT problema), potpuni rešavači uvek mogu ili da nađu rešenje (vrednosti promenljivih za koje je formula tačna) ili da pokažu da takvo rešenje ne postoji. Sa druge strane, stohastički rešavači nisu u mogućnosti da pokažu da rešenje ne postoji, ali često veoma efikasno mogu da pronađu zadovoljavajuće vrednosti promenljivih za veoma velike ulazne formule. Većina savremenih potpunih rešavača je zasnovana na Davis-Putnam-Logemann-Loveland-ovoj proceduri (često skraćeno nazivanom DPLL) [DP60, DLL62]. Počevši od rešavača GRASP i SATO i nastavljajući kroz rešavače Chaff, BerkMin i MiniSAT [MMZ+01, GN02, ES04] izuzetan napredak u performansama je postignut u prethodnih nekoliko godina. Zahvaljujući ovim unapređenjima, današnji SAT rešavači su u mogućnosti da rešavaju formule sa desetinama hiljada promenljivih i milionima klauza. Ovaj uspeh se pripisuje:

1. Konceptualnim unapređenjima originalne DPLL procedure čiji je cilj smanjivanje prostora pretrage. Neka unapređenja ovog tipa su *povratni skokovi (eng. backjumping)*, *učenje klauza vođeno analizom konflikata (eng. conflict-driven learning)* i *otpočinjanje iznova (eng. restarting)*.

2. Velikim brojem veoma složenih heurističkih komponenti kao što su na primer *strategije za izbor literala (eng. decision strategy)*, *strategije otpočinjanja iznova (eng. restart strategy)*, *strategije zaboravljanja (eng. forget strategy)*, itd.

3. Boljim tehnikama implementacije, kao što su, na primer, *shema dva posmatrana literala (eng. two-watched literals scheme)*, zatim efikasne strukture podataka za zapis formula i klauza, itd.

S obzirom na to da SAT rešavači postaju veoma složeni sistemi, njihovo precizno opisivanje kao i dokazivanje njihove korektnosti postaje veoma težak zadatak. U nekim radovima, SAT rešavači su prilično apstraktno opisani kao algoritmi visokog nivoa, dok se u nekim radovima može naći opis implementacije sistemskog nivoa sa obiljem efikasnih implementacionih tehnika i trikova. Na žalost, i dalje postoji veliki jaz između ove dve vrste opisa. Prezentacije višeg nivoa, iako često veoma precizne i dopunjene dokazima korektnosti, ne sadrže dovoljno detalja kako bi mogle da se prevedu u realni programski kôd. S druge strane, prezenacije niskog nivoa se obično daju u obliku pseudokoda i najčešće se ne bave pitanjima korektnosti. Veoma često jedna funkcija u pseodokodu sadrži nekoliko algoritama višeg nivoa ili se dešava da je jedan algoritam višeg nivoa razbijen u nekoliko funkcija pseudokoda što kôd čini veoma teškim za razumevanje i održavanje i teško je sprovesti analizu njegove korektnosti.

Ciljevi ove teze su:

1. Opisivanje svih detalja savremenih SAT rešavača na metodičan način koji čitaocima bez predznanja u ovoj oblasti omogućava njihovo razumevanje.
2. Opisivanje apstraktnog formalnog okvira za izgradnju efikasnih i korektnih SAT rešavača.
3. Opisivanje implementacije jezgra SAT rešavača u pseudokodu, tako da kôd prati opis algoritama višeg nivoa i jasno razdvaja koncepte prisutne u okviru rešavača (što ga čini mnogo jednostavnijim za razumevanje, održavanje, modifikaciju i dokazivanje korektnosti), a sadrži sve implementacione tehnike i trikove niskog nivoa (što ga čini efikasnim koliko god je to moguće).
4. Formalna verifikacija korektnosti SAT rešavača opisanih kroz apstraktni formalni okvir i kroz konkretne implementacije.
5. Fleksibilna implementacija realnog savremenog SAT rešavača zasnovanog na opisanom jezgru i objektno-orijentisanoj paradigmi.
6. Demonstracija snage savremenih SAT rešavača, kroz prikaz nekih od postojećih i razvoj nekih novih primena u rešavanju praktičnih problema.

1.1 Doprinosi teze

Originalni doprinosi ove teze prate njene osnovne ciljeve. Deo rezultata ove teze je već objavljen, dok su delovi prijavljeni za objavljivanje ili su u pripremi.

Po prvi put je dat opis implementacije SAT rešavača koja eksplicitno razdvaja koncepte prisutne u savremenoj SAT tehnologiji. Koncepti se uvode postupno, te on može da služi i kao materijal za upoznavanje modernih SAT

rešavača. Ova vrsta implementacije je veoma pogodna i za formalnu verifikaciju. Ovaj opis i skica dokaza korektnosti su objavljeni u časopisu *Journal of Automated Reasoning*, u radu [Mar09a].

Najznačajniji doprinosi teze su na polju verifikacije SAT rešavača. S obzirom da je korektnost SAT rešavača izuzetno značajna, dokazi korektnosti formalnog okvira za izgradnju SAT rešavača moraju biti izvedeni formalno i detaljno. U ovoj tezi je po prvi put je urađena potpuno formalna verifikacija modernih SAT rešavača opisanih veoma precizno i sa jako puno detalja. Celokupna formalizacija je urađena u okviru sistema za dokazivanje teorema Isabelle i korektnost dokaza je mehanički proveriva od strane računara što u ogromnoj meri povećava uverenje u njihovu korektnost. SAT rešavači su opisani na nekoliko različitih načina i svi oni su formalno verifikovani. Po prvi put urađena je potpuno formalna verifikacija klasične DPLL procedure i ona je objavljena u časopisu *Informatica*, u radu [MJ09a]. Urađena je, takođe prvi put, formalna verifikacija apstraktnih sistema promena stanja koji opisuju rad modernih SAT rešavača i nju sadrži rad [MJ09c] koji je u pripremi. Takođe po prvi put, tehnikom plitkog utapanja u logiku višeg reda, izvršena je formalna verifikacija implementacije savremenog SAT rešavača koja uključuje i tehnike implementacije (npr. shema dva posmatrana literala) koje ranije ni u kom obliku nisu bile verifikovane. Ovaj dokaz korektnosti je opisan u radu [Mar09c] koji je prijavljen za objavljivanje. Kompletan Isabelle izvorni kôd formalizacije i sva tri dokaza korektnosti su objavljeni u radu [Mar08], koji je dostupan u okviru kolekcije formalnih dokaza *Archive of Formal Proofs* (`http://afp.sf.net`). Celokupan projekat formalizacije i verifikacije je sažeto prikazan u radu [MJ09b] koji je prijavljen za objavljivanje.

Kao plod rada na ovoj tezi, razvijen je SAT rešavač ArgoSAT koji je izgrađen u potpunosti u skladu sa navednim ciljevima. Rešavač je javno dostupan (veb stranica rešavača ArgoSAT je `http://argo.matf.bg.ac.rs/`). Ovo je jedan od retkih rešavača koji odlikuje fleksibilna objektno-orijentisana arhitektura koja znatno olakšava razumevanje, održavanje i korišćenje koda. Arhitektura insistira na jasnoj i eksplicitnoj razdvojenosti različitih komponenti sistema i omogućava koegzistenciju različitih tehnika (naročito heuristika) u okviru rešavača. Iako postoje rešavači koji koriste rešenja zasnovana na dobrim principima objektno-orijentisanog dizajna, arhitektura predložena u ovoj tezi ide najdalje u ovom pravcu. Predložena arhitektura je opisana i u radu [Mar09b] koji je prijavljen za objavljivanje.

Sistem ArgoSAT je korišćen kao eksperimentalna platforma prilikom istraživanja mogućnosti automatskog izbora između heurističkih politika na osnovu sintaksnih svojstava formule čija se zadovoljivost ispituje i ohrabrujući rezultati ovog istraživanja su prikazani u radu [NMJ09] koji je prijavljen za objavljivanje.

U okviru ove teze je detaljno razvijena tehnika izrade rasporeda časova svođenjem na SAT problem. Ova tehnika je uspešno primenjena, uz korišćenje rešavača ArgoSAT, i do sada je izrađeno desetak rasporeda časova za tri fakulteta i jednu srednju školu u Beogradu. Rad koji opisuje ovu tehniku je prijavljen za objavljivanje [Mar09d].

Sistem ArgoSAT je korišćen i u okviru SMT rešavača ArgoLib. Jedna od primena sistema ArgoLib je u proveri prekoračenja bafera u programima napisanim u programskom jeziku C, i ona je opisana u radu [VJMT08] saopštenom u okviru konferencije *Balcor*.

1.2 Organizacija teze

Osnove. U uvodnom delu teze dat je pregled oblasti verifikacije softvera (glava 2) i pregled sistema za formalno dokazivanje teorema (glava 3). Nakon toga, dat je kratak neformalan pregled klasičnih i modernih SAT tehnologija (glava 4).

Formalizacija. U prvom delu teze (deo *Formalizacija*), prikazana je detaljna formalizacija logike iskaznih formula u konjunktivnoj normalnoj formi (glava 6), kao i formalizacija savremenih DPLL sistema (glava 7). Formalizacija je data u obliku apstraktnih sistema promena stanja i pokriva najviši nivo arhitekture velikog broja savremenih SAT rešavača. U okviru formalizacije, dokazana su mnoga svojstva ovih sistema, među kojima je svakako najvažnija njihova korektnost koja uključuje parcijalnu korektnost i zaustavljanje.

Implementacija. U drugom delu teze (deo *Implementacija*), opisano je nekoliko nivoa i oblika implementacije SAT rešavača koji se direktno oslanjaju na formalni okvir razrađen u prvom delu teze.

1. U okviru glave 9, razvijena je implementacija modernog SAT rešavača u imperativnom pseudojeziku koja pored naprednog DPLL jezgra uključuje veliki broj heuristika, kao i implementacionih tehnika i trikova prisutnih u savremenim SAT rešavačima. Takođe, korišćenjem Horove logike izvršena je verifikacija napisanog pseudokoda.

2. U okviru glave 10, razvijene su implementaciji SAT rešavača direktno u okviru logike višeg reda sistema Isabelle (koja je za ovu priliku posmatrana kao čist funkcionalni programski jezik). SAT rešavači su verifikovani tehnikom plitkog utapanja u logiku višeg reda, tj. predstavljeni su u obliku skupa rekurzivnih funkcija i njihova korektnost je dokazana uglavnom korišćenjem indukcije i jednakosnog rezonovanja. Pristup je iskorišćen kako za verifikaciju klasične DPLL procedure (poglavlje 10.1), tako i za proveru savremenog SAT rešavača analognog rešavaču opisanom u imperativnom pseudojeziku (poglavlje 10.2). Na osnovu definicija rešavača u okviru logike, automatski je generisan kôd rešavača u nekoliko savremenih funkcionalnih programskih jezika, i za njega se sa najvećom mogućom merom pouzdanosti može tvrditi da je korektan.

3. U okviru glave 11, razvijena je implementacija SAT rešavača korišćenjem objektno-orijentisane paradigme, koja je takođe zasnovana na verifikovanom DPLL jezgru. Definisana je modularna arhitektura kôda zasnovana na projektnim obrascima (eng. design patterns) i dokumentovana je odgovarajućim UML dijagramima kao i fragmentima realnog C++ koda.

Primene. U trećem delu teze (deo *Primene*), u cilju demonstriranja moći savremenih SAT rešavača, opisano je nekoliko njihovih praktičnih primena. U glavi 13, opisana je tehnika izgradnje rasporeda časova korišćenjem svođenja na SAT problem i korišćenjem SAT rešavača. U glavi 15, opisana je tehnika ispitivanja zadovoljivosti formula u odnosu na teoriju prvog reda (eng. satisfiability modulo theory - SMT).

Diskusija i zaključci. U glavi 16, dat je pregled relevantnih rezultata. Prikazana je kratka istorija razvoja SAT rešavača, rezultati na polju verifikacije SAT i SMT rešavača kao i pregled različitih opisa rešavača. U glavi 17 dat je kratak pregled mogućih pravaca za dalji rad u ovoj oblasti. U glavi 18, navedena su završna razmatranja i diskutovano je na koji način ova teza ispunjava ciljeve zacrtane na početku rada na njoj.

1.3 Zahvalnost

Želeo bih ovom prilikom da se zahvalim svom mentoru dr Predragu Janičiću koji me je uveo u problematiku automatskog rezonovanja i koji me je nesebično vodio tokom celokupnog trajanja mojih poslediplomskih studija. Profesorove sugestije i veoma detaljne recenzije svih mojih radova su mi značajno pomogle da ovladam tehnikama naučno-istraživačkog rada, a zahvaljujući profesorovoj prijatnost i neposrednosti u ličnom kontaktu ni u jednom trenutku se nisam osećao kao student već kao njegov ravnopravan saradnik i, mogu slobodno reći, prijatelj. Takođe, želeo bih da se zahvalim svim profesorima i kolegama sa Katedre za računarstvo, a naročito profesoru dr Dušku Vitasu koji je svojim toplim prijateljskim pristupom pokazao beskrajno poverenje u mene kao mlađeg kolegu i asistenta. Pored ovoga, želeo bih da se zahvalim i članovima komisije dr Žarku Mijajloviću, dr Savi Krstiću i dr Zoranu Ognjanoviću. Profesoru Mijajloviću dugujem zahvalnost i zbog činjenice da me je prvi uputio na sistem Isabelle koji se pokazao veoma važnim u mom naučnom radu. Naročita čast mi je to što je profesor Krstić pristao da bude član komisije za pregled i ocenu ove teze. Rad profesora Krstića na formalizaciji SAT i SMT rešavača predstavlja jedan od temelja na kojem je ova teza izgrađena. Takođe, želim da zahvalim dr Zoranu Ognjanoviću i kolegama sa Matematičkog instituta u Beogradu što su nam ustupili računarske resurse potrebne za naša istraživanja. Želim da se zahvalim i mladom kolegi Mladenu Nikoliću koji je bio glavni korisnik SAT rešavača korišćenog u okviru ove teze i čije su dragocene sugestije i eksperimenti u značajnoj meri doprineli kvalitetu razvijenog sistema.

Posebnu zahvalnost dugujem svojoj porodici — mojoj Mileni i mojoj maloj Ani koja je sa svojih godinu dana učinila proces nastajanja ove teze veoma neobičnim i zanimljivim. Želim ovu tezu da posvetim svojoj prerano preminuloj mami Brani, koja me je izvela na put kojim trenutno hodim i naučila da je život najveći poklon koji imamo i da i u doba najvećih problema možemo pronaći nešto čemu ćemo se radovati i što će nam pomoći da prevaziđemo prepreke.

Osnove

2
Verifikacija softvera

U ovoj glavi će biti opisani osnovni pojmovi verifikacije softvera. Verifikacija softvera je iz dva razloga izrazito značajni aspekt ove teze: (i) u okviru teze se obrađuju SAT rešavači koji se kao alati koriste u okviru verifikacije softvera i (ii) u okviru teze se tehnikama verifikacije softvera dokazuje korektnost samih SAT rešavača.

Iako često ne obraćamo pažnju na to, softver je u današnjem svetu prisutan na svakom koraku: softver kontroliše mnogo toga — od daljinskih upravljača za televizore, automobila, i bankovnih računa, do nuklearnih elektrana, aviona i svemirskih letelica. Uz sav taj softver, neminovno su prisutne i greške nastale tokom njegovog razvoja. Greška u funkcionisanju daljinskog upravljača može biti uznemirujuća, ali greška u funkcionisanju nuklearne elektrane može biti razorna. Eksplozija rakete *Ariane (fr. Ariane 5)* 1996. uzrokovana konverzijom broja iz šezdesetčetvorobitnog realnog u šesnaestobitni celobrojni zapis koja je dovela do prekoračenja, zatim greška u numeričkom koprocesoru procesora Pentium 1994. uzrokovana pogrešnim indeksima u for petlji u okviru softvera koji je radio dizajn čipa i pad orbitera poslatog na Mars 1999. kako bi merio klimu uzrokovan činjenicom da je deo softvera koristio metričke, a deo softvera engleske jedinice su katastrofe koje se godinama prepričavaju i postale su deo folklora. Međutim, fatalne sofverske greške se i dalje neprestano javljaju i u elektronskim medijima se lako pronalazi pregršt novih primera softverskih grešaka koje koštaju svetsku ekonomiju milijarde dolara. Evo nekih najzanimljivijih:

- Ne naročito opasan, ali veoma zanimljiv primer greške je greška u programu *Microsoft Excel 2007* koji, zbog greške u algoritmu formatiranja brojeva pre prikazivanja, rezultat izračunavanja izraza $77.1 * 850$ prikazuje kao $100,000$ (iako je interno korektno sačuvan).

- 14. Septembra 2004. godine, preko četiristo aviona je istovremeno izgubilo vezu sa kontrolom leta u blizini aerodroma u Los Anđelesu (eng. Los Angeles). Na sreću, zahvaljujući rezervnoj opremi unutar samim aviona, do nesreće nije došlo. Uzrok gubitka veze je bila greška prekoračenja u brojaču milisekundi u okviru sistema za komunikaciju sa avionima. Da ironija bude veća, ova greška je bila ranije otkrivena, ali, pošto je do otkrića došlo kada je već sistem bio isporučen i instaliran na nekoliko

aerodroma, njegova jednostavna popravka i zamena nije bila moguća. Umesto toga, preporučeno je da se sistem resetuje svakih 30 dana kako do prekoračenja ne bi došlo. Procedura nije ispoštovana i greška se javila posle tačno 2^{32} milisekundi, odnosno 49.7 dana od uključivanja sistema.

- Pad satelita *Kriosat (eng. Cryosat)* 2005. koštao je Evropsku Uniju oko sto trideset i pet milona evra. Pad je uzrokovan greškom u softveru zbog koje nije na vreme došlo do razdvajanja satelita i rakete koja ga je nosila.

- Više od pet procenata penzionera i primalaca socijalne pomoći u Nemačkoj je privremeno ostalo bez svog novca kada je 2005. godine uveden novi računarski sistem. Greška je nastala zbog toga što je sistem, koji je zahtevao desetocifreni zapis svih brojeva računa, kod starijih računa koji su imali osam ili devet cifara brojeve dopunjavao nulama, ali sa desne strane umesto sa leve kako je pravilno.

- Dell i Apple su tokom 2006. godine morali da korisnicima zamene više od pet miliona laptop računara zbog greške u dizajnu baterije kompanije Sony koja je uzrokovala da se nekoliko računara zapali.

Verifikacija softvera je proces dokazivanja da računarski program ispunjava uslove svoje specifikacije (koja opisuje očekivano ponašanje programa). Učeći lekcije iz većih softverskih krahova, veliki napor se ulaže u istraživanja na ovom polju i verifikacija softvera je jedna od najznačajnijih oblasti računarstva. Iako se potpuno korektan softver s razlogom smatra nedostižnim idealom, povećanje poverenja u softver je zadatak mnogih današnjih istraživanja. Puno rada je uloženo i u formalizaciju semantike programskih jezika, kompilatora, protokola za komunikaciju, bezbednosnih protokola itd.

Dva osnovna pristupa verifikaciji su *dinamička verifikacija* koja podrazumeva proveru korektnosti u fazi izvršavanja programa, najčešće putem *testiranja* i *statička verifikacija* koja podrazumeva analizu programskog koda korišćenjem formalnih metoda i matematičkog aparata. U nastavku će biti razmatrana samo formalna verifikacija softvera.

Rani rezultati formalne verifikacije softvera potiču još iz 1950-tih godina. Među pionirima u ovom polju su bili Alan Tjuring (eng. Alan Turing), Džon fon Nojman (eng. John von Neumann) i Džon Makarti (eng. John McCarthy). Kasnih 1960-tih godina, Rober Flojd (eng. Robert Floyd) uvodi jednakosno rezonovanje na dijagramima toka, a Entoni Hor (eng. Anthony Hoare) uvodi Horovu logiku (eng. Hoare logic) — vrstu aksiomatske semantike programskih jezika. Sve do ranih 1980-tih, Flojd-Horov pristup je bio dominantan pristup verifikaciji i oba autora su za svoj rad nagrađena prestižnom Tjuringovom nagradom. Dokazana je korektnost mnogih fundamentalnih algoritama i struktura podataka, ali je verifikacija uglavnom rađena ručno i uključivala je manipulaciju veoma kompleksnim formulama, tako da je postepeno postalo preovlađujuće mišljenje da na ovaj način nije moguće (ili bar nije praktično) izvršiti verifikaciju realnih programskih sistema, bez sumnji u korektnost samog procesa verifikacije. Kao odgovor na ovaj problem, javlja se automatizacija procesa verifikacije.

Jedan od pristupa automatizovane verifikacije je korišćenje automatskih dokazivača teorema koji bilo potpuno automatski ili interaktivno pomažu u procesu dokazivanja uslova korektnosti. Kako bi se postigao najveći mogući

stepen pouzdanosti, dokazi korektnosti moraju biti takvi da ih specijalizovani programi za proveru dokaza mogu automatski mehanički proveriti. Mnoge rane rezultate u automatskoj verifikaciji su napravili Robert Bojer (eng. Robert Boyer) i Džej Stroter Mur (eng. J Strother Moore) korišćenjem svog znamenitog dokazivača. Dokazivači teorema koji se najčešće koriste za verifikaciju softvera danas su Isabelle, HOL, Coq, PVS, Nuprl, itd. Pošto je ovaj pristup dominantno zastupljen u ovoj tezi, o njemu će više biti reči u nastavku.

Drugi pristup, automatskoj verifikaciji korišćen od 1980-tih godina, se zasniva na tehnici poznatoj kao *provera modela (eng. model checking).* [CE82, Eme08, Cla08] koju su razvili Edmund Klark (eng. Edmund Clarke), Alen Emerson (eng. Allen Emerson) i Džozef Sifakis (eng. Joseph Sifakis) i koji zu za to (isto kao svojevremeno Hor i Flojd) nagrađeni prestižnom Tjuringovom nagradom 2007. godine. Proveravanje modela podrazumeva potpuno automatsku proveru da li uprošćeni model programa odgovara njegovoj formalnoj specifikaciji. Programi se obično predstavljaju kao sistemi sa konačno mnogo stanja (eng. finite state machines). Svojstva programa koja se obično proveravaju su uslovi bezbednosti kao što je odsustvo kružnih blokada (eng. deadlock) i sličnih stanja koji dovode do kraha sistema. Logike u kojima se obično formulišu svojstva programa su najčešće temporalne logike (LTL, CTL).

3
Sistemi za formalno dokazivanje teorema

Formalno dokazivanje teorema podrazumeva iskazivanje matematičkih tvrđenja na striktno definisanom formalnom jeziku i njihovo dokazivanje u preciznom logičkom okviru. Zbog kompleksnosti formalnog dokazivanja teorema, razvijeni su mnogi softverski sistemi koji potpomažu ovaj proces. Uloga ovih sistema je dvojaka. S jedne strane, oni kontrolišu korektnost dokaza, kontrolišući osnovanost svakog pojedinačnog izvođenja. S druge strane, oni pokušavaju da olakšaju proces dokazivanja automatizujući proces pronalaženja dokaza ili bar njihovih delova. Izgradnja potpuno automatizovanih dokazivača je često izuzetno teška (čak i nemoguća zbog neodlučivosti odgovarajućih teorija), dok je manuelno formalno dokazivanje teorema uz korišćenje računara samo za proveru korektnosti dokaza izuzetno dugotrajan i mukotrpan posao. Iz ovog razloga, današnji sistemi obično uspostavljaju balans između ova dva ekstrema. Naime, iako je razvijen veliki broj *potpuno automatskih* metoda dokazivanja teorema ovi dokazivači su uglavnom za jednostavnije logike i njihove fragmente. Sa druge strane, veliki broj dokazivača teorema koji radi sa bogatijim logikama (na primer, logikom višeg reda) je *poluautomatski (eng. semi-automatic)* i zahteva određeni stepen interakcije sa korisnikom tokom procesa dokazivanja. U potpuno automatizovane dokazivače teorema, moguće je ubrojati i SAT rešavače, zatim SMT rešavače koji su u mogućnosti da rezonuju o odlučivim teorijama formulisanim u okviru logike prvog reda (o njima će više biti rečeno u glavi 15), zatim dokazivače zasnovane na jednakosnom rezonovanju (eng. equational reasoning) i tehnikama prezapisivanja (eng. rewriting), zatim mnoge dokazivače zasnovane na metodu rezolucije koji rezonuju o valjanosti opštih formula logike prvog reda i slično. Poluautomatski dokazivači se često još nazivaju i *asistenti u dokazivanju teorema (eng. proof assistants)* i o njima će više biti reči u nastavku.

U današnje vreme, mnoge značajne matematičke teorije su formalizovane u okviru nekog od sistema za formalno dokazivanje teorema. Pregledni rad [Har08] detaljnije opisuje ovu oblast. U najznačajnije postignute rezultate do danas, mogu se ubrojati formalizacija Teoreme o prostim brojevima (eng. Prime number theorem) koju je uradio Džeremi Avigad (eng. Jerremy Avigad), zatim dokaz teoreme o obojivosti grafa sa četiri boje (eng. Four color theorem) koji je konstruisao Žorž Gontije (fr. Georges Gonthier) i izrada formalnog dokaza Keplerove konjekture (eng. Kepler conjecture) koja je u toku i koja se radi u

	Ime sistema	Autori
1.	HOL Light, HOL4, ProofPower	John Harrison, Konrad Slindt
2.	Mizar	Andrzej Trybulec
3.	PVS	Bart Jacobs, John Rushby
4.	Coq	Laurent Théry
5.	Otter/Ivy	Michael Beeson, William McCune
6.	Isabelle	Markus Wenzel, Larry Paulson
7.	Alfa/Agda	Thierry Coquand
8.	ACL2	David Russinoff, Ruben Gamboa
9.	PhoX	Christophe Raffalli, Paul Rozière
10.	IMPS	William Farmer
11.	Metamath	Norman Megill
12.	Theorema	Wolfgang Windsteiger, Bruno Buchberger, Markus Rosenkranz
13.	Lego	Conor McBride
14.	Nuprl	Paul Jackson
15.	Omega	Christoph Benzmüller, Armin Fiedler, Andreas Meier, Martin Pollet
16.	B method	Dominique Cansell
17.	Minlog	Helmut Schwichtenberg

Tabela 3.1: Sistemi poređeni u [Wie06]

okviru projekta Flyspeck čiji je rukovodilac Tomas Hejls (eng. Thomas Hales) — do sada je, na primer, dokazana Žordanova teorema (eng. Jordan curve theorem).

Savremeni asistenti za dokazivanje teorema U uporednoj studiji [Wie06] izvršena je uporedna analiza sedamnaest različitih sistema za formalno dokazivanje teorema. Sistemi koje studija obuhvata su navedeni u tabeli 3.1. Ovo su ujedno i sistemi koji se danas najčešće i najšire upotrebljavaju. Studija je pravljena tako što je od autora ili istaknutih korisnika ovih sistema zatraženo da u svom sistemu sprovedu formali dokaz jednog matematičkog tvrđenja (teoreme o iracionalnosti broja $\sqrt{2}$), i da zatim popune upitnik o različitim karakteristikama svog sistema. Iako prilično heterogena, ova studija pomaže da se stekne uvid u osnovne osobine svakog od analiziranih sistema. S obzirom da je u izradi ove teze korišćen isključivo dokazivač Isabelle, u nastavku će biti prikazan njegov opis, dok se za opis karakteristika ostalih nabrojanih dokazivača čitalac upućuje na studiju [Wie06] kao i na dokumentaciju samih sistema.

LCF pristup izgradnji dokazivača teorema. Jedan od najuticajnijih poluautomatskih dokazivača teorema u istoriji bio je dokazivač LCF. Pošto je i dokazivač Isabelle u značajnoj meri zasnovan na dokazivaču LCF, u nastavku će biti dat kratak prikaz razvoja dokazivača LCF i biće opisani principi koje je LCF uveo, a koji su prisutni u velikom broju savremenih dokazivača (materijal je uglavnom preuzet iz [Gor00]).

U svojoj originalnoj verziji (poznatoj kao *Stanfordski LCF*) koju je na Univerzitetu Stanford razvio Robin Milner (eng. Robin Milner) sa koautorima 1972. godine, ovaj dokazivač je vršio interaktivno generisanje i proveru formalnih dokaza teorema o izračunljivim funkcijama nad raznim domenima značajnim u računarstvu (npr., celim brojevima, listama, semantici programskih jezika). Dokazivač je pomagao korisnicima tako što je mogao da raščlanjuje tvrđenja koja se dokazuju na jednostavnija podtvrđenja i da neke od njih automatski

dokaže korišćenjem mehanizma uprošćavanja. Korišćeni formalni logički okvir je logika izračunljivih funkcija (eng. logic of computable functions) koju je uveo Dejna Skot (eng. Dana Scott).

Razvoj dokazivača je nastavljen u Edinburgu, i dokazivač poznat kao *Edinburški LCF* se pojavio oko 1977. godine. Ovaj dokazivač je uveo novinu koja će značajno uticati na dalji razvoj dokazivača. Naime, ograničenja Stanfordskog LCF sistema su bila to što je (i) veličina dokaza bila ograničena raspoloživom memorijom računara i (ii) fiksni skup komandi koje je moguće koristiti u dokazima nije bilo jednostavno proširiti. Kako bi se ograničenje (i) prevazišlo, Milnerova ideja je bila da se umesto pamćenja celokupnih dokaza pamte samo konačni rezultati tj. formulacije dokazanih teorema (slično kao što predavač na tabli nakon izvedenog dokaza tvrđenja kako bi uštedeo prostor može da obriše dokaz i ostavi samo teoremu koju koristi u daljem radu). Kako bi se osiguralo da su teoreme koje su na ovaj način prisutne u sistemu zaista i dokazane, Milner je sproveo narednu brilijantnu ideju koja se danas naziva *LCF pristup (eng. LCF approach)*. Naime, u okviru funkcionalnog programskog jezika u kome se dokazivač implementira (jezik ML u slučaju sistema LCF) definiše se apstraktni tip podataka *thm* koji predstavlja teoreme. Predefinisane vrednosti ovog tipa odgovaraju aksiomama, a operacije odgovaraju pravilima izvođenja logičkog sistema. Striktna tipiziranost programskog jezika zatim obezbeđuje da se teoreme koje su prisutne u sistemu (instance tipa *thm*) zaista grade od aksioma primenom pravila izvođenja. Nove kompleksnije taktike dokazivanja se zatim mogu slobodno programirati u okviru implementacionog funkcionalnog programskog jezika, pri čemu je saglasnost obezbeđena njegovom striktom tipiziranošću jezika i saglasnošću jezgra učaurenog u apstraktni tip *thm*.

Razvoj dokazivača LCF je nastavljen tokom prve polovine 1980-tih godina kroz saradnju sa Univerzitetom u Kembridžu i značajan doprinos razvoju projekta su dali Lari Polson (eng. Larry Paulson) i Žerar Huet (fr. Gérard Huet). Dokazivač je značajno unapređen kroz proširenja i bolje implementacije i prikazana je njegova praktična upotrebljivost kroz mnogobrojne primene.

Na principima koje je uveo LCF, izgrađeni su mnogi savremeni dokazivači teorema. Navedimo na primer familiju dokazivača koju je započeo Majk Gordon (eng. Mike Gordon) dokazivačem HOL, a koja uključuje i komercijalni dokazivač ProofPower i dokazivač HOL Light koja je razvio Džon Harison (eng. John Harrison). Tu su zatim Nuprl koji je razvio Konstable (eng. Constable) sa saradnicima, Coq koji su razvili Tijeri Kokan (fr. Thierry Coquand) i Žerar Huet (fr. Gérard Huet) sa saradnicima i mnogi drugi.

3.1 Isabelle

Isabelle je asistent za dokazivanje teorema koji se razvija od sredine 1980-tih godina. Iako je mnogo istraživača doprinelo razvoju ovog sistema najznačajniji njegovi autori su Lari Polson (eng. Larry Paulson) sa Univerziteta u Kembridžu i Tobijas Nipkov (nem. Tobias Nipkow) i Markus Vencel (nem. Markus Wenzel) sa Tehničkog univerziteta u Minhenu. Isabelle se najčešće primenjuje kao alat za formalizovanje čisto matematičkih teorija, kao i za formalnu verifikaciju softvera i hardvera koja uključuje dokazivanje svojstava programskih sistema, jezika i protokola o čemu je već bilo reči u glavi 2. Isabelle sistem obuhvata veoma bogatu biblioteku formalno verifikovanih matematičkih teorija, koja uključuje i elementarnu teoriju brojeva (npr. Gausov (nem. Carl

Friedrich Gauß) zakon kvadratnih reciprociteta), matematičku analizu (osnovne osobine graničnih vrednosti, izvoda i integrala), algebru (do Silovljeve (nor. Peter Sylow) teoreome), i teoriju skupova (uključujući svojstva aksiome izbora). Takođe, uključeni su i mnogi primeri koji dolaze iz problema programske verifikacije. Veliki broj formalizacija i matematike i računarstva je dostupan kroz *Arhiv formalnih dokaza (eng. Archive of Formal Proofs)*[1].

Isabelle [NPW02] je *generički* dokazivač. U doba kada je sistem nastajao, dokazivači su, prateći LCF pristup, obično tvrdo kodirali logiku sa kojom rade kodirajući njena pravila izvođenja kao funkcije implementacionog programskog jezika. Ovim je dokazivač bio tesno vezan za objektnu logiku (tj. formalni sistem) koju podržava i nije bilo moguće isti dokazivač koristiti za rezonovanje u drugim formalnim sistemima. Isabelle je generički sistem što znači da je dokazivač koji pruža podršku za različite formalne sisteme. Ovo je ostvareno tako što Isabelle implementira *meta-logiku* poznatu kao $\mathcal{M}$ ili Isabelle/Pure [Pau89] u okviru koje se zatim definišu razne objektne logike. Ova meta-logika predstavlja fragment intuicionističke logike višeg reda sa implikacijom ($\Longrightarrow$), univerzalnim kvantifikatorom ($\bigwedge$) i jednakošću ($\equiv$).[2] Pravila meta-logike se mogu kombinovati primenjujući unifikaciju višeg reda i rezoluciju. Objektne logike u kojima se zatim efektivno vrše formalizacije se definišu u okviru meta-logike uvođenjem novih tipova, konstanti i aksioma. Sa sintaksnog stanovišta, niske simbola objektne logike se predstavljaju niskama simbola meta logike. Sa semantičkog stanovišta, potrebno je razmatrati semantiku meta-logike gde svaki tip označava skup, svaki term označava element odgovarajućeg skupa i gde logičke oznake imaju svoja uobičajena značenja. Ukoliko objektne logike imaju definisanu semantiku, njihova reprezentacija u okviru meta-logike čuva tu semantiku [Pau89]. Ovim, Isabelle može da služi i kao sistem za brz i efikasan razvoj različitih logičkih deduktivnih sistema. Prilikom definisanja meta-logike, Isabelle prati LCF pristup kako bi se obezbedila korektnost sistema. Sistem je izgrađen nad programskim jezikom ML, međutim, kranji korisnici uglavnom nemaju potrebe za direktnom interakcijom sa samim programskim jezikom već sa sistemom komuniciraju na specijalizovanom jeziku višeg nivoa. Sa druge strane, napredniji korisnici imaju mogućnost da u okviru jezika ML implementiraju različita proširenja samog sistema Isabelle i mnoge dodatne pakete, procedure odlučivanja i slične alate.

Neke od podržanih objektnih logika su:

HOL - logika višeg reda

HOLCF - Skotova logika izračunljivih funkcija u logici višeg reda.

FOL - Klasična i intuicionistička logika prvog reda.

ZF - Cermelo-Frenkelova (eng. Zermelo-Fraenkel) teorija skupova izgrađena u okviru FOL.

LCF - Skotova logika izračunljivih funkcija u logici prvog reda.

Sequents - Sekventi za logiku prvog reda, modalnu i linearnu logiku.

[1] `http://afp.sf.net`

[2] Nestandardne oznake su korišćene kako bi objektne logike mogle da koriste standardne.

CTT - Konstruktivna teorija tipova (ekstenzionalna verzija teorije tipova Martin-Lofa (šve. Martin-Löf).

Najkorišćenija logika sistema Isabelle danas je logika višeg reda (eng. Higher Order Logic) i ovaj dokazivač se naziva *Isabelle/HOL*. Ovaj sistem uključuje napredne načine zadavanja specifikacija, npr. bogat skup tipova podataka, podršku za kreiranje induktivnih definicija, podršku za definisanje funkcija primitivnom i opštom rekurzijom, itd.

U nastavku ćemo navesti neke od ključnih karakteristika savremenog sistema Isabelle.

- U svojim novijim verzijama, dokazi u sistemu Isabelle se zapisuju u specijalizovanom jeziku *Isar*. Isar je deklarativni jezik i on omogućava da napisani dokazi budu razumljivi i pogodni za čitanje kako računarima tako i ljudima.
- Kako bi se ubrzao proces dokazivanja, sistem Isabelle uključuje veliki broj algoritama za automatsko dokazivanje teorema. Među ovima su najpoznatiji klasični dokazivač koji se zasniva na klasičnom rezonovanju, simplifikator koji vrši jednakosno rezonovanje, zatim specijalizovane procedure odlučivanja za linearnu aritmetiku, itd.
- Apstraktne specifikacije je moguće zadavati kroz sistem modula (poznatih kao *locale*) ili putem klasa tipova (eng. *type classes*).
- Isabelle pruža izuzetnu podršku za matematičku notaciju i uobičajena notacija može biti korišćena u okviru zapisa teorija. Takođe, moguće je uključivati i fragmente zapisane u jeziku LaTeX i na osnovu ovoga generisati dokumente u obliku pogodnom za štampu.
- Isabelle/HOL pruža mogućnost da se na osnovu izvršnih specifikacija generiše kôd u jezicima SML, OCaml i Haskell.
- Dokumentacija koja opisuje korišćenje sistema Isabelle i njegovih koncepata je veoma bogata i javno dostupna. Na ovom polju vredi istaknuti tutorijal [NPW02].
- Isabelle sistem je tesno integrisan sa sistemom *Emacs* kroz sistem *Proof-General* specijalizovan za formalno dokazivanje teorema.

4
SAT problem i SAT rešavači

U ovoj glavi će neformalno biti uveden pojam problema iskazne zadovoljivosti (SAT problema) i biće dat pregled postojećih procedura koje se koriste za njegovo rešavanje. U osnovi svih ovih procedura je *sistematična pretraga (eng. search)* koja na neki način nabraja sva potencijalna rešenja (u ovom slučaju valuacije). Ovakvi algoritmi se obično realizuju kao algoritmi *pretrage sa povratkom (eng. backtrack search)*. Pretraga se dopunjuje različitim mehanizmima *zaključivanja (eng. inference)* koja služe da redukuju prostor pretrage i da odbace valuacije za koje se, iz nekog razloga, zna da ne mogu biti potencijalna rešenja. Prebacivanje težišta algoritama sa komponente pretrage na komponente zaključivanja često dovodi do znatno efikasnijih SAT rešavača.

4.1 SAT problem - notacija i osnovni pojmovi

U ovom poglavlju će na prilično neformalan biti uvedeni neki od osnovnih pojmova koji se koriste prilikom formulisanja i rešavanja SAT problema. Formalna definicija svih ovih pojmova u okviru strogo definisanog meta-logičkog okvira će biti data u delu *Formalizacija*.

U okviru definicija osnovnih logičkih pojmova, koristićemo konačne sekvence elemenata koje ćemo nazivati *listama (eng. list)*[1]. Pripadnost listi ćemo označavati skupovnom oznakom $\in$. Slično, $l_1 \subseteq l_2$ označava da je svaki element liste l_1 ujedno i element liste l_2. Uklanjanje svih pojavljivanja elementa e iz liste l će biti označeno sa $l \setminus e$.

Pretpostavlja se da su formule čija se zadovoljivost ispituje u *konjunktivnoj normalnoj formi (KNF) (eng. conjunctive normal form (CNF))*. Iskazne formule u KNF su konjunkcije članova koji se nazivaju klauze, a koji predstavljaju disjunkcije (eventualno negiranih) promenljivih. Na primer,

$$(x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4 \vee x_5) \wedge (x_3).$$

[1] Za konačne konačne sekvence elemenata često se koristi termin *niz*, a termin *lista* se često vezuje za dinamičke strukture podataka. Korišćenje termina *lista* za konačne sekvence je preuzeto iz sistema Isabelle koji je korišćen kao alat u okviru koga je izvršena celokupna formalizacija.

Pretpostavljamo da je dat prebrojiv skup *promenljivih (eng. variable)*. *Literal (eng. literal)* je ili promenljiva ili njena negacija. *Suprotan literal (eng. opposite literal)* literala l označavamo sa $\overline{l}$. Ukoliko je literal l promenljiva x_i, njemu suprotan literal je negacija promenljive $\neg x_i$, a ukoliko je literal l negacija promenljive $\neg x_i$, njemu suprotan literal je sama promenljiva x_i. Liste literala nazivamo *klauzama (eng. clause)*. Liste klauza nazivamo *KNF formulama (eng. CNF formula)*. Pošto ćemo razmatrati jedino KNF formule, obično ćemo koristiti samo kraći termin *formula*.

Pod *valuacijama (eng. valuation)* ćemo podrazumevati liste literala. Valuacija je *neprotivrečna (eng. consistent)* ukoliko ne sadrži dva međusobno suprotna literala. Primetimo da svaka neprotivrečna valuacija jednoznačno određuje preslikavanje nekog skupa promenljivih u skup istinitosnih vrednosti (npr. u skup $\{0, 1\}$).[2]

Literal l je *tačan u valuaciji* v (što označavamo sa $v \vDash l$) ukoliko je njen član, a l je *netačan u valuaciji* v (što označavamo sa $v \vDash \neg\, l$) ukoliko je njemu suprotan literal $\overline{l}$ njen član. Klauza c je *tačna u valuaciji* v (što označavamo sa $v \vDash c$) ukoliko je bar jedan njen literal tačan u valuaciji v, a *netačna u valuaciji* v (što označavamo sa $v \vDash \neg\, c$) ukoliko su svi njeni literali netačni u v. Formula F je *tačna u valuaciji* v (što označavamo sa $v \vDash F$) ukoliko su joj sve klauze tačne u valuaciji v, a *netačna u valuaciji* v ukoliko joj je bar jedna klauza netačna u valuaciji v. Pošto pojam valuacije obuhvata parcijalne valuacije u odnosu na neki unapred dat skup promenljivih, istinitosna vrednost literala (klauza, formula) može da bude i nedefinisana u datoj valuaciji.

Za valuaciju kažemo da je *model (eng. model)* formule (klauze, literala) ukoliko je neprotivrečna i ukoliko je formula (klauza, literal) u njoj tačna. Formula je *zadovoljiva* akko ima model. *SAT problem* predstavlja problem ispitivanja da li KNF formula zadovoljiva.

Formula F *povlači (eng. entails)* formulu F' (klauzu c, literal l) (što označavamo sa $F \vDash F'$ ($F \vDash c$, $F \vDash l$) ukoliko je svaki model formule F ujedno i model formule F' (klauze c, literala l). U ovom slučaju kažemo i da je formula F' (klauza c, literal l) *semantička posledica* formule F. Dve formule su *logički ekvivalentne (eng. equivalent)* ukoliko je svaki model jedne formule ujedno i model druge i obratno. Dve formule su *ekvizadovoljive (eng. equisatisfiable)* ukoliko je prva formula zadovoljiva ako i samo ako je zadovoljiva i druga.

4.2 Načini opisivanja SAT rešavača

Iako je većina SAT rešavača zasnovana na klasičnoj DPLL proceduri definisanoj još 1960-tih godina, napretkom SAT tehnologija, algoritmi i strukture podataka koji su u osnovi SAT rešavača postaju sve kompleksniji i njihova prezentacija postaje veoma zahtevan zadatak. U materijalima u kojima se opisuju SAT rešavači, koriste se najčešće ili opisi u vidu pseudokoda ili opisi u vidu apstraktnih sistema promena stanja.

[2] Iako klasični udžbenici logike pojam valuacije definišu upravo kao ovakva preslikavanja, ove definicije ne obuhvataju redosled literala u okviru valuacije što je izuzetno značajan koncept za SAT rešavanje, te je zbog toga odabrana definicija valuacije kao liste literala.

4.2.1 Opisi u vidu pseudokoda.

Opisivanje algoritama korišćenjem pseudokoda je klasičan pristup koji koji se i dalje koristi. On podrazumeva korišćenje (obično imperativnog) pseudojezika koji poseduju većinu osobina realnih programskih jezika u kojima se rešavači implementiraju. Prednosti korišćenja ovog načina opisivanje je to što su opisi često prilično detaljni i veoma su bliski realnim implementacijama. Međutim, iako je ovaj način prezentacije najčešće korišćen, on ima mnoge nedostatke:

1. Korišćenjem pseudokoda ne može se napraviti razlika između suštine algoritma i njegove implementacije (tj. ovim pristupom postiže se nizak nivo apstrakcije).

2. Korektnost algoritama u formi pseudokoda se komplikovano dokazuje. Dokazi korektnosti ovako izloženih algoritama se retko sreću u literaturi, a kada ih ima, oni su najčešće neformalni i neprecizni. Kako je programska paradigma najčešće imperativna, za formalni dokaz korektnosti je neophodno koristiti formalni okvir kakav je na primer Horova (eng. Hoare) logika.

4.2.2 Opisi u vidu apstraktnih sistema promena stanja.

Tokom prethodnih nekoliko godina objavljeno je nekoliko radova u kojima su SAT rešavači zasnovani na DPLL algoritmu opisani u vidu *apstraktnih sistema promena stanja (eng. abstract state transition systems)*. Prednost ovog pristupa je u tome da su SAT rešavači opisani kao matematički objekati o kojima se može jednostavnije formalno rezonovati. Ovi opisi potpuno i precizno opisuju najviši nivo arhitekture SAT rešavača. Osnovni nedostatak ovih opisa je da su obično veoma apstraktni i udaljeni od nivoa konkretne implementacije.

Ovim opisima se rad rešavača predstavlja kao niz prelazaka između *stanja (eng. state)*. Za prelaske između stanja, mogu se koristiti isključivo data *pravila prelaska (eng. transition rules)*. Svako pravilo definiše (i) uslove koji određuju da li je pravilo moguće primeniti i (ii) efekat pravila koji određuje način na koji pravilo menja stanje na koje se primenjuje. Svaki skup pravila prelaska definiše *relaciju prelaska (eng. transition relation)* definisanu na skupu stanja. Za stanje se kaže da je *završno (eng. final)* ukoliko nijedno od pravila ne može da se na njega primeni. Rešavači počinju izvršavanje od *početnog stanja (eng. initial state)* koje je određeno formulom čija se zadovoljivost ispituje i koriste isključivo postojeća pravila prelaska. Neka stanja ukazuju na to da je polazna formula zadovoljiva i njih nazivamo *prihvatajućim (eng. accepting)* stanjima, dok neka ukazuju da ona to nije i njih nazivamo *odbacujućim (eng. rejecting)* stanjima. Za ovako definisane apstraktne sisteme promene stanja se obično dokazuju svojstva *saglasnosti (eng. soundness)*, *zaustavljanja (eng. termination)* i *potpunosti (eng. completeness)*.

1. *Svojstvo saglasnosti*: ako sistem stigne u stanje koje ukazuje da je formula zadovoljiva (nezadovoljiva), onda formula zaista jeste zadovoljiva (nezadovoljiva).

2. *Svojstvo zaustavljanja*: ako se krene iz bilo kog početnog stanja u konačnom broju koraka se stiže do nekog završnog stanja.

3. *Svojstvo potpunosti*: svako završno stanje ukazuje ili da je formula zadovoljiva ili da je nezadovoljiva.

4. *Svojstvo korektnosti*: za svaku formulu se iz njom određenog početog stanja stiže u završno stanje koje ukazuje da je formula zadovoljiva (nezadovoljiva) ako i samo ako ona zaista jeste zadovoljiva (nezadovoljiva).

Naglasimo na ovom mestu da se ovako definisana potpunost i saglasnost sistema pravila donekle razlikuje od definicija potpunosti i saglasnosti koje se mogu sresti u literaturi. Naime, često se pod pojmom saglasnosti podrazumeva isključivo saglasnost za nezadovoljive formule (koja garantuje da ako sistem dođe u nezadovoljivo stanje, polazna formula je nezadovoljiva), a pod pojom potpunosti se podrazumeva isključivo potpunost za nezadovoljive formule (koja garantuje da se za svaku nezadovoljivu formulu može doći u stanje koje je nezadovoljivo). U tom slučaju, pošto formule mogu biti isključivo ili zadovoljive ili nezadovoljive, uz pretpostavku zaustavljanja, na osnovu saglasnosti za nezadovoljive formule direktno sledi potpunost za zadovoljive formule i na osnovu potpunosti za nezadovoljive formule direktno sledi saglasnost za zadovoljive formule. Za razliku od klasičnog pristupa, pristup koji je u ovoj tezi dat, može jednostavno da se uopšti i na sisteme koji ne predstavljaju procedure odlučivanja, tj. daju više različitih odgovora. U svetlu ovog tumačenja, saglasnost za pojedinačnu vrstu odgovora ukazuje na to da ukoliko taj odgovor dat, on je zaista i ispravan odgovor. Dakle, saglasnost obezbeđuje da sistem nikada ne daje pogrešan odgovor. Potpunost obezbeđuje da sistem za svaki ulaz (pod pretpostavkom zaustavljanja) daje neki od mogućih odgovora, odnosno da ne postoji mogućnost da se sistem zaustavi a da nije dao nikakav odgovor. Primetimo da u našem sistemu svojstvo korektnosti direktno sledi iz svojstava saglasnosti, zaustavljanja i potpunosti. Zaista, jedan smer korektnosti je saglasnost, dok je drugi smer svojstvo potpunosti za svaku pojedinačnu vrstu odgovora (koje govori da će za svaku zadovoljivu (nezadovoljivu) sistem završiti u zadovoljivom (nezadovoljivom) stanju) i ono sledi iz svojstva saglasnosti za sve vrste odgovora, zaustavljanja i pokrivenosti. Takođe, iz svojstva potpunosti za sve vrste odgovora sledi svojstvo zaustavljanja, pokrivenosti i saglasnosti za svaku pojedinačnu vrstu odgovora.

4.3 Klasična DPLL procedura

Većina potpunih SAT rešavača se direktno zasniva na modifikacijama *klasične DPLL procedure*. Početak razvoja ove procedure se vezuje za Dejvis-Putnamov algoritam koji su 1960. razvili Martin Dejvis (eng. Martin Davis) i Hilari Putnam (eng. Hilary Putnam) [DP60]. Dve godine kasnije, 1962., Martin Dejvis, Džordž Logeman (eng. George Logemann) i Donald Loveland (eng. Donald W. Loveland) uvode poboljšanu verziju procedure, u kojoj zamenju tzv. *pravilo eliminacije (eng. elimination rule)* tzv. *pravilom grananja (eng. split rule)* i danas se ova novija verzija algoritma naziva DPLL algoritam [DLL62]. DPLL algoritam se primenjuje na iskazne formule u konjunktivnoj normalnoj formi. Dok pravilo eliminacije problem čija se zadovoljivost ispituje svodi na jedan, često veći, problem, pravilom grananja se problem čija se zadovoljivost ispituje svodi na dva manja potproblema (po jedan za svaku istinitosnu vrednost izabrane promenljive). Osnova DPLL procedure je *sistematični*

algoritam pretrage sa vraćanjem (eng. backtrack search). Pri tome se za smanjivanje prostora pretrage koriste dve tehnike: jedna je zasnovana na korišćenju svojstava tzv. *jediničnih klauza (eng. unit clauses)*, a druga na korišćenju svojstava tzv. *literali jedinistvenog polariteta (eng. pure literals)*. Složenost DPLL algoritma je eksponencijalna u odnosu na veličinu ulazne formule [].

4.3.1 Opis u vidu pseudokoda

Osnovna komponenta DPLL algoritma je pretraga sa povratkom (eng. backtrack search) prikazana na slici 4.1.

```
function search (F : Formula) : (SAT, UNSAT)
begin
  if F je prazna then
    return SAT
  else if postoji prazna klauza u F then
    return UNSAT
  else begin
    odaberi literal l koji se javlja u F
    if DPLL(F[l → ⊤]) = SAT then return SAT
    else if DPLL(F[l → ⊥]) = SAT then return SAT
    else return UNSAT
  end
end
```

Slika 4.1: Klasični algoritam pretrage

Kada se ovaj algoritam pretrage proširi komponentama zaključivanja na osnovu jediničnih klauza i klauza jedinstvenog polariteta, dobija se klasična DPLL procedura. Njena neformalna rekurzivna formulacija je data na slici 4.2 (F je KNF formula (skup iskaznih klauza) čija se zadovoljivost ispituje, dok $F[l \rightarrow \top]$ označava formulu koja se od F dobija kada se literal l zameni sa $\top$, a literal $\overline{l}$ sa $\bot$, nakon čega se vrši uprošćavanje). Prikazana rekurzivana implementacija je praktično neupotrebljiva za veće formule, zbog toga što zahteva kreiranje nove uprošćene formule i njeno prosleđivanje tokom svakog rekurzivnog poziva, što je izrazito zahtevno s obzirom na to da formule mogu biti jako veliki objekti.

4.3.2 Opis u vidu sistema promena stanja

U ovom i narednom poglavlju ćemo pokazati kako je korišćenjem pravila prelaska moguće modelovati mnoge SAT algoritme zasnovane na upravo opisanoj klasičnoj DPLL proceduri, počevši od najjednostavnijih pa sve do onih koji se koriste u savremenim SAT rešavačima. Način na koji ćemo zapisivati pravila prelaska prati sistem koji su uveli Krstić i Goel u [KG07].

Ime pravila:

$$\frac{uslov_1, \quad \ldots, \quad uslov_k}{efekat}$$

```
function DPLL (F : Formula) : (SAT, UNSAT)
begin
  if F je prazna then
    return SAT
  else if postoji prazna klauza u F then
    return UNSAT
  else if postoji literal l koji se javlja u F,          ZAKLJUČIVANJE
        takav da se l̄ ne javlja u F then
    return DPLL(F[l → ⊤])
  else if F sadrži jediničnu klauzu [l] then
    return DPLL(F[l → ⊤])
  else begin                                             PRETRAGA
    odaberi literal l koji se javlja u F
    if DPLL(F[l → ⊤]) = SAT then
      return SAT
    else
      return DPLL(F[l → ⊥])
  end
end
```

Slika 4.2: Klasična DPLL procedura - neformalna rekurzivna formulacija

Iznad linije su navedeni uslovi $uslov_1, \ldots, uslov_k$ koji moraju biti ispunjeni da bi pravilo moglo da se primeni, a ispod linije je naveden način na koji primena pravila menja stanje procedure.

Osnovna pretraga

Ova procedura po svojoj suštini odgovara naivnoj search proceduri 4.3.1, ali implementacija je u potpunosti drugačija. Za razliku od implementacije date na slici 4.1 gde se u svakom rekurzivnom pozivu vrši eksplicitna izmena tekuće formule u skladu sa valuacijom koja je samo implicitno poznata, savremeni SAT rešavači tokom procesa rešavanja eksplicitno čuvaju tekuću parcijalnu valuaciju. U toj valuaciji, moguće je razlikovati dve vrste literala — *pretpostavljene literale (eng. decision literals)* i *izvedene literale (eng. implied literals)*. Pretpostavljeni literali se obično u literaturi označavaju uspravnom crtom ispred njih (na primer $|\, l$) ili malim slovom d (na primer l^d) ili oznakom $\top$ (na primer $l^\top$) i u nastavku će biti korišćene sve tri oznake. Valuacije sa označenim pretpostavljenim literalima ćemo nazivati *označene valuacije (eng. assertion trail)*. Pošto je redosled literala u označenim valuacijama bitan za proces rešavanja, označene valuacije će biti predstavljene listama označenih literala. Stanje rešavača je određeno tekućom označenom parcijalnom valuacijom M i formulom F čija se zadovoljivost ispituje. Parametar procedure je skup literala L zatvoren za negaciju, koji je najšešće izgrađen nad skupom promenljivih (vars F) koje se javljaju u okviru polazne formule F.

Najjednostavniji sistem pravila opisuje proceduru koja koristi samo pretragu. Ovaj sistem je moguće opisati korišćenjem sledeća dva pravila:

Decide:
$$\frac{l \in L, \quad l, \overline{l} \notin M}{M := M \,|\, l}$$

Backtrack[3]:
$$\frac{M = M' \,|\, l\, M'', \quad (\text{decisions } M'') = [\,], \quad M \models_{\neg} F}{M := M'\ \overline{l}}$$

Pravilo Decide dodaje u označenu valuaciju proizvoljni literal l koji nije do tada u njoj definisan, a koji pripada nekom unapred fiksiranom skupu literala L (obično su to oni literali koji se javljaju u polaznoj formuli). Literal l se dodaje kao pretpostavljeni literal. Kaže se da je u tom slučaju izvršeno *grananje (eng. branching)* na osnovu literala l.

Pravilo Backtrack se primenjuje kada se ustanovi da tekuća formula F netačna u valuaciji M. Tada kažemo da je došlo do *konflikta (eng. conflict)*, a klauze formule F koje su netačne u M nazivamo *konfliktnim klauzama (eng. conflict clauses)*. U tom slučaju se poslednji pretpostavljeni literal u valuaciji M i svi literali iza njega uklanjaju iz valuacije M i nakon toga se njegova negacija se postavlja u valuaciju M, ali ovaj put kao izvedeni literal.

Kao što je već rečeno, završna stanja su ona u kojima nije moguće primeniti nijedno pravilo. Ukoliko u završnom stanju važi $M \models_{\neg} F$, formula F je nezadovoljiva, dok je u suprotnom formula zadovoljiva i M je njen model.

Redosled primene pravila nije definisan i biće pokazano da korektnost procedure ne zavisi od redosleda primene pravila. Međutim, bolja efikasnost procedure se dobija ukoliko se pravilu Backtrack da veći prioritet nego pravilu Decide, odnosno kada se pravilo Backtrack primenjuje kad god su ispunjeni uslovi za njegovu primenu.

Primer 4.1. Neka je $F = [[-1, 2], [-3, 4], [-1, -3, 5], [-2, -4, -5], [-2, 3, 5, -6], [-1, 3, -5, -6], [1, -6], [1, 7]]$. Formula je predstavljena listom klauza, svaka klauza je lista literala, a literali su označeni brojevima[4]. U tabeli 4.1 su prikazani koraci koje rešavač zasnovan samo na pravilima Decide i Backtrack primenjuje prilikom ispitivanja zadovoljivosti F. U tabeli su prikazana stanja kroz koje rešavač prolazi (tj. tekuće parcijalne valuacije) i pravila koja se primenjuju. Pošto nije navedena konkretna strategija izbora literala, ovo je samo jedan od mnogo mogućih načina rešavanja formule F.

Pošto se Backtrack primenjuje samo u situaciji konflikta, uz svaki njen poziv je navedena konfliktna klauzu, tj. klauzu koja je netačna u tekućoj valuaciji M. Primetimo da izbor konfliktne klauze nije jedinstven. U poslednjem navedenom stanju je skup promenljivih koje se javljaju u formuli F (obeležen sa $(\text{vars } F)$) isti kao skup promenljivih koje se javljaju u valuaciji M, a važi $M \not\models_{\neg} F$, tako da je formula zadovoljiva i M je njen model.

Jedinične klauze

Prva važna optimizacija procesa pretrage koju uvodi klasični DPLL algoritam je korišćenje *jediničnih klauza (eng. unit clauses)*. Naime, osnovna pretraga ne koristi nikakve elemente zaključivanja kako bi bila pronađena zado-

[3] Oznaka $(\text{decisions } M'') = [\,]$, označava da u M'' ne postoje pretpostavljeni literali

[4] Ovaj način predstavljanja je odabran jer direktno odgovara najčešćim načinima implementacije SAT rešavača

Primenjena funkcija	sat?	M
	UNDEF	[]
Decide (l = -6)	UNDEF	$[\vert -6]$
Decide (l = 1)	UNDEF	$[\vert -6, \vert 1]$
Decide (l = 3)	UNDEF	$[\vert -6, \vert 1, \vert 3]$
Decide (l = -4)	UNDEF	$[\vert -6, \vert 1, \vert 3, \vert -4]$
Backtrack ($M \models\neg [-3, 4]$)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4]$
Decide (l = 2)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4, \vert 2]$
Decide (l = -5)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4, \vert 2, \vert -5]$
Backtrack ($M \models\neg [-1, -3, 5]$)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4, \vert 2, 5]$
Backtrack ($M \models\neg [-2, -4, -5]$)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4, -2]$
Backtrack ($M \models\neg [-1, 2]$)	UNDEF	$[\vert -6, \vert 1, -3]$
Decide (l = -2)	UNDEF	$[\vert -6, \vert 1, -3, \vert -2]$
Backtrack ($M \models\neg [-1, 2]$)	UNDEF	$[\vert -6, \vert 1, -3, 2]$
Decide (l = 7)	UNDEF	$[\vert -6, \vert 1, -3, 2, \vert 7]$
Decide (l = 4)	UNDEF	$[\vert -6, \vert 1, -3, 2, \vert 7, \vert 4]$
Decide (l = 5)	UNDEF	$[\vert -6, \vert 1, -3, 2, \vert 7, \vert 4, \vert 5]$
Backtrack ($M \models\neg [-2, -4, -5]$)	UNDEF	$[\vert -6, \vert 1, -3, 2, \vert 7, \vert 4, -5]$
$M \not\models\neg F$, (vars M) = (vars F)	SAT	$[\vert -6, \vert 1, -3, 2, \vert 7, \vert 4, -5]$

Tabela 4.1: Rad rešavača u primeru 4.1

voljavajuća valuacija. Za klauzu kažemo da je jedinična u odnosu na neku valuaciju ukoliko ima tačno jedan literal nedefinisan u v koji nazivamo *jedinični literal*, dok su joj svi ostali literali netačni u valuaciji v. Da bi jedinične klauze bile tačne, potrebno je valuaciju proširiti njihovim jediničnim literalom. Za literale koji su dodani u valuaciju kao na osnovu neke jedinične klauzu, kažemo da su rezultat *propagacije jediničnih klauza (eng. unit propagation)*. Korišćenje jediničnih klauza kao mehanizma zaključivanja dovodi do ogromnog smanjivanja prostora pretrage.

Na primer, u primeru 4.1 je napravljena pretpostavka -4, iako je literal 3 bio element tekuće parcijalne valuacije, a formula F je sadržala klauzu $[-3, 4]$. Kako bi ova klauza bila tačna, literal 4 mora da bude tačan u trenutku kada je literal 3 deo parcijalne valuacije, i poželjno ga je dodati kao izvedeni literal odmah nakon trenutka u kome je 3 dodan u valuaciju. Ukoliko je literal 3 element valuacije M, a ni 4 ni -4 nisu, tada važi da je $[-3, 4]$ jedinična klauza u odnosu na M sa jediničnim literalom 4.

U sistemu sa progacijom jediničnih klauza se pored pravila Decide i Backtrack koristi se i naredno pravilo.

UnitPropagate:

$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F, \quad \overline{l}_1, \ldots, \overline{l}_k \in M, \quad l, \overline{l} \notin M}{M := M\ l}$$

Primer 4.2. Neka je $F = [[-1, 2], [-3, 4], [-1, -3, 5], [-2, -4, -5], [-2, 3, 5, -6], [-1, 3, -5, -6], [1, -6], [1, 7]]$. Jedan od mogućih načina rada rešavača je prikazan u tabeli 4.2.

Literali jedinistvenog polariteta

Druga veoma važna optimizacija koju uvodi klasični DPLL algoritam je korišćenje literala koji se pojavljuju samo u jednom polaritetu. Korišćenje literala jedinstvenog polariteta se zasniva na činjenici da se nakon dodavanja literala

Primenjeno pravilo	sat?	M
	UNDEF	[]
Decide (l = 6)	UNDEF	$[\vert 6]$
UnitPropagate (c = $[1,-6]$, l = 1)	UNDEF	$[\vert 6, 1]$
UnitPropagate (c = $[-1,2]$, l = 2)	UNDEF	$[\vert 6, 1, 2]$
Decide (l = 7)	UNDEF	$[\vert 6, 1, 2, \vert 7]$
Decide (l = 3)	UNDEF	$[\vert 6, 1, 2, \vert 7, \vert 3]$
UnitPropagate (c = $[-3,4]$, l = 4)	UNDEF	$[\vert 6, 1, 2, \vert 7, \vert 3, 4]$
UnitPropagate (c = $[-1,-3,5]$, l = 5)	UNDEF	$[\vert 6, 1, 2, \vert 7, \vert 3, 4, 5]$
Backtrack ($M \models\neg [-2,-4,-5]$)	UNDEF	$[\vert 6, 1, 2, \vert 7, -3]$
UnitPropagate (c = $[-2,3,5,-6]$, l = 5)	UNDEF	$[\vert 6, 1, 2, \vert 7, -3, 5]$
Backtrack ($M \models\neg [-1,3,-5,-6]$)	UNDEF	$[\vert 6, 1, 2, -7]$
Decide (l = 3)	UNDEF	$[\vert 6, 1, 2, -7, \vert 3]$
UnitPropagate (c = $[-3,4]$, l = 4)	UNDEF	$[\vert 6, 1, 2, -7, \vert 3, 4]$
UnitPropagate (c = $[-1,-3,5]$, l = 5)	UNDEF	$[\vert 6, 1, 2, -7, \vert 3, 4, 5]$
Backtrack ($M \models\neg [-2,-4,-5]$)	UNDEF	$[\vert 6, 1, 2, -7, -3]$
UnitPropagate (c = $[-2,3,5,-6]$, l = 5)	UNDEF	$[\vert 6, 1, 2, -7, -3, 5]$
Backtrack ($M \models\neg [-1,3,-5,-6]$)	UNDEF	$[-6]$
Decide (l = 1)	UNDEF	$[-6, \vert 1]$
UnitPropagate (c = $[-1,2]$, l = 2)	UNDEF	$[-6, \vert 1, 2]$
Decide (l = 7)	UNDEF	$[-6, \vert 1, 2, \vert 7]$
Decide (l = 3)	UNDEF	$[-6, \vert 1, 2, \vert 7, \vert 3]$
UnitPropagate (c = $[-3,4]$, l = 4)	UNDEF	$[-6, \vert 1, 2, \vert 7, \vert 3, 4]$
UnitPropagate (c = $[-1,-3,5]$, l = 5)	UNDEF	$[-6, \vert 1, 2, \vert 7, \vert 3, 4, 5]$
Backtrack ($M \models\neg [-2,-4,-5]$)	UNDEF	$[-6, \vert 1, 2, \vert 7, -3]$
Decide (l = 4)	UNDEF	$[-6, \vert 1, 2, \vert 7, -3, \vert 4]$
UnitPropagate (c = $[-2,-4,-5]$, l = -5)	UNDEF	$[-6, \vert 1, 2, \vert 7, -3, \vert 4, -5]$
$M \not\models\neg F$, (vars M) = (vars F)	SAT	$[-6, \vert 1, 2, \vert 7, -3, \vert 4, -5]$

Tabela 4.2: Rad rešavača u primeru 4.2

koji se javlja samo u jednom polaritetu u formuli F u tekuću valuaciju M i to kao izvedenog literala, zadovoljivost formule F ne menja.

U primeru 4.2 formula F sadrži literal 7, a ne sadrži literal -7. Pošto je cilj rešavača da zadovolji sve klauze, dodavanje literala 7 u valuaciju je poželjna operacija jer se tako zadovoljavaju sve one klauze koje sadrže literal 7. Pošto formula ne sadrži literal -7, time se ni u jednoj drugoj klauzi ne smanjuje mogućnost da eventualno postane zadovoljena kada se izgradi konačna valuacija.

U opisima rešavača preko sistema pravila, ponekad se pojavljuje i naredno pravilo.

PureLiteral:

$$\frac{l \in F, \quad \overline{l} \notin F, \quad l, \overline{l} \notin M}{M := M\ l}$$

Pravilo PureLiteral detektuje literale jedinstvenog polariteta u formuli F koji do tada nisu definisani u valuaciji M i dodaje ih u tekuću valuaciju kao izvedene literale. Primetimo da je ovo pravilo znatno slabije od korišćenja literala jedinstvenog polariteta u rekurzivnoj definiciji DPLL funkcije opisanoj u poglavlju 4.3.1. Naime, u rekurzivnoj implementaciji formula se svakim rekurzivnim pozivom uprošćavala, što daje mogućnost pojave mnogo više literala jedinstvenog polariteta. S druge strane, pravilo PureLiteral u rešavaču kod kojeg se formule ne menjaju tokom rešavanja ima smisla koristiti isključivo na početku rešavanja. Iz ovog razloga, pravilo PureLiteral se u modernim rešavačima koristi isključivo u fazi pretprocesiranja formule i ne koristi se tokom rada rešavača.

Praksa pokazuje da je detekcija literala jedinstvenog polariteta veoma skupa operacija i da korišćenje ovog pravila vodi neefikasnijim rešavačima. Zbog toga u daljem razmatranju nećemo koristiti ovo pravilo.

4.4 Unapređenja klasične DPLL procedure

U ovom poglavlju će korišćenjem sistema pravila biti opisana unapređenja osnovne DPLL procedure koja su prisutna u većini savremenih SAT rešavača.

4.4.1 Učenje i zaboravljanje

Učenje

Ukoliko se pažljivo razmatra rad rešavača u primeru 4.2 može se primetiti da je u nekoliko koraka pretpostavljen literal 3 nakon što je literal 1 već bio element valuacije. Niz koraka koji su izvedeni nakon toga pokazuju da ova dva literala nisu međusobno kompatibilna, tj. ne mogu istovremeno biti članovi nekog modela formule. Pretpostavka 3 je morala biti izmenjena zbog toga što se iz prisustva literala 1 u valuaciji moglo zaključiti da literal -3 mora da bude deo valuacije. Primetimo da se ovo dogodilo u dva različita konteksta (i pod pretpostavkom 6 i pod pretpostavkom -6). Činjenica da 1 i 3 nisu kompatibilni literali se može predstaviti klauzom $[-1, -3]$ koja predstavlja logičku posledicu formule F. Ukoliko bi ova klauza bila deo formule F, ona bi učestvovala u procesu propagacije jediničnih klauza i na osnovu nje bi bilo moguće zaključiti neophodnost literala -3 odmah nakon dodavanja literala 1 u valuaciju. Mehanizam *učenja (eng. clause learning)* omogućava rešavačima da tokom procesa rešavanja prošire skup klauza koje sačinjavaju formulu (redundantnim) klauzama koje su posledice formule F.

Učenje se opisuje narednim pravilom.

Learn:
$$\frac{F \vDash c, \quad c \subseteq L}{F := F \mathbin{@} c}$$

Uslov $F \vDash c$ zahteva da je klauza c logička posledica formule F i on obezbeđuje da je nova formula logički ekvivalentna[5] sa polaznom. Uslov $c \subseteq L$ zahteva da su svi literali u klauzi koja se uči sadržani u skupu literala L i time obezbeđuje da se pravilom Learn ne uvode literali koji se ne javljaju u skupu L. Ovaj uslov je bitan kako bi se obezbedilo zaustavljanje procedure.

Pravilo ne specifikuje način na koji se klauza c određuje. Najčešće je slučaj da se uče klauze koje su određene kao rezultat analize konflikta o čemu će više biti reči u poglavlju 4.4.2.

Zaboravljanje

Tokom rada rešavača sa učenjem, broj klauza tekuće formule raste i formula se uvećava. Ukoliko formula postane jako velika, ispitivanje da li je ona u skladu sa tekućom parcijalnom valuacijom (tj. provera uslova $M \vDash \neg F$) i

[5] Ukoliko se samo ispituje zadovoljivost formule, dovoljno je da zahtevati da je nova formula ekvizadovoljiva sa polaznom, međutim, ukoliko se zahteva i izgradnja modela u slučaju da je formula zadovoljiva, potrebno je zahtevati logičku ekvivalentnost.

pronalaženje jediničnih klauza postaje jako sporo. Zato je potrebno s vremena na vreme uklanjati neke (redundantne) klauze iz F. Neophodno je da važi da je formula nakon uklanjanja klauza ekvivalentna polaznoj formuli. Rešavači obično ovaj uslov obezbeđuju tako što se uklanjaju isključivo naučene klauze. Klauze početne formule se obično ne uklanjaju. Zaboravljanje suvišnih klauza se može opisati narednim pravilom.

Forget:
$$\frac{F \setminus c \models c}{F := F \setminus c}$$

Uslov $F \setminus c \models c$ obezbeđuje da je klauza koja se uklanja zaista redundantna.

4.4.2 Povratni skokovi i analiza konflikata

Ogromni napredak u razvoju SAT rešavača je napravljen kada je klasično vraćanje tokom pretrage (eng. backtracking) zamenjeno naprednijim *povratnim skokovima vođenim analizom konflikata (conflict-driven backjumping)*.

Povratni skokovi

Ukoliko se opet analizira rad rešavača u primeru 4.2, može se videti da je niz koraka koji je primenjen nakon što je pretpostavljen literal 7 pokazao da ni 3 ni -3 nisu u skladu sa prethodnim odlukama. Zbog toga, je Backtrack operacijom zaključeno da mora da važi literal -7. Nakon toga, potpuno identičan niz koraka je izvršen koji je pokazao da, opet, ni 3 ni -3 nisu u skladu sa prethodnim pretpostavkama. Na kraju je Backtrack operacijom zaključeno da je pretpostavka 6 bila pogrešna i da literal -6 mora da važi. Pažljiva analiza koraka koji su doveli do nekonzistentnosti može da pokaže da je promenljiva 7 u potpunosti irelevantna za nastali konflikt i da je zbog toga ponavljanje procesa kada je 7 zamenjeno sa -7 bilo nepotrebno i predstavljalo čist gubitak vremena.

Ova redundantnost je posledica činjenice da pravilo Backtrack uvek poništava samo poslednju pretpostavku koja je napravljena, bez obzira na istinski razlog koji je doveo do nekonzistentnosti. Povratni skokovi predstavljaju vid naprednijeg, tzv. nehronološkog povratka u pretrazi i oni dozvoljavaju rešavačima da poništе nekoliko pretpostavki odjednom, sve do poslednjeg pretpostavljenog literala koji je zaista učestvovao u konfliktu. Ovim se izbegavaju nepotrebne redundantnosti u procesu rešavanja.

Povratni skokovi su obično vođeni klauzom koju ćemo nazivati *klauzom povratnog skoka (eng. backjump clause)* i označavati sa C. Ova klauza je semantička posledica formule F i poželjno je da tačno odgovara promenljivima koje su dovele do konflikta. Kada se konstruiše klauza povratnog skoka, literali koji se nalaze pri vrhu tekuće označene valuacije M se uklanjaju, sve dok klauza povratnog skoka ne postane jedinična klauza u odnosu na tu tekuću valuaciju. Od tog trenutka, njen jedinični literal se postavlja na vrh valuacije M čime se konflikt razrešava i nakon toga proces pretrage se nastavlja uobičajenim tokom.

Povratni skokovi se realizuju korišćenjem narednog pravila.

Backjump:
$$\frac{C = l \vee l_1 \vee \ldots \vee l_k, \quad F \models C, \quad M = M' \,|\, d\, M'', \quad \overline{l}_1, \ldots, \overline{l}_k \in M', \quad \overline{l} \in M''}{M := M'\ l}$$

Pravilo Backjump je moguće primeniti kada je klauza povratnog skoka C takva da postoji pretpostavljeni literal d u M takav da se iznad njega u M nalazi tačno jedan literal $\overline{l}$ suprotan literalu l klauze C, dok se svi ostali literali suprotni literalima klauze C nalaze u M ispod njega. Pravilo Backjump poništava literal d i sve literale iznad njega i u valuaciju dodaje literal l. Broj pretpostavljenih literala ispred literala d se naziva *nivo povratnog skoka (eng. backjump level)*. Uslov da je d pretpostavljeni literal je bitan samo kako bi se obezbedilo zaustavljanje.

Primetimo da se klasični povratak (eng. backtracking) može smatrati specijalnim slučajem povratnog skoka (eng. backjumping). U tom slučaju se klauza povratnog skoka gradi od literala koji su suprotni svim pretpostavljenim literalima u valuaciji M. Ta klauza postaje jedinična klauza tačno u trenutku kada se poslednji pretpostavljeni literal ukloni iz M i na osnovu nje se može zaključiti da mora da njemu suprotni literal mora da bude deo valuacije. Ovako napravljena klauza povratnog skoka, naravno, ne oslikava na pravi način konflikt koji se dogodio.

Analiza konflikata

Da bi klauze povratnog skoka istinski oslikavale konflikt koji se razrešava i time dovele do najvećeg mogućeg odsecanja prostora pretrage, one se konstruišu u procesu koji je poznat pod imenom *analiza konflikta (eng. conflict analysis)*.

Proces analize konflikata se nekada opisuje korišćenjem pristupa zasnovanog na grafovima i klauze povratnog skoka se tada dobijaju obilaskom tzv. grafa zaključivanja (implication graph) [ZMMM01]. Proces analize konflikata se nekad opisuje i kao proces rezolucije koji kreće od konfliktne klauze i nastavlja se sa klauzama koje su uzroci propagacije konfliktnih literala [ZM02].

Postoji nekoliko strategija za analizu konflikata [SS96, ZMMM01, ZM02]. Njihove varijacije uključuju razne načine izbora klauze koja vodi povratni skok. Ipak, najveći broj strategija za analizu konflikata se zasniva na sledećoj tehnici:

1. Proces analize konflikta započinje sa konfliktnom klauzom (tj. sa klauzom formule F koja je postala netačna u M). Klauza analize konflikata C se u tom trenutku inicijalizuje na konfliktnu klauzu.

2. Svaki literal sadržan u tekućoj klauzi analize konflikata C je netačan u tekućoj valuaciji M i predstavlja ili pretpostavku procedure pretrage ili rezultat neke propagacije. Za svaki propagirani literal l moguće je pronaći klauzu c koja je bila uzrok propagacije literala l. Ove klauze se zovu *razlozi propagacije (reason clauses)*. Propagirani literali iz klauze C se onda zamenjuju (govorićemo *objašnjavaju (explain)*) ostalim literalima iz klauza koji predstavljaju razloge njihove propagacije. Ovim se nastavlja proces analize konflikata.

Analiza konflikata se može opisati korišćenjem naredna dva pravila.

Conflict:

$$\frac{l_1 \vee \ldots \vee l_k \in F, \quad \overline{l}_1, \ldots, \overline{l}_k \in M}{C := l_1 \vee \ldots \vee l_k}$$

Explain:
$$\frac{\overline{l} \in C, \quad l \vee l_1 \vee \ldots \vee l_k \in F, \quad \overline{l}_1, \ldots, \overline{l}_k \prec^M l}{C := (C \setminus \overline{l}) \vee l_1 \vee \ldots \vee l_k}$$

Pravilo Conflict odgovara prvom koraku, a pravilo Explain odgovara drugom koraku gore opisane procedure. Faza analize konflikta prestaje primenom povratnog skoka. Uslov $\overline{l}_1, \ldots, \overline{l}_k \prec^M l$ označava da literali $\overline{l}_1, \ldots, \overline{l}_k$ prethode literalu l u valuaciji M.

Opisana procedura se ponavlja sve dok se ne ispuni neki od uslova za završetak analize kada se klauza analize konflikata proglašava klauzom povratnog skoka. Uslovi završetka moraju biti definisani na način koji obezbeđuje da klauza C zaista zadovoljava uslove potrebne za povratni skok.

Kao što je već pomenuto, postoji nekoliko različitih strategija i one određuju različite uslove za završetak procesa analize konflikata. U nastavku ćemo koristiti samo jednu koja je poznata pod nazivom *prva tačka jedinstvene implikacije (eng. first unique implication point (firstUIP))*, pošto se ona koristi u većini vodećih SAT rešavača i pošto se ona na većini testova pokazala superiornijom u odnosu na ostale strategije. Ukoliko se koristi firstUIP strategija, proces analize konflikata se prekida u prvom trenutku u kome klauza C postane klauza povratnog skoka odnosno čim se dogodi da iza poslednjeg pretpostavljenog literala u M postoji tačno jedan literal klauze C.

Napomenimo još da je veoma čest slučaj da se klauze povratnog skoka uče korišćenjem pravila Learn, ne bi li se u budućnosti sprečilo ponavljanje iste vrste konflikta.

U narednom primeru prikazujemo rad rešavača zasnovanog na pravilima Decide, UnitPropagate, Conflict, Explain, Learn i Backjump.

Primer 4.3. Neka je $F = [[-1, 2], [-3, 4], [-1, -3, 5], [-2, -4, -5], [-2, 3, 5, -6], [-1, 3, -5, -6], [1, -6], [1, 7]]$. Jedno od mogućih izvršavanja je prikazano u tabeli 4.3.

Kao što smo već pomenuli, analiza konflikata se može posmatrati i kao proces rezolucije. Stabla rezolucije koja odgovaraju analizama konflikata iz primera 4.3 su prikazana na slici 4.3.

Minimalizacija klauze povratnog skoka

U nekim slučajevima, klauza povratnog skoka koja se dobije postupkom firstUIP analize konflikata se može naknadno minimalizovati. Klauze sa manjim brojem literala dovode do većeg kresanja prostora pretrage i vode bržoj propagaciji jediničnih klauza. Nekoliko različitih tehnika minimalizacije je predloženo u literaturi. Kao ilustraciju prikazaćemo *obuhvatne rezolucije (eng. subsumption resolution)* uvedene u rešavaču MiniSAT [ES04]. U primeru 4.3, primenom firstUIP analize konflikta je dobijena klauza $[-1, -2, -3]$. Međutim, pošto je literal 2 posledica literala 1 (na osnovu klauze $[-1, 2]$), moguće je još jednom primeniti pravilo Explain (tj. uraditi još jedan korak rezolucije) i dobiti manju klauzu $[-1, -3]$. Kažemo da klauza $[-1, -2, -3]$ *obuhvata (eng. subsumes)* klauzu $[-1, -3]$. Osnovna varijanta obuhvatne rezolucije se zasniva na primeni pravila rezolucije (Explain) na način koji obezbeđuje da će klauza C dobijena primenom rezolucije biti obuhvaćena polaznom vrednošću klauze pre primene pravila rezolucije, što garantuje da opisani postupak sve vreme smanjuje tekuću vrednost klauze C.

pravilo	*sat*?	M	F	C
`Decide` `(l = 6)`	`UNDEF`	$[\|6]$	F_0	-
`UnitPropagate` `(c = [1, −6], l = 1)`	`UNDEF`	$[\|6, 1]$	F_0	-
`UnitPropagate` `(c = [−1, 2], l = 2)`	`UNDEF`	$[\|6, 1, 2]$	F_0	-
`Decide` `(l = 7)`	`UNDEF`	$[\|6, 1, 2, \|7]$	F_0	-
`Decide` `(l = 3)`	`UNDEF`	$[\|6, 1, 2, \|7, \|3]$	F_0	-
`UnitPropagate` `(c = [−3, 4], l = 4)`	`UNDEF`	$[\|6, 1, 2, \|7, \|3, 4]$	F_0	-
`UnitPropagate` `(c = [−1, −3, 5], l = 5)`	`UNDEF`	$[\|6, 1, 2, \|7, \|3, 4, 5]$	F_0	-
`Conflict` $(M \models \neg[-2, -4, -5])$	`UNDEF`	$[\|6, 1, 2, \|7, \|3, 4, 5]$	F_0	$[-2, -4, -5]$
`Explain` `(l = 5, reason = [−1, −3, 5])`	`UNDEF`	$[\|6, 1, 2, \|7, \|3, 4, 5]$	F_0	$[-1, -2, -3, -4]$
`Explain` `(l = 4, reason = [−3, 4])`	`UNDEF`	$[\|6, 1, 2, \|7, \|3, 4, 5]$	F_0	$[-1, -2, -3]$
`Learn` $(C = [-1, -2, -3])$	`UNDEF`	$[\|6, 1, 2, \|7, \|3, 4, 5]$	$F_0@[-1, -2, -3]$	$[-1, -2, -3]$
`Backjump` $(C = [-1, -2, -3]$, `l = 3, level = 1)`	`UNDEF`	$[\|6, 1, 2, -3]$	$F_0@[-1, -2, -3]$	-
`UnitPropagate` `(c = [−2, 3, 5, −6], l = 5)`	`UNDEF`	$[\|6, 1, 2, -3, 5]$	$F_0@[-1, -2, -3]$	-
`Conflict` $(M \models \neg[-1, 3, -5, -6])$	`UNDEF`	$[\|6, 1, 2, -3, 5]$	$F_0@[-1, -2, -3]$	$[-1, 3, -5, -6]$
`Explain` `(l = 5, reason = [−2, 3, 5, −6])`	`UNDEF`	$[\|6, 1, 2, -3, 5]$	$F_0@[-1, -2, -3]$	$[-1, -2, 3, -6]$
`Explain` `(l = −3, reason = [−1, −2, −3])`	`UNDEF`	$[\|6, 1, 2, -3, 5]$	$F_0@[-1, -2, -3]$	$[-1, -2, -6]$
`Explain` `(l = 2, reason = [−1, 2])`	`UNDEF`	$[\|6, 1, 2, -3, 5]$	$F_0@[-1, -2, -3]$	$[-1, -6]$
`Explain` `(l = 1, reason = [−6, 1])`	`UNDEF`	$[\|6, 1, 2, -3, 5]$	$F_0@[-1, -2, -3]$	$[-6]$
`Learn` $(C = [-6])$	`UNDEF`	$[\|6, 1, 2, -3, 5]$	$F_0@[-1, -2, -3]@[-6]$	$[-6]$
`Backjump` $(C = [-6]$, `l = 6, level = 0)`	`UNDEF`	$[-6]$	$F_0@[-1, -2, -3]@[-6]$	-
`Decide` `(l = 1)`	`UNDEF`	$[-6, \|1]$	$F_0@[-1, -2, -3]@[-6]$	-
`UnitPropagate` `(c = [−1, 2], l = 2)`	`UNDEF`	$[-6, \|1, 2]$	$F_0@[-1, -2, -3]@[-6]$	-
`UnitPropagate` `(c = [−1, −2, −3], l = −3)`	`UNDEF`	$[-6, \|1, 2, -3]$	$F_0@[-1, -2, -3]@[-6]$	-
`Decide` `(l = 7)`	`UNDEF`	$[-6, \|1, 2, -3, \|7]$	$F_0@[-1, -2, -3]@[-6]$	-
`Decide` `(l = 4)`	`UNDEF`	$[-6, \|1, 2, -3, \|7, \|4]$	$F_0@[-1, -2, -3]@[-6]$	-
`UnitPropagate` `(c = [−2, −4, −5], l = −5)`	`UNDEF`	$[-6, \|1, 2, -3, \|7, \|4, -5]$	$F_0@[-1, -2, -3]@[-6]$	-
$M \not\models \neg F_0$, (`vars` M) = (`vars` F_0)	`SAT`	$[-6, \|1, 2, -3, \|7, \|4, -5]$	-	-

Tabela 4.3: Prikaz izvršavanja iz primera 4.3

$$\dfrac{\dfrac{[-2,-4,-5] \qquad [-1,-3,5]}{[-1,-2,-3,-4]} \qquad [-3,4]}{[-1,-2,-3]}$$

$$\dfrac{\dfrac{\dfrac{[-1,3,-5,-6] \qquad [-2,3,5,-6]}{[-1,-2,3,-6]} \qquad [-1,-2,-3]}{[-1,-2,-6]} \qquad [-1,2]}{\dfrac{[-1,-6] \qquad [1,-6]}{[-6]}}$$

Slika 4.3: Stabla rezolucije koja odgovaraju analizama konflikata iz primera 4.3

Problem sa osnovnom varijantom algoritma minimalizacije je da se pravilo Explain ne primenjuje u slučaju da klauza koja je razlog propagacije literala l i sa kojom je potrebno izvršiti rezoluciju sadrži neki literal l' različit od l, a koji ne pripada tekućoj vrednosti klauze C, jer bi se tako u klauzi C pojavili novi literali (svakako i literal l') i ona ne bi bila obuhvaćena svojom polaznom vrednošću. No, u nekim slučajevima se ispostavlja da je literal l' nastao kao rezultat propagacije literala koji se već nalaze u klauzi C što dovodi do toga da dalja primena pravila Explain dovodi do njegove eliminacije i do konačne vrednosti klauze C koja jeste obuhvaćena polaznom vrednošću. Za razliku od osnovne varijante algoritma minimalizacije koja zahteva da se klauza C u svakom koraku smanjuje i da je obuhvaćena polaznom vrednošću, naprednije tehnike minimalizacije dopuštaju klauzi C da privremeno i poraste uvodeći nove literale, ali samo ako je moguće te novo uvedene literale eliminisati daljim koracima rezolucije tako da krajnja vrednost bude klauza koja je obuhvaćena polaznom.

4.4.3 Otpočinjanje iznova

Još jedna važna karakteristika savremenih SAT rešavača je mogućnost *otpočinjanja pretrage iz početka (restarting)*. S vremena na vreme rešavači poništavaju celokupnu valuaciju M sve do prvog pretpostavljenog literala. Ovo može da dovede do boljih performansi zbog činjenice da novi početak može da navede pretragu na novu (obično mnogo lakšu) putanju. Teorijski je pokazano da je sistem koji uključuje učenje i otpočinjanje iznova eksponencijalno moćniji u odnosu na klasični DPLL algoritam [BKS04]. Međutim, neograničeno otpočinjanje iznova može da ugrozi zaustavljanje procedure. Eksperimenti ukazuju da pažljivo dizajnirana politika vršenja otpočinjanja iznova može značajno da doprinese efikasnosti SAT rešavača [Hua07b].

Otpočinjanje iznova je moguće opisati sledećim pravilom:

Restart:

$$\dfrac{\top}{M := M^{[0]}}$$

$M^{[0]}$ označava tzv. nulti nivo označene valuacije, odnosno prefiks valuacije M pre pojave prvog pretpostavljenog literala.

4.4.4 Dva sistema

U ovom poglavlju prikazujemo dve postojeće kompletne formalizacije SAT rešavača opisane korišćenjem pravila prelaska. Pravila će biti navedena na isti način kako je to urađeno u originalnim radovima.

Roberto Niuvenhuis (hol. Roberto Nieuwenhuis) i koautori [NOT06] opisuju stanje rešavača na osnovu skupa klauza F i parcijalne valuacije M (što se zapisuje kao $M||F$). Specijalno stanje $FailState$ se razlikuje od svih ostalih stanja. Početno stanje prilikom rešavanja formule F_0 je stanje $\emptyset \,||\, F_0$. Skup pravila kojima se prelazi iz stanja u stanje je dat na slici 4.4. Dokazana je sledeća teorema (teorema 2.12 u [NOT06]) koja ukazuje na činjenicu da prikazani skup pravila predstavlja proceduru odlučivanja za SAT. [6]

Teorema 4.1. *Ako $\emptyset \parallel F_0 \rightarrow^* S$, pri čemu je S završno stanje u odnosu na pravila sa slike 4.4, onda*

1. *S je $FailState$ ako i samo ako je F nezadovoljiva.*
2. *Ako je S oblika $M||F$, onda je M model formule F_0.*

Prikazani sistem je veoma grub. On obuhvata razne strategije koje se mogu sresti u današnjim SAT rešavačima. Međutim, cena ovoga je da je jako puno detalja neophodno precizirati da bi se sistem mogao prevesti u realnu implementaciju. Na primer, sistem ni u jednom trenutku ne precizira kako se proverava uslov $M \models \neg C$ u pravilima UnitPropagate i Fail, niti kako se pronalazi klauza $C' \vee l'$ u pravilu Backjump.

Krstić i Goel [KG07] opisuju sistem pravila datih na slici 4.5. Ovaj sistem daje znatno detaljniji opis nekih faza rešavanja (naročito faze analize konflikata) nego prethodno opisani sistem. Stanja su oblika $(M \,||\, F \,||\, C)$, odnosno uz formulu F i parcijalnu valuaciju M, stanje rešavača se karakteriše i objektom C koji može biti bilo skup literala, bilo poseban simbol no_cflct i koji ima značajnu ulogu tokom analize konflikata. Početno stanje sistema za skup klauza F_0 je stanje $\emptyset \,||\, F_0 \,||\, no\_cflct$. Pravila se zapisuju tako što su iznad linija napisani uslovi koji omogućuju primenu pravila, dok su ispod linija dodele koje opisuju način na koji se menja stanje nakon primene pravila. Dokazana je teorema (teorema 4 u [KG07]) koja garantuje da i ovaj sistem pravila opisuje proceduru odlučivanja za SAT[7].

Teorema 4.2. *Svi lanci primena pravila sistema* DPLL *su konačni. Ukoliko se za dati početni skup klauza F_0,* DPLL *zaustavlja u stanju $F \parallel M \parallel C$, tada važi:*

1. *$C = no\_cflct$ ili $C = \emptyset$;*
2. *Ako je $C = \emptyset$, tada je F_0 nezadovoljiva;*

[6] Kako bi se osiguralo zaustavljanje, neophodno je postaviti dodate uslove vezane za primenu pravila Restart, Learn i Forget.

[7] Kako bi se osiguralo zaustavljanje, potrebno je nametnuti dodatne uslove na primenu pravila Restart i Forget

UnitPropagate:

$$M \parallel F, C \vee l \quad \rightarrow \quad M\ l \parallel F, C \vee l \quad \text{ako} \begin{cases} M \vDash \neg C \\ l \text{ nije definisan u } M \end{cases}$$

PureLiteral:

$$M \parallel F \quad \rightarrow \quad M\ l \parallel F \quad \text{ako} \begin{cases} l \text{ se javlja u nekoj klauzi iz } F \\ \overline{l} \text{ se ne javlja ni u jednoj klauzi iz } F \\ l \text{ nije definisan u } M \end{cases}$$

Decide:

$$M \parallel F \quad \rightarrow \quad M\ l^d \parallel F \quad \text{ako} \begin{cases} l \text{ ili } \overline{l} \text{ se pojavljuje u } F \\ l \text{ nije definisan u } M \end{cases}$$

Fail:

$$M \parallel F, C \quad \rightarrow \quad FailState \quad \text{ako} \begin{cases} M \vDash \neg C \\ M \text{ ne sadrži pretpostavljene literale} \end{cases}$$

Backtrack:

$$M\ l^d\ N \parallel F, C \quad \rightarrow \quad M\ \overline{l} \parallel F, C \quad \text{ako} \begin{cases} M\ l^d N \vDash \neg C \\ N \text{ ne sadrži pretpostavljene literale} \end{cases}$$

Backjump:

$$M\ l^d\ N \parallel F, C \quad \rightarrow \quad M\ l' \parallel F, C \quad \text{ako} \begin{cases} M\ l^d N \vDash \neg C \\ \text{postoji klauza } C' \vee l' \text{ takva da :} \\ \quad F, C \vDash C' \vee l' \text{ i } M \vDash \neg C', \\ \quad l' \text{ nije definisan u } M, \\ \quad l' \text{ ili } \overline{l'} \text{ se javlja u } F \text{ ili u } M\ l^d\ N \end{cases}$$

Learn:

$$M \parallel F \quad \rightarrow \quad M \parallel F, C \quad \text{ako} \begin{cases} \text{svi literali iz } C \text{ se javljaju u } F \\ F \vDash C \end{cases}$$

Forget:

$$M \parallel F, C \quad \rightarrow \quad M \parallel F \quad \text{ako } \{F \vDash C\}$$

Restart:

$$M \parallel F \quad \rightarrow \quad \emptyset \parallel F$$

Slika 4.4: Pravila sistema DPLL iz [NOT06]

3. Ako je $C = no\_cflct$ tada je M model formule F_0.

4.5 Heurističke komponente SAT rešavača

Iako navedena pravila opisuju globalni način funkcionisanja SAT rešavača, redosled njihove primene nije jasno preciziran. Takođe, u datom stanju je često moguće primeniti više pravila ili isto pravilo na različite načine. Za efikasno navođenje procesa rešavanja koriste se različite heuristike o kojima će biti reči u ovom poglavlju.

4.5.1 Strategije izbora pretpostavljenih literala

Pravilo Decide ne definiše precizno način na koji se određuje literal l koji se pretpostavlja, već jedino zahteva da je l član unapred zadatog skupa literala L, kao i da nije do tada definisan u valuaciji M. Izbor ovog literala ne utiče na korektnost rešavača (što će i biti dokazano u poglavlju 7.2.1), međutim može veoma značajno da utiče na efikasnost rešavača. Strategiju koja definiše način na koji se bira literal u pravilu Decide se naziva *strategija izbora pretpostavljenih literala (eng. decision strategy)*.

Decide:
$$\frac{l \in L \qquad l, \overline{l} \notin M}{M := M\ l^d}$$
UnitPropag:
$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F \qquad \overline{l}_1, \ldots, \overline{l}_k \in M \qquad l, \overline{l} \notin M}{M := M\ l}$$
Conflict:
$$\frac{C = no\_cflct \qquad \overline{l}_1 \vee \ldots \vee \overline{l}_k \in F \qquad l_1, \ldots, l_k \in M}{C := \{l_1, \ldots, l_k\}}$$
Explain:
$$\frac{l \in C \qquad l \vee \overline{l}_1 \vee \ldots \vee \overline{l}_k \in F \qquad l_1, \ldots, l_k \prec l}{C := C \cup \{l_1, \ldots, l_k\} \setminus \{l\}}$$
Learn:
$$\frac{C = \{l_1, \ldots, l_k\} \qquad \overline{l}_1 \vee \ldots \vee \overline{l}_k \notin F}{F := F \cup \{\overline{l}_1 \vee \ldots \vee \overline{l}_k\}}$$
Backjump:
$$\frac{C = \{l, l_1, \ldots, l_k\} \qquad \overline{l} \vee \overline{l}_1 \vee \ldots \vee \overline{l}_k \in F \qquad \text{level } l > m \geq \text{level } l_i}{C := no\_cflct \qquad M := M^{[m]}\ \overline{l}}$$
Forget:
$$\frac{C = no\_cflct \qquad c \in F \qquad F \setminus c \vDash c}{F := F \setminus c}$$
Restart:
$$\frac{C = no\_cflct}{M := M^{[0]}}$$

Slika 4.5: Pravila sistema DPLL iz [KG07]

Pretpostavlja se da ne postoji idealna strategija koja bi bila efikasna za sve vrste formula, već praksa pokazuje da različite strategije pokazuju različite rezultate na različitim vrstama formula. Iako se za teorijska istraživanja često rešavaju slučajno generisane formule *(eng. random instances)*, u primenama najznačajnije mesto zauzimaju tzv. industrijske formule *(eng. industrial instances)*.

Pokazuje se da je jedan od presudnih faktora prlikom izgradnje strategije izbora pretpostavljenih literala koja postiže dobre rezultate na industrijskim problemima tzv. lokalnost pretrage. Pokazuje se da je za utvrđivanje zadovoljivosti veoma velikih industrijskih problema ponekad dovoljno vršiti grananje na osnovu veoma malog broja promenljivih koje se javljaju u formuli, a koje su na neki način povezane preko klauza formule. Sa druge strane, veoma je poželjno da strategija u velikoj meri bude dinamična i da se prilagođava promenama nastalim u tekućoj valuaciji. Naime, i jedna jedina izmena u tekućoj valuaciji može učiniti određene delove formule relevantnim odnosno potpuno irelevantnim za grananje. Na primer, ukoliko formula opisuje elektronski sklop koji sadrži AND kolo određeno promenljivim x i y, postavljanje literala $\overline{x}$ ili $\overline{y}$ u tekuću valuaciju u potpunosti deaktivira deo formule koji predstavlja onaj deo kola koji je vezan za izlaz kapije.

Pretpostavljeni literal u pravilu Decide se obično bira u dve nezavisne faze. U prvoj fazi se bira promenljiva, dok se u drugoj fazi bira njen polaritet.

U slučaju da je formula zadovoljiva, izbor polariteta pretpostavljenih literala, očigledno, može da ima značajnu ulogu za efikasno pronalaženje modela. Zaista, ukoliko postoji model koji sadrži literal l, a ne postoji model koji sadrži literal $\overline{l}$, odabir literala l pre literala $\overline{l}$ dovodi bržem pronalaženju

rešenja. U slučaju nezadovoljivih formula odabir polariteta pretpostavljenih literala ne mora da igra tako značajnu ulogu. Zaista, pretpostavimo da rešavač pretpostavi literal l i pokaže da je ta pretpostavka neodrživa. Kada se ta pretpostavka promeni, odnosno kada se u valauciji literal l zameni literalom $\overline{l}$, efikasnost pokazivanja nezadovoljivosti zavisi od klauza koje su naučene pod pretpostavkom l. Međutim, pošto se uče isključivo konfliktne klauze, ni jedna od njih ne može da sadrži literal l. S druge strane, naučene klauze koje sadrže $\overline{l}$ su zadovoljene nakon dodavanja literala $\overline{l}$ u valuaciju i od tog trenutka nemaju nikakvog efekta na pretragu. Zbog ovoga, na pretragu u kontekstu literala $\overline{l}$ imaju uticaja samo one naučene klauze koje ne sadrže ni literal l ni $\overline{l}$. Da je početni polaritet bio suprotan, tj. da je prvo isproban literal $\overline{l}$, a tek nakon toga literal l, na veličinu prostora pretrage bi opet uticale samo one klauze koje su nastale kao rezultat konflikta koji u sebi ne sadrže ni l ni $\overline{l}$ i samim tim ne zavise od početnog izbora polariteta. Ovo ukazuje na to da izbor polariteta u ovom slučaju nije bitan. Ipak, u slučaju kada rešavač koristi tehniku otpočinjanja iznova, izbor polariteta može da ima značajnu ulogu. Razlog ovome je činjenica da se nakon otpočinjanja iznova stablo pretrage u potpunosti menja. U tom slučaju ne postoji garancija da će uopšte i biti ispitivan onaj deo stabla koji odgovara alternativnom polaritetu od polariteta koji je inicijalno bio izabran.

U nastavku će biti opisano nekoliko heuristika za izbor literala.

Heuristika slučajnog izbora

Najjednostavnija heuristika izbora pretpostavljenih literala bira proizvoljni literal iz L nedefinisan u M korišćenjem generatora pseudo-slučajnih brojeva

VSIDS heuristika

Jedna od najčešće korišćenih heuristika za izbor pretpostavljenih literala je *VSIDS - variable state independent decaying sum* heuristika uvedena u rešavaču zChaff [MMZ+01]. Modifikacije ove heuristike su korišćene u nekoliko uticajnih rešavača, uključujući i rešavač MiniSAT [ES04]. Na ovom mestu ćemo opisati ovu heuristiku u obliku koji je korišćen u okviru rešavača MiniSAT. Ova heuristika nezavisno bira promenljivu grananja njen polaritet.

Kako bi se odredila promenljiva pretpostavljenog literala, svakoj promenljivoj se dodeljuje broj koji se naziva *aktivnost*. Prilikom svake primene pravila Conflict svim promenljivim koje su članovi konfliktne klauze se uvećava aktivnost. Slično se i prilikom primene pravila Explain svim promenljivim koje su članovi klauze koja je razlog propagacije uvećava aktivnost. Prilikom primene pravila Decide, bira se promenljiva sa najvećom aktivnošću. Međutim, kako bi se poboljšala lokalnost procesa pretrage, s vremena na vreme se smanjuje aktivnost svih promenljivih. Ovim se postiže da se prilikom primene pravila Decide prednost daje promenljivima koje su učestvovale u najskorijim konfliktima. Rešavač MiniSAT vrši ovo smanjivanje aktivnost prilikom svake primene pravila Conflict.

Postoje različite mogućnosti za izbor polariteta literala i obično su veoma jednostavne. U svojoj podrazumevanoj konfiguraciji rešavač MiniSAT uvek bira literal negativnog polariteta.

Berkmin heuristika

Jedna od osnovnih zamerki VSIDS heuristici je da nije dovoljno dinamična. Kao što je već navedeno, dodavanje i jednog jedinog literala u tekuću valuaciju može da učini velike delove formule pasivnim i irelevantnim za njenu zadovoljivost. Ukoliko pretpostavimo da je u nekom trenutku zbog izmene tekuće valuacije došlo do promene relevantnog skupa promenljivih na osnovu kojeg je poželjno vršiti grananje, potrebno je da protekne neko vreme dok aktivnosti svih promenljivih u VSIDS heuristici ne opadnu dovoljno da bi se identifikovao nov relevantni skup promenljivih pogodnih za grananje. U međuvremenu, rešavač gubi vreme vršeći grananje na osnovu promenljivih koje su do skora bile relevantne, ali na osnovu promene u tekućoj valuaciji one to iznenada više nisu jer se javljaju u okviru pasiviziranih delova formule.

Rešavač Berkmin [GN02] na specifičan način proširuje VSIDS heuristiku pokušavajući da ispravi ove nedostatke. Klauze formule F čine listu pri čemu se prilikom svake primene pravila Learn naučene klauze postavljaju na kraj te liste. To znači da se u svakom trenutku sve inicijalne klauze u listi nalaze ispred naučenih klauza. U svakom trenutku rada rešavača postoji poslednja klauza u listi koja nije zadovoljena i ona se naziva *poslednja nezadovoljena klauza*.

Za klauzu C formule F koja je poslednja klauza u listi takva da nije tačna u M kažemo da je poslednja nezadovoljena klauza u F. Berkmin heuristika se ponaša različito u slučajevima kada je C početna i kada je C naučena klauza.

- U slučaju da je C naučena klauza, heuristika bira promenljivu na osnovu koje se vrši grananje isključivo među promenljivim klauze C koje nisu definisane u M. Izbor između ovih promenljivih se vrši na osnovu njihove aktivnosti. Iako u odnosu na ceo skup nedefinisanih promenljivih skupa L, promenljive klauze C mogu da imaju prilično nisku aktivnost, rezultat visoke aktivnosti promenljivih koje se ne nalaze u klauzi C može da bude rezultat njihove aktivnosti u konfliktima koji su se desili pre učenja klauze C. Dakle, većina ovih aktivnih promenljivih može da bude potpuno irelevantna za konflikt koji je doveo do učenja klauze C i njemu sličnih konflikata.

  Neka je l literal koji je član klauze C nedefinisan u M i koji odgovara promenljivoj v sa najvećom aktivnošću koja se javlja u klauzi C. Berkmin heuristika ne pokušava direktno da zadovolji klauzu C dodajući literal l valuaciji M, već koristeći posebnu heuristiku određuje polaritet literala. Naime, polaritet literala se određuje na način koji pokušava da izjednači broj pojavljivanja literala l i $\overline{l}$ u naučenim klauzama formule F. Ovo se postiže time što se održava broj pojavljivanja svakog od literala u naučenim klauzama i prilikom izbora bira se literal sa većim brojem pojavljivanja. Time veći broj naučenih klauza odjednom biva zadovoljen. Sve buduće naučene klauze su rezultat konflikta i samim tim mogu da sadrže isključivo literal koji je suprotan upravo odabranom polaritetu.

  Nije teško primetiti da se korišćenjem ovakve heuristike za izbor polariteta zadovoljavanje klauze C ne odlaže previše. Naime, ukoliko se umesto literala l u valuaciju doda literal $\overline{l}$ i to ne dovede do daljih konflikata, C ostaje poslednja nezadovoljena klauza i postupak se ponavlja dok C ne postane ili zadovoljena ili jedinična klauza nakon čega se njen jedinični literal propagira.

- U slučaju da je C inicijalna klauza, tada su i sve ostale klauze formule F inicijalne klauze. U tom slučaju se promenljiva na osnovu koje se vrši grananje bira među svim promenljivim koje nisu definisane u M i to na osnovu njihove aktivnosti.

  U ovom slučaju se polaritet literala određuje tako što se za svaki polaritet literala l pokušava proceniti broj binarnih klauza (klauza sa dva literala) u njegovoj „okolini". Ovaj broj procenjuje koliko će literala biti zaključeno kao rezultat propagacije jediničnih klauza na osnovu pretpostavke l. Strategija izbora polariteta bira polaritet koji će dovesti do veće propagacije odnosno bira polaritet koji ima veći broj binarnih klauza u svojoj okolini.

4.5.2 Strategije započinjanja iznova

Osnovna dilema prilikom primene pravila Restart je određivanje trenutka kada treba poništiti sve napravljene pretpostavke i njihove posledice. Kako bi imali garanciju zaustavljanja pretrage, otpočinjanje iznova ne sme biti previše intenzivno i tokom pretrage mora da se proređuje. Trenutak otpočinjanja iznova se najčešće određuje na osnovu broja konflikta (tj. broja primene pravila Conflict).

Konstantna politika

U slučaju korišćenja konstantne politike, otpočinjanje iznova se vrši nakon svakih N konflikata, gde je N fiksirani broj. Rešavač zChaff[MMZ+01] koristi ovu politiku za $N = 700$, a rešavača Berkmin za $N = 550$. Ova politika ne garantuje zaustavljanje.

Linearna (aritmetička) politika

U slučaju korišćenja linearne politike, otpočinjanje iznova se vrši kada broj konflikta nastalih od prethodnog početka dostigne naredni element unapred datog aritmetičkog niza. Na primer, otpočinjanje iznova se vrši nakon prvih 100 konflikata, zatim nakon narednih 200 konflikata, narednih 300 konflikata itd.

Eksponencijalna (geometrijska) politika

U slučaju korišćenja eksponencijalne politike, otpočinjanje iznova se vrši kada broj konflikta nastalih od prethodnog početka dostigne naredni element unapred datog geometrijskog niza niza. Na primer, ako je prvi član niza 100, a faktor uvećanja 1.5 otpočinjanje iznova se vrši nakon prvih 100 konflikata, zatim nakon narednih 150 konflikata, narednih 225 konflikata itd.

Lubijeva optimalna strategija

Lubijeva (eng. Luby) strategija je izumljena za potrebe određene klase stohastičkih algoritama poznatih kao *Las Vegas* algoritmi. U kontekstu Las Vegas algoritama ova strategija se pokazuje kao univerzalno optimalna strategija, tj. (i) njeno očekivano vreme izvršavanja je samo za logaritamski faktor sporije od optimalnog vremena izvršavanja za svaku instancu problema i (ii) ni

jedna druga univerzalna strategija ne može postići više od konstantnog faktora ubrzanja u odnosu na Lubijevu strategiju [Hua07b]. Iako ovakva svojstva Lubijeve strategije nisu pokazana u kontekstu strategija otpočinjanja iznova kod SAT rešavača, ona se ipak u novije vreme sve intenzivnije koristi.

Osnovni Lubijev niz je $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots$ i definisan je na sledeći način:

$$L_i = \begin{cases} 2^{k-1}, & i = 2^k - 1 \\ L_{i-2^{k-1}+1}, & 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

Skalirani Lubijev niz se dobija kada se svaki član osnovnog Lubijevog niza pomnoži konstantom L_c. Za vrednosti konstante L_c rešavači obično koriste brojeve $32, 64, 128, 256$ ili 512.

Otpočinjanje iznova se vrši kada broj konflikata proteklih od prethodnog početka dostigne narednu tačku skaliranog Lubijevog niza.

4.5.3 Strategije zaboravljanja

Prilikom primene pravila Forget je potrebno odrediti trenutak u kome se pravilo primenjuje, kao i koje klauze je potrebno zaboraviti.

Trenutak primene pravila zaboravljanja je određen veličinom tekuće formule F. Ova veličina se najčešće procenjuje njenim tekućim brojem klauza. Pored ovoga, moguće je posmatrati i ukupan broj literala koji sačinjavaju klauze u formuli F. Prilikom svake primene pravila zaboravljanja, uklanja se određeni fiksirani procenat klauza formule F. Većina heuristika dopušta da maksimalna veličina formule postepeno raste, uvećavajući postepeno gornju granicu veličine formule na kojoj se zaboravljanje vrši. Ovo uvećavanje može biti linearno ili eksponencijalno.

Prilikom zaboravljanja, najčešće se uklanjaju isključivo naučene klauze. Kriterijumi koji određuju da li određenu naučenu klauzu treba zaboraviti su njena veličina određena brojem literala i njena aktivnost određena brojem propagacija u kojima je učestvovala kao i brojem konflikata u čijem je razrešavanju učestvovala.

4.6 Pristupi za dostizanje pouzdanih SAT rešavača

Jedan od najvažniji ciljeva računarstva je izgradnja pouzdanog softvera u čije je rezultatima moguće verovati. Ovo je naročito važno za algoritme i programe koji imaju značajne i važne primene, među kojima su, svakako, i SAT rešavači.

Jedan od pristupa za dostizanje većeg stepena poverenja u rezultate SAT rešavača je tehnika *proveravanja dokaza (eng. proof-checking)*. Ova tehnika se veoma uspešno koristi tokom prethodnih nekoliko godina. Da bi se ova tehnika mogla primeniti, neophodno je izvršiti izmene SAT rešavača tako da uz odgovor *sat* ili *unsat* koji označava da je formula zadovoljiva, odnosno nezadovoljiva, pruži i argumente koji nas uveravaju da je dati odgovor tačan. Za zadovoljive formule, ovi argumenti su jednostavno modeli, tj. zadovoljavajuće valuacije. Za nezadovoljive formule, argumenti su obično objekti koji opisuju korake rezolucije čijom se primenom iz polazne formule izvodi prazna klauza. Ovakvi dokazi su obično veoma glomazni i kompleksni. Dokazi pruženi od strane SAT rešavača se zatim nezavisno proveravaju programima koji se nazivaju *proveravači dokaza (eng. proof-checkers)*. Proveru dokaza je relativno jednostavno

implementirati, međutim, ovaj pristup ima važne nedostatke. Na primer, za svaku formulu koja se reši, potrebno je nezavisno proveriti njen dokaz. Generisanje dokaza nezadovoljivosti donekle usporava rad rešavača, dokazi su obično veliki i mogu da zauzmu i do nekoliko gigabajta skladišnog prostora, a i sâmo vreme provere dokaza može da traje značajno vreme [Gel07]. Pošto proveravači dokaza moraju biti pouzdani i neophodno je imati poverenje u njihove odgovore, oni moraju biti veoma jednostavni programi tako da se u njihovu pouzdanost može uveriti manuelnom inspekcijom koda. S druge strane, kako bi provera dokaza bila efikasna, oni moraju da koriste specijalizovanu funkcionalnost operativnog sistema što značajno umanjuje nivo njihove pouzdanosti (na primer, proveravač korišćen u okviru SAT takmičenja koristi `mmap` funkciju operativnog sistema Linux [Gel07].)

Drugi pristup za dostizanje većeg stepena poverenja u rezultate SAT rešavača je *verifikacija samog rešavača*. Na ovaj način, umesto da se proverava dokaz svake pojedinačne formule, provera korektnosti rešavača se izvršava samo jednom. Sa druge strane, ovaj pristup je značajno teže sprovesti u delo jer zahteva formalnu analizu celokupnog ponašanja rešavača. Ipak, u ovoj tezi se koristi ovaj pristup jer verujemo da donosi značajne prednosti kada se jednom uspešno sprovede:

- Iako dodatno vreme potrebno za generisanje dokaza nezadovoljivosti tokom pretrage nije nesavladivo (ono iznosi između 10% i 30% ukupnog vremena rešavanja [Gel07]), ono se ipak može izbeći ukoliko je sâm rešavač verifikovan.

- Verifikacija savremenih SAT rešavača može pomoći u boljem teorijskom razumevanju kako i zašto oni rade. Rigorozna matematička analiza ovih sistema može da otkrije i neka moguća poboljšanja algoritama i tehnika koje se koriste i da utiče na razvoj novih, boljih, SAT rešavača.

- Verifikovani SAT rešavači mogu da služe kao *poverljivo jezgro (eng. trusted kernel)* proveravača koji proveravaju rezultate drugih nepouzdanih dokazivača kao što su BDD-ovi, proveravači modela (eng. model checkers), SMT rešavači, itd. Takođe, provera određenih funkcionalnosti SAT rešavača (pre svega propagacije bulovskih ograničenja (eng. Boolean constraint propagation), može da bude osnova za izgradnju proverenih i istovremeno efikasnih proveravača za SAT.

- Želimo da prikažemo da je, zahvaljujući skorašnjem napretku u tehnologiji automatskog i poluautomatskog formalnog dokazivanja teorema, konačno došlo vreme kada je moguće imati potpuno verifikovan softver koji nije trivijalan i ima važne primene. Rad na ovom polju doprinosi *Velikom izazovu verifikacije (eng. Verification Grand Challenge)* [VSTTE].

Deo I

Formalizacija

5
Opis meta-logike

Cilj ovog dela teze je da opiše potpunu formalizaciju SAT rešavača u strogo formalnom matematičkom okviru i da kroz niz tvrđenja iskaže i dokaže njihovu korektnost. Kako bi moglo da se rezonuje o korektnosti implementacije SAT rešavača, neophodno je formalno definisati sve pojmove o kojima se rezonuje. Ovo uključuje i detaljnu formalizaciju iskazne logike, kao i algoritama i struktura podataka koje se koriste u SAT rešavačima. Ova formalizacija će biti urađena u okviru logike višeg reda koju ćemo nazivati meta-logikom i koja je inspirisana sistemom Isabelle/HOL i koja će biti opisana u okviru ove glave. Za njen detaljniji opis meta-logike sistema Isabelle, zainteresovani čitalac se upućuje na dokumentaciju i izvorni kôd ovog sistema [NPW02].

Formule i osnovni logički veznici meta-logike koju koristimo se pišu na uobičajeni način ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$). Ipak, ponekad ćemo pisati $\Longrightarrow$ umesto $\Rightarrow$ i $\Longleftrightarrow$ umesto $\Leftrightarrow$ kako bi se poboljšala čitljivost. Simbol $=$ označava sintaksnu jednakost dva izraza. Primene funkcijsih simbola ćemo zapisivati u prefiksnom obliku, kao na primer $(\mathsf{f}\ x_1\ \ldots\ x_n)$. Egzistencijalno kvantifikovane formule će biti označavane sa $\exists x.\ P(x)$, a univerzalno kvantifikovane formule sa $\forall x.\ P(x)$. Logika podržava i ternarni veznik *if–then–else*: $if\ f\ then\ f_1\ else\ f_2$ označava formulu $f \Rightarrow f_1\ \wedge\ \neg f \Rightarrow f_2$.

Meta-logika podržava tipove i svi simboli imaju dodeljene tipove. Međutim, kako bi se pojednostavila prezentacija, tipovi će se često izostavljati prilikom zapisa formula.

U okviru ove meta-logike treba izgraditi teoriju koja će omogućiti formalno rezonovanje o korektnosti SAT rešavača. Teoriju ćemo izgraditi navođenjem *definicija* koje uvode nove pojmove i zatim dokazivanjem tvrđenja koja govore o svojstvima tih pojmova. Tvrđenja će biti organizovana u *stavove*, *leme* i *teoreme*. Stavovi će biti dati bez dokaza[1] i obično će sadržati jednostavna tehnička tvrđenja. Leme će sadržati složenija pomoćna tvrđenja koja se koriste u okviru drugih dokaza, dok će teoreme će sadržati najznačajnije rezultate.

Pretpostavićemo da nam je prilikom izgradnje teorije koju definišemo na raspolaganju teoriju lista. Sintaksa operacija nad listama je data u prvoj koloni naredne tabele, dok je njihova semantika neformalno opisana u drugoj koloni.

[1] Dokazi svih tvrđenja (uključujući i stavove) navedenih u ovoj tezi su formalizovani u okviru sistema Isabelle i u tom obliku su dostupni su sa web adrese `http://argo.matf.bg.ac.rs`

$[\,]$	prazna lista.
$[e_1, \ldots, e_n]$	lista koja se sastoji od n elemenata $e_1, \ldots, e_n$.
$e \mathbin{@} list$	lista koja se dobija nadovezivanjem elementa e na početak liste $list$.
$list \mathbin{@} e$	lista koja se dobija nadovezivanjem elementa e na kraj liste $list$.
$list_1 @ list_2$	lista koja se dobija nadovezivanjem listi $list_1$ i $list_2$.
$e \in list$	e je element liste list $list$.
$list_1 \subseteq list_2$	svi elementi liste $list_1$ su ujedno i elementi liste $list_2$.
$list \setminus e$	lista koja se dobija uklanjanjem svih pojavljivanja elementa e iz liste $list$
$list_1 \setminus list_2$	lista koja se dobija od liste $list_1$ uklanjanjem svih elemenata liste $list_2$.
(head $list$)	početni element liste $list$ (pod pretpostavkom da nije prazna).
(first $list$)	sinonim za (head $list$).
(second $list$)	drugi element liste $list$ (pod pretpostavkom da ima bar dva elementa).
(tail $list$)	lista koja se dobija uklanjanjem prvog elementa neprazne liste $list$
(last $list$)	poslednji element neprazne liste
(butlast $list$)	lista koja se dobija uklanjanjem poslednjeg elementa neprazne liste $list$
(length $list$)	dužina liste $list$.
(distinct $list$)	true akko lista $list$ ne sadrži elemente koji se javljaju više puta.

Sve navedene operacije nad listama su direktno podržane u sistemu Isabelle, te se za njihovo strogo formalno definisanje zainteresovani čitalac upućuje na dokumentaciju i izvorni kôd ovog sistema [NPW02]. U nastavku će biti uvedene još neke operacije nad listama koje sistem Isabelle ne podržava direktno.

Definicija 5.1. Prefiks do datog elementa *liste* l, *što označavamo sa* (prefixToElement e l) *čine svi elementi liste* l *koji se pojavljuju pre prve prve pojave literala* l, *uključujući i literal* l. *Ovu operaciju je moguće formalno definisati korišćenjem primitivne rekurzije na sledeći način.*

$$\begin{aligned}(\mathsf{prefixToElement}\ e\ [\,]) &= [\,] \\ (\mathsf{prefixToElement}\ e\ (e' \mathbin{@} l')) &= \\ &(if\ e = e'\ then\ [e]\ else\ e' \mathbin{@} (\mathsf{prefixToElement}\ e\ l'))\end{aligned}$$

Definicija 5.2. Prefiks pre datog elementa *liste* l, *što označavamo sa* (prefixBeforeElement e l) *čine svi elementi liste* l *koji se pojavljuju pre prve prve pojave literala* l, *ne uključujući i literal* l. *Ovu operaciju je moguće formalno definisati korišćenjem primitivne rekurzije na sledeći način.*

$$\begin{aligned}(\mathsf{prefixBeforeElement}\ e\ [\,]) &= [\,] \\ (\mathsf{prefixBeforeElement}\ e\ (e' \mathbin{@} l')) &= \\ &(if\ e = e'\ then\ [\,]\ else\ e' \mathbin{@} (\mathsf{prefixBeforeElement}\ e\ l'))\end{aligned}$$

Definicija 5.3. *Neka su* l_1 *i* l_2 *elementi liste* L. *Kažemo da* l_1 prethodi elementu l_2 u listi L, *što označavamo sa* $l_1 \prec^L l_2$, *ukoliko je prva pozicija na kojoj se pojavljuje element* l_1 *strogo manja od prve pozicije na kojoj se pojavljuje* l_2.

Takođe, pretpostavićemo i da na raspolaganju imamo i teoriju (konačnih) skupova. Sintaksa operacija nad skupovima je data u prvoj koloni naredne tabele, dok je njihova semantika neformalno opisana u drugoj koloni.

$\{\}$	prazan skup.
$e \in set$	element e pripada skupu set.
$set_1 \cup set_2$	unija skupova set_1 i set_2.
$\|set\|$	broj elemenata konačnog skupa set

Skupovi u ovom obliku su direktno podržani u sistemu Isabelle, te se za njihovo strogo formalno zasnivanje zainteresovani čitalac upućuje na dokumentaciju i izvorni kôd ovog sistema[NPW02].

Takođe, pretpostavićemo i da naša teorija uključuje i preslikavanja između konačnih skupova.

$\{a_1 \mapsto b_1, \ldots, a_k \mapsto b_k\}$	preslikavanje elemenata $\{a_1, \ldots, a_k\}$, redom u elemente $\{b_1, \ldots, b_k\}$
$H(a_i)$	slika elementa a_i u preslikavanju H (pod pretpostavkom da je ova slika definisana)

Sistem Isabelle takođe direktno podržava i ovako uvedena preslikavanja.

Sada će biti uveden pojam relacije i neka važna svojstva relacija koja će biti korišćena u nastavku teze. Iako je i većina ovih pojmova direktno podržana sistemom Isabelle, iz želje da tekst ove teze čini zaokruženu celinu, navodimo ih na ovom mestu.

Uređeni parovi će biti označavani sa (a, b).

Definicija 5.4. *Binarna relacija ρ na skupu A je proizvoljni podskup skupa $A \times A$. $(a_1, a_2) \in \rho$ se zapisuje u infiksnom obliku kao $a_1 \, \rho \, a_2$.*

Definicija 5.5. *Relacija $<$ je relacija* striktnog uređenja *ako važi:*

1. *(irefleksivnost)* $\neg(x < x)$
2. *(tranzitivnost)* $x < y \wedge y < z \implies x < z$

Definicija 5.6. *Relacija $<$ je* dobro zasnovana (eng. well-founded) *ako važi:*

$$\forall P.\ (\forall x.\ (\forall y.\ y < x \implies P(y)) \implies P(x)) \implies (\forall x.\ P(x)))$$

Naredna teorema koju navodimo bez dokaza daje karakterizaciju dobro zasnovanih relacija preko tzv. *principa najmanjeg elementa*.

Teorema 5.1. *Relacija $<$ je dobro zasnovana ako i samo ako važi*

$$\forall\ Q.\ (\exists\ a \in Q) \implies (\exists\ a_{min} \in Q.\ (\forall\ a'.\ a' < a_{min} \implies a' \notin Q))$$

6

Formalno zasnivanje logike iskaznih KNF formula

U ovoj glavi će biti formalizovana većina pojmova uvedenih u 4 i to u okviru meta-logike višeg reda opisane glavi 5. Kao što je rečeno, ova meta-logika podržava tipove. Sledećom definicijom se uvode osnovni tipovi koji opisuju formule u konjunktivnoj normalnoj formi.

Definicija 6.1.

Boolean	`true` or `false`
Nat	prirodan broj
Variable	prirodan broj.
Literal	pozitivna promenljiva ($+vbl$) ili negativna promenljiva ($-vbl$)
Clause	lista elemenata tipa `Literal`
Formula	lista elemenata tipa `Clause`[1]
Valuation	lista elemenata tipa `Literal`

Iako meta-logika zahteva da svakom simbolu bude pridružen tip, prilikom zapisa formula tipovi najčešće neće biti eksplicitno navođeni, već će se koristiti sledeća konvencija: literali (tj. promenljive tipa `Literal`) će biti označavani sa l (npr. $l, l', l_0, l_1, l_2, \ldots$); promenljive će biti označavani sa vbl; klauze će biti označavane sa c; formule će biti označavane sa F; valuacije će biti označavane sa v.

Iako su i klauze i valuacije predstavljene listama literala, njihovo značenje je drugačinje jer klauze predstavljaju disjunkcije, a valuacije konjunkcije literala. Svakoj valuaciji je moguće pridružiti formulu koja odgovara konjunkciji njenih literala. Često u izlaganju nećemo praviti eksplicitnu razliku između valuacije i njoj pridružene konjunkcije literala.

U narednom tekstu ćemo definisati osnovne operacije iskazne logike. Iako su sve naredne definicije formalizovane (najčešće korišćenjem primitivne rekurzije), kako bismo uprostili prezentaciju navešćemo ih samo u prirodno-jezičkom obliku.

Za početak će biti data definicija promenljive datog literala.

Definicija 6.2. *Promenljiva literala $+vbl$ i $-vbl$ je promenljiva vbl.*

[1] Primetimo da se ovim uvode isključivo formule u konjunktivnoj normalnoj formi.

Naglasimo da notacija koju koristimo nije u potpunosti jednoznačna i da neki simboli imaju nekoliko različitih značenja. Na primer, simbol $\in$ označava i pripadnost elementa skupu i pripadnost elementa listi. Takođe, isti simbol ćemo koristiti i da zapišemo da literal pripada formuli.

Definicija 6.3. *Formula F sadrži literal l (tj. literal l se pojavljuje u formuli F), što označavamo sa $l \in F$, akko $\exists c.\ c \in F \wedge l \in c$.*

Simbol vars takođe ima nekoliko značenja i koristi se da označi skup promenljivih koje se pojavljuju u datoj klauzi, datoj formuli ili u datom literalu.

Definicija 6.4. Skup promenljivih koje se javljaju u klauzi c *se označava sa* $(\mathsf{vars}\ c)$. Skup promenljivih koje se javljaju u formuli F *se označava sa* $(\mathsf{vars}\ F)$. Skup promenljivih koje se javljaju u valuaciji v *se označava sa* $(\mathsf{vars}\ v)$.

Osnovna svojstva skupova promenljivih su navedena u narednom stavu koji u ovom tekstu navodimo bez dokaza:

Stav 6.1.

- $l \in c \implies (\mathsf{var}\ l) \in (\mathsf{vars}\ c)$
- $l \in F \implies (\mathsf{var}\ l) \in (\mathsf{vars}\ F)$
- $c \in F \implies (\mathsf{vars}\ c) \subseteq (\mathsf{vars}\ F)$
- $(\mathsf{vars}\ (c_1 \mathbin{@} c_2)) = (\mathsf{vars}\ c_1) \cup (\mathsf{vars}\ c_2)$
- $(\mathsf{vars}\ (F_1 \mathbin{@} F_2)) = (\mathsf{vars}\ F_1) \cup (\mathsf{vars}\ F_2)$
- $(\mathsf{vars}\ c) \setminus (\mathsf{var}\ l) \subseteq (\mathsf{vars}\ (c \setminus l)) \subseteq (\mathsf{vars}\ c)$
- $(\mathsf{vars}\ F) \setminus (\mathsf{vars}\ c) \subseteq (\mathsf{vars}\ (F \setminus c)) \subseteq (\mathsf{vars}\ F)$
- $vbl \in (\mathsf{vars}\ c) \implies \exists l.\ l \in c \wedge (\mathsf{var}\ l) = vbl$
- $vbl \in (\mathsf{vars}\ F) \implies \exists l.\ l \in F \wedge (\mathsf{var}\ l) = vbl$

Definicija 6.5. Suprotan literal literala l, *što označavamo sa* $\overline{l}$, *se definiše kao:* $\overline{+vbl} = -vbl$, $\overline{-vbl} = +vbl$.

Osnovna svojstva suprotnih literala su navedena u narednom stavu koji navodimo bez dokaza.

Stav 6.2.

- $\overline{\overline{l}} = l$
- $\overline{l_1} = l_2 \iff \overline{l_2} = l_1$
- $\overline{l_1} = \overline{l_2} \iff l_1 = l_2$
- $\overline{l} \neq l$
- $(\mathsf{var}\ \overline{l}) = (\mathsf{var}\ l)$
- $(\mathsf{var}\ l_1) = (\mathsf{var}\ l_2) \iff l_1 = l_2 \vee \overline{l_1} = l_2$

Semantika iskazne logike (tačnost, netačnost, relacija zadovoljenja) se definiše na sledeći način.

Definicija 6.6. *Literal l je* tačan u valuaciji v, *što označavamo sa $v \vDash l$, akko $l \in v$.*

Klauza c je tačna u valuaciji v, *što označavamo sa $v \vDash c$, akko $\exists l.\ l \in c \wedge v \vDash l$.*

Formula F je tačna u valuaciji v, *što označavamo sa $v \vDash F$, akko $\forall c.\ c \in F \Rightarrow v \vDash c$.*

Pisaćemo $v \not\models l$ da označimo da l nije tačan u v, $v \not\models c$ da označimo da c nije tačna v i $v \not\models F$ da označimo da F nije tačna u v.

Definicija 6.7. *Literal l je* netačan u valuaciji *v, što označavamo sa $v \models\neg l$, akko $\overline{l} \in v$.*

Klauza c je netačna u valuaciji *v, što označavamo sa $v \models\neg c$, akko $\forall l.\ l \in c \Rightarrow v \models\neg l$.*

Formula F je netačna u valuaciji *v, što označavamo sa $v \models\neg F$, akko $\exists c.\ c \in F \wedge v \models\neg c$.*

Pisaćemo $v \not\models\neg l$ kako bismo označili da l nije netačan u v, $v \not\models\neg c$ da označimo da c nije netačna u v i $v \not\models\neg F$ da označimo da F nije netačna u v.

Osnovna svojstva uvedenih pojmova su sadržana u narednim stavovima koje navodimo bez dokaza.

Stav 6.3.

- $(\mathsf{var}\ l) \in (\mathsf{vars}\ v) \iff v \models l \vee v \models\neg l$
- $v \models (c_1 \mathbin{@} c_2) \mathbin{@} c_3 \iff v \models c_1 \mathbin{@} (c_2 \mathbin{@} c_3)$
- $v \models c_1 \mathbin{@} c_2 \iff v \models c_2 \mathbin{@} c_1$
- $v \models c_1 \mathbin{@} c_2 \iff v \models c_1 \vee v \models c_2$
- $v \models c \implies (v \mathbin{@} v') \models c$
- $(v_1 \mathbin{@} v_2) \models c \implies (v_2 \mathbin{@} v_1) \models c$
- $v \models\neg c \implies (v \mathbin{@} v') \models\neg c$
- $(v_1 \mathbin{@} v_2) \models\neg c \implies (v_2 \mathbin{@} v_1) \models\neg c$
- $c \neq [\,] \implies [\,] \not\models\neg c$
- $v \models\neg c \implies v \models\neg c \setminus l$
- $v \models\neg c \ \wedge\ c' \subseteq c \implies v \models\neg c'$
- $v \models\neg [\,]$
- $v \models\neg c \implies (\mathsf{vars}\ c) \subseteq (\mathsf{vars}\ v)$

Stav 6.4.

- $v \models (F_1 \mathbin{@} F_2) \mathbin{@} F_3 \iff v \models F_1 \mathbin{@} (F_2 \mathbin{@} F_3)$
- $v \models F_1 \mathbin{@} F_2 \iff v \models F_2 \mathbin{@} F_1$
- $v \models F_1 \mathbin{@} F_2 \iff v \models F_1 \wedge v \models F_2$
- $v \models F \implies v \models F \setminus c$
- $v \models F \ \wedge\ F' \subseteq F \implies v \models F'$
- $v \models [\,]$
- $v \models F \implies (v \mathbin{@} v') \models F$
- $(v_1 \mathbin{@} v_2) \models F \implies (v_2 \mathbin{@} v_1) \models F$
- $v \models\neg F \implies (v \mathbin{@} v') \models\neg F$
- $(v_1 \mathbin{@} v_2) \models\neg F \implies (v_2 \mathbin{@} v_1) \models\neg F$
- $v \models\neg F \implies v \models\neg F \mathbin{@} F'$

Definicija 6.8. *Valuacija v je* protivrečna*, što označavamo sa* (inconsistent v)*, akko sadrži dva međusobno suprotna literala tj. $\exists l.\ v \models l \wedge v \models \overline{l}$. Valuacija je* neprotivrečna*, što označavamo sa* (consistent v)*, akko nije protivrečna.*

Osnovna svojstva vezana za neprotivrečnost valuacija su navedena u narednom stavu koji navodimo bez dokaza.

Stav 6.5.

- $v \vDash c \wedge v \vDash \neg\, c \implies (\mathsf{inconsistent}\ v)$
- $v \vDash F \wedge v \vDash \neg\, F \implies (\mathsf{inconsistent}\ v)$
- $(\mathsf{inconsistent}\ (v_1 \mathbin{@} v_2)) \iff$
  $(\mathsf{inconsistent}\ v_1) \vee (\mathsf{inconsistent}\ v_2) \vee (\exists l.\ v_1 \vDash l \wedge v_2 \vDash \neg\, l)$
- $(\mathsf{inconsistent}\ (v \setminus l)) \implies (\mathsf{inconsistent}\ v)$
- $v_1 \subseteq v_2 \wedge (\mathsf{inconsistent}\ v_1) \implies (\mathsf{inconsistent}\ v_2)$

Definicija 6.9. *Valuacija v je* totalna *u odnosu na skup promenjivih Vbl, što označavamo sa* $(\mathsf{total}\ v\ Vbl)$*, akko* $(\mathsf{vars}\ v) \supseteq Vbl$.

Osnovna svojstva totalnih valuacija su sadržana u na narednom stavu koji navodimo bez dokaza.

Stav 6.6.

- $Vbl_1 \subseteq Vbl_2 \wedge (\mathsf{total}\ v\ Vbl_2) \implies (\mathsf{total}\ v\ Vbl_1)$
- $c \in F \wedge (\mathsf{total}\ v\ (\mathsf{vars}\ F)) \implies (\mathsf{total}\ v\ (\mathsf{vars}\ c))$
- $(\mathsf{total}\ v\ (\mathsf{vars}\ c)) \wedge l \in c \implies v \vDash l \vee v \vDash \neg\, l$
- $(\mathsf{total}\ v\ (\mathsf{vars}\ c)) \implies v \vDash c \vee v \vDash \neg\, c$
- $(\mathsf{total}\ v\ (\mathsf{vars}\ F)) \wedge l \in F \implies v \vDash l \vee v \vDash \neg\, l$
- $(\mathsf{total}\ v\ (\mathsf{vars}\ F)) \wedge c \in F \implies v \vDash c \vee v \vDash \neg\, c$
- $(\mathsf{total}\ v\ (\mathsf{vars}\ F)) \implies v \vDash F \vee v \vDash \neg\, F$
- $(\mathsf{total}\ v\ (\mathsf{vars}\ (F \setminus [l]))) \wedge (\mathsf{var}\ l) \in (\mathsf{vars}\ v) \implies (\mathsf{total}\ v\ (\mathsf{vars}\ F))$

Definicija 6.10. Model *formule F je neprotivrečna valuacija u kojoj je formula F tačna. Formula F je* zadovoljiva*, što označavamo sa* $(\mathsf{sat}\ F)$*, akko ima model tj. ako važi* $\exists v.\ (\mathsf{consistent}\ v) \wedge v \vDash F$.

Osnovna svojstva modela i zadovoljivosti su navedena u narednom stavu koji navodimo bez dokaza.

Stav 6.7.

- $(\mathsf{sat}\ [\,])$
- $[\,] \in F \implies \neg(\mathsf{sat}\ F)$
- $F' \subseteq F \wedge (\mathsf{sat}\ F) \implies (\mathsf{sat}\ F')$
- $(\mathsf{sat}\ (F_1 \mathbin{@} F_2)) \implies (\mathsf{sat}\ F_1) \wedge (\mathsf{sat}\ F_2)$
- $(\mathsf{model}\ v\ F) \wedge (\mathsf{var}\ l) \notin (\mathsf{vars}\ v) \implies (\mathsf{model}\ (v \mathbin{@} l)\ F)$
- $(\mathsf{model}\ v\ F) \wedge vbl \notin (\mathsf{vars}\ F) \implies \exists v'.\ (\mathsf{model}\ v'\ F) \wedge vbl \notin (\mathsf{vars}\ v')$

Definicija 6.11. Klauza c povlači literal l*, što označavamo sa $c \vDash l$, akko je literal l tačan u svakom modelu klauze c. Kažemo i da je literal l* semantička posledica *klauze c.*

Definicija 6.12. Formula F povlači literal l*, što označavamo sa $F \vDash l$, akko je literal l tačan u svakom modelu formule F. Kažemo i da je literal l* semantička posledica *formule F.*

Definicija 6.13. Formula F povlači klauzu c*, što označavamo sa $F \vDash c$, akko je klauza c tačna u svakom modelu formule F. Kažemo i da je klauza c* semantička posledica *formule F.*

Definicija 6.14. Formula F povlači valuaciju v, *što označavamo sa* $F \vDash v$, *akko povlači sve njene literale tj.* $\forall l.\ l \in v \Rightarrow F \vDash l$. *Kažemo i da je valuacija* v semantička posledica *formule* F.

Definicija 6.15. Formula F_1 povlači formulu F_2 *što označavamo sa* $F_1 \vDash F_2$, *ukoliko je svaki model formule* F_1 *ujedno i model formule* F_2. *Kažemo i da je formula* F_2 semantička posledica *formule* F_1.

Osnovna svojstva upravo uvedenih pojmova su navedena u narednom stavu koji navodimo bez dokaza.

Stav 6.8.

- $[l] \vDash l$
- $c \in F \wedge c \vDash l \implies F \vDash l$
- $F \vDash l \implies (F \mathbin{@} F') \vDash l$
- $(F \setminus c) \vDash l \implies F \vDash l$
- $F \vDash l \wedge F \subseteq F' \implies F' \vDash l$
- $c \in F \implies F \vDash c$
- $F \vDash c \implies (F \mathbin{@} F') \vDash c$
- $F \vDash [\,] \iff \neg(\mathsf{sat}\ F)$
- $F_1 \vDash F_2 \wedge (\mathsf{sat}\ F_1) \implies (\mathsf{sat}\ F_2)$
- $F_1 \vDash F_2 \iff \forall c.\ c \in F_2 \implies F_1 \vDash c$
- $F_1 \vDash F_2 \wedge F_2 \vDash c \implies F_1 \vDash c$
- $F_1 \vDash F_2 \wedge F_2 \vDash F_3 \implies F_1 \vDash F_3$
- $(F_1 \mathbin{@} F_2) \vDash l \wedge F_1' \vDash F_1 \implies (F_1' \mathbin{@} F_2) \vDash l$
- $v \vDash \neg F \wedge F \vDash v \implies \neg(\mathsf{sat}\ F)$
- $\neg(\mathsf{sat}\ F) \wedge [l] \in F \implies (F \setminus [l]) \vDash \overline{l}$

Definicija 6.16. *Formule* F_1 *i* F_2 *su* logički ekvivalentne, *što označavamo sa* $F_1 \equiv F_2$, *ukoliko je svaki model formule* F_1 *ujedno i model formule* F_2 *i obratno, tj. ako* $F_1 \vDash F_2$ *i* $F_2 \vDash F_1$.

Osnovna svojstva ekvivalentnih formula su sadržana u narednom stavu koji navodimo bez dokaza.

Stav 6.9.

- $F \equiv F$
- $F_1 \equiv F_2 \iff F_2 \equiv F_1$
- $F_1 \equiv F_2 \wedge F_2 \equiv F_3 \implies F_1 \equiv F_3$
- $F_1 \equiv F_1' \implies (F_1 \mathbin{@} F_2) \equiv (F_1' \mathbin{@} F_2)$
- $F_1 \equiv F_2 \implies (\mathsf{sat}\ F_1) = (\mathsf{sat}\ F_2)$
- $F_1 \equiv F_2 \wedge F_2 \vDash c \implies F_1 \equiv (F_2 \mathbin{@} c)$
- $F_1 \equiv F_2 \wedge F_2 \setminus c \vDash c \implies F_1 \equiv (F_2 \setminus c)$
- $F_1 \equiv F_1' \wedge (F_1 \mathbin{@} F_2) \vDash l \implies (F_1' \mathbin{@} F_2) \vDash l$

Definicija 6.17. Rezolventa *klauza* c_1 *i* c_2 *u odnosu na literal* l, *što označavamo sa* $(\mathsf{resolvent}\ c_1\ c_2\ l)$ *je klauza* $(c_1 \setminus l) @ (c_2 \setminus \overline{l})$.

Stav 6.10. *Ako važi* $(\mathsf{consistent}\ v)$ *i* $v \vDash c_1$ *i* $v \vDash c_2$, *onda važi* $v \vDash (\mathsf{resolvent}\ c_1\ c_2\ l)$.

Dokaz. Pošto važi $v \vDash c_1$, postoji literal l' takav da $l' \in c_1$ i $l' \in v$. Pošto važi $v \vDash c_2$, postoji literal l'' takav da $l'' \in c_2$ i $l'' \in v$.

Ako $l' \neq l$, tada je $l' \in (\mathsf{resolvent}\ c_1\ c_2\ l)$ i $v \vDash (\mathsf{resolvent}\ c_1\ c_2\ l)$.

Ako je $l' = l$, tada mora da važi $l'' \neq \overline{l}$. Zaista, ukoliko ovo ne bi bio slučaj, tada bi i l i $\overline{l}$ bili tačni u v, što je u suprotnosti sa pretpostavkom $(\mathsf{consistent}\ v)$. Ali tada važi $l'' \in (\mathsf{resolvent}\ c_1\ c_2\ l)$ i $v \vDash (\mathsf{resolvent}\ c_1\ c_2\ l)$. □

7 Formalizacija sistema promena stanja za SAT

U ovoj glavi će biti formalno definisana i dokazana svojstva modernih SAT algoritama zasnovanih na modifikacijama DPLL algoritma, a opisanih korišćenjem sistema pravila po uzoru na sistem pravila dat u [KG07] i na pravila prikazana u glavi 4.

7.1 Označene valuacije (Assertion Trail)

Kao što je to već nagovešteno u glavi 4, u cilju efikasnije implementacije DPLL algoritma, neophodno je donekle proširiti pojam valuacije. Tokom procesa rešavanja, SAT rešavači čuvaju tekuću parcijalnu valuaciju. U toj valuaciji, moguće je razlikovati dve različite vrste literala — *pretpostavljene literale (decision literals)* i *izvedene literale (implied literals)*. Valuacije koje uz svaki literal čuvaju oznaku koja označava da li se radi o pretpostavljenom ili izvedenom literalu ćemo nazivati *označene valuacije (assertion trail)*. Kada se implementiraju, označene valuacije funkcionišu kao stek i literali se dodaju i uklanjaju sa njihovog kraja. U formalnu teoriju koju gradimo uvodimo sledeći tip.

Definicija 7.1.

Trail	lista literala pri čemu su neki od njih označeni.

Jedan od mogućih načina definisanja označenih valuacija je u vidu niza uređenih parova oblika `(Literal, Boolean)`. Promenljive tipa `Trail` ćemo označavati sa M (npr. $M, M', M_0, \ldots$).

Primer 7.1. Označena valuacija M može biti

$$[+1, |-2, +6, |+5, -3, +4, |-7].$$

Pretpostavljeni literali su označeni simbolom | sa svoje leve strane. Alternativno, u nekim slučajevima, uz literale ćemo eksplicitno navoditi oznake $\top$ i $\bot$. Tako bi prethodna valuacija bila navedena kao

$$[+1^{\bot}, -2^{\top}, +6^{\bot}, +5^{\top}, -3^{\bot}, +4^{\bot}, -7^{\top}].$$

Ponekad će pretpostavljeni literali biti označeni sa ^{d} (što je skraćeno za engleski termin *decision*). Tako bi prethodna valuacija bila navedena kao

$$[+1, -2^d, +6, +5^d, -3, +4, -7^d].$$

Definicija 7.2. (isDecision l) *važi ako i samo ako je* l *pretpostavljeni literal.*

Definicija 7.3. Lista pretpostavljenih literala*, što označavamo sa* (decisions M)*, je lista svih označenih elemenata (tj. svih pretpostavljenih literala) iz označene valuacije* M*, tj., korišćenjem primitivne rekruzije,*

$$\begin{array}{rcl} (\text{decisions } [\,]) & = & [\,] \\ (\text{decisions } l \mathbin{@} M') & = & (if\ (\text{isDecision } l)\ then\ l \mathbin{@} (\text{decisions } M')\ else\ (\text{decisions } M')) \end{array}$$

Definicija 7.4. Poslednji pretpostavljeni literal*, što označavamo sa* (lastDecision M)*, je poslednji označeni element liste* M*, tj.*

$$(\text{lastDecision } M) = (\text{last } (\text{decisions } M)).$$

Definicija 7.5. Lista pretpostavljenih literala do datog literala*, što označavamo sa* (decisionsTo l M)*, je lista svih označenih elemenata liste* M *koji prethode prvoj pojavi literala* l*, uključujući i* l *ukoliko je označen, tj.*

$$(\text{decisionsTo } l\ M) = (\text{decisions } (\text{prefixToElement } l\ M))$$

Podsetimo se da je funkcija prefixToElement uvedena u glavi 5.

Primer 7.2. Za označenu valuaciju iz Primera 7.1 važi:

$$\begin{array}{rcl} (\text{decisions } M) & = & [-2, +5, -7], \\ (\text{lastDecision } M) & = & -7, \\ (\text{decisionsTo } +4\ M) & = & [-2, +5], \\ (\text{decisionsTo } -7\ M) & = & [-2, +5, -7]. \end{array}$$

Definicija 7.6. Tekući nivo *označene valuacije* M*, što označavamo sa* (currentLevel M)*, je broj označenih literala u* M*, tj.*

$$(\text{currentLevel } M) = (\text{length } (\text{decisions } M)).$$

Definicija 7.7. Nivo literala l *u označenoj valuaciji* M*, što označavamo sa* (level l M)*, je broj označenih elemenata u označenoj valuaciji koji prethode prvoj pojavi literala* l*, uključujući i* l *ukoliko je označen, tj.*

$$(\text{level } l\ M) = (\text{length } (\text{decisionsTo } l\ M)).$$

Definicija 7.8. Prefiks do datog nivoa*, što označavamo sa* (prefixToLevel $level$ M)*, je prefiks označene valuacije* M *koji sadrži sve elemente* M *čiji su nivoi manji ili jednaki od broja* $level$*, tj. važi:*

$$(\text{isPrefix } (\text{prefixToLevel } level\ M)\ M),$$

$$l \in (\text{prefixToLevel } level\ M) \iff l \in M\ \wedge (\text{level } l\ M) \leq level$$

Podsetimo se da je funkcija isPrefix uvedena u glavi 5.

Definicija 7.9. Prefiks pre poslednjeg pretpostavljenog literala, *što označavamo sa* (prefixBeforeLastDecision M), *je prefiks označene valuacije* M *do poslednjeg označenog elementa, ne uključujući njega samoga,*[1] *tj., korišćenjem primitivne rekurzije,*

$$\begin{aligned}
(\mathsf{prefixBeforeLastDecision}\ [\,]) &= [\,] \\
(\mathsf{prefixBeforeLastDecision}\ l \mathbin{@} M') &= (if\ (\mathsf{isDecision}\ l) \wedge (\mathsf{decisions}\ M') = [\,]\ then \\
&\qquad [\,] \\
&\quad else \\
&\qquad l \mathbin{@} (\mathsf{prefixBeforeLastDecision}\ M'))
\end{aligned}$$

U slučaju da važi (distinct M), važi da je

$$(\mathsf{prefixBeforeLastDecision}\ M) = (\mathsf{prefixBeforeElement}\ (\mathsf{lastDecision}\ M)\ M).$$

Primer 7.3. Za označenu valuaciju M iz primera 7.1 važi

$$\begin{aligned}
(\mathsf{level}\ +1\ M) &= 0, \\
(\mathsf{level}\ +4\ M) &= 2, \\
(\mathsf{level}\ -7\ M) &= 3, \\
(\mathsf{currentLevel}\ M) &= 3, \\
(\mathsf{prefixToLevel}\ 1\ M) &= [+1, |+2, +6], \\
(\mathsf{prefixBeforeLastDecision}\ M) &= [+1, |-2, +6, |+5, -3, +4].
\end{aligned}$$

Definicija 7.10. Poslednji tačan literal klauze c *u odnosu na valuaciju* M, *što označavamo sa* (lastAssertedLiteral c M), *je literal klauze* c *koji je tačan u* M, *takav da nijedan drugi literal iz* c *nije iza njega u* M.

$$\begin{aligned}
l = (\mathsf{lastAssertedLiteral}\ c\ M) \iff\ & l \in c\ \wedge\ M \vDash l\ \wedge \\
& \forall\, l'.\ l' \in c\ \wedge\ M \vDash l'\ \Longrightarrow\ \neg(l \prec^M l')
\end{aligned}$$

Podsetimo se da relacija $\prec^M$, uvedena u glavi 5, označava da jedan element prethodi drugom u okviru liste M.

Definicija 7.11. Najveći nivo klauze c *u odnosu na označenu valuaciju* M, *što označavamo sa* (maxLevel c M), *je najveći od nivoa svih literala iz* c *koji su tačni u* M, *tj.*

$$(\mathsf{maxLevel}\ c\ M) = (\mathsf{level}\ (\mathsf{lastAssertedLiteral}\ c\ M)\ M).$$

Primer 7.4. Neka je $c = [+4, +6, -3]$ i M označena valuacija iz Primera 7.1. Tada važi

$$\begin{aligned}
(\mathsf{lastAssertedLiteral}\ c\ M) &= +4, \\
(\mathsf{maxLevel}\ c\ M) &= 2.
\end{aligned}$$

[1] Primetimo da su neke od uvedenih funkcija parcijalne i da nisu definisane za sve označene valuacije. Na primer, (prefixBeforeLastDecision M) je definisan samo za označene valuacije koje imaju bar jedan označeni literal.

7.1.1 Svojstva označenih valuacija

Naredni stavovi koje navodimo bez dokaza ilustruju osnovna svojstva označenih valuacija koja se koriste u dokazima koji slede.

Naredni stav govori o svojstvima funkcije prefixToElement.

Stav 7.1.

- (prefixToElement e $(l_1$ @ $l_2)) =$
  $(if\ e \in l_1\ then$ (prefixToElement e $l_1)$
  $else\ l_1$ @ (prefixToElement e $l_2))$
- (isPrefix P $M)$ $\wedge$ $e \in P$ $\Longrightarrow$
  (prefixToElement e $P)$ = (prefixToElement e $M)$

Naredni stav govori o svojstvima funkcije decisions.

Stav 7.2.

- $l \in$ (decisions M) $\Longleftrightarrow$ $l \in M$ $\wedge$ (isDecision l)
- (isPrefix P M) $\Longrightarrow$ (isPrefix (decisions P) (decisions M))
- (distinct M) $\wedge$ (isPrefix P M) $\wedge$ $e \in P$ $\wedge$ $e \in$ (decisions M) $\Longrightarrow$
  $e \in$ (decisions P)

Naredni stav govori o svojstvima funkcije decisionsTo.

Stav 7.3.

- (isPrefix (decisionsTo M l) (decisions M))
- $l \notin M$ $\Longrightarrow$ (decisionsTo M l) = (decisions M)
- (distinct M) $\wedge$ $l \in M$ $\wedge$ (isDecision l) $\Longrightarrow$ $l \in$ (decisionsTo l M)
- (decisionsTo l $(M_1$ @ $M_2)) =$ $(if\ l \in M_1\ then$ (decisionsTo l $M_1)$
  $else\ M_1$ @ (decisionsTo l $M_2))$
- (isPrefix P $M)$ $\wedge$ $l \in P$ $\Longrightarrow$ (decisionsTo l $P)$ = (decisionsTo l $M)$

Naredni stav govori o svojstvima funkcije lastDecision.

Stav 7.4.

- (distinct M) $\Longrightarrow$ (decisionsTo (lastDecision M) M) = (decisions M)
- (decisions M) =
  (decisions (prefixBeforeLastDecision M)) @ (lastDecision M)

Naredni stav govori o svojstvima funkcije level.

Stav 7.5.

- (level l $M) \geq 0$
- (distinct M) $\wedge$ (isDecision l) $\Longrightarrow$ (level l $M) \geq 1$
- $l \in M$ $\Longrightarrow$ (level l $(M_1$ @ $M_2)) =$ (level l $M_1)$
- $l_1 \prec^M l_2$ $\Longrightarrow$ (level l_1 $M) \leq$ (level l_2 $M)$
- (distinct M) $\wedge$ $l_1 \prec^M l_2$ $\wedge$ (isDecision l_2) $\Longrightarrow$ (level l_1 $M) \leq$ (level l_2 $M)$
- (distinct M) $\wedge$ (isDecision l_1) $\wedge$ (isDecision l_2) $\wedge$ $l_1 \neq l_2$ $\Longrightarrow$
  (level l_1 $M) \neq$ (level l_2 $M)$

Naredni stav govori o svojstvima funkcije currentLevel.

Stav 7.6.

- $(\mathsf{currentLevel}\ (M\,@\,l^{\perp})) = (\mathsf{currentLevel}\ M)$
- $(\mathsf{isPrefix}\ P\ M) \implies (\mathsf{currentLevel}\ P) \leq (\mathsf{currentLevel}\ M)$
- $l \in M \implies (\mathsf{level}\ l\ M) \leq (\mathsf{currentLevel}\ M)$
- $l \notin M \implies (\mathsf{level}\ l\ (M\,@\,l)) = (\mathsf{currentLevel}\ M)$

Naredni stav govori o svojstvima funkcije prefixToLevel.

Stav 7.7.

- $(\mathsf{isPrefix}\ (\mathsf{prefixToLevel}\ level\ M)\ M)$
- $0 \leq level \wedge level < (\mathsf{currentLevel}\ M) \implies$
  $M = (\mathsf{prefixToLevel}\ level\ M)\,@\,M' \wedge M' \neq [\,] \wedge (\mathsf{isDecision}\ (\mathsf{head}\ M'))$
- $l \in M\ \wedge\ (\mathsf{level}\ l\ M) \leq level \implies l \in (\mathsf{prefixToLevel}\ level\ M)$
- $level \geq 0\ \wedge\ level < (\mathsf{level}\ l\ M) \implies l \notin (\mathsf{prefixToLevel}\ level\ M)$
- $level \geq 0 \implies (\mathsf{currentLevel}\ (\mathsf{prefixToLevel}\ level\ M)) \leq level$
- $level \geq 0\ \wedge\ l \in (\mathsf{prefixToLevel}\ level\ M) \implies (\mathsf{level}\ l\ M) \leq level$
- $l \in (\mathsf{prefixToLevel}\ level\ M) \implies$
  $(\mathsf{level}\ l\ (\mathsf{prefixToLevel}\ level\ M)) = (\mathsf{level}\ l\ M)$

7.2 Osnovna pretraga (Decide, Backtrack)

U ovom poglavlju će korišćenjem sistema pravila biti formulisana osnovna proceduru za proveru iskazne zadovoljivosti koja je neformalno bila uvedena u poglavlju 4.3.2. Osnova ranije rekurzivno definisane procedure je operacija zamene koja tokom rada intenzivno menja i redukuje formulu kojoj se zadovoljivost ispituje. Za razliku od toga, nerekurzivno definisana procedura koju ćemo u ovom poglavlju navesti, tokom celokupnog procesa rešavanja ne menja formulu kojoj se ispituje zadovoljivost. Umesto izmene formule, procedura održava i menja označenu valuaciju koja predstavlja tekuću (parcijalnu) valuaciju koja je pridružena formuli čija se zadovoljivost ispituje. Procedura koju ćemo opisati u vidu sistema pravila je pogodna za nerekurzivnu implementaciju što će biti pokazano u delu o implementaciji.

Naprednije procedure koje koriste tehniku učenja menjaju kako tekuću označenu valuaciju, tako i formulu čija se zadovoljivost ispituje i to dodajući joj i uklanjajući joj klauze. Pošto će većina tih procedura biti formalizovana na uniforman način, od početka je odabrano da se stanje procedure okarakteriše kao par koji sadrži označenu valuaciju i formulu kojoj se zadovoljivost ispituje. Jedini fiksirani parametar procedure je skup promenljivih $DecisionVars$ koji se smatra ulaznim parametrom sistema. Iako nije nužno da je $DecisionVars$ jednak skupu promenljivih ulazne formule, to najčešće jeste slučaj.

Definicija 7.12. Stanje *procedure osnovne pretrage je par* (M, F)*, gde je* M *označena valuacija, a* F *formula.*

Pravila procedure osnovne pretrage su:

Decide:

$$\frac{(\mathsf{var}\ l) \in DecisionVars, \quad l, \overline{l} \notin M}{M := M\,|\,l}$$

Backtrack:
$$\frac{M = M' \,|\, l\, M'', \quad (\text{decisions } M'') = [\,], \quad M \models\neg F}{M := M' \,\overline{l}}$$

Naredna definicija formalizuje ova pravila u okviru meta-logike i teorije navedene u glavi 5 i glavi 6. Svako pravilo definiše relaciju prelaza na skupu stanja. Fiksirani parametar je skup promenljivih $DecisionVars$ koji se smatra ulaznim parametrom sistema.

Definicija 7.13.
$(\text{decide } (M_1, F_1)\ (M_2, F_2)) \iff$

$$\exists l. \quad (\text{var } l) \in DecisionVars \ \wedge\ l \notin M_1 \ \wedge\ \overline{l} \notin M_1 \ \wedge\ M_2 = M_1 \mathbin{@} l^{\top} \ \wedge\ F_2 = F_1$$

$(\text{backtrack } (M_1, F_1)\ (M_2, F_2)) \iff$

$$M_1 \models\neg F_1 \ \wedge\ (\text{decisions } M_1) \neq [\,] \ \wedge$$
$$M_2 = (\text{prefixBeforeLastDecision } M_1) \mathbin{@} \overline{(\text{lastDecision } M_1)}^{\perp} \ \wedge\ F_2 = F_1$$

Definicija 7.14.

$$(M_1, F_1) \to (M_2, F_2) \iff (\text{decide } (M_1, F_1)\ (M_2, F_2)) \vee (\text{backtrack } (M_1, F_1)\ (M_2, F_2))$$

n-ti stepen relacije $\to$ označavamo sa $\to^n$, njeno tranzitivno zatvorenje označavamo sa $\to^+$, a njeno tranzitivno i refleksivno zatvorenje označavamo sa $\to^*$.

Definicija 7.15. *Stanje (M, F) je završno u odnosu na relaciju $\to$, ukoliko je minimalni element te relacije, odnosno ukoliko ne postoji stanje (M', F') tako da važi $(M, F) \to (M', F')$.*

Razlikovaćemo dve različite vrste stanja[2]:

1. *Prihvatajuća stanja (eng. accepting states)* - stanja u kojima važi $M \not\models\neg F$ i ne postoji literal takav da je $(\text{var } l) \in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$.
2. *Odbacujuća stanja (eng. rejecting states)* - stanja u kojima važi $M \models\neg F$ i $(\text{decisions } M) = [\,]$.

7.2.1 Saglasnost osnovne pretrage

Cilj ovog poglavlja je da dokažemo teoremu o saglasnosti osnovne pretrage. Saglasnost za nezadovoljive formule govori da ukoliko sistem dospe u odbacujuće stanje, polazna formula je zaista nezadovoljiva. Saglasnost za zadovoljive formule govori da ukoliko dospe u prihvatajuće stanje, polazna formula je zaista zadovoljiva.

Teorema 7.1. *Ukoliko važi $([\,], F_0) \to^* (M, F)$, tada:*

[2]Prihvatajuća stanja se nazivaju i *zadovoljiva* stanja pošto ukazuju na zadovoljivost formule, a odbacujuća stanja se nazivaju i *nezadovoljiva*.

1. *Ukoliko je*

   a) $(\mathsf{vars}\ F_0) \subseteq DecisionVars$,

   b) $M \models_{\neg} F$ *i*

   c) *pravilo* Decide *ne može da se primeni (tj. ne postoji literal* l *takav da je* $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ *i* $\overline{l} \notin M$*),*

   tada je formula F_0 *zadovoljiva i* M *je njen model (tj. važi* $(\mathsf{sat}\ F_0)$ *i* $(\mathsf{model}\ M\ F_0)$*).*

2. *Ukoliko važi*

   a) $M \models_{\neg} F$,

   b) *pravilo* Backtrack *ne može da se primeni (tj.* $(\mathsf{decisions}\ M) = [\,]$*),*

   tada je formula F_0 *nezadovoljiva (tj. važi* $\neg(\mathsf{sat}\ F_0)$*).*

U nastavku poglavlja ćemo formulisati i dokazati niz pomoćnih tvrđenja koja će pomoći da se ova teorema dokaže. Na kraju poglavlja, ova teorema će ponovo biti navedena i tada će biti izložen i njen formalni dokaz.

Kako bismo dokazali teoremu o saglasnosti, formulisaćemo naredne uslove i pokazati da su oni invarijante za oba pravila.

$Invarijanta_{consistent}$:	$(\mathsf{consistent}\ M)$
$Invarijanta_{distinct}$:	$(\mathsf{distinct}\ M)$
$Invarijanta_{equiv}$:	$F \equiv F_0$
$Invarijanta_{varsF}$:	$(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$
$Invarijanta_{impliedLiterals}$:	$\forall l.\ l \in M \implies F\ @\ (\mathsf{decisionsTo}\ l\ M) \models l$

$Invarijanta_{consistent}$ (tj. uslov $(\mathsf{consistent}\ M)$) obezbeđuje da je valuacija M uvek neprotivrečna i da time u svakom koraku predstavlja potencijalni model formule.

$Invarijanta_{distinct}$ (tj. uslov $(\mathsf{distinct}\ M)$) obezbeđuje da M ne sadrži ponovljene literale, što čini važan uslov za zaustavljanje procedure kao i važnu pretpostavku određenih funkcija (na primer, $(\mathsf{decisionsTo}\ l\ M)$).

$Invarijanta_{equiv}$ (tj. uslov $F \equiv F_0$) govori o tome da je tekuća formula F u svakom trenutku logički ekvivalentna polaznoj.

$Invarijanta_{varsF}$ (tj. uslov $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$) obezbeđuje da se tokom rada procedure u formulu F ne uvode nove promenljive, osim eventualno promenljivih iz skupa $DecisionVars$.

$Invarijanta_{impliedLiterals}$ (tj. uslov $\forall l.\ l \in M \implies F\ @\ (\mathsf{decisionsTo}\ l\ M) \models l$) obezbeđuje da su svi izvedeni literali u valuaciji M posledica tekuće formule F i svih pretpostavljenih literala koji im prethode.

Primetimo da osnovna pretraga zadovoljava uslov $F = F_0$ koji je jači od invarijanti $Invarijanta_{equiv}$ i $Invarijanta_{varsF}$ i svakako obezbeđuje njihovo važenje. Ipak, modifikacije procedure pretrage koje će biti formulisane u narednim poglavljima će menjati formulu, pa se koristi širi skup invarijanti koji smo naveli.

U nastavku teksta dokazujemo da navedeni uslovi važe u svakom stanju izvršavanja procedure i da su stoga zaista invarijante.

Naredna lema koja se odnosi na početna stanja trivijalno važi.

Lema 7.1. *U početnom stanju* $([\,], F_0)$ *su sve invarijante zadovljene.*

Naredna lema, čiji će dokaz biti implicitno dat malo kasnije, obezbeđuje da primena pravila prelaska čuva invarijante.

Lema 7.2. *Ukoliko* $(M, F) \to (M', F')$ *i invarijante su zadovoljene u stanju* (M, F)*, tada su invarijante zadovoljene i u stanju* (M', F')*.*

Dokaz ove leme se zasniva na zasebnoj analizi raznih slučajeva iz definicije relacije $\to$ i raznih invarijanti. Za svaku kombinaciju pojedinačnog pravila prelaska i pojedinačne invarijante biće formulisana i dokazana posebna lema koja govori da to invarijanta ostaje očuvana nakon primene tog pojedinačnog pravila. No, pre nego što ovo bude urađeno, dokažimo da ove dve navedene leme zaista obezbeđuju da su invarijante zadovljene tokom celokupnog rada sistema.

Lema 7.3. *Ukoliko je* $([\,], F_0) \to^* (M, F)$*, tada su invarijante zadovljene u stanju* (M, F)*.*

Dokaz. Tvrđenje dokazujemo indukcijom po broju primenjenih pravila.

Na osnovu leme 7.1, invarijante važe u stanju $([\,], F_0)$, tako da je baza indukcije zadovoljena.

Pretpostavimo kao induktivnu hipotezu da tvrđenje važi za sva izvođenja dužine n, tj. da ukoliko važi $([\,], F_0) \to^n (M', F')$, onda stanje (M', F') zadovoljava sve invarijante.

Razmatrajmo izvođenje dužine $n+1$, tj. neka važi $([\,], F_0) \to^{n+1} (M, F)$. Tada postoji stanje (M', F') tako da $([\,], F_0) \to^n (M', F')$ i $(M', F') \to (M, F)$. Na osnovu induktivne hipoteze invarijante su zadovoljene u staju (M', F'). Tada su, na osnovu leme 7.2, invarijante zadovoljene i u stanju (M, F) i tvrđenje je dokazano. □

Pre nego što bude dokazano da invarijante nastavljaju da važe i nakon primene pravila prelaska, biće dokazano nekoliko tvrđenja koja karakterišu invarijantu $Invarijanta_{impliedLiterals}$ ili su njene posledice.

Najpre navodimo slabiju formu invarijante $Invarijanta_{impliedLiterals}$, koja govori o tome da je celokupna valuacija M u svakom koraku semantička posledica formule F @ (decisions M).

Lema 7.4. *Ukoliko važi* $\forall l.\ l \in M \implies F$ @ (decisionsTo l M) $\vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za valuaciju* M*) onda je valuacija* M *posledica formule* F @ (decisions M)*, odnosno* F @ (decisions M) $\vDash M$*.*

Dokaz. Pošto za svaki literal l iz M važi da je (decisionsTo l M) podlista liste (decisions M), tvrđenje trivijalno važi. □

Naredna lema govori o tome da ukoliko se otkrije činjenica da je formula F netačna u valuaciji M, tada neki od pretpostavljenih literala iz M mora da se zameni kako bi se opet dobio potencijalni model formule F.

Lema 7.5. *Ukoliko važi:*

(a) $\forall l.\ l \in M \implies F$ @ (decisionsTo l M) $\vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za valuaciju* M*),*

(b) $M \vDash_{\neg} F$

tada važi $\neg(\mathsf{sat}\ (F\ @\ (\mathsf{decisions}\ M)))$

Dokaz. Pošto, na osnovu pretpostavke, važi $M \models\neg F$, na osnovu monotonosti, takođe važi $M \models\neg(F\ @\ \ (\mathsf{decisions}\ M))$. Pošto je, na osnovu pretpostavke, $Invarijanta_{impliedLiterals}$ zadovoljena za M, na osnovu leme 7.4 važi da je valuacija M posledica formule $F\ @\ (\mathsf{decisions}\ M)$, odnosno $F\ @\ (\mathsf{decisions}\ M) \models M$. Pošto je formula $F\ @\ (\mathsf{decisions}\ M)$ netačna u valuaciji M koja joj je semantička posledica, ona na osnovu stava 6.8 ne može biti zadovoljiva. □

Sada će biti formulisane i dokazane pomenute leme koje obezbeđuju važenje leme 7.2 tj. obezbeđuju da invarijante ostaju ispunjene nakon primene bilo kog pravila.

Dokazaćemo leme koje se odnose isključivo na invarijantu $Invarijanta_{impliedLiterals}$. Naime, za ostale invarijante je prilično jednostavno pokazati da ostaju očuvane, tako da te dokaze preskačemo. Sledi dokaz da pravila Decide i Backtrack održavaju invarijantu $Invarijanta_{impliedLiterals}$.

Invarijanta$_{\text{impliedLiterals}}$

Decide

Lema 7.6. *Ukoliko važi*

(a) $\forall l.\ l \in M \implies F\ @\ (\mathsf{decisionsTo}\ l\ M) \models l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M*),*

(b) $l \notin M$

(c) $M' = M\ @\ l^{\top}$

tada $\forall l'.\ l' \in M' \implies F\ @\ (\mathsf{decisionsTo}\ l'\ M') \models l'$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M'*),*

Dokaz. Neka je $l' \in M'$. Razmotrimo sledeća dva slučaja:

1. Ako je $l' \in M$, tada je $(\mathsf{decisionsTo}\ l'\ M') = (\mathsf{decisionsTo}\ l'\ M)$, tako da tvrđenje važi na osnovu pretpostavke $Invarijanta_{impliedLiterals}$ za M.
2. U suprotnom je $l' = l$. Pošto važi $l \notin M$ i pošto se l u M' dodaje kao pretpostavljeni literal, važi da je $(\mathsf{decisionsTo}\ l\ M') = (\mathsf{decisions}\ M)\ @\ l$. Kako trivijalno važi da $F\ @\ (\mathsf{decisions}\ M)\ @\ l \models l$, tvrđenje je dokazano.

□

Backtrack

Lema 7.7. *Ukoliko važi*

(a) $(\mathsf{consistent}\ M)$ *(tj.* $Invarijanta_{consistent}$ *je zadovoljena),*

(b) $(\mathsf{distinct}\ M)$ *(tj.* $Invarijanta_{distinct}$ *je zadovoljena)*

(c) $\forall l.\ l \in M \implies F\ @\ (\mathsf{decisionsTo}\ l\ M) \models l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M*),*

(d) $M \models\neg F$*,*

(e) $M' = (\mathsf{prefixBeforeLastDecision}\ M)\ @\ \overline{(\mathsf{lastDecision}\ M)}^{\bot}$,

tada $\forall l'.\ l' \in M' \implies F\ @\ (\mathsf{decisionsTo}\ l'\ M') \vDash l'$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M'*).*

Dokaz. Neka je $l' \in M'$. Razmotrimo sledeća dva slučaja:

1. Ako je l' element liste $(\mathsf{prefixBeforeLastDecision}\ M)$, tada važi da $l' \in M$, $(\mathsf{decisionsTo}\ l'\ M') = (\mathsf{decisionsTo}\ l'\ M)$ i uslov $F\ @\ (\mathsf{decisionsTo}\ l'\ M) \vDash l'$ trivijalno važi jer je $Invarijanta_{impliedLiterals}$ zadovoljena za M na osnovu pretpostavke.

2. Označimo sa l literal $(\mathsf{lastDecision}\ M)$. Tada je $l' = \overline{l}$ i potrebno je pokazati da važi $F\ @\ (\mathsf{decisionsTo}\ \overline{l}\ M') \vDash \overline{l}$. Pošto je l poslednji pretpostavljeni literal u M, koji se na osnovu $(\mathsf{distinct}\ M)$ ne javlja prethodno u M, važi da je $(\mathsf{decisionsTo}\ l\ M) = (\mathsf{decisions}\ M)$.

   S druge strane, pošto se literal $\overline{l}$ u M' postavlja kao izvedeni literal, koji se zbog $(\mathsf{consistent}\ M)$ ne javlja u M, važi da je $(\mathsf{decisionsTo}\ \overline{l}\ M') = (\mathsf{decisions}\ (\mathsf{prefixBeforeLastDecision}\ M)) = (\mathsf{decisions}\ M) \setminus l$.

   Pošto je $Invarijanta_{impliedLiterals}$ zadovoljena za M i važi $M \vDash\neg F$, na osnovu leme 7.5 sledi da formula $F\ @\ (\mathsf{decisions}\ M)$ nije zadovoljiva. Zato, na osnovu stava 6.8 mora da važi $F\ @\ ((\mathsf{decisions}\ M) \setminus l) \vDash \overline{l}$, što je upravo tvrđenje koje je trebalo dokazati.

□

Dokazana tvrđenja obezbeđuju važenje leme 7.2, pa samim tim i leme 7.3. Na osnovu ovoga se može dokazati i saglasnost procedure.

Naredna lema garantuje saglasnost za zadovoljive formule.

Lema 7.8. *Ukoliko važi* $DecisionVars \supseteq (\mathsf{vars}\ F_0)$ *i*

(a) $(\mathsf{consistent}\ M)$ *(tj.* $Invarijanta_{consistent}$ *je zadovoljena),*

(b) $F \equiv F_0$ *(tj.* $Invarijanta_{equiv}$ *je zadovoljena),*

(c) $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsF}$ *je zadovoljena),*

(d) $M \nvDash\neg F$,

(e) Ne postoji literal l *takav da važi* $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ *i* $\overline{l} \notin M$ *(tj. pravilo* Decide *nije moguće primeniti).*

tada je M *model formule* F_0 *(tj.* $(\mathsf{model}\ M\ F_0)$*).*

Dokaz. Pošto pravilo Decide ne može da se primeni, važi da je $(\mathsf{vars}\ M) \supseteq DecisionVars = (\mathsf{vars}\ F_0) \cup DecisionVars \supseteq (\mathsf{vars}\ F)$. Dakle, pošto je M totalna valuacija u odnosu na promenljive koje se pojavljuju u F, formula F je u njoj ili tačna (tj. $M \vDash F$) ili netačna (tj. $M \vDash\neg F$). Pošto važi $M \nvDash\neg F$, mora da važi $M \vDash F$. Pošto važi i $(\mathsf{consistent}\ M)$, valuacija M je model formule F (tj. važi $(\mathsf{model}\ M\ F)$). Pošto važi da su formule F i F_0 logički ekvivalentne (tj. važi $F \equiv F_0$), važi i $(\mathsf{model}\ M\ F_0)$ pa je formula F_0 zadovoljiva (tj. $(\mathsf{sat}\ F_0)$). □

Naredna lema garantuje saglasnost za nezadovoljive formule.

Lema 7.9. *Ukoliko važi*

(a) $F \equiv F_0$ *(tj.* $Invarijanta_{equiv}$ *je zadovoljena),*

(b) $\forall l.\ l \in M \implies F \ @\ (\mathsf{decisionsTo}\ l\ M) \vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena),*

(c) $M \vDash_{\neg} F$,

(d) $(\mathsf{decisions}\ M) = [\,]$,

tada formula F_0 *nije zadovoljiva, tj.* $\neg(\mathsf{sat}\ F_0)$.

Dokaz. Na osnovu leme 7.5 važi da $\neg(\mathsf{sat}\ (F\ @\ (\mathsf{decisions}\ M)))$. Pošto je $(\mathsf{decisions}\ M) = [\,]$, važi $\neg(\mathsf{sat}\ F)$. Pošto je $F \equiv F_0$, na osnovu stava 6.9 važi $\neg(\mathsf{sat}\ F_0)$. □

Napokon, dokazujemo teoremu o saglasnosti:

Teorema 7.1. *Ukoliko* $([\,], F_0) \rightarrow^* (M, F)$, *tada:*

1. *Ukoliko je*

   a) $DecisionVars \supseteq (\mathsf{vars}\ F_0)$,

   b) $M \nvDash_{\neg} F$ *i*

   c) pravilo Decide *ne može da se primeni (tj. ne postoji literal* l *takav da je* $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ *i* $\overline{l} \notin M$*),*

   tada je formula F_0 *zadovoljiva i* M *je njen model (tj. važi* $(\mathsf{sat}\ F_0)$ *i* $(\mathsf{model}\ M\ F_0)$*).*

2. *Ukoliko važi*

   a) $M \vDash_{\neg} F$,

   b) pravilo Backtrack *ne može da se primeni (tj.* $(\mathsf{decisions}\ M) = [\,]$*),*

   tada je formula F_0 *nezadovoljiva (tj. važi* $\neg(\mathsf{sat}\ F_0)$*).*

Dokaz. Na osnovu leme 7.3 važi da su invarijante zadovoljene u stanju (M, F).

Ukoliko je $(\mathsf{vars}\ F_0) \subseteq DecisionVars$, $M \nvDash_{\neg} F$ i pravilo Decide ne može da se primeni, ispunjeni su uslovi za primenu leme 7.8 i time važi $(\mathsf{sat}\ F_0)$ i $(\mathsf{model}\ M\ F_0)$.

Ukoliko važi $M \vDash_{\neg} F_0$ i $(\mathsf{decisions}\ M) = [\,]$, zadovoljeni su uslovi za primenu leme 7.9 i time važi $\neg(\mathsf{sat}\ F_0)$. □

7.2.2 Zaustavljanje osnovne pretrage

Cilj ovog poglavlja je da dokažemo teoremu o zaustavljanju osnovne pretrage koja govori o tome da za proizvolju polaznu formulu F_0 sistem iz početnog stanja $([\,], F_0)$ nakon konačnog broja koraka dolazi u neko završno stanje.

Teorema 7.2. *Relacija $\rightarrow$ je dobro zasnovano uređenje, odnosno ne postoji beskonačan lanac $([\,], F_0) \rightarrow (M_1, F_1) \rightarrow (M_2, F_2) \rightarrow \ldots$.*

Ova teorema će biti dokazana tako što će biti konstruisano dobro zasnovano uređenje (označavaćemo ga sa $\succ^{Vbl}_{trail}$) označenih valuacija M i pokazati da važi

$$(M_1, F_1) \rightarrow (M_2, F_2) \implies M_1 \succ^{Vbl}_{trail} M_2.$$

Na kraju poglavlja, ova teorema će ponovo biti navedena i tada će biti izložen i njen formalni dokaz. U nastavku teksta ćemo opisati izgradnju relacije $\succ^{Vbl}_{trail}$.

Uvedimo najpre uređenje na skupu označenih literala.

Definicija 7.16. $l_1 \prec^{lit} l_2 \iff (\mathsf{isDecision}\ l_1) \wedge \neg(\mathsf{isDecision}\ l_2)$

Sada ćemo uvesti definiciju *leksikografskog proširenja (eng. lexicographic extension)* relacije koja će nam omogućiti da definišemo uređenje $\succ_{trail}$ koje će biti u skladu sa primenom pravila prelaska tj., takvo da se nakon svake primena pravila prelaska komponenta M stanja u odnosu na njega smanjuje.

Definicija 7.17. *Ako je $<$ uređenje skupa X tada je njegovo* leksikografsko proširenje *$<_{lex}$ relacija definisana na skupu listi čiji su elementi iz skupa X i definisano je sa:*

$$\begin{aligned} s <_{lex} t \quad \iff \quad & (\exists\ r.\ t = s \mathbin{@} r \ \wedge\ r \neq [\,]) \ \vee \\ & (\exists\ r\, s'\, t'\, a\, b.\ s = r \mathbin{@} a \mathbin{@} s' \ \wedge\ t = r \mathbin{@} b \mathbin{@} t' \ \wedge\ a < b) \end{aligned}$$

Sada ćemo uvesti relaciju leksikografskog uređenja označenih valuacija.

Definicija 7.18.

$$M_1 \succ_{trail} M_2 \iff M_1 \prec^{lit}_{lex} M_2,$$

pri čemu je $\prec^{lit}_{lex}$ leksikografsko proširenje relacije $\prec^{lit}$.

U nastavku će biti dokazane leme koje opisuju svojstva relacije $\succ_{trail}$ koja će biti iskorišćena da se pokaže da pravila Decide, UnitPropagate, Backtrack i Backjump umanjuju označenu valuaciju u odnosu na relaciju $\succ_{trail}$.

Lema 7.10. *Za proizvoljne označene valuacije M i M', pri čemu je $M' \neq [\,]$ važi:*

$$M \succ_{trail} M \mathbin{@} M'$$

Dokaz. Potrebno i dovoljno je dokazati da je $M \prec^{lit}_{lex} M \mathbin{@} M'$. Pošto važi da je $M' \neq [\,]$, tvrđenje neposredno sledi na osnovu prvog slučaja u definiciji leksikografskog proširenja relacije. □

Lema 7.11. *Neka je M označena valuacija i $P = (\mathsf{prefixToLevel}\ level\ M)$, pri čemu je $0 \leq level < (\mathsf{currentLevel}\ M)$. Tada je*

$$M \ \succ_{trail} \ P \mathbin{@} l^{\perp}$$

Dokaz. Potrebno i dovoljno je dokazati da je $M \prec^{lit}_{lex} P @ l^{\perp}$. Pošto važi $0 \leq level < (\mathsf{currentLevel}\ M)$, na osnovu stava 7.7 važi da postoji označena valuacija M', takva da važi $M = P @ M'$, pri čemu je $M' \neq [\,]$ i $(\mathsf{isDecision}\ (\mathsf{head}\ M'))$. Pošto važi $(\mathsf{isDecision}\ (\mathsf{head}\ M'))$ i $\neg(\mathsf{isDecision}\ l^{\perp})$, na osnovu definicije relacije $\prec^{lit}$, važi da je $(\mathsf{head}\ M') \prec^{lit} l^{\perp}$. Tvrđenje tada važi na osnovu drugog slučaja u definiciji leksikografskog proširenja relacije. □

Sada možemo da dokažemo teoremu koja povezuje relacije $\rightarrow$ i $\succ_{trail}$.

Lema 7.12. *Ako su M_1 i M_2 označene valuacije i F_1 i F_2 formule, tada važi*

$$(M_1, F_1) \rightarrow (M_2, F_2) \implies M_1 \succ_{trail} M_2$$

Dokaz. Pošto važi $(M_1, F_1) \rightarrow (M_2, F_2)$, važi ili $(\mathsf{decide}\ (M_1, F_1)\ (M_2, F_2))$ ili $(\mathsf{backtrack}\ (M_1, F_1)\ (M_2, F_2))$.

Ukoliko važi $(\mathsf{decide}\ (M_1, F_1)\ (M_2, F_2))$, tada je za neki literal l, $M_2 = M_1 @ l^{\top}$, tako da na osnovu leme 7.10 važi $M_1 \succ_{trail} M_2$.

Ukoliko važi $(\mathsf{backtrack}\ (M_1, F_1)\ (M_2, F_2))$, tada je

$$M_2 = (\mathsf{prefixBeforeLastDecision}\ M_1) @ \overline{(\mathsf{lastDecision}\ M_1)}^{\perp}.$$

Pošto je $(\mathsf{decisions}\ M_1) \neq [\,]$, važi da je $(\mathsf{currentLevel}\ M_1) > 1$. Zato je i

$$(\mathsf{prefixBeforeLastDecision}\ M_1) = (\mathsf{prefixToLevel}\ ((\mathsf{currentLevel}\ M_1) - 1)\ M_1),$$

i na osnovu leme 7.11, važi $M_1 \succ_{trail} M_2$. □

Činjenica da je $\prec^{lit}$ dobro zasnovano uređenje ne može iskoristiti da bi se pokazalo da je $\succ_{trail}$ dobro zasnovano uređenje, jer je relacija $\succ_{trail}$ definisana preko relacije $\prec^{lit}$, a ne preko relacije $\succ^{lit}$. Čak i da to nije bio slučaj, leksikografsko proširenje relacije dobro zasnovanog uređenja ne mora da bude dobro zasnovano uređenje kao što to pokazuje naredni primer.

Primer 7.5. Neka je $1 > 0$. Niz $1 >_{lex} 01 >_{lex} 001 >_{lex} 0001 >_{lex} \ldots$ je beskonačna regresija u relaciji $>_{lex}$.

Kao što je pokazano u poglavlju 7.2.1, tokom rada osnovne pretrage invarijante $Invarijanta_{consistent}$ (tj. $(\mathsf{consistent}\ M)$) i $Invarijanta_{distinct}$ (tj. $(\mathsf{distinct}\ M)$) su zadovoljene. Ako izvršimo restrikciju uređenja $\succ_{trail}$ na skup označenih valuacija koje su neprotivrečne, nemaju duplikata i sadrže samo promenljive skupa Vbl, dobijamo relaciju uređenja $\succ^{Vbl}_{trail}$ koja je i dalje saglasna sa relacijom $\rightarrow$. Za razliku od relacije $\succ_{trail}$, relacija $\succ^{Vbl}_{trail}$ je definisana na konačnom skupu i na osnovu toga će jednostavno biti pokazano da čini dobro zasnovano uređenje. Narednom definicijom formalno uvodimo relaciju $\succ^{Vbl}_{trail}$.

Definicija 7.19.

$$\begin{aligned} M_1 \succ^{Vbl}_{trail} M_2 \quad \Longleftrightarrow \quad & (\mathsf{consistent}\ M_1) \wedge (\mathsf{distinct}\ M_1) \wedge (\mathsf{vars}\ M_1) \subseteq Vbl \\ & (\mathsf{consistent}\ M_2) \wedge (\mathsf{distinct}\ M_2) \wedge (\mathsf{vars}\ M_2) \subseteq Vbl \\ & M_1 \succ_{trail} M_2 \end{aligned}$$

Narednu lemu dajemo bez dokaza.

Lema 7.13. *Ako je dat konačan skup promenljivih Vbl tada je skup*

$$\{M \mid (\mathsf{vars}\ M) \subseteq Vbl\ \wedge\ (\mathsf{distinct}\ M)\ \wedge\ (\mathsf{consistent}\ M)\}$$

konačan.

Pokažimo da je relacija $\succ_{trail}$ aciklična.

U tom cilju, pokažimo prvo da su obe relacije $\prec^{lit}$ i $\succ_{trail}$ tranzitivne i irefleksivne.

Bez dokaza koristimo naredni stav.

Stav 7.8.

- *Leksikografsko proširenje tranzitivne relacije je tranzitivna relacija.*
- *Leksikografsko proširenje irefleksivne relacije je irefleksivna relacija.*

Lema 7.14. *$\prec^{lit}$ je tranzitivna i irefleksivna relacija.*

Dokaz. Ako je $l_1 \prec^{lit} l_2$ onda važi $\neg(\mathsf{isDecision}\ l_2)$. Ukoliko bi važilo $l_2 \prec^{lit} l_3$, tada bi važilo i $(\mathsf{isDecision}\ l_2)$ što je kontradikcija.

Ukoliko bi važilo $l \prec^{lit} l$, tada bi istovremeno važilo i $(\mathsf{isDecision}\ l)$ i $\neg(\mathsf{isDecision}\ l)$, što je kontradikcija. □

Lema 7.15. *$\succ_{trail}$ je tranzitivna relacija i irefleksivna relacija.*

Dokaz. Tvrđenje neposredno sledi na osnovu definicije relacije $\succ_{trail}$, stava 7.8 i leme 7.14. □

Lema 7.16. *$\succ_{trail}$ je aciklična relacija, odnosno ne postoji M takav da važi $M \succ^{+}_{trail} M$.*

Dokaz. Pošto je na osnovu leme 7.15 relacija $\succ_{trail}$ tranzitivna, ona je jednaka svom tranzitivnom zatvorenju, tj. važi $\succ^{+}_{trail} \equiv \succ_{trail}$. Ako bi važilo $M \succ^{+}_{trail} M$, tada bi važilo i $M \succ_{trail} M$, što je nemoguće jer je na osnovu leme 7.15 relacija $\succ_{trail}$ irefleksivna. □

Sada možemo da pokažemo da je $\succ^{Vbl}_{trail}$ relacija dobro zasnovanog uređenja.

Lema 7.17. *Ako je skup Vbl konačan, relacija $\succ^{Vbl}_{trail}$ je relacija dobro zasnovanog uređenja.*

Dokaz. Pošto je na osnovu leme 7.16 relacija $\succ_{trail}$ aciklična, aciklična je i njena restrikcija $\succ^{Vbl}_{trail}$. Na osnovu leme 7.13 skup

$$\{M \mid (\mathsf{vars}\ M) \subseteq Vbl\ \wedge\ (\mathsf{distinct}\ M)\ \wedge\ (\mathsf{consistent}\ M)\}$$

je konačan. Iz ovoga sledi da je $\succ^{Vbl}_{trail}$ aciklično uređenje na konačnom skupu, te je stoga i dobro zasnovano uređenje. □

Invarijante obezbeđuju da u svakom stanju važi $(\mathsf{consistent}\ M)$ i $(\mathsf{distinct}\ M)$. Potrebno je još pokazati da su svi literali označene valuacije M elementi nekog konačnog skupa. Pokazaćemo da važi sledeća invarijanta:

$$Invarijanta_{varsM}\colon\quad (\mathsf{vars}\ M)\ \subseteq\ (\mathsf{vars}\ F_0) \cup DecisionVars$$

Pošto je u početnom stanju $M = [\,]$, lema 7.1 i dalje važi.

Dokaz leme 7.2 je neophodno proširiti tako da obuhvati i slučajeve koji odgovaraju invarijanti $Invarijanta_{varsM}$, što ćemo implicitno uraditi u tekstu koji sledi.

Dokažimo da $Invarijanta_{varsM}$ ostaje zadovoljena nakon primene oba pravila.

Invarijanta$_{\text{varsM}}$

Decide

Lema 7.18. *Ako važi:*

(a) $(\mathsf{vars}\ M) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsM}$ *je zadovoljena za* M*),*

(b) $(\mathsf{var}\ l) \in DecisionVars$*,*

(c) $M' = M\ @\ l^{\top}$*,*

tada važi $(\mathsf{vars}\ M') \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsM}$ *je zadovoljena za* M'*).*

Dokaz. Važi $(\mathsf{vars}\ M') = (\mathsf{vars}\ M) \cup \{(\mathsf{var}\ l)\}$, pa tvrđenje trivijalno sledi na osnovu pretpostavki. □

Backtrack

Lema 7.19. *Ako važi:*

(a) $(\mathsf{vars}\ M) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsM}$ *je zadovoljena za* M*),*

(b) $M' = (\mathsf{prefixBeforeLastDecision}\ M)\ @\ \overline{(\mathsf{lastDecision}\ M)}^{\perp}$*,*

tada važi $(\mathsf{vars}\ M') \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsM}$ *je zadovoljena za* M'*).*

Dokaz. Važi $(\mathsf{vars}\ M') \subseteq (\mathsf{vars}\ M)$, pa tvrđenje trivijalno sledi na osnovu pretpostavki. □

Na osnovu dokazanih lema 7.18 i 7.18, lema 7.2, a samim tim i lema 7.3 važe i za prošireni skup invarijanti.

Pre dokaza teoreme o zaustavljanju, dokažimo dve jednostavne pomoćne leme o dobrim uređenjima.

Lema 7.20. *Neka je* $>$ *uređenje definisano na skupu* n*-torki na sledeći način.*

$$(x_1, \ldots, x_i, \ldots, x_n) > (x'_1, \ldots, x'_i, \ldots, x'_n) \iff x_i >^i x'_i,$$

pri čemu je $>^i$ *neko dobro zasnovano uređenje. Tada je* $i >$ *dobro zasnovano uređenje.*

Dokaz. Dokažimo princip najmanjeg elementa za relaciju $>$:

$$\forall\, Q.\ (\exists\, a \in Q) \implies (\exists\, a_{min} \in Q.\ (\forall\, a'.\ a_{min} > a' \implies a' \notin Q))$$

Neka je Q skup n-torki i a jedan njegov element. Neka je

$$Q_i = \{x_i \mid (x_1, \ldots, x_i, \ldots, x_n) \in Q\}.$$

Skup Q_i je neprazan jer svakako sadrži i-tu koordinatu elementa a. Tada, na osnovu principa najmanjeg elementa za $>^i$, važi da postoji x_i^{min}, takav da je

$$x_i^{min} \in Q_i \qquad \text{i} \qquad \forall\, x_i'.\ x_i^{min} > x_i' \implies x_i' \notin Q_i.$$

Pošto je $x_i^{min} \in Q_i$, postoji neki element $a_{min} \in Q$, čija je i-ta koordinata x_i^{min}, tj.

$$a_{min} = (x_1, \ldots, x_i^{min}, \ldots, x_n).$$

Tvrdimo da je a_{min} minimalni element skupa Q. Ukoliko bi postojao $a' = (x_1', \ldots, x_i', \ldots, x_n')$, takav da je $a' \in Q$ i $a_{min} > a'$, tada bi važilo da je $x_i^{min} > x_i'$. Međutim, tada bi sledilo da $x_i' \notin Q_i$, iako postoji elemenat skupa Q (to je element a'), takav da mu je i-ta koordinata jednaka x_i', što je u kontradikciji sa definicijom skupa Q_i. Dakle, ako je $a_{min} > a'$, tada $a' \notin Q$. □

Lema 7.21. *Ako važi $x > y \implies x \succ y$ i $\succ$ je dobro zasnovano uređenje, tada je $i >$ dobro zasnovano uređenje.*

Dokaz. Dokažimo princip najmanjeg elementa za relaciju $>$, tj.

$$\forall\, Q.\ (\exists\, a \in Q) \implies (\exists\, a_{min} \in Q.\ (\forall\, a'.\ a_{min} > a' \implies a' \notin Q)).$$

Neka je Q skup i a njegov element. Na osnovu principa najmanjeg elementa za $\succ$, važi da postoji a_{min}, takav da važi $a_{min} \in Q$ i $\forall\, a'.\ a_{min} \succ a' \implies a' \notin Q$. Tvrdimo da je a_{min} najmanji element i za relaciju $>$. Zaista, neka je a' element takav da važi $a_{min} > a'$. Na osnovu pretpostavke važi i $a_{min} \succ a'$ i sledi da $a' \notin Q$. □

Sada je moguće dokazati teoremu o zaustavljanju.

Teorema 7.2. *Relacija $\to$ je dobro zasnovano uređenje, tj. ne postoji beskonačan lanac $([\,], F_0) \to (M_1, F_1) \to (M_2, F_2) \to \ldots$.*

Dokaz. Definišimo relaciju

$$(M_1, F_1) \succ (M_2, F_2) \iff M_1 \succ_{trail}^{Vbl} M_2.$$

Pošto je, na osnovu leme 7.17, $\succ_{trail}^{Vbl}$ dobro zasnovano uređenje, na osnovu leme 7.20, i $\succ$ je dobro zasnovano uređenje. Pokažimo još da ako (M_1, F_1) zadovoljava invarijante, onda važi

$$(M_1, F_1) \to (M_2, F_2) \implies (M_1, F_1) \succ (M_2, F_2).$$

Zaista, na osnovu leme 7.12, sledi da važi $M_1 \succ_{trail} M_2$. Na osnovu leme 7.2 sledi da su invarijante zadovoljene i u stanju (M_2, F_2). Invarijante obezbeđuju da važi (consistent M_1), (distinct M_1), (vars M_1) $\subseteq$ (vars F_0) $\cup$ $DecisionVars$, (consistent M_2), (distinct M_2) i (vars M_2) $\subseteq$ (vars F_0) $\cup$ $DecisionVars$. Zbog toga je $M_1 \succ_{trail}^{Vbl} M_2$. Tvrđenje onda sledi na osnovu leme 7.21 i leme 7.3, odnosno činjenice da sva stanja dostupna iz početnog zadovoljavaju invarijante $Invarijanta_{consistent}$, $Invarijanta_{distinct}$ i $Invarijanta_{varsM}$. □

7.2.3 Potpunost osnovne pretrage

Sada će biti dokazano svojstvo potpunosti koje govori da su sva završna stanja ili prihvatajuća ili odbacujuća.

Teorema 7.3. *U završnom stanju (M, F) važi ili*

(a) $M \models_{\neg} F$ *i* $(\mathsf{decisions}\ M) = [\,]$ *ili*

(b) $M \not\models_{\neg} F$ *i ne postoji literal takav da je* $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ *i* $\overline{l} \notin M$.

Dokaz. Neka je (M, F) završno stanje. Trivijalno važi ili $M \models_{\neg} F$ ili $M \not\models_{\neg} F$. Ukoliko važi $M \models_{\neg} F$, pošto pravilo Backtrack ne može da se primeni jer je stanje završno, mora da važi $(\mathsf{decisions}\ M) = [\,]$. Ukoliko važi $M \not\models_{\neg} F$, pošto pravilo Decide ne može da se primeni jer je stanje završno, mora da važi da ne postoji literal takav da je $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$. □

Kao što je već rečeno, potpunost za pojedinačne vrste odgovora je jednostavna posledica upravo dokazane potpunosti, zaustavljanja i saglasnosti. Potpunost za nezadovoljive formule garantuje da će se sistem zaustaviti u odbacujućem stanju, a za zadovoljive formule da će se sistem zaustaviti u prihvatajućem stanju.

Teorema 7.4. *Za formulu F_0 važi da $([\,], F_0) \rightarrow^* (M, F)$, pri čemu je (M, F) završno stanje i važi:*

1. *Ako je formula F_0 zadovoljiva, tada važi $M \not\models_{\neg} F$ i ne postoji literal takav da je $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$.*

2. *Ako je formula F_0 nezadovoljiva i $(\mathsf{vars}\ F_0) \subseteq DecisionVars$, tada važi $M \models_{\neg} F$ i $(\mathsf{decisions}\ M) = [\,]$.*

Dokaz. Na osnovu teoreme 7.2 važi da postoji broj n i završno stanje (M, F) tako da $([\,], F_0) \rightarrow^n (M, F)$ i (M, F) je završno stanje. U završnom stanju (M, F), na osnovu teoreme 7.3 važi

(a) $M \not\models_{\neg} F$ i ne postoji literal l takav da je $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$

(b) ili $M \models_{\neg} F$ i $(\mathsf{decisions}\ M) = [\,]$.

Pretpostavimo da je F_0 zadovoljiva formula, a da važi $M \models_{\neg} F$ i $(\mathsf{decisions}\ M) = [\,]$. Tada, međutim, na osnovu teoreme 7.1, važi da je F_0 nezadovoljiva, što je u kontradikciji sa našom polaznom pretpostavkom. Dakle za zadovoljive formule mora da važi $M \not\models_{\neg} F$ i ne postoji literal l takav da je $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$.

Slično, pretpostavimo da je F_0 nezadovoljiva formula i $DecisionVars \supseteq (\mathsf{vars}\ F_0)$, a da važi $M \not\models_{\neg} F$ i ne postoji literal l takav da je $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$. Na osnovu teoreme 7.1 važi da je F_0 zadovoljiva formula, što je u kontradikciji sa našom polaznom pretpostavkom. Dakle, za nezadovoljive formule mora da važi $M \models_{\neg} F$ i $(\mathsf{decisions}\ M) = [\,]$. □

7.2.4 Korektnost osnovne pretrage

Teorema 7.5. *Neka je* (vars F_0) $\subseteq DecisionsVars$. *Za svaku formulu* F_0 *postoji završno stanje* (M, F) *tako da važi* $([], F_0) \rightarrow^* (M, F)$ *i formula* F_0 *je zadovoljiva ako i samo ako važi* $M \not\models \neg F$.

Dokaz. Dokaz sledi na osnovu teorema 7.1, 7.2 i 7.3. □

7.3 DPLL pretraga (UnitPropagate, PureLiteral)

U ovom poglavlju ćemo dati opis DPLL procedure pretrage koja je opisana u poglavlju 4.3.2 i koja odgovara rekurzivno opisanom algoritmu datom u poglavlju 4.3.1, ali ovaj put korišćenjem sistema pravila prelaza, što omogućava nerekurzivnu implementaciju. Osnovu ove procedure, naravno, čini osnovna pretraga.

Pored pravila Decide i Backtrack, DPLL pretraga koristi još i pravilo:

UnitPropagate:
$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F, \quad \overline{l}_1, \ldots, \overline{l}_k \in M, \quad l, \overline{l} \notin M}{M := M\ l}$$

Pravilo UnitPropagate koristi jedinične klauze kao mehanizam zaključivanja. Međutim, pošto se prilikom nerekurzivne implementacije procedure formula F i njene klauze ne menjaju, definiciju jediničnih klauza (definicija 10.3) je neophodno izmeniti. Naime, klauza je jedinična u odnosu na valuaciju v ako su joj svi literali netačni u valuaciji v, osim jednog literala koji je nedefinisan u valuaciji v.

Definicija 7.20. *Klauza* c *je* jedinična klauza (unit clause) *u valuaciji* v *sa* jediničnim literalom (unit literal) l, *što označavamo sa* (isUnit $c\ l\ v$), *akko* $l \in c$, $v \not\models l$, $v \not\models \neg l$ *i* $v \models \neg (c \setminus l)$ *(tj.* $\forall l'.\ l' \in c \wedge l' \neq l \Rightarrow v \models \neg l'$*).*

Da bi jedinična klauza bila tačna u valuaciji njen jedinični literal mora biti tačan i zato ga pravilo UnitPropagate postavlja u valuaciju kao izvedeni literal.

Sada će pravilo UnitPropagate biti formalno uvedeno kao relacija između stanja.

Definicija 7.21.
(unitPropagate $(M_1, F_1)\ (M_2, F_2)$) $\iff$

$$\exists c\ l. \qquad \begin{array}{l} c \in F_1 \ \wedge\ (\text{isUnit}\ c\ l\ M_1) \ \wedge \\ F_2 = F_1 \ \wedge\ M_2 = M_1\ @\ l^{\perp} \end{array}$$

Narednom definicijom se menja definicija 7.14 relacije $\rightarrow$ tako da obuhvati i pravilo UnitPropagate.

Definicija 7.14'.

$$\begin{array}{lll} (M_1, F_1) \rightarrow (M_2, F_2) & \iff & (\text{decide}\ (M_1, F_1)\ (M_2, F_2)) \ \vee \\ & & (\text{backtrack}\ (M_1, F_1)\ (M_2, F_2)) \ \vee \\ & & (\text{unitPropagate}\ (M_1, F_1)\ (M_2, F_2)) \end{array}$$

U nastavku teksta, svako pominjanje relacije $\longrightarrow$ će se odnositi na upravo uvedenu definiciju, sve dok se naglasi drugačije.

Uvođenje novog pravila ne menja raniju definiciju prihvatajućih i odbacujućih stanja.

7.3.1 Saglasnost DPLL pretrage

Teorema saglasnosti 7.1 i dalje ostaje na snazi i kada se uključi pravilo UnitPropagate. Dokaz se opet zasniva na lemama 7.3, 7.8 i 7.9. Pošto se dokaz leme 7.3 oslanja na dokaz leme 7.2, potrebno je dopuniti dokaz leme 7.2 tako što će se dokazati da pravilo UnitPropagate održava invarijante. Opet ćemo to eksplicitno pokazati samo za invarijantu $Invarijanta_{impliedLiterals}$, dok se dokazi za ostale invarijante veoma jednostavno izvode.

Najpre ćemo pokazati lemu koja se koristi u nekoliko dokaza koji slede.

Lema 7.22. *Ukoliko važi*

(a) c *je klauza za koju važi* $F \models c$ *i*

(b) c *je jedinična klauza u valuaciji* v *sa jediničnim literalom* l*, tj.* $(\mathsf{isUnit}\ c\ l\ v)$*,*

tada važi $F\ @\ v \models l$*.*

Dokaz. Neka je v' model formule $F\ @\ v$. To znači da je v' model formule F i da v' sadrži sve literale valuacije v. Pošto važi $F \models c$ i pošto važi $v' \models F$, mora da važi i $v' \models c$. Ali, pošto važi $(\mathsf{isUnit}\ c\ l\ v)$, svi literali klauze c različiti od literala l su netačni u v. Pošto je v sadržano u v', svi literali klauze c različiti od literala l su netačni i u v'. Pošto je v' model, važi $(\mathsf{consistent}\ v')$, tako da nijedan literal iz c, različit od literala l ne može da bude tačan u v'. Tada, pošto je $v' \models c$, mora da važi da je l tačan u v', tj. $v' \models l$. □

Invarijanta$_{\text{impliedLiterals}}$
UnitPropagate

Sada ćemo pokazati da $Invarijanta_{impliedLiterals}$ ostaje zadovoljena nakon primene pravila UnitPropagate.

Lema 7.23. *Ukoliko važi*

(a) $\forall l'.\ l' \in M \implies F\ @\ (\mathsf{decisionsTo}\ l'\ M) \models l'$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M*),*

(b) c *je klauza za koju važi*[3] $F \models c$*,*

(c) c *je jedinična klauza u* M *sa jediničnim literalom* l*, (tj.* $(\mathsf{isUnit}\ c\ l\ M)$*),*

(d) $M' = M\ @\ l^{\perp}$*,*

tada važi $\forall l'.\ l' \in M' \implies F\ @\ (\mathsf{decisionsTo}\ l'\ M') \models l'$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M'*).*

Dokaz. Za sve literale l' iz M važi (decisionsTo l' M') = (decisionsTo l' M) i uslov F @ (decisionsTo l' M') $\models l'$ na osnovu pretpostavke trivijalno važi. Dakle, potrebno je još pokazati da važi F @ (decisionsTo l M') $\models l$. Pošto l nije pretpostavljeni literal u M', važi (decisionsTo l M') = (decisions M) i potrebno je dokazati F @ (decisions M) $\models l$. Pošto važi $F \models c$ i (isUnit c l M), na osnovu leme 7.22, važi

$$F \text{ @ } M \models l$$

Pošto je $Invarijanta_{impliedLiterals}$ zadovoljena za M, na osnovu leme 7.4, važi

$$F \text{ @ (decisions } M) \models M.$$

Na osnovu monotonosti važi i

$$F \text{ @ (decisions } M) \models F,$$

te, na osnovu stava 6.8, važi i

$$F \text{ @ (decisions } M) \models F \text{ @ } M$$

Pošto F @ (decisions M) $\models F$ @ M i F @ $M \models l$, uslov F @ (decisions M) $\models l$ važi na osnovu stava 6.8 o tranzitivnosti relacije $\models$. □

U poglavlju 7.3.2 o zaustavljanju će biti pokazano da pravilo UnitPropagate održava i invarijante relevantne za zaustavljanje. Zato lema 7.2 važi, a samim tim važi i lema 7.3.

7.3.2 Zaustavljanje DPLL pretrage

Teorema 7.2 o zaustavljanju ostaje na snazi i kada je definicija relacije $\rightarrow$ izmenjena tako da obuhvati i pravilo UnitPropagate. Dokaz se i dalje svodi na korišćenje relacije $\succ_{trail}$, pri čemu čemu je potrebno proširiti dokaz leme 7.12 tako da obuhvati i slučaj kada je primenjeno pravilo UnitPropagate.

Lema 7.12. *Ako su M_1 i M_2 označene valuacije, a F_1 i F_2 formule, tada važi*

$$(M_1, F_1) \rightarrow (M_2, F_2) \implies M_1 \succ_{trail} M_2$$

Dokaz. Ranije navedeni dokaz (dat na strani 64) proširujemo slučajem u kome je primenjeno pravilo UnitPropagate. Ukoliko važi (unitPropagate (M_1, F_1) (M_2, F_2)), tada za neki literal l važi $M_2 = M_1$ @ $l^{\perp}$, tako da na, osnovu leme 7.10, važi $M_1 \succ_{trail} M_2$. □

Da bi ranije navedeni dokaz teoreme 7.2 mogao da se iskoristi i za prošireni sistem, potrebno je još pokazati da invarijante važe u svakom koraku sistema, odnosno potrebno je pokazati da lema 7.2 i samim tim i lema 7.3 važi i kada se uključi pravilo UnitPropagate. Za to je potrebno još pokazati da pravilo UnitPropagate održava invarijantu $Invarijanta_{varsM}$, što će sada i biti urađeno.

$Invarijanta_{varsM}$
UnitPropagate

[3] Primetimo da je uslov $c \in F$ zamenjen slabijim uslovom $F \models c$.

Lema 7.24. *Ako važi:*

(a) $(\mathsf{vars}\ M) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsM}$ *je zadovoljena za* M*),*

(b) $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsF}$ *je zadovoljena za* F*),*

(c) $c \in F$*,*

(d) $(\mathsf{isUnit}\ c\ l\ M)$*,*

(e) $M' = M\ @\ l^{\perp}$*,*

tada važi $(\mathsf{vars}\ M') \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsM}$ *je zadovoljena za* M'*).*

Dokaz. Pošto važi $(\mathsf{isUnit}\ c\ l\ M)$ literal l je član klauze c, tj. $l \in c$. Pošto je $c \in F$, važi $l \in F$ i zato je $(\mathsf{var}\ l) \in (\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$. Pošto je $(\mathsf{vars}\ M') = (\mathsf{vars}\ M) \cup \{(\mathsf{var}\ l)\}$, tvrđenje neposredno sledi na osnovu pretpostavki. □

Ovim važi lema 7.2 i samim tim i lema 7.3. Teorema 7.2 i dalje važi uz ranije navedeni dokaz.

7.3.3 Potpunost DPLL pretrage

Teoreme 7.3 i 7.4 i dalje važe uz nepromenjene dokaze.

7.3.4 Korektnost DPLL pretrage

Teorema 7.5 važi uz neizmenjen dokaz.

7.4 Povratni skokovi (Backjump)

U ovom poglavlju ćemo formalizovati i dokazati korektnost rešavača zasnovanog na povratnim skokovima.

Backjump:
$$\frac{M = M' \,|\, l'\, M'', \quad F \vDash l \vee l_1 \vee \ldots \vee l_k, \quad \overline{l}_1, \ldots, \overline{l}_k \in M', \quad \overline{l} \notin M', \quad (\mathsf{var}\ l) \in Vbl}{M := M'\ l}$$

Ovo pravilo se koristi umesto pravila Backtrack. Za razliku od pravila Backtrack koje poništava samo poslednji nivo tekuće označene valuacije i menja polaritet poslednjem pretpostavljenom literalu, pravilo Backjump može odjednom da poništi nekoliko pretpostavljenih literala i nekoliko nivoa označene valuacije. Primena ovog pravila je određena *klauzom povratnog skoka* (u pravilu označenom sa $l \vee l_1 \vee \ldots \vee l_k$) koja je element formule F (ili je njena posledica) i koja ima osobinu da je jedinična u odnosu na neki prefiks M' označene valuacije M, sa jediničnim literalom l. Pravilo Backjump poništava sve pretpostavljene literale iza prefiksa M' i dodaje l kao izvedeni literal nakon prefiksa M'.

Činjenica da je M' prefiks do nekog nivoa, odnosno da je prvi literal l' koji sledi nakon M' u M pretpostavljeni literal, je potrebna samo da bi se

obezbedilo zaustavljanje procedure. Saglasnost se može pokazati i pod slabijom pretpostavkom koja dozvoljava da je P proizvoljni prefiks označene valuacije M.

Pravilo Backjump u obliku u kojem je uvedeno ne precizira način na koji se određuje klauza povratnog skoka $l \vee l_1 \vee \ldots \vee l_k$. Primetimo da se čak ni ne zahteva da ona bude član formule F, već je dovoljno samo da bude njena semantička posledica. Time je ostavljena mogućnost korišćenja različitih strategija za njen odabir. S druge strane, ovako široko definisano pravilo je prilično daleko od nivoa specifikacije koji je potreban za implementaciju. Zbog toga će u poglavlju 7.7 biti navedena konkretnija varijanta ovog pravila koja klauzu povratnog skoka određuje kao rezultat faze analize konflikta.

Naredna definicija formalno uvodi Backjump pravilo.

Definicija 7.22.

$$(\mathsf{backjump}\ (M_1, F_1)\ (M_2, F_2)) \iff$$

$$\begin{aligned}\exists\ c\ l\ level.\quad & P = (\mathsf{prefixToLevel}\ level\ M)\ \wedge\ 0 \leq level < (\mathsf{currentLevel}\ M)\ \wedge \\ & (\mathsf{isUnit}\ c\ l\ P)\ \wedge \\ & F_1 \models c\ \wedge\ (\mathsf{var}\ l) \in (\mathsf{vars}\ F_0) \cup DecisionVars\ \wedge \\ & F_2 = F_1\ \wedge\ M_2 = P\,@\,l^{\perp}\end{aligned}$$

Sistem sa povratnim skokovima koristi pravila Decide, Backjump i UnitPropagate.

Definicija 7.14".

$$\begin{aligned}(M_1, F_1) \rightarrow (M_2, F_2) \quad \iff \quad & (\mathsf{decide}\ (M_1, F_1)\ (M_2, F_2))\ \vee \\ & (\mathsf{backjump}\ (M_1, F_1)\ (M_2, F_2))\ \vee \\ & (\mathsf{unitPropagate}\ (M_1, F_1)\ (M_2, F_2))\end{aligned}$$

U nastavku teksta, svako pominjanje relacije $\rightarrow$ će se odnositi na upravo uvedenu definiciju, sve dok se naglasi drugačije.

7.4.1 Saglasnost sistema sa povratnim skokovima

Teorema saglasnosti 7.1 i dalje ostaje na snazi i kada se uključi pravilo Backjump. Dokaz se i dalje zasniva na lemi 7.3. Da bi se dokazala lema 7.3, potrebno je izmeniti dokaz leme 7.2 odnosno potrebno je dokazati da pravilo Backjump održava invarijante. I ovaj put će to biti eksplicitno pokazano samo za invarijantu $Invarijanta_{impliedLiterals}$, dok se dokazi za ostale invarijante veoma jednostavno izvode.

Najpre pokažimo da $Invarijanta_{impliedLiterals}$ ostaje zadovoljena kada se umesto označene valuacije posmatra njen proizvoljni prefiks.

Lema 7.25. *Ako važi:*

(a) $\forall l.\ l \in M \implies F\ @\ (\mathsf{decisionsTo}\ l\ M) \models l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M*),*

(b) P *je prefiks označene valuacije* M *(tj.* $(\mathsf{isPrefix}\ P\ M)$*),*

tada važi $\forall l.\ l \in P \implies F \mathbin{@} (\mathsf{decisionsTo}\ l\ P) \vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* P*).*

Dokaz. Neka je l literal takav da važi $l \in P$. Tada važi $l \in M$ i pošto je $Invarijanta_{impliedLiterals}$ zadovoljena za M, važi i $F \mathbin{@} (\mathsf{decisionsTo}\ l\ M) \vDash l$. Takođe važi $(\mathsf{decisionsTo}\ l\ P) = (\mathsf{decisionsTo}\ l\ M)$ i na osnovu toga važi $F \mathbin{@} (\mathsf{decisionsTo}\ l\ P) \vDash l$. Pošto je l proizvoljni literal iz P, tvrđenje sledi. □

Sada pokažimo da pravilo Backjump održava invarijantu $Invarijanta_{impliedLiterals}$.

Invarijanta$_{\mathbf{impliedLiterals}}$

Backjump

Lema 7.26. *Ako važi:*

(a) $\forall l.\ l \in M \implies F \mathbin{@} (\mathsf{decisionsTo}\ l\ M) \vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M*),*

(b) P *je prefiks označene valuacije* M *(tj.* $(\mathsf{isPrefix}\ P\ M)$*),*

(c) Klauza c *je posledica formule* F *(tj.* $F \vDash c$*),*

(d) Klauza c *je jedinična sa jediničnim literalom* l *u odnosu na valuaciju* P *(tj.* $(\mathsf{isUnit}\ c\ l\ P)$*),*

(e) $M' = P \mathbin{@} l^{\perp}$*,*

tada važi $\forall l'.\ l' \in M' \implies F \mathbin{@} (\mathsf{decisionsTo}\ l'\ M') \vDash l'$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M'*),*

Dokaz. Na osnovu leme 7.25 važi $\forall l'.\ l' \in P \implies F \mathbin{@} (\mathsf{decisionsTo}\ l'\ P) \vDash l'$ (tj. $Invarijanta_{impliedLiterals}$ je zadovoljena za P). Na osnovu leme 7.23 o pravilu UnitPropagate, tvrđenje sledi. □

U poglavlju 7.4.2 o zaustavljanju će biti pokazano da pravilo Backjump održava i invarijante koje su relevantne za zaustavljanje. Zato lema 7.2 važi, a na osnovu toga važi i lema 7.3.

7.4.2 Zaustavljanje sistema sa povratnim skokovima

Teorema 7.2 o zaustavljanju ostaje na snazi kada se relacija $\rightarrow$ izmeni tako da obuhvati pravilo Backjump. Dokaz se i dalje svodi na korišćenje relacije $\succ^{Vbl}_{trail}$, pri čemu čemu je potrebno proširiti dokaz leme 7.12 tako da obuhvati i slučaj kada je primenjeno pravilo Backjump.

Lema 7.12. *Ako su* M_1 *i* M_2 *označene valuacije,* F_1 *i* F_2 *formule, tada važi*

$$(M_1, F_1) \rightarrow (M_2, F_2) \implies M_1 \succ_{trail} M_2$$

Dokaz. Ranije navedeni dokaz (dat na strani 71) proširujemo slučajem kada je primenjeno pravilo Backjump. Ukoliko važi $(\mathsf{backjump}\ (M_1, F_1)\ (M_2, F_2))$, tada je za neki literal l, $M_2 = P \mathbin{@} l^{\perp}$, pri čemu je $P = (\mathsf{prefixToLevel}\ level\ M)$ gde je $0 \leq level < (\mathsf{currentLevel}\ M)$. Na osnovu leme 7.11 važi $M_1 \succ_{trail} M_2$. □

Da bi ranije navedini dokaz teoreme 7.2 mogao da se iskoristi, potrebno je još pokazati da važe leme 7.2 i 7.3. Za to je potrebno modifikovati dokaz leme 7.2 i još pokazati da pravilo Backjump održava invarijantu $Invarijanta_{varsM}$, što će sada i biti urađeno.

Invarijanta$_{\text{varsM}}$
Backjump

Lema 7.27. *Ako važi:*

(a) $(\mathsf{vars}\ M) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj. $Invarijanta_{varsM}$ je zadovoljena za M),*

(b) $(\mathsf{isPrefix}\ P\ M)$ *i* $(\mathsf{var}\ l) \in (\mathsf{vars}\ F_0) \cup DecisionVars$,

(c) $M' = P\ @\ l^{\perp}$,

tada važi $(\mathsf{vars}\ M') \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj. $Invarijanta_{varsM}$ je zadovoljena za M').*

Dokaz. Pošto važi $(\mathsf{isPrefix}\ P\ M)$, važi da je $(\mathsf{vars}\ P) \subseteq (\mathsf{vars}\ M) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$. Pošto je $(\mathsf{vars}\ M') = (\mathsf{vars}\ P) \cup \{(\mathsf{var}\ l)\}$, tvrđenje neposredno sledi na osnovu pretpostavki. □

Na osnovu ovoga lema 7.2 važi, a na osnovu toga važi i lema 7.3. Zato važi i teorema 7.2 važi uz nepromenjeni dokaz.

7.4.3 Alternativni uslovi primenljivosti povratnih skokova

Cilj ovog poglavlja je da uvede nekoliko operativnijih uslova koji karakterišu primenljivost povratnog skoka. Jedan od dovoljnih uslova da bi klauza c bila klauza povratnog skoka je da ima tzv. *tačku jedinstvene implikacije (distinct implication point)*.

Definicija 7.23. *Klauza c koja je netačna u valuaciji M ima* tačku jedinstvene implikacije *što označavamo sa* $(\mathsf{isUIP}\ c\ M)$ *ako je nivo njenog poslednjeg pobijenog literala l striktno veći od nivoa njenih ostalih pobijenih literala, tj. važi:*

$$(\mathsf{isUIP}\ l\ c\ M) \iff \begin{array}{l} M \vDash \neg\, c \\ \overline{l} = (\mathsf{lastAssertedLiteral}\ \overline{c}\ M)\ \wedge \\ \forall\, l'.\ l' \in c \setminus l \implies (\mathsf{level}\ \overline{l'}\ M) < (\mathsf{level}\ \overline{l}\ M) \end{array}$$

U slučaju postojanja tačke jedinistvene implikacije, stvara se mogućnost povratnog skoka na neki raniji nivo označene valuacije, pri čemu taj nivo predstavlja tzv. *nivo povratnog skoka (backjump level)* i mora da zadovoljava uslove date narednom definicijom.

Definicija 7.24. Nivo povratnog skoka *za datu klazu povratnog skoka C (netačnu u valuaciji M) je bilo koji nivo koji je strogo manji od nivoa njenog najkasnije pobijenog literala, a veći ili jednak od nivoa njenih ostalih literala, tj. to je prirodan broj level koji zadovoljava:*

(isBackjumpLevel $level$ l c M) $\iff$

$$\begin{aligned}
&M \vDash \neg C \\
&\overline{l} = (\mathsf{lastAssertedLiteral}\ \overline{c}\ M)\ \wedge \\
&0 \leq level < (\mathsf{level}\ \overline{l}\ M)\ \wedge \\
&\forall\ l'.\ l' \in c \setminus l\ \implies\ (\mathsf{level}\ \overline{l'}\ M) \leq level
\end{aligned}$$

Pokažimo da uslov da klauza ima tačku jedinstvene implikacije u svim netrivijalnim slučajevima obezbeđuje da postoji odgovarajući nivo povratnog skoka.

Lema 7.28. *Ukoliko važi:*

(a) (distinct M) *(tj. $Invarijanta_{distinct}$ je zadovoljena),*

(b) (isUIP l c M),

(c) (level $\overline{l}$ M) > 0,

tada postoji broj level takav da važi (isBackjumpLevel $level$ l c M).

Dokaz. Razmotrimo sledeće slučajeve:

1. $c \setminus l = [\,]$. Tada je 0 traženi nivo.
2. $c \setminus l \neq [\,]$. Neka je $\overline{l'} = (\mathsf{lastAssertedLiteral}\ \overline{c \setminus l}\ M)$ i neka je $level = (\mathsf{level}\ \overline{l'}\ M)$. Tada je $l' \in C \setminus l$ i $\overline{l'} \in M$. Pošto je $l' \in C \setminus l$, a važi (isUIP l c M), sledi da je

   $$level = (\mathsf{level}\ \overline{l'}\ M) < (\mathsf{level}\ \overline{l}\ M).$$

   Razmotrimo literal l'' takav da je $l'' \in c \setminus l$. Pošto je $\overline{l'} = (\mathsf{lastAssertedLiteral}\ \overline{c \setminus l}\ M)$ i važi (distinct M), važi da je

   $$(\mathsf{level}\ \overline{l''}\ M) \leq (\mathsf{level}\ \overline{l'}\ M).$$

□

Pokažimo sada da činjenica da je broj *level* nivo povratnog skoka za klauzu C, uz određene dodatne uslove, obezbeđuje da je klauza C jedinična u odgovarajućem prefiksu označene valuacije M. Ovo, uz još nekoliko uslova, daje mogućnost za primenu Backjump pravila.

Lema 7.29. *Ako važi:*

(a) (consistent M) *(tj. $Invarijanta_{consistent}$ je zadovoljena),*

(b) (distinct M) *(tj. $Invarijanta_{distinct}$ je zadovoljena),*

(c) (isBackjumpLevel $level$ l C M),

tada važi (isUnit C l (prefixToLevel $level$ M)).

Dokaz. Pošto važi (isBackjumpLevel $level$ l C M), sledi da važi

$$\overline{l} = (\text{lastAssertedLiteral } \overline{C}\ M),$$

$$0 \leq level < (\text{level } \overline{l}\ M),$$

i

$$\forall\, l'.\ l' \in C \setminus l \implies (\text{level } \overline{l'}\ M) \leq level.$$

Iz uslova $\overline{l} = (\text{lastAssertedLiteral } \overline{C}\ M)$ sledi da je $l \in C$ i $\overline{l} \in M$. Pošto je $\overline{l} \in M$, važi da je $level < (\text{level } \overline{l}\ M) \leq (\text{currentLevel } M)$.

Označimo sa P valuaciju (prefixToLevel $level$ M)).

Pošto važi $0 \leq level < (\text{level } \overline{l}\ M)$ i (distinct M) sledi da

$$\overline{l} \notin P,$$

jer se literal ne može javljati na nivoima nižim od svog nivoa.

Pošto je $\overline{l} \in M$, a važi (consistent M) sledi da važi $l \notin M$ i pošto je P prefiks od M, važi i

$$l \notin P.$$

Pokažimo još da su svi literali iz C različiti od l netačni u prefiksu P. Neka je l' literal iz C različit od l. Pošto je $l' \in C$ i $M \models\neg C$ sledi da je $\overline{l'} \in M$. Tada važi i da je $(\text{level } \overline{l'}\ M) \leq level$. Pošto je $P = (\text{prefixToLevel } level\ M))$, $\overline{l'}$ mora da bude element P tj. važi $P \models\neg l'$ i zato

$$P \models\neg C \setminus l.$$

Na osnovu $l \notin P$, $\overline{l} \notin P$ i $P \models\neg C \setminus l$ tvrđenje sledi. □

Jedan od dovoljnih (ali svakako ne i potrebnih) uslova za postojanje tačke jedinstvene implikacije i nivoa povratnog skoka je da se klauza sastoji isključivo od pretpostavljenih literala[4].

Lema 7.30. *Ako važi:*

(a) (distinct M), *(tj. $Invarijanta_{distinct}$ je zadovoljena)*

(b) $\overline{c} \subseteq$ (decisions M),

(c) $\overline{l} = (\text{lastAssertedLiteral } \overline{c}\ M)$,

tada važi (isUIP l c M).

Dokaz. Pošto je $\overline{c} \subseteq$ (decisions M), trivijalno važi i $M \models\neg c$.

Neka je $l' \in c \setminus l$. Pošto je $\overline{l} = (\text{lastAssertedLiteral } \overline{c}\ M)$, važi da je

$$(\text{level } \overline{l'}\ M) \leq (\text{level } \overline{l}\ M).$$

Pošto je $\overline{c} \subseteq$ (decisions M), važi $\overline{l'} \in$ (decisions M) i $\overline{l} \in$ (decisions M). Zato je

$$(\text{level } \overline{l'}\ M) \neq (\text{level } \overline{l}\ M),$$

pa tvrđenje sledi. □

[4] Ovo je slučaj kada se sa klauza povratnog skoka gradi kao kod primene pravila Backtrack

U slučaju kada klauza sastoji isključivo od pretpostavljenih literala, naravno, postoji odgovarajući nivo povratnog skoka.

Lema 7.31. *Ako važi:*

(a) (distinct M), *(tj. $Invarijanta_{distinct}$ je zadovoljena)*

(b) $\overline{c} \subseteq$ (decisions M),

(c) $\overline{l} =$ (lastAssertedLiteral $\overline{c}$ M),

tada postoji level takav da važi (isBackjumpLevel *level* l c M).

Dokaz. Na osnovu leme 7.30 važi

$$(\mathsf{isUIP}\ l\ c\ M).$$

Pošto je $c \neq [\,]$ i $\overline{c} \subseteq$ (decisions M), klauza $\overline{c}$ sadrži bar jedan literal pretpostavljen u M, te je (level $\overline{l}$ M) > 0. Tvrđenje onda sledi na osnovu leme 7.28. □

Upravo dokazane leme ukazuju na to da je u svakom trenutku u kome označena valuacija sadrži bar jedan pretpostavljeni literal, klauzu povratnog skoka moguće izgraditi upravo od pretpostavljenih literala.

7.4.4 Potpunost sistema sa povratnim skokovima

Teorema 7.3 i dalje važi. Međutim, njen dokaz mora biti modifikovan zato što iz činjenice da je (M, F) završno stanje ne sledi trivijalno da važi (decisions M) $= [\,]$. Naime, neophodno je eksplicitno pokazati da ako pravilo Backjump ne može da se primeni u konfliktnoj situaciji (kada $M \models\neg F$), onda važi da je (decisions M) $= [\,]$. Ovaj dokaz se zasniva upravo na činjenici da je klauzu povratnog skoka moguće izgraditi od pretpostavljenih literala.

Lema 7.32. *Ako važi:*

(a) (consistent M) *(tj. $Invarijanta_{consistent}$ je zadovoljena),*

(b) (distinct M), *(tj. $Invarijanta_{distinct}$ je zadovoljena)*

(c) (vars M) $\subseteq$ (vars F_0) $\cup$ $DecisionVars$ *(tj. $Invarijanta_{varsM}$ je zadovoljena),*

(d) $c \neq [\,]$ *i* $\overline{c} \subseteq$ (decisions M),

(e) $F \models c$,

tada pravilo Backjump[5] *može da se primeni, odnosno važi:*

$$\begin{aligned}\exists l\ level.\quad & P = (\mathsf{prefixToLevel}\ level\ M)\ \wedge\ 0 \leq level < (\mathsf{currentLevel}\ M)\ \wedge \\ & (\mathsf{isUnit}\ c\ l\ P)\ \wedge \\ & F \models c\ \wedge\ (\mathsf{var}\ l) \in (\mathsf{vars}\ F_0) \cup DecisionVars\end{aligned}$$

Dokaz. Neka je $\overline{l} = (\mathsf{lastDecision}\ M) = (\mathsf{lastAssertedLiteral}\ \overline{c}\ M)$. Na osnovu leme 7.31 sledi da postoji nivo $level$ takav da važi $(\mathsf{isBackjumpLevel}\ level\ l\ c\ M)$. Tada je

$$0 \leq level < (\mathsf{level}\ \overline{l}\ M) \leq (\mathsf{currentLevel}\ M).$$

Neka je

$$P = (\mathsf{prefixToLevel}\ level\ M).$$

Na osnovu leme 7.29 važi

$$(\mathsf{isUnit}\ c\ l\ P).$$

Pošto je $\overline{c} \subseteq (\mathsf{decisions}\ M)$, važi da je $(\mathsf{vars}\ c) \subseteq (\mathsf{vars}\ M) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ i pošto je $l \in c$, važi

$$(\mathsf{var}\ l) \in (\mathsf{vars}\ F_0) \cup DecisionVars,$$

čime je tvrđenje dokazano. □

Lema 7.33. *Ako važi:*

(a) $(\mathsf{consistent}\ M)$ *(tj.* $Invarijanta_{consistent}$ *je zadovoljena),*

(b) $(\mathsf{distinct}\ M)$ *(tj.* $Invarijanta_{distinct}$ *je zadovoljena),*

(c) $\forall l.\ l \in M \implies F\ @\ (\mathsf{decisionsTo}\ l\ M) \vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M*),*

(d) $(\mathsf{vars}\ M) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsM}$ *je zadovoljena),*

(e) $M \vDash_{\neg} F$*,*

(f) $(\mathsf{decisions}\ M) \neq [\,]$*,*

tada pravilo Backjump *može da se primeni.*

Dokaz. Neka je $c = \overline{(\mathsf{decisions}\ M)}$. Na osnovu leme 7.5, važi $\neg(\mathsf{sat}\ F\ @\ (\mathsf{decisions}\ M))$, odnosno $\neg(\mathsf{sat}\ F\ @\ \overline{c})$. Zbog toga važi $F \vDash c$. Pošto je i $c \neq [\,]$ i $\overline{c} \subseteq (\mathsf{decisions}\ M)$, tvrđenje sledi na osnovu leme 7.32. □

Na osnovu ovoga, moguće je dokazati sledeću teorema o potpunosti.

Teorema 7.3. *U završnom stanju* (M, F) *u kome su invarijante zadovoljene važi ili*

(a) $M \vDash_{\neg} F$ *i* $(\mathsf{decisions}\ M) = [\,]$ *ili*

(b) $M \nvDash_{\neg} F$ *i ne postoji literal takav da je* $(\mathsf{var}\ l) \in DecisionVars$*,* $l \notin M$ *i* $\overline{l} \notin M$*.*

Teorema 7.4 važi uz neizmenjeni dokaz.

7.4.5 Korektnost sistema sa povratnim skokovima

Teorema 7.5 važi uz neizmenjen dokaz.

7.5 Učenje (Learn, Forget)

Postupak učenja se zasniva na primeni sledećih pravila.

Learn:
$$\frac{F \models c, \quad (\text{vars } c) \subseteq Vbl}{F := F \ @ \ c}$$

Forget:
$$\frac{F \setminus c \models c}{F := F \setminus c}$$

Pravilo Learn bira proizvoljnu klauzu c za koju važi da je semantička posledica formule F i definisana je nad promenljivama nekog unapred fiksiranog skupa Vbl i dodaje je u formulu F. Strategija pronalaženja klauze c nije specifikovana ovom varijantom pravila Learn. U poglavlju 7.7 će biti definisan konkretniji opis sistema, u kome će biti precizirana strategija i način primenjivanja pravila Learn.

Naredna definicija formalno uvodi ova pravila.

Definicija 7.25.
$(\text{learn } (M_1, F_1)\ (M_2, F_2)) \iff$

$$\exists\ c\ .\quad F_1 \models c\ \wedge\ (\text{vars } c) \subseteq (\text{vars } F_0) \cup DecisionVars\ \wedge\ F_2 = F_1\ @\ c\ \wedge\ M_2 = M_1$$

$(\text{forget } (M_1, F_1)\ (M_2, F_2)) \iff$

$$\exists\ c\ .\quad F_1 \setminus c \models c\ \wedge\ F_2 = F_1 \setminus c\ \wedge\ M_2 = M_1$$

Sistem sa učenjem koristi pravila Decide, Backjump, UnitPropagate, Learn i Forget i može se modelovati preko naredne definicije relacije $\rightarrow$.

Definicija 7.14$^{(iii)}$.

$$\begin{aligned}(M_1, F_1) \rightarrow (M_2, F_2) \quad \iff \quad & (\text{decide } (M_1, F_1)\ (M_2, F_2)) \ \vee \\ & (\text{backjump } (M_1, F_1)\ (M_2, F_2)) \ \vee \\ & (\text{unitPropagate } (M_1, F_1)\ (M_2, F_2)) \ \vee \\ & (\text{learn } (M_1, F_1)\ (M_2, F_2)) \ \vee \\ & (\text{forget } (M_1, F_1)\ (M_2, F_2))\end{aligned}$$

U nastavku teksta, svako pominjanje relacije $\rightarrow$ će se odnositi na upravo uvedenu definiciju, sve dok se naglasi drugačije.

7.5.1 Saglasnost sistema sa učenjem

Teorema saglasnosti 7.1 i dalje ostaje na snazi. Opet je neophodno proširiti dokaz leme 7.2.

Pošto pravila Learn i Forget ne menjaju označenu valuaciju M, sve invarijante koje se odnose samo na M su trivijalno očuvane.

Pokažimo da pravila Learn i Forget održavaju invarijante $Invarijanta_{equiv}$, $Invarijanta_{varsF}$ i $Invarijanta_{impliedLiterals}$.

$Invarijanta_{equiv}$

Learn

Lema 7.34. *Ako važi:*

(a) $F \equiv F_0$ *(tj.* $Invarijanta_{equiv}$ *je zadovoljena za formulu* F*),*

(b) $F \models c$,

(c) $F' = F \mathbin{@} c$,

tada važi $F' \equiv F_0$ *(tj.* $Invarijanta_{equiv}$ *je zadovoljena za formulu* F'*).*

Dokaz. Sledi neposredno na osnovu stava 6.9. □

Forget

Lema 7.35. *Ako važi:*

(a) $F \equiv F_0$ *(tj.* $Invarijanta_{equiv}$ *je zadovoljena za formulu* F*),*

(b) $F \setminus c \models c$,

(c) $F' = F \setminus c$,

tada važi $F' \equiv F_0$ *(tj.* $Invarijanta_{equiv}$ *je zadovoljena za formulu* F'*).*

Dokaz. Sledi neposredno na osnovu stava 6.9. □

$Invarijanta_{varsF}$

Learn

Lema 7.36. *Ako važi:*

(a) $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsF}$ *je zadovoljena za* F*),*

(b) $(\mathsf{vars}\ c) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$,

(c) $F' = F \mathbin{@} c$,

tada važi $(\mathsf{vars}\ F') \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsF}$ *je zadovoljena za* F'*).*

Dokaz. Kako je $(\mathsf{vars}\ F') = (\mathsf{vars}\ F) \cup (\mathsf{vars}\ c)$, tvrđenje neposredno sledi na osnovu pretpostavki. □

Forget

Lema 7.37. *Ako važi:*

(a) $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsF}$ *je zadovoljena za* F*),*

(b) $F' = F \setminus c$,

tada važi (vars F') $\subseteq$ (vars F_0) $\cup$ $DecisionVars$ *(tj.* $Invarijanta_{varsF}$ *je zadovoljena za* F'*).*

Dokaz. Kako je (vars F') $\subseteq$ (vars F), tvrđenje neposredno sledi na osnovu pretpostavki. □

$Invarijanta_{impliedLiterals}$

Learn

Lema 7.38. *Ako važi:*

(a) $\forall l.\ l \in M \implies F$ @ (decisionsTo l M) $\vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* F*),*

(b) $F' = F$ @ c*,*

tada važi $\forall l.\ l \in M \implies F'$ @ (decisionsTo l M) $\vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* F'*),*

Dokaz. Neka je l literal iz M. Na osnovu pretpostavke važi F @ (decisionsTo l M) $\vDash l$. Pošto je $F \subseteq F'$, vaǐ i F @ (decisionsTo l M) $\subseteq F'$ @ (decisionsTo l M). Tada na osnovu stava 6.8 sledi F' @ (decisionsTo l M) $\vDash l$. □

Forget

Lema 7.39. *Ako važi:*

(a) $\forall l.\ l \in M \implies F$ @ (decisionsTo l M) $\vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* F*),*

(b) $F \setminus c \vDash c$*,*

(c) $F' = F \setminus c$*,*

tada važi $\forall l.\ l \in M \implies F'$ @ (decisionsTo l M) $\vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* F'*),*

Dokaz. Neka je l literal iz M. Pošto $F \vDash c$, na osnovu stava 6.8 važi $F \setminus c \vDash F$. Zato osnovu monotonosti važi i F' @ (decisionsTo l M) $\vDash F$ @ (decisionsTo l M). Pošto, na osnovu pretpostavke, važi F @ (decisionsTo l M) $\vDash l$, na osnovu tranzitivnosti sledi i F' @ (decisionsTo l M) $\vDash l$. □

Leme 7.2 i dalje važi i direktna je posledica prethodno dokazanih lema. Samim tim, važi i lema 7.3.

Pošto leme 7.8 i 7.9 i dalje važe, teorema 7.1 na osnovu ovoga neposredno sledi.

7.5.2 Zaustavljanje sistema sa učenjem

Sistem sa učenjem koji koristi pravila data u definiciji 7.25 ne mora da se zaustavlja. Naime, pravilo Learn i Forget je moguće naizmenično primenjivati za istu klauzu c. Takođe, pravilo Learn je moguće primenjivati nabrajajući sve klauze koje su semantičke posledice formule F. U poglavlju 7.7 će biti definisana strategija učenja koja zahteva da se pravilo Learn uvek primenjuje uz pravilo Backjump i za nju će biti pokazano svojstvo zaustavljanja.

7.5.3 Potpunost sistema sa učenjem

U slučaju da se sistem zaustavi, pošto su sve invarijante zadovoljene, teoreme 7.3 i 7.4 i dalje važe.

7.5.4 Korektnost sistema sa učenjem

Teorema 7.5 važi uz neizmenjen dokaz.

7.6 Otpočinjanje iznova (Restart)

Otpočinjanje ispočetka se zasniva na upotrebi pravila Restart.

Restart:

$$\frac{\top}{M := (\mathsf{prefixToLevel}\ 0\ M)}$$

Naredna definicija i formalno uvodi ovo pravilo.

Definicija 7.26.
$(\mathsf{restart}\ (M_1, F_1)\ (M_2, F_2)) \iff M_2 = (\mathsf{prefixToLevel}\ 0\ M_1)\ \wedge\ F_2 = F_1$

Ovo pravilo ima smisla primenjivati isključivo kada se koristi učenje i kada se selekcija literala u pravilu Decide vrši na način koji može da iskoristi rezultate prethodnog rada koji pravilom Restart biva poništen.

7.6.1 Saglasnost sistema sa otpočinjanjem iznova

Trivijalno je pokazati da ovo pravilo održava sve invarijante. Za invarijantu $Invarijanta_{impliedLiterals}$ to važi na osnovu leme 7.25.

7.6.2 Zaustavljanje sistema sa otpočinjanjem iznova

Kao i kod sistema sa učenjem, zaustavljanje sistema koji koristi otpočinjanje iznova ne mora da važi. Ipak, uz određene dodatne uslove koji nameću restrikcije na pravilo Restart, zaustavljanje može da se obezbedi.

7.6.3 Potpunost sistema sa otpočinjanjem iznova

U slučaju da se sistem zaustavi, pošto su sve invarijante zadovoljene, teorema 7.4 i dalje važi.

7.6.4 Korektnost sistema sa otpočinjanjem iznova

Teorema 7.5 važi uz neizmenjen dokaz.

7.7 Analiza konflikata (Conflict, Explain)

U ovom poglavlju ćemo formalizovati i dokazati korektnost DPLL sistema sa analizom konflikata datog u [KG07]. Najpre će biti navedena pravila ovog sistema na način koji je suštinski isti sa opisom datim u [KG07].

U odnosu na ranije opisane sisteme u kojima je stanje bilo predstavljeno samo tekućom (parcijalnom) označenom valuacijom i tekućim skupom klauza, u ovom sistemu se stanje proširuje sa:

1. Bulovskom promenljivom $conflict$ koja označava da li je u tekuća valuacija u koliziji sa tekućim skupom klauza, tj. da li je u toku faza analize konflikta,

2. klauzom C koju ćemo nazivati tekućom klauzom analize konflikta.

Definicija 7.12'. *Stanje sistema sa analizom konflikata je uređena četvorka* $(M, F, C, conflict)$*, gde je* M *označena valuacija,* F *formula,* C *klauza i* $conflict$ *Bulovska promenljiva.*

Pravila sistema su:

Decide:
$$\frac{l \in L, \quad l, \overline{l} \notin M}{M := M \,|\, l}$$

UnitPropagate:
$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F, \quad \overline{l}_1, \ldots, \overline{l}_k \in M, \quad l, \overline{l} \notin M}{M := M \; l}$$

Conflict:
$$\frac{conflict = \bot, \quad c \in F, \quad M \vDash \neg c}{conflict := \top, \quad C := c}$$

Explain:
$$\frac{conflict = \top, \quad \overline{l} \in C, \quad l \vee \overline{l}_1 \vee \ldots \vee \overline{l}_k \in F, \quad l_1, \ldots, l_k \prec^M l}{C := (C \setminus \overline{l}) \vee (\overline{l}_1 \vee \ldots \vee \overline{l}_k)}$$

Learn:
$$\frac{conflict = \top, \quad C \notin F}{F := F \mathbin{@} C}$$

Backjump:
$$\frac{conflict = \top, \quad C = l \vee l_1 \vee l_2 \vee \ldots \vee l_k, \quad C \in F, \quad (\mathsf{level}\ \overline{l}_i) \leq level < (\mathsf{level}\ \overline{l})}{conflict := \bot, \quad M := M^{[level]} \; l}$$

Kako bismo mogli formalno da uvedemo pravila ovoga sistema, uvodimo sledeću definiciju.

Definicija 7.27. *Klauza* c *je* uzrok propagacije *literala* l *u valuaciji* v*, što označavamo sa* $(\mathsf{isReason}\ c\ l\ v)$ *akko* $l \in c$*,* $v \vDash l$*,* $v \vDash \neg (c \setminus l)$ *i za svaki literal* $l' \in (c \setminus l)$*, literal* $\overline{l'}$ *prethodi literalu* l *u* v*, tj.* $\overline{l'} \prec^M l$*.*

Kada smo uveli potrebne definicije, možemo i formalno da uvedemo sva pravila ovoga sistema:

Definicija 7.28.
$(\mathsf{decide}\ (M_1, F_1, C_1, conflict_1)\ (M_2, F_2, C_2, conflict_2)) \iff$

$$\exists l. \quad (\mathsf{var}\ l) \in DecisionVars \ \wedge\ l \notin M_1 \ \wedge\ \overline{l} \notin M_1 \ \wedge$$
$$M_2 = M_1 \mathbin{@} l^{\top} \ \wedge\ F_2 = F_1 \ \wedge\ C_2 = C_1 \ \wedge\ conflict_2 = conflict_1$$

$$(\mathsf{unitPropagate}\ (M_1, F_1, C_1, conflict_1)\ (M_2, F_2, C_2, conflict_2)) \iff$$
$$\exists c\ l. \quad F_1 \vDash c\ \wedge\ (\mathsf{var}\ l) \in (\mathsf{vars}\ F_0) \cup DecisionVars\ \wedge\ (\mathsf{isUnit}\ c\ l\ M_1)\ \wedge$$
$$M_2 = M_1 @ l^{\perp}\ \wedge\ F_2 = F_1\ \wedge\ C_2 = C_1\ \wedge\ conflict_2 = conflict_1$$

$$(\mathsf{conflict}\ (M_1, F_1, C_1, conflict_1)\ (M_2, F_2, C_2, conflict_2)) \iff$$
$$\exists c. \quad conflict_1 = \perp\ \wedge\ F_1 \vDash c\ \wedge\ M_1 \vDash \neg c \wedge$$
$$M_2 = M_1\ \wedge\ F_2 = F_1\ \wedge\ C_2 = c\ \wedge\ conflict_2 = \top$$

$$(\mathsf{explain}\ (M_1, F_1, C_1, conflict_1)\ (M_2, F_2, C_2, conflict_2)) \iff$$
$$\exists\ l\ c. \quad conflict_1 = \top\ \wedge\ l \in C_1\ \wedge\ (\mathsf{isReason}\ c\ \bar{l}\ M_1)\ \wedge\ F_1 \vDash c$$
$$M_2 = M_1\ \wedge\ F_2 = F_1\ \wedge\ C_2 = (\mathsf{resolve}\ C\ c\ l)\ \wedge\ conflict_2 = \top$$

$$(\mathsf{backjump}\ (M_1, F_1, C_1, conflict_1)\ (M_2, F_2, C_2, conflict_2)) \iff$$
$$\exists c\ l. \quad conflict_1 = \top\ \wedge\ (\mathsf{isBackjumpLevel}\ level\ l\ C_1\ M_1)\ \wedge$$
$$M_2 = (\mathsf{prefixToLevel}\ level\ M_1) @ l^{\perp}\ \wedge\ F_2 = F_1\ \wedge$$
$$C_2 = C_1\ \wedge\ conflict_2 = conflict_1$$

$$(\mathsf{learn}\ (M_1, F_1)\ (M_2, F_2)) \iff$$
$$conflict_1 = \top\ \wedge\ C_1 \notin F_1$$
$$M_2 = M_1\ \wedge\ F_2 = F_1 @ C_1\ \wedge\ C_2 = C_1\ \wedge\ conflict_2 = conflict_1$$

Definicija 7.14$^{(iv)}$.

$$\begin{aligned}(M_1, F_1) \to (M_2, F_2) \iff\ & (\mathsf{decide}\ (M_1, F_1)\ (M_2, F_2))\ \vee \\ & (\mathsf{unitPropagate}\ (M_1, F_1)\ (M_2, F_2))\ \vee \\ & (\mathsf{conflict}\ (M_1, F_1)\ (M_2, F_2))\ \vee \\ & (\mathsf{explain}\ (M_1, F_1)\ (M_2, F_2))\ \vee \\ & (\mathsf{learn}\ (M_1, F_1)\ (M_2, F_2))\ \vee \\ & (\mathsf{backjump}\ (M_1, F_1)\ (M_2, F_2))\end{aligned}$$

U nastavku teksta, svako pominjanje relacije $\to$ će se odnositi na upravo uvedenu definiciju, sve dok se naglasi drugačije.

Primetimo da su u pravilima UnitPropagate, Conflict, Explain i Backjump uslovi pripadnosti klauze formuli (tj. uslovi obilka $c \in F$) zamenjeni slabijim uslovima obilka $F \vDash c$. Iako većina implementacija podrazumeva da su navedne klauze elementi formule F, u nastavku će biti pokazano da su i ovi slabiji uslovi dovoljni za pokazivanje korektnosti algoritma. Ovakva izmena, iako ne previše bitna za SAT rešavače predstavlja odličnu sponu prema SMT rešavačima o kojima će biti reči u glavi 15. Naglasimo još da je, radi obezbeđivanja zaustavljanja, u pravilu UnitPropagate bilo potrebno dodatno zahtevati da važi da je literal koji se dodaje u valuaciju definisan za neku promenljivu iz skupa $(\mathsf{vars}\ F_0) \cup DecisionVars$.

Za sistem za analizu konflikata značajne su sledeće vrste stanja:

1. *Prihvatajuća stanja* - stanja u kojima važi $conflict = \bot$, $M \not\models\neg F$ i ne postoji literal takav da je $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$.

2. *Odbacujuća stanja* - stanja u kojima važi $conflict = \top$ i $C = [\,]$.

7.7.1 Saglasnost sistema sa analizom konflikata

Teorema o saglasnosti sistema sa analizom konfikata je izmenjena tako da u obzir uzima novu definiciju prihvatajućih i odbacujućih stanja.

Teorema 7.1'. *Ako* $([\,], F_0, [\,], \bot) \rightarrow^* (M, F, C, conflict)$, *tada:*

1. *Ako važi*

   a) $DecisionVars \supseteq (\mathsf{vars}\ F_0)$,

   b) $M \not\models\neg F$,

   c) *pravilo* Decide *nije primenljivo (tj. ne postoji literal* l *takav da je* $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ *i* $\overline{l} \notin M$*),*

   tada je formula F_0 *zadovoljiva i* M *joj je model, (tj. važi* $(\mathsf{sat}\ F_0)$ *i* $(\mathsf{model}\ M\ F_0)$*).*

2. *Ako važi*

   a) $conflict = \top$,

   b) $C = [\,]$,

   tada je formula F_0 *nezadovoljiva (tj. važi* $\neg(\mathsf{sat}\ F_0)$*.*

Ova teorema će ponovo biti navedena na kraju ovog poglavlja i tada će biti dat njen formalni dokaz. Pre toga, pokažimo da tokom analize konflikata važe i sledeće dve invarijante:

$$Invarijanta_{Cfalse}: \quad conflict \implies M \models\neg C$$
$$Invarijanta_{Centailed}: \quad conflict \implies F \models C$$

Pošto je u početnom stanju $conflict = \bot$, važi sledeća lema koja odgovara lemi 7.1.

Lema 7.1'. *U stanju* $([\,], F_0, [\,], \bot)$ *su zadovoljene sve invarijante.*

Cilj nam je da dokažemo sledeću lemu analognu lemi 7.2 koja obezbeđuje da primena pravila čuva invarijante.

Lema 7.2'. *Ukoliko* $(M, F, C, conflict) \rightarrow (M', F', C', conflict')$ *i invarijante su zadovoljene u stanju* $(M, F, C, conflict)$*, tada su invarijante zadovoljene i u stanju* $(M', F', C', conflict')$*.*

Na osnovu prethodne dve leme, moguće je dokazati lemu analognu lemi 7.3 koja pokazuje da su invarijante zadovoljene tokom celokupnog rada sistema.

Lema 7.3'. *Ukoliko je* $([\,], F_0, [\,], \bot) \rightarrow^* (M, F, C, conflict)$*, tada su invarijante zadovoljene u stanju* $(M, F, C, conflict)$*.*

Dokaz. Analogno dokazu leme 7.3. □

Dokažimo sada lemu 7.2', tj. pokažimo da sva pravila sistema sa analizom konflikata čuvaju invarijante potrebne za dokaz saglasnosti sistema.

$\text{Invarijanta}_{\text{equiv}}$

Sva pravila osim Learn ne menjaju F, pa trivijalno održavaju ovu invarijantu.

Learn

Lema 7.40. *Ako važi:*

(a) $F \equiv F_0$ *(tj.* $Invarijanta_{equiv}$ *je zadovoljena za formulu* F*),*

(b) $conflict \implies F \models C$ *(tj.* $Invarijanta_{Centailed}$ *je zadovoljena),*

(c) $conflict = \top$*,*

(d) $F' = F \mathbin{@} C$*,*

tada važi $F' \equiv F_0$ *(tj.* $Invarijanta_{equiv}$ *je zadovoljena za formulu* F'*).*

Dokaz. Pošto, na osnovu $conflict = \top$ i $conflict \implies F \models C$, važi i $F \models C$, tvrđenje sledi na osnovu leme 7.34. □

$\text{Invarijanta}_{\text{varsF}}$

Sva pravila osim Learn ne menjaju F pa trivijalno održavaju ovu invarijantu.

Learn

Lema 7.41. *Ako važi:*

(a) $(\text{vars } F) \subseteq (\text{vars } F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsF}$ *je zadovoljena za* F*),*

(b) $(\text{vars } M) \subseteq (\text{vars } F_0) \cup DecisionVars$*, (tj.* $Invarijanta_{varsM}$ *je zadovoljena za* M*),*

(c) $conflict \implies M \models\neg C$ *(tj.* $Invarijanta_{Cfalse}$ *je zadovoljena),*

(d) $conflict = \top$*,*

(e) $F' = F \mathbin{@} C$*,*

tada važi $(\text{vars } F') \subseteq (\text{vars } F_0) \cup DecisionVars$ *(tj.* $Invarijanta_{varsF}$ *je zadovoljena za* F'*).*

Dokaz. Pošto, na osnovu $conflict = \top$ i $conflict \implies M \models\neg C$, važi $M \models\neg C$, važi da je $(\text{vars } C) \subseteq (\text{vars } M) \subseteq (\text{vars } F_0) \cup DecisionVars$. Tada tvrđenje sledi na osnovu leme 7.36. □

$\text{Invarijanta}_{\text{impliedLiterals}}$

Pravila Decide, UnitPropagate i Learn čuvaju ovu invarijantu na na osnovu lema 7.6, 7.23 i 7.38. Pravila Conflict i Explain ne menjaju ni M ni F pa trivijalno održavaju ovu invarijantu.

Backjump

Sada možemo dokazati da i konkretizovana verzija pravila Backjump održava invarijantu $Invarijanta_{impliedLiterals}$.

Lema 7.42. *Ako važi:*

(a) (consistent M) *(tj. Invarijanta$_{consistent}$ je zadovoljena),*

(b) (distinct M) *(tj. Invarijanta$_{distinct}$ je zadovoljena),*

(c) $\forall l.\ l \in M \implies F \mathbin{@}$ (decisionsTo $l\ M$) $\vDash l$ *(tj. Invarijanta$_{impliedLiterals}$ je zadovoljena za M),*

(d) $conflict \implies F \vDash C$ *(tj. Invarijanta$_{Centailed}$ je zadovoljena),*

(e) $conflict = \top$,

(f) *level je nivo povratnog skoka, tj.* (isBackjumpLevel $level\ l\ C\ M$),

(g) $M' =$ (prefixToLevel $level\ M$) $@\, l^{\perp}$,

tada važi $\forall l'.\ l' \in M' \implies F \mathbin{@}$ (decisionsTo $l'\ M'$) $\vDash l'$ *(tj. Invarijanta$_{impliedLiterals}$ je zadovoljena za M'),*

Dokaz. Na osnovu leme 7.29 važi (isUnit $C\ l$ (prefixToLevel $level\ M$)). Na osnovu $conflict = \top$ i $conflict \implies F \vDash C$ važi i $F \vDash C$. Time su ispunjeni svi uslovi za primenu leme 7.26, na osnovu čega tvrđenje sledi. □

Invarijanta$_{Cfalse}$

Nakon primene pravila Backjump, promenljiva $conflict$ se postavlja na $\perp$, pa je ova invarijanta trivijalno ispunjena. Pravilo Learn ne menja ni M ni C tako da trivijalno održava ovu invarijantu. Pravila Decide i UnitPropagate samo proširuju tekuću valuaciju M, tako da održavaju invarijantu na osnovu monotonosti opisane u stavu 6.3 (tj. svojstva $v \vDash \neg c \implies (v \mathbin{@} v') \vDash \neg c$). Pošto pravilo Conflict postavlja C na klauzu c za koju važi $M \vDash \neg c$, invarijanta trivijalno važi.

Explain

Lema 7.43. *Ukoliko važi:*

(a) $conflict \implies M \vDash \neg C$ *(i.e., Invarijanta$_{Cfalse}$ je zadovoljena za C),*

(b) *l je literal takav da je $l \in C$ i c je klauza koja je razlog za propagaciju literal $\overline{l}$ (tj. važi* (isReason $c\ \overline{l}\ M$)),

(c) $C' =$ (resolvent $C\ c\ l$),

(d) $conflict = \top$,

tada $conflict \implies M \vDash \neg C'$ *(tj. Invarijanta$_{Cfalse}$ je zadovoljena za C').*

Dokaz. Pošto $M \vDash \neg C$ svi literali klauze C, uključujući i l su netačni u M. Pošto je (isReason $c\ \overline{l}\ M$), svi literali klauze c osim $\overline{l}$ (koji je tačan), su netačni u M, odnosno važi $M \vDash \neg (c \backslash \overline{l})$. Klauza C' tj. (resolvent $C\ c\ l$) $= (C \backslash l) \mathbin{@} (c \backslash \overline{l})$, se sastoji od svih literala iz C osim l i svih literala iz c osim $\overline{l}$ i zato je netačna u M, tj. važi $M \vDash \neg C'$. □

$Invarijanta_{Centailed}$

Nakon primene pravila Backjump, promenljiva $conflict$ se postavlja na $\bot$, pa je ova invarijanta trivijalno ispunjena. Pravila Decide i UnitPropagate ne menjaju ni C ni F pa trivijalno održavaju ovu invarijantu. Pošto pravilo Conflict postavlja C na klauzu c za koju važi $F \models c$, invarijanta trivijalno važi. Pošto pravilo Learn proširuje formulu F, invarijanta važi na osnovu monotonosti, odnosno osobine $F \models C \implies (F \mathbin{@} c) \models C$ iz stava 6.8.

Explain

$Invarijanta_{Centailed}$ ostaje zadovoljena nakon primene pravila Explain kao rezultat naredne leme.

Lema 7.44. *Ukoliko važi:*

(a) $conflict \implies F \models C$ *(tj.* $Invarijanta_{Centailed}$ *je zadovoljena za* C*),*

(b) $F \models c$,

(c) $conflict = \top$,

(d) $C' = (\mathsf{resolvent}\ C\ c\ l)$, *za neki literal* l,

tada važi $conflict \implies F \models C'$ *(tj.* $Invarijanta_{Centailed}$ *je zadovoljena* C'*).*

Dokaz. Neka je M' model formule F, tj. $M' \models F$. Kako važi $conflict = \top$, sledi da $F \models C$ i na osnovu toga $M' \models C$. Pošto važi $F \models c$, sledi da $M' \models c$. Kako je M' model, važi $(\mathsf{consistent}\ M')$. Tada, na osnovu stava 6.10, važi $M' \models (\mathsf{resolvent}\ C\ c\ \bar{l})$. □

U poglavlju 7.7.2 o zaustavljanju će biti pokazano da i invarijanta $Invarijanta_{varsM}$ ostaje zadovoljena nakon primene pravila pravila. Pošto sva pravila održavaju sve do sada navedne invarijante, važi lema 7.2' i na osnovu toga i lema 7.3'.

Iako i dalje važi lema 7.9 ovde ćemo alternativni dokaz iskoristiti saglasnosti za nezadovoljive formule koji u obzir uzima novu definiciju odbacujućih stanja i koji ne koristi invarijantu $Invarijanta_{impliedLiterals}$.

Lema 7.45. *Ako važi:*

(a) $F \equiv F_0$, *(tj.* $Invarijanta_{equiv}$ *je zadovoljena)*

(b) $conflict \implies F \models C$ *(tj.* $Invarijanta_{Centailed}$ *je zadovoljena),*

(c) $conflict = \top$

(d) $C = [\,]$

tada je F_0 *nezadovoljiva (tj. važi* $\neg(\mathsf{sat}\ F_0)$*).*

Dokaz. Na osnovu stava 6.8 sledi $\neg(\mathsf{sat}\ F)$, pa pošto je $F \equiv F_0$ važi $\neg(\mathsf{sat}\ F_0)$. □

Sada možemo dokazati teoremu o saglasnosti:

Teorema 7.1'. *Ako* $([\,], F_0, [\,], \bot) \rightarrow^* (M, F, C, conflict)$, *tada:*

1. *Ako važi*

   a) $DecisionVars \supseteq (\mathsf{vars}\ F_0)$,

   b) $M \not\models_{\neg} F$,

   c) *pravilo* Decide *nije primenljivo (tj. ne postoji literal* l *takav da je* $(\mathsf{var}\ l) \in DecisionVars$, $l \notin M$ *i* $\overline{l} \notin M$*),*

   tada je formula F_0 *zadovoljiva i* M *joj je model, (tj. važi* $(\mathsf{sat}\ F_0)$ *i* $(\mathsf{model}\ M\ F_0)$*).*

2. *Ako važi*

   a) $conflict = \top$,

   b) $C = [\,]$,

   tada je formula F_0 *nezadovoljiva (tj. važi* $\neg(\mathsf{sat}\ F_0)$*.*

Dokaz. Na osnovu leme 7.3' sve invarijante su zadovoljene u stanju $(M, F, C, conflict)$.

1. Svi uslovi za primenu leme 7.8 su ispunjeni, te važi $(\mathsf{sat}\ F_0)$ i $(\mathsf{model}\ M\ F_0)$.
2. Svi uslovi za primenu leme 7.45 su ispunjeni, te važi $\neg(\mathsf{sat}\ F_0)$.

□

7.7.2 Zaustavljanje sistema sa analizom konflikata

Teorema o zaustavljanju sistema sa analizom konflikata će biti dokazana tako što će biti izgradnjeno dobro zasnovano uređenje $\succ$ koje je u skladu sa relacijom $\rightarrow$, tj. važi da iz $(M_1, F_1, C_1, conflict_1) \rightarrow (M_2, F_2, C_2, conflict_2)$ sledi $(M_1, F_1, C_1, conflict_1) \succ (M_2, F_2, C_2, conflict_2)$. Ovo uređenje će biti izgrađeno kao leksikografska kombinacija četiri jednostavnija uređenja koja odgovaraju svakoj od komponenata stanja.

Pravila koja menjaju M (Decide, UnitPropagate i Backjump) ga menjaju na način koji obezbeđuje napredak procedure u odnosu na uređenje $\succ_{trail}$ definisano u poglavlju 7.2.2. Podsetimo, da bi se obezbedilo da je uređenje $\succ_{trail}$ dobro zasnovano uređenje, bilo je dovoljno dokazati da važi $Invarijanta_{varsM}$.

Sada ćemo pokazati da sva pravila sistema sa analizom konflikata održavaju i invarijantu $Invarijanta_{varsM}$.

Invarijanta$_{\text{varsM}}$

Pravila Conflict, Explain i Learn ne menjaju M tako da je invarijanta trivijalno očuvana. Pravilo Decide održava invarijantu na osnovu leme 7.18.

UnitPropagate

Pošto se u novoj definiciji pravila UnitPropagate više ne zahteva da je $c \in F$, lema 7.24 ne može više da se koristi. Međutim, pošto se eksplicitno zahteva uslov $(\mathsf{vars}\ l) \in (\mathsf{vars}\ F_0) \cup DecisionVars$, invarijanta trivijalno važi.

Backjump

Lema 7.46. *Ako važi:*

(a) (vars M) $\subseteq$ (vars F_0) $\cup$ $DecisionVars$ *(tj.* $Invarijanta_{varsM}$ *je zadovoljena za* M*)*,

(b) $P =$ (prefixToLevel $level$ M),

(c) (isBackjumpLevel $level$ l C M),

(d) $conflict = \top$,

(e) $M' = P$ @ l,

tada je (vars M') $\subseteq$ (vars F_0) $\cup$ $DecisionVars$ *(tj.* $Invarijanta_{varsM}$ *je zadovoljena za* M*)*,

Dokaz. Pošto važi (isPrefix P M) i (vars M) $\subseteq$ (vars F_0) $\cup$ $DecisionVars$, važi da je (vars P) $\subseteq$ (vars F_0) $\cup$ $DecisionVars$. Pošto je

$$(\text{vars } M') = (\text{vars } P) \cup \{(\text{var } l)\},$$

dovoljno je dokazati da je (var l) $\in$ (vars F_0) $\cup$ $DecisionVars$. Pošto važi (isBackjumpLevel $level$ l C M), sledi da je $\overline{l} =$ (lastAssertedLiteral $\overline{C}$ M), pa je $\overline{l} \in M$. Tada tvrđenje sledi iz (vars M) $\subseteq$ (vars F_0) $\cup$ $DecisionVars$. □

Na osnovu ovoga sva pravila održavaju i invarijantu $Invarijanta_{varsM}$ i lema 7.2' važi.

Problem u analizi zaustavljanja procedure predstavljaju pravila Conflict, Explain i Learn koja ne menjaju M. Iako ova pravila ne smanjuju stanje rešavača u odnosu na uređenje $\succ_{trail}$, za svako od njih će biti definisano posebno dobro zasnovano uređenje i biće pokazano da ona smanjuju odgovarajuće komponente stanja u odnosu na tako definisana uređenja.

Započnimo sa trivijalnim uređenjem Bulovskih promenljivih. Reći ćemo da je

$$conflict_1 \succ_{confflag} conflict_2 \iff conflict_1 = \bot \ \wedge \ conflict_2 = \top.$$

Sada trivijalno važi naredna lema koju navodimo bez dokaza.

Lema 7.47.

$$(\text{conflict } (M_1, F_1, C_1, conflict_1)\ (M_2, F_2, C_2, conflict_2)) \implies conflict_1 \succ_{confflag} conflict_2$$

Lema 7.48. *Relacija* $\succ_{confflag}$ *je relacija dobro zasnovana.*

Dokaz. Trivijalno ($\top$ je minimalni element svakog skupa Bulovskih promenljivih). □

Sada je potrebno da definišemo uređenje na skupu klauza C, tako da to uređenje bude u skladu sa Explain pravilom, odnosno da garantuje da se komponenta C stanja smanjuje sa svakom primenom pravila Explain. Neformalno, nakon svake primene pravila Explain neki literal l klauze C koji je zbog invarijante $M \vDash \neg C$ netačan u M, se zamenjuje sa nekoliko drugih literala koji su opet netačni u M, ali za koje važe da njihovi suprotni literali prethode literalu $\overline{l}$ u označenoj valuaciji M. Redosled literala u označenoj valuaciji M dakle određuje određeni poredak klauza čiji su literali definisani u M.

Zato najpre pokažimo da poredak elemenata u listi koji je uveden kroz definiciju 5.3 predstavlja relaciju dobro zasnovanog uređenja.

Lema 7.49. *Uređenje $\prec^L$ je dobro zasnovano uređenje.*

Dokaz. Dokazaćemo princip minimalnog elementa, tj.

$$\forall\, Q.\ (\exists\, a \in Q) \implies (\exists\, a_{min} \in Q.\ (\forall\, a'.\ a' \prec^L a_{min} \implies a' \notin Q))$$

Neka je Q skup elemenata. Pretpostavimo da je $a \in Q$. Neka je Q_L lista svih elemenata liste L koji se nalaze u skupu Q i to u istom poretku u kom se javljaju u listi L.

Ukoliko je $Q_L = [\,]$, tada je trivijalno $a_{min} = a$, jer da bi važilo $a' \prec^L a_{min}$ moralo bi da važi da je $a_{min} = a \in Q$, što je nemoguće jer je $Q_L = [\,]$.

Dakle, mora da važi $Q_L \neq [\,]$. Tada je $a_{min} = (\mathsf{head}\ Q_L)$. Zaista, neka je a' element za koji važi $a' \prec^L a_{min}$. Tada je $a' \in L$. Ako bi važilo $a' \in Q$, tada bi važilo da je $a' \in Q_L$. Međutim, tada bi moralo da važi da je $a' \prec^{Q_L} a_{min}$, što je nemoguće sa obzirom da je a_{min} prvi element liste Q_L. □

Da bismo sproveli opisanu konstrukciju, potrebno je da uvedemo pojam multiskupa.

Definicija 7.29. Multiskup *je „skup" u kome je dozvoljeno ponavljanje elemenata. Formalnije, multiskup elemenata skupa X je preslikavanja $S : X \mapsto \mathbb{N}_0$. Multiskup je konačan ukoliko je $\{x \mid S(x) > 0\}$ konačan.* Unija multiskupova *se definiše kao $(S \cup T)(x) = S(x) + T(x)$.*

Sada ćemo uvesti definiciju multiskup proširenja relacije (multiset extension).

Definicija 7.30. *Ako je $<$ uređenje skupa X tada je njegovo* multiskup proširenje *$<_{mult}$ relacija definisana na skupu n-torki skupa X koje razmatramo kao multiskupove elemenata skupa x i obležavamo sa $\langle s_1, \ldots, s_n \rangle$. Relacija $<_{mult}$ je tranzitivno zatvorenje relacije $<_{mult_1}$, pri čemu važi:*

$$\begin{aligned} S_1 <_{mult_1} S_2 \iff \exists\ & S\ S'_1\ s_2. \\ & S_1 = S \cup S'_1 \ \wedge\ S_2 = S \cup \langle s_2 \rangle \ \wedge \\ & \forall\, s_1.\ s_1 \in S'_1 \implies s_1 < s_2 \end{aligned}$$

Dakle, uređenje klauza će biti multiskup proširenje relacije $\prec^M$ određene poretkom literala u valuaciji M. Zaista, literal na osnovu kojega se vrši rezolucija i koji se uklanja iz klauze C bi u prethodnoj definiciji imao ulogu literala s_2, dok bi literali koji se dodaju u klauzu C imali ulogu skupa S_1. Međutim,

problem može da nastane ukoliko klauza C sadrži nekoliko pojavljivanja literala koji se uklanja. Naša definicija funkcije resolve je takva da se jednim korakom rezolucije istovremeno uklanjaju sva pojavljivanja literala, tako da prethodna definicija ne bi mogla ovako direktno da se primeni, već bi bilo potrebno koristiti svojstva tranzitivnog zatvorenja. Kako bismo jednostavno razrešli ovu situaciju, naša definicija poretka klauza vrši uklanjanje duplikata pre nego što se samo poređenje izvrši. U realnim rešavačima se obično duplikati iz svih klauza efektivno uklone u fazi pretprocesiranja i ta invarijanta se održava tokom celog rada sistema. Naš formalni opis to ne zahteva i zaustavljanje se na ovaj način može pokazati čak i ako klauze sadrže ponovljene literale.

Definicija 7.31.

$$C_1 \succ^M_{confclause} C_2 \iff \langle(\mathsf{removeDups}\ \overline{C_2})\rangle \prec^M_{mult} \langle(\mathsf{removeDups}\ \overline{C_1})\rangle$$

Lema 7.50. *Ako je* $l \in C$ *i* $(\mathsf{isReason}\ c\ \overline{l}\ M)$ *tada je* $C \succ^M_{confclause} (\mathsf{resolve}\ C\ c\ l)$.

Dokaz. Neka je

$$\begin{aligned} S_1 &= \langle(\mathsf{removeDups}\ \overline{(\mathsf{resolve}\ C\ c\ l)})\rangle, \\ S_2 &= \langle(\mathsf{removeDups}\ \overline{C})\rangle. \end{aligned}$$

Pokažimo da je $S_1 \prec^M_{mult} S_2$. Neka je

$$\begin{aligned} S &= \langle(\mathsf{removeDups}\ \overline{C \setminus l})\rangle, \\ S'_1 &= \langle(\mathsf{removeDups}\ \overline{(c \setminus C) \setminus \overline{l}})\rangle, \\ s_2 &= \overline{l}. \end{aligned}$$

Pošto je $l \in C$, važi da je $S_2 = S \cup \langle s_2\rangle$.

Pokažimo da važi $S_1 = S \cup S'_1$. Na osnovu definicije funkcije resolve važi $S_1 = \langle(\mathsf{removeDups}\ \overline{(C \setminus l)@(c \setminus \overline{l})})\rangle$. Na osnovu osobina multiskupova i funkcije removeDups važi $S_1 = \langle(\mathsf{removeDups}\ \overline{C \setminus l})\rangle \cup \langle(\mathsf{removeDups}\ \overline{(c \setminus \overline{l}) \setminus (\overline{C \setminus l})})\rangle$. Dalje, važi da je $\overline{(c \setminus C) \setminus \overline{l}} = \overline{c \setminus \overline{l}} \setminus \overline{C \setminus l}$. Zaista, važi $l \notin (c \setminus \overline{l})$ i zato je $(c \setminus \overline{l}) \setminus C = (c \setminus \overline{l}) \setminus (C \setminus l)$ pa tvrđenje sledi.

Pokažimo još da $\forall\ s_1.\ s_1 \in S'_1 \implies s_1 \prec^M s_2$. Neka je $s_1 \in S'_1$. Tada je $\overline{s}_1 \in c$, $\overline{s}_1 \notin C$ i $s_1 \neq l$. Pošto još važi i $(\mathsf{isReason}\ c\ \overline{l}\ M)$ sledi da $s_1 \prec^M \overline{l}$, pa tvrđenje sledi pošto je $s_2 = \overline{l}$. □

Lema 7.51.

$$(\mathsf{explain}\ (M, F, C_1, conflict)\ (M, F, C_2, conflict)) \implies C_1 \succ^M_{confclause} C_2$$

Dokaz. Pošto važi $(\mathsf{explain}\ (M, F, C_1, conflict)\ (M, F, C_2, conflict))$ važi da postoji literal l i klauza c tako da važi $l \in C_1$, $(\mathsf{isReason}\ c\ \overline{l}\ M)$ i $C_2 = (\mathsf{resolve}\ C\ c\ l)$. Tvrđenje onda važi na osnovu leme 7.50. □

Lema 7.52. *Relacija* $\succ^M_{confclause}$ *je relacija dobro zasnovanog uređenja.*

Dokaz. Posmatrajmo funkciju F takvu da je $F(C) = \langle(\mathsf{removeDups}\ \overline{C})\rangle$. Važi da je $\succ^M_{confclause} = F^{-1}(\prec^M_{mult})$. Pošto je na osnovu leme 7.49 relacija $\prec^M$ dobro zasnovanog uređenje, a dobra uređenja ostaju dobra i kada se izvrši multiskup proširenje i nađe inverzna slika, tvrđenje sledi. □

Na kraju definišimo uređenje na skupu formula koje će da bude kompatibilno sa pravilom Learn. Pošto ovo pravilo formuli dodaje klauzu C uvodimo sledeću definiciju:

Definicija 7.32.

$$F_1 \succ^{C}_{formula} F_2 \iff C \notin F_1 \wedge C \in F_2$$

Ovo znači da je stanje postaje naprednije nakon što se nauči klauza C.

Lema 7.53. *Relacija $\succ^{C}_{formula}$ predstavlja relaciju dobro zasnovanog uređenja.*

Dokaz. Dokažimo princip najmanjeg elementa.

$$\forall\, Q.\ (\exists\, F \in Q) \implies (\exists\, F_{min} \in Q.\ (\forall\, F'.\ F_{min} \succ^{C}_{formula} F' \implies F' \notin Q))$$

Neka je Q skup formula i F formula tog skupa.

Ukoliko postoji formula iz Q koja sadrži klauzu C, tada se ona može uzeti za F_{min}. Zaista, ukoliko bi za neko F' važilo $F_{min} \succ^{C}_{formula} F'$, tada bi moralo da važi $C \notin F_{min}$ što je kontradikcija.

Ukoliko ne postoji formula iz Q koja sadži klauzu C, onda se za F_{min} može uzeti bilo koja formula iz Q. Sigurno je $F \in Q$ pa je $F_{min} = F$. Zaista, ukoliko bi važilo $F_{min} \succ^{C}_{formula} F'$ i važilo $F' \in Q$, tada bi važilo da je $C \in F'$, pa bi postojala formula iz Q koja sadrži C što je kontradikcija. □

Lema 7.54.

$$(\mathsf{learn}\ (M, F_1, C, conflict)\ (M, F_2, C, conflict)) \implies F_1 \succ^{C}_{formula} F_2$$

Dokaz. Pošto važi $C \notin F_1$ i $C \in F_2$, tvrđenje trivijalno sledi. □

Sada možemo uvesti konačno uređenje na stanjima. Ono predstavlja leksikografsku kombinaciju uređenja $\succ^{Vbl}_{trail}$, $\succ_{conflflag}$, $\succ^{M}_{confclause}$ i $\succ^{C}_{formula}$.

Definicija 7.33. $(M_1, F_1, C_1, conflict_1) \succ (M_2, F_2, C_2, conflict_2) \iff$

$$\begin{aligned}
&M_1 \succ^{Vbl}_{trail} M_2 \ \vee \\
&M_1 = M_2 \ \wedge\ conflict_1 \succ_{conflflag} conflict_2 \vee \\
&M_1 = M_2 \ \wedge\ conflict_1 = conflict_2 \ \wedge\ C_1 \succ^{M_1}_{confclause} C_2 \vee \\
&M_1 = M_2 \ \wedge\ conflict_1 = conflict_2 \ \wedge\ C_1 = C_2 \vee\ F_1 \succ^{C_1}_{formula} F_2
\end{aligned}$$

Lema 7.55. *Neka je $>$ uređenje definisano na skupu uređenih parova kao*

$$(x_1, y_1) > (x_2, y_2) \iff x_1 >_x x_2 \ \vee\ x_1 = x_2 \ \wedge\ y_1 >_y y_2,$$

pri čemu su $>_x$ i $>_y$ dobra uređenja. Tada je $i >$ dobro zasnovano uređenje.

Dokaz. Neka je $\succ_x^-$ uređenje za koje važi $(x_1, y_1) \succ_x^- (x_2, y_2)$ ako i samo ako je $x_1 >_x x_2$. Pošto je $>_x$ dobro zasnovano uređenje, na osnovu leme 7.20, dobro zasnovano je i $\succ_x^-$.

Slično, neka je $\succ_y^-$ uređenje za koje važi $(x_1, y_1) \succ_y^- (x_2, y_2)$ ako i samo ako je $y_1 >_y y_2$. Pošto je $>_y$ dobro zasnovano uređenje, na osnovu leme 7.20, dobro zasnovano je i $\succ_y^-$.

Dokažimo princip najmanjeg elementa za $>$.

$$\forall\, Q.\ (\exists\, a \in Q) \implies (\exists\, a_{min} \in Q.\ (\forall\, a'.\ a_{min} > a' \implies a' \notin Q))$$

Neka je Q skup uređenih parova i a jedan njegov element. Na osnovu principa najmanjeg elementa za relaciju $\succ_x^-$, postoji neki par (x_{min}, y) takav da važi $(x_{min}, y) \in Q$ i

$$\forall\, a'.\ (x_{min}, y) \succ_x^- a' \implies a' \notin Q.$$

Posmatrajmo skup $Q_{x_{min}} = \{(x, y) \mid x = x_{min}\}$. Važi da je $(x_{min}, y) \in Q_{x_{min}}$. Tada, na osnovu principa najmanjeg elementa za relaciju $\succ_y^-$ postoji element (x_{min}, y_{min}) takav da važi $(x_{min}, y_{min}) \in Q_{x_{min}}$ i

$$\forall\, a'.\ (x_{min}, y_{min}) \succ_y^- a' \implies a' \notin Q_{x_{min}}.$$

Tvrdimo da je (x_{min}, y_{min}) najmanji element skupa Q. Pretpostavimo da postoji $a' = (x', y') \in Q$ takav da $(x_{min}, y_{min}) > (x', y')$.

Ako je $x_{min} >_x x'$, tada je i $(x_{min}, y) \succ_x^- (x', y')$, pa sledi da $a' \notin Q$, što je kontradikcija.

Ako je $x_{min} = x' \wedge y_{min} > y'$, tada je $a' \in Q_{x_{min}}$ i $(x_{min}, y_{min}) \succ_y^- (x', y')$. Međutim, tada bi moralo da važi da $a' \notin Q_{x_{min}}$, što je kontradikcija.

Dakle za svako a' za koje je $(x_{min}, y_{min}) > a'$ mora da važi $a' \notin Q$. □

Lema 7.56. *Relacija $\succ$ je relacija dobro zasnovanog uređenja.*

Dokaz. Tvrđenje sledi na osnovu lema 7.17, 7.48, 7.52, 7.53 trostrukom primenom leme 7.55. □

Lema 7.57. *Ako su invarijante zadovoljene u stanju $(M_1, F_1, C_1, conflict_1)$ i važi*

$$(M_1, F_1, C_1, conflict_1) \rightarrow (M_2, F_2, C_2, conflict_2),$$

tada važi

$$(M_1, F_1, C_1, conflict_1) \succ (M_2, F_2, C_2, conflict_2).$$

Dokaz. Sledi na osnovu lema 7.12, 7.51, 7.47, 7.54, 7.2. □

Teorema o zaustavljanju na osnovu ovoga trivijalno sledi.

7.7.3 Potpunost sistema sa analizom konflikata

Cilj ovog poglavlja je da dokaže potpunost sistema sa analizom konflikata. Ključni korak u tom smeru je dokaz teoreme koja tvrdi da su i u sistemu sa analizom konflikata sva završna stanja ili prihvatajuća ili odbacujuća.

Jedan od najvažnijih uslova za mogućnost analize konflikata je postojanje klauza koje predstavljaju razloge propagacije literala i koji omogućavaju primenu pravila Explain. Literali se u tekuću označenu valuaciju M postavljaju

kao označeni literali isključivo kao rezultat primene pravila UnitPropagate ili Backjump. U oba slučaja se propagacija vrši na osnovu poznate klauze koja predstavlja razlog za propagaciju i koju je moguće na neki način pridružiti izvedenom literalu. U fazi analize konflikata, ove klauze se mogu koristiti kao klauze koje omogućavaju primenu pravila Explain. Invarijanta koju ćemo sada formulisati govori upravo o ovom svojstvu:

$$Invarijanta_{reasonClauses}: \quad \forall\, l.\ l \in M \ \wedge\ l \notin (\mathsf{decisions}\ M) \implies \exists\, c.\ (\mathsf{isReason}\ c\ l\ M) \ \wedge\ F \vDash c$$

Pošto je u početnom stanju $M = [\,]$, invarijanta tada trivijalno važi.

Pokažimo da sva pravila održavaju ovu invarijantu.

$Invarijanta_{reasonClauses}$

Pravila Conflict i Explain ne menjaju ni M ni F, pa trivijalno održavaju ovu invarijantu. Pravilo Decide ne dodaje izvedene literale u označenu valuaciju, pa i ono trivijalno održava ovu invarijantu. Pošto pravilo Learn proširuje formulu F, invarijanta važi na osnovu monotonosti, tj. osobine $F \vDash c \implies (F \mathbin{@} C) \vDash c$ iz stava 6.8.

Dokažimo i sledeću jednostavnu lemu, koja će obezbediti važenje invarijante nakon primene pravila UnitPropagate i Backjump

Lema 7.58.

$$(\mathsf{isUnit}\ c\ l\ M) \implies (\mathsf{isReason}\ c\ l\ (M \mathbin{@} l))$$

Dokaz. Na osnovu $(\mathsf{isUnit}\ c\ l\ M)$ sledi $M \vDash\neg (c \setminus l)$, $M \nvDash\neg l$ i $M \nvDash l$. Zato, na osnovu monotonosti sledi $(M \mathbin{@} l) \vDash\neg (c \setminus l)$. $(M \mathbin{@} l) \vDash l$ trivijalno važi. Ako je $l' \in (c \setminus l)$ tada je $\overline{l'} \in M$, pa time važi $\overline{l'} \prec^M l$. □

UnitPropagate

Lema 7.59. *Ako važi:*

(a) $\forall\, l.\ l \in M \ \wedge\ l \notin (\mathsf{decisions}\ M) \implies \exists\, c.\ (\mathsf{isReason}\ c\ l\ M) \ \wedge\ F \vDash c$ *(tj.* $Invarijanta_{reasonClauses}$ *je zadovoljena za* M*),*

(b) c *je klauza takva da* $F \implies c$ *i* $(\mathsf{isUnit}\ c\ l\ M)$*.*

(c) $M' = M \mathbin{@} l$*,*

tada važi $\forall\, l.\ l \in M' \ \wedge\ l \notin (\mathsf{decisions}\ M') \implies \exists\, c.\ (\mathsf{isReason}\ c\ l\ M') \ \wedge\ F \vDash c$ *(tj.* $Invarijanta_{reasonClauses}$ *je zadovoljena za* M'*),*

Dokaz. Za izvedene literale iz M, tvrđenje trivijalno sledi. Što se tiče literala l, tvrđenje sledi na osnovu leme 7.58. □

Backjump

Lema 7.60. *Ako važi:*

(a) $\forall\, l.\ l \in M \ \wedge\ l \notin (\mathsf{decisions}\ M) \implies \exists\, c.\ (\mathsf{isReason}\ c\ l\ M) \ \wedge\ F \vDash c$ *(tj.* $Invarijanta_{reasonClauses}$ *je zadovoljena za* M*),*

(b) $(\mathsf{consistent}\ M)$ *(tj.* $Invarijanta_{consistent}$ *je zadovoljena),*

(c) $(\mathsf{distinct}\ M)$ *(tj.* $Invarijanta_{distinct}$ *je zadovoljena),*

(d) $\forall l.\ l \in M \implies F$ @ (decisionsTo $l\ M$) $\vDash l$ *(tj.* $Invarijanta_{impliedLiterals}$ *je zadovoljena za* M*),*

(e) $conflict \implies F \vDash C$ *(tj.* $Invarijanta_{Centailed}$ *je zadovoljena),*

(f) $conflict = \top$*,*

(g) level je nivo povratnog skoka, tj. (isBackjumpLevel $level\ l\ C\ M$)*,*

(h) $M' =$ (prefixToLevel $level\ M$) @ $l^{\perp}$*,*

tada važi $\forall\ l.\ l \in M' \ \wedge\ l \notin$ (decisions M') $\implies \exists\ c.$ (isReason $c\ l\ M'$) $\wedge\ F \vDash c$ *(tj.* $Invarijanta_{reasonClauses}$ *je zadovoljena za* M'*),*

Dokaz. Za izvedene literale iz M, tvrđenje trivijalno sledi, pošto je P prefiks od M, a invarijante $Invarijanta_{reasonClauses}$ trivijalno ostaje zadovoljena kada se umesto označene valuacije M posmatra njen proizvoljni prefiks. Što se tiče literala l, tvrđenje sledi na osnovu leme 7.58. Zaista, na osnovu leme 7.29 važi (isUnit $C\ l$ (prefixToLevel $level\ M$)). Na osnovu $conflict = \top$ i $conflict \implies F \vDash C$ važi i $F \vDash C$. □

Sada ćemo dokazati dve leme koje karakterišu završna stanja sistema sa analizom konflikata i koje predstavljaju dva različita slučaja u teoremi potpunosti.

Lema 7.61. *Ako u stanju* $(M, F, C, conflict)$ *važi da je:*

(a) $conflict = \top$*,*

(b) (distinct M) *(tj.* $Invarijanta_{distinct}$ *je zadovoljena),*

(c) $conflict \implies M \vDash\neg C$ *(tj.* $Invarijanta_{Cfalse}$ *je zadovoljena),*

(d) pravila Explain *i* Backjump *nisu primenljiva,*

tada važi $C = [\,]$*.*

Dokaz. Pretpostavimo suprotno, odnosno pretpostavimo da važi da je $C \neq [\,]$. Razlikujmo sledeće slučajeve:

1. $\overline{C} \subseteq$ (decisions M). Neka je $\overline{l} =$ (lastAssertedLiteral $\overline{C}\ M$). Tada na osnovu leme 7.31, postoji nivo $level$ takav da važi (isBackjumpLevel $level\ l\ c\ M$). Tada, pošto je $conflict = \top$, pravilo Backjump može da se primeni, što je kontradikcija.
2. Postoji literal l, takav da je $l \in C$ i važi $\overline{l} \notin$ (decisions M). Pošto je $l \in C$ i $M \vDash\neg C$, važi $\overline{l} \in M$.

□

Lema 7.62. *Ako u stanju* $(M, F, C, conflict)$ *važi da je* $conflictFlag = \perp$ *i pravilo* Conflict *nije primenljivo, tada važi* $M \nvDash\neg F$*.*

Dokaz. Pretpostavimo suprotno, odnosno pretpostavimo da važi $M \models \neg F$. Tada postoji klauza c takva da je $c \in F$ i $M \models \neg c$. Iz $c \in F$ sledi i $F \models c$. Pošto je $conflict = \bot$, $F \models c$ i $M \models \neg c$, svi uslovi za primenu pravila Conflict su ispunjeni, što je u suprotnosti sa polaznom pretpostavkom. □

Sada dokazujemo teoremu o potpunosti sistema koja govori da su sva završna stanja sistema ili prihvatajuća ili odbacujuća.

Teorema 7.3'. *Neka je $(M, F, C, conflict)$ završno stanje sistema sa analizom konflikata u kome su zadovoljeni uslovi* (distinct M) *(tj. $Invarijanta_{distinct}$) i $conflict \implies M \models \neg C$ (tj. $Invarijanta_{Cfalse}$). Tada važi:*

(a) ili je $conflict = \bot$, $M \not\models \neg F$ i ne postoji literal takav da je (var l) $\in DecisionVars$, $l \notin M$ *i* $\overline{l} \notin M$.

(b) ili je $conflict = \top$ i $C = []$.

Dokaz. U stanju $(M, F, C, conflict)$ važi ili $conflict = \top$ ili $conflict = \bot$. Ukoliko važi $conflict = \bot$, prvi slučaj sledi na osnovu leme 7.62 i činjenice da pravilo Decide ne može da se primeni. Ukoliko važi $conflict = \top$, drugi slučaj sledi iz leme 7.61. □

Potpunost za zadovoljive i potpunost za nezadovoljive formule je posledica zaustavljanja, potpunosti i saglasnosti. Potpunost za nezadovoljive formule govori da će se sistem zaustaviti u odbacujućem stanju, a za zadovoljive formule da će se sistem zaustaviti u prihvatajućem stanju.

Teorema 7.4'. *Za svaku formulu F_0 važi da $([], F_0, [], \bot) \rightarrow^* (M, F, C, conflict)$, pri čemu je $(M, F, C, conflict)$ završno stanje, pri čemu važi:*

1. *Ako je formula F_0 zadovoljiva, tada važi $conflict = \bot$ i $M \not\models \neg F$.*
2. *Ako je formula F_0 nezadovoljiva i $DecisionVars \supseteq$ (vars F_0), tada važi $conflict = \top$ i $C = []$.*

Dokaz. Na osnovu teoreme 7.2', važi da algoritam nakon konačnog broja koraka stiže u završno stanje, tj. $([], F_0, [], \bot) \rightarrow^n (M, F, C, conflict)$. U tom stanju važi ili $conflict = \top$ ili $conflict = \bot$.

Na osnovu leme 7.3' sve invarijante važe u stanju $(M, F, C, conflict)$.

Na osnovu teoreme 7.3' važi da je u stanju $(M, F, C, conflict)$ ili $conflict = \top$ i $C = []$ ili je $conflict = \bot$, $M \not\models \neg F$ i ne postoji literal l takav da je (var l) $\in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$.

Pretpostavimo da je F_0 zadovoljiva formula, a da važi $conflict = \top$ i $C = []$. Tada na osnovu teoreme 7.1' važi da je F_0 nezadovoljiva, što je u kontradikciji sa našom polaznom pretpostavkom. Dakle za zadovoljive formule važi da je u završnom stanju $conflict = \bot$, $M \not\models \neg F$ i ne postoji literal l takav da je (var l) $\in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$.

Slično, pretpostavimo da je F_0 nezadovoljiva formula i $DecisionVars \supseteq$ (vars F_0), a da važi $conflict = \bot$, $M \not\models \neg F$ i ne postoji literal l takav da je (var l) $\in DecisionVars$, $l \notin M$ i $\overline{l} \notin M$. Tada na osnovu teoreme 7.1', važi da je F_0 zadovoljiva formula što je u kontradikciji sa našom polaznom pretpostavkom. Dakle, za zadovoljive formule važi da je $conflict = \top$ i $C = []$. □

7.7.4 Korektnost sistema sa analizom konflikata

Teorema 7.5'. *Neka je* $DecisionVars \supseteq (\mathsf{vars}\ F_0)$. *Za svaku formulu* F_0 *postoji završno stanje* $(M, F, C, conflict)$ *tako da važi* $([], F_0, [], \bot) \rightarrow^* (M, F, C, conflict)$ *i formula* F_0 *je zadovoljiva ako i samo ako važi* $conflict = \bot$.

Ovim su za sistem sa analizom konflikata dokazana sva svojstva definisana na strani 21.

8
Formalizacija u okviru sistema Isabelle

U ovoj glavi će biti prikazani primeri formalizacije teorija opisanih u ovom delu teze u okviru sistema Isabelle/Isar[1].

Formalizacija iskazne logike KNF formula. Osnovni tipovi dati u definiciji 6.1 su u okviru sistema Isabelle definisani na sledeći način:

```
types     Variable  = nat
datatype  Literal   = Pos Variable | Neg Variable
types     Clause    = "Literal list"
types     Formula   = "Clause list"
```

Jednostavne operacije kao što su, na primer, određivanje promenljive datog literala (uvedeno definicijom 6.2) ili određivanje suprotnog literala (uvedeno definicijom 6.5) datom literalu se formalizuju korišćenjem primitivne rekurzije na sledeći način:

```
primrec var ::  "Literal ⇒ Variable" -- the variable of a literal
where
  "var (Pos v) = v"
| "var (Neg v) = v"
```

```
primrec opposite ::  "Literal ⇒ Literal" -- the opposite of a literal
where
  "opposite (Pos v) = (Neg v)"
| "opposite (Neg v) = (Pos v)"
```

Pojam valuacije se uvodi na sledeći način:

```
types     Valuation = "Literal list"
```

Vrednost literala u valuaciji je definisana na sledeći način:

[1] Celokupna formalizacija je dostupna sa `http://argo.matf.bg.ac.rs`.

```
definition literalTrue ::  "Literal ⇒ Valuation ⇒ bool"
where
literalTrue_def [simp]:
  "literalTrue literal valuation = literal el valuation"

definition literalFalse ::  "Literal ⇒ Valuation ⇒ bool"
where
literalFalse_def [simp]:
  "literalFalse literal valuation = opposite literal el valuation"
```

Iako se istinitosna vrednost klauze u datoj valuaciji uvodi korišćenjem primitivne rekurzije, naredne dve leme daju karakterizaciju ovog pojma u skladu sa definicijom 6.7.

```
lemma clauseTrueIffContainsTrueLiteral:
  fixes clause ::  Clause and valuation ::  Valuation
  shows "clauseTrue clause valuation =
    (∃ literal.  literal el clause ∧ literalTrue literal valuation)"
by (induct clause) auto

lemma clauseFalseIffAllLiteralsAreFalse:
  fixes clause ::  Clause and valuation ::  Valuation
  shows "clauseFalse clause valuation =
    (∀ literal. literal el clause ⟶ literalFalse literal valuation)"
by (induct clause) auto
```

Sličan je slučaj i sa istinitosnom vrednošću formula:

```
lemma formulaTrueIffAllClausesAreTrue:
  fixes formula ::  Formula and valuation ::  Valuation
  shows "formulaTrue formula valuation =
    (∀ clause.  clause el formula ⟶ clauseTrue clause valuation)"
by (induct formula) auto

lemma formulaFalseIffContainsFalseClause:
  fixes formula ::  Formula and valuation ::  Valuation
  shows "formulaFalse formula valuation =
    (∃ clause.  clause el formula ∧ clauseFalse clause valuation)"
by (induct formula) auto
```

Naredna lema daje karakterizaciju pojma neprotivrečne valuacije (i ovaj put definisanog korišćenjem primitivne rekurzije) koja je u skladu sa definicijom 6.8:

```
lemma inconsistentCharacterization:
  fixes valuation ::  Valuation
  shows "inconsistent valuation =
    (∃ literal.  literalTrue literal valuation ∧
                 literalFalse literal valuation)"
by (induct valuation) auto
```

Na kraju, pojam modela i zadovoljivosti se uvodi na sledeći način:

```
definition model ::  "Valuation ⇒ Formula ⇒ bool"
where modelFormula_def [simp]:
  "model valuation formula ==
    consistent valuation ∧ formulaTrue formula valuation"

definition satisfiable ::  "Formula ⇒ bool"
where
  "satisfiable formula = (∃ valuation.  model valuation formula)"
```

Svi stavovi navedeni na početku ovog dela teze su formalno dokazani u okviru sistema Isabelle. Primera radi, naredna lema formalizuje dokaz pretposlednje stavke stava 6.8, odnosno tvrđenja $v \vDash \neg F \ \wedge \ F \vDash v \implies \neg(\mathsf{sat}F)$.

```
lemma formulaFalseInEntailedValuationIsUnsatisfiable:
  fixes formula ::  Formula and valuation ::  Valuation
  assumes "formulaFalse formula valuation" and
          "formulaEntailsValuation formula valuation"
shows "¬ satisfiable formula"
proof -
  from ‘formulaFalse formula valuation‘ obtain clause ::  Clause
  where "clause el formula" and "clauseFalse clause valuation"
    by (auto simp add:formulaFalseIffContainsFalseClause)
  {
    fix valuation' ::  Valuation
    assume "model valuation' formula"
    with ‘clause el formula‘ obtain literal ::  Literal
    where "literal el clause" and "literalTrue literal valuation'"
      by (auto simp add:  formulaTrueIffAllClausesAreTrue
                          clauseTrueIffContainsTrueLiteral)
    with ‘clauseFalse clause valuation‘
    have "literalFalse literal valuation"
      by (auto simp add:clauseFalseIffAllLiteralsAreFalse)
    with ‘formulaEntailsValuation formula valuation‘
    have "formulaEntailsLiteral formula (opposite literal)"
      unfolding formulaEntailsValuation_def
      by simp
    with ‘model valuation' formula‘
    have "literalFalse literal valuation'"
      by (auto simp add:formulaEntailsLiteral_def)
    from ‘literalTrue literal valuation'‘
         ‘literalFalse literal valuation'‘
         ‘model valuation' formula‘
    have "False"
      by (simp add:inconsistentCharacterization)
  }
  thus ?thesis
    by (auto simp add:satisfiable_def)
qed
```

Isti ovaj dokaz bi neformalno mogao da se iskaže na sledeći način:

Dokaz. Pošto važi $v \vDash \neg F$, postoji klauza c koja pripada formuli F i koja je netačna u valuaciji v, tj. $c \in F$ i $v \vDash \neg c$. Pretpostavimo suprotno tvrđenju leme da je formula F zadovoljiva, odnosno da postoji neprotivrečna valuacija

v' koja je model formule F. Tada je i klauza c tačna u valuaciji v' pa postoji literal l koji pripada klauzi c i koji je tačan u v', tj. $l \in c$ i $v' \vDash l$. Pošto važi $v \vDash \neg c$, i $l \in c$, literal l je netačan u valuaciji v, pa je literal $\overline{l}$ tačan u v. Na osnovu pretpostavke leme, valuacija v je logička posledica formule F, pa pošto joj $\overline{l}$ pripada važi da je literal $\overline{l}$ logička posledica formule F, tj. $F \vDash \overline{l}$. Na osnovu ovoga, $\overline{l}$ mora biti tačan u svakoj valuaciji u kojoj je tačna formula F, pa samim tim i u valuaciji v'. Međutim, ovo je u suprotnosti sa činjenicom da je i literal l tačan u v' tj. $v' \vDash l$, jer je pretpostavljeno da je v' neprotivrečna valuacija. Zbog dobijene kontradikcije, pretpostavka da je F zadovoljiva formula je neodrživa, pa tvrđenje leme važi. □

Formalizacija sistema promena stanja. U nastavku će biti ukratko prikazan način na koji su sistemi promena stanja za SAT definisani u okviru sistema Isabelle. Na ovaj način izvršena je formalizacija tri različita sistema: DPLL pretrage, sistema sa povratnim skokovima (definisanog po uzoru na [NOT06]) i sistema sa analizom konflikata (definisanog po uzoru na [KG07]).

Stanje sistema sa analizom konflikata je (umesto uređene četvorke) predstavljeno sledećim slogom.

```
record State =
"getF"              ::  Formula
"getM"              ::  LiteralTrail
"getConflictFlag"   ::  bool
"getC"              ::  Clause
```

Relacije kojima se formalizuju pravila prelaska su definisane po uzoru na definiciju 7.28. Na primer, pravilo Decide je formalizovano na sledeći način.

```
definition
appliedDecide::  "State ⇒ State ⇒ Variable set ⇒ bool"
where
"appliedDecide stateA stateB decisionVars =
  ∃ l.
     (var l) ∈ decisionVars ∧
     ¬ l el (elements (getM stateA)) ∧
     ¬ opposite l el (elements (getM stateA))∧

     getF stateB = getF stateA ∧
     getM stateB = getM stateA @ [(l, True)] ∧
     getConflictFlag stateB = getConflictFlag stateA ∧
     getC stateB = getC stateA
"
definition
applicableDecide ::  "State ⇒ Variable set ⇒ bool"
where
"applicableDecide state decisionVars =
  ∃ state'.  appliedDecide state state' decisionVars"
```

Relacija prelaska je definisana na sledeći način, uz korišćenje već definisanog pojma tranzitivno-refleksivnog zatvorenja relacije (koje se označava sa *).

```
definition
transition ::  "State ⇒ State ⇒ Formula ⇒ Variable set ⇒ bool"
where
"transition stateA stateB F0 decisionVars =
   appliedDecide        stateA stateB decisionVars ∨
   appliedUnitPropagate stateA stateB F0 decisionVars ∨
   appliedConflict      stateA stateB ∨
   appliedExplain       stateA stateB ∨
   appliedLearn         stateA stateB ∨
   appliedBackjump      stateA stateB
"
definition "transitionRelation F0 decisionVars =
   ((stateA, stateB). transition stateA stateB F0 decisionVars)*"
```

Početna i završna stanja su definisana na sledeći način:

```
definition isInitialState ::  "State ⇒ Formula ⇒ bool"
where
"isInitialState state F0 =
   getF state = F0 ∧
   getM state = [] ∧
   getConflictFlag state = False ∧
   getC state = []"

definition isFinalState ::  "State ⇒ Formula ⇒ Variable set ⇒ bool"
where
"isFinalState state F0 decisionVars =
   ¬ (∃ state'.  transition state state' F0 decisionVars)"
```

Na kraju su formulisane i dokazane teoreme o zaustavljanju i parcijalnoj korektnosti. Teorema o zaustavljanju govori o tome da je relacija prelaska dobro zasnovana relacija na skupu stanja dostupnih iz nekog početnog stanja.

```
theorem wfTransitionRelation:
   fixes decisionVars ::  "Variable set" and F0 ::  "Formula"
   assumes "finite decisionVars" and "isInitialState state0 F0"
   shows "wf {(stateB, stateA).
              (state0, stateA) ∈ transitionRelation F0 decisionVars ∧
              (transition stateA stateB F0 decisionVars)}"
```

Posledica ove teoreme govori o tome da ne postoje beskonačni lanci prelazaka koji počinu od nekog početnog stanja. Primetimo da je, iako se u svakodnevnoj matematici pojam beskonačnog lanca olako koristi, njegova formalizacija prilično kompleksna[2]. Kako bi se ovaj pojam zaobišao, naredna posledica je formulisana tako da tvrdi da ne postoji skup stanja kojem bi pripadala stanja nekog beskonačnog lanca (pa samim tim ne može da postoji ni beskonačni lanac).

[2]Definicija beskonačne liste zahteva korišćenje ko-induktivne definicije.

```
corollary noInfiniteTransitionChains:
  fixes F0 ::  Formula and decisionVars ::  "Variable set"
  assumes "finite decisionVars"
  shows "¬ ∃ Q::(State set).
              ∃ state0 ∈ Q. isInitialState state0 F0 ∧
              ∀ state ∈ Q. ∃ state' ∈ Q.
                transition state state' F0 decisionVars
"
```

```
theorem partialCorrectness:
  fixes F0 ::  Formula and decisionVars ::  "Variable set" and
       state0 ::  State and state ::  State
  assumes
    "vars F0 ⊆ decisionVars" and

    "isInitialState state0 F0" and
    "(state0, state) ∈ transitionRelation F0 decisionVars" and
    "isFinalState state F0 decisionVars"

  shows
  "satisfiable F0 = (¬ getConflictFlag state)"
```

Pomenimo još, kao zanimljivost, da kompleksnost formalnog dokaza naizgled trivijalnih činjenica može biti iznenađujuće velika. U okviru celokupne formalizacije u okviru sistema Isabelle, najkompleksniji formalni dokaz je bio dokaz leme 7.13, koji rađen indukcijom po konačnom skupu Vbl i koji je dugačak oko 300 linija.

```
lemma finiteDistinctAndConsistentTrailsWithGivenVariableSet:
  fixes Vbl ::  "Variable set"
  assumes "finite Vbl"
  shows "finite {(M::LiteralTrail).  vars (elements M) = Vbl ∧
                                     distinct (elements M)}"
```

Deo II

Implementacija

9 Implementacija u imperativnom pseudojeziku

U ovom poglavlju će biti prikazane osnove implementacije modernih SAT rešavača. Opisi će biti dati u obliku pseudokoda napisanog u imperativnom pseudojeziku napravljenom po uzoru na jezik Pascal. Postupan, tutorijalski, pristup izlaganju koji je bio prisutan u prethodnim delovima teze, biće prisutan i u ovoj glavi, tako što će implementacija biti prikazivana i nadgrađivana kroz nekoliko verzija rešavača. Biće navedena implementacija osnovne pretrage, DPLL pretrage i sistema sa analizom konflikata, a nakon toga će biti dat prikaz implementacije određenih tehnika niskog nivoa koje doprinose efikasnijoj implementaciji svih ovih sistema.

Celokupna implementacija koja će biti prikazana je zasnovana na formalnom okviru definisanom u delu *Formalizacija*, tj. zasniva se na apstraktim sistemima promena stanja. Međutim, kako bi se dobila efektivna implementacija, neke aspekte ovih sistema je neophodno konkretizovati. Na primer, iako su svi opisani sistemi pravila nedeterministički i redosled primene pravila ne utiče na korektnost procedura koje formalizuju, prilikom implementacije sistem je neophodno učiniti determinističkim i neophodno je fiksirati strategiju primenjivanja pravila. Takođe, neka pravila je moguće primeniti na više načina i prilikom izgradnje konkretne implementacije neophodno je fiksirati neki od načina na koji će se dato pravilo primeniti. U nekim slučajevima, evidentno je da određeni način i redosled primene pravila dovodi do veće efikasnosti. Tako je, na primer, jasno da zaključivanje treba da ima prioritet u odnosu na pretragu, odnosno, evidentno je da pravilo propagacije jediničnih klauza Unit-Propagate treba da ima prioritet u odnosu na pravilo pretpostavljanja Decide. S druge strane, postoje aspekti za koje nije poznata strategija koja *a priori* dovodi do najveće efikasnosti. Tako, na primer, nije jasno kako izabrati literal u pravilu Decide. Implementacija koja će biti specifikovana u ovom poglavlju, napravljena je tako da „tvrdo" kodira aspekte primene pravila koji evidentno dovode do dobre efikasnosti, dok se razrešavanje diskutabilnih aspekata ostavlja heurističkim komponentama koje ostavlja nespecifikovanim.

Na kraju ove glave, biće ukratko opisan i način verifikovanja navedenog pseudokoda.

9.1 Opis pseudojezika

Svi algoritmi u narednom tekstu će biti specifikovani u pseudojeziku koji podseća na jezik Pascal.

Svaka specifikacija algoritma počinje spiskom globalnih promenljivih koji se navode u sledećem obliku:

```
var
   var_1 :  Type_1
   ...
   var_k :  Type_k
```

$\texttt{Type}_i$ može da bude bilo koji tip koji je podržan meta-teorijom.

Definicije funkcija su oblika:

```
function name (arg_1 :  Type_1, ..., arg_k :  Type_k) :  Type
begin
   niz iskaza
end
```

gde je $\texttt{Type}_i$ tip argumenta $\texttt{arg}_i$ i `Type` je povratni tip funkcije. Ukoliko funkcije ne vraća vrednost, njen povratni tip se ne navodi. Funkcije vraćaju vrednost tako što ih dodeljuju specijalnoj promenljivoj `ret`. Takođe je podržana i eksplicitna `return` naredba. Parametri se prenose po vrednosti. Parametri u listi parametara koji su označi sa `var` se prenose po referenci. Funkcije označene kao `const` ne menjaju stanje programa. Deklaracije lokalnih promenljivih neće biti navođene ukoliko je njihov tip jasan iz konteksta.

Nizovi *iskaza* su odvojeni simbolom `;`. Jezik podržava sledeće *iskaze*.

```
x := izraz
if uslov then iskaz
if uslov then iskaz else iskaz
while uslov do iskaz
repeat iskaz until uslov
function_name(arg_1, ..., arg_n)
return
```

Uslovi i *izrazi* uključuju promenljive, pozive funkcija i čak i izraze meta-teorije. Blokovska struktura programa će biti naglašena nazubljivanjem kôda.

Dopustićemo i korišćenje izraza meta-teorije u oviru specifikacije algoritama. Postoje dvojaki razlozi za to. Prvo, želimo da uprostimo prezentaciju tako što ćemo preskočiti eksplictnu implementaciju nekih trivijalnih koncepata. Na primer, pošto naš programski jezik koristi liste, direktno ćemo koristiti operacije i oznake iz meta-teorije, imajući u vidu da se njihova korektna implementacija može jednostavno napraviti. Drugo, tokom razvoja algoritma u prvim fazama namerno ostavljamo neke uslove na prilično apstraktnom nivou i opis njihove konretne implementacije ostavljamo za kasnije. U tim slučajevima koristićemo izraze meta-teorije kako bismo implicitno specificirali (post)uslove za neimplementirane delove koda. Na primer, pisaćemo `if` $M \vDash F$ `then`, bez navođenja kako se uslov $M \vDash F$ može efektivno implementirati. Ovi neimplementirani delovi koda će biti označeni, kao što je to urađeno u datom primeru. Umesto ovoga, moguće bi bilo definisati funkciju

`function satisfies(M : Trail, F : Formula) : Boolean`, čiji bi postuslov bio $\{ret \iff M \vDash F\}$. U tom slučaju bi navedeni test bio zamenjen sa `if satisfies(M, F) then`.

9.2 Implementacija osnovnog algoritma pretrage

Za početak opišimo implementaciju osnovnog algoritma pretrage. Formalna osnova algoritma je osnovna pretraga zasnovana na pravilima Decide i Backtrack i opisana u poglavlju 7.2. Algoritam ispituje zadovoljivost formule F_0. Označena valuacija M predstavlja tekuću parcijalnu valuaciju. Na početku rada algoritma, valuacija M je prazna. Literali se dodaju u valuaciju M korišćenjem funkcije applyDecide, pri čemu se tako dodati literali označavaju kao pretpostavljeni. Kada god se novi literal doda u valuaciju M proverava se da li je formula F_0 postala netačna, tj. da li je došlo do konflikta. Kada se konflikt ustanovi, poziva se funkcija applyBacktrack koja uklanja poslednji pretpostavljeni literal iz valuacije M i njegov suprotni literal dodaje u M, ali ovaj put kao zaključeni literal. Ukoliko se konflikt javi kada u valuaciji više nema pretpostavljenih literala, tada je formula F_0 nezadovoljiva. Ukoliko M sadrži sve promenljive fiksiranog skupa $DecisionVars$ (za koji se pretpostavlja da sadrži sve promenljive formule F_0), a nije došlo do konflikta, tada je formula F_0 zadovoljiva i M je njen model.

SAT Solver v.1

```
var                                                    +
  satFlag : {UNDEF, SAT, UNSAT}                        +
  F0 : Formula                                         +
  M : Trail                                            +
```

```
function solve (F0 : Formula) : {SAT, UNSAT}           +
begin                                                  +
  satFlag = UNDEF;                                     +
  M := [];                                             +
  while satFlag = UNDEF do                             +
    if M ⊨¬ F0 then                                    +
      if (decisions M) = [] then                       +
        satFlag := UNSAT                               +
      else                                             +
        applyBacktrack()                               +
    else                                               +
      if (vars M) ⊇ DecisionVars then                  +
        satFlag := SAT                                 +
      else                                             +
        applyDecide()                                  +
end                                                    +
```

```
function applyDecide()                                          +
begin                                                           +
  l := selectLiteral();                                         +
  assertLiteral(l, true)                                        +
end                                                             +
```

```
function applyBacktrack()                                       +
begin                                                           +
  l := (lastDecision M);                                        +
  M := (prefixBeforeLastDecision M);                            +
  assertLiteral(¬l, false)                                      +
end                                                             +
```

```
function assertLiteral(l : Literal, decision : Boolean)         +
begin                                                           +
  M := M @ [(l, decision)]                                      +
end                                                             +
```

```
{DecisionVars \ (vars M) ≠ ∅}
const function selectLiteral() : Literal                        +
{(var ret) ∈ DecisionVars ∧ (var ret) ∉ (vars M)}
```

Skoro svi izrazi meta-teorije dati u prethodnom kôdu se mogu jednostavno efektivno implementirati u realnom programskom jeziku. Uslov (decisions M) = [] se može zameniti ekvivalentnim uslovom (currentLevel M) = 0. Izuzetak je uslov $M \models \neg F_0$ koji je veoma visokog nivoa i nije jednostavno napraviti njegovu efikasnu implementaciju. Jednostavno izračunavanje istinitosnih vrednosti svih klauza u F_0 vodi ka veoma neefikasnom rešavaču i zato je neohodno razviti efikasniju implementaciju. Jedan način da se ovo uradi je objašnjen u poglavlju 9.9.

Funkciju selectLiteral ostavljamo nedefinisanom i samo tražimo da zadovoljava navedene uslove (date u obliku Horove trojke).

9.3 Implementacija DPLL pretrage

Naredna verzija algoritma implementira DPLL pretragu formalizovanu u poglavlju 7.3. Pravilo UnitPropagate je implementirano kroz funkciju applyUnitPropagate koja pronalazi jedinične literale u formuli F_0 i dodaje ih u valuaciju M kao zaključene literale. Funkcija exhaustiveUnitPropagate iscrpno primenjuje ovo pravilo, sve dok je njegova primena moguća.

SAT Solver v.2

```
function solve (F0 : Formula) : (SAT, UNSAT)
.....................
  while satFlag = UNDEF do
    exhaustiveUnitPropagate();                    +
    if M ⊨¬ F0 then
.....................
```

```
function exhaustiveUnitPropagate()                +
begin                                             +
  repeat                                          +
    ret := applyUnitPropagate();                  +
  until M ⊨¬ F0 ∨ ret = false                     +
end                                               +
```

```
function applyUnitPropagate() : Boolean           +
begin                                             +
  if ∃c.∃l. c ∈ F0 ∧ (isUnit c l M) then          +
    assertLiteral (l, false);                     +
    ret := true                                   +
  else                                            +
    ret := false                                  +
end                                               +
```

Opet je većinu izraza meta-teorije moguće jednostavno implementirati u realnom programskom jeziku. Izuzetak je ovaj put uslov $\exists c.\exists l.\ c \in F_0 \wedge (\text{isUnit } c\ l\ M)$ koji se koristi da bi se otkrile jedinične klauze. Efikasni način da se implementira ovaj uslov će biti naveden u poglavlju 9.9.

9.4 Implementacija sistema sa analizom konflikata

U ovom poglavlju ćemo navesti implementaciju rešavača zasnovanog na sistemu sa analizom konflikata koji je formalizovan u poglavlju 7.7. Ova implementacija se zasniva na analizi konflikata koja je opisana pravilima Conflict i Explain, učenju opisanom pravilom Learn i povratnim skokovima opisanim pravilom Backjump.

Ova verzija rešavača radi veoma slično kao *SAT solver v.2* do tačke kada je $M \models\neg F$ tj. do tačke kada dođe do konflikta. U tom trenutku se vrši analiza konflikta koja je implementirana kroz funkcije applyConflict i applyExplain. Funkcija applyConflict inicijalizuje klauzu C na neku klauzu formule F koja je netačna u M. Funkcija applyExplain uklanja literal l iz klauze C izvođenjem jednog koraka rezolucije između klauze C i klauze koja je razlog za propagaciju literala $\overline{l}$.

Ukoliko se konflikt desi na nekom višem nivou označene valuacije M (tj. kada postoji bar jedan pretpostavljeni literal u M), konstruiše se klauza povratnog skoga koja zadovoljava uslov postojanja tačke jedinistvne implikacije (firstUIP). Ova se klauza konstruiše korišćenjem funkcije applyExplainUIP. Ona iterativno uklanja poslednje zadovoljene literale iz $\overline{C}$ pozivajući applyExplain sve do prvog trenutka u kome klauza C ne dostigne tačku jedinstvene implikacije. Funkcija applyLearn dodaje konstruisanu klauzu povratnog skoka C u formulu F. Funkcija applyBackjump uklanja literale iz valuacije M sve dok C ne postane jedinična klauza kada dodaje njen jedinični literal u M. Primetimo da se dodavanje ovog literala ne mora uraditi od strane funkcije već se može prepustiti funkciji applyUnitPropagate. Zato neke implementacije funkcije applyBackjump preskaču dve poslednje linije kôda koji ovde navodimo. Kada se konflikt razreši, rešavač nastavlja da radi kao *SAT solver v.2*.

Ukoliko se desi konflikt na nultom nivou (tj. kada ne postoji ni jedan pretpostavljeni literal u M), tada se efektivno konstruiše prazna klauza C korišćenjem funkcije applyExplainEmpty. Ona iterativno uklanja sve poslednje zadovoljene literale iz $\overline{C}$ pozivajući applyExplain, sve dok klauza C ne postane prazna. Moguće je proširiti rešavače mogućnošću generisanja dokaza nezadovoljivosti formule zasnovnane na korišćenju rezolucije i ova eksplicitna konstrukcija prazne klauza čini taj proces uniformnijim.

SAT Solver v.3

```
var
   satFlag : {UNDEF, SAT, UNSAT}
   M : Trail
   F : Formula                                                    *
   C : Clause                                                     +
   reason : Literal => Clause                                     +
```

```
function solve (F0 : Formula) : (SAT, UNSAT)
begin
   satFlag = UNDEF;
   M := [];
   F := F0;                                                       +
   while satFlag = UNDEF do
      exhaustiveUnitPropagate();
      if M ⊨¬ F then
         applyConflict();                                         +
         if (currentLevel M) = 0 then
            applyExplainEmpty();                                  +
            applyLearn();                                         +
            satFlag = UNSAT
         else
            applyExplainUIP();                                    +
            applyLearn();                                         +
            applyBackjump()                                       *
      else
```

```
        if |(vars M)| = |Vbl| then
          satFlag = SAT
        else
          applyDecide()
end
```

```
function applyUnitPropagate() :  Boolean
begin
  if ∃c.∃l.  c ∈ F0 ∧ (isUnit c l M) then
    assertLiteral (l, false);
    setReason(l, c);
    ...
```

```
function applyConflict()
begin
  C := getConflictClause()
end
```

```
function applyExplainUIP()
begin
  while ¬ isUIP()
    applyExplain((lastAssertedLiteral C̄ M))
end
```

```
const function isUIP() :  Boolean
begin
  l := (lastAssertedLiteral C̄ M);
  if ∃l'.  l'∈C̄ ∧ l'≠l ∧ (level l' M) = (level l M) then
    ret := false
  else
    ret := true
end
```

```
function applyExplainEmpty()
begin
  while C ≠ []
    applyExplain((lastAssertedLiteral C̄ M))
end
```

```
function applyExplain(l : Literal)                    +
begin                                                 +
  reason := getReason(l);                             +
  C := (resolvent C reason l̄)                         +
end                                                   +
```

```
function applyLearn()                                 +
begin                                                 +
  F = F @ C                                           +
end                                                   +
```

```
function applyBackjump()                              +
begin                                                 +
  l := (lastAssertedLiteral C̄ M);                     +
  level := getBackjumpLevel();                        +
  M := (prefixToLevel M level);                       +
  assertLiteral(l̄, false)                             +
  setReason(l̄, C);                                    +
end                                                   +
```

```
const function getBackjumpLevel : int                 +
begin                                                 +
  l := (lastAssertedLiteral C̄ M);                     +
  if C̄ \ l ≠ [] then                                  +
    ret := (maxLevel C̄ \ l M)                         +
  else                                                +
    ret := 0                                          +
end                                                   +
```

```
const function getReason(l : Literal) : Clause        +
begin                                                 +
  ret := reason(l)                                    +
end                                                   +
```

```
function setReason(l : Literal, c : Clause)           +
begin                                                 +
```

```
  reason(l) := c                                                    +
end                                                                 +
```

```
{M ⊨¬ F}
const function getConflictClause() :  Clause                        +
{M ⊨¬ ret}
```

Primetimo da su ponovo neke funkcije ostavljene bez implementacije. Funkcija getConflictClause vraća proizvoljnu klauzu formule F koja je netačna u M. Ona zadovoljava Horovu trojku $\{M \models\neg F\}$ getConflictClause() $\{M \models\neg ret\}$.

Funkcija getReason se koristi da bi se pronašla klauza koja je razlog propagacije datog zaključenog literala. Ona zadovoljava Horovu trojku $\{l \in M \wedge l \notin (\text{decisions } M)\}$ `getReason(l)` $\{ret \in F \wedge (\text{isReason } ret\ l\ M)\}$. Svi zakjlučeni literali u M su zaključeni kao rezultat propagacije jediničnih klauza ili kao rezultat povratnih skokova i razlozi njihove propagacije se pamte u preslikavanju *reason* korišćenjem funkcije setReason.[1]

Efikasne strukture podataka

Kôd naveden u poglavlju 9.4 ostavlja funkciju resolvent nedefinisanom. U nastavku će biti prikazan način na koji ona efikasno može biti implementirana. Implementacija koju navodimo je veoma slična onoj korišćenoj u okviru rešavača MiniSAT [ES04]. Bez velikog gubitka na opštosti, pretpostavićemo da će konfliktna klauza (i samim tim i klauza C) sadržati literal sa tekućeg nivoa označene valuacije, tj. $(\text{currentLevel } M) = (\text{maxLevel } \overline{C}\ M)$. Ovo će uvek biti slučaj ako postoji garancija da se ne vrše pretpostavke u slučaju kada $M \models\neg F$, što je svakako slučaj u implementaciji koju navodimo.[2] Takođe, poželjno je pretpostaviti da C ne sadrži višestruka pojavljivanja istog literala. Zato je u svakom pozivu funkcije resolvent, kada se gradi unija dve klauze, potrebno ukloniti višestruka pojavljivanja literala. Kako bi se postigla efikasnija implementacija, umesto obične liste literala, za reprezentaciju klauze C potrebno je koristiti neku bolju strukturu podataka. U implementaciji koju predlažemo, klauza C će biti predstavljena preslikavanjem C_H koje preslikava literale u bulovske promenljive, tako da važi $C_H(l) = true$ ako i samo ako je $l \in C$. Primetimo da je korišćenjem ovakve strukture podataka moguće u konstantnom vremenu odgovoriti da li se neki literal sadrži u klauzi C i zato je pogodna za širok spektar operacija sa ovom klauzom. Kada se završi proces analize konflikta, lista literala klauze C se može rekonstruisati prolaskom kroz preslikavanje C i traženjem literala koji se slikaju u $\top$. Primetimo da ovaj relativno zahtevan obilazak može da se izbegne, korišćenjem činjenice da se tokom firstUIP analize konflikata samo literali sa najvišeg nivoa (tj. tekućeg nivoa označene valuacije M) uklanjaju iz klauze C. Dakle, kada se literal sa nekog nižeg nivoa ubaci u klauzu C, on ne može da se ukloni do kraja firstUIP procesa rezolucije. Takođe, pošto na kraju analize konflikata važi UIP uslov (tj. u C

[1] U realnom programskom jeziku, umesto samih klauza, dovoljno bi bilo pamtiti samo pokazivače na klauze.

[2] Ovo je veoma važan uslov i za korektnost sheme dva posmatrana literala čiju ćemo implementaciju opisati u poglavlju 9.9.

postoji tačka jedinstvene implikacije), klauza C sadrži tačno jedan literal sa tekućeg (najvišeg) nivoa označene valuacije M. Zato je korisno održavati listu C_P koja sadrži literale klauze C pobijenih na nižim nivoima u M i održavati literal C_l koji je poslednji literal iz $\overline{C}$ koji se nalazi u M. Kada se analiza završi, na osnovu C_P i C_l je moguće rekonstruisati klauzu C, bez potrebe za obilaskom preslikavanja C_H. Kako bi se optimizovalo ispitivanje postojanja tačke jedinstvene implikacije (tj. funkcija isUIP), održava se i broj C_n literala sa najvišeg nivoa valuacije M (što je u našem slučaju (currentLevel M)).

Naredni kôd predstavlja implementaciju algoritma *SAT solver v.3* koja koristi opisane strukture podataka. Klauza C (odnosno njene komponente C_H, C_P, C_n) se menjaju samo kroz funkcije addLiteral i removeLiteral, dok funkcija findLastAssertedLiteral postavlja literal C_l.

SAT Solver v.4

```
...
C_H : Literal => Boolean                                    *
C_P : Clause                                                *
C_l : Literal                                               *
C_n : nat                                                   *
...
```

```
function applyConflict()
begin
  C_H := {};                                                *
  foreach l : l ∈ getConflictClause() do                    *
    addLiteral(l)                                           *
  findLastAssertedLiteral();                                +
end
```

```
function applyExplainUIP()
begin
  while ¬ isUIP() do
    applyExplain(C_l);                                      *
  buildC()                                                  +
end
```

```
function buildC()                                           +
  C := C_P @ ¬C_l                                           +
end                                                         +
```

```
const function isUIP() : Boolean
begin
  if C_n = 1 then ret := true else ret := false             *
end
```

```
function applyExplain(l : Literal)
begin
  reason := getReason(l);
  resolve(reason, l)                                    *
  findLastAssertedLiteral();                            +
end
```

```
function resolve(clause : Clause, l : Literal)          +
begin                                                   +
  removeLiteral(¬l);                                    +
  foreach l': l' ∈ clause ∧ l' ≠ l do                   +
    addLiteral(l')                                      +
end                                                     +
```

```
function applyBackjump()
begin
  level := getBackjumpLevel();
  M := (prefixToLevel M level);
  assertLiteral(¬C_l, false)                            *
  setReason(¬C_l, C);                                   *
end
```

```
const function getBackjumpLevel : int
begin
  if C_P ≠ [] then                                      *
    ret := (maxLevel ¬C_P M)                            *
  else
    ret := 0
end
```

```
function addLiteral(l : Literal)                        +
begin                                                   +
  if C_H(l) = false then                                +
    C_H(l) := true;                                     +
```

```
    if (level ¬l M) = (currentLevel M) then             +
      C_n := C_n + 1                                    +
    else                                                +
      C_P := C_P @ l                                    +
end                                                     +
```

```
function removeLiteral(l : Literal)                    +
begin                                                  +
  C_H(l) := false;                                     +
  if (level ¬l M) = (currentLevel M) then              +
    C_n := C_n - 1                                     +
  else                                                 +
    C_P := C_P \ l                                     +
end                                                    +
```

```
function findLastAssertedLiteral()                     +
begin                                                  +
  repeat                                               +
    C_l := (last M)                                    +
  until C_H(¬C_l) = true                               +
end                                                    +
```

9.5 Minimalizacija klauze povratnog skoka

Naredna verzija SAT rešavača uvodi minimalizaciju klauze analize konflikta (neformalno opisanu u poglavlju 4.4.2) i proširuje SAT rešavač v.3.

SAT Solver v.5

```
function solve (F_0 : Formula) : {SAT, UNSAT}
.....................
  applyExplainUIP();
  applyExplainSubsumption();                           +
  applyLearn();
  applyBackjump()
.....................
```

```
function applyExplainSubsumption()                     +
begin                                                  +
  foreach l: l ∈ ¬C ∧ l ∉ (decisions M) do              +
    if subsumes(C, getReason(l) \ l) then              +
      applyExplain(l)                                  +
end                                                    +
```

```
const function subsumes (c1 : Clause, c2 : Clause) : Bool  +
begin                                                  +
  if c2 ⊆ c1 then ret := true else ret := false        +
end                                                    +
```

Provera obuhvatanja mora da se uradi veoma pažljivo, kako ne bi postala usko grlo ovog dela rešavača. Strukture podataka uvedene u 9.4 omogućavaju veoma efikasnu implementaciju.

9.6 Zaboravljanje

Naredni kôd implementira pravilo Forget, koje je neformalno opisano u poglavlju 4.4.1, a zatim formalizovano u okviru sistema sa učenjem i zaboravljanjem u poglavlju 7.5.

SAT Solver v.6

```
function solve (F0 : Formula) : (SAT, UNSAT)
.........................
          if shouldForget() then                    +
             applyForget()                          +
          applyDecide()
        end
.........................
```

```
function applyForget()                              +
begin                                               +
  newF := [];                                       +
  foreach c : c ∈ F do                              +
    if shouldForgetClause(c) ∧ isLearnt(c) ∧ ¬isReason(c) then  +
      removeClause(c)                               +
    else                                            +
      newF := newF @ c;                             +
  F := newF                                         +
end                                                 +
```

```
{⊤}
const function shouldForget() : Boolean             +
{⊤}
```

```
{⊤}
const function shouldForgetClause(Clause : c) : Boolean  +
{⊤}
```

Funkcija isLearnt nije specifikovana. Njena implementacija može biti veoma jednostavna. Na primer, formula F može prirodno da se podeli na početne klauze F_0 i naučene klauze F_l i tada funkcija isLearnt proverava da li je c element formule F_l.

Funkcija isReason određuje da li je klauza c razlog za propagaciju nekog njenog literala.

Funkcija shouldForget određuje kada započeti sa zaboravljanjem, dok funkcija function shouldForgetClause određuje koje klauze zaboraviti. Zajednički, ove dve funkcije čine strategiju zaboravljanja. Obično se strategija zaboravlja zasniva na broju naučenih klauza, ali je takođe moguće i korišćenje alternativnih kriterijuma (npr. ukupan broj literala u formuli). Klauze se obično zaboravljaju kao rezultat njihove slabe aktivnosti, odnosno slabog učestvovanja u propagacijama i konfliktima ili kao rezultat njihove dužine.

9.7 Otpočinjanje iznova

Naredni kôd implementira pravilo Restart koje je neformalno opisano u poglavlju 4.4.3, a zatim i formalizovano u okviru sistema sa otpočinjanjem iznova u poglavlju 7.6.

SAT Solver v.7

```
function solve (F_0 : Formula) : (SAT, UNSAT)
.....................
       else
          if shouldRestart() then                    +
             applyRestart()                          +
          applyDecide()
.....................
```

```
function applyRestart()                              +
begin                                                +
  M := (prefixToLevel M 0)                           +
end                                                  +
```

```
{⊤}
const function shouldRestart() : Boolean             +
{⊤}
```

9.8 Literali nultog nivoa

Pošto važi $(\mathsf{level}\ l\ M) = 0 \implies (\mathsf{decisionsTo}\ M\ l) = [\,]$, invarijanta $Invarijanta_{impliedLiterals}$ (uvedena u poglavlju 7.2.1, a koja bi trebalo da važi i tokom izvršavanja koda datog u ovoj glavi) obezbeđuje da su literali na nultom nivou označene valuacije logičke posledice same formule. Ovi literali imaju posebnu ulogu tokom procesa rešavanja.

Jednočlane klauze

Pod pretpostavkom da se literali sa nultog nivoa označene valuacije iz nje nikada ne uklanjaju, može se obezbediti da formula F ni u jednom trenutku ne sadrži klauze sa manje od dva literala. Zaista, umesto dodavanja jednočlane

klauze $[l]$ u formulu F, moguće je njen literal l dodati direktno na nulti nivo označene valuacije M. Ova tehnika omogućava jednostavniju implementaciju sheme dva posmatrana literala koja će biti opisana u poglavlju §9.9. U slučaju implementacije zasnovane na ovoj tehnici, neophodno je zameniti naredbu F := F_0 u prethodnoj verziji koda. SAT rešavač v.8 se zasniva na SAT rešavaču v.5 bez obzira da li su unete izmene iz SAT rešavača v.6 ili v.7.

SAT Solver v.8

```
function solve (F0 : Formula) : (SAT, UNSAT)
...
  M := [];
  foreach clause: clause ∈ F0 do                    *
    addClause(clause);                              *
...
```

```
function addClause(clause : Clause)                 +
begin                                               +
  clause := removeDuplicateLiterals(clause);        +
  clause := removeFalsifiedLiterals(clause);        +
  if containsTrueLiteral(clause) then               +
    return;                                         +
  else if (length clause) = 0 then                  +
    satFlag := UNSAT                                +
  else if (length clause) = 1 then                  +
    assertLiteral((head clause), false);            +
    exhaustiveUnitPropagate()                       +
  else if isTautological(clause) then               +
    return;                                         +
  else                                              +
    F := F @ clause                                 +
end                                                 +
```

```
const function containsTrueLiteral(clause: Clause) : Boolean   +
begin                                               +
  ret := false;                                     +
  foreach l : l ∈ clause do                         +
    if M ⊨ l then ret := true                       +
end                                                 +
```

```
const function removeDuplicateLiterals(clause : Clause) : Clause
+
begin                                                          +
  ret := [];                                                   +
  foreach l : l ∈ clause do                                    +
    if l ∉ ret then ret := ret @ l;                            +
end                                                            +
```

```
const function removeFalsifiedLiterals(clause : Clause) : Clause +
begin                                                            +
  ret := [];                                                     +
  foreach l : l ∈ clause do                                      +
    if M ⊭¬l then ret := ret @ l;                                +
end                                                              +
```

```
const function isTautological(c : clause)                      +
  if ∃ l. l∈c ∧ l̄∈c then ret:=true else ret:=false             +
end                                                            +
```

Data procedura inicijalizacije takođe obezbeđuje da ni jedna klauza u F ne sadrži ponovljene literali, kao i da ne sadrži suprotne literale.

Proces učenja je neophodno promeniti na način koji obezbeđuje učenje isključivo klauza sa bar dva (različita) literala:

```
function applyLearn()
begin
  if (length C) > 1 then                                       +
    F = F @ C
end
```

Pošto se učenje ne primenjuje na jednočlane klauze već se njihovi literali postavljaju direktno na nulti nivo označene valuacije M, ovim literalima neće biti pridružena klauza iz F koja predstavlja razlog njihove propagacije. Ovo znači da se oni tokom analize konflikata ne mogu objašnjavati i uklanjati iz klauze C korišćenjem funkcije applyExplain (i dalje je ispravno pozivati funkciju getReason(l) za sve zaključene literale sa viših nivoa označene valuacije M). Kako bi se ovaj problem prevazišao, klauza C se tokom analize konflikata gradi tako da ne sadrži nijedan literal sa nultog nivoa. Ovi se literali jednostavno preskaču tokom procesa analize konflikata i funkcija applyExplain se modifikuje tako da nakon njene primene klauza C postaje $(\mathsf{resolvent}\ C\ c\ \bar{l})\ \setminus\ \overline{(\mathsf{prefixToLevel}\ M\ 0)}$. Kada se koriste efikasne strukture podataka koje su opisane u §9.4, funkcija addLiteral se menja na sledeći način.

```
function addLiteral(l :  Literal)
begin
  if C_H(l) = false then
    if (level ¬l M) ≠ 0 then                                    +
      C_H(l) := true;
      if (level ¬l M) = (currentLevel M)
        C_n := C_n + 1
      else
        C_P := C_P @ l
end
```

Primetimo da ove izmene komplikuju generisanje dokaza nezadovoljivosti, pošto se neki literali klauze C iz nje uklanjaju bez eksplicitnog poziva funkcije applyExplain.

Uprošćavanje skupa klauza

Uvek kada se literal l dodaje na nulti nivo označene valuacije M, kako bi se smanjilo korišćenje memorije, poželjno je uprostiti skup klauza F. Naime, sve klauze koje sadrže l se mogu ukloniti pošto su zadovoljene. Dalje, literal $\overline{l}$ se može ukloniti iz svih preostalih klauza. Međutim, pošto ovakva operacija uprošćavanja može da bude skupa, rešavači je ne primenjuju ovako često već samo s vremena na vreme.

9.9 Posmatrani literali

U nastavku će biti opisana tehnike koja omogućavaju efikasnu detekciju netačnih i jediničnih klauza.

9.9.1 Shema jednog posmatranog literala

Da bi se mogao implementirati u potpunosti funkcionalan rešavač, uslov $M \models \neg F$ je neophodno efektivno implementirati. Naivna implementacija koja izračunava istinitosnu vrednosti svih klauza formule F bi bila izrazito neefikasna. Klauza je netačna u valuaciji M ako i samo ako su svi njeni literali netačni u M. Ukoliko klauza c sadrži literal l koji nije netačan u M (tj. $M \not\models \neg l$), onda klauza c ne može biti netačna u M, bez obzira na istinitosne vrednosti njenih ostalih literala. Ovo svojstvo je osnova *sheme jednog posmatranog literala (eng. one-watch literal scheme)*. Ideja je u svakoj klauzi ,,posmatrati" njen proizvoljni literal koji nije netačan u M. Svaki put kada se neki literal, recimo l, doda u valuaciju M, da bi se proverilo da li važi $M \models \neg F$ dovoljno je proveriti da li su postale netačne samo one klauze kojima je baš $\overline{l}$ posmatrani literal (jer da je njihov posmatrani literal neki drugi literal l', on bi i dalje zadržao svojstvo da nije netačan u M). Pitanje koje ostaje je kako pronaći one klauze formule F kojima je $\overline{l}$ posmatran literal. Obilazak svih klauza i pregledanje njihovih posmatranih literala bi bilo neefikasno. Umesto toga, koriste se *liste posmatranih literala (eng. watch lists)* koje predstavljaju svojevrsni indeks u kome se čuvaju ovakve informacije. Lista posmatranog literala l ukazuje na sve one klauze kojima je literal l posmatran.

Opisana shema jednog posmatranog literala se ne koristi u modernim rešavačima. Umesto nje, koristi se *shema dva posmatrana literala (eng. two-watch literals scheme)* koja, pored provere da li je formula postala netačna u tekućoj parcijalnoj valuaciji, omogućava i pronalaženje svih jediničnih klauza formule u odnosu na tekuću valuaciju. Pominjanje sheme jednog posmatranog literala, urađeno je u cilju lakšeg razumevanja sheme dva posmatrana literala.

9.9.2 Shema dva posmatrana literala

Kako bi se efikasno implementirala propagacija jedničnih klauza, proveru uslova $\exists$`c.`$\exists$`l.  c` $\in F \wedge$ `(isUnit c l` M`)` je potrebno efikasno implementirati. Ni ovaj put naivna provera svake klauze formule F i utvrđivanje da li je ona postala jednična ne dolazi u obzir iz razloga efikasnosti. Umesto toga koristi se *shema dva posmatrana literala* koja efikasno pronalazi i netačne i jedinične klauze. Klauza c ne može biti jedinična u valuaciji ukoliko ili sadrži tačan literal ili sadrži dva literala koja nisu netačna u toj valuaciji. Iz tog razloga, shema dva posmatrana literala svakoj klauzi formule pridružuje njena dva različita literala koja se ,,posmatraju". Provera da li je klauza postala netačna ili jedinična se vrši samo u trenutku kada neki od njenih posmatranih literala postane netačan.

Struktura podataka koja je u osnovi sheme dva posmatrana literala predstavlja preslikavanje koje svakoj klauzi pridružuje njene posmatrane literale. Ipak, većina implementacija SAT rešavača ne gradi efektivno ovo preslikavanje već podrazumeva da se tip podataka `Clause` proširuje tako da postane slog koji sadrži kako listu literala koji čine klauzu tako i njene posmatrane literala. Još češće od ovoga se koristi konvencija da su posmatrani literali upravo prvi i drugi literal u listi literala koji čine klauzu. Bez obzira na njihovu stvarnu reprezentaciju, za označavanje posmatranih literala klauze c koristićemo oznake (`watch1` c) i (`watch2` c). Kao što je opisano u poglavlju §9.8, tekući skup klauza F sadrži samo klauze koje imaju bar dva literala što u mnogome olakšava implementaciju. Jedinične klauze i njihovi jednинični literali koje shema dva posmatrana literala detektuje, smeštaju se u *red jediničnih literala (eng. unit propagation queue)* Q, iz kojeg se uzimaju i prebacuju u tekuću parcijalnu valuaciju. Bulovska promenljiva $conflictFlag$ se koristi za čuvanje informacije o tome da li je formula postala netačna u tekućoj parcijalnoj valuaciji, tj. da li je došlo do konflikta.

Klauzama se pristupa samo kada neki od njihovih posmatranih literala postane netačan. Tada se ispituju njihovi ostali literali kako bi se utvrdilo da li je klauza postala jedinična, netačna ili (u međuvremenu) zadovoljena. Ukoliko ništa od ovoga nije slučaj, njeni posmatrani literali se ažuriraju. Kako bi se olakšala implementacija, ukoliko literal (`watch1` c) postane netačan, posmatrani literali se zamenjuju i moguće je, dakle, pretpostaviti da će netačan literal uvek biti baš (`watch2` c). Mogući su naredni slučajevi.

1. Ukoliko je moguće brzo i efikasno utvrditi da klauza sadrži tačan literal t, tada nema potrebe za promenom posmatranih literala. Objašnjenje za ovo je sledeće; da bi klauza postala jedinična ili netačna, literal t mora biti uklonjen iz M. Pošto je posmatrani literal koji je upravo postao netačan, to postao nakon što je t postao tačan, prilikom uklanjanja literala t iz M i posmatrani literal će biti nedefinisan i klauza će imati dva literala za koja

se zna da nisu netačni. Pitanje koje ostaje otvroreno je kako proveriti da li tačan literal t postoji. Učitavanje cele klauze iz glavne memorije i prolazak kroz sve njene literale se pokazuje kao neefikasno i vrši se samo ako brza heuristička provera ne uspe. Raniji rešavači su proveravali samo da li je literal (watch1 c) tačan u M. U velikom procentu slučajeva se ovo zaista i dešava čime se značajno smanjuje broj učitavanja klauza iz glavne memorije što je veoma skupa operacija. Kako bi se ovaj procenat još više povećao, noviji rešavači čuvaju i po nekoliko nasumice odabranih literala klauze i proveravaju da li je neki od njih tačan. Ovi literali se čuvaju u posebnim strukturama podataka za koje se očekuje da većinu vremena budu prisutne u keš memoriji, tako da i dalje nema potrebe za učitavanjem cele klauze iz glavne memorije.

2. Tek ukoliko brza provera nije utvrdila postojanje tačnog literala t, klauza se učitava iz glavne memorije i ispituju se svi njeni literali.

   a) Ukoliko postoji literal l koji nije posmatran literal klauze i nije netačan u M, on se proglažava za (watch2 c). U ovom slučaju literal (watch1 c) ne može da bude netačan u M. Zaista, ukoliko bi on bio netačan, u trenutku kada je postao netačan, bio bi zamenjen sa (watch2 c) (koji ne bi mogao da bude netačan iz istog razloga), i literal l bi tada već postao (watch2 c).

   b) Ukoliko su svi literali koji nisu posmatrani netačni u valuaciji M, a literal (watch1 c) nije definisan u M, onda je klauza upravo postala jedinična klauza i literal (watch1 c) se ubacuje u Q gde čeka na propagaciju. Posmatrani literali se u ovom slučaju ne menjaju. Objašnjenje za ovo je da se će nakon propagacije jediničnih klauza posmatrani literali biti poslednji literali klauze koji su postali definisani u M i da bi klauza opet postala jedinična ili netačna neophodno je da se neki literal povuku iz valuacije M u kom slučaju će posmatrani literali opet postati nedefinisani.

   c) Ukoliko su i svi literali koji nisu posmatrani i literal (watch1 c) netačni u M, cela klauza je postala netačna i zastavica $conflictFlag$ se podiže. Posmatrani literali se ne menjaju iz istog razloga kao u prethodnom slučaju.

Pod pretpostavkom da se propagacija jediničnih klauza izvršava iscrpno i da se pravilo Decide ne primenjuje nakon što se detektuje konflikt, nakon uklanjanja dela valuacije M tokom primene pravila Backjump u formuli F neće biti netačnih klauza i jedina jedinična klauza će biti upravo naučena klauza povratnog skoka. Ovo važno svojstvo sheme dva posmatrana literala obezbeđuje da prilikom primene pravila Backjump nema potrebe za ažuriranjem posmatranih literala i ostalih struktura podataka vezanih za ovu sheme što omogućava primenu pravila Backjump u konstantnom vremenu.

U narednoj tabeli predstavljen je efekat funkcije notifyWatches. Slova T, F i U redom označavaju tačne, netačne i nedefinisane literale.

before assert(M)			after assert(M')			after notify(M')			$effect$
(w1 c)	(w2 c)	$other$	(w1 c)	(w1 c)	$other$	(w1 c)	(w2 c)	$other$	
T	U	U/T	T	F	U/T	T	F	U/T	
U	T	U/T	F	T	U/T	T	F	U/T	swap
U	U	U/T	U	F	U/T	U	U/T	?	(w2 c) := $other$
U	U	U/T	F	U	U/T	U	U/T	?	swap, (w2 c) := $other$
T	U	F	T	F	F	T	F	F	
U	T	F	F	T	F	T	F	F	swap
U	U	F	U	F	F	U	F	F	$Q := Q$ @ (w1 c)
U	U	F	F	U	F	U	F	F	swap, $Q := Q$ @ (w1 c)
U	F	F	F	F	F	F	F	F	swap, $conflictFlag := true$

Upravo opisani proces čini suštinu procedure notifyWatches date u kodu koji sledi.

SAT Solver v.9

```
...
Q : Literal list                                                    +
conflictFlag : Boolean                                              +
```

```
function addClause(clause : Clause)
begin
...
  else
    setWatch1(clause, (first clause));                              +
    setWatch2(clause, (second clause));                             +
    F := F @ clause
end
```

```
function setWatch1(clause : Clause, l : Literal)                    +
begin                                                               +
  (watch1 clause) := l                                              +
end                                                                 +
```

```
function setWatch2(clause : Clause, l : Literal)                    +
begin                                                               +
  (watch2 clause) := l                                              +
end                                                                 +
```

```
function assertLiteral(l : Literal, decision : Boolean)
begin
  M := M @ [(l, decision)];
  notifyWatches(¬l)                                                 +
end
```

```
function notifyWatches (l : Literal)                                    +
begin                                                                   +
  foreach clause : clause ∈ F ∧                                         +
          ((watch1 clause) = l ∨ (watch2 clause) = l) do                +
    if (watch1 clause) = l then                                         +
      swapWatches(clause);                                              +
    if M ⊭ (watch1 clause) then                                         +
      if (∃l'. isNonWatchedUnfalisfiedLiteral(l', clause)) then         +
        setWatch2(clause, l')                                           +
      else if M ⊨¬ (watch1 clause) then                                 +
        conflictFlag := true;                                           +
        conflictClause := clause                                        +
      else                                                              +
        if (watch1 clause) ∉ Q then                                     +
          Q := Q @ (watch1 clause)                                      +
end                                                                     +
```

```
const function isNonWatchedUnfalsifiedLiteral(l : Literal, clause
: Clause)                                                               +
begin                                                                   +
  ret := l ∈ clause ∧                                                   +
       l ≠ (watch1 clause) ∧ l ≠ (watch2 clause) ∧                      +
       M ⊭¬ l                                                           +
end                                                                     +
```

```
function buildC()
begin
  C := C_P @ C̄_l
  if C_P ≠ [] then                                                      +
    setWatch1(C, C̄_l);                                                  +
    setWatch2(C, (lastAssertedLiteral C̄_P M))                          +
end
```

```
function applyUnitPropagate() : Boolean
begin
  if Q = [] then                                                        *
    ret := False                                                        *
  else                                                                  *
```

```
    assertLiteral ((head Q), false);                    *
    Q := (tail Q);                                      *
    ret := True                                         *
end
```

```
function applyBackjump()
begin
  level := getBackjumpLevel();
  M := (prefixToLevel M level);
  conflictFlag := False;
  Q := [C̄_l]                                            +
end
```

```
function isReason(c : Clause) : Boolean
begin                                                   *
  if getReason((watch1 c)) = c then                     *
    ret := true                                         *
  else                                                  *
    ret := false                                        *
end
```

Neke implementacije posmatraju red jediničinih literala Q kao neprocesirani deo valuacije M. Ova modifikacija dovodi do nešto bržeg otkrivanja konflikata. U datom kodu ona bi se implementirala tako što bi se svi uslovi oblika $M \models$ i $M \models\neg$ u kodu SAT rešavača v.9 redom zamenili uslovima oblika $M \mathbin{@} Q \models$ i $M \mathbin{@} Q \models\neg$.

Liste klauza za posmatrane literale

Kako bi se efikasno implementiralo pronalaženje klauza kojima je dati literal posmatran, koriste se liste klauza za posmatrane literale (eng. watch lists).

SAT Solver v.10

```
function setWatch1(clause : Clause, l : Literal)
begin
  (watch1 clause) := l
  W(l) := W(l) @ clause                                 +
end
```

```
function setWatch2(clause : Clause, l : Literal)
begin
  (watch2 clause) := l
  W(l) := W(l) @ clause                                 +
end
```

```
function notifyWatches (l :  Literal)
begin
  newWL := [];                                                    +
  foreach clause :  clause ∈ W(l)                                 *
    if (watch1 clause) = l then
      swapWatches(clause);
    if M ⊨ (watch1 clause) then                                   +
      newWL := newWL @ clause                                     +
    else
      if (∃l'.  isNonWatchedUnfalisfiedLiteral(l', clause)) then
        setWatch2(clause, l')
      else if M ⊨¬ (watch1 clause) then
        conflictFlag := true;
        conflictClause := clause;
        newWL := newWL @ clause                                   +
      else
        Q := Q @ (watch1 clause);
        newWL := newWL @ clause                                   +
  W(l) := newWL                                                   +
end
```

```
function removeClause(c :  Clause)                                +
begin                                                             +
  W((watch1 c)) := W((watch1 c)) \ c;                             +
  W((watch2 c)) := W((watch2 c)) \ c                              +
end                                                               +
```

U trenutku kada literal l prestaje da bude posmatrani literal neke klauze c, njegova lista $W(l)$ mora biti ažurirana i klauza c mora biti uklonjena iz nje. Tokom jednog poziva funkcije notifyWatches, vrši se obilazak ove liste i za očekivati je da će tom prilikom biti potrebno izvršiti veliki broj takvih uklanjanja. Pokazuje se da je efikasnije iznova izgraditi listu $W(l)$ za literal l nego nad njom više puta vršiti operaciju ažuriranja. Ovo je upravo uloga liste `newWl` i klauze za koje se posmatrani literal nije promenio se u nju umeću. Ovo možda deluje kao neobično rešenje, ali je u stvari veoma važna tehnika. Pošto predstavlja „srce" operacije propagacije, funkcija notifyWatches od krucijalnog značaja za ukupnu efikasnost rešavača, pošto njeno izvršavanje može da oduzme i do 80% ukupnog vremena rešavanja [ES04].

9.10 Dokazivanje korektnosti imperativne implementacije

Verifikacija korektnosti programa izraženih u imperativnim programskim jezicima se obično vrši u okviru Flojd-Horove (skr. Horove) logike [Hoa69, Flo67]. To je formalni sistem koji obezbeđuje skup logičkih pravila koja se koriste za rezonovanje o korektnosti računarskih programa sa strogošću matematičke logike. Centralni objekat Horove logike čini *Horova tojka* koja opisuje kako izvršavanje dela koda menja stanje izračunavanja. Horova trojka je oblika $\{P\}$

`code` $\{Q\}$, gde su P (*preduslov (eng. precondition)*) i Q (*pauslov (eng. post-condition)*) logičke formule, a `code` je deo koda izražen u programskom jeziku. Horovu trojku treba tumačiti kao: „ako uslov P važi pre nego što se programski kôd `code` izvrši i ako se izvršavanje kôd `code` zaustavi, uslov Q će važiti neposredno nakon što se kôd `code` izvrši". Horove trojke se grade i kombinuju na osnovu pravila formulisanih za svaki konstrukt korišćenog programskog jezika. Na primer, pravilo koje opisuje semantiku `while` naredbe je:

$$\frac{\{P \wedge B\}\ \ S\ \ \{P\}}{\{P\}\ \ \texttt{while}\ B\ \texttt{do}\ S\ \ \{\neg B \wedge P\}}$$

Krajnji domet verifikacije u Horovom stilu bi bio dokazati korektnost realne C/C++ implementacije SAT rešavača. Ipak, pošto bi ovo bio veoma kompleksan poduhvat, na ovaj način, izvršena je samo verifikacija jezgra SAT rešavača koje je opisano u prethodnom delu ove glave.

I ovaj put su korišćene globalne invarijante navedene u poglavlju 7. Dodatno, za svaku funkciju u kôdu su definisani preduslovi i pauslovi koje bi ona trebalo da zadovoljava i ovi uslovi su povezani kroz Horove trojke. Tako je, na primer, u okviru rešavača *SAT solver v.9* preduslov za funkciju applyUnitPropagate uslov da su svi literali koji se nalaze u Q jedinični literali za klauze iz F u odnosu na valuaciju M. Naredna Horova trojka govori da je ovaj uslov održan i nakon primene funkcije applyUnitPropagate.

$$\begin{gathered}\{\forall l.\ l \in Q \longrightarrow \exists\ c.\ c \in F\ \ \wedge\ \ \mathsf{isUnit}\ c\ l\ M\} \\ \mathsf{applyUnitPropagate}() \\ \{\forall l.\ l \in Q \longrightarrow \exists\ c.\ c \in F\ \ \wedge\ \ \mathsf{isUnit}\ c\ l\ M\}\end{gathered}$$

Primetimo da celokupna implementacija navedena u okviru ove glave ostavlja heurističke komponente neimplementiranim već samo navodi Horove trojke koje bi implementacija trebalo da zadovolji. Na primer, kroz sve verzije rešavača, funkcija selectLiteral je navedena samo putem Horove trojke

$$\{\mathsf{vars}\ M \neq \mathsf{vars}\ F_0\}\ \ \mathsf{selectLiteral}()\ \ \{\mathsf{var}\ ret \in \mathsf{vars}\ F_0 \wedge \mathsf{var}\ ret \notin \mathsf{vars}\ M\}$$

Ovim je ostavljen prostor za korišćenje različitih heuristika u okviru istog DPLL jezgra, što čini rešavač znatno fleksibilnijim. Jedino što je potrebno dokazati prilikom zadavanja konkretne implementacije ovakvih funkcija je da zadovoljavaju odgovarajuću navedenu trojku.

Kada su formulisane odgovarajući preduslovi i pauslovi, oni su ručno spajani na osnovu pravila Horove logike i dobijeni su uslovi korektnosti koji su zatim formalno dokazani. Verifikacija koja je rađena se bavila samo problemom parcijalne korektnosti koda, dok problem zaustavljanja nije razmatran. Ključna teorema parcijalne korektnosti opisuje uslove za funkciju solve koja je ulazna tačka rešavača i koja, ako se zaustavi, postavlja vrednost promenljive $satFlag$ (na SAT ili na $UNSAT$) i vraća je pozivaocu. Osnovni rezultat verifikacije je naredna teorema (dokazana za sve izložene verzije rešavača).

Teorema 9.1 (Parcijalna korektnost). *SAT rešavač zadovoljava narednu Horovu trojku:*
$\{\top\}\ \ \mathsf{solve}\,(F_0)\ \ \{(satFlag = UNSAT \wedge \neg\mathsf{sat}\ F_0) \vee (satFlag = SAT \wedge M \vDash F_0)\}$

10 Implementacija u logici višeg reda

Kako bi se olakšao proces rezonovanja o implementaciji SAT rešavača i povećao stepen pouzdanosti verifikacije, u ovom poglavlju će biti opisan postupak implementacije SAT rešavača direktno u okviru logike višeg reda (koja se za ovu priliku tretira kao čist funkcionalni programski jezik). Ova tehnika je poznata pod imenog *plitko utapanje u logiku višeg reda (eng. shallow embedding to HOL)* [BKH+08]. Korišćenjem ove tehnike, SAT rešavač se predstavlja u obliku funkcija uglavnom definisanih korišćenjem jednakosti i primitivne i totalne rekurzije. Ovo omogućava rezonovanje o svojstvima programa tj. njegovu verifikaciju korišćenjem preciznog matematičkog aparata koji uglavnom uključuje jednakosno rezonovanje i matematičku indukciju. Nije potebno koristiti bilo kakvu pomoćnu logiku (npr. Horovu) niti model denotacione ili operacione semantike programskog jezika. Na osnovu logičkih definicija, u okviru sistema Isabelle, moguće je automatski ekstrahovati izvršni kôd na nekom od poznatijih funkcionalnih jezika (Haskell, SML, OCaml), za koji se sa izuzetno velikim stepenom pouzdanosti može tvrditi da je korektan [Haf08]. Ovaj način verifikacije biće sproveden za klasičnu DPLL proceduru kao i za savremeni SAT rešavač.

10.1 Klasična DPLL procedura

U ovoj glavi će biti u okviru logike višeg reda biti definisana klasična DPLL procedura koja je neformalno opisana u poglavlju 4.3.1. Ovako definisana procedura nije efikasna i ne koristi se u savremenim rešavačima, ali istorijski zauzima veoma značajno mesto i zbog toga je ovde navodimo. Implementacija koja će biti prikazana koristi pojmove meta-teorije uvedene u glavama 5 i 6. Procedura će biti definisana kao rekurzivna funkcija i to na način koji se može sresti u većini klasičnih udžbenika logike. Takođe, biće dokazana i totalna korektnost ovako definisane procedure, pri čemu naš formalni dokaz u grubim crtama sledi neformalni dokaz dat u [Dav94]. Celokupna implementacija i verifikacija je urađena u okviru sistema Isabelle, međutim, kako bi materijal mogao da bude razumljiv i čitaocima koji ne poznaju sistem Isabelle definicija je prikazana u neformalnom, matematičkom jeziku, a u posebnom poglavlju su prikažu samo fragmenti Isabelle koda.

10.1.1 Zamena literala

Jedna od osnovnih operacija klasične DPLL procedure je zamena literala logičkim konstantama $\top$ i $\bot$ i uprošćavanje tako dobijene formule. Sledeća definicija formalno uvodi ovu operaciju.

Definicija 10.1. Zamena literala l konstantom $\top$ u formuli F, *što označavamo sa* $F[l \to \top]$, *je formula koja se dobija od formule* F *brisanjem svih klauza koje sadrže* l *i brisanjem svih poljavljivanje literala* $\overline{l}$. *Definišemo je korišćenjem primitivne rekurzije:*

$$[\,][l \to \top] = [\,]$$

$$\begin{aligned}(c \mathbin{@} F)[l \to \top] = \ & \textit{if } l \in c \textit{ then} \\ & \quad F[l \to \top] \\ & \textit{else if } \overline{l} \in c \textit{ then} \\ & \quad (c \setminus \overline{l}) \mathbin{@} F[l \to \top] \\ & \textit{else} \\ & \quad c \mathbin{@} F[l \to \top]\end{aligned}$$

Zamena literala l konstantom $\bot$ u formuli F, *što označavamo sa* $F[l \to \bot]$, *sintaksno je jednaka formuli* $F[\overline{l} \to \top]$.

Neka od osnovnih svojstava ove operacije su navedena u narednom stavu koji navodimo bez dokaza.

Stav 10.1.

(1)	$(\mathsf{var}\ l) \notin F[l \to \top]$
(2)	$(\mathsf{var}\ l) \notin (\mathsf{var}\ F) \implies F[l \to \top] = F$
(3)	$l \notin F \wedge \overline{l} \notin F \implies F[l \to \top] = F[l \to \bot] = F$
(4)	$(\mathsf{model}\ v\ F) \wedge\ l \in v \implies (\mathsf{model}\ v\ F[l \to \top])$
(5)	$(\mathsf{model}\ v\ F) \wedge\ (\mathsf{var}\ l) \notin (\mathsf{vars}\ v) \implies v \vDash F[l \to \top]$
(6)	$v \vDash F[l \to \top] \implies (l \mathbin{@} v) \vDash F$

Naredna važna lema, poznata kao lema o split pravilu, pokazuje da se korišćenjem operacije zamene, ispitivanje zadovoljivosti formule može svesti na ispitivanje zadovoljivosti dve manje formule. Pošto je ovo fundamentalna lema u dokazu korektnosti klasičnog DPLL algoritma, navešćemo i skicu njenog formalnog dokaza.

Lema 10.1.

$$(\mathsf{sat}\ F) \iff (\mathsf{sat}\ F[l \to \top]) \ \vee\ (\mathsf{sat}\ F[l \to \bot])$$

Dokaz. ($\Rightarrow$) : Pretpostavimo da je formula F zadovoljiva, odnosno da važi $(\mathsf{sat}\ F)$. Ovo znači da postoji valuacija v takva da važi $(\mathsf{model}\ v\ F)$, tj. $(\mathsf{consistent}\ v)$ i $v \vDash F$. Razmotrimo sledeća dva slučaja:

1. $(\mathsf{var}\ l) \notin (\mathsf{vars}\ v)$: na osnovu stava 10.1, tačka (5), iz činjenice

$$(\mathsf{model}\ v\ F)\ \wedge\ (\mathsf{var}\ l) \notin (\mathsf{vars}\ v) \implies (\mathsf{model}\ v\ F[l \to \top])$$

sledi da je $F[l \to \top]$ zadovoljiva, odnosno $(\mathsf{sat}\ F[l \to \top])$.

2. $(\mathsf{var}\ l) \in (\mathsf{vars}\ v)$: na osnovu stava 6.3, iz činjenice

$$(\mathsf{var}\ l) \in (\mathsf{vars}\ v) \iff v \vDash l \vee v \vDash \neg l$$

sledi da važi $v \vDash l$ ili $v \vDash \neg l$.

a) U slučaju $v \vDash l$, na osnovu stava 10.1, tačka (4), iz činjenice

$$(\mathsf{model}\ v\ F)\ \wedge\ l \in v \implies (\mathsf{model}\ v\ F[l \to \top])$$

sledi da važi $(\mathsf{sat}\ F[l \to \top])$.

b) U slučaju $v \vDash \overline{l}$, na osnovu stava 10.1, tačka (4), iz činjenice

$$(\mathsf{model}\ v\ F)\ \wedge\ \overline{l} \in v \implies (\mathsf{model}\ v\ F[\overline{l} \to \top])$$

sledi da važi $(\mathsf{sat}\ F[l \to \bot])$.

($\Leftarrow$) : Pretpostavimo da važi $(\mathsf{sat}\ F[l \to \top])$. Ovo znači da postoji valuacija v takva da važi $(\mathsf{model}\ v\ F)$, tj. $(\mathsf{consistent}\ v)$ i $v \vDash F[l \to \top]$. Na osnovu stava 10.1, tačka (1) važi $var(l) \notin F[l \to \top]$. Na osnovu stava 6.7, tj. svojstva

$$(\mathsf{model}\ v\ F)\ \wedge\ (\mathsf{var}\ l) \notin (\mathsf{vars}\ F) \implies$$
$$\exists v'.\ ((\mathsf{model}\ v'\ F)\ \wedge\ (\mathsf{var}\ l) \notin (\mathsf{vars}\ v'))$$

instanciranog formulom $F[l \to \top]$ i valuacijom v, sledi da postoji valuacija v' takva da važi $(\mathsf{consistent}\ v')$, $v' \vDash F[l \to \top]$ i $(\mathsf{var}\ l) \notin (\mathsf{vars}\ v')$. Onda, na osnovu stava 10.1, tačka (6), iz činjenice

$$v' \vDash F[l \to \top] \implies (l\ @\ v') \vDash F$$

sledi $(l\ @\ v') \vDash F$. Pošto važi $(\mathsf{var}\ l) \notin (\mathsf{vars}\ v')$, na osnovu stava 6.3, sledi da $\overline{l} \notin v'$. Pošto važi $(\mathsf{consistent}\ v')$, na osnovu stava 6.5, važi i $(\mathsf{consistent}\ (l\ @\ v'))$. Konačno, važi i $(\mathsf{sat}\ F)$ pošto je $l\ @\ v'$ model formule F.

Pretpostavimo da važi $(\mathsf{sat}\ F[l \to \bot])$. Dokaz da u tom slučaju važi $(\mathsf{sat}\ F)$ je potpuno analogan prethodnom slučaju, ako se umesto literala l koristi literal $\overline{l}$.

□

Na osnovu prethodne leme moguće je definisati naivnu proceduru za ispitivanje zadovoljivosti formule. Procedura bira proizvoljan literal l (korišćenjem funkcije $\mathsf{selectLiteral}$), a zatim rekurzivno ispituje zadovoljivost formula $F[l \to \top]$ i $F[l \to \bot]$. Uslov za izlaz iz rekurzije su prazna formula, odnosno formula koja sadrži praznu klauzu.

Definicija 10.2.

$(\mathsf{search}\ F) \iff$
if $F = [\,]$ *then*
$\top$
else if $[\,] \in F$ *then*
$\bot$
else
$(\mathsf{search}\ F[(\mathsf{selectLiteral}\ F) \to \top])\ \vee\ (\mathsf{search}\ F[(\mathsf{selectLiteral}\ F) \to \bot])$

U nekim slučajevima nije neophodno ispitivati zadovoljivost obe formule iz prethodne leme. Ukoliko se ustanovi da je prvi disjunkt zadovoljen, nema potrebe za proverom drugog. Takođe, u nekim specijalnim slučajevima koje navodimo u nastavku, ispitivanje jednog od disjunkata nije neophodno vršiti. Naredne optimizacije se smatraju sastavnim delom klasične DPLL procedure i razlikuju je u odnosu na upravo definisani osnovni algoritam pretrage.

10.1.2 Jedinične klauze (Unit clauses)

Jedna vrsta optimizacije naivne procedure za ispitivanje zadovoljivosti se zasniva na korišćenju *jediničnih klauza (eng. unit clauses)*. Obratimo pažnju da, pošto klasični DPLL algoritam ne koristi koncept označene valuacije, definicija pojma jediničnih klauza je različita od definicije jedinične klauze u valuaciji 7.20.

Definicija 10.3. *Klauza c je* jedinična klauza (eng. unit clause) *ako i samo ako ima tačno jedan literal, tj. $c = [l]$. Tada kažemo da je l* jedinični literal (unit literal).

Naredna lema pokazuje da se, kada formula sadrži jediničnu klauzu, provera njene zadovoljivosti može svesti na ispitivanje zadovoljivosti samo jedne manje formule. Kako važi da $[l] \in F \implies \neg(\mathsf{sat}\ F[l \to \bot])$, naredna lema je direktna posledica leme o split pravilu (Lema 10.1).

Lema 10.2.

$$[l] \in F \implies ((\mathsf{sat}\ F) \iff (\mathsf{sat}\ F[l \to \top]))$$

10.1.3 Literali jedinstvenog polariteta (Pure literals)

Naredna vrsta optimizacije naivne procedure zasnovane na iscrpnoj primeni split pravila se zasniva na literalima koji se u formuli pojavljuju samo u *jednom polaritetu (pure literals*).

Definicija 10.4. *Literal l je* literal jedinistvenog polariteta *u formuli F ako i samo ako važi $l \in F$ i $\overline{l} \notin F$.*

Ako važi $\overline{l} \notin F$, formula $F[l \to \top]$ sadrži tačno klauze formule F koje ne sadrže l. Te klauze takođe pripadaju i formuli $F[l \to \bot]$. Dakle, važi:

$$l \in F \wedge \overline{l} \notin F \implies F[l \to \top] \subseteq F[l \to \bot]$$

Pošto važi $F[l \to \top] \subseteq F[l \to \bot] \implies ((\mathsf{sat}\ F[l \to \bot]) \implies (\mathsf{sat}\ F[l \to \top]))$, naredna lema je posledica navedenog tvrđenja i leme o split pravilu (Lema 10.1).

Lema 10.3.

$$l \in F \wedge \overline{l} \notin F \implies ((\mathsf{sat}\ F) \iff (\mathsf{sat}\ F[l \to \top]))$$

10.1.4 Definicija DPLL procedure

Korišćenjem uvedenih pojmova, možemo formulisati formalnu definiciju klasičnog DPLL algoritma.

Definicija 10.5.
$(\mathsf{dpll}\ F) \iff$
if $F = [\,]$ *then*
$\top$
else if $[\,] \in F$ *then*
$\bot$
else if $(\mathsf{hasPureLiteral}\ F)$ *then*
$(\mathsf{dpll}\ F[(\mathsf{getPureLiteral}\ F) \to \top])$
else if $(\mathsf{hasUnitLiteral}\ F)$ *then*
$(DPLL\ F[(\mathsf{getUnitLiteral}\ F) \to \top])$
else if $(\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \top])$ *then*
$\top$
else $(\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \bot])$

Primetimo da je neophodno efektivno definisati funkcije hasUnitLiteral, getUnitLiteral, hasPureLiteral, getPureLiteral i selectLiteral kako bi se dobila efektivna implementacija DPLL procedure. Ovo je moguće uraditi na mnogo različitih načina. Izbor implementacije može značajno da utiče na efikasnost procedure, ali nema uticaja na njenu korektnost sve dok zadovoljavaju uslove naredne specifikacije:

- $(\mathsf{hasUnitLiteral}\ F) \implies [(\mathsf{getUnitLiteral}\ F)] \in F$
- $(\mathsf{hasPureLiteral}\ F) \implies (\mathsf{getPureLiteral}\ F) \in F \ \wedge \overline{(\mathsf{getPureLiteral}\ F)} \notin F$
- $F \neq [\,] \wedge [\,] \notin F \implies (\mathsf{selectLiteral}\ F) \in F$

10.1.5 Zaustavljanje klasične DPLL procedure

Potrebno je pokazati da je prethodnom definicijom dobro definisana jedna totalna funkcija, odnosno dokazati da se klasični DPLL algoritam, dat kroz definiciju 10.5, zaustavlja za svaku formulu F. U ovom cilju, pokazaćemo da se ukupan broj literala u formuli smanjuje svakim rekurzivnim pozivom.

Definicija 10.6. $(\mathsf{numLiterals}\ F)$ *je ukupan broj literala u svim klauzama formule* F. *On se primitivnom rekurzijom definiše na sledeći način.*

$$\begin{aligned} (\mathsf{numLiterals}\ [\,]) &= 0 \\ (\mathsf{numLiterals}\ (c\ @\ F)) &= |c| + (\mathsf{numLiterals}\ F) \end{aligned}$$

Naredni stav koji navodimo bez dokaza daje svojstva koja odgovaraju svim rekurzivnim pozivima i osiguravaju zaustavljanje DPLL algoritma:

Stav 10.2.

- $l \in F \implies (\mathsf{numLiterals}\ F[l \to \top]) < (\mathsf{numLiterals}\ F)$
- $l \in F \implies (\mathsf{numLiterals}\ F[l \to \bot]) < (\mathsf{numLiterals}\ F)$
- $F \neq [] \wedge [] \notin F \implies$
  $(\mathsf{numLiterals}\ F[(\mathsf{selectLiteral}\ F) \to \top]) < (\mathsf{numLiterals}\ F)$
- $F \neq [] \wedge [] \notin F \implies$
  $(\mathsf{numLiterals}\ F[(\mathsf{selectLiteral}\ F) \to \bot]) < (\mathsf{numLiterals}\ F)$
- $(\mathsf{hasUnitLiteral}\ F) \implies$
  $(\mathsf{numLiterals}\ F[(\mathsf{getUnitLiteral}\ F) \to \top]) < (\mathsf{numLiterals}\ F)$
- $(\mathsf{hasPureLiteral}\ F) \implies$
  $(\mathsf{numLiterals}\ F[(\mathsf{getPureLiteral}\ F) \to \top]) < (\mathsf{numLiterals}\ F)$

Teorema 10.1. *Definicija 10.5 uvodi dobro definisanu totalnu funkciju.*

Dokaz. Direktno na osnovu stava 10.2. □

10.1.6 Korektnost klasične DPLL procedure

Konačno, dokazujemo i parcijalnu korektnost date implementacije.

Teorema 10.2.

$$(\mathsf{dpll}\ F) \iff (\mathsf{sat}\ F)$$

Dokaz. Dokaz sprovodimo indukcijom. Bazu indukcije predstavljaju slučajeve u kojima se ne vrši rekurzivni poziv. Postoje dve takve grane u kodu

- Ako je $F = []$ tada je $(\mathsf{dpll}\ F) = \top$ i $(\mathsf{sat}\ F) = \top$, na osnovu stava 6.7.
- Ako je $F \neq []$ i $[] \in F$, tada je $(\mathsf{dpll}\ F) = \bot$ and $(\mathsf{sat}\ F) = \bot$, na osnovu stava 6.7.

Sada pretpostavimo da tvrđenje teoreme važi za svaki rekurzivni poziv i na osnovu toga pokažimo da tvrđenje važi i za samu proceduru, odnosno, pretpostavimo da važi sledeća induktivna hipoteza:

$$(\mathsf{hasPureLiteral}\ F) \wedge [] \notin F \wedge F \neq [] \implies (\mathsf{dpll}\ F[(\mathsf{getPureLiteral}\ F) \to \top]) \iff (\mathsf{sat}\ F[(\mathsf{getPureLiteral}\ F) \to \top])$$

$$(\mathsf{hasUnitLiteral}\ F) \wedge \neg(\mathsf{hasPureLiteral}\ F) \wedge [] \notin F \wedge F \neq [] \implies (\mathsf{dpll}\ F[(\mathsf{getUnitLiteral}\ F) \to \top]) \iff (\mathsf{sat}\ F[(\mathsf{getUnitLiteral}\ F) \to \top])$$

$$\neg(\mathsf{hasUnitLiteral}\ F) \wedge \neg(\mathsf{hasPureLiteral}\ F) \wedge [] \notin F \wedge F \neq [] \implies (\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \top]) \iff (\mathsf{sat}\ F[(\mathsf{selectLiteral}\ F) \to \top])$$

$$\neg(\mathsf{hasUnitLiteral}\ F) \wedge \neg(\mathsf{hasPureLiteral}\ F) \wedge [] \notin F \wedge F \neq [] \wedge \neg(\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \top]) \implies (\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \bot]) \iff (\mathsf{sat}\ F[(\mathsf{selectLiteral}\ F) \to \bot])$$

Razmotrimo *if–then–else* grane u definiciji dpll funkcije:

- Ako je $F \neq [\,]$ i $[\,] \notin F$ i $(\mathsf{hasPureLiteral}\ F)$, tada na osnovu definicije dpll funkcije važi:

  $$(\mathsf{dpll}\ F) \iff (\mathsf{dpll}\ F[(\mathsf{getPureLiteral}\ F) \to \top])$$

  Takođe, na osnovu induktivne hipoteze važi:

  $$(\mathsf{dpll}\ F[(\mathsf{getPureLiteral}\ F) \to \top]) \iff (\mathsf{sat}\ F[(\mathsf{getPureLiteral}\ F) \to \top])$$

  Na osnovu specifikacije funkcije getPureLiteral i pretpostavke ($\mathsf{hasPureLiteral}F$), važi $(\mathsf{getPureLiteral}\ F) \in F \ \wedge\ \overline{(\mathsf{getPureLiteral}\ F)} \notin F$. Tada, na osnovu leme 10.3, važi:

  $$(\mathsf{sat}\ F[(\mathsf{getPureLiteral}\ F) \to \top]) \iff (\mathsf{sat}\ F).$$

  Dakle, važi $(\mathsf{dpll}\ F) \iff (\mathsf{sat}\ F)$.

- Ako je $F \neq [\,]$, $[\,] \notin F$, $\neg(\mathsf{hasPureLiteral}\ F)$ i $(\mathsf{hasUnitLiteral}\ F)$, tada na osnovu definicije dpll funkcije važi:

  $$(\mathsf{dpll}\ F) \iff (\mathsf{dpll}\ F[(\mathsf{getUnitLiteral}\ F) \to \top])$$

  Takođe, na osnovu induktivne hipoteze važi:

  $$(\mathsf{dpll}\ F[(\mathsf{getUnitLiteral}\ F) \to \top]) \iff (\mathsf{sat}\ F[(\mathsf{getUnitLiteral}\ F) \to \top])$$

  Na osnovu specifikacije funkcije getUnitLiteral i pretpostavke $(\mathsf{hasUnitLiteral}\ F)$, važi $[(\mathsf{getUnitLiteral}\ F)] \in F$. Tada, na osnovu leme 10.2, važi:

  $$(\mathsf{sat}\ F[(\mathsf{getUnitLiteral}\ F) \to \top]) \iff (\mathsf{sat}\ F)$$

  Dakle, važi $(\mathsf{dpll}\ F) \iff (\mathsf{sat}\ F)$.

- Pretpostavimo da važi $F \neq [\,]$, $[\,] \notin F$, $\neg(\mathsf{hasPureLiteral}\ F)$ i $\neg(\mathsf{hasUnitLiteral}\ F)$. Na osnovu definicije dpll funkcije i definicije $if - then - else$ veznika važi:

  $$(\mathsf{dpll}\ F) \iff (\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \top]) \Rightarrow \top \ \wedge$$
  $$\neg(\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \top]) \Rightarrow (\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \bot])$$

  Dakle, važi

  $$(\mathsf{dpll}\ F) \iff (\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \top]) \vee (\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \bot]).$$

  Ukoliko važi $(\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \top])$, tada na osnovu induktivne hipoteze važi $(\mathsf{sat}\ F[(\mathsf{selectLiteral}\ F) \to \top])$. U suprotnom, ako važi $\neg(\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \top])$ i $(\mathsf{dpll}\ F[(\mathsf{selectLiteral}\ F) \to \bot])$, tada na osnovu induktivne hipoteze važi $(\mathsf{sat}\ F[(\mathsf{selectLiteral}\ F) \to \bot])$. Dakle, važi:

  $$(\mathsf{dpll}\ F) \iff (\mathsf{sat}\ F[(\mathsf{selectLiteral}\ F) \to \top]) \vee (\mathsf{sat}\ F[(\mathsf{selectLiteral}\ F) \to \bot]).$$

  Na osnovu leme 10.1 važi:

  $$(\mathsf{sat}\ F) \iff (\mathsf{sat}\ F[(\mathsf{selectLiteral}\ F) \to \top]) \vee (\mathsf{sat}\ F[(\mathsf{selectLiteral}\ F) \to \bot])$$

  Dakle, važi $(\mathsf{dpll}\ F) \iff (\mathsf{sat}\ F)$.

  □

10.1.7 Formalizacija u okviru sistema Isabelle

U nastavku će biti prikazano nekoliko fragmenata Isabelle koda koji odgovaraju upravo uvedenim definicijama, tvrđenjima i dokazima.

Klasična DPLL procedura uvedena definicijom 10.5 se u sistemu Isabelle može definisati na sledeći način:

```
function dpll::"Formula => bool"
where
"(dpll formula) =
 (if (formula = []) then
       True
  else if ([] el formula) then
       False
  else if (hasPureLiteral formula) then
      (dpll (setLiteralTrue
                (getPureLiteral formula) formula))
  else if (hasUnitLiteral formula) then
      (dpll (setLiteralTrue
                (getUnitLiteral formula) formula))
  else if (dpll (setLiteralTrue
                    (selectLiteral formula) formula)) then
       True
  else
      (dpll (setLiteralTrue
                (opposite (selectLiteral formula)) formula))
 )"
by pat_completenesss auto
termination
by (relation "measure (% formula. (numLiterals formula))")
   (auto simp add: dpllTermination_1 dpllTermination_2
                   dpllTermination_3 dpllTermination_4)
```

Teorema 10.2 i njen dokaz su formalizovani na sledeći način:

```
lemma dpllCorrectness: "(dpll F) = (satisfiable F)"
proof (induct F rule: dpll.induct)
  case (inductiveStep formula)
  note inductive_hypothesis = this
  show ?case
  proof (cases "formula = []")
    case True
    thus ?thesis
      by (simp add:emptyFormulaIsSatisfiable)
  next
    case False
    show ?thesis
    proof (cases "[] el formula")
      case True
      with `formula ~= []` show ?thesis
        by (simp add:formulaWithEmptyClauseIsUnsatisfiable)
```

```
    next
      case False
      show ?thesis
      proof (cases "hasPureLiteral formula")
        case True
        let ?pl = "getPureLiteral formula"
        hence "?pl el formula" and "~opposite ?pl el formula"
          by (auto simp add: getPureLiteralIsPure)
        with `formula ~= []` `~[] el formula`
             `hasPureLiteral formula`
             inductive_hypothesis
             pureLiteraRule [of "?pl" "formula"]
          show ?thesis
          by auto
      next
        case False
        show ?thesis
        proof (cases "hasUnitLiteral formula")
          case True
          let ?ul = "getUnitLiteral formula"
          hence "[?ul] el formula"
            by (simp add: getUnitLiteralIsUnit)
          with `formula ~= []` `~[] el formula`
               `~hasPureLiteral formula` `hasUnitLiteral formula`
               inductive_hypothesis
               unitLiteralRule [of "?ul" "formula"]
          show ?thesis
            by auto
        next
          case False
          with `formula ~= []` `~[] el formula`
               `~hasPureLiteral formula` `~hasUnitLiteral formula`
               inductive_hypothesis
          show ?thesis
            using split_rule[of "formula" "selectLiteral formula"]
            by auto
        qed
      qed
    qed
  qed
qed
```

10.2 Savremeni SAT rešavač

U ovom poglavlju će u okviru logike višeg reda biti definisan moderan SAT rešavač. Kao i u slučaju klasične DPLL procedure, korišćena je tehnika plitkog utapanja u logiku višeg reda. Rešavač je zasnovan na imperativnom pseudokodu navedenom u glavi 9. Kako bi se olakšalo razumevanje i kako bi nova implementacija bila predstavljena kao zaokružena celina, uz navođenje nove implemetacije nekih algoritama ponovljeni su i njihovi opisi već dati u glavi 9. Za razliku od prethodnog poglavlja gde je implementacija prikazana u neformalnoj, matematičkoj sintaksi, u ovom poglavlju će tokom celokupnog izlaganja biti korišćena preciznija sintaksa sistema Isabelle, tj. biće prikazan celokupan originalni izvorni kod implementacije. Kôd koristi pojmove meta-teorije uvedene u glavama 5, 6 i 8. Različiti koncepti i algoritmi će biti prikazivani u posebnim potpoglavljima. Uz prikaz implementacije, biće navođene i invarijante koji opisuju promenljive i odnose između promenljivih koje su korišćene u okviru implementacije. Ovakve invarijante potpuno karakterišu ulogu određenih promenljivih i time rasvetljavaju rad celokupnog sistema. S obzirom da će ove invarijante biti prikazane istovremeno sa kodom implementacije, možemo reći da je implementacija vođena specifikacijom. Osnovni problem koji je neophodno rešiti prilikom kreiranja ovakve implementacije je odnos prema stanju rešavača, odnosno globalnim promenljivim koje su korišćene u okviru imperativne implementacije jer funkcije logike višeg reda ne mogu da imaju bočne efekte. Ovaj problem će biti razrešen tehnikama *monadičkog programiranja (eng. monadic programming)* koje su opisane u poglavlju 10.2.1.

10.2.1 Monadičko programiranje

U ovom poglavlju će biti uvedeni osnovni pojmovi *monadičkog funkcionalnog programiranja*, koje je prvobitno korišćeno u jeziku Haskell [LJ94]. Pošto je i u sistemu Isabelle moguće koristiti ovaj stil programiranja i pošto odskora sistem Isabelle pruža i određenu ugrađenu podršku za njega, celokupan kôd u ovom poglavlju će biti kreiran u okviru sistema Isabelle.

Jedan od osnovnih zadataka monadičkog stila programiranja je da se u okviru čisto funkcionalnog programskog jezika modeluju funkcije koje imaju određeni *bočni efekat (eng. side-effect)*, naravno, zadržavajući svojstvo *referencijalne transparentnosti (emg. referential transparency)* koje garantuje da za iste ulazne argumente funkcija uvek izračunava istu vrednost. Bočni efekti mogu biti različiti: izmene vrednosti globalnih promenljivih, komunikacija sa spoljnim svetom vršenjem ulaza i izlaza, korišćenje izuzetaka za signaliziranje grešaka, itd. U nastavku ćemo se baviti isključivo modelovanjem izračunavanja koja koriste globalno stanje.

U imperativnim i objektno-orijentisanim programskim jezicima, stanje programa je implicitno predstavljeno globalnim promenljivim i promenljivim članicama klasa. Funkcije programa pristupaju i menjaju ovakvo stanje obično kroz svoje bočne efekte. Funkcije definisane u okviru čistih funkcionalnih jezika (pa samim tim ni u logici višeg reda) ne mogu da imaju bočne efekte tako da je stanje programa neophodno upakovati i predstaviti slogom koji se zatim eksplicitno prosleđuje kroz funkcijske pozive. Pretpostavimo da želimo da modelujemo funkcije koje kao bočni efekat menjaju tekuće stanje sistema. Ovo je moguće modelovati tako što se globalno stanje predstavlja posebnim tipom *State* i tako

što se svaka funkcija koja menja stanje prilagodi tako što joj se tekuće stanje prosleđuje kao argument, a novo stanje vraća uz rezultat njenog izračunavanja. To znači da se svaka funkcija tipa $'a_1 \Rightarrow 'a_2 \Rightarrow \ldots \Rightarrow 'a_k \Rightarrow 'a$ preuređuje tako da postaje funkcija tipa $'a_1 \Rightarrow 'a_2 \Rightarrow \ldots \Rightarrow 'a_k \Rightarrow State \Rightarrow ('a, State)$. Nakon instanciranja svojih prvih k argumenata, funkcija postaje tipa $State \Rightarrow ('a, State)$. Pošto je ovaj tip od posebne važnosti, predstavljaćemo ga imenom $StateTransformer$.

```
types 'a StateTransformer = "State ⇒ ('a × State)"
```

Primer 10.1. Pretpostavimo da stanje sistema opisuje slog koje sadrži celobrojne promenljive `x`, `y` i `z`. Moguće je definisati transformatore stanja koji vrše čitanje promenljvih na sledeći način.

```
definition readX ::  int StateTransformer
where
readX = (λ state.  (x state, state))
```

Prikazani transformator vraća komponentu `x` stanja koje mu je prosleđeno, ne menjajući pri tom stanje.

Slično, moguće je definisati i transformatore koje vrše izmene odgovarajućih komponenata stanja i to na sledeći način.

```
definition updateX ::  int ⇒ unit StateTransformer
where
updateX newX = (λ state.  ((), state(| x := newX |)))
```

Prikazani transformator menja stanje koje mu je prosleđeno, tako što upisuje datu vrednost u komponentu `x`. Transformator, osim izmenjenog stanja, ne vraća neku smislenu vrednost, što je modelovano specijalnim tipom `unit` i njegovom jedinom vrednošću `()`, koji bi odgovarao tipu `void` u jezicima C/C++.

Jedan od načina da se kreiraju složenije procedure je da se sva međustanja i međurezultati koja nastaju tokom rada procedure eksplicitno navode u kodu, što može smanjiti njegovu čitljivost, povećati mogućnost pravljenja greške i učiniti kod veoma teškim za održavanje. Razmotrimo naredni primer.

Primer 10.2. Elementarni transformatori stanja se mogu kombinovati i primenjivati sekvencijalno kako bi se modelovale složene procedure. Tako je moguće kreirati transformator stanja koji stanje menja tako što se vrednost komponente `z` postavlja na razliku vrednosti `x` i `y` polaznog stanja, vraćajući pri tom kao rezultat zbir vrednosti `x` i `y`:

```
(λ state₀. let (result₁, state₁) = readX state₀ in
           let (result₂, state₂) = readY state₁ in
           let (result₃, state₃) = updateZ x-y state₂ in
           (result₁ + result₂, state₃)
```

Kako bi se stanje „sakrilo” i uklonilo iz sintakse zapisa, definišu se kombinatori koji ulančavaju transformatore stanja i primenjuju ih sekvancijalno. U

tom cilju, uvodi se kombinator `bind` (obično infiksno zapisan kao `>>=`) koji uzastopno primenjuje dva data transformatora stanja, uz to da se nakon primene prvog transformatora stanja rezultat njegovog izračunavanja prosleđuje drugom (naravno, uz stanje dobijeno nakon primene prvog transformatora).

```
definition bind ::  'a StateTransformer ⇒
                    ('a ⇒ 'b StateTransformer) ⇒
                    'b StateTransformer                (infixr >>= 100)
where
f >>= g = (λ state₀.  let (result₁, state₁) = f state₀ in
                      (g result₁) state₁)
```

Drugi, često korišćeni, način ulančavanja je dat kombinatorom `chain` (obično infiksno zapisan kao `>>`) koji je veoma sličan kombinatoru `bind`, pri čemu je drugi transformator stanja takav da zanemaruje rezultat izračunavanja prvog (pri čemu se, naravno, stanje koje je dobijeno nakon primene prvog transformatora ne zanemaruje).

```
definition chain ::  "'a StateTransformer ⇒
                      'b StateTransformer ⇒
                      'b StateTransformer"            (infixr ">>" 100)
where
"m >> g = (m >>= (λ x.  g))"
```

Uz kombinatore ulančavanja, uvodimo i kombinator `return` koji samo vraća datu vrednost bez promene stanja.

```
definition return ::  "'a ⇒ 'a StateTransformer"
where
"return x = (λ state₀.  (x, state₀))"
```

Tip `StateTransformer` zajedno sa kombinatorima `bind` i `return` zadovoljava sledeće *monadičke zakone (eng. monadic laws)*:

```
return a >>= f    = f a
f >>= return      = f
(f >>= g) >>= h   = f >>= (λ x.  g x >>= h)
```

Ukoliko se uvede kombinator kompozicije (poznat kao Kleisli operator i koji se obično zapisuje kao `>=>`) koji se primenjuje na funkcije koje primaju po jedan argument i vraćaju transformatore stanja, navedeni zapisi poprimaju još lepši oblik.

```
definition kleisli ::  ('a ⇒ 'b StateTransformer) ⇒
                       ('b ⇒ 'c StateTransformer) ⇒
                       ('a ⇒ 'c StateTransformer)    (infixr >=> 100)
where
f >=> g = (λ x.  f x >>= (λ y.  g y))
```

```
return >=> f     = f
f >=> return     = f
(f >=> g) >=> h  = f >=> (g >=> h)
```

U ovom obliku, još je jasnije da prva dva zakona pokazuju da se `return` ponaša kao jedinični element u odnosu na `>=>`, dok treći zakon pokazuje asocijativnost kombinatora `>=>`. Ovo pokazuje da je ovako definisani transformatori stanja pripadaju kategoriji poznatoj u teoriji kategorija kao *monada (eng. monad)*.

Korišćenjem uvedenih kombinatora, procedura iz primera 10.2 može da se predstavi na sledeći način.

```
readX >>= (λ x.  readY >>= (λ y.  (updateZ x-y >> return x+y))
```

U ovoj notaciji, stanje nije više eksplicitno vidljivo, ali je zapis i dalje nečitak i udaljen od uobičajenog zapisa prisutnog u imperativnim jezicima. Kako bi se ovo ispravilo, Isabelle (po uzoru na jezik Haskell) uvodi pogodnu `do` notaciju. U ovoj sintaksi, prethodna procedura se zapisuje na sledeći način.

```
do
  x ← readX;
  y ← readY;
  updateZ x-y;
  return x+y
done
```

Ovo u velikoj meri odgovara sintaksi korišćenoj u okviru imperativnih programskih jezika.

10.2.2 Stanje rešavača

Kao što je već rečeno, zbog nepostojanja globalnih promenljivih u okviru logike višeg reda i zbog nemogućnosti korišćenja bočnih efekata, stanje SAT rešavača u okviru naše implementacije je predstavljeno sledećim slogom[1].

```
record State =
  "getSATFlag"         ::  ExtendedBool
  "getF"               ::  Formula
  "getM"               ::  LiteralTrail
  "getConflictFlag"    ::  bool
  "getConflictClause"  ::  pClause
  "getQ"               ::  "Literal list"
  "getReason"          ::  "Literal ⇒ pClause option"
  "getWatch1"          ::  "pClause ⇒ Literal option"
  "getWatch2"          ::  "pClause ⇒ Literal option"
  "getWatchList"       ::  "Literal ⇒ pClause list"
  "getC"               ::  Clause
  "getCl"              ::  Literal
  "getCll"             ::  Literal
```

Osnovne komponente ovako definisanog stanja rešavača su sledeće.

- *Status procesa rešavanja* koji je predstavljen promenljivom $SATFlag$ čija je vrednost $UNDEF$ do trenutka kada se ne utvrdi da je formula zadovoljiva (kada se $SATFlag$ postavlja na $TRUE$) ili da je nezadovoljiva

[1] Tip podataka `pClause` je sinonim za prirodne brojeve `nat` i predstavlja "pokazivače" na klauze, tj. poziciju klauze u nizu klauza koji predstavlja formulu.

(kada se $SATFlag$ postavlja na $FALSE$). Ova promenljiva je, dakle okarakterisana narednim uslovom.

$Invarijanta_{SATFlag}$[2]:

$$SATFlag = True \leftrightarrow (\mathsf{sat}\ F_0)\ \wedge\ SATFlag = False \leftrightarrow \neg(\mathsf{sat}\ F_0),$$

pri čemu je F_0 formula čija se zadovoljivost ispituje.

Dokazivanje da je ova karakterizacija tačna nakon primene procesa rešavanja je osnovni zadatak prilikom utvrđivanja parcijalne korektnosti i ovo svojstvo će biti dokazano u poglavlju 10.2.9.

- *Tekuća označena parcijalna valuacija* M (sa $\widehat{M}$ će biti označena lista literala označene valuacije M tj. valuacija koja se dobija od označene valuacije M ukoliko se oznake literala zanemare). Nju karakterišu naredne invarijante (već opisane u delu *Formalizacija*):

  $Invarijanta_{Mconsistent}$: $(\mathsf{consistent}\ \widehat{M})$
  $Invarijanta_{Mdistinct}$: $(\mathsf{distinct}\ \widehat{M})$,
  $Invarijanta_{Mvars}$: $(\mathsf{vars}\ M) \subseteq (\mathsf{vars}\ F_0) \cup decisionVars$

- *Tekuća formula* F koja se naziva i *tekući skup klauza*. Ona se tokom procesa rešavanja menja i njeni članovi su ili (uprošćene) klauze polazne formule F_0 ili njene posledice naučene tokom procesa rešavanja. Naravno, pošto su polazne klauze u formuli F izgradjene od literala formule F_0 i pošto se naučene klauze grade od literala iz M i formule F, formula F zadovoljava narednu invarijantu (već navedenu u delu *Formalizacija*):

  $Invarijanta_{Fvars}$: $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup decisionVars$

Komponente stanja zadovoljavaju i naredna svojstva.

Sve klauze u formuli F imaju bar dva različita literala. Jednočlane klauze $[l]$ se ne dodaju u formulu F već se njihov jedini literal l odmah dodaje u M. Zaista, dodavanje ovakve klauza u F je nepotrebno jer njen jedini literal l mora biti sadržan u svakom modelu, a klauza biva zadovoljena čim se izvrši njegovo dodavanje. Kako bi se osigurala korektnost, od trenutka kada se literal l doda u M, on se iz nje ne sme uklanjati. Ovo će biti slučaj i u implementaciji koju ovde opisujemo jer se ovakvi literali dodaju isključivo na nulti nivo označene valuacije M, a on se nikada ne uklanja.

Kao što je rečeno, sve klauze u F su logičke posledice formule F_0. Takođe, nulti nivo označene valuacije M sadrži literale koji su logičke posledice formule F_0. Naredna invarijanta opisuje vezu između polazne formule F_0, formule F i označene valuacije M i ona igra veoma važnu ulogu u dokazivanju saglasnosti i potpunosti implementacije koju gradimo. Ova invarijanta govori da je formula F potpuno okarakterisana formulom F i nultim nivoom označene valuacije M.

$Invarijanta_{equivalent}$: $F_0 \equiv F \mathbin{@} \langle \mathsf{prefix}\widehat{\mathsf{ToLevel}}\ 0\ M \rangle$

Činjenica da F sadrži samo klauze sa bar dva različita literala olakšava i implementaciju sheme dva posmatrana literala (koja će biti opisana u 10.2.5).

Ostale komponente stanja rešavača se koriste tokom pojedničnih faza procesa rešavanja i biće opisane u odgovarajućim poglavljima.

[2] Reći ćemo da *stanje zadovoljava invarijantu* odnosno da *invarijanta važi u stanju* ukoliko komponente stanja (`getXXX`) zadovoljavaju uslov zadat invarijantom.

10.2.3 Inicijalizacija

U ovom poglavlju će biti opisan proces inicijalizacije rešavača nakon što se zada formula F_0 čija se zadovoljivost ispituje. Ovaj metod inicijalizacije u potpunosti odgovara rešavaču SAT Solver v.8 čiji je pseudokod prikazan na strani 121.

Funkcija initialize prosleđuje svaku klauzu formule F_0 funkciji addClause koja u skladu sa tim ažurira stanje rešavača (proširujući F u slučaju višečlanih, odnosno M u slučaju jednočlanih klauza).

```
primrec initialize ::  "Formula ⇒ unit StateTransformer"
where
  "initialize [] = return ()"
| "initialize (clause # formula) =
  do
    addClause clause;
    initialize formula
  done
```

Funkcija initialize se direktno poziva samo za initialState, kada u označenoj valuaciji M još nema pretpostavljenih literala.

```
definition initialState ::  "State"
where
"initialState =
  (| getSATFlag = UNDEF,
    getF = [],
    getM = [],
    getConflictFlag = False,
    getConflictClause = 0,
    getQ = [], getWatch1 = λ c.  None,
    getWatch2 = λ c.  None,
    getWatchList = λ l.  [],
    getReason = λ l.  None,
    getC = arbitrary,
    getCl = arbitrary,
    getCll = arbitrary
  |)
"
```

Pre nego što prikažemo funkciju addClause, definisaćemo pomoćnu funkciju removeFalseLiterals koja se koristi za uprošćavanje klauza. Ona uklanja sve literale iz date klauze koji su netačni u datoj valuaciji.

```
definition removeFalseLiterals ::  "Clause ⇒ Valuation ⇒ Clause"
where
"removeFalseLiterals clause valuation =
  filter (λ l.  valuation ⊭¬ l) clause"
```

Funkcija addClause (koje se poziva isključivo iz funkcije initialize) prvo preprocesira klauzu uklanjajući iz nje ponovljene literale i uklanjajući literale netačne u označenoj valuaciji M. Nakon toga, može da nastupi nekoliko slučajeva.

- Ukoliko je klauza zadovoljena u tekućoj označenoj valuaciji M, ona se samo preskače. Obrazloženje za ovo je da, ukoliko postoji zadovoljavajuća valuacija za formulu F_0, ona će biti proširenje tekuće valuacije M i svakako će zadovoljavati i preskočenu klauzu.

- Ukoliko je nakon preprocesiranja klauza prazna, tada je formula F_0 nezadovoljiva i $SATFlag$ se postavlja a $False$, pošto se prazna klauza ne može zadovoljiti ni u jednoj valuaciji.

- Tautološke klauze (tj. klauze koje sadrže dva suprotna literala) se takođe preskaču, pošto se one uvek mogu zadovoljiti.

Jedino dva preostala slučaja menjaju F ili M.

1. Kao što je opisano, jednočlane klauze $[l]$ imaju poseban tretman. Pošto se one mogu zadovoljiti samo ako je literal l tačan u M, l se odmah dodaje u M, nakon čega se izvodi jedna iscrpna runda propagacije jediničnih klauza (videti poglavlje 10.2.6) koja može da otkrije dalje posledice dodavanja literala l.

2. Klauze koje imaju više literala se dodaju u u F i strukture podataka koje se odnose na shemu dva posmatrana literala se ažuriraju (videti poglavlje 10.2.5).

```
definition addClause ::  "Clause ⇒ unit StateTransformer"
where
"addClause clause =
  do
    M ← readM;
    let clause' = (remdups (removeFalseLiterals clause (elements M)));
    (if (¬ clauseTrue clause' (elements M)) then
      (if clause'=[] then
        updateSATFlag FALSE
      else (if (length clause' = 1) then
        do
          assertLiteral (hd clause') False;
          exhaustiveUnitPropagate
        done
      else (if (¬ clauseTautology clause') then
        do
          F ← readF;
          let clauseIndex = length F;
          updateF (F @ [clause']);
          setWatch1 clauseIndex (nth clause' 0);
          setWatch2 clauseIndex (nth clause' 1)
        done
      )))
    )
  done
"
```

10.2.4 Glavna petlja rešavača

Jedina funkcija koju bi krajnji korisnici rešavača trebalo direktno da pozivaju je funkcija solve. Ona prvo sprovodi inicijalizaciju, a zatim izvodi glavnu petlju rešavača sve dok je status procesa rešavanja nepoznat (tj. dok je vrednost promenljive $SATFlag\,UNDEF$). Prvi put kada se promenljiva $SATFlag$ promeni, glavna petlja rešavača staje i vrednost $SATFlag$ predstavlja rezltat rada rešavača. Implementacija glavne petlje rešavača u mnogome odgovara rešavačima zasnovanim na rešavaču SAT Solver v.3, čiji je pseudokod prikazan na strani 112.

```
definition solve ::  "Formula ⇒ ExtendedBool"
where
"solve F0 = getResult (runTransformer (solve_formula F0) initialState)"
```

```
definition solve_formula ::  "Formula ⇒ ExtendedBool StateTransformer"
where
"solve_formula F0 =
  do
    initialize F0;
    solve_loop (vars F0);
    readSATFlag
  done
"
```

```
function (domintros, tailrec)
  solve_loop ::  "Variable set ⇒ unit StateTransformer"
where
"solve_loop Vbl =
  do
    SATFlag ← readSATFlag;
    (if (SATFlag = UNDEF) then
      do
        solve_loop_body Vbl;
        solve_loop Vbl
      done
    )
  done
"
by pat_completeness auto
```

Primetimo da je funkcija solve_loop definisana opštom rekurzijom i da je njeno zaustavljanje netrivijalno.

Telo glavne petlje počinje iscrpnom rundom propagacije jediničnih klauza. Nakon toga, mogu da nastupe četiri slučaja.

1. Ukoliko se otkrije da važi $M \models\neg F$, kažemo da je došlo do *konflikta*.

   a) Ukoliko u M nema pretpostavljenih literala, kažemo da je došlo do *konflikta nultog nivoa* i tada se zaključuje da je formula F_0 nezadovoljiva. U tom slučaju, promenljiva $SATFlag$ se postavlja na $FALSE$.

b) Ukoliko u M ima pretpostavljenih literala, tada se izvodi postupak *analize i razrešavanja konflikta* (videti poglavlje 10.2.8).

2. Ukoliko važi $M \models \neg F$, tada su mogući sledeći slučajevi.

   a) Ukoliko su sve promenljive koje pripadaju unapred datom skupu $decisionVars$ definisane u tekućoj označenoj valuaciji M, zaključuje se da je formula zadovoljiva. U tom slučaju, promenljiva $SATFlag$ se postavlja na $TRUE$. Skup $decisionVars$ mora da zadovolji neke uslove kako bi se mogla garantovati korektnost ovog zaključka. Na primer, dovoljno je zahtevati da je $(\mathsf{vars}\ F_0) \subseteq decisionVars$, kao što je to slučaj u ovoj implementaciji.

   b) Ukoliko postoje neke varijable koje se koriste za pretpostavljanje, a koje nisu definisane u M, pravi se nova odluka (videti poglavlje 10.2.7) i novi pretpostavljeni literal se dodaje u valuaciju.

Otkrivanje klauza formule F koje su netačne ili jedinične u valuaciji $\widehat{M}$ mora biti izvršeno efikasno, kako ovaj zadatak ne bi postao usko grlo procedure. Jedan efikasan način da se ovo postigne je opisan u poglavlju 10.2.5.

```
definition solve_loop_body ::  "Variable set ⇒ unit StateTransformer"
where
"solve_loop_body decisionVars =
  do
    exhaustiveUnitPropagate;
    conflictFlag ← readConflictFlag;
    M ← readM;
    (if conflictFlag then
      (if (currentLevel M) = 0 then
        updateSATFlag FALSE
      else
        do
          applyConflict;
          applyExplainUIP;
          applyLearn;
          applyBackjump
        done
      )
    else
      (if (vars (elements M) ⊇ decisionVars) then
        updateSATFlag TRUE
      else
        applyDecide decisionVars
      )
    )
  done
"
```

10.2.5 Otkrivanje konfliktnih i jediničnih klauza

Svaki put kada se literal doda u označenu valuaciju M, potrebno je proveriti formulu F i pronaći sve klauze koje su u međuvremenu postale jednične ili

netačne. Efikasan način da se ovo uradi je implementiran kroz rešavač SAT Solver v.9 čiji je pseudokod prikazan na strani 126. Rezultati ove provere se čuvaju kroz sledeće komponente stanja rešavača.

- Promenljiva $conflictFlag$ se postavlja na tačno kada se otkrije da je tekuća formula F netačna u valuaciji $\widehat{M}$. Invarijanta koja potpuno karakteriše ovu promenljivu je:

  $Invarijanta_{conflictFlagCharacterization}$: $conflictFlag \iff \widehat{M} \models\neg F$

- Broj $conflictClause$ je indeks klauze u formuli F koja je netačna u valuaciji $\widehat{M}$. Invarijanta koja potpuno karakteriše ovu promenljivu je:

  $Invarijanta_{conflictClauseCharacterization}$:

  $$conflictFlag \implies conflictClause < |F| \ \wedge \ \widehat{M} \models\neg (F\,!\,conflictClause)$$

- Lista Q je lista literala koji su jedinični literali za klauze formule F u odnosu na tekuću valuaciju $\widehat{M}$. Ovi literali su spremni da se dodaju u M kao rezultat *propagacije jediničnih klauza*. Lista Q se ponaša kao red i u potpunosti je okarakterisana narednom invarijantom.

  $Invarijanta_{QCharacterization}$:

  $$\neg conflictFlag \implies (\forall l.\ l \in Q \iff (\exists c.\ c \in F \wedge (\mathsf{isUnitClause}\ c\ l\ \widehat{M})))$$

  Primetimo da ovaj uslov garantuje *potpunost za propagaciju jediničnih klauza* tj., garantuje da su svi jedinični literali klauza iz F u odnosu na M sadržani u Q. Ovo svojstvo nije neophodno za saglasnost i potpunost cele procedure, ali, ukoliko je zadovoljeno, dovodi do veće efikasnosti.

  Takođe, Q ne bi trebalo da sadrži ponovljene literale.

  $Invarijanta_{Qdistinct}$: $(\mathsf{distinct}\ Q)$

  Pošto se Q gradi od literala iz F, njen domen (tj. skup promenljivih) je isti kao domen za F.

  $Invarijanta_{Qvars}$: $(\mathsf{vars}\ Q) \subseteq (\mathsf{vars}\ F_0) \cup decisionVars$

- Preslikavanje $reason$ pridružuje literalima iz Q indekse klauza iz F za koje su oni jedinični literali. Pošto se ovo preslikavanje ne menja kada se literali iz Q dodaju u M, ono nastavlja da preslikava izvedene literale iz M u indekse klauza koji su razlog njihove propagacije. Literalima sa nultog nivoa tekuće označene valuacije M nisu pridružene klauze. Ovo je zbog toga što literali sa nultog nivoa imaju posebnu ulogu u procesu rešavanja pošto mogu biti dodati u M kao rezultat propagacije jednočlanih klauza koje se ne smeštaju eksplicitno u F (kao što je već objašnjeno u poglavlju 10.2.2). Svi prethodni uslovi su okarakterisani kroz narednu, kompleksnu, invarijantu:

  $Invarijanta_{reasonCharacterization}$:

  $$\begin{aligned}&((\mathsf{currentLevel}\ M) > 0 \implies \forall l.\ l \in Q \implies\\ &\qquad (\exists c.\ (reason\ l) = (\mathsf{Some}\ c) \wedge c < |F| \wedge (\mathsf{isUnit}\ (F\,!\,c)\ l\ \widehat{M}))) \ \wedge\\ &(\forall l.\ l \in \widehat{M} \wedge l \notin (\mathsf{decisions}\ M) \wedge (\mathsf{level}\ l) > 0 \implies\\ &\qquad (\exists c.\ (reason\ l) = (\mathsf{Some}\ c) \wedge c < |F| \wedge (\mathsf{isReason}\ (F\,!\,c)\ l\ \widehat{M})))\end{aligned}$$

Shema dva posmatrana literala

Efikasan način da se pronađu netačni i jedinični literali daje *shema dva posmatrana literala* (već opisana u poglavlju 9.9 i implementirana kroz rešavače SAT Solver v.9 i SAT Solver v.10). Ona uvodi sledeće promenljive u stanje:

- Preslikavanja $watch_1$ i $watch_2$ dodeljuju dva različita istaknuta literala svakoj klauzi formule F. Ovaj uslov se karakteriše narednim invarijantama:

  $Invarijanta_{watchesEl}$:

  $$\forall c.\ c < |F| \implies \exists w1\ w2.\quad (watch_1\ c) = (\mathsf{Some}\ w1) \wedge w1 \in F\,!\,c \wedge (watch_2\ c) = (\mathsf{Some}\ w2) \wedge w2 \in F\,!\,c$$

  $Invarijanta_{watchesDiffer}$:

  $$\forall c.\ c < |F| \implies (watch_1\ c) \neq (watch_2\ c)$$

- Preslikavanje $watchList$ dodeljuje svakom literalu l spisak indeksa klauza iz F koji odgovaraju klauzama za koje je l posmatrani literal. Ovo je opisano narednom invarijantom.

  $Invarijanta_{watchListsCharacterization}$:

  $$\forall l\ c.\quad c \in (watchList\ l) \iff c < |F| \wedge \big((watch_1\ c) = (\mathsf{Some}\ l) \vee (watch_2\ c) = (\mathsf{Some}\ l)\big)$$

  Takođe, važi da lista klauza ni jednog posmatranog literala ne sadrži ponovljene klauze.

  $Invarijanta_{watchListsDistinct}$: $\forall l.\ (\mathsf{distinct}\ (watchList\ l))$

U nastavku opisujemo funkciju assertLiteral koja u M dodaje literale (bilo pretpostavljene, bilo izvedene). Promenljive $conflictFlag$, $conflictClause$, Q i $reason$ se tada ažuriraju korišćenjem sheme dva posmatrana literala kodirane kroz funkciju notifyWatches.

```
definition assertLiteral ::  "Literal ⇒ bool ⇒ unit StateTransformer"
where
"assertLiteral literal decision =
  do
    M ← readM;
    updateM (M @ [(literal, decision)]);
    notifyWatches (opposite literal)
  done
"
```

Pre nego što definišemo funkciju notifyWatches, definišimo nekoliko pomoćnih funkcija.

Funkcije setWatch1 i setWatch2 proglašavaju dati literal za novi posmatrani literal date klauze dodajući odgovarajući indeks klauze u njegovu listu klauza[3].

[3] Prikazujemo samo kôd funkcije setWatch1 pošto je kôd funkcije setWatch2 veoma sličan.

```
definition addToWatchList ::
  "Literal ⇒ pClause ⇒ unit StateTransformer"
"addToWatchList literal clause =
  updateWatchList literal (λ watchList.  clause # watchList)
"
definition setWatch1 ::  "pClause ⇒ Literal ⇒ unit StateTransformer"
where
"setWatch1 clause literal =
  do
    updateWatch1 clause (Some literal);
    addToWatchList literal clause
  done
"
```

Funkcija swapWatches vrši razmenu dva posmatrana literala u datoj klauzi.

```
definition swapWatches ::  "pClause ⇒ unit StateTransformer"
where
"swapWatches clause =
  do
    wa ← readWatch1 clause;
    wb ← readWatch2 clause;
    updateWatch1 clause wb;
    updateWatch2 clause wa
  done
"
```

Funkcija getNonWatchedUnfalseLiteral proverava da li data klauza sadrži literal koji nije posmatran i koji nije netačan u M.

```
primrec getNonWatchedUnfalseLiteral ::
  "Clause ⇒ Literal ⇒ Literal ⇒ LiteralTrail ⇒ Literal option"
where
  "getNonWatchedUnfalseLiteral [] w1 w2 M = None"
| "getNonWatchedUnfalseLiteral (literal # clause) w1 w2 M =
    (if literal ≠ w1 ∧ literal ≠ w2 ∧ (elements M) ⊨¬ literal then
      Some literal
    else
      getNonWatchedUnfalseLiteral clause w1 w2 M
    )
  "
```

U nastavku je objašnjena suština sheme dva posmatrana literala data kroz funkcije notifyWatches i notifyWatches_loop. Shema dva posmatrana literala se zasniva na činjenici da posmatrani literal klauze može biti netačan u valuaciji $\widehat{M}$ samo kada je sama klauza tačna, netačna ili jedinična u $\widehat{M}$. U svim ostalim slučajevima (kada je vrednost klauze nedefinisana i nije jedinična), oba posmatrana literala nisu netačna. Ovaj uslov je dat kroz naredne invarijante (za $i = 1$ i $i = 2$).

$$\begin{aligned}\forall c.\ c < |F| \implies M \models\neg (watch_i\ c) \implies \\ (\exists l.\ l \in c \wedge M \models l \ \wedge\ \mathsf{level}\ l \leq \mathsf{level}\ \overline{(watch_i\ c)}) \vee \\ (\forall l.\ l \in c \wedge l \neq (watch_1\ c) \wedge l \neq (watch_2\ c) \implies \\ M \models\neg l \ \wedge\ \mathsf{level}\ \overline{l} \leq \mathsf{level}\ \overline{(watch_i\ c)}).\end{aligned}$$

Dodatni uslovi koji su nametnuti na nivoe literala su neophodni kako bi se mogla pokazati korektnost operacije povratnih skokova, kao što će biti obajšnjeno u poglavlju 10.2.8.

Tokom operacije assertLiteral, označena valuacija M se proširuje novim literalom l. Kada se ovo dogodi, sve klauze koje nemaju $\overline{l}$ kao svoj posmatrani literal i dalje zadovoljavaju uslove invarijante $Invarijanta_{watchCharacterization}$ tako da ne mogu da budu jedinične ni netačne u proširenoj M. Jedine klauze koje su nakon proširenja mogle postati jedinične ili netačne su one kojima je $\overline{l}$ posmatrani literal. Ovo su tačno one klauze čiji su indeksi članovi liste $(watchList\ \overline{l})$. Funkcija notifyWatches poziva funkciju notifyWatches_loop koja obilazi ovu listu i obrađuje jednu po jednu klauzu koja je u njoj sadržana. Kako bi se olakšala implementacija, za svaki obradjeni indeks klauze c, posmatrani literali se menjaju ukoliko je to neophodno kako bi se obezbedili da važi $(watch_2\ c) = \overline{l}$ i literal $(watch_2\ c)$ je netačan. Mogu da nastupe naredni slučajevi.

1. Ukoliko se utvrdi da je $(watch_1\ c)$ tačan u proširenoj valuaciji, tada nema potrebe menjati posmatrane literale, pošto klauza c zadovoljava uslove invarijante $Invarijanta_{watchCharacterization}$ i za proširenu valuaciju.

2. Ukoliko literal $(watch_1\ c)$ nije tačan u proširenoj valuaciji, pristupa se klauzi $F!c$ i njeni literali se ispituju korišćenjem funkcije getNonWatchedUnfalseLiteral.

   a) Ukoliko postoji neposmatrani literal l koji nije netačan u proširenoj valuaciji on postaje novi $(watch_2\ c)$.

   b) Ukoliko su svi neposmatrani literali netačni u M, a netačan je i $(watch_1\ c)$, onda je cela klauza netačna i $conflictFlag$ se postavlja na tačno. Posmatrani literali se ne menjaju pošto će oba postati nedefinisani u M, ukoliko se primeni operacija povratnog skoka (videti poglavlje 10.2.8).

   c) Ukoliko su svi neposmatrani literali netačni u M, a literal $(watch_1\ c)$ nije netačan, tada je klauza upravo postala jedinična i njen $(watch_1\ c)$ se umeće se u red Q (ako se već tamo ne nalazi). Razlog za njegovu propagaciju se postavlja na c. Ni u ovom slučaju se ne menjaju posmatrani literali, zbog toga što će klauza imati tačan literal nakon propagacije i time će zadovoljavati invarijantu $Invarijanta_{watchCharacterization}$.

Kada literal koji nije bio posmatran postane novi $(watch_2\ c)$, literal $\overline{l}$ prestaje da bude posmatrani za klauzu određenu indeksom c i indeks c bi trebalo ukloniti iz njemu pridružene liste klauza. Pošto se ovo dešava mnogo puta tokom obilaska koji vrši funkcija notifyWatches_loop, ispostavlja se da je

efikasnije iznova izgraditi listu indeksa klauza u kojima je $\overline{l}$ posmatrani literal, nego uklanjati jedan po jedan indeks iz već postojeće liste. Ovo je uloga parametra *newWl* u funkciji notifyWatches_loop.

```
definition notifyWatches ::  "Literal ⇒ unit StateTransformer"
where
"notifyWatches literal =
   do
     wl ← readWatchList literal;
     notifyWatches_loop literal wl []
   done
"
```

```
primrec notifyWatches_loop ::
  "Literal ⇒ pClause list ⇒ pClause list ⇒ unit StateTransformer"
where
  "notifyWatches_loop literal [] newWl =
    updateWatchList literal (λ wl.  newWl)"
| "notifyWatches_loop literal (clause # list') newWl =
  do
    w1' ← readWatch1 clause;
    w2' ← readWatch2 clause;
    if (Some literal = w1') then (swapWatches clause);
    (case w1' of Some w1 ⇒ (
     case w2' of Some w2 ⇒ (
      do
        M ← readM;
        (if literalTrue w1 (elements M) then
          notifyWatches_loop literal list' (clause # newWl)
        else
          do
            F ← readF;
            let ul = getNonWatchedUnfalseLiteral (F!clause) w1 w2 M in
            (case ul of
              Some l' ⇒
              do
                setWatch2 clause l';
                notifyWatches_loop literal list' newWl
              done
            | None ⇒
              (if (literalFalse w1 (elements M)) then
                do
                  updateConflictFlag True;
                  updateConflictClause clause;
                  notifyWatches_loop literal list' (clause # newWl)
                done
              else
                do
                  Q ← readQ;
                  if (¬ w1 el Q) then (updateQ (Q @ [w1]));
```

```
              updateReason w1 clause;
              notifyWatches_loop literal list' (clause # newWl)
            done
          )
        )
      done
    )
  done
 )))
done
"
```

Invarijante $Invarijanta_{watchListsCharacterization}$ i $Invarijanta_{watchesEl}$ zajedno obezbeđuju da za svaku klauzu postoje dva posmatrana literal (tako da nedostajuće `None` grane u `case` izrazima zaista i nisu potrebne).

10.2.6 Propagacija jediničnih klauza

Operacija propagacije jediničnih klauza dodaje u označenu valuaciju M literale koje su jedinični za klauze iz F. Pošto je shema dva posmatrana literala potpuna za detekciju netačnih i jediničnih klauza (pošto funkcija assertLiteral čuva invarijante $Invarijanta_{conflictFlagCharacterization}$ i $Invarijanta_{QCharacterization}$), svi literali koji su jedinični za klauze iz F se nalaze u Q. Zahvaljujući ovome, propagacija jediničnih klauza se trivijalno implementira — literali se uzimaju iz reda Q i dodaju se u M sve dok se Q ne isprazni ili dok se ne otkrije konflikt.

```
definition applyUnitPropagate ::  "unit StateTransformer"
where
"applyUnitPropagate =
  do
    Q ← readQ;
    assertLiteral (hd Q) False;
    Q' ← readQ;
    updateQ (tl Q')
  done
"
```

```
function (domintros, tailrec)
  exhaustiveUnitPropagate ::  "unit StateTransformer"
where
"exhaustiveUnitPropagate =
  do
    conflictFlag ← readConflictFlag;
    Q ← readQ;
```

```
    (if (¬ conflictFlag ∧ Q ≠ []) then
      do
        applyUnitPropagate;
        exhaustiveUnitPropagate
      done
    )
  done
"
by pat_completeness auto
```

Primetimo da zaustavljanje funkcije exhaustiveUnitPropagate nije trivijalno pošto je funkcija definisana korišćenjem opšte rekurzije.

10.2.7 Heuristika izbora literala

U trenutku kada se iscrpi mogućnost zaključivanja na osnovu jediničnih klauza potrebno je pokrenuti proces pretrage. Ova pretraga je vođena odlukama određenim *heuristikom izbora literala*. Srce ove heuristike pretstavlja funkcija selectLiteral čiji je zadatak da odabere literal čija je promenljiva pripada fiksiranom skupa promenljivih $decisionVars$, a čija vrednost nije definisana u označenoj valuaciji M. Ovakav izbor je moguće vršiti na osnovu različitih kriterijuma i eksperimenti govore da je kriterijum izbora literala često od presudnog značaja za efikasnost rešavača. Ipak, u ovoj tezi nećemo razmatrati konkretne kriterijume izbora već ćemo razmatrati samo njihov efekat zadat narednim postuslovom.

```
consts selectLiteral ::  "Variable set ⇒ Literal StateTransformer"
axioms selectLiteral_def:
"let diff = decisionVars \ vars (elements (getM state)) in
  diff ≠ ∅ ⟶ var (selectLiteral state decisionVars) ∈ diff"
```

```
definition applyDecide ::  "Variable set ⇒ unit StateTransformer"
where
"applyDecide decisionVars =
  do
    l ← selectLiteral decisionVars;
    assertLiteral l True
  done
"
```

10.2.8 Razrešavanje konflikata

Procedura razrešavanja konflikata koja će u ovom poglavlju biti formalizovana se zasniva na pseudokod-u rešavača SAT Solver v.4 prikazanog na strani 116. Procedura razrešavanja konflikata se sastoji od *analize konflikata*, *učenja* i *povratnih skokova* i izvršava se kada god se ustanovi konflikt pri čemu u tekućoj označenoj valuaciji M postoji bar jedan pretpostavljeni literal (kada se ustanovi konflikt na nultom nivou označene valuacije M, formula se proglašava nezadovoljivom i ne vrši se razrešavanje konflikta). Nakon primene procedure razrešavanja konflikta, određeni broj literala se uklanja sa vrha označene valuacije M i restauriše se ne-konfliktno stanje. Za razliku od klasične pretrage

sa povratkom (eng. backtracking) koja uklanja samo poslednji pretpostavljeni literal iz M, operacija povratnog skoka vrši tzv. ne-hronološki povratak koji teži da ukloni što veći broj pretpostavljenih literala iz M. Povratni skokovi su vođeni *klauzama povratnog skoka (eng. backjump clause)* koje su posledice polazne formule F_0 i koje odgovaraju valuacijama koje su dovele do konflikta. Kada se konstruiše klauza povratnog skoka, literali sa vrha označene valuacije M se uklanjaju sve dok klauza povratnog skoka ne postane jedinična klauza u M. Od tog trenutka, propagira se njen jedinični literal i nastavlja se proces pretrage. Klauze povratnog skoka se konstruišu u procesu poznatom kao *analiza konflikata (eng. conflict analysis)*.

Nekoliko komponenata stanja rešavača se koristi tokom razrešavanja konflikata.

- Klauza C predstavlja tekuću klauzu analize konflikata koja, kada se proces analize konflikta završi, postaje klauza povratnog skoka. Ova klauza je okarakterisana narednim invarijantama[4].

  $Invarijanta_{CFalse}$: $conflictFlag \implies M \vDash \neg C$

  $Invarijanta_{CEntailed}$: $conflictFlag \implies F_0 \vDash C$

Naredne promenljive reprezentuju određene delove i svojstva klauze C i njihovo čuvanje u okviru stanja rešavača se vrši samo kako bi se dobilo na efikasnosti.

- Literal C_l literal koji pripada klauzi $\overline{C}$ koji je poslednji postavljen u označenu valuaciju M. Okarakterisan je narednom invarijantom.

  $Invarijanta_{ClCharacterization}$:

  $$conflictFlag \implies (\mathsf{isLastAssertedLiteral}\ C_l\ \overline{C}\ M)$$

- Literal C_{ll} je poslednji literal klauze $\overline{C} \setminus C_l$ koji je postavljen u označenu valuaciju M. Okarakterisan je narednom invarijantom.

  $Invarijanta_{CllCharacterization}$:

  $$conflictFlag \wedge \overline{C} \setminus C_l \neq [] \implies (\mathsf{isLastAssertedLiteral}\ C_{ll}\ (\overline{C} \setminus C_l)\ M)$$

- Broj C_n je broj literala klauze $\overline{C}$ koji se nalaze na poslednjem nivou označene valuacije M. Okarakterisan je narednom invarijantom.

  $Invarijanta_{CnCharacterization}$:

  $$conflictFlag \longrightarrow$$
  $$C_n = (\mathsf{length}\ (\mathsf{filter}\ (\lambda\ l.\ \mathsf{level}\ \overline{l}\ M = \mathsf{currentLevel}\ M)\ (\mathsf{remdups}\ C)))$$

[4]Sve invarijante koje su relevantne za proces razrešavanja konflikata moraju da važe samo dok se konflikt ne razreši. U formulacijama koje su navedene, dodat je uslov $conflictFlag \implies$ kako bi se ove invarijante mogle smatrati globalnim invarijantama i kako bi se tretirale ekvivalentno ostalim definisanim invarijantama.

Analiza konflikata

Kako bi se implementirala procedura analize konflikata, uvodimo nekoliko pomoćnih funkcija.

Funkcija findLastAssertedLiteral se koristi kako bi se postavila vrednost promenljive C_l na osnovu tekućih vrednosti promenljivih C i M.

```
definition findLastAssertedLiteral ::  "unit StateTransformer"
where
"findLastAssertedLiteral =
  do
    C ← readC;
    M ← readM;
    updateCl (getLastAssertedLiteral (opposite C) (elements M))
  done
"
```

Funkcija countCurrentLevelLiterals se koristi kako bi se postavila vrednost promenljive C_n na osnovu tekućih vrednosti promenljivih C i M.

```
definition countCurrentLevelLiterals ::  "unit StateTransformer"
where
"countCurrentLevelLiterals =
  do
    M ← readM;
    C ← readC;
    let cl = currentLevel M;
    let cll = filter (λ l.  elementLevel (opposite l) M = cl) C;
    updateCn (length cll)
  done
"
```

Pošto neki literali na nultom nivou označene valuacije M nemaju pridružene klauze u F koje predstvljaju razloge njihove propagacije, zahteva se da klauza C ne sadrži literale sa nultog nivoa M. Takođe, razumno je zahtevati da klauza C ne sadrži ponovljene literale. Funkcija setConflictAnalysisClause postavlja klauzu C na datu klauzu, ali tako što je prvo preprocesira uklanjanjem svih duplikata i svih literala koji se nalaze na nultom nivou valuacije M. Ova funkcija, takođe, ažurira i vrednosti promenljivih C_l i C_n.

```
definition setConflictAnalysisClause ::  "unit StateTransformer"
where
"setConflictAnalysisClause clause =
  do
    M ← readM;
    let oppM0 = oppositeLiteralList (elements (prefixToLevel 0 M));
    updateC (remdups (list_diff clause oppM0));
    findLastAssertedLiteral;
    countCurrentLevelLiterals
  done
"
```

Algoritam analize konflikata se može opisati na sledeći način.

- Proces analize konflikata počinje sa konfliktnom klauzom (klauzom formule F koja je postala netačna u M) i klauza C se inicijalizuje na konfliktnu klauzu korišćenjem funkcije applyConflict.

```
definition applyConflict ::  "unit StateTransformer"
where
"applyConflict =
  do
    F ← readF;
    conflictClause ← readConflictClause;
    setConflictAnalysisClause (nth F conflictClause)
  done
"
```

- Svaki literal koji je član tekuće klauze C je netačan u tekućoj valuaciji $\widehat{M}$ i ili je pretpostavljeni literal ili je rezultat neke propagacije. Pošto C ne sadrži literale nultog nivoa, za svaki propagirani literal l, postoji klauza c na osnovu koje je došlo do propagacije. Ove se klauze nazivaju *klauze razloga propagacije (eng. reason clauses)* i zadovoljavaju uslov (isReason c l $\widehat{M}$). Propagirani literali tekuće klauze C se onda menjanju (kaže se i *objašnjavaju (eng. explain)*) ostalim literalima iz klauze razloga propagacije, čime se proces analize nastavlja. Korak objašnjavanja se može shvatiti i kao rezolucija između tekuće klauze analize konflikta C i klauze koja je razlog propagacije nekog njenog literala. Ova rezolucija je implementirana kroz funkciju applyExplain.

```
definition applyExplain ::  "Literal ⇒ unit StateTransformer"
where
"applyExplain literal =
  do
    reason' ← readReason literal;
    (case reason' of Some reason ⇒
      do
        C ← readC;
        F ← readF;
        let res = resolve C (nth F reason) (opposite literal);
        setConflictAnalysisClause res
      done
    )
  done
"
```

Primetimo da invarijanta $Invarijanta_{reasonCharacterization}$ garantuje da svaki propagirani literal ima pridruženu klauzu koja je razlog njegove propagacije te nedostajuća `None` grana u `case` izrazu zaista nije neophodna.

- Procedura analize konflikata koja je implementirana uvek objašnjava literal iz $\overline{C}$ koji je poslednji dodat u M i ponavlja se sve dok se isUIP uslov ne zadovolji, tj., dok se ne dogodi da u $\overline{C}$ postoji takav literal l da se svi ostali literali iz $\overline{C}$ nalaze u M na striktno manjem nivou nego što je

nivo literala l. Ovaj se uslov jednostavno proverava ispitivanjem vrednosti promenljive C_n. Implementacija ove tehnike je data kroz funkciju applyExplainUIP.

```
function (domintros, tailrec)
  applyExplainUIP ::  "unit StateTransformer"
where
"applyExplainUIP =
  do
    Cn ← readCn;
    (if (Cn ≠ 1) then
      do
        Cl ← readCl;
        applyExplain Cl;
        applyExplainUIP
      done
    )
  done
"
by pat_completeness auto
```

Primetimo da zaustavljanje ove funkcije nije trivijalno jer je funkcija definisana opštom rekurzijom.

Učenje

Tokom procesa učenja, formula F (ili nulti nivo označene valuacije M u slučaju jednočlanih klauza) se proširuje naučenim (redundantnim) klauzama koje su logičke posledice formule F_0. U našoj implementaciji (kao što je to uobičajeno slučaj kod SAT rešavača), jedine klauze koje se uče su klauze povratnog skoka. Pošto je zahtevano da sve klauze u F imaju više od dva različita literala, ako klauza povratnog skoka C sadrži samo jedan literal, učenje se ne izvodi eksplicitno (već samo implicitno tokom izvođenja povratnog skoka). Implementacija procesa učenja je data funkcijom applyLearn. Nako proširivanja formule F klauzom C, posmatrani literali klauze C se postavljaju tako da klauza C zadovolji uslove invarijante $Invarijanta_{watchCharacterization}$. Istovremeno, izračunava se literal C_{ll} i pamti se u okviru stanja.

```
definition applyLearn ::  "unit StateTransformer"
where
"applyLearn =
  do
    Cl ← readCl;
    C ← readC;
    (if (C ≠ [opposite Cl]) then
      do
        F ← readF;
        M ← readM;
        updateF (F @ [C]);
        let l = Cl;
        let oppCminusl = removeAll l (opposite C);
        let ll = getLastAssertedLiteral oppCminusl (elements M);
```

```
          let clauseIndex = length F;
          setWatch1 clauseIndex (opposite l);
          setWatch2 clauseIndex (opposite ll);
          updateCll ll
        done
    )
  done
"
```

Povratni skokovi

Operacija povratnog skoka se sastoji od uklanjanja literala sa vrha označene valuacije M do najmanjeg nivoa na kojem klauza povratnog skoka C postaje jedinična, posle čega se njen jedinični literal $\overline{C_l}$ propagira. Ovaj nivo se pronalazi korišćenjem funkcije getBackjumpLevel.

```
definition getBackjumpLevel ::  "nat StateTransformer"
where
"getBackjumpLevel =
  do
    C ← readC;
    Cl ← readCl;
    (if C = [opposite Cl] then
      return 0
    else
      do
        Cll ← readCll;
        M ← readM;
        return (elementLevel Cll M)
      done
    )
  done
"
```

Funkcija `applyBackjump` izvodi operaciju povratnog skoka.

```
definition applyBackjump ::  "unit StateTransformer"
where
"applyBackjump =
  do
    level ← getBackjumpLevel;
    Cl ← readCl;
    M ← readM;
    F ← readF;
    updateConflictFlag False;
    updateQ [];
    updateM (prefixToLevel level M);
    if (level > 0) then updateReason (opposite Cl) (length F - 1);
    assertLiteral (opposite Cl) False
  done
"
```

Primetimo da je nakon uzimanja prefiksa označene valuacije M zaključeno da je konflikt uspešno razrešen (pa je $conflictFlag$ postavljena na netačno) i

da nema jediničnih klauza u F u odnosu na uzeti prefiks od M (pa je red Q ispražnjen). Da bi ovi zaključi bili ispravni, potrebno je da se ne vrše nove pretpostavke i ne dodaju pretpostavljeni literali u M od trenutka kada postoji konflikt između M i F. Takođe, zahteva se da propagacija jediničnih klauza bude iscrpna i da se pretpostavke ne donose ni kada u F postoje jedinične klauze. Ovi uslovi su nametnuti narednim invarijantama.

$Invarijanta_{noDecisionsWhenConflict}$:

$$\forall level' < (\mathsf{currentLevel}\ M) \implies (\mathsf{prefixToLevel}\ level'\ M) \models \neg F$$

$Invarijanta_{noDecisionsWhenUnit}$:

$$\forall level' < (\mathsf{currentLevel}\ M) \implies \\ \neg\exists c\ l.\ c \in F \wedge (\mathsf{isUnitClause}\ c\ l\ (\mathsf{prefixToLevel}\ level'\ M))$$

10.2.9 Delovi dokaza korektnosti

Invarijante koje su do sada navedene u okviru tekućeg poglavlja 10.2 su dovoljne da se dokaže korektnost procedure. Najzahtevniji deo ovog dokaza je da se pokaže da svaka funkcija rešavača održava svaku od navedenih invarijanti. Ovi dokazi se donekle oslanjaju na slične dokaze navedene za apstraktne sisteme prelaza navedene u delu *Formalizacija*. Celokupni dokazi su dostupni u [Mar08] i ovde neće biti navedeni.

U nastavku će biti prikazane tehike koje su korišćene kako bi se dokazalo zaustavljanje glavne funkcije rešavača (funkcije solve). Takođe, biće formulisana teorema korektnosti i biće prikazan najviši nivo njenog dokaza.

Zaustavljanje

U kodu predstavljenom u okviru poglavlja 10.2 jedino su funkcije exhaustiveUnitPropagate, explainUIP i solve_loop definisane opštom rekurzijom i nije trivijalno da se zaustavljaju. Jedina funkcija koju bi krajnji korisnici trebalo direktno da pozivaju je funkcija solve i ona predstavlja jedinistvenu tačku ulaska u rešavač. Ovo znači da se sve tri funkcije koje su definisane oštom rekurzijom pozivaju samo indirektno od strane fukcije solve i da su svi parametri koji se ovim funkcijama prosleđuju automatski izračunati od strane rešavača. Zato se ove funkcije mogu smatrati parcijalnim funkcijama i nije neophodno dokazivati da se one zaustavljaju za sve vrednosti svojih ulaznih parametera. Dovoljno je dokazati da se zaustavljaju za one vrednosti ulaznih parametera koje im se zaista mogu proslediti u toku procesa rešavanja koji počinje iz nekog početnog stanja.

Kako bismo modelovali ovu vrstu parcijalnosti, korišćene su ugrađene funkcije sistema Isabelle [Kra08].

1. Primetimo da su sve tri funkcije definisane opštom rekurzijom definisane repnom rekurzijom i označene korišćenjem direktive `tailrec`. Ovo je veoma važno zbog toga što omogućava generatoru koda da radi sa parcijalnim funkcijama.

2. Kada se n-arna funkcija f definiše korišćenjem opšte rekurzije, sistem automatski generiše i predikat f_dom koji proverava da li se n-torka (a_1,

..., a_n) nalazi u domenu funkcije f (tj., da li se funkcija f zaustavlja za ulaz $(a_1, \ldots, a_n)$). Ukoliko se definicija funkcije još označi i direktivom `domintros`, Isabelle generiše i automatski dokazuje teoremu oblika

$$g \implies (\mathsf{f\_dom}\ (f_1(a_1), \ldots, f_n(a_n))) \implies (\mathsf{f\_dom}\ (a_1, \ldots, a_n)),$$

za svaki rekurzivni poziv $\mathsf{f}(f_1(a_1), \ldots, f_n(a_n))$ u definiciji funkcije f, gde je g uslov koji dovodi do rekurzivnog poziva (eng. guard). Dok se ne dokaže zaustavljanje funkcije f tj. dok se ne dokaže da je funkcija f totalna, uobičajena teorema indukcije za funkciju f (koja bi u sistemu bila nazvana `f.induct`) ne može da se dokaže i koristi. Međutim, kada se funkcija f definiše, sistem automatski dokazuje slabiju teoremu parcijalne indukcije (koja se naziva `f.pinduct`). Ona se razlikuje od uobičajene sheme indukcije jer dodaje dodatnu pretpostavku o pripadnosti domenu f_dom i u induktivnu bazu i u induktivne korake. Ovi predikati pripadnosti domenu se prenose i pretpostavljaju u svim lemama o funkciji f koje se dokazuju (parcijalnom) indukcijom. Ipak, kako bi se zaokružio dokaz koraktnosti, u jednom trenutku potrebno je osloboditi se ovakvih dodatnih pretpostavki. Ovo se radi tako što se pokaže da svi ulazi koji se zaista prosleđuju funkciji f zaista pripadaju njenom domenu tj. zadovoljavaju uslov f_dom.

U našem slučaju, znamo da se invarijante čuvaju tokom čitavog rada rešavača i da svako stanje na koje se primenjuju funkcije našeg rešavača zadovoljava sve navedene invarijante. Pokazaćemo da neke od ovih invarijanti povlače uslove pripadnosti domenu, tj. da se naše tri funkcije definisane opštom rekurzijom zaustavljaju za sva stanja koja zadovoljavaju invarijante.

Kao ilustraciju, skiciraćemo dokaz da se funkcija `exhaustiveUnitPropagate` (str. 154) zaustavlja ako njeno ulazno stanje zadovoljava određene invarijante.

Kako bi se ovo pokazalo, koriste se dobro zasnovana uređenja uvedene u delu *Formalizacija*. Za zaustavljanje funkcije `exhaustiveUnitPropagate` značajno je uređenje $\succ^{Vbl}_{trail}$ uvedeno definicijom 7.19. Korišćenjem ovog uređenja će biti pokazano da ukoliko stanje zadovoljava određene invarijante, ono pripada domenu funkcije exhaustiveUnitPropagate (tj. da se ova funkcija zaustavlja kada se primeni na to stanje).

Lema 10.4. *Ako je skup $decisionVars$ konačan i stanje state je takvo da:*

(a) invarijante $Invarijanta_{Mconsistent}$ (str. 144) i $Invarijanta_{Mdistinct}$ (str. 144) važe u stanju state,

(b) invarijante $Invarijanta_{Mvars}$ (str. 144), $Invarijanta_{Fvars}$ (str. 144) i $Invarijanta_{Qvars}$ (str. 149) važe u stanju state,

(c) invarijante $Invarijanta_{conflictFlagCharacterization}$ (str. 149), $Invarijanta_{QCharacterization}$ (str. 149) i $Invarijanta_{Qdistinct}$ (str. 149) važe u stanju state,

(d) invarijante $Invarijanta_{watchListsCharacterization}$ (str. 150) i $Invarijanta_{watchListsDistinct}$ (str. 150) važe u stanju state,

(e) invarijante $Invarijanta_{watchesEl}$ (str. 150), $Invarijanta_{watchesDiffer}$ (str. 150) i $Invarijanta_{watchCharacterization}$ (str. 151) važe u stanju state,

tada se funkcija exhaustiveUnitPropagate *zaustavlja kada se primeni na stanje* $state$ *tj.,* (exhaustiveUnitPropagate_dom $state$).

Dokaz. Ukoliko je Q prazan ili je $conflictFlag$ tačno u stanju $state$, tada se funkcija exhaustiveUnitPropagate trivijalno zaustavlja i stanje $state$ je u njenom domenu. Zato pretpostavimo da Q nije prazan i da je $conflictFlag$ netačno.

Dokaz se izvodi dobro-zasnovanom indukcijom po uređenju $\succ_{trail}^{Vbl}$ (uvedenom definicijom 7.19). Pretpostavimo, kao induktivnu hipotezu, da tvrđenje važi za sva stanja $state'$ za koje je $state \succ_{trail}^{Vbl} state'$. Neka je $state' =$ (applyUnitPropagate $state$). Pošto invarijante važe u stanju $state$ i pošto ih funkcija applyUnitPropagate održava, one važe i u stanju $state'$[5]. Pošto je označena valuacija M u stanju $state'$ proširena nekim literalom, na osnovu leme 7.10, važi $state \succ_{trail}^{Vbl} state'$. Tada, na osnovu induktivne hipoteze, važi (exhaustiveUnitPropagate_dom $state'$). Tvrđenje leme onda sledi na osnovu teoreme o uvođenju domena `exhaustiveUnitPropagate.domintros`:

$$\neg conflictFlag_{state} \wedge Q_{state} \neq [] \implies \\ (\text{exhaustiveUnitPropagate\_dom } (\text{applyUnitPropagate } state)) \implies \\ (\text{exhaustiveUnitPropagate\_dom } state).$$

□

Zaustavljanje (za relevantne ulaze) funkcija applyExplainUIP i solve_loop se dokazuje na veoma sličan način. Dokaz zaustavljanja za funkciju solve_loop takođe koristi uređenje $\succ_{trail}^{Vbl}$, a dokaz za funkciju applyExplainUIP koristi uređenje $\succ_{confclause}^{M}$ uvedeno u delu *Formalizacija* definicijom 7.31.

Totalna korektnost

Totalna korektnost `solve` funkcije je iskazana narednom teoremom.

Teorema 10.3.

$$((\text{solve } F_0) = TRUE \wedge (\text{sat } F_0)) \vee ((\text{solve } F_0) = FALSE \wedge \neg(\text{sat } F_0))$$

Pod pretpostavkom da sve invarijante važe u svakom stanju koje se dostigne tokom izvršavanja funkcije solve, dokaz teoreme 10.3 se zasniva na dve leme o saglasnosti koje odgovaraju mestima u kodu na kojima se zastavica $SATFlag$ menja.

Lema 10.5. *Ukoliko u nekom stanju* $state$ *važi da:*

(a) invarijanta $Invarijanta_{equivalent}$ *(str. 144) važi u stanju* $state$*,*

(b) invarijanta $Invarijanta_{conflictFlagCharacterization}$ *(str. 149) važi u stanju* $state$*,*

[5] Primetimo da je neophodno pokazati samo da invarijante $Invarijanta_{consistentM}$, $Invarijanta_{distinctM}$ i $Invarijanta_{varsM}$ moraju da važe u stanjima $state$ i $state'$ kako bi mogli da koristimo uređenje $\succ_{trail}^{Vbl}$. Međutim, neophodno je bilo pretpostaviti dodatne invarijante u pretpostavkama ove leme, zato što su one neophodne da bi se pokazalo da navedene tri invarijante neophodne za dokaz nastavljaju da važe i nakon što se primeni funkcija applyUnitPropagate.

(c) $conflictFlag$ *je tačno u stanju state,*

(d) $(\mathsf{currentLevel}\ M) = 0$ *važi u stanju state,*

tada važi $\neg(\mathsf{sat}\ F_0)$.

Dokaz. Iz pretpostavke $(\mathsf{currentLevel}\ M) = 0$ važi da je $(\mathsf{prefixToLevel}\ 0\ M) = M$. Zato, na osnovu invarijante $Invarijanta_{equivalent}$, sledi da je $F \mathbin{@} \langle \widehat{M} \rangle \equiv F_0$. Pošto na osnovu pretpostavke o vrednosti $conflictFlag$ i invarijante $Invarijanta_{conflictFlagCharacterization}$ važi $M \models \neg F$, na osnovu monotonosti važi i $M \models \neg F \mathbin{@} \langle \widehat{M} \rangle$. Pošto važi $F \mathbin{@} \langle \widehat{M} \rangle \models M$, formula $F \mathbin{@} \langle \widehat{M} \rangle$ je netačna u valuaciji koja joj je semantička posledica, pa je, na osnovu stava 6.8, nezadovoljiva. Pošto je F_0 logički ekvivalentna formula formuli $F \mathbin{@} \langle \widehat{M} \rangle$, i ona je nezadovoljiva. □

Lema 10.6. *Ukoliko u nekom stanju state važi da:*

(a) $(\mathsf{vars}\ F_0) \subseteq decisionVars$,

(b) invarijanta $Invarijanta_{Mconsistent}$ *(str. 144) važi u stanju state,*

(c) invarijanta $Invarijanta_{Fvars}$ *(str. 144) važi u stanju state,*

(d) invarijanta $Invarijanta_{equivalent}$ *(str. 144) važi u stanju state,*

(e) invarijanta $Invarijanta_{conflictFlagCharacterization}$ *(str. 149) važi u stanju state,*

(f) $conflictFlag$ *je netačno u stanju state,*

(g) $(\mathsf{vars}\ \widehat{M}) \supseteq decisionVars$,

tada važi $(\mathsf{sat}\ F_0)$ *i* $(\mathsf{model}\ \widehat{M}\ F_0)$.

Dokaz. Na osnovu invarijante $Invarijanta_{Fvars}$, sledi da je $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup decisionVars$. Uz uslov $(\mathsf{vars}\ F_0) \subseteq decisionVars$, važi da je $(\mathsf{vars}\ F) \subseteq decisionVars$. Uz uslov $(\mathsf{vars}\ \widehat{M}) \supseteq decisionVars$, važi da $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ \widehat{M})$ i $\widehat{M}$ je totalna valuacija u odnosu na skup promenljivih formule F. Tada je, na osnovu stava 6.6, ili važi $\widehat{M} \models \neg F$ ili važi $\widehat{M} \models F$. Pošto je $conflictFlag$ netačno, na osnovu invarijante $Invarijanta_{conflictFlagCharacterization}$ sledi $\widehat{M} \not\models \neg F$, tako da mora da važi $\widehat{M} \models F$. Trivijalno važi $\widehat{M} \models \langle \widehat{\mathsf{prefixToLevel}}\ 0\ M \rangle$ i valuacija $\widehat{M}$ je neprotivrečna na osnovu invarijante $Invarijanta_{Mconsistent}$. Zato je $\widehat{M}$ model formule $F \mathbin{@} \langle \widehat{\mathsf{prefixToLevel}}\ 0\ M \rangle$. Pošto važi $F_0 \equiv F \mathbin{@} \langle \widehat{\mathsf{prefixToLevel}}\ 0\ M \rangle$, važi i da je $\widehat{M}$ model formule F_0 i F_0 je zadovoljiva, tj. važi $(\mathsf{sat}\ F_0)$. □

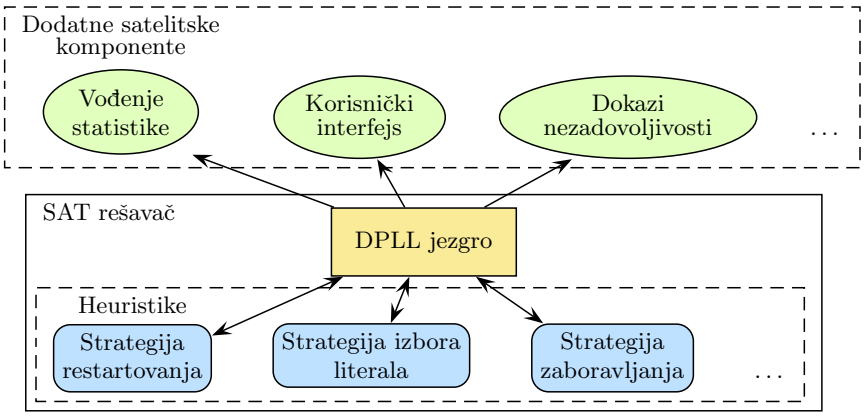
11

Fleksibilna objektno-orijentisana implementacija

Iako su tokom protekle decenije SAT rešavači izuzetno napredovali, jako je malo pažnje posvećeno dizajnu rešavača kao softverskih sistema. Naime, pored implementacije samog DPLL algoritma, neophodno je obezbediti celokupan sistem koji nudi interfejs prema korisnicima i omogućava im da jednostavno koriste funkcionalnost rešavača, vrše njegovo konfigurisanje i koriste ga kao deo širih softverskih sistema. U ovom poglavlju će biti naglašena važnost softverskog dizajna SAT rešavača i biće prikazana rešenja prisutna u rešavaču ArgoSAT. Mnoge pojedinačne ideje su prisutne i u nekim drugim dostupnim rešavačima otvorenog koda[1], no u ovom tekstu, kao i u rešavaču ArgoSAT one će biti objedinjene i prikazane na uniforman način. Arihtektura predstavlja nadgradnju DPLL jezgra rešavača opisanog u glavi 9. Takođe, arhitektura se zasniva na objektno-orijentisanoj paradigmi i koristi poznata rešenja i obrasce objektno-orijentisanog dizajna *(eng. design patterns)* [GHJ+95]. Stil programiranja je u mnogome inspirisan i idejama izloženim u [FB+99]. Tokom izlaganja, odgovarajuća rešenja će biti predstavljena korišćenjem dijagrama iskazanih na jeziku UML [BRJ05].

Osnovna osobina predložene arhitekture je učauravanje različitih koncepata prisutnih u SAT rešavačima u zasebne module. Naime, ukoliko je razvoj SAT rešavača vođen isključivo potrebama dostizanja maksimalne brzine (koja jeste ključna za mogućnost praktične upotrebljivosti) i ukoliko se posebna pažnja ne obrati na softverski dizajn, postoji opasnost da kôd postane monolitan, nefleksibilan i prepun ,,tvrdo-kodiranih" delova. Čest je slučaj da kôd implementira pregršt pametnih implementacionih trikova, ali kao posledica toga njegovo održavanje i modifikovanje postaje teško. Često dolazi do međusobnog mešanja različitih slojeva rešavača (algoritama višeg nivoa, implementacionih tehnika nižeg nivoa i heurističkih komponenti). Jedna funkcija u kôdu može da implementira više različitih koncepata SAT rešavanja i, obratno, implementacija nekog jedinstvenog koncepta (na primer heuristike) može da bude rasprostrta kroz više različitih funkcija u kôdu. Nasuprot tome, ovde predložen pristup nastoji da izoluje različite koncepte prisutne u okviru rešavača u zasebne module. Takođe, pravi se jasna razlika između tri navedena sloja rešavača. Sve ovo

[1] Po mišljenju autora, Sat4J (`http://www.sat4j.org`) je rešavač u kojem je najviše pažnje stavljeno na softverski dizajn.



Slika 11.1: Jezgro DPLL algoritma i satelitske komponente

dovodi do kôda koji je jasan, lak za čitanje, razumevanje i održavanje i o čijim se svojstvima može jednostavno rezonovati. Najvažniji korak u dostizanju ovog cilja je razdvajanje jezgra DPLL algoritma od raznih satelitskih komponenti (na primer, heurističkih komponenti, korisničkog interfejsa, generisanja dokaza nezadovoljivosti, vođenja statistika) kao što je to prikazano na slici 11.1.

Kao posledica fleksibilne arhitekture, postaje moguće istovremeno postojanje više različitih tehnika (posebno heuristika) u okviru istog SAT rešavača. Takođe, izmene postojećih i dodavanje novih komponenti postaje jednostavnije. Koristi od toga su višestruke. Prvo, moguće je praviti fer poređenja između različitih tehnika što čini rešavač pogodnim za izvođenje eksperimentalnih istraživanja na polju iskazne zadovoljivosti. Dalje, rešavač je u mogućnosti da između više podržanih tehnika odabere onu koja je najpogodnija za rešavanje date ulazne formule. Naime, razne heuristike obično pokazuju različito ponašanje na raznim klasama formula i veoma je redak slučaj da je neka od njih univerzalno nadmoćnija od svih njoj konkurentskih heuristika. Na primer, u radu [Hua07a] je dato poređenje strategija za izbor literala poteklih iz rešavača MiniSAT [ES04] i Berkmin [GN02] i zaključeno je da za svaku od njih postoji familija formula na kojoj je ona znatno bolja od druge. Skorašnja istraživanja pokazuju da je na osnovu određenih sintaksnih karakteristika iskazne formule, moguće unapred pogoditi heuristiku koja će dati najbolje rezultate [HHH$^+$06, XHH$^+$08, NMJ09].

Kôd u jeziku C++ koji sledi je direktno preuzet iz rešavača ArgoSAT, pri čemu su radi jednostavnosti prikaza izostavljeni neki tehnički detalji (npr. konstruktori i destruktori klasa nisu navođeni u tekstu).

11.1 Osnovni tipovi podataka

U ovom poglavlju će biti opisano kako su kodirani osnovni tipovi podataka — promenljive, literali, klauze i valuacije.

Promenljive i literali. S obzirom da je skup promenljivih koje se javljaju u svakoj formuli konačan, najjednostavnije je skup n promenljivih identifikovati

sa brojevima od 0 do $n-1$. Za očekivati je da je n relativno mali broj i da je sve promenljive moguće predstaviti promenljivima tipa `unsigned short` ili `unsigned`. Iz ovog razloga, tip `Variable` je napravljen kao sinonim za tip `unsigned`:

```
typedef unsigned Variable;
```

I skup literala je konačan, tako da je i njih prirodno predstaviti prirodnim brojevima.

```
typedef unsigned Literal;
```

Jedan od mogućih načina kodiranja je da se svakoj promenljivoj v pridruže pozitivan literal $2 \cdot v$ i negativan literal $2 \cdot v + 1$.

Važno je naglasiti da je, protivno svim zakonima objektno-orijentisanog dizajna, odlučeno da se ne konstruišu posebne klase kojima bi se reprezentovale promenljive, odnosno literali. Razlog ovome je to što se korišćenjem primitivnih tipova podataka umesto objekata dobija na efikasnosti celog sistema. Ipak, operacije i dodatne informacije koje su u vezi sa promenljivima i literalima su grupisane i predstavljene u okviru posebnih klasa koje uglavnom sadrže statičke metode. Tako, na primer, klasa `Literals` objedinjuje operacije za rad sa literalima.

```
class Literals {
 public:
  static Literal     opposite(Literal literal);
  static bool        isPositive(Literal literal);
  static bool        isNegative(Literal literal);
  static Literal     variable(Literal literal);
  static Literal     literal(Variable variable, bool positive);
  static Literal     fromInt(int num);
  static std::string toString(Literal literal);
};
```

Naglasimo još da je, s obzirom na jako česta korišćenja ovih operacija, potrebno obratiti pažnju na njihovu što efikasniju implementaciju. U sistemu ArgoSAT, one se uglavnom implementiraju korišćenjem bitskih operatora.

Klauze. Klauze se ponašaju kao nizovi literala i osnovne operacije koje se izvode nad klauzama su upravo operacije sa nizom literala (utvrđivanje broja literala, pristup pojedinim literalima, dodavanje, brisanje, uklanjanje duplikata itd). Raspoložive su još operacije za proveru statusa klauze u datoj valuaciji (da li je tačna, netačna ili jedinična), kao i mogućnost uprošćavanja uklanjanjem netačnih literala.

U sistemu ArgoSAT klauze su predstavljene klasom `Clause`.

```
class Clause {
public:
  size_t   size       ()         const;
  Literal  operator[] (size_t i) const;
  Literal& operator[] (size_t i);
```

```
  typedef std::vector<Literal>::const_iterator const_iterator;
  typedef std::vector<Literal>::iterator       iterator;
  const_iterator begin () const;
  iterator       begin () const;
  const_iterator end   () const;
  iterator       end   () const;

  const_iterator findLiteral    (Literal literal) const;
  iterator       findLiteral    (Literal literal);
  void           addLiteral     (Literal literal);
  bool           removeLiteral  (Literal literal);

  void removeDuplicateLiterals();

  bool isTrue  (const Valuation& v) const;
  bool isFalse (const Valuation& v) const;
  bool isUnit  (const Valuation& v) const;
  const_iterator getUnitLiteral (const Valuation& v) const;

  void removeFalsifiedLiterals(const Valuation& v);

private:
  std::vector<Literal> _literals;
};
```

Implementacija ove klase u sistemu ArgoSAT koristi standardnu kolekciju `std::vector` i većina operacija nad klauzama koje se odnose na pristup kolekciji literala se jednostavno delegiraju operacijama klase `std::vector`.

Valuacije. Osnovna funkcionalnost klase kojom se predstavljaju valuacije je da odredi vrednost svake promenljive iz unapred zadatog skupa promenljivih. Pošto se tokom SAT rešavanja koriste valuacije koje mogu da budu i parcijalne, vrednosti promenljivih mogu da budu tačne, netačne i nedefinisane — mogu da budu tipa `ExtendedBolean` tj. mogu da pripadaju skupu {`TRUE, FALSE, UNDEF`}. Sa obzirom da se valuacije grade inkrementalno, tj. da se proširuju i smanjuju dodavanjem i uklanjanjem literala, literali se smeštaju u niz. Valuacije se ponašaju u LIFO maniru i metodama `push` i `pop` se elementi dodaju, odnosno uklanjaju iz niza čime se postiže funkcionalnost steka.

U sistemu ArgoSAT, valuacije su predstavljene klasom `Valuation`.

```
class Valuation {
public:
  void    push (Literal l);
  Literal pop  ();

  bool isTrue  (Literal l)  const;
  bool isFalse (Literal l)  const;
  bool isUndef (Literal l)  const;
  ExtendedBolean variableValue(Variable v) const;
```

```
private:
  std::vector<Literal> _literals;
  std::vector<ExtendedBoolean> _variableValues;
};
```

Pošto valuacije zahtevaju funkcionalnost steka, njihovi literali se pamte kao nizovi, tj. smešteni su u kolekciju `std::vector`. Međutim, s obzirom da je ispitivanje vrednosti promenljive u datoj valuaciji jako često korišćena operacija, njena efikasnost je izuzetno značajna za efikasnost celog rešavača. Iz tog razloga se, pored pomenutog niza literala, vrši indeksiranje vrednosti svih promenljivih, odnosno kreira se i održava niz koji kodira preslikavanje promenljivih u njihove vrednosti. Ovim se donekle usložnjavaju i interfejs i implementacija klase. Naime, neophodno je da valuacija poznaje broj promenljivih koje se javljaju u formuli koja se rešava. Pretpostavljeno je da su valuacije sve vreme konzistentne, odnosno da ne sadrži dva međusobno suprotna literala. Takođe, neophodno je sve vreme vršiti sinhronizaciju niza literala sa indeksom vrednosti promenljivih.

Označene valuacije. Kao što je već opisano, označene valuacije predstavljaju posebne vrste valuacija. Pored osnovne funkcionalnosti valuacije, označene valuacije razlikuju dve vrste literala: pretpostavljene i izvedene. Prilikom dodavanja literala u označenu valuaciju potrebno je naglasiti da li se literal dodaje kao pretpostavljeni ili kao zaključeni literal. Pretpostavljeni literali dele valuaciju na nivoe i od označenih valuacija se zahteva da mogu da odrede nivo svakog svog literala (odnosno promenljive), kao i ukupan broj nivoa.

U sistemu ArgoSAT, označene valuacije su predstavljene klasom `Trail` koja nasleđuje klausu `Valuation`.

```
class Trail : public Valuation {
public:
  void     push             (Literal l, bool decision);
  unsigned currentLevel     ()   const;
  unsigned level            (Literal l) const;
  unsigned levelVariable    (Variable v) const;

private:
  std::vector<unsigned> _levels;
  std::vector<unsigned> _variableLevels;
}
```

Kako bi se poboljšala efikasnost, umesto eksplicitnog beleženja oznake vrste uz svaki literal, čuva se samo niz `_levels` koji beleži broj literala na svakom nivou. Takođe, efikasnosti radi, nivo svake promenljive je indeksiran i čuva se u nizu `_variableLevels`.

11.2 API rešavača

Rešavač je u sistemu ArgoSAT predstavljen klasom `Solver`.

Osnovna fukcionalnost koju ova klasa nudi svojim korisnicima (eng. Application Programming Interface) je data sledećim funkcijama:

```
class Solver {
 public:
    void addInitialClause(std::vector<Literal>& literals);
    void solve();
    bool isSatisfiable();
    const Valuation& getModel() const;
    ...
};
```

Ovim se omogućava zadavanje formule koja se rešava dodavanjem njenih klauza, zatim ispitivanje njene zadovoljivosti, očitavanje rezultata i očitavanje eventualnih modela. Rešavanje problema se vrši pozivom metode `solve`. Nakon završetka procesa rešavanja problema, status je moguće očitati korišćenjem metoda `isSatisfiable`, a ukoliko je formula zadovoljiva, njen model je moguće očitati korišćenjem metoda `getModel`. Rešavač se, dakle, iz ostatka sistema najčešće koristi na sledeći način:

```
Solver solver;
solver.addInitialClause(...);
...
solver.addInitialClause(...);
solver.solve();
if (solver.isUnsatisfiable())
      ... report unsatisfiability ...
else if (solver.isSatisfiable()) {
      model = solver.getModel();
      ... report satifiabilitiy ...
}
```

Primetimo da su rešavači usko vezani za formulu koja se rešava, tako da je za svako rešavanje nove formule neophodno kreirati novu instancu rešavača.

Umesto direktnog dodavanja klauza korišćenjem metode `addInitialClause`, korisnicima je ostavljena mogućnost korišćenja gotovih i implementacije novih generatora problema. Generatori problema imaju zadatak da inicijalizuju rešavač skupom promenljivih i klauza koje predstavljaju formulu čija se zadovoljivost ispituje. Svi generatori nasleđuju klasu `ProblemGenerator` sa sledećim interfejsom:

```
class ProblemGenerator {
 public:
  virtual void generateProblem(Solver* solver) = 0;
};
```

Najčešće korišćeni generator problema je `ProblemGeneratorDIMACS` koji čita specifikaciju iz datoteke u DIMACS formatu. U fazi razvoja koda intenzivno su korišćeni generatori koji generišu neke od poznatijih problema pravilne strukture (npr. `ProblemGeneratorNQueens`, `ProblemGeneratorPigeons`, `ProblemGeneratorSudoku`).

11.3 DPLL jezgro rešavača

Većina današnjih rešavača zasnovanih na DPLL algoritmu implicitno implementiraju sisteme promene stanja za SAT. Arhitektura koju opisujemo predlaže da se ova veza učini eksplicitnom.

Osnovna komponenta koja bi trebalo da bude izolovana u okviru rešavača bi trebalo da bude *DPLL jezgro* koje ima tačno funkcionalnost apstraktnih sistema promena stanja. Interfejs ovakve komponente bi trebalo ekplicitno da odgovara pravilima prelaska između stanja (povratnim skokovima, učenju itd.) i da podržava kako primenu ovih pravila tako i proveru uslova njihove primenjivosti. Implementacija ove komponente bi trebalo da sadrži pregršt pametnih implementacionih tehnika i trikova niskog nivoa (kao što je, na primer, shema dva posmatrana literala), kako bi primena pravila bila što efikasnija. Na ovaj način, najkompleksniji delovi rešavača ostaju izolovani u okviru implementacije komponente koja čini jezgro. Ukoliko se napravi implementacija jezgra na čiju ispravnost je moguće osloniti se (eng. trusted implementation) ili ukoliko se izvrši formalna verifikacija implementacije jezgra, korektnost celog sistema se onda zasniva na korektnosti sistema promena stanja za SAT (pod uslovom da heurističke komponente zadovoljavaju određene dopunske, obično trivijalne, uslove).

Implementacija DPLL jezgra u sistemu ArgoSAT je izvršena u okviru klase `Solver` i direktno je zasnovana na pseudokod u opisanom u glavi 9.

Po uzoru na pseudokod iz glave 9, za svako pravilo je definisana funkcija koja se na njega odnosi. Tako je, na primer, funkcija koja implementira pravilo Decide implementirana na sledeći način:

```
void Solver::applyDecide() {
    Literal decisionLiteral =
              _literalSelectionStrategy->selectLiteral();
    assertLiteral(decisionLiteral, true);
    ...
}
```

Kako bi se dobio efektivan i efikasan sistem, način na koji se pravila prelaska stanja primenjuju, kao i redosled njihove primene je potrebno detaljno precizirati. Primetimo da je, analogno implementaciji u pseudokodu datoj u poglavlju 9, u implementaciji metode `applyDecide` izbor literala prepušten posebnoj heurističkoj komponenti. Ponovo po uzoru na pseudokodprikazan u glavi 9, osnovna strategija primene pravila je data kroz metodu `solve`. Aspekti primene pravila koji su nedvosmisleno optimalni su ,,tvrdo-kodirani" u okviru ove metode. Tako je, na primer, dat prioritet pravilu UnitPropagate u odnosu na pravilo Decide. Aspekti primene pravila za koje nije potpuno ustanovljena optimalna strategija su prepušteni heurističkim komponentama. Osnovni cilj koji naša arhitektura dostiže je razdvajanje ovakvih komponenti od jezgra DPLL algoritma, tako da ih je moguće eksterno održavati, bez potrebe za modifikovanjem samog jezgra.

```
void Solver::solve() {
  while(_satisfiable == UNDEF) {
    exhaustiveUnitPropagate();
```

```
    if (isConflicting()) {
      if (_trail.currentLevel() == 0) {
                _satisfiable = FALSE;
      } else {
                applyConflict();
                applyExplainUIP();
                applySubsumption();
                applyBackjump();
      }
    } else {
      if (_restartStrategy->shouldRestart()) {
                applyRestart();
                applySimplify();
      }

      if (_forgetStrategy->shouldForget()) {
                applyForget();
      }

      if (!allVariablesAssigned()) {
                applyDecide();
      } else {
                _satisfiable = TRUE;
                generateModel();
                backtrackToLevel(0);
      }
    }
  }
}
```

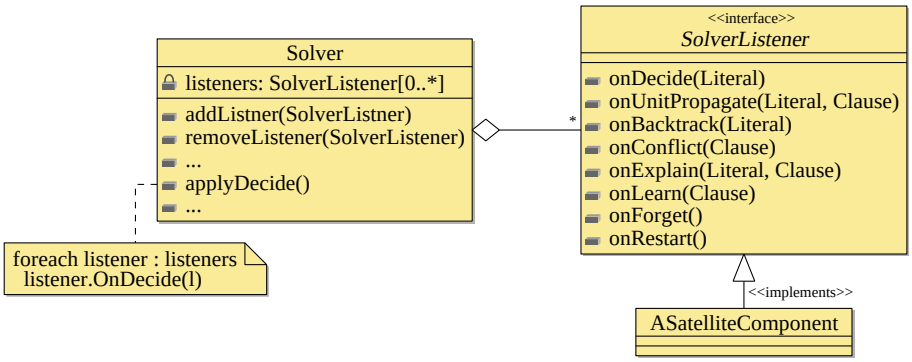
11.3.1 Komunikacija između jezgra i satelitskih komponenti

Interfejs jezgra DPLL mora da omogući komunikaciju sa satelitskim komponentama.

Prvo, unutrašnje stanje rešavača (označena valuacija M i formula F podeljena na početne i naučene klauze) mora da bude dostupno (isključivo za čitanje (eng. read only)):

```
class Solver {
public: ...
  const Trail& getTrail() const;
  const std::vector<const Clause*>& getInitialClauses () const;
  const std::vector<const Clause*>& getLearntClauses  () const;
};
```

Dalje, satelitske komponente moraju biti obaveštavanje o svim relevantnim akcijama koje sam rešavač preduzima. Pogodan način da se ovo realizuje je korišćenjem obrasca *Posmatrač (eng. Observer)* [GHJ+95]. UML dijagram koji prikazuje mehanizam obaveštavanja je prikazan na slici 11.2.



Slika 11.2: UML dijagram koji prikazuje komunikaciju između rešavača i satelitskih komponenti

U ovom slučaju, svi objekti koji su zainteresovani za praćenje akcija rešavača bi trebalo da pripadaju klasi koja implementira interfejs `SolverListener`. Ovaj interfejs direktno odgovara pravilima prelaska iz [KG07]:

```
class SolverListener {
 public:
  virtual void onDecide    (Literal l)                {}
  virtual void onPropagate (Literal l, Clause* clause) {}
  virtual void onBacktrack (Literal l)                {}
  virtual void onConflict  (Clause* conflictClause)   {}
  virtual void onExplain   (Literal l, Clause* clause) {}
  virtual void onLearn     (Clause* clause)           {}
  virtual void onForget    ()                         {}
  virtual void onRestart   ()                         {}
};
```

Posmatrači bi trebalo da se registruju rešavaču pozivajući njegov metod `addListener` i ovo se obično radi u okviru njihovog konstruktora. Posmatrači mogu i da zahtevaju da prestanu da budu obaveštavani o ponašanju rešavača pozivajući njegov metod `removeListner` i ovo se obično radi u okviru njihovog destruktora.[2]

```
class Solver {
public: ...
  void addListener    (SolverListener* listener) const;
  void removeListener (SolverListener* listener) const;
private: ...
  std::vector<SolverListener*> _listeners;
};
```

Na svim mestima u kodu DPLL jezgra rešavača koja odgovaraju primenama pravila prelaska između stanja, svi registrovani posmatrači se obaveštavaju o

[2]U nastavku teksta, konstruktori i destruktori klasa neće biti prikazivani, zbog toga što obično implementiraju trivijalne operacije (na primer registrovanje posmatrača, alociranje memorije, inicijalizaciju polja objekata itd.)

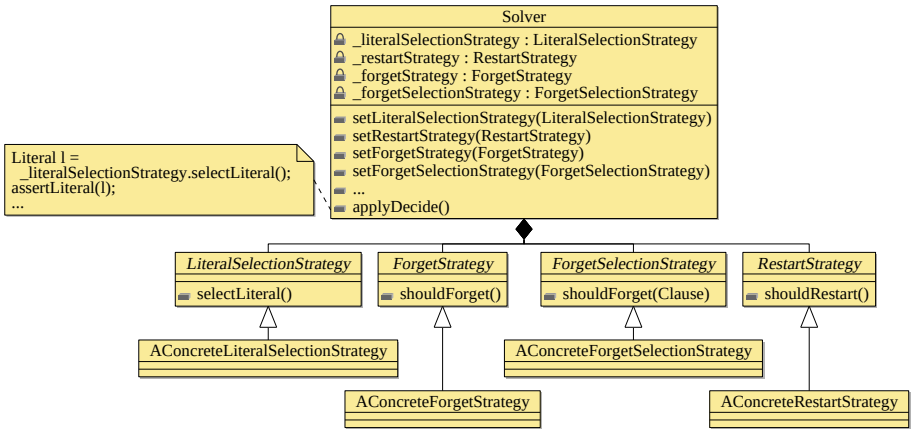
primeni pravila kako bi mogli da preduzmu odgovarajuće akcije. Tako je, u implementaciju metoda `applyDecide`, potrebno još dodati kôd koji vrši obaveštavanje posmatrača.

```
void Solver::applyDecide() {
    Literal decisionLiteral =
              _literalSelectionStrategy->selectLiteral();
    assertLiteral(decisionLiteral, true);

    for (li = _listeners.begin(); li != _listeners.end(); li++)
      (*li)->onDecide(decisionLiteral);
}
```

11.4 Heurističke komponente

Većina modernih rešavača se zasniva na istom algoritmu i najveće razlike među njima potiču od korišćenja različitih heurističkih komponenti. S obzirom na činjenicu da dobar izbor heuristika može biti presudan faktor u efikasnosti rešavača, kao i da nijedna heuristika nije apriori najbolja za svaku formulu, mogućnost korišćenja različitih heuristika je veoma poželjna prilikom izgradnje SAT rešavača. Samim tim, prilikom implementacije je poželjno izdvajanje heurističkih komponenti u posebne module koji su nezavisni od centralnog DPLL algoritma.

Arhitektura koju opisujemo predlaže da se izdvajanje heurističkih komponenti iz jezgra rešavača i podrška za istovremeno postojanje više različitih implementacija neke heuristike bazira na korišćenjo projektnog obrasca *Strategy* [GHJ+95]. UML dijagram zasnovan na ovom obrascu koji prikazuje vezu između rešavača i heurističkih komponenti je prikazan na slici 11.3.



Slika 11.3: UML dijagram koji prikazuje vezu između rešavača i heurističkih komponenti

U okviru rešavača ArgoSAT, korišćene su četiri vrste strategija: strategija izbora literala `LiteralSelectionStrategy` koja precizira način primene

pravila Decide, strategija otpočinjanja iznova `RestartStrategy` koja određuje trenutak primene pravila Restart i dve strategije zaboravljanja — `ForgetStrategy` i `ForgetSelectionStrategy` koje određuju trenutak kada se otpočinje sa primenom pravila Forget i vrše odabir klauza koje se zaboravljaju.

U skladu sa ovim, rešavač se proširuje odgovarajućim poljima i metodama za postavljanje konkretnih strategija.

```
class Solver {
   ...
public:
   void setLiteralSelectionStrategy(
           LiteralSelectionStrategy* literalSelectionStrategy);
   void setForgetStrategy(
           ForgetStrategy* forgetStrategy);
   void setForgetSelectionStrategy(
           ForgetSelectionStrategy* forgetSelectionStrategy);
   void setRestartStrategy(
           RestartStrategy* restartStrategy);

private:
  LiteralSelectionStrategy* _literalSelectionStrategy;
  ForgetStrategy* _forgetStrategy;
  ForgetSelectionStrategy* _forgetSelectionStrategy;
  RestartStrategy* _restartStrategy;
};
```

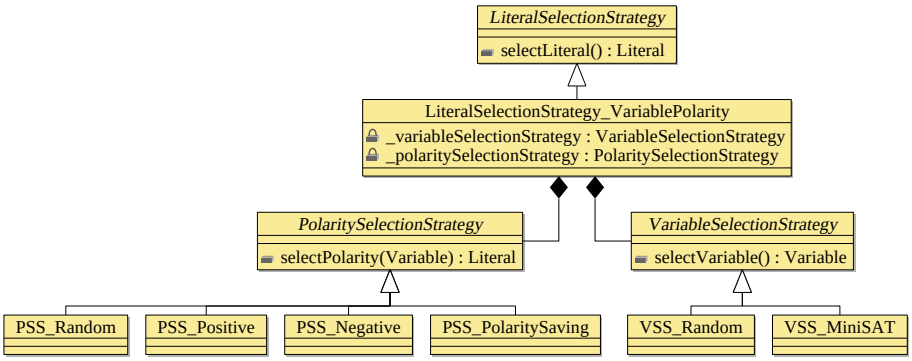
11.4.1 Izbor literala

Sve strategije za izbor literala su predstavljene apstraktnom klasom `Literal-SelectionStrategy`, čija je jedina funkcionalnost da na neki način izabere literal potreban za primenu pravila Decide.

```
class LiteralSelectionStrategy {
  public:
    virtual Literal selectLiteral() = 0;
};
```

Većina strategija za izbor literala prvo nezavisno odabere promenljivu pa tek nakon toga odaberu njen polaritet. UML dijagram koji prikazuje ovakvu arhitekturu je prikazan na slici 11.4.

```
class LiteralSelectionStrategy_VariablePolarity
   : public LiteralSelectionStrategy {
public:
  Literal getLiteral() {
    Variable var = _variableSelectionStrategy->getVariable();
    Literal lit  = _polaritySelectionStrategy->getLiteral(var);
    return lit;
  }
```



Slika 11.4: UML dijagram koji prikazuje strategije izbora literala koje nezavisno biraju promenljivu grananja, a tek nakon toga njen polaritet.

```
private:
  VariableSelectionStrategy* _variableSelectionStrategy;
  PolaritySelectionStrategy* _polaritySelectionStrategy;
};
```

```
class VariableSelectionStrategy {
public:
   virtual Variable getVariable() = 0;
};
```

```
class PolaritySelectionStrategy {
public:
   virtual Literal getLiteral(Variable variable) = 0;
};
```

MiniSAT strategija za izbor literala. Za primer, pokazaćemo kako se veoma složena strategija za izbor promenljivih kao što je strategija korišćena u okviru rešavača MiniSAT, može jednostavno implementirati u predloženom okviru. Opis ove strategije je već dat u poglavlju 4.5.1. Njena implementacija će, naravno, biti potpuno odvojena od implementacije jezgra DPLL algoritma. Takođe, ovaj primer pokazuje kako je moguće napraviti jasnu razliku između funkcionalnosti heuristike višeg nivoa i tehnika njene implementacije nižeg nivoa, što znatno olakšava razumevanje kôda.

Osnovna ideja MiniSAT strategije za izbor promenljivih je da se dâ prioritet promenljivim koje su učestvovale u skorašnjim konfliktima. Ovo se postiže tako što se svakoj promenljivoj dodeli numerički *skor* koji meri njena aktivnost u konfliktima. Kada se vrši odabir promenljive za grananje, bira se promenljiva najveća među svim promenljivima koje su u tom trenutku nedefinisane u tekućoj označenoj valuaciji. Skorovi promenljivih se dinamično menjaju kada promenljive učestvuju u konfliktima. Prilikom svake primene pravila Conflict i pravila Explain, skor svake promenljive koja pripada klauzi se uvećava.

Pošto strategija teži da dâ prioritet skorašnjim konfliktima, prilikom svake primene pravila Conflict, skorovi svih promenljivih se umanjuju. Primetimo da pošto ova strategija mora da reaguje na akcije koje rešavač izvodi, neophodno je da implementira interfejs `SolverListener`.

Kako bi se optimizovalo pronalaženje promenljivih sa najvišim skorovima, promenljive se čuvaju u okviru *hîp (eng. heap)* strukture podataka uređene po aktivnostima promenljivih. Iz razloga efikasnosti, hip se ažurira ,,lenjo", tj., promenljive se smeštaju u hip svaki put kada se uklone iz označene valuacije, ali se ne uklanjaju iz hipa svaki put kada se dodaju u valuaciju. Ovaj višak promenljivih na hipu se filtrira u trenutku kada se vrši izbor promenljive i to samo dok se ne pronađe promenljiva sa najvišim skorom koja je nedefinisana u valuaciji.

Skorovi aktivnosti se čuvaju i održavaju u okviru posebne klase. Ova klasa čuva sve aktivnosti promenljivih i njen interfejs mora da podrži operaciju uvećavanja skora datoj promenljivoj, kao i umanjivanja skorova svih promenljivih istovremeno. Implementaciju ove klase je moguće napraviti na nekoliko različitih načina i jedan veoma efikasan način će biti prikazan u nastavku.

```
class MinisatVariableSelectionStrategy :
         public VariableSelectionStrategy,
         public SolverListener {
public:
   ...
   Variable getVariable() {
      Variable maxVar;
      do {
          maxVar = _activityHeap.pop_heap();
      } while (!(_solver.getTrail().isUndefVariable(maxVar)));
      return maxVar;
   }

    void onConflict(Clause* clause) {
      _activities.decayAll();
      bumpVariablesInClause(clause);
    }

    void onExplain(Literal literal, Clause* clause) {
      bumpVariablesInClause(clause);
    }

    void onBacktrack(Literal literal) {
      Variable variable = getVariable(literal);
      if (!_activityHeap.contains(variable))
          _activityHeap.push_heap(variable);
    }
```

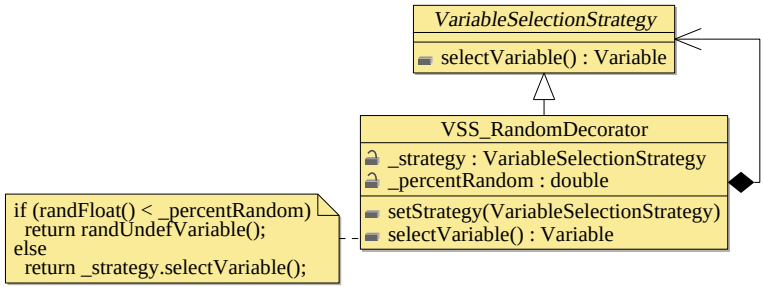
```
private:
    void init() {
      for (Variable var = 0; var < _solver.numVars(); var++)
         _activityHeap.push_heap(var);
    }

    void bumpVariablesInClause(Clause* clause) {
      for (lit = clause->begin(); lit != clause->end(); lit++)
         bumpVariableActivity(getVariable(*lit));
    }

    void bumpVariableActivity(Variable variable) {
       _activities.bump(variable);
       if (_activityHeap.contains(variable))
          _activityHeap.increase(variable);
    }

    const Solver& _solver;
    Activities _activities;
    Heap<Variable, Activities::Comparator> _activityHeap;
};
```

Implementacija klase `Activities` koristi pametnu tehniku uvedenu u rešavaču MiniSAT. Prvo, skorovi aktivnosti se beleže kao brojevi u pokretnom zarezu. Kako bi se dobilo na efikasnosti, kada se zahteva umanjivanje skorova svih promenljivih, oni se ne umanjuju efektivno, već se samo povećava tzv. *faktor uvećanja* za koji se aktivnosti uvećavaju naredni put kada se uvećanje zatraži. Na ovaj način, relativni odnosi između skorova ostaju nepromenjeni, ali je implementancija, naravno, mnogo brža. Ipak, postoji jedna začkoljica: pošto se faktor uvećanja eksponencijalno uvećava, postoji realna mogućnost da dođe do prekoračenja opsega zapisa brojeva u pokretnom zarezu. Kako bi se ovo sprečilo, svi skorovi se s vremena na vreme skaliraju i smanjuju.

```
class Activities {
public:
  Activities(double bumpAmount, double decayFactor,
             size_t numVars)
    : _bumpAmount(bumpAmount), _decayFactor(decayFactor) {
       _activities.resize(numVars);
       _activities.assign(numVars, 0.0);
  }

  void bump(Variable variable) {
    _activities[variable] += _bumpAmount;
    if (_activities[variable] > MAX_ACTIVITY)
      rescaleActivities();
  }
```

```
  void decayAll() {
    _bumpAmount *= _decayFactor;
  }

  double getActivity(Variable variable) const {
     return _activities[variable];
  }
  ...

private:
  void rescaleActivities() {
    for (Variable var = 0; var < _activities.size(); var++)
           _activities[var] *= 1.0/MAX_ACTIVITY;
    _bumpAmount *= 1.0/MAX_ACTIVITY;
  }
  std::vector<double> _activities;
  double _bumpAmount;
  double _decayFactor;
};
```

Randomizacija. Određene studije pokazuju da dodavanje odredjenog, malog procenta slučajnih odluka prilikom izbora promenljive grananja može da značajno poboljša efikasnost rešavača [GSK98]. Pošto je ovo moguće uraditi za bilo koju fiksiranu strategiju izbora promenljivih, pogodan način da se ovo ,,ukrašavanje" uradi je projektni obrazac *Dekorator (eng. Decorator)* [GHJ+95]. UML dijagram koji ilustruje ovu tehniku je prikazan na slici 11.5.



Slika 11.5: UML dijagram koji prikazuje obogaćivanje strategije izbora promenljivih dodavanjem određenog procenta slučajnih odluka.

```
class VariableSelectionStrategyRandomDecorator
       : public VariableSelectionStrategy {
public:
    ...
```

```
    Variable selectVariable() {
       if (randFloat() <= _percentRandom)
          // make random decision return
          randUndefVariable();
       else
          // fallback to default return
         _strategy->selectVariable();
    }

private:
    VariableSelectionStrategy* _strategy;
    float _percentRandom;
};
```

Izbor polariteta literala. Što se strategija za izbor polariteta literala date promenljive tiče, najelementarnije i najjednostavnije za implementaciju su strategije koje sve vreme dodeljuju jedan isti polaritet (bilo pozitivan, bilo negativan) i strategija koja polaritet odabire na slučajan način. Kao primer netrivijalne politike izbora polariteta, prikazaćemo implementaciju strategije poznate pod imenom *keširanje polariteta (eng. phase caching)*. U okviru ove strategije, svakoj promenljivoj se dodeljuje željeni polaritet. Kada god se literal doda u tekuću označenu valuaciju (bilo kao pretpostavljeni, bilo kao izvedeni literal), njegov polaritet definiše budući željeni polaritet njegove promenljive. Kada se literal ukloni iz tekuće označene valuacije (tokom povratnih skokova ili otpočinjanja iznova), njegov željeni polaritet se ne menja.

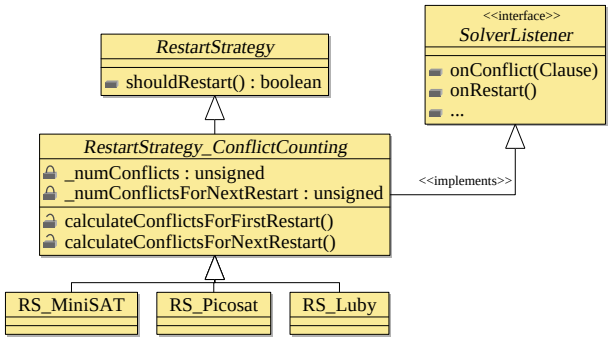
```
class PolaritySelectionStrategyCaching :
   public PolaritySelectionStrategy, public SolverListener {
public:
  ...

  Literal getLiteral(Variable variable) {
    return Literal(variable, _preferredPolarity[variable]);
  }

  void onAssert(Literal literal) {
    _preferredPolarity[getVariable(literal)] =
         isPositive(literal);
  }

private:
  std::vector<bool> _preferredPolarity;
};
```

11.4.2 Otpočinjanje iznova

Jedina funkcionalnost koju bi *strategija otpočinjanja iznova* trebalo da pruži je određivanje da li je u nekom trenutku potrebno izvršiti otpočinjanje rešavanja iznova.

```
class RestartStrategy {
public:
   virtual bool shouldRestart() = 0;
};
```

Većina strategija koje se u praksi koriste se zasnivaju na brojanju konflikata i vrše otpočinjanje iznova kada broj konflikata dostigne određenu vrednost. Ovu zajedničku funkcionalnost je poželjno izdvojiti u međuklasu. UML dijagram koji prikazuje ovakvu arhitekturu je prikazan na slici 11.6.



Slika 11.6: UML dijagram koji prikazuje strategije otpočinjanja iznova.

```
class RestartStrategyConflictCounting :
   public RestartStrategy, public SolverListener {
public:
  void init() {
    _numRestarts = 0; _numConflicts = 0;
    calculateConfilctsForFirstRestart();
  }
  void onConflict(Clause* conflictClause) {
    _numConflicts++;
  }
  void onRestart() {
    _numRestarts++; _numConflicts = 0;
    calculateConfilctsForNextRestart();
  }
  bool shouldRestart() {
    return _numConflicts >= _numConflictsForNextRestart;
  }
 protected:
  virtual void calculateConflictsForFirstRestart() = 0;
  virtual void calculateConflictsForNextRestart() = 0;
  size_t _numRestarts;
  size_t _numConflicts;
  size_t _numConflictsForNextRestart;
};
```

Kako bi se dobile konkretne strategije, potrebno je jedino specifikovati funkciju koje određuju broj konflikata potrebnih za prvo otpočinjanje iznova, kao i funkciju koja određuje broj konflikata potrebnih za svako sledeće otpočinjanje iznova.

Rešavač MiniSAT [ES04] koristi sledeću strategiju:

```
class RestartStrategyMinisat :
  public RestartStrategyConflictCounting {
  ...

  void calculateConflictsForFirstRestart() {
    _numConflictsForNextRestart = _numConflictsForFirstRestart;
  }

  void calculateConflictsForNextRestart() {
    _numConflictsForNextRestart *= _restartInc;
  }

private:
  unsigned _numConflictsForFirstRestart;
  float _restartInc;
};
```

Rešavač Picosat [Bie08] koristi sledeću strategiju:

```
class RestartStrategyPicosat :
  public RestartStrategyConflictCounting {
...
  void calculateConfilctsForFirstRestart() {
    _numConflictsForNextRestart = _numConflictsForFirstRestart;
  }

  void calculateConfilctsForNextRestart() {
    if (_inner >= _outer) {
      _outer *= _restartInc;
      _inner = _numConflictsForFirstRestart;
    } else {
      _inner *= _restartInc;
    }
    _numConflictsForNextRestart = _inner;
  }
private:
  unsigned _numConflictsForFirstRestart;
  float _restartInc;
  unsigned _inner, _outer;
};
```

Lubijeva optimalna strategija se može kodirati na sledeći način:

```
class RestartStrategyLuby
   : public RestartStrategyConflictCounting {
...
    void calculateConfilctsForFirstRestart() {
      _numConflictsForNextRestart =
          luby_super(_numRestarts + 1)*_factor;
    }

    void calculateConfilctsForNextRestart() {
      _numConflictsForNextRestart =
          luby_super(_numRestarts + 1)*_factor;
    }

private:
    size_t luby_super(size_t i) {
      size_t power;
      size_t k;

      // let 2^k be the least power of 2 >= (i+1)
      k = 1;
      power = 2;
      while (power < (i + 1)) {
        k += 1;
        power *= 2;
      }
      if (power == (i + 1))
        return (power / 2);
      return (luby_super(i - (power / 2) + 1));
    }

  unsigned _factor;
};
```

11.4.3 Zaboravljanje

Iako pravilo Forget definiše samo zaboravljanje jedne klauze, zaboravljanje se obično vrši u grupama. Rešavač ArgoSAT i pseudokod na kome je on zasnovan od heurističkih strategija zaboravljanja zahtevaju da odrede trenutak kada je potrebno započeti iscrpnu primenu pravila Forget i zatim, kada ta primena započne, da odrede grupu klauza koje se istovremeno zaboravljaju. Ove dve funkcionalnosti su razdvojene u dve klase `ForgetStrategy` i `ForgetSelectionStrategy`, kako bi one mogle nezavisno da variraju.

Klasa `ForgetStrategy`, odnosno njena metoda `shouldForget`, daju odgovor na pitanje da li je potrebno otpočeti proces zaboravljanja. Njene naslednice obično vrše zaboravljanje u trenutku kada broj klauza premaši neki prag ili kada ukupno zauzeće memorije postane veće nego što je to poželjno.

```
class ForgetStrategy {
 public:
  virtual bool shouldForget() = 0;
};
```

Klasa `ForgetSelectionStrategy` i njena metoda `select` vrše odabir klauza koje je potrebno zaboraviti.

```
class ForgetSelectionStrategy {
 public:
  virtual void select(std::vector<Clause*>::iterator b_in,
                      std::vector<Clause*>::iterator e_in,
                      std::vector<Clause*>::iterator& b_out,
                      std::vector<Clause*>::iterator& e_out) = 0;
};
```

Metodi se prosleđuje (deo) niza klauza predstavljen iteratorima `b_in` i `e_in`, a ona vraća deo niza koji je potrebno zaboraviti određen iteratorima `b_out` i `e_out`, permutujući pri tom niz ukoliko je neophodno.

11.5 Ostale satelitske komponente

Jednom kada je jezgro DPLL algoritma učaureno u poseban modul sa jasnim i bogatim programskim interfejsom, moguće je proširivati osnovni SAT rešavač mnogim satelitskim komponentama koje vrše razne sporedne funkcije. U ovom poglavlju veoma sažeto ćemo nabrojati neke od njih.

Vođenje statistika. Statistike sakupljene tokom procesa rešavanja (na primer, broj pretpostavljenih literala, broj konflikata, tekuća brzina rešavanja merena kroz broj novododatih literala u valuaciju po sekundi) mogu da pruže dublji uvid u proces rešavanja i pomognu istraživanja na polju rešavanja SAT problema. Opisana arhitektura omogućava veoma jednostavnu implementaciju vođenja raznih vrsta statistika i ovo je moguće uraditi kroz satelitske komponente, ne dirajući jezgro samog rešavača.

Korisnički interfejs. Kako bi se korisnicima dao neki prikaz tekućeg stanja i napretka procesa rešavanja, poželjno je implementirati određenu vrstu logovanja i prikazivanja izlaza različitog nivoa detaljnosti. U okviru naše arhitekture, komponente korisničkog interfejsa je moguće implementirati potpuno nezavisno od samog SAT rešavača. Kao primer, pokazujemo kako je veoma jednostavno implementirati tekstualni izlaz koji pomaže da se prati koja pravila prelaska između stanja rešavač primenjuje tokom procesa rešavanja.

```
class DotLoggingSolverListener : public SolverListener {
public:...
  void onAssert(Literal l) {
        if (_solver.getTrail().currentLevel() == 0)
           output("z");
  }
```

```
  void onDecide(Literal l) {
        if (_decisions++ == 1000)
           output("."), _decisions = 0;
  }

  void onRestart() { output("r\n"); }

  void onForget () { output("f");   }
};
```

Ukoliko je potrebno prikazati i detaljnije informacije, poželjno je implementirati i grafički korisnički interfejs (GUI) i vizuelizaciju raspoloživih podataka. Na slici 11.7 je prikazan grafički korisnički interfejs rešavača ArgoSAT koji je, takođe, implementiran kao zasebna satelitska komponenta, bez modifikovanja jezgra samog rešavača. GUI je implementiran korišćenjem biblioteka gtkmm i glibmm.

Slika 11.7: Grafički korisnički interfejs rešavača ArgoSAT. Gornji panel prikazuje grafik na kome se vidi broj tekući broj klauza nakon svake primene zaboravljanja, dok donji panel prikazuje primenjena pravila prelaska (kodirana različitim bojama). Sa grafika se jasno očitava MiniSAT strategija zaboravljanja koja nakon svake primene pravila Restart uvećava dopušteni broj klauza u sistemu.

Generisanje dokaza. Kao što je već opisano u poglavlju 4.6, jedan od načina da se povećava stepen poverenja u rezultate rešavača je generisanje eksplicitnih dokaza nezadovoljivosti i njihova naknadna provera. Iako postoji nekoliko standardnih formata za dokaze, svi oni se mogu generisati kroz satelitske komponente. Naglasimo ipak, da su ipak potrebne određene modifikacije i samog jezgra rešavača, pre svega što se tiče tretmana literala nultog nivoa i izvođenja prazne klauze nakon ustanovljavanja nezadovoljivosti, o čemu je već donekle bilo reči u glavi 9.

12
Eksperimentalna evaluacija

Sistem ArgoSAT je intenzivno testiran na korpusima iskaznih formula koji su potekli sa SAT takmičenja (eng. SAT competition) održanih 2002. i 2007. godine[1] i SAT utrke (eng. SAT race) održane 2008. godine[2]. Svi eksperimenti su rađeni na IBM Cluster 1350 klaster računaru sa 32 procesora.

Prvi važan rezultat je da tokom svih eksperimenata ni na jednom pokretanju rešavača nije uočena njegova nesaglasnost što značajno opravdava napor koji je u okviru ove teze uložen u verifikaciju.

Eksperimenti su takođe za cilj imali da ocene koliko se gubi na efikasnosti na račun fleksibilnosti implementacije. Kako bi se ovo procenilo upoređen je rešavač ArgoSAT 1.0 (sa podrazumevanim parametrima koji odgovaraju podrazumevanim parametrima rešavača MiniSAT) sa jezgrom rešavača MiniSAT 2.0 (preprocesiranje formula i napredne tehnike minimalizacije konfliktih klauza su isključene jer nisu podržane u okviru rešavača ArgoSAT). Rešavano je 100 instanci iz dva skupa instanci korišćenih za takmičenje *SAT race 2008*, pri čemu je vreme raspoloživo za svaku instancu 20min. Rešavač ArgoSAT je rešio 79(=42+37) instance sa središnjim vremenom 93.2s, dok je rešavač MiniSAT rešio 80(=40+40) instance sa središnjim vremenom 69.9s. Takođe upoređena su vremena za rešavač ArgoSAT 1.0 sa ranijim verzijama istog sistema (u kojima su heuristike tvrdo kodirane, slično kao što je to urađeno u rešavaču MiniSAT). Fleksibilna implementacija se pokazala kao sporija, ali je razlika bila manja od 3% na svim instancama koje su korišćene za testiranje.

Prilikom korišćenja SAT rešavača, važno pitanje predstavlja izbor pogodnih heurističkih komponenti i njihovih odgovarajućih parametara. U [Nik08, NMJ09], proučavano je pitanje mogućnosti automatskog izbora pogodnih heurističkih politika unapred, na osnovu sintaksnih karakteristika formule koja se rešava. Istraživanje je kao eksperimentalnu platformu koristilo SAT rešavač ArgoSAT i uključivalo je ekstenzivno pokretanje rešavača za 60 različitih kombinacija heurističkih politika na skupu od 1964 formule korpusa sa SAT takmičenja 2002. Ukupno utrošeno procesorsko vreme je bilo duže od 1000 dana. Istraživanje je, kao prvo, pokazalo nekoliko očekivanih rezultata.

1. Vreme rešavanja za datu formulu značajno zavisi od izbora heuristike

[1] Korpusi su dostupni sa adrese `http://www.sat-competition.org`.

[2] Korpus je dostupan sa adrese `http://baldur.iti.uka.de/sat-race-2008/downloads.html`.

koja se koristi.

2. Vreme rešavanja za datu formulu (naročito ako je ona zadovoljiva) može da značajno da varira pod uticajem različitih pseudo-slučajnih faktora[3].

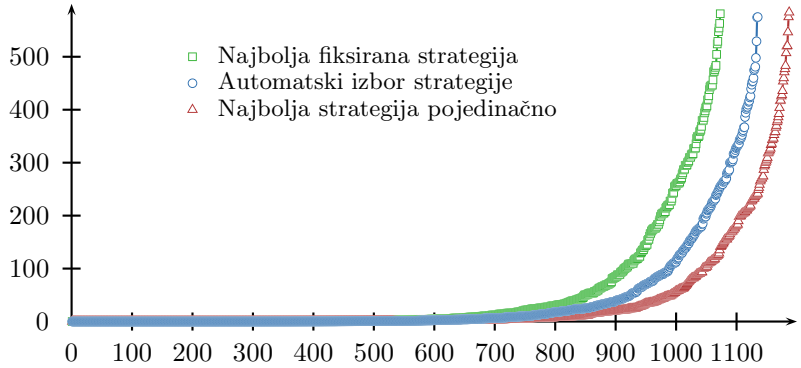
Iz ovog razloga, mnogo stabilniji i pouzdaniji rezultati se dobijaju ukoliko se razmatra ponašanje rešavača na širem korpusu formula nego ukoliko se razmatra ponašanje na pojedinačnim formulama.

U tabeli 12.1 su prikazani rezultati rešavača ArgoSAT za tri različite metodologije izbora heurističkih politika na celom korpusu SAT2002. Vreme raspoloživo za rešavanje svake pojedinačne formule bilo je 600s.

Metoda izbora strategija	Broj rešenih formula	Medijana vremena rešavanja
Najbolja fiksirana strategija	1073	207.4s
Automatski izbor strategije	1135	92.6s
Najbolja strategija pojedinačno	1187	45.8s

Tabela 12.1: Rezultati eksperimenata na korpusu SAT2002

Od 60 različitih heurističkih strategija, metoda pod imenom *Najbolja fiksirana strategija*, bira onu strategiju sa kojom se najveći broj formula može rešiti i tu strategiju koristi za rešavanje celokupnog korpusa. Metoda pod imenom *Najbolja strategija pojedinačno* za svaku formulu pojedinačno bira najbolju strategiju rešavanja. Jasno da ova politika daje donju granicu vremena rešavanja. Metoda pod imenom *Automatski izbor strategije* je originalni doprinos kolege Nikolića i ta metoda vrši izbor strategije automatski na osnovu sintaksnih karakteristika formule koja se rešava. Slika 12.1 grafički prikazuje odnos ove tri metode izbora heurističkih strategija.



Slika 12.1: Sortirana vremena rešavanja za tri različite metode izbora heurističkih strategija. Redni brojevi formula su prikazani na X osi, a vremena u sekundama na Y osi.

Rezultati iz radova [Nik08, NMJ09] pokazuju da postoji korelacija između sintaksnih svojstava formula i strategija pogodnih za njihovo rešavanje i da se

[3] U okviru istraživanja, vršena je permutacija redosleda klauza u okviru svake od formula, kao i permutacija redosleda literala u okviru klauza i rezultati su pokazali da vreme može značajno da varira samo pod uticajem ovog faktora

ta veza može automatski detektovati. Unapređenje koje se može dobiti izborom pogodnih heurističkih strategija je veoma značajno što opravdava polaznu pretpostavku u dizajnu rešavača ArgoSAT: dizajn i fleksibilnost programskog koda SAT rešavača predstavljaju izuzetno značajan aspekt ove vrste programa i nije redak slučaj da se zahvaljujući mogućnosti koegzistencije nekoliko heurističkih politika i lakoj proširivosti rešavača novim heurističkim politikama može dobiti mnogo više nego intenzivnom optimizacijom nekih tehničkih detalja same implementacije.

Radi provere kvaliteta metodologije automatskog izbora strategija rešavanja, rešavač ArgoSAT je nezavisno testiran i na korpusu SAT2007. Rezultati su prikazani u tabeli 12.2. Pošto je ovaj korpus značajno zahtevniji od korpusa SAT2002, nijedna metoda nije uspela rešiti polovinu formula te je umesto medijane prikazano vreme potrebno za rešavanje 20% formula iz korpusa.

Metoda izbora strategija	Broj rešenih formula	20-ti percentil vremena rešavanja
Najbolja fiksirana strategija	219	311.6s
Automatski izbor strategije	239	249.5s

Tabela 12.2: Rezultati eksperimenata na korpusu SAT2007

Iako sistem ArgoSAT nije imao za cilj mogućnost takmičenja u brzini sa konkurenstkim sistemima, prikazani eksperimenti ukazuju da uprkos tome sistem ArgoSAT pokazuje odlične rezultate. Naime, isti eksperiment je izršen za 5 komponentnih rešavača koji se koriste u okviru sistema Satzilla[XHH$^+$08] i sistem ArgoSAT je u ovim eksperimentima bio među prva tri rešavača (računajući broj rešenih formula za 600s korpusa SAT2007).

Deo III

Primene

13 Automatsko generisanje rasporeda časova

U ovoj glavi će biti prikazana tehnika automatskog generisanja rasporeda časova korišćenjem SAT rešavača. Svi uslovi i ograničenja koje raspored časova nameće se predstavljaju u vidu iskazne formula u KNF i nakon toga se, korišćenjem SAT rešavača, nalaze njeni modeli. Svaki pronađeni model određuje jedan ispravan raspored časova. Ukoliko je formula nezadovoljiva, neka ograničenja je neophodno oslabiti i proces se ponavlja sve dok se ne pronađe rešenje.

Iskustvo pokazuje da različite obrazovne institucije nameću veoma širok spektar ograničenja koje je potrebno imati u vidu prilikom izrade rasporeda časova. Moga od ovih ograničenja su obavezna i moraju biti zadovoljena kako bi raspored časova bio ispravan. Zato je neophodno imati sistem za generisanje rasporeda koji bi bio dovoljno opšt da omogući korisnicima da iskažu veoma različite vrste zahteva. Iako se sistem koji opisujemo ne može smatrati univerzalnim alatom za generisanje svih vrsta rasporeda časova, biće pokazano da se on veoma jednostavno može prilagođavati tako da omogući formulisanje veoma raznorodnih ograničenja — često mnogo raznorodnijih nego što većina alternativnih pristupa podržava.

13.1 Opis problema

Problem raspoređivanja časova zahteva dodeljivanje vremenskih perioda svakom od unapred zadatih časova, poštujući pri tome unapred zadata ograničenja. *Problem raspoređivanja časova sa salama* dodatno zahteva da se svakom času dodeli i sala u kojoj će se čas održavati, poštujući pritom unapred zadata ograničenja.

Veoma važna pretpostavka u pristupu koji će biti opisan je da se raspored pravi na bazi nastavnih nedelja, odnosno da je raspored nastave u svim radnim nedeljama u semestru tokom kojih se nastava održava identičan. Radna nedelja se sastoji od nekoliko *radnih dana*. Svaki radni dan je podeljen na veći broj *termina* jednake dužine. *Časove* je potrebno rasporediti tačno u ovako definisane termine. Napomenimo da časovi mogu da traju i duže od jednog termina. Svaki čas podrazumeva da se predaje neki *predmet*. Čas drži jedan ili više *nastavnika*. Času prisustvuje jedna ili više *grupa* studenata. Pretpostavlja se da su grupe, nastavnici i svi časovi koje je potrebno rasporediti svakoj od

grupa i svakom od nastavnika unapred poznati[1].

Ostala ograničenja koja raspored časova mora da zadovolji se razlikuju od institucije do institucije. Ipak, moguće je uočiti neke vrste zahteva svojstvene većini institucija za koje se konstruiše raspored časova. Ovo omogućava korišćenje istog kodiranja i korišćenje istog alata za pravljenje rasporeda u nekoliko različitih institucija. Uočene su dve vrste uslova. *Uslovi korektnosti* su neophodni za ispravnost rasporeda i neophodno je da svi budu zadovljeni kako bi uopšte bilo moguće održavati nastavu. *Uslovi udobnosti* predstavljaju želje nastavnog osoblja i studenata. Neke od ovih uslova je takođe potrebno smatrati *tvrdim* ograničenjima koje je neophodno zadovoljiti, dok se neka ograničenja smatraju *mekim* ograničenjima koja nije neophodno poštovati ukoliko to nije moguće.

Za rešavanje problema rasporeda časova se često koriste tehnike optimizacije i mnoge od njih uvode veliki broj mekih ograničenja i pokušavaju da pronađu rešenja koja ih zadovljavaju. Sa druge strane, pošto pristup koji ovde opisujemo koristi SAT rešavače, sva ograničenja rasporeda je neophodno predstaviti iskaznim klauzama i smatrati ih tvrdim ograničenjima. Zato se generiše iskazna formula koja formuliše sve tvrde i što veći broj mekih početnih uslova. Ovo obezbeđuje da će svako rešenje koje SAT rešavač eventualno pronađe biti dovoljno dobro da se može smatrati konačnim rasporedom. Jedino u slučaju kada se pokaže da su sva formulisana ograničenja nezadovoljiva, manje važna meka ograničenja se uklanjaju. Alternativni pristup ovome je krenuti samo od tvrdih ograničenja, a zatim dodavati meka ograničenja sve dok se ne pronađe rešenje koje je dovoljno kvalitetno.

U nastavku će biti opisane osnovne vrste uslova korektnosti i udobnosti obrađivane u našem pristupu.

Uslovi korektnosti

- Svaki čas iz unapred zadate liste časova mora da bude raspoređen tačno jednom u rasporedu.
- Nastavnik ne može da predaje dva različita predmeta u istom terminu. Moguće je zahtevati da nastavnik predaje isti predmet nekolikim grupama studenata istovremeno.
- Grupa može da prisustvuje samo jednom času u datom terminu.
- Samo jedan nastavnik može držati čas u datoj sali u datom terminu.

Uslovi udobnosti

Zabranjeni i zahtevani radni sati

- Nastavnicima je dozvoljeno da iskažu termie u kojima ne mogu da drže nastavu. Ovi uslovi se zadaju na nekoliko različitih načina (npr. traži se da neko ne može da predaje ponedeljkom između 10h i 15h, ili da neko ne može da predaje utorkom, ili da neko ne može da predaje pre 10h i posle 18h nijednog dana).

[1] U literaturi je takođe opisan problem formiranja grupa studenata na osnovu predmeta koje su odabrali da slušaju.

- Nekim nastavnicima (npr. gostujućim profesorima, starijim profesorima) je dozvoljeno da eksplicitno navedu termine u kojima je potrebno rasporediti njihove časove (npr. sredom od 13h do 15h i petkom od 9h do 13h).

- Grupama je takođe dopušteno da imaju zabranjene termine (npr. studenti starijih godina ne smeju da imaju nastavu u terminima nekih seminara).

Preklapanje grupa i preklapanje nastavnika.

- Moguće je zahtevati da različite grupe nemaju nastavu u istim terminima (npr. pošto studenti druge godine moraju da slušaju neke časove iz prve godine, studenti druge i prve godine ne smeju da imaju istovremeno nastavu).

- Neki od nastavnika zahtevaju da nemaju nastavu u istim danima kao i neke njihove kolege (npr. ako u laboratoriji u svakom trenutku mora da bude prisutan jedan od dva nastavnika, tako da ne smeju istovremeno da budu zauzeti u nastavi).

- Neki nastavici zahtevaju da imaju nastavu istovremeno (ili barem u istim danima) kao neke njihove kolege (npr. ako žele da organizju sastanke pre ili posle nastave u tim danima).

Broj nastavnih dana.

- Neki nastavnici traže da im se časovi rasporede u dat broj radnih dana (npr. samo dva dana u nedelji, bez obzira koji su to dani i koliko časova imaju u tim danima jer žele da se bave istraživačkim radom ostalim danima).

- Zakon može da zahteva da nastavnici u srednjim školama imaju nastavu svaki radni dan u nedelji, tj. da predaju pet radnih dana.

Trajanje nastavnog dana.

- Grupe učenika/studenata mogu da imaju najviše N (obično 7) radnih sati u toku dana, uključuju i pauze u nastavi. Izuzeci od ovog pravila su dozvoljeni samo ako to uprava institucije eksplicitno zatraži.

- Nastavnici mogu da imaju najviše N (obično 6) nastavnih sati u toku dana, uključujući pauze. Izuzeci od ovog pravila su dozvoljeni samo ako to uprava institucije eksplicitno zatraži.

- Zahteva se da i profesori i nastavnici imaju bar dva časa u danima u kojima imaju nastavu, kako ne bi morali da dolaze na fakultet samo zbog jednog časa.

Pauze u toku nastave.

- Nastavnici obično zahtevaju da nemaju pauze u između časova. Ipak, neki nastavnici eksplicitno zahtevaju da imaju pauze.
- Odeljenja u srednjim školama ne smeju da imaju pauze u toku nastave (osim odmora koji su već uračunati u termine nastave).
- Grupama na fakultetima je obično dozvoljeno da imaju pauze, ali je njihov broj i trajanje obično ograničeno.
- Nekada se eksplicitno zahteva da grupe studenata imaju pauze (npr. ukoliko nastava traje veći broj termina u toku radnog dana, zahteva se pauza kako bi studenti mogli da ručaju).

Časovi sa fiksiranim ili zabranjenim terminima.

- U srednjoj školi se obično zahteva da se neki zahtevniji predmeti (npr. matematika) ne stavljaju u poslednji termin u toku radnog dana jer su učenici tada umorni i teže prate zahtevne sadržaje.
- Neke predmete je neophodno stavljati kao prve ili poslednje časove u toku dana (npr. za neki predmet je potrebno rasporediti časove u tri uzastopna termina i to ili u prva tri ili u poslednja tri termina u toku radnog dana).
- Neke izborne predmete ne pohađaju svi studenti. Stoga je njih potrebno rasporediti u početne ili krajnje termine u toku radnog dana, kako studenti koji ne pohađaju ove predmete ne bi imali pauzu u terminu njihovog održavanja. Ovi časovi mogu biti raspoređeni u bilo koji od termina u toku dana, sve dok ta grupa studenata nema nastavu bilo pre, bilo posle njih.

Uzastopni dani.

- Neki predmeti se drže nekoliko časova u toku nedelje (npr. predmet koji ima četiri časa u toku nedelje je potrebno rasporediti u obliku $2 + 2$, tj. dva časa je potrebno staviti u jedan, a dva časa u drugi dan. Iz pedagoških razloga, zahteva se da ovo ne budu dva uzastopna dana.
- Neki od nastavnika koji ne predaju svaki dan u nedelji zahtevaju da im se časovi rasporede u uzastopne radne dane. Na ovaj način, ukoliko uzimaju slobodne dane, ti slobodni dani mogu da se spoje sa vikendom bez remećenja uobičajenog procesa nastave.

Menjanje smena i zgrada.

- U institucijama koje rade po smenama, ponekad se zahteva da nastavnici ne rade u različitim smenama u toku istog dana.
- U ustanovama u kojima se nastava odvija u nekoliko udaljenih zgrada, ni studenti ni nastavnici ne bi trebalo da imaju nastavu u različitim zgradama u toku jednog dana.

13.2 SAT kodiranje

U ovom poglavlju će biti opisano kodiranje uslova rasporeda časova u obliku iskazne formule. Prvo će biti prikazano kodiranje osnovne varijante problema rasporeda časova, a nakon toga će biti opisana dva različita načina rešavanja problema raspoređivanja časova sa salama — naivno i naprednije kodiranje.

13.2.1 Osnovno kodiranje

Za svaku kombinaciju časa i termina uvodi se posebna iskazna promenljiva. Ove promenljive čine *osnovne promenljive* kodiranja i njihove istinitosne vrednosti određuju celokupan raspored časova. Iako je teorijski moguće sve uslove rasporeda časova izraziti direktnim korišćenjem isključivo osnovnih promenljivih, da bi formulisanje uslova bilo pojednostavljeno, biće uvedeno i nekoliko vrsta pomoćnih iskaznih promenljivih. Veze između ovih osnovnih i izvedenih promenljivih kao i uslovi rasporeda će biti kodirani korišćenjem iskaznih klauza. Ovako formiran skup klauza čini iskaznu KNF formulu koja opisuje problem rasporeda časova i čiji su modele rešenje ovog problema. U tekstu će neki uslovi biti navedeni u formi ekvivalencija i implikacija kako bi se olakšala čitljivost. Ovako formulisani uslovi se korišćenjem jednostavnih iskaznih ekvivalencija trivijalno prevode u klauzalnu formu.

Promenljive i njihove međusobne veze

Radno vreme. Skup radnih dana u instituciji u kojoj se pravi raspored časova će biti označen sa days. Svaki radni dan je podeljen na nekoliko termina iste dužine (npr. jednočasovni termini na fakultetima ili četrdesetpetominutni termini u srednjim školama). Skup termina za dan $d \in \mathsf{days}$ će biti označavan sa $\mathsf{periods}(d)$.

Časovi. Svi časovi koje je potrebno rasporediti će biti označavani četvorkom $tsgn$ koja označava da nastavnik (teacher) t predaje predmet (subject) s grupi (group) g n-ti put u toku nedelje. Svaki čas ima određeno trajanje koje će biti označeno sa $\mathsf{duration}(tsgn)$ i izraženu u broju termina koje zauzima. Na primer, ukoliko nastavnik T predaje predmet S grupi G dva puta u nedelji, jedan put dva vezana časa, a drugi put 3 vezana časa, njegovi časovi bi bili označeni sa $TSG1$ i $TSG2$, pri čemu je $\mathsf{duration}(TSG1) = 2$ i $\mathsf{duration}(TSG2) = 3$. Lista časova datog nastavnika t će biti označena sa $\mathsf{lessons}(t)$. Lista časova za datu grupu g će biti označena sa $\mathsf{lessons}(g)$.

Osnovne promenljive. Promenljiva x'_{tsgndp} se definiše za svaki čas $tsgn$. Ona kodira uslov da čas $tsgn$ počinje u danu d i terminu p. Pošto se časovi mogu držati samo u toku radnog vremena, promenljive x'_{tsgndp} se definišu isključivo za termine p za koje važi $\min(\mathsf{periods}(d)) \leq p \leq \max(\mathsf{periods}(d)) - \mathsf{duration}(tsgn) + 1$. Ove promenljive su jedine *osnovne promenljive* u sistemu jer njihove vrednosti jednoznačno određuju ceo raspored.

Izvedene promenljive. Kako bi bilo moguće jednostavnije definisanje različitih vrsta ograničenja rasporeda, uvedeno je nekoliko vrsta *izvedenih promenljivih*.

Promenljiva x_{tsgndp} se definiše za svaki čas $tsgn$, svaki radni dan d i termin p. Ona kodira uslov da se čas $tsgn$ održava u danu d, u terminu p. Naredne implikacije[2] predstavljaju vezu između termina početka časa i termina njegovog održavanja:

$$x'_{tsgndp_1} \Rightarrow x_{tsgndp_2},$$

pri čemu je $d \in \mathsf{days}$, $\min(\mathsf{periods}(d)) \leq p_1 \leq \max(\mathsf{periods}(d)) - \mathsf{duration}(tsgn) + 1$ i $p_1 \leq p_2 \leq p_1 + \mathsf{duration}(tsgn) - 1$.

$$x_{tsgndp_2} \Rightarrow \bigvee_{\substack{p_2 - \mathsf{duration}(tsgn) + 1 \leq p_1 \leq p_2, \\ \min(\mathsf{periods}(d)) \leq p_1 \leq \max(\mathsf{periods}(d)) - \mathsf{duration}(tsgn) + 1}} x'_{tsgndp_1},$$

pri čemu je $d \in \mathsf{days}$ i $\min(\mathsf{periods}(d)) \leq p_2 \leq \max(\mathsf{periods}(d))$.

Promenljiva x_{tsgnd} se definiše za svaki čas $tsgn$ i svaki radni dan d. Ona predstavlja činjenicu da se čas $tgsn$ održava u danu d. Naredne implikacije povezuju ove promenljive sa promenljivim koje određuju termine dodeljene časovima:

$$x_{tsgndp} \Rightarrow x_{tsgnd},$$

za $p \in \mathsf{periods}(d)$.

$$x_{tgsnd} \Rightarrow \bigvee_{p \in \mathsf{periods}(d)} x_{tsgndp}$$

Takođe, naredne implikacije povezuju ove promenljive sa promenljivima koje određuju termine u kojima časovi počinju:

$$x'_{tsgndp} \Rightarrow x_{tsgnd},$$

za $p \in \mathsf{periods}(d)$.

$$x_{tsgnd} \Rightarrow \bigvee_{\min(\mathsf{periods}(d)) \leq p \leq \max(\mathsf{periods}(d)) - \mathsf{duration}(tsgn) + 1} x'_{tsgndp}$$

Primetimo da je jedna od prethodne dve grupe implikacija suvišna i nema je potrebe uključivati u formulu.

Promenljiva x_{tdp} se definiše za svakog nastavnika t, svaki radni dan d i termin p. Ona predstavlja činjenicu da nastavnik t ima neki čas u danu d i terminu p. Naredne implikacije povezuju ove promenljive sa promenljivim koje određuju termine dodeljene časovima:

$$x_{tsgndp} \Rightarrow x_{tdp},$$

pri čemu je $tsgn \in \mathsf{lessons}(t)$ i $\mathsf{lessons}(t)$ je lista svih časova $tsgn$ za nastavnika t,

$$x_{tdp} \Rightarrow \bigvee_{tsgn \in \mathsf{lessons}(t)} x_{tsgndp}.$$

[2] Primetimo da se implikacije trivijalno prevode u klauzalnu formu korišćenjem tautologije $p \Rightarrow q \equiv \neg p \vee q$.

Slično, promenljiva x_{gdp} se definiše za svaku grupu g, radni dan d i termin p. Njom se predstavlja činjenica da grupa g ima neki čas u danu d i terminu p. Naredne implikacije povezuju ove promenljive sa promenljivim koje određuju termine dodeljene časovima:

$$x_{tsgndp} \Rightarrow x_{gdp},$$

pri čemu važi $tsgn \in \mathsf{lessons}(g)$ i $\mathsf{lessons}(g)$ je lista svih časova dodeljenih $tsgn$ grupi g.

$$x_{gdp} \Rightarrow \bigvee_{tsgn \in \mathsf{lessons}(g)} x_{tsgndp}.$$

Promenljiva x_{td} se definiše za svakog nastavnika t i svaki radni dan d. Njom se predstavlja činjenica da nastavnik t ima neki čas u danu d. Tada važe naredne implikacije:

$$x_{tdp} \Rightarrow x_{td},$$

pri čemu je $h \in \mathsf{periods}(d)$.

$$x_{td} \Rightarrow \bigvee_{p \in \mathsf{periods}(d)} x_{tdp}.$$

Promenljiva x_{tp} se definiše za svakog nastavnika t i svaki termin p. Njom se predstavlja činjenica da nastavnik t ima časove u terminu p. Tada važe naredne implikacije:

$$x_{tdp} \Rightarrow x_{tp},$$

za $d \in \mathsf{days}$.

$$x_{tp} \Rightarrow \bigvee_{d \in \mathsf{days}} x_{tdp}$$

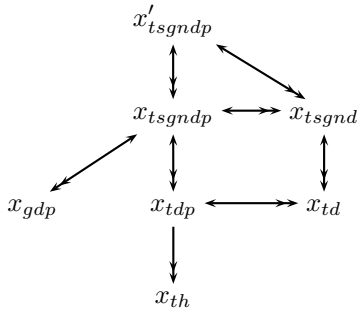
Poslednja vrsta implikacija se obično izostavlja jer se ove promenljive obično koriste da bi se određenim nastavnicima zabranio rad u određenim terminima (npr. mnogi nastavnici ne vole jutarnje časove). U tom slučaju, prva vrsta implikacija se bolje razume kada se razmatra njena kontrapozicija: $\overline{x}_{tp} \Rightarrow \overline{x}_{tdp}$.

Pošto su studenti obično na raspolaganju u svim terminima, promenljive x_{gd} i x_{gp} se ne uvode (međutim, moguće ih je uvesti na potpuno analogan način).

Promenljive i veze između njih su prikazane na slici 13.1.

Pauze u rasporedu. Kažemo da nastavnik ili (ili grupa) ima pauzu u rasporedu (eng. idle period) u nekom terminu ukoliko u nema čas u tom terminu, ali ima časove u tom danu pre i posle tog termina. Nakraća pauza u rasporedu je jedan termin, a najveća moguća pauza se javlja ukoliko neko ima časove samo prvi i poslednji termin u nekom danu.

Promenljiva i^k_{tdp} se definiše za svakog nastavnika t, dan d, termin p i broj k tdj. $1 \leq k \leq \mathsf{duration}(d) - 2$ i $\min(\mathsf{periods}(d)) + 1 \leq p \leq \max(\mathsf{periods}(d)) - k$. Ona predstavlja činjenicu da nastavnik t ima pauzu u nastavi dužine k termina



Slika 13.1: Promenljive i veze između njih.

u danu d, pri čemu pauza počinje u terminu p. Ove promenljive se definišu narednim ekvivalencijama.

$$i^k_{tdp} \Leftrightarrow \left(x_{td(p-1)} \ \wedge \bigwedge_{0 \le j < k} \overline{x}_{td(p+j)} \ \wedge \ x_{td(p+k)} \right)$$

Ekvivalencije se prevode u klauzalni oblik na sledeći način:

$$\begin{aligned} i^k_{tdp} &\Rightarrow x_{td(p-1)} \\ i^k_{tdp} &\Rightarrow \overline{x}_{td(p+j)}, \quad 0 \le j < k \\ i^k_{tdp} &\Rightarrow x_{td(p+k)} \end{aligned}$$

i

$$\left(\overline{x}_{td(p-1)} \ \vee \bigvee_{0 \le j < k} x_{td(p+j)} \ \vee \ \overline{x}_{td(p+k)} \right) \vee \overline{i^k_{tdp}}.$$

Promenljiva i^k_{td} se definiše za svakog nastavnika t, radni dan d i broj k tdj. $1 \le k \le \mathsf{duration}(d) - 2$. Ona predstavlja činjenicu da nastavnik t ima pauzu u rasporedu dužine k tokom dana d. Važe naredne implikacije:

$$i^k_{tdp} \Rightarrow i^k_{td},$$

za svaki termin p tdj. $\min(\mathsf{periods}(d)) + 1 \le p \le \max(\mathsf{periods}(d)) - k$ i

$$i^k_{td} \Rightarrow \bigvee_{\min(\mathsf{periods}(d))+1 \le p \le \max(\mathsf{periods}(d))-k} i^k_{tdp}.$$

Takođe, promenljiva i^k_t se definiše za svakog nastavnika t i broj k tdj. $1 \le k \le \mathsf{duration}(d) - 2$. Ona predstavlja činjenicu da nastavnik nekada ima pauzu u rasporedu dužine k. Važe naredne implikacije:

$$i^k_{td} \Rightarrow i^k_t,$$

za svaki $d \in \mathsf{days}$ i

$$i^k_t \Rightarrow \bigvee_{d \in \mathsf{days}} i^k_{td}.$$

Promeljiva i_{tdp} se definiše za svakog nastavnika t, radni dan d i termin p takve da važi $\min(\mathsf{periods}(d)) + 1 \leq p \leq \max(\mathsf{periods}(d)) - 1$. Ona predstavlja činjenicu da nastavnik t ima pauzu u rasporedu u danu d koja počinje u terminu p. Jasno je da za svako k takvo da je $1 \leq k \leq \max(\mathsf{periods}(d)) - p$ važi:

$$i^k_{tdp} \Rightarrow i_{tdp}$$

i

$$i_{tdp} \Rightarrow \bigvee_{1 \leq k \leq \max(\mathsf{periods}(d)) - p} i^k_{tdp}.$$

Sve promeljive i implikacije koje su u ovom poglavlju definisane za nastavnike se na potpuno analogan način mogu definisati i za grupe.

Zadavanje različitih uslova rasporeda

U nastavku će biti pokazano kako je moguće kodirati uslove rasporeda opisane u §13.1 korišćenjem promenljivih definisanih u §13.2.1. Bogat skup promenljivih koje su uvedene omogućava čak i definisanje dodatnih vrsta ograničenja koja nisu navedena u poglavlju §13.1, što ovaj pristup generisanja rasporeda časova čini veoma fleksibilnim. Svaki pojedinačni uslov rasporeda časova se predstavlja kroz jednu ili više iskaznih klauza. Ove klauze zajedno sa klauzama koje predstavljaju veze između promenljivih i koje su uvedene u poglavlju 13.2.1, čine konačnu iskaznu formulu čija se zadovljivost ispituje u cilju nalaženja rasporeda časova.

Za početak uvedimo nekoliko konstrukcija koje će nam omogućiti da jednostavnije zapišemo različite uslove rasporeda.

Označimo sa $\mathsf{single}(\{v_1, \ldots, v_k\})$ činjenicu da najviše jedna od promenljivih $v_1, \ldots, v_k$ može biti tačna. Trivijalan način da se kodira ovaj uslov je korišćenje kvadratnog broja klauza:

$$\mathsf{single}(\{v_1, \ldots, v_k\}) = \bigwedge_{1 \leq i < j \leq k} (\overline{v_i} \vee \overline{v_j}).$$

Ovaj koncept se može generalisati na proizvoljan broj *uslova kardinalnosti (eng. cardinality constraints)*. $\mathsf{cardinality}(\{v_1, \ldots, v_k\}) \leq m$ označava da najviše m promenljivih iz skupa $v_1, \ldots, v_k$ može istovremeno biti tačno. Jedan od efikasnih načina da se uslovi kardinalnosti opišu skupom iskaznih klauza je opisan u [Sin05].

Uslovi korektnosti.

- Svaki čas naveden u podeli nastave mora da bude prisutan u konačnom rasporedu. Pošto se svaki čas održava u nekom od radnih dana

  $$\bigvee_{d \in \mathsf{days}} x_{tsgnd}$$

  mora da važi za svaki čas $tsgn$. Ukoliko ovo važi, na osnovu implikacije

  $$x_{tsgnd} \Rightarrow \bigvee_{\min(\mathsf{periods}(d)) \leq p \leq \max(\mathsf{periods}(d)) - \mathsf{duration}(tsgn) + 1} x'_{tsgndp}$$

koji je već ranije uvedena, času će biti dodeljen bar jedan termin njegovog održavanja.

- Svaka čas naveden u podeli nastave treba da bude prisutan tačno jednom u konačnom rasporedu. Zato je početak svakog časa jednoznačno odredjen konačnim rasporedom i uslov

$$\mathsf{single}(\{x'_{tsgndp} \mid d \in \mathsf{days},\ p \in \mathsf{periods}(d)\})$$

mora da važi za svaki čas $tsgn$. Kako bi se smanjio ukupan broj klauza, ovaj uslov je moguće razbiti na sledeća dva:

$$\mathsf{single}(\{x_{tsgnd} \mid d \in \mathsf{days}\})$$

i za svaki $d \in \mathsf{days}$

$$\mathsf{single}(\{x'_{tsgndp} \mid p \in \mathsf{periods}(d)\}).$$

- Svaka grupa u jednom trenutku može da prisustvuje najviše jednom času. Dakle, za svaku grupu g, svaki dan $d \in \mathsf{days}$ i svaki termin $p \in \mathsf{periods}(d)$

$$\mathsf{single}(\{x_{tsgndp} \mid tsgn \in \mathsf{lessons}(g)\})$$

mora da važi.

- Svaki nastavnik može da drži najviše jedan čas istovremeno. Ipak, u nekim slučajevima se zahteva da se više različitih studentskih grupa (npr. $g_1, \ldots, g_k$) objedini i da zajedno slušaju predavanje. Uvaj zahtev se kodira ekvivalencijom

$$x_{tsg_1ndp} \Leftrightarrow x_{tsg_jndp}, \quad 1 < j \leq k.$$

U ovom slučaju $\overline{\mathsf{lessons}}(t)$ je skup koji sadrži tačno jednog predstavnika (e.g. tsg_1n) za svaku objedinjenu grupu studenata.

Dakle, pošto se zahteva da svaki nastavnik predaje samo jedan čas u datom terminu, za svakog nastavnika t, svaki $d \in \mathsf{days}$ i svaki $p \in \mathsf{periods}(d)$ uslov

$$\mathsf{single}(\{x_{tsgndp} \mid tsgn \in \overline{\mathsf{lessons}}(t)\})$$

mora da važi.[3]

Uslovi udobnosti.

Zabranjeni i zahtevani radni dani sati. Zabranjeni i zahtevani radni dani, odnosno termini za nastavnike se direktno kodiraju preko jednočlanih klauza koje sadrže (negirane) promenljive x_{tdp}, x_{td}, and x_{tp}. Slično, zabranjeni sati za grupe se kodiraju korišćenjem promenljivih x_{gdp}.

[3] Sličnu tehniku bi bilo moguće upotrebiti i u slučaju da se traži da više nastavnika predaje istovremeno istim studentima.

Preklapanje grupa i preklapanje nastavnika. Uslov da se rasporedi za dve grupe g_1 i g_2 ne preklapaju se kodira korišćenjem implikacija $x_{g_1dp} \Rightarrow \overline{x_{g_2dp}}$ i $x_{g_2dp} \Rightarrow \overline{x_{g_1dp}}$, za svaki radni dan d i termin p. Preklapanje nastavnih sati za nastavno osoblje se kodira na analogan način.

Broj nastavnih dana. Uslov da nastavnik t predaje tačno n dana nedeljno se može kodirati kao

$$\mathsf{cardinality}(\{x_{td} \mid d \in \mathsf{days}\}) \leq n \ \wedge \ \mathsf{cardinality}(\{\overline{x_{td}} \mid d \in \mathsf{days}\}) \leq |\mathsf{days}| - n.$$

Trajanje nastavnog dana. Trajanje radnog dana za grupe studenata se kodira uvođenjem promenljivih l^k_{gd} za svaku grupu g, dan d i broj $k \leq |\mathsf{periods}(d)|$. Promeljiva l^k_{gd} predstavlja činjenicu da se radni dan za grupu g prostire kroz bar k termina (uključujući i pauze u nastavi) u danu d. Definiše se preko:

$$x_{gdp} \wedge x_{gd(p+k-1)} \Rightarrow l^k_{gd},$$

za sve termine p takve da je $\min(\mathsf{periods}(d)) \leq p \leq \max(\mathsf{periods}(d)) - k + 1$, i

$$l^k_{gd} \Rightarrow \bigvee_{\min(\mathsf{periods}(d)) \leq p \leq \max(\mathsf{periods}(d)) - k + 1} (x_{gdp} \wedge x_{gd(p+k-1)}).$$

Uslov da se trajanje radnog dana za grupu ograniči na n termina periods se kodira kroz jednočlane klauze oblika $\overline{l^k_{gd}}$, za svako $k > n$.

Uslov da radni dan za grupu traje bar n termina se kodira kao $x_{gd} \Rightarrow l^n_{gd}$.

Trajanje radnog dana za nastavnike se kodira analogno.

Pauze u toku nastave. Uslov da nastavnik t ne sme da ima pauze u nastavi dužine k se zadaje korišćenjem jednočlane klauze koja sadrži literal $\overline{i^k_t}$.

Uslov da nastavnik t ne sme da ima više od jedne pauze u toku dana se može zadati kao

$$\mathsf{single}(\{i_{tdp} \mid \min(\mathsf{periods}(d)) + 1 \leq p \leq \max(\mathsf{hours}(d) - 1)\}).$$

Uslov da nastavnik t može da ima najviše n pauza u toku nastave u toku cele nedelje se može kodirati preko narednog uslova kardinalnosti

$$\mathsf{cardinality}(\{i_{tdp} \mid d \in \mathsf{days}, \ p \in \mathsf{periods}(d)\}) \leq n.$$

Druge vrste ograničenja za pauze u toku nastave, kao i ograničenja za pauze koje imaju grupe studenata se mogu kodirati na sličan način.

Časovi sa fiksiranim ili zabranjenim terminima. Uslov da čas mora da počne isključivo u terminu p_1, p_2, ..., ili p_n se kodira sa

$$x_{tsgnd} \Rightarrow x'_{tsgndp_1} \vee \ldots \vee x'_{tsgndp_n}.$$

Uslov da čas $tsgn$ mora biti prvi ili poslednji čas za grupu g u danu d, se kodira sa

$$x'_{tsgndp} \Rightarrow \left(\bigwedge_{\min(\mathsf{periods}(d)) \leq p' < p} \overline{x_{gdp'}} \right) \vee \left(\bigwedge_{p + \mathsf{duration}(tsgn) \leq p' \leq \max(\mathsf{periods}(d))} \overline{x_{gdp'}} \right).$$

Uzastopni dani. Ukoliko se želi zabraniti da se različiti časovi iz istog predmeta održavaju u uzastopnim danima, uslov

$$x_{tsgnd} \Rightarrow \overline{x_{tsg(n+1)(d+1)}}$$

mora da važi za svaki dan d osim za poslednji radni dan u nedelji.

Menjanje smena i zgrada. Pravljenje rasporeda za različite smene i različite zgrade je moguće praviti zasebno. U tom slučaju, u prvoj fazi se kreira raspored za jednu smenu (ili zgradu). Kada se u drugoj fazi kreira raspored za drugu smenu ili zgradu, svi uslovi koji se odnose na menjanje smena i zgrada se, korišćenjem rasporeda napravljenog u prvoj fazi, mogu predstaviti kao uslovi zabranjenih termina.

Kompleksnost kodiranja

Skupom promenljivih očigledno dominiraju promenljive x'_{tgsndp} i x_{tgsndp}. Zato, je veličina broja promenljivih $O(n_l \cdot n_d \cdot n_p)$, gde je n_l ukupan broj časova koje je potrebno rasporediti, n_d je broj radnih dana i n_p je broj termina u danu.

Uslovi tipa single se kodiraju kvadratnim skupom klauza, tako da oni dominiraju ukupnom veličinom skupa klauza tj. formule. Zahtev da se svaki čas u rasporedu raspored tačno jednom uvodi $O(n_l \cdot (n_d^2 + n_d \cdot n_p^2))$ klauza. Uslov da svaka grupa može da pohađa samo jedan čas u fiksiranom terminu uvodi $O(n_d \cdot n_p \cdot n_g \cdot {n_{gl}}^2)$ klauza, gde je n_g broj grupa, a n_{gl} broj časova po grupi. Slično, uslov da svaki nastavnik ima najviše jedan čas u fiksiranom terminu uvodi $O(n_d \cdot n_p \cdot n_t \cdot {n_{tl}}^2)$ klauza, gde je n_t broj nastavnika, a n_{tl} broj časova po nastavniku.

13.2.2 Naivno kodiranje raspoređivanja sala

U ovom i u narednom poglavlju će biti opisan način da se reši varijanta problema rasporeda časova u kome je pored termina časovima potrebno dodeliti i sale.

Osnovne promenljive. Jedan način da se kodiraju uslovi zauzeća sala je da se skup osnovnih promenljivih promeni tako da promenljive pored časa i termina kodiraju i salu u kojoj se čas održava. Pošto je poželjno da se sale ne menjaju tokom trajanja časa koji traje nekoliko vezanih termina, sala u kojoj se čas održava se je jedinstveno određena salom u kojoj je čas započeo. U tom slučaju, za svaki čas $tsgn$, dan d, termin p i salu r, uvodi se promenljiva $x'_{tsgndpr}$ koja označava da čas $tsgn$ počinje u danu d, terminu p i da se održava u sali r. Pošto se obično pretpostavlja da su sale međusobno različite, tj. da postoje časovi koji ne mogu biti raspoređeni u neke sale, mnoge od promenljivih $x'_{tsgndpr}$ nemaju smisla i mogu se izostaviti. Za svaki čas $tsgn$, rooms($tsgn$) označava skup sala u kojima čas $tsgn$ može biti raspoređen. Tada se promenljive $x'_{tsgndpr}$ uvode samo za one sale r koje pripadaju skupu rooms($tsgn$).

Izvedene promenljive. Izvedena promenljiva $x_{tsgndpr}$ se definiše za svaki čas $tsgn$, dan d, termin p i salu r koja se nalazi u rooms($tsgn$). Ove promeljivim

se označava da se čas $tsgn$ održava u danu d, terminu p u sali r. Pošto se zahteva da se sala ne menja tokom trajanja časova koji se protežu kroz nekoliko termina, važe naredne implikacije:

$$x'_{tsgndp_1r} \Rightarrow x_{tsgndp_2r},$$

gde je $d \in \mathsf{days}$, $\min(\mathsf{periods}(d)) \leq p_1 \leq \max(\mathsf{periods}(d)) - \mathsf{duration}(tsgn) + 1$ i $p_1 \leq p_2 \leq p_1 + \mathsf{duration}(tsgn) - 1$ i

$$x_{tsgndp_2r} \Rightarrow \bigvee_{\substack{p_2 - \mathsf{duration}(tsgn) + 1 \leq p_1 \leq p_2, \\ \min(\mathsf{periods}(d)) \leq p_1 \leq \max(\mathsf{periods}(d)) - \mathsf{duration}(tsgn) + 1}} x'_{tsgndp_1r},$$

za $d \in \mathsf{days}$ and $\min(\mathsf{periods}(d)) \leq p_2 \leq \max(\mathsf{periods}(d))$.

Za svakog nastavnika t, dan d, termin p i salu r, definiše se promenljiva x_{tdpr}. Ona predstavlja činjenicu da nastavnik t zauzima salu r u terminu p dana d.

Sve promenljive osim $x'_{tsgndpr}$ postaju izvedene promenljive i važe sledeće implikacije:

$$x'_{tsgndpr} \Rightarrow x'_{tsgndp}, \qquad x'_{tsgndp} \Rightarrow \bigvee_{r \in \mathsf{rooms}(tsgn)} x'_{tsgndpr},$$

$$x_{tsgndpr} \Rightarrow x_{tsgndp}, \qquad x_{tsgndp} \Rightarrow \bigvee_{r \in \mathsf{rooms}(tsgn)} x_{tsgndpr},$$

$$x_{tdpr} \Rightarrow x_{tdp}, \qquad x_{tdp} \Rightarrow \bigvee_{r \in \mathsf{rooms}} x_{tdpr}.$$

Promenljive i njihovi odnosi su prikazani na slici 13.2.

Uslovi ispravnosti. Za svaki dan d, termin p i salu r, uslovi ispravnosti zauzeća sala se kodiraja sa

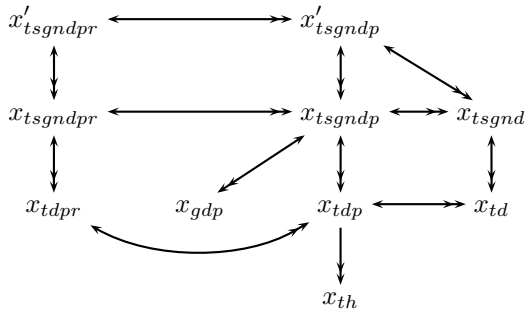
$$\mathsf{single}\{x_{tdpr} \mid t \in \mathsf{teachers}\}.$$

Pošto je jedna sala uvek dovoljna da bi se održao čas, za svaki dan d, termin p i salu r, važe sledeće implikacije:

$$\mathsf{single}\{x_{tdpr} \mid r \in \mathsf{rooms}\}.$$

Kompleksnost kodiranja

Kada se raspoređuju i sale, skupom promenljivih dominiraju promenljive $x'_{tsgndpr}$ i $x_{tsgndpr}$ i njihov broj je $O(n_l \cdot n_d \cdot n_p \cdot n_r)$, gde je n_l ukupan broj časova u rasporedu, n_d broj radnih dana, n_p broj termina u danu i n_r broj sala.



Slika 13.2: Promenljive i odnosi među njima - raspoređivanje sala

13.2.3 Kodiranje raspoređivanja sala zasnovano na uslovima kardinalnosti

U ovom poglavlju ćemo uvesti novi način raspoređivanja sala koji je zasnovan na korišćenju uslova kardinalnosti. U tom slučaju raspoređivanje se vrši u dve faze. U prvoj fazi, samo se termini dodeljuju časovima dok se u drugoj fazi dodeljuju sale. Tokom prve faze neopodno je nametnuti dodatna ograničenja kako bi dodeljivanje sala u drugoj fazi uopšte bilo moguće.

Razmotrimo za početak najjednostavniji slučaj alokacije sala. Pretpostavimo da ustanova ima n_r sala i da su sve sale jednake. Tada je u prvoj fazi raspoređivanja neophodno obezbediti da u svakom terminu najviše n_r nastavnika ima nastavu. Ovo se obezbeđuje uslovom

$$\mathsf{cardinality}(\{x_{tdp} \mid t \in \mathsf{teachers}\}) \leq n_r,$$

za svaki dan d i termin p.

Razmotrimo slučaj kada je za neke časove neophodno obezbediti specifične sale. Na primer, pretpostavimo dda se časovi iz programiranja mogu držati isključivo u salama koje imaju računare i da postoji n_{cl} takvih sala. Pored datog opšteg uslova za broj nastavnika po terminu, neophodno je još obezbediti da se u svakom terminu održava najviše n_{cl} časova programiranja. Ovo se kodira narednim uslovom

$$\mathsf{cardinality}(\{x_{tgsndp} \mid tsgn \in \text{časovi programiranja}\}) \leq n_{cl},$$

za svaki dan d i termin p.

Kada se prva faza završi, prelazi se na samo raspoređivanje sala. Ulaz u ovu fazu predstavljaju vrednosti svih promenljivih x_{tsgndp}. Tada se uvode promenljive $x_{tsgndpr}$ i x_{tdpr} i povezuju sa x_{tsgndp} na isti način kao i kod naivnog raspoređivanja sala, ali samo za one promenljive x_{tsgndp} i x_{tdp} koje su tačne u modelu koji je pronađen u prvoj fazi. Takođe, uslovi korektnosti su identični kao i kod naivnog raspoređivanja sala. Ukoliko se zahteva da se sale ne menjaju tokom časova koji traju više termina, zadaju se uslovi oblika $x_{tsgndpr} \Leftrightarrow x_{tsgnd(p+1)r}$, za svaki čas $tsgn$ kojemu su u danu d dodeljeni termini p i $p+1$ i za svaku salu r.

Kompleksnost kodiranja

Prva faza kodiranja ne uvodi nove promenljive i nasleđuje složenost $O(n_l \cdot n_d \cdot n_p)$ kodiranja osnovnog problema rasporeda. Uslovi kardinalnosti uvode $O(n_d \cdot n_p \cdot n_t \cdot n_r)$ pomoćnih promenljivih i $O(n_d \cdot n_p \cdot n_t \cdot n_r)$ klauza.

U drugoj fazi ima samo $O(n_l \cdot n_r)$ promenljivih i ona je mnogo jednostavnija od prve faze.

13.3 Implementacija

U ovom poglavlju ćemo ukratko opisati implementaciju u potpunosti razvijenu u jeziku C++.

Sistem za automatsko konstruisanje rasporeda časova se sastoji iz sledeće tri komponente:

SAT koder - Uloga SAT kodera je da na osnovu specifikacije uslova rasporeda časova zadate na neki specijalizovani način izgradi iskaznu formulu u standardnom DIMACS formatu. Pokazalo se da nam je za specifikovanje uslova rasporeda časova najpogodnije korišćenje tekstualnih ulaznih datoteka. Iako naša tekuća implementacija ne poseduje grafički korisnički interfejs koji bi korisnicima koji ne poznaju iskazno rezonovanje olakšao zadavanje uslova rasporeda, njegova implementacija je moguća je i planirana je za budućnost. U ovom trenutku, specifikacija uslova rasporeda časova se zadaje u ulaznoj ASCII kodiranoj datoteci koja sadrži listu časova koje je potrebno rasporediti i listu dodatnih uslova rasporeda. Koristi se specijalizovana sintaksa. Na primer:

```
days: mon tue wed thu fri
periods: 1-7
lessons:
   teacher1     group1, group2  subject1   2+1  room1
   teacher2     group1          subject2   3    room1, room2
   teacher2     group2          subject2   3    room1, room2
requirements:
   -teacher1_mon
   -group2_tue_7 | -group2_thu_1
```

U datom primeru radi se pet radnih dana i svaki radni dan ima 7 termina. Nastavnik $teacher_1$ predaje predmet $subject_1$ istovremeno grupama $group_1$ i $group_2$ u sobi $room_1$ dva puta u toku nedelje — jedno predavanje traje dva termina, a drugo jedan. Nastavnik $teacher_2$ predaje predmet $subject_2$ posebno grupama $group_1$ i $group_2$ jednom nedeljno u trajanju od tri termina bilo u sali $room_1$ ili sali $room_2$. Traži se da nastavnik $teacher_1$ nema nastavu ponedeljkom i da grupa $group_2$ nema nastavu ili utorkom u sedmom ili četvrtkom u prvom terminu.

SAT rešavač - Kada se generiše formula u DIMACS formatu, ona se prosleđuje SAT rešavaču koji traži njene modele. Naš sistem koristi rešavač ArgoSAT razvijen u okviru ove teze. Takođe su vršeni eksperimenti i sa

korišćenjem drugih SAT rešavača (na primer MiniSAT, Yices) i dobijeni su veoma slični rezultati.

SAT dekoder - Ukoliko rešavač uspešno pronađe model formula, dekoder ga prevodi u raspored časova koji je u formatu čitljivom za opštu populaciju. U našem sistemu se koristi HTML format i posebna hipertekstualna stranica se generiše za svakog nastavnika, grupu i salu.[4]

[4] Primer generisanog rasporeda je dostupan na veb adresi `http://www.matf.bg.ac.rs/~filip/timetable/`

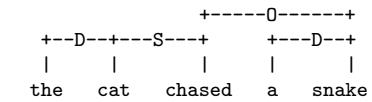
14
Parsiranje prirodnog jezika

U ovoj glavi ukratko će biti opisana primena SAT rešavača na parsiranje prirodnog jezika. Ovakvu primenu SAT rešavača, autor je istraživao u sklopu projekta ,,Guglovo leto koda 2008" (eng. Google Summer of Code 2008). Formalizam koji je korišćen u okviru projekta je formalizam *link gramatika (eng. link grammars)* i cilj projekta bio je izgradnja efikasnog parsera zasnovanog na ovom formalizmu. U ovom poglavlju će biti dat skraćen i uprošćen prikaz formalizma link gramatika, biće prikazano konstruisano SAT kodiranje i biće diskutovani osnovni rezultati ovog projekta. Detaljnije informacije se mogu pronaći u [Mar09e].

14.1 Link gramatike

Link gramatike predstavljaju formalizam za opis sintakse prirodnog jezika. Formalizam je razvijen na Karnegi-Melon Univerzitetu (eng. CMU) 1991. godine i uporedo sa formalizmom razvijena je i implementacija link parsera [ST91]. Iako je izražajna moć link gramatika ekvivalentna izražajnoj moći kontekstnoslobodnih gramatika, pokazuje se da je predstavljanje sintaksnih pravila prirodnog jezika znatno jednostavnije ukoliko se koriste link gramtike.

Sintaksna struktura rečenice se opisuje *vezama (eng. links)* koje se uspostavljaju između reči. Vezama su pridružene *labele (eng. labels)* koji odgovaraju odnosima i ulogama reči u rečenici. Na primer:



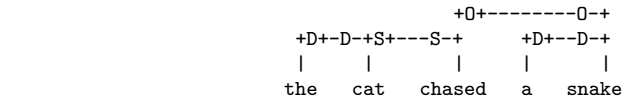
U datom primeru, veza `D` se ostvaruje između člana i imenice koju taj član određuje, veza `S` se ostvaruje između subjekta i predikata u rečenici, dok se veza `O` ostvaruje između predikata i objekta.

Veze moraju da zadovolje određene vrste globalnih uslova.

Planarnost - veze ne smeju da se presecaju (kada se nacrtaju iznad reči).

Povezanost - veze povezuju sve reči u rečenici u jedinstvenu celinu (tj. graf ostvaren vezama je povezan graf).

Svaka reč jezika može da gradi precizno definisan skup veza. Zaista, svaka veza se ostvaruje između dva *konektora (eng. connectors)* — jednog koji ukazuje na desno i označen je sa `+` i jednog koji ukazuje na levo i označen je sa `-`. Na primer:



Svaku reč odlikuje fiksiran skup konektora koji je karakterišu. Reči iz prethodnog primera, opremljene su sledećim konektorima:

the, a	`D+`
cat, snake	`D-`, `O-`, `S+`
Mary	`O-`, `S+`
ran	`S-`
chased	`S-`, `O+`

Dodatno, način na koje je moguće iskoristiti konektore svake od reči se ograničava zadavanjem *etiketa (eng. tags)* za svaku reč. Reči i njihove etikete zadaju se *rečnikom*. Etikete predstavljaju izraze koji se grade od konektora korišćenjem veznika `&` i `or`. Na primer:

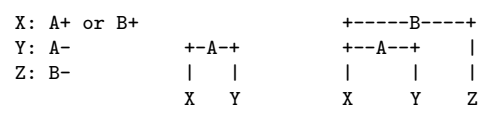
the, a	`D+`
cat, snake	`D- & (O- or S+)`
Mary	`O- or S+`
ran	`S-`
chased	`S- & O+`

Pored uslova planarnosti i povezanosti, da bi povezivanje bilo ispravno, potrebno je da je ispunjen i sledeći uslov.

Zadovoljenost - etiketa svake od reči u rečenici mora da bude zadovoljena.

Iako etikete podsećaju na iskazne formule, zadovoljenost etiketa nije moguće direktno prevesti na iskaznu zadovoljenost. Naime, veznici `&` i `or` ne odgovaraju direktno veznicima $\wedge$ i $\vee$ kako se možda na prvi pogled može očekivati, već imaju drugačiju semantiku. Zadovoljenost etiketa link gramatika se definiše na sledeći način:

- Iskaznim slovima odgovaraju konektori. Konektor se smatra zadovoljenim ako i samo ako učestvuje u izgradnji neke veze. Konektor može da učestvuje u izgradnji najviše jedne veze[1].
- Veznik `or` odgovara (n-arnoj) *ekskluzivnoj* disjunkciji. Dakle, etiketa f_1 `or` ... `or` f_n se smatra zadovoljenom ako i samo ako je tačno jedna od etiketa $f_1, \ldots, f_n$ zadovoljena. Na primer, razmatrajmo sledeća povezivanja.



[1] Proširenja osnovnog formalizma dopuštaju i *multikonektore* koji mogu da učestvuju u izgradnji više veza.

Prvo povezivanje je ispravno jer je u etiketi za reč `X` zadovoljen tačno jedan konektor što čini celu etiketu zadovoljenom, dok drugo povezivanje nije ispravno jer su zadovoljena oba konektora što čini celu etiketu nezadovoljenom.

- Veznik `&` odgovara (n-arnoj) *„ekskluzivnoj"* konjunkciji. Dakle, etiketa f_1 `&` $\ldots$ `&` f_n je zadovoljena ako su sve etikete $f_1, \ldots, f_n$ zadovoljene. Obratno, ako etiketa f_1 `&` $\ldots$ `&` f_n nije zadovoljena, onda ni jedna od etiketa $f_1, \ldots, f_n$ ne može biti zadovoljena. Drugim rečima, etiketa f_1 `&` $\ldots$ `&` f_n i sve etikete $f_1, \ldots, f_n$ su ekvivalentne i u svakom trenutku moraju imati istu istinitosnu vrednost. Na primer, razmatrajmo sledeća povezivanje.

```
X: A+ or (B+ & C+)        +-----B----+
Y: A-                     +--A--+    |
Z: B-                     |     |    |
                          X     Y    Z
```

Ovo povezivanje nije ispravno. Zaista, da bi etiketa na reči `X` bila zadovoljena potrebno je da je zadovoljen tačno jedan argument operatora `or`. Pošto je `A+` zadovoljen zbog uspostavljene veze `A`, etiketa `B+ & C+` ne sme da bude zadovoljena. To znači da nijedan od konektora `B+` i `C+` ne sme biti zadovoljen, što je kontradikcija jer `B+` jeste zadovoljen vezom `B`.

Dodatno, veznik `&` nije komutativan i međusobni redosled navođenja konektora koji vezuju na desno kao i međusobni redosled navođenja konektora koji vezuju na levo je bitan.[2] Važi pravilo da konektori navedeni levo od `&` moraju da budu vezani za bliže reči nego oni koji su navedeni desno od `&`.

Na primer:

```
                 +------B----+      +------A----+
X: A+ & B+       +--A--+     |      +--B--+     |
Y: A-            |     |     |      |     |     |
Z: B-            X     Y     Z      X     Z     Y
```

prvo povezivanje je ipravno, dok drugo nije.

14.2 SAT kodiranje

S obzirom da uslovi zadovoljenosti etiketa podsećaju na uslove iskazne zadovoljivosti, polazna pretpostavka projekta bila je da se efikasan parser može napraviti ukoliko se uslovi koje nameću link gramatike kodiraju formulama iskazne logike i ukoliko se zatim upotrebe efikasni SAT rešavači za njihovo rešavanje. U ovom poglavlju će biti opisano konstruisano SAT kodiranje.

Prvo, uočimo da se svaki konektor koji se javlja na etiketama reči u rečenici jednoznačno može identifikovati kao uređeni par (w_i, p_i), gde je w_i redni broj reči u rečenici čija etiketa sadrži konektor, a p_i pozicija u DFS obilasku stabla odgovarajuće etikete.

Osnovne promenljive u SAT kodiranju su promenljive oblika

$$link_{cc}((w_i, p_i), (w_j, p_j))$$

[2] Međusobni redosled navođenja konektora koji vezuju na levo i onih koji vezuju na desno nije bitan.

koje se formiraju za sve moguće veze koje se mogu uspostaviti. Promenljiva $link_{cc}((w_i, p_i), (w_j, p_j))$ određuje da li je uspostavljena veza između konektora (w_i, p_i) i konektora (w_j, p_j). Istinitosne vrednosti ovih promenljivih određuju celokupno povezivanje rečenice.

Kodiranje uslova zadovoljenosti etiketa. Inspiracija za ovo kodiranje dolazi od Cejtinove (rus. Цейтин) transformacije koja se koristi za prevođenje proizvoljne iskazne formule u ekvizadovoljivu KNF formulu. Svakom čvoru stabla izraza etikete pridružuje se nova iskazna promenljiva koja govori o zadovoljenosti odgovarajućeg dela etikete. Razlikuju se naredni slučajevi.

Disjunkcije - Neka čvoru veznika `or` odgovara promenljiva v a njegovim potomcima promenljive $v\_d_1, \ldots, v\_d_k$. Tada važi

$$v \Leftrightarrow v\_d_1 \ \vee \ \ldots \ \vee \ v\_d_k,$$

pri čemu je prethodna disjunkcija ekskluzivna. Ova formula se prevodi u KNF i kodiranje se proširuje klauzama:

$$\begin{gathered} v \ \Rightarrow \ v\_d_1 \ \vee \ \ldots \ \vee \ v\_d_k \\ v\_d_1 \ \Rightarrow \ v \\ \ldots \\ v\_d_k \ \Rightarrow \ v \\ \neg v\_d_i \ \vee \ \neg v\_d_j, \quad 1 \leq i < j \leq k \end{gathered}$$

Konjunkcije - Neka čvoru veznika `&` odgovara promenljiva v a njegovim potomcima promenljive $v\_c_1, \ldots, v\_c_k$. Tada je potrebno formulu proširiti klauzama:

$$\begin{gathered} v \ \Rightarrow \ v\_c_1 \\ \ldots \\ v \ \Rightarrow \ v\_c_k \\ v\_c_1 \ \Rightarrow \ v \\ \ldots \\ v\_c_k \ \Rightarrow \ v \end{gathered}$$

Napomena: Pošto su sve promenljive v, v_c_1, $\ldots$, v_c_k međusobno logički ekvivalentne, moguće ih je sve zameniti istim iskaznim slovom i preskočiti dodavanje klauza koje kodiraju njihovu ekvivalentnost.

Konektori - Listovi stabla izraza etikete odgovaraju pojedinačnim konektorima. Neka se u nekom listu nalazi konektor (w_i, p_i), neka ovom listu odgovara promenljiva v i neka je W skup konektora (w_j, p_j) takvih da je moguće uspostaviti vezu između njih i konektora (w_i, p_i). Tada važi:

$$v \Leftrightarrow \bigvee_{(w_j, p_j) \in W} link_{cc}((w_i, p_i), (w_j, p_j)).$$

Pošto konektor može da učestvuje u izgradnji najviše jedne veze, prethodna disjunkcija mora biti ekskluzivna. Prethodni uslovi se prevode u KNF na isti način kao i u slučaju veznika `or` i kodiranje se proširuje dobijenim klauzama.

Kodiranje uslova poretka kod konjunkcije. Pretpostavimo da je za svaki konektor (w_i, p_i) i svaku reč w_j definisana promenljiva

$$link_{cw}((w_i, p_i), w_j)$$

koja određuje da li je konektor (w_i, p_i) uspostavio vezu sa nekim konektorom na reči w_j. Neka je W skup svih konektora (w_j, p_j) iz etikete reči na poziji w_j takvih da mogu da se povežu sa konektorom (w_i, p_i). Tada važi

$$link_{cw}((w_i, p_i), w_j) \Leftrightarrow \bigvee_{(w_j, p_j) \in W} link_{cc}((w_i, p_i), (w_j, p_j)),$$

što definiše promenljive $link_{cw}((w_i, p_i), w_j)$ preko baznih promenljivih. Ove formule se na uobičajen način prevode u KNF i njma se proširuje kodiranje.

Prvi korak u kodiranju uslova poretka je identifikovanje parova konektora u okviru neke etikete za koje je potrebno generisate uslove poretka. Trivijalno (ali ne i optimalno) rešenje je za svaku pojavu operatora `&` posmatrati parove konektora koji se dobiju kombinovanjem svih konektora iste vrste iz njegovog i-tog i $i+1$-vog operanda[3]. Neka su tako (w_k, p_i') i (w_k, p_i''), $p_i' < pi''$ dva konektora etikete reči na poziciji w_k za koje je potrebno generisati uslove poretka. Uslovi poretka su tada predstavljeni kao:

$$\neg link_{cw}((w_k, p_i'), w_i) \vee \neg link_{cw}((w_k, p_i''), w_j), \quad \mathcal{B}(w_i, w_j, w_k),$$

i njima se proširuje kodiranje.

Kodiranje uslova planarnosti. Pretpostavimo da je za svake dve reči w_i i w_j uvedena promenljiva

$$link_{ww}(w_i, w_j)$$

koja označava da su reči na pozicijama w_i i w_j povezane nekom vezom. Ovo važi ako i samo ako postoje dva konektora (w_i, p_i) i (w_j, p_j) takva da važi $link_{cc}((w_i, p_i), (w_j, p_j))$. Dakle, mora da važi

$$link_{ww}(w_i, w_j) \Leftrightarrow \bigvee_{p_i, p_j} link_{cc}((w_i, p_i), (w_j, p_j)).$$

Pošto nije moguće postojanje dve različite veze između istih reči, ova disjunkcija mora biti ekskluzivna. Ove formule definišu promenljive $link_{ww}(w_i, w_j)$ preko baznih promenljivih, prevode se u KNF na uobičajen način i njma se proširuje kodiranje.

Korišćenjem ovih promenljivih, uslovi planarnosti se mogu kodirati kao:

$$\neg link_{ww}(w_{i_1}, w_{i_2}) \ \vee \ \neg link_{ww}(w_{j_1}, w_{j_2}), \quad w_{j_1} < w_{i_1} < w_{j_2} < w_{i_2},$$

i njima se proširuje kodiranje.

[3] Opis efikasnijeg rešenja je dostupan u [Mar09e].

Kodiranje uslova povezanosti. Eksperimentalno je utvrđeno da provera povezanosti *a posteriori* (tj. tek nakon konstrukcije veza) daje značajno bolje rezultate nego proširivanje kodiranja na način koji bi obezbedio da svaki iskazni model odgovara povezanoj rečenici. Prilikom a posteriori provere povezanosti, razmatra se graf koji odgovara pronađenim vezama u rečenici, a koji se gradi na osnovu vrednosti promenljivih $link_{ww}(w_i, w_j)$. Kada se ovaj graf konstruiše, jednostavnim DFS obilaskom se nabrajaju sve njegove komponente povezanosti. Ukoliko postoji više komponenti, posebne klauze se dodaju u kodiranje kako bi se sprečilo nastajanje iste vrste nepovezanosti kod budućih povezivanja rečenice (u slučaju da se zahteva konstruisanje još povezivanja). Ove klauze se zasnivaju na činjenici da je, kako bi se dobio povezan graf, svaku komponentu povezanosti potrebno povezati bar jednom granom sa čvorovima koje ne komponenta sadrži. Tako se za svaku komponentu povezanosti C_k the u kodiraje dodaje klauza [4].

$$\bigvee_{w_i \in C_k, w_j \notin C_k} link_{ww}(w_i, w_j).$$

14.3 Diskusija i zaključci

U ovoj glavi opisana je osnovna ideja primene SAT rešavača u okviru parsera prirodnog jezika koji se zasniva na formalizmu link gramatika. Iako su u ovoj tezi prikazani samo osnovna varijanta formalizma link gramatika i odgovarajuće SAT kodiranje, a mnoga važna i kompleksna proširenja koje ovaj formalizam uvodi nisu obrađena, u projektu koji je rađen tokom 2008. godine obrađen je celovit formalizam i napravljena je potpuna implementacija. Eksperimentalna evaluacija izgrađenog sistema i njegovo poređenje sa klasičnim link parserom daju sledeće zaključke.

- Ukoliko na raspolaganju imamo efikasan SAT rešavač otvorenog koda, celokupnu implementaciju parsera je moguće napraviti uz jedan čovek-mesec rada, što je značajno manje nego što je bilo potrebno za izgradnju originalnog link parsera koji je veoma domenski specijalizovan.

- Rečenice najčešće odlikuje više sintaksno ispravnih povezivanja, pri čemu je najčešće samo jedno od njih (ili eventualno mali broj) semantički ispravno. Parser zasnovan na SAT rešavanju pokazuje bolje rezultate ukoliko se razmatra vreme potrebo za konstruisaje prvog sintaksno ispravnog povezivanja. Međutim, ukoliko se razmatra vreme potrebno za konstruisanje svih sintaksno ispravnih povezivanja, originalni link parser pokazuje bolje rezultate. U praksi najčešće nije potrebno konstruisanje svih sintaksno ispravnih povezivanja, već se obično traži konstrukcija semantički ispravnog povezivanja. Kako bi se SAT rešavač naterao da prvo pronađe iskazne modele koji imaju veću šansu da budu i semantički ispravni, potrebno je izmeniti njegove heuristike navođenja (pre svega selekciju literala) tako da u obziru uzmu statističke informacije o prirodnom jeziku i ovo je glavni pravac daljeg rada na ovoj problematici.

[4] Ili $link_{ww}(w_j, w_i)$ ako je $w_j < w_i$.

15
SMT rešavači

U mnogim primena kodiranje problema u čistoj iskaznoj logici i korišćenje SAT rešavača nije najoptimalniji izbor. Veoma često je mnogo bolje problem izraziti formulom bogatije logike (najčešće logike prvog reda) i zatim ispitati njenu zadovoljivost. Međutim, često u primenama nije potrebno ispitati opštu zadovoljivost formule u okviru logike prvog reda, već je potrebno ispitati njenu zadovoljivost u odnosu na neku pogodnu fiksiranu teoriju prvog reda T. Na primer, ukoliko je data formula

$$x < y \wedge \neg(x < y + 0),$$

obično nije relevantno pitanje da li postoji interpretacija simbola $<$, $+$ i 0 takva da je formula zadovoljiva, već je relevantno pitanje ispitati da li je formula zadovoljiva ukoliko se zna da su x i y racionalni brojevi, da je $+$ operacija sabiranja, 0 neutral za sabiranje, a $<$ prirodno uređenje racionalnih brojeva. Problem ispitivanja zadovoljivosti formula logike prvog reda u odnosu na jednu ili više teorija, naziva se *SMT problem (eng. satisfiability modulo theories problem)*. Programi koji rešavaju SMT problem se nazivaju *SMT rešavači*.

Opšti automatski dokazivači teorema za logiku prvog reda (obično zasnovani na rezoluciji) su doživeli značajan napredak u poslednjih dvadesetak godina. Ovi alati se mogu koristiti i za rešavanje SMT problema ako se prilikom ispitivanja zadovoljivosti formuli pridruže i aksiome teorije T. Međutim, to se često pokazuje kao loš izbor. Naime, često teorije nije moguće aksiomatizovati konačnim skupovima aksioma. Čak i kada je moguće konstruisati odgovarajuću aksiomatizaciju, u praksi se pokazuje da je umesto opštih dokazivača za logiku prvog reda mnogo pogodnije koristiti specijalizovane procedure odlučivanja za pojedine teorije. S obzirom na to da formule čija se zadovoljivost ispituje obično imaju veoma bogatu iskaznu strukturu, u novije vreme SMT rešavači kombinuju procedure odlučivanja za pojedine teorije sa efikasnim SAT rešavačima.

U nastavku ove glave biće prikazani osnovni načini na koje se SAT rešavači koriste u okviru SMT rešavača. Takođe, ukratko će biti opisan SMT rešavač ArgoLib čije su ranije verzije opisane u [MJ04, Mar05], a čije novije verzije koriste SAT rešavač ArgoSAT. Takođe, biće opisana procedura odlučivanja za linearnu aritmetiku nad racionalnim brojevima koja je predložena u [DM06a, DM06b], a implementirana i u okviru sistema ArgoLib.

15.1 Logika prvog reda

U ovom poglavlju će logika prvog reda i SMT problem biti formalno opisani. Pogodnosti radi, SMT problem se obično formuliše u okviru *višesortne logike prvog reda*, iako se, sasvim ekvivalentno, sva ograničenja koja sorte (tipovi) nameću mogu predstaviti odgovarajućim unarnim predikatima u okviru klasične logike prvog reda. Uvedimo za početak osnovne definicije[1].

15.1.1 Sintaksa

U nastavku ćemo razmatrati isključivo *bazne (eng. ground) formule* tj. formule koje ne sadrže promenljive (ni kvantifikatore) i definisane su nad nekim jezikom $\mathcal{L}$. Uvedimo najpre precizno pojam višesortnog jezika.

Definicija 15.1. Višesortni jezik (signatura) prvog reda *(skr. jezik) je uređena petorka* $\mathcal{L} = (\Sigma, \Pi, S, sort)$ *pri čemu je:*

- Σ — *skup* funkcijskih simbola*;*
- Π — *skup* relacijskih simbola*;*
- S — *skup* sorti*;*
- $sort$ — *funkcija* $sort : \Sigma \cup \Pi \to \mathcal{P}(S)$ *koja svakoj promenljivoj dodeljuje jedinstvenu sortu* τ_i*, svakom relacijskom simbolu dodeljuje* k*-torku sorti* $(\tau_1, \ldots, \tau_k)$*, a svakom funkcijskom simbolu dodeljuje* $k+1$*-torku sorti* $(\tau_1, \ldots, \tau_k, \tau)$*. Za funkcijski simbol* f *kažemo da je sorte* τ*.*

  Broj k *nazivamo arnost funkcijskog odnosno predikatskog simbola i on definiše funkciju* $ar : \Sigma \cup \Pi \to \mathbb{N}$*.*

Skupovi Σ, Π *i* S *su dva po dva disjunktni.*
Funkcijske simbole arnosti 0 *nazivamo* konstantama*, a relacijske simbole arnosti* 0 *nazivamo* logičkim konstantama*.*

Komponovanjem funkcijskih simbola, grade se $\mathcal{L}$-termi.

Definicija 15.2. *Neka je dat jezik* $\mathcal{L}$*.* $\mathcal{L}$ *– termi su elementi induktivnog skupa* $Term_{\mathcal{L}}$ *definisanog putem narednog svojstva:*

1. *ako su* $t_1, \ldots, t_k$ $\mathcal{L}$*-termi sorti, redom,* $(\tau_1, \ldots, \tau_k)$ *i* $f \in \Sigma$ *funkcijski simbol za koji je* $sort(f) = (\tau_1, \ldots, \tau_k, \tau)$*, tada je i* $f(t_1, \ldots, t_k)$ $\mathcal{L}$*-term sorte* τ*, tj.*

   $$f \in \Sigma \ \wedge \ t_1 \in Term_{\mathcal{L}} \ \wedge \ \ldots \ \wedge \ t_k \in Term_{\mathcal{L}} \implies f(t_1, \ldots, t_k) \in Term_{\mathcal{L}}.$$

   Specijalno, sve konstante (tj. funkcijski simboli arnosti nula) su $\mathcal{L}$*-termi.*

Od $\mathcal{L}$-termova, primenom relacijskih simbola, grade se $\mathcal{L}$-atomičke formule.

Definicija 15.3. *Neka je dat jezik* $\mathcal{L}$*.* $\mathcal{L}$-atomičke formule (skr.$\mathcal{L}$-atomi) *su elementi induktivnog skupa* $Atom_{\mathcal{L}}$ *definisanog putem narednih svojstava:*

[1] Ovaj put formalizacija neće biti urađena strogo kao što je to bio slučaj sa formalizacijom iskazne logike

1. *ako su t_1 i t_2 $\mathcal{L}$-termi iste sorte, tada je*

$$t_1 = t_2$$

$\mathcal{L}$-atom, tj.

$$t_1 \in Term_{\mathcal{L}} \wedge t_2 \in Term_{\mathcal{L}} \Rightarrow t_1 = t_2 \in Atom_{\mathcal{L}};$$

2. *ako je $p \in \Pi$ relacijski simbol jezika $\mathcal{L}$ za koji je $sort(p) = (\tau_1, \ldots, \tau_k)$, a $t_1, \ldots, t_k$ su $\mathcal{L}$-termi sorti, redom, $(\tau_1, \ldots, \tau_k)$, tada je*

$$p(t_1, \ldots, t_k)$$

$\mathcal{L}$-atom, tj.

$$p \in \Pi \ \wedge \ t_1 \in Term_{\mathcal{L}} \ \wedge \ \ldots \ \wedge \ t_k \in Term_{\mathcal{L}} \ \Rightarrow \ p(t_1, \ldots, t_k) \in Atom_{\mathcal{L}}.$$

Specijalno, sve logičke konstante (tj. relacijski simboli arnosti nula) su $\mathcal{L}$-atomi.

Iako opšta logika prvog reda, dalje, dopušta izgradnju $\mathcal{L}$-formula kombinovanjem $\mathcal{L}$-atomičkih formula korišćenjem logičkih operatora i kvantifikatora, za SMT problem, najčešće su značajne isključivo $\mathcal{L}$-formule u konjunktivnoj normalnoj formi koj koje ne sadrže kvantifikatore. Zato $\mathcal{L}$-formule definišemo na sledeći način (analogno definiciji 6.1).

Definicija 15.4. *Neka je dat jezik $\mathcal{L}$.*
$\mathcal{L}$-literal (skr. literal) je ili atom a ili negacija atoma $\neg a$.
$\mathcal{L}$-klauza (skr. klauza) je lista $\mathcal{L}$-literala.
$\mathcal{L}$-formula (skr. formula) je lista $\mathcal{L}$-klauza.

15.1.2 Semantika

Kako bi mogla da se uvede semantika $\mathcal{L}$-formula, neophodno je interpretirati značenje nelogičkih simbola koji se u njoj javljaju, tj. dati interpretaciju jezika $\mathcal{L}$.

Definicija 15.5. *Neka je dat jezik $\mathcal{L} = (\Sigma, \Pi, S, sort)$. $\mathcal{L}$-model (ili $\mathcal{L}$-strukturu) $\mathfrak{A}$ jezika $\mathcal{L}$ čine sledeći elementi.*

- *Za svaku sortu $\tau_i \in S$ po jedan neprazan skup A_i koji se naziva domenom sorte τ_i.*
- *Za svaki funkcijski simbol $f \in \Sigma$ takav da je $sort(f) = (\tau_{i_1}, \ldots, \tau_{i_k}, \tau_j)$ funckija $\hat{f} : A_{i_1} \times \ldots A_{i_k} \mapsto A_j$. Specijalno, funkcija $\hat{f}$ svakoj konstanti dodeljuje element odgovarajuće sorte A_j.*
- *Za svaki relacijski simbol $p \in \Pi$ tako da je $sort(f) = (\tau_{i_1}, \ldots, \tau_{i_k})$ relacija $\hat{p} \subseteq A_{i_1} \times \ldots A_{i_k}$. Specijalno, za svaku logičku konstantu nularna relacija $\hat{p}$ predstavlja njenu istinitosnu vrednost.*

Kada je fiksiran model $\mathfrak{A}$ jezika $\mathcal{L}$, svakoj $\mathcal{L}$-formuli je moguće odrediti istinitosnu vrednost u modelu $\mathfrak{A}$ i tako definisati pojam zadovoljivosti. S obzirom da su $\mathcal{L}$-formule po svojoj definiciji bazne, naredne definicije su prilično uprošćene u odnosu na opštu definiciju zadovoljivosti u okviru logike prvog reda.

Definicija 15.6. *Neka je dat jezik $\mathcal{L}$ i odgovarajuća $\mathcal{L}$-struktura $\mathfrak{A}$. Funkcija $\bar{\mathfrak{A}} : Term_{\mathcal{L}} \mapsto A_1 \cup \ldots \cup A_n$ dodeljuje $\mathcal{L}$-termima odgovarajuće vrednosti $\mathcal{L}$-terma u strukturi $\mathfrak{A}$ i definisana je primitivnom rekurzijom po induktivnom skupu $Term_{\mathcal{L}}$.*

- *Ukoliko je $t = f(t_1, \ldots, t_k)$ tada je*

$$\bar{\mathfrak{A}}(t) = \hat{f}(\bar{\mathfrak{A}}(t_1), \ldots, \bar{\mathfrak{A}}(t_k)).$$

Specijalno, ako je $t = f$, pri čemu je f konstanta, tada je

$$\bar{\mathfrak{A}}(t) = \hat{f}.$$

Definicija 15.7. *Neka je dat jezik $\mathcal{L}$ i odgovarajuća $\mathcal{L}$-struktura $\mathfrak{A}$.*

Kažemo da $\mathfrak{A}$ zadovoljava $\mathcal{L}$-atom a, što označavamo sa $\mathfrak{A} \vDash a$, ako važi:

- *Ukoliko je a oblika $t_1 = t_2$ tada je $\mathfrak{A} \vDash a$ ako i samo ako je $\bar{\mathfrak{A}}(t_1) = \bar{\mathfrak{A}}(t_2)$ (gde je $\bar{\mathfrak{A}}$ funkcija vrednosti $\mathcal{L}$-termova);*
- *Ukoliko je a oblika $p(t_1, \ldots, t_k)$ tada je $\mathfrak{A} \vDash a$ ako i samo ako važi $\hat{p}(\bar{\mathfrak{A}}(t_1), \ldots, \bar{\mathfrak{A}}(t_k))$ (gde je $\bar{\mathfrak{A}}$ funkcija vrednosti $\mathcal{L}$-termova);*

U suprotnom kažemo da $\mathfrak{A}$ ne zadovoljava $\mathcal{L}$-atom a, što označavamo sa $\mathfrak{A} \nvDash a$.

Slično, kažemo da $\mathfrak{A}$ zadovoljava $\mathcal{L}$-literal l, što označavamo sa $\mathfrak{A} \vDash l$, ako važi:

- *Ukoliko je $\mathcal{L}$-literal l jednak $\mathcal{L}$-atomu a, onda je $\mathfrak{A} \vDash l$ ako i samo ako je važi $\mathfrak{A} \vDash a$.*
- *Ukoliko je $\mathcal{L}$-literal l negacija $\mathcal{L}$-atoma a (tj. $l = \neg a$), onda je $\mathfrak{A} \vDash l$ ako i samo ako ne važi $\mathfrak{A} \vDash a$, tj. ako važi $\mathfrak{A} \nvDash a$.*

U suprotnom kažemo da $\mathfrak{A}$ ne zadovoljava $\mathcal{L}$-literal a, što označavamo sa $\mathfrak{A} \nvDash l$.

U trenutku kada je definisana zadovoljenost $\mathcal{L}$-literala, moguće definisati zadovoljivost $\mathcal{L}$-formula analogno definiciji 6.7:

Definicija 15.8. *Neka je dat jezik $\mathcal{L}$ i odgovarajuća $\mathcal{L}$-struktura $\mathfrak{A}$.*

$\mathcal{L}$-struktura $\mathfrak{A}$ zadovoljava $\mathcal{L}$-klauzu c, što označavamo sa $\mathfrak{A} \vDash c$, ako i samo ako $\exists l.\ l \in c \wedge \mathfrak{A} \vDash l$.

$\mathcal{L}$-struktura $\mathfrak{A}$ zadovoljava $\mathcal{L}$-formulu F, što označavamo sa $\mathfrak{A} \vDash F$, ako i samo ako $\forall c.\ c \in F \Rightarrow \mathfrak{A} \vDash c$.

Ako važi $\mathfrak{A} \vDash F$ tada kažemo da je F zadovoljiva i da je $\mathfrak{A}$ model formule F.

Primetimo da su $\mathcal{L}$-strukture u nekom smislu *totalne*, tj. omogućavaju određivanje istinitosne vrednosti svih $\mathcal{L}$-literala. Zaista, pojam netačnosti je moguće definisati tako da $\mathfrak{A} \vDash \neg l$ važi ako i samo ako $\mathfrak{A} \nvDash l$. Ovakvo svojstvo totalnosti se dalje prenosi i na $\mathcal{L}$-klauze i $\mathcal{L}$-formule.

U nastavku će biti uveden pojam $\mathcal{L}$-valuacije, analogan pojmu valuacije uvedenom kroz definiciju 6.1 i biće prikazana njegova veza sa zadovoljivošću $\mathcal{L}$-formula. Ova veza predstavlja ključnu sponu između iskazne logike i logike prvog reda KNF formula i omogućava korišćenje SAT rešavača u okviru rešavanja SMT problema.

Definicija 15.9. $\mathcal{L}$-valuacija *je lista* $\mathcal{L}$*-literala.*

U tom slučaju, zadovoljenost (tj. tačnost) se definiše analogno definiciji 6.7.

Definicija 15.10.
$\mathcal{L}$-literal l je tačan u $\mathcal{L}$-valuaciji $\mathfrak{v}$, što označavamo sa $\mathfrak{v} \vDash l$ ako i samo ako $l \in \mathfrak{v}$.

$\mathcal{L}$-klauza c je tačna u $\mathcal{L}$-valuaciji $\mathfrak{v}$, što označavamo sa $\mathfrak{v} \vDash c$, ako i samo ako $\exists l.\ l \in c \wedge \mathfrak{v} \vDash l$.

$\mathcal{L}$-formula F je tačna u $\mathcal{L}$-valuaciji $\mathfrak{v}$, što označavamo sa $\mathfrak{v} \vDash F$, ako i samo ako $\forall c.\ c \in f \Rightarrow \mathfrak{v} \vDash c$.

Zbog identične strukture definicija 6.7 i 15.10, SAT rešavači se mogu direktno koristiti za pronalaženje neprotivrečnih $\mathcal{L}$-valuacija u kojima je data formula F tačna[2]. Međutim, kako bi se uspostavila veza između pronađenih $\mathcal{L}$-valuacija i zadovljivosti $\mathcal{L}$-formula uvedene kroz definiciju 15.8, neophodno je pojačati pojam neprotivrečnosti valuacija.

Definicija 15.11. *$\mathcal{L}$-valuacija $\mathfrak{v}$ je* neprotivrečna *u strukturi $\mathfrak{A}$ ako za svaki $\mathcal{L}$-literal l iz $\mathfrak{v}$ važi $\mathfrak{A} \vDash l$, tj. $\forall l.\ l \in \mathfrak{v} \Rightarrow \mathfrak{A} \vDash l$.*

Važi naredni stav koja daje vezu između $\mathcal{L}$-valuacija i $\mathcal{L}$-modela.

Stav 15.1. *Neka je dat jezik $\mathcal{L}$, $\mathcal{L}$-struktura $\mathfrak{A}$ i $\mathcal{L}$-formula F. Važi $\mathfrak{A} \vDash F$ ako i samo ako postoji $\mathcal{L}$-valuacija $\mathfrak{v}$ neprotivrečna u strukturi $\mathfrak{A}$ takva da $\mathfrak{v} \vDash F$.*

15.1.3 Teorije

Kao što je već na početku ove glave nagoveštено, za SMT problem nije relevantno ispitivanje postojanje ma kakvog $\mathcal{L}$-modela za $\mathcal{L}$-formulu čija se zadovoljivost ispituje, već se obično pretraga ograničava na unapred zadatu klasu $\mathcal{L}$-struktura. Teorije obično ograničavaju interpretaciju nekih funkcijskih i relacijskih simbola. Simboli čija se interpretacija definiše u okviru neke teorije $\mathcal{T}$ čine jezik te teorije $\mathcal{L}_\mathcal{T}$. Dakle, u skladu sa novijim tekstovima o SMT problemu, teorije prvog reda defišemo semantički.

Definicija 15.12. Teorija $\mathcal{T}$ *jezika $\mathcal{L}_\mathcal{T}$ predstavlja familiju $\mathcal{L}_\mathcal{T}$-struktura $\{\mathfrak{A}^\mathcal{T}\}$.*

Narednom definicijom se uvodi zadovoljivost formula definisanih nad jezikom teorije $\mathcal{T}$.

Definicija 15.13. *Neka je data teorija $\mathcal{T}$. $\mathcal{L}_\mathcal{T}$-formula F je $\mathcal{T}$-zadovoljiva ako i samo ako ima $\mathcal{T}$-model tj. akko ima $\mathcal{L}_\mathcal{T}$-model $\mathfrak{A}^\mathcal{T}$ (tj. $\mathfrak{A}^\mathcal{T} \vDash F$), pri čemu je $\mathfrak{A}^\mathcal{T}$ neki od modela teorije $\mathcal{T}$.*

Međutim, u okviru SMT problema se obično ispituje zadovoljivost formula koje pored simbola teorije sadrže dodatne simbole (obično konstanti).

Definicija 15.13'. *Neka je data teorija $\mathcal{T}$ i jezik $\mathcal{L}$ koji je proširenje jezika $\mathcal{L}_\mathcal{T}$. $\mathcal{L}$-formula F je $\mathcal{T}$-zadovoljiva ako i samo ako ima $\mathcal{T}$-model, tj. akko postoji $\mathcal{L}$-model $\mathfrak{A}$ (tj. važi $\mathfrak{A} \vDash F$) koji predstavlja proširenje nekog $\mathcal{L}_\mathcal{T}$-modela $\mathfrak{A}^\mathcal{T}$ teorije $\mathcal{T}$.*

[2] Ova veza bi se mogla učiniti još eksplicitnijom ukoliko bi se uvelo bijektivno preslikavanje između skupa $\mathcal{L}$-literala i skupa iskaznih literala poznato pod imenom *iskazna apstrakcija*.

Dakle, SMT problem je problem ispitivanja $\mathcal{T}$-zadovoljivosti neke $\mathcal{L}$-formule F, pri čemu je $\mathcal{L}$ neko od dopuštenih proširenja jezika $\mathcal{L}_\mathcal{T}$ teorije $\mathcal{T}$.

Ponovo ćemo uspostaviti vezu sa iskaznim valuacijama.

Definicija 15.14. *$\mathcal{L}$-valuacija $\mathfrak{v}$ je $\mathcal{T}$-neprotivrečna akko ima $\mathcal{T}$-model tj. akko postoji $\mathcal{L}$-model $\mathfrak{A}$ koji proširuje neki model $\mathfrak{A}^\mathcal{T}$ teorije $\mathcal{T}$ takav je $\mathfrak{v}$ neprotivrečna u $\mathfrak{A}$ (tj. za svaki $\mathcal{L}-$literal l iz $\mathfrak{v}$ važi $\mathfrak{A} \vDash l$).*

Stav 15.2. *Neka je data teorija $\mathcal{T}$ i jezik $\mathcal{L}$ koji je proširenje jezika $\mathcal{L}_\mathcal{T}$. $\mathcal{L}$-formula F je $\mathcal{T}$-zadovoljiva ako i samo postoji $\mathcal{T}$-neprotivrečna $\mathcal{L}$-valuacija $\mathfrak{v}$ takva da $\mathfrak{v} \vDash F$.*

Pošto su sve $\mathcal{L}$-valuacije neprotivrečne u strukturi $\mathfrak{A}$, neprotivrečne i u iskaznom smislu (datom kroz definiciju 6.8), procedura za rešavanje SMT problema (tj. ispitivanje $\mathcal{T}$-zadovoljivosti $\mathcal{L}$ formula) može se konstruisati tako što se SAT rešavač koristi za konstruisanje svih (iskazno) neprotivrečnih $\mathcal{L}$-valuacija, nakon čega se za ovako konstruisane $\mathcal{L}$-valuacije, specijalizovanim procedurama odlučivanja, proverava njihova $\mathcal{T}$-neprotivrečnost.

Slično kao i u iskaznom slučaju, moguće je definisati pojam semantičkih posledica.

Definicija 15.15. *Neka je data teorija $\mathcal{T}$ i jezik $\mathcal{L}$ koji je proširenje jezika $\mathcal{L}_\mathcal{T}$.*

$\mathcal{L}$-formula F $\mathcal{T}$-povlači $\mathcal{L}$-literal l, što označavamo sa $F \vDash_\mathcal{T} l$, akko je svaki $\mathcal{T}$-model $\mathcal{L}$-formule F ujedno i $\mathcal{T}$-model $\mathcal{L}$-literala l.

$\mathcal{L}$-formula F $\mathcal{T}$-povlači $\mathcal{L}$-klauzu c, što označavamo sa $F \vDash_\mathcal{T} c$, akko je svaki $\mathcal{T}$-model $\mathcal{L}$-formule F ujedno i $\mathcal{T}$-model $\mathcal{L}$-klauze c.

$\mathcal{L}$-formula F $\mathcal{T}$-povlači $\mathcal{L}$-formulu F', što označavamo sa $F \vDash_\mathcal{T} F'$, akko je svaki $\mathcal{T}$-model $\mathcal{L}$-formule F ujedno i $\mathcal{T}$-model $\mathcal{L}$-formule F'.

Najčešće korišćene teorije. U praksi se koristi veliki broj odlučivih teorija ili njihovih odlučivih fragmenata. U nastavku će biti nabrojano nekoliko teorije koje se najčešće koriste u primenama u verifikaciji softvera i hardvera.

- *Linearna aritmetika nad realnim brojevima (Linear Real Arithmetic - LRA).* Ovo je teorija jezika $\mathcal{L}_{LRA}$ takvog da je $\Sigma_{LRA} = \{0, 1, +, -\}$, $\Pi_{LRA} = \{<\}$, $S_{LRA} = \{Q\}$. Teorija je određena modelom koji je određen skupom racionalnih brojeva $\mathbb{Q}$ kao domenom sorte Q u kome je $+$ interpretirana kao operacija sabiranja, $-$ kao operacija oduzimanja, 0 kao nula, 1 kao jedinica, a $<$ kao klasično uređenje racionalnih brojeva. Pri izgradnji formula dopušteno je da se koriste i neinterpretirane konstante kao i skraćeni zapisi relacija (npr. $>, \leq, \geq, \neq$).

  Primer 15.1. Ispitati da li postoje racionalni brojevi koji zadovoljavaju

  $$x \leq y + z \wedge (y < 3 \vee 2 \cdot y + 1 > 5) \wedge z > x + 2.$$

  Zadovoljivost nad realnim brojevima se zatim svodi na zadovoljivost nad racionalnim brojevima.

- *Linearna aritmetika nad celim brojevima (Linear Integer Arithmetic - LIA).* Ovo je veoma slična teorija teoriji *LRA*, pri čemu je domen jedine sorte Z skup celih brojeva $\mathbb{Z}$ i operacije su interpretirane u okviru ovog skupa.

Primer 15.2. Ispitati da li postoje celi brojevi koji zadovoljavaju

$$x > y + 2 \cdot z \wedge (x = 2 \vee x + y + 2 \cdot z > 0) \wedge x < 1.$$

- *Logika razlika (Difference Logic).* Ova teorija predstavlja oslabljenje linearne aritmetike gde su u okviru $\mathcal{L}$-formula dopušteni isključivo $\mathcal{L}$-atomi oblika $x - y < b$, gde je b racionalan broj. Opet se razlikuju racionalni i celobrojna varijanta, pri čemu su modeli određeni na isti način kao kao kod teorija linearne aritmetike.

- *Teorija jednakosti sa neinterpretiranim funkcijskim simbolima (Equality with Uninterpreted Functions - EUF).* Ovo teorija praznog jezika $\mathcal{L}_{EUF}$ pri čemu se za skup dopuštenih modela uzimaju svi mogući modeli. S druge strane, formule čija se zadovoljivost ispituje obično uključuju dodatne simbole konstanti, ali i simbole funkcija (koji nemaju unapred fiksiranu interpretaciju u modelu, te se nazivaju neinterpretiranim funkcijskim simbolima).

  Primer 15.3. Ispitati da li postoje elementi x, y i z takvi da važi

  $$y = f(x) \wedge z = f(y) \wedge g(z) \neq g(f(f(x))).$$

  S obzirom da se f i g interpretiraju nekim funkcijama u okviru modela, za njih važi svojstvo kongruentnosti, tj. važi $\forall\, x\, y.\ x = y \Rightarrow f(x) = f(y)$ (i slično za funkciju g).

- *Teorija nizova (Arrays)* Ovo je teorija jezika $\mathcal{L}_{Arrays}$ sa dva funkcijska simbola $\Sigma_{Arrays} = \{\mathsf{read}, \mathsf{write}\}$ i tri sorte (nizova, indeksa i vrednosti) $S_{Arrays} = \{A, I, V\}$. Simbol read je sorte (A, I, V), a simbol write je sorte (A, I, V, A), gde read označava čitanje elementa niza na datoj poziciji, a write označava upis u niz na datu poziciju. Skup modela se definiše kao skup svih $\mathcal{L}_{Arrays}$-modela koji zadovoljavaju aksiome:

  $$\begin{aligned} &\forall\, A\, i\, j\, v. \\ &\quad i = j \ \Rightarrow\ \mathsf{read}(\mathsf{write}(A, i, v), j) = v) \wedge \\ &\quad i \neq j \ \Rightarrow\ \mathsf{read}(\mathsf{write}(A, i, v), j) = \mathsf{read}(A, j) \end{aligned}$$

  Ukoliko je još zadovoljena i aksioma

  $$\forall A, B.\ (\forall i.\ (\mathsf{read}(A, i) = \mathsf{read}(B, i))) \Rightarrow\ a = b,$$

  teorija se naziva *ekstenzionalna teorija nizova*.

  Primer 15.4. U cilju ispitivanja da li nakon izvršavanja koda

  ```
  A[A[i]] = a;
  b = A[A[i]];
  ```

  važi $b = a$, potrebno je ispitati da li je zadovoljiva formula

  $$\mathsf{write}(A, \mathsf{read}(A, i), a) = B \wedge b = \mathsf{read}(B, \mathsf{read}(B, i)) \wedge a \neq b$$

Teorija T može da bude i kombinacija nekoliko različitih teorija.

Primer 15.5. Razmotrimo, na primer, formulu

$$z = f(x - y) \wedge x = y + z \wedge x \neq f(f(z)) + y.$$

Ova formula sadrži simbole + i −, koji pripadaju teoriji linearne aritmetike, ali ona ne predstavlja formulu linearne aritmetike zbog prisustva funcijskog simbola f. Simbol f pripada teoriji jednakosti sa neinterpretiranim funkcijskim simbolima. Dakle, navedena formula je formulisana nad dve teorije za koje se može pokazati da je problem zadovoljivosti odlučiv (preciznije, odlučiv je samo problem zadovoljivosti za egzistencijalno kvantifikovani fragment teorije jednakosti sa neinterpretiranim funkcijskim simbolima). Pre postupka rešavanja, vrši se transformacija kojom se polazna formula transformiše u

$$a = x - y \wedge z = f(a) \wedge x = y + z \wedge x \neq b + y \wedge b = f(f(z)).$$

Ovim se postiže da je moguće razdvojiti uslove koji pripadaju teoriji LRA ($a = x - y$, $x = y + z$ i $x \neq b + y$) i uslove koji pripadaju teoriji EUF ($z = f(a)$ i $b = f(f(z))$). Na osnovu uslova za teoriju LRA sledi da je $a = z$, zatim na osnovu toga i uslova za teoriju EUF sledi da je $b = a$, što daje nezadovoljivost u teoriji LRA. Dakle, teorije prilikom rešavanja vrše komunikaciju preko propagiranja jednakosti, što je suština tzv. *Nelson-Oppen-ove sheme za kombinovanje procedura odlučivanja*[NO79].

15.2 Apstraktni opisi SMT rešavača u vidu sistema promena stanja

Pravila koja smo uveli glavi 4 kako bismo opisali SAT rešavače se veoma lako mogu prilagoditi kako bi se opisali SMT rešavači. U ovoj glavi, pravila će biti zasnovana na sistemu datom u [NOT06]. Za početak, umesto literala, klauza, formula i valuacija koriste se $\mathcal{L}$-literali, $\mathcal{L}$-klauze, $\mathcal{L}$-formule i $\mathcal{L}$-valuacije (kako bismo pojednostavili izlaganje, ovo u nastavku neće biti eksplicitno naglašavano).

Pravila koja se i dalje koriste (a koja ni u čemu ne zavise od teorije $\mathcal{T}$), su Decide, UnitPropagate, Restart. Pored ovoga, koriste se i sledeća pravila (koja zavise od teorije):

T-Learn:
$$\frac{F \models_{\mathcal{T}} c \quad (\mathsf{atoms}\ c) \subseteq \mathit{Atoms}}{F = F\ @\ c}$$

T-Forget:
$$\frac{(F \setminus c) \models_{\mathcal{T}} c}{F := F \setminus c}$$

T-Backjump:
$$\frac{C = l \vee l_1 \vee \ldots \vee l_k \quad F \models_{\mathcal{T}} C \quad M = M' \,|\, d\, M'' \quad \overline{l}_1, \ldots, \overline{l}_k \in M' \quad \overline{l} \in M''}{M := M'\ l}$$

T-Propagate:
$$\frac{M \models_{\mathcal{T}} l \quad (\mathsf{atom}\ l) \in \mathit{Atoms} \quad l, \overline{l} \notin M}{M := M\ l}$$

Osnovna funkcionalnost koja je neophodna da bi se mogao implementirati SMT rešavač je postojanje mogućnosti provere da li je valuacija M $\mathcal{T}$-neprotivrečna valuacija.

Redosled primene pravila određuje tzv. SMT strategiju.

Osnovi lenji pristup. Jedna od najjednostavnijih strategija je tzv. *osnovni lenji pristup*. Osnovni lenji pristup primenjuje isključivo iskazna pravila dok ne stigne u stanje $M \,||\, F$ koje je završno stanje u odnosu na ta pravila. Ukoliko je stanje takvo da se iz njega zaključuje da je F nezadovoljiva formula u iskaznom smislu, ona je i $\mathcal{T}$-nezadovoljiva. Ukoliko je stanje $M \,||\, F$ takvo da M predstavlja iskazni model za F, proverava se da li je M $\mathcal{T}$-neprotivrečna valuacija. Ukoliko jeste, formula F je $\mathcal{T}$-zadovoljiva i pronađen je njen željeni $\mathcal{T}$-model. Ukoliko nije, tada postoje $\mathcal{L}$-literali $l_1, \ldots, l_k$ koji su članovi valuacije M, takvi da $[\,] \vDash_{\mathcal{T}} \overline{l}_1 \vee \ldots \vee \overline{l}_k$ (u najjednostavnijoj implementaciji, moguće je uzeti sve literale iz M). Tada se ova klauza uči primenom pravila $\mathcal{T}$-Learn i ona ima ulogu da spreči da se u budućnosti stigne do iste valuacije M. Nakon ovoga, proces pretrage kreće iz početka primenom pravila Restart.

Inkrementalnost $\mathcal{T}$-rešavača. Mana prethodnog pristupa je što se provera $\mathcal{T}$-neprotivrečnosti valuacije M vrši tek kada je ona potpuno izgrađena i kada predstavlja iskazni model formule. Pošto se valuacija M gradi inkrementalno dodajući jedan po jedan literal, mnogo efikasniji pristup bi bio da se prekine sa izgradnjom valuacije M čim se ustanovi da je T-protivrečna. Ovo znači da je sa svakim dodavanjem novog literala u valuaciju M potrebno ispitati njenu $\mathcal{T}$-neprotivrečnost. Ukoliko se ova provera vrši svaki put iz početka, postoji realna opasnost da će ovo dovesti do neefikasnosti. *Inkrementalni $\mathcal{T}$-rešavači* su u mogućnosti da prilikom provere $\mathcal{T}$-neprotivrečnosti proširene valuacije, iskoriste prethodne rezultate dobije prilikom provere $\mathcal{T}$-konzistentnosti valuacije pre njenog proširenja. Dakle, osnovna razlika SMT rešavača sa inkrementalnim $\mathcal{T}$ rešavačem u odnosu na osnovni lenji pristup je što se provera $\mathcal{T}$-neprotivrečnosti valuacije M vrši i u stanjima koja nisu završna u odnosu na iskazna pravila.

,,On-line" SAT rešavači. Mana prethodnih pristupa je i da se u trenutku kada se otkrije $\mathcal{T}$-protivrečnost (parcijalne) valuacije M rešavanje započinje iz početka. Umesto otpočinjanja iz početka, ,,on-line" SAT rešavači prilikom otkrivanja $\mathcal{T}$-nekonzistentnosti korišćenjem pravila $\mathcal{T}$-Backjump vraćaju rešavač u najnaprednije stanje u kome je valuacija M i dalje bila $\mathcal{T}$-neprotivrečna i pretragu nastavljaju odatle.

Propagacija na osnovu teorije. Prethodni pristupi ne koriste pravilo Theory-Propagate. To sa sobom povlači da se informacije dostupne na onsovu teorije $\mathcal{T}$ ne koriste kako bi navodile proces pretrage, već se samo koriste kako bi naknadno proveravale $\mathcal{T}$-konzistentnost pronađenih valuacija. Ukoliko se uključi korišćenje pravila TheoryPropagate, rešavači po običaju postaju znatno efikasniji.

15.2.1 DPLL(T) arhitektura

DPLL(T) arhitektura predložena u [GHN+04, NOT06] zagovara eksplicitno razdvanje SAT rešavača $DPLL(X)$ od rešavača za pojedine teorije $Solver_{\mathcal{T}}$. Arhitektura zahteva od rešavača za teorije sledeće operacije.

- Operacija kojom $DPLL(X)$ obaveštava $Solver_{\mathcal{T}}$ da je određen literal postao član tekuće valuacije M.
- Operacija kojom $Solver_{\mathcal{T}}$ proverava da li je tekuća (parcijalna) valuacija M postala $\mathcal{T}$-protivrečna. Ovu operaciju je moguće vršiti sa različitim stepenima „snage": za neke teorije $\mathcal{T}$ za koje je skupo vršiti kompletnu proveru $\mathcal{T}$-neprotivrečnosti, ali postoje nekompletne metode koje su u stanju da brže detektuju protivrečnost, efikasnije je koristiti te brže metode tokom izgradnje valuacije M, a skuplje kompletne metode pozivati samo kada je izgrađena celokupna valuacija M.
- Operacija kojom $Solver_{\mathcal{T}}$ objašnjava detektovanu $\mathcal{T}$-protivrečnost. Objašnjenje dolazi u obliku (što manjeg) skupa literala $\{l_1, \ldots, l_k\}$ takvog da važi $[] \vDash_{\mathcal{T}} \overline{l}_1 \vee \ldots \vee \overline{l}_k$ i koristi pri analizi konflikata..
- Operacija kojom $Solver_{\mathcal{T}}$ identifikuje literale koje su $\mathcal{T}$-posledice tekuće valuacije M. I ovu operaciju je moguće vršiti sa različitim nivoima „snage".
- Operacija kojom se pronalazi objašnjenje za $\mathcal{T}$-propagaciju literala l. Objašnjenje opet dolazi u obliku (što manjeg) skupa literala $\{l_1, \ldots, l_k\}$ takvog da važi $[] \vDash_{\mathcal{T}} \overline{l}_1 \vee \ldots \vee \overline{l}_k \vee l$ i koristi pri analizi konflikata.
- Operacija kojom se $Solver_{\mathcal{T}}$ obaveštava o uklanjanju nekih literala iz valuacije M.

15.3 Rešavač za linearnu racionalnu aritmetiku zasnovan na simpleks metodi

Kao primer procedure odlučivanja (tj. rešavača $Solver_{\mathcal{T}}$) za jednu konkretnu teoriju izgrađene u skladu sa DPLL(T) arhitekturom, u ovom poglavlju će biti opisana procedura odlučivanja za linearnu racionalnu aritmetiku (LRA) koja je zasnovana na simpleks metodi. Ova procedura je implementirana u okviru rešavača ArgoLib koji kao svoju $DPLL(X)$ komponentu koristi SAT rešavač ArgoSAT. Rešavač ArgoLib zasnovan na ovoj proceduri je učestvovao na i takmičenju SMT rešavača 2007. godine i rezultati su dostupni sa stranice SMT takmičenja `http://www.smtcomp.org`.

15.3.1 O linearnom programiranju i simpleks metodi

Linearna aritmetika. Kao što je već prikazano u poglavlju 15.1.3, *Linearna aritmetika* (nad racionalnim brojevima (LRA) ili nad celim brojevima (LIA)) je fragment aritmetike (nad racionalnim ili celim brojevima) koji uključuje sabiranje, ali ne množenje, osim množenja konstantama. Formule linearne aritmetik su formula logike prvog reda čiji su atomi jednakosti, različitosti ili nejednakosti oblika $a_1x_1 + \ldots + a_nx_n \bowtie b$, gde su $a_1, \ldots, a_n$ i b racionalni

brojevi, $x_1, \ldots, x_n$ su (racionalne ili celobrojne) promenljive i $\bowtie$ je jedan od operatora $=, \leq, <, >, \geq$ ili $\neq$.

Linearna aritmetika (i nad racionalnim i nad celim brojevima) je odlučiva (čak i kada je dozvoljeno korišćenje kvantifikatora u okviru formula). Ovo znači da prostoji *procedura odlučivanja* koja vraća *true* ako i samo ako je ulazna rečenica teorema teorije linearne aritmetike, a koja vraća *false* u suprotnom. Dva najznačajnija metoda za odlučivanje zadovoljivosti formula linearne aritmetike su *Furije-Mockinova procedura (eng. Fourier-Motzkin procedure)* [LM92] i *simpleks metoda (eng. simplex method)* [DM06a].

Linearna aritmetika ima brojne primene verifikaciji softvera i hardvera (naročito njen univerzalno kvantifikovani fragment). Razlozi za ovo su to što se u ovoj teoriji mogu modelovati mnogi tipovi uslova koji nastaju iz potrebe rezonovanja o svojstvima softvera i hardvera, kao i to što je ova teorija odlučiva i postoji dugačka tradicija proučavaja njenih procedura odlučivanja. Procedure odlučivanja za LRA su mnogo brže od procedura odlučivanja za LIA.

Linearno programiranje. *Linearno programiranje*, poznato još i kao linearna optimizacija, je problem maksimizacije ili minimizacije linearne funkcije na konveksnom poliedru koji je definisan putem linearnih ograničenja. Problem linearnog programiranja se sastoji od niza linearnih nejednakosti nad realnim promenljivim i linearne funkcije (definisane nad istim realnim promenljivima) koju je potrebno maksimizovati. Problem linearnog programiranja, u svojoj standardnoj formi, je maksimizovati funkciju datu sa $c^t x$, zadovoljavajući uz to uslovi $Ax \leq b$, za $b \geq 0$, $x \geq 0$, gde su x, b i c vektori iz $\mathbb{R}^n$ i A je realna $m \times n$ matrica.

Simpleks metoda. *Simpleks metoda (eng. simplex method)* se ubraja među najznačajnije algoritme prethodnog veka.[3] To je metoda za rešavanje linearnih problema optimizacije [Dan63] i njena složenost najgoreg slučaja je eksponencijalna u odnosu na broj promenljivih [KMS72]. Uprkos tome, metoda se pokazuje veoma efikasnom u praksi i konvergira u polinomijalnom vremenu za veliki broj ulaznih problema, uključujući i određene klase slučajno generisanih problema ([NW99], [FGW02]). Pored osnovne varijante simpleks metode namenjene za rešavanje problema optimizacije, formulisane su i mnoge druge varijante uključujući i varijantu koja rešava problem odlučivanja za linearna ograničenja tj. odgovara na pitanje da li je dat skup linearnih ograničenja (linearnih jednakosti i nejednakosti) zadovoljiv.

Simpleks metoda za problem optimizacije iterativno pronalazi *mogućna rešenja (eng. feasible solutions)* koja zadovoljavaju sva data ograničenja pri čemu se *gramzivo (eng. greedily)* pokušava da maksimizuje funkciju cilja.

Geometrijski posmatrano, skup linearnih nejednakosti definiše zatvoreni konveksni politop (koji se naziva simpleks), koji je definisan kao presek poluprostora n-dimenzionog Euklidskog prostora. Simpleks metoda započinje pretragu u nekom početnom čvoru i kreće se duž stranica politopa dok ne dostigne čvor sa optimalnim rešenjem. U svakoj iteraciji se bira čvor koji je susedan tekućem, ali tako da se ne umanji vrednost ciljne funkcije. Ukoliko takav čvor ne postoji, konačno rešenje problema je pronađeno. Obično postoje više susednih čvorova

[3] Na primer, časopis *Computing in Science and Engineering* ga je uvrstio u spisak deset najznačajnih algoritama dvadesetog veka

koji ne umanjuju ciljnu funkciju i potrebno je odabrati neki od njih. Ovaj izbor je određen *pravilom pivotiranja (eng. pivot rule)* i u praksi se koriste različita ovakva pravila.

Osnovna varijanta simpleks metode se modifikuje tako da može da rešava još neke probleme osim standardnog problema optimizacije linearnog programiranja (na primer, neke promenljive mogu da budu bez uslova, neki koeficijenti b_i mogu biti negativni, moguće je tražiti minimalno umesto maksimalnog rešenja, itd).

Problem odlučivanja za linearnu aritmetiku se svodi na to da se pronađe makar jedno mogućno rešenje. *Dualni simpleks (eng. dual simpleks)* [Lem54] je naročito pogodna varijanta u slučajevima kada se uslovi dodaju inkrementalno. Nakon dodavanja novog uslova, problem koji je prethodno bio rešen se ponovno optimizuje, ali krenuvši od prethodno rešenja, tako da se do novog rešenja znatno efikasnije stiže.

15.3.2 Opis rešavača

U nastavku će biti opisan $Solver_{LRA}$ koji je zasnovan na posebnoj varijanti dualnog simpleks metoda koju su razvili Duterte (eng. Duterte) i de Mora (eng. de Moura) i koja je upotrebljenja u njihovom SMT rešavaču Yices [DM06b]. Procedura se sastoji iz faze pretprocesiranja i faze rešavanja.

Pretprocesiranje. Prvi korak procedure je da se ulazna formula Φ prevede u ekvizadovoljivu formulu $\Phi_= \wedge \Phi'$, gde je $\Phi_=$ konjunkcija linearnih jednakosti, a Φ' je formula u kojoj su iskaznim operatorima povezani *elementarni atomi* oblika $x_i \bowtie b$, gde je x_i promenljiva, a b je racionalna konstanta. Ova transformacija se vrši prilično pravolinijski tako što se za svaki linearni term t_i koji nije promenljiva a koji se pojavljuje kao leva strana atoma $t_i \bowtie b$ u formuli Φ uvodi do tada nekorišćena promenljiva s_i.

Primer 15.6. Ako je $\Phi \equiv x \geq 0 \wedge x + y < 0 \wedge 2x + 3y > 1$, tada je $\Phi' \equiv x \geq 0 \ \wedge \ s_1 < 0 \ \wedge \ s_2 > 1$, a $\Phi_=$ je $s_1 = x + y \wedge s_2 = 2x + 3y$.

U narednom koraku pretprocesiranja, sve nejednakosti oblika $x \neq b$ se prevode u $x < b \ \vee \ x > b$. Nakon toga se svaka stroga nejednakost oblika $x < b$ zamenjuje sa $x \leq b - \delta$, gde δ ima ulogu *dovoljno malog* racionalnog broja. Slično, svaka nejednakost oblika $x > b$ se zamenjuje sa $x \geq b + \delta$. Nakon ove transformacije, formula Φ' više ne sadrži stroge nejednakosti.

Primer 15.7. Nakon drugog koraka pretprocesiranja, formula Φ' iz primera 15.6 postaje $x \geq 0 \ \wedge \ s_1 \leq -\delta \ \wedge \ s_2 \geq 1 + \delta$.

Broj δ se ne računa unapred, već se sa njim radi na simboličkom nivou. Njegovo efektivno izračunavanje se vrši samo kada se zatraži konkretan, racionalan model formule za koju je ustanovljeno da je zadovoljena nad $\mathbb{Q}$. Ovo znači da se nakon faze pretprocesiranja, sva izračunavanja vrše u polju $\mathbb{Q}_\delta$, gde je Q_δ skup $\{a + b\,\delta \mid a, b \in Q\}$. Dok se operacije sabiranja i oduzimanja elemenata iz Q_δ definišu trivijalno, poredak elemenata skupa Q_δ se definiše na sledeći način: $a_1 + b_1\,\delta \bowtie a_2 + b_2\,\delta$ ako i samo ako je $a_1 \bowtie a_2 \vee (a_1 = a_2 \wedge b_1 \bowtie b_2)$, gde je $\bowtie \in \{\leq, \geq\}$. Može se pokazati da je originalna formula zadovoljiva nad $\mathbb{Q}$ ako i samo ako je formula dobijena njenom transformacijom zadovoljana nad $\mathbb{Q}_\delta$. Za detalje, videti [DM06b].

Inkrementalni simpleks algoritam Formula $\Phi_=$ je konjunkcija jednakosti i ona se ne menja tokom procesa pretrage, tako da je moguće predati je rešavaču pre nego što rešavanje i pretraga za modelom uopšte počne. Neka su $x_1, \ldots, x_n$ sve promenljive koje se javljaju u $\Phi_= \wedge \Phi'$ (tj. sve promenljive koje se javljaju u polaznoj formuli Φ i m dodatnih promenljivih $s_1, \ldots, s_m$). Ukoliko sve promenljive pređu na levu stranu jednačina, formula $\Phi_=$ se u matričnom obliku može predstaviti kao $Ax = 0$, gde je A matrica dimenzija $m \times n$, $m \leq n$ i x je vektor od n promenljivih. No, umesto toga, sistem će biti čuvan u obliku koji je rešen po m promenljivih, tj., tablo koji se izvodi iz matrice A će biti zapisan u obliku:

$$x_i \quad = \quad \sum_{x_j \in \mathcal{N}} a_{ij} x_j, \qquad x_i \in \mathcal{B}.$$

Promenljive sa leve strane će biti nazivane *baznim promenljivim*, a promenljive sa desne strane će biti nazivane *ne-baznim promenljivim*. Tekući skup baznih promenljivih će biti označavan sa $\mathcal{B}$ dok će tekući skup ne baznih promenljivih biti označavan sa $\mathcal{N}$. Bazne promenljive se ne javljaju sa desnih strana tabloa. U prvom koraku, samo će dodatne promenljive s_i biti bazne promenljive.

Formula Φ' je proizvoljna iskazna kombinacija elementarnih atoma oblika $x_i \bowtie b$, pri čemu je $b \in \mathbb{Q}_\delta$. Kao što je već opisano u poglavlju 15.2.1, iskaznu strukturu formule obrađuje $DPLL(X)$ komponenta sistema koja je SAT rešavač, tako da simpleks rešavač samo treba da proverava zadovoljivost konjunkcija elementarnih atoma formule Φ' (pri čemu se ovi atomi dodaju i uklanjaju jedan po jedan). Zbog svoje posebne strukture, ($x \leq u$ ili $x \geq l$), konjunkcije elementarnih atoma određuju gornje i donje granice promenljivih.

Dakle, formula Φ je zadovoljiva ukoliko postoji vrednost $x \in \mathbb{Q}_\delta^n$ koja zadovoljava

$$Ax = 0 \quad \text{i} \quad l_j \leq x_j \leq u_j \text{ for } j = 1, \ldots, n,$$

gde je l_j ili element $\mathbb{Q}_\delta$ ili $-\infty$, a u_j je ili element $\mathbb{Q}_\delta$ ili $+\infty$. Stanje rešavača uključuje:

1. Tablo izveden iz formule $\Phi_=$, zapisan u obliku:

$$x_i \quad = \quad \sum_{x_j \in \mathcal{N}} a_{ij} x_j, \qquad x_i \in \mathcal{B}.$$

2. Poznate donje i gornje granice l_i i u_i za svaku promenljivu x_i, koje su dobijene iz u tom trenutku dodatih atoma formule Φ'.

3. Tekuće valuacije, tj. preslikavanja β koja dodeljuje vrednost $\beta(x_i) \in \mathbb{Q}_\delta$ svakoj promenljivoj x_i.

Na početku, sve donje granice su postavljene na $-\infty$, a sve gornje granice na $+\infty$, dok β dodeljuje nulu svakoj promenljivoj x_i.

Osnovna invarijanta algoritma (uslov koji važi nakon svakog koraka) je da β sve vreme zadovoljava tablo (tj., $A\beta(x) = 0$) i da beta β sve vreme zadovoljava granice (tj., $\forall x_j \in \mathcal{B} \cup \mathcal{N},\ l_j \leq \beta(x_j) \leq u_j$).

Kada se doda novi elementarni atom, stanje se ažurira. Pošto su različitosti i striktne nejednakosti uklonjene tokom faze pretprocesiranja, samo se dodaju jednakosti i nestroge nejednakosti. Dodavanje jednakosti $x_i = b$, se dalje svodi na dodavanje dve nejednakosti $x_i \leq b$ i $x_i \geq b$.

Nakon dodavanja nejednakosti oblika $x_i \leq b$ (slučaj dodavanja nejednakosti oblika $x_i \geq b$ se obrađuje na veoma sličan način), vrednost b se upoređuje sa tekućim granicama za x_i i, ukoliko je potrebno, granice se menjaju:

- Ako je b veće od u_i, nejednakosti $x_i \leq b$ ne donosi nikakvu novu informaciju i stanje se ne menja.
- Ako je b manje od l_i, tada stanje postaje nekonzistentno i detektuje se nezadovoljivost.
- U ostalim slučajevima, gornja granica u_i za promenljivu x_i is se umanjuje i postavlja na vrednost b.

Ako promenljiva x_i nije bazna (tj., kada je $x_i \in \mathcal{N}$) i kada njena vrednost $\beta(x_i)$ ne zadovoljava ažurirane granice l_i ili u_i, vrednost $\beta(x_i)$ je neophodno ažurirati. Ako važi $\beta(x_i) > u_i$ (slučaj $\beta(x_i) < l_i$ se obrađuje na veoma sličan način), vrednost $\beta(x_i)$ se umanjuje i postavlja na u_i. Sa svakom promenom vrednosti ne-baznih promenljivih (u odnosu na preslikavanje β), vrednosti baznih promenljivih je potrebno ažurirati tako da bi tablo ostao zadovoljen.

Problem nastaje ukoliko je x_i bazna promenljiva (tj., kada je $x_i \in \mathcal{B}$) i kada njena vrednost $\beta(x_i)$ ne zadovoljava granice l_i ili u_i. Ukoliko važi da je $\beta(x_i) > u_i$ (slučaj kada je $\beta(x_i) < l_i$ se obrađuje na sličan način), vrednost $\beta(x_i)$ je neophodno umanjiti i postaviti na u_i. Kako bi jednačina tabloa $x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j$ ostala zadovoljena, neophodno je da postoji ne bazna promenljiva x_j takva da je njenu vrednost $\beta(x_j)$ moguće umanjiti (ukoliko je njoj odgovarajući koeficijent a_{ij} pozitivan, tj. $a_{ij} > 0$) ili uvećati (ukoliko je njoj odgovarajući koeficijent a_{ij} negativan, tj. važi $a_{ij} < 0$). Ukoliko ne postoji ne-bazna promenljiva x_j koja dopušta ovakvu vrstu promene (zbog toga što su sve vrednosti promenljivih x_j već postavljene na svoje donje odnosno gornje granice), stanje je nekonzistentno i konstatuje se nezadovoljivost. Ukoliko postoji ne-bazna promenljiva x_j koja dopušta promenu vrednosti, sprovdi se operacija *pivotiranja*. Jednačina $x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j$ se rešava po x_j i promenljiva x_j se uvrštava u sve ostale jednačine tabloa. Tako promenljiva x_j postaje bazna promenljiva, dok x_i postaje ne-bazna promenljiva i njena se vrednost (u preslikavanju β) postavlja na u_i. No, ovo možda dovodi do toga da neka druga bazna promenljiva naruši svoje granice i proces se iterativno nastavlja sve dok sve promenljive ne zadovolje svoje granice ili dok se ne detektuje nezadovoljivost. Varijanta *Blandovog pravila (eng. Bland's rule)* koje se zasniva na fiksiranom poretku među promenljivim se može koristiti kako bi se obezbedilo zaustavljanje procesa.

U ovoj varijanti simpleks metode, tokom operacije povratka u pretrazi (eng. backtracking), neophodno je samo promeniti granice, pri čemu valuacija β i tablo ostaju nepromenjeni i nema potrebe za vršenjem pivotiranja. Ovo je veoma važno svojstvo za efikasnost algoritma.

Objašnjenja dobijenih nezadovoljivosti se generišu na osnovu granica promenljivih koje se javljaju u okviru jednačine koja je postala narušena. Za više detalja o generisanju objašnjena i izvođenju propagacije na osnovu teorije videti [DM06b].

Implementacija opisanog algoritma je data u nastavku. Proceduru `assert` poziva $DPLL(X)$ komponenta kada god je potrebno dodati atom oblika $x_i \bowtie b$.

Ova procedura automatski vši proveru i ažurira granice i vrednosti za sve nebazne promenljive, s obzirom na to da je ova operacija veoma jeftina i da ne zahteva pivotiranje. Procedura `check` se koristi kako bi se proverile granice i ažurirale vrednosti svih baznih promenljivih. Ona vši iterativno menja valuaciju β korišćenjem pivotiranja sve dok sve granice ne postanu zadovoljene ili dok se ne konstatuje nekonzistentnost stanja. Promena vrednosti baznih promenljivih može biti veoma računski zahtevna operacija i proceduru `check` je potrebno pozivati samo s vremena na vreme. Ovo može donekle odložiti detektovanje nezadovoljivosti, ali obično dovodi do opšteg poboljšanja efikasnosti. Procedure `update` i `pivotAndUpdate` su pomoćne procedure.

```
function assert(x_i ⋈ b)
begin
   if (⋈ is =) then
      assert(x_i ≤ b);
      assert(x_i ≥ b)
   else if (⋈ is ≤) then
      if (b ≥ u_i) then return satisfiable;
      if (b < l_i) then return unsatisfiable;
      u_i := b;
      if (x_i ∈ 𝒩 and β(x_i) > b) then
         update(x_i, b)
   else if (⋈ is ≥) then
      if (b ≤ l_i) then return satisfiable;
      if (b > u_i) then return unsatisfiable;
      l_i := b;
      if (x_i ∈ 𝒩 and β(x_i) < b)
         update(x_i, b)
end
```

```
function check()
begin
   loop
      select the smallest x_i ∈ ℬ such that β(x_i) < l_i or β(x_i) > u_i;
      if there is no such x_i then return satisfiable'
      if β(x_i) < l_i then
         select the smallest x_j ∈ 𝒩 such that
            (a_ij > 0 and β(x_j) < u_j) or (a_ij < 0 and β(x_j) > l_j);
         if there is no such x_j then return unsatisfiable;
         pivotAndUpdate(x_i, l_i, x_j);
      if β(x_i) > u_i then
         select the smallest x_j ∈ 𝒩 such that
            (a_ij < 0 and β(x_j) < u_j) or (a_ij > 0 and β(x_j) > l_j);
         if there is no such x_j then return unsatisfiable;
         pivotAndUpdate(x_i, u_i, x_j)
      end loop
   end
```

```
function pivotAndUpdate(x_i, v, x_j)
begin
   θ := (v − β(x_i)) / a_ij;
   β(x_i) := v;
   β(x_j) := β(x_j) + θ;
   for each x_k ∈ B \ {x_i}
      β(x_k) := β(x_k) + a_kj θ;
   pivot(x_i, x_j)
end
```

```
function update(x_i, v)
begin
   for each x_j ∈ B
      β(x_j) := β(x_j) + a_ji (v − β(x_i));
      β(x_i) := v
end
```

Primer 15.8. Proverimo zadovoljivost konjunkcije

$$x \geq 1 \ \wedge \ y \leq 1 \ \wedge \ x + y \leq 0 \ \wedge \ y - x \geq 0.$$

Nakon početne transformacije tablo postaje:

$$\begin{aligned} s_1 &= x + y \\ s_2 &= -x + y \end{aligned}$$

i $\mathcal{B} = \{s_1, s_2\}$, $\mathcal{N} = \{x, y\}$. Formula Φ' je $x \geq 1 \ \wedge \ y \leq 1 \ \wedge \ s_1 \leq 0 \ \wedge \ s_2 \geq 0$.

Početna valuacija je $\beta(x) = 0, \beta(y) = 0, \beta(s_1) = 0, \beta(s_2) = 0$, a početne granice su $-\infty \leq x \leq +\infty$, $-\infty \leq y \leq +\infty$, $-\infty \leq s_1 \leq +\infty$, $-\infty \leq s_2 \leq +\infty$.

Kada se doda atom $x \geq 1$, granice za x postaju $1 \leq x \leq +\infty$, dok valuacija postaje $\beta(x) = 1, \beta(y) = 0, \beta(s_1) = 1, \beta(s_2) = -1$. Nema potrebe za vršenjem operacije pivotiranja.

Kada se doda atom $y \leq 1$, granice za y postaju $-\infty \leq y \leq 1$ i valuacija se ne menja jer y zadovoljava nove granice. Nema potrebe za vršenjem operacije pivotiranja.

Kada se doda atom $s_1 \leq 0$, granice za s_1 postaju $-\infty \leq s_1 \leq 0$. Vrednost $\beta(s_1) = 1$ narušava ove granice i $\beta(s_1)$ je potrebno smanjiti na vrednost 0. Pošto je s_1 bazna promenljiva, neophodno je izvršiti operaciju pivotiranja. Vrednost promenljive x je već na svojoj donjoj granici i nije je moguće smanjiti. Vrednost promenljive y je moguće smanjiti, pa se promenljiva y bira za promenljivu pivotiranja. Nakon pivotiranja, tablo postaje

$$\begin{aligned} y &= s_1 - x \\ s_2 &= -2x + s_1 \end{aligned}$$

i y postaje bazna, a s_1 postaje ne-bazna promenljiva. Izmenjena valuacija postaje $\beta(x) = 1, \beta(y) = -1, \beta(s_1) = 0, \beta(s_2) = -2$.

Napokon, kada se doda atom $s_2 \geq 0$, granice za promenljivu s_2 postaju $0 \leq s_2 \leq +\infty$. Tekuća vrednost $\beta(s_2) = -2$ narušava ove granice i vrednost

$\beta(s_2)$ je potrebno uvećati bar na vrednost 0. Pošto je s_2 bazna promenljiva, neophodno je izvršiti operaciju pivotiranja. Na osnovu jednačine $s_2 = -2x+s_1$, vrednost s_2 je moguće povećati samo ukoliko se x smanji ili ukoliko se s_1 uveća. Pošto je vrednost x već postavljena na svoju donju granicu, a vrednost s_1 već postavljena na svoju gornju granicu, konstatuje se nezadovoljivost.

Objašnjenje detektovane nekonzistentnosti je formula $x \geq 1 \wedge x + y \leq 0 \wedge y - x \geq 0$. Ona je sama za sebe nezadovoljiva i *minimalna* je u smislu da je svaki njen podskup zadovoljiv. Do ovog objašnjenja se dolazi na osnovu granica promenljivih koje učestvuju u jednakosti koja je narušena.

Diskusija i zaključi

16
Pregled relevantnih rezultata

Istorijski pregled razvoja SAT rešavavča. Razvoj SAT rešavača počinje originalnom formulacijom DPLL algoritma 1960-tih godina i traje već skoro pedeset godina. Neki od starijih SAT rešavača zasnovanih na DPLL algoritmu su Tableau (NTAB), POSIT, 2cl i CSAT [GKSS07]. U poslednjih petnaestak godina, svedoci smo značajnog razvoja i unapređenja SAT rešavača zasnovanih na DPLL okviru. Pronađene su mnoge praktične primene ovih alata, što je poguralo razvoj rešavača i rodilo motivaciju za istraživanje sve efikasnijih algoritama. Tokom 1990-tih, nastala je nova generacija rešavača kao što su SATO [Zha97], Chaff [MMZ$^+$01] i BerkMin [GN02] u kojima je značajna pažnja stavljena optimizovanju različitih aspekata DPLL algoritma. Godišnja SAT takmičenja dovela su do istraživanja mnogih novih tehnika, razvoja mnogih pametnih implementacija rešavača, kao i kreiranja širokih korpusa instanci koje se koriste za evaluaciju rešavača (ovi korpusi uključuju kako instance iz realnih, praktičnih primena, tako i teške ručno pravljene i slučajno generisane instance). Neki od najuspešnijih rešavača na skoro održanim SAT takmičenjima, a koji su zasnovani na DPLL algoritmu, su Rsat [PD07], Picosat [Bie08], Minisat [ES04], Tinisat [Hua07a]. SAT4J (`http://sat4j.org`) je rešavač implementiran u jeziku Java u kome je značajna pažnja posvećenja softverskom dizajnu.

Nehronološko vraćanje prilikom pretrage (tj. povratni skokovi vođeni konfliktima) je prvi put predloženo u *problemima zadovoljavanja ograničenja (eng. constraint satisfaction problems (CSP)* [BHZ06]. Ova tehnika je, zajedno sa tehnikom učenja klauza, prvi put uključena u SAT rešavače sredinom 1990-tih kroz rešavač GRASP koji su napravili Silva i Skalah (eng. Silva i Skallah) [MSS99] i rešavač rel_sat [BS97] koji su napravili Bajardo i Šrag (eng. Bayardo i Schrag). Minimalizaciju konfliktnih klauza su uveli Ên i Sorenson (šved. Eén i Sörensson) u njihovom rešavaču MiniSAT [ES04]. Randomizovano otpočinjanje iznova su uveli Gomez (eng. Gomes et al.) i koautori [GSK98], a dalje su ga razvili Baptista i Markes-Silva (eng. Baptista i Marques-Silva) [BMS00]. Shemu dva posmatrana literala uveli su Moskijevič i koautori (eng. Moskewicz et al.) u rešavaču zChaff [MMZ$^+$01].

Pregled opisa SAT rešavača. Originalni opisi DPLL procedure su objavljeni u radovima [DP60, DLL62]. Tokom skoro pola veka dugog perioda razvoja SAT rešavača, objavljeno je mnoštvo radova u kojima je dat opis pojedinačnih

SAT tehnika. Međutim, ono što je u kontekstu ove teze značajnije su pregledni radovi koji daju kompletnije prikaze SAT tehnologije i detaljnije opise konkretnih SAT sistema. Navešćemo neke od takvih.

Pregledni rad [ZM02] opisuje tehnike koje su se pokazale uspešne do početka 2002. godine. Za programere SAT rešavača, naročito značajan je rad [ES04], koji u pseudo jeziku daje detaljan i kompletan opis arhitekture i implementacije modernog SAT rešavača. Posebno poglavlje monografije o reprezentovanju znanja [HLP07] je posvećeno SAT rešavačima [GKSS07]. Takođe, posebno poglavlje monografije posvećene procedurama odlučivanja [KS08] je posvećeno SAT rešavačima. U radu [BHZ06] daje se uporedni pregled tehnika rešavanja problema zadovoljavanja ograničenja (CSP) i rešavanja SAT problema.

Monografija [BHMW09] je u potpunosti posvećena opisu SAT rešavača. U prvom delu je dat detaljan opis istorije problema ispitivanja iskazne zadovoljivosti, opisana je konjunktivna normalna forma formula i odgovarajući algoritmi konverzije, a zatim je dat pregled kako kompletnih, tako i stohastičkih algoritama za ispitivanje iskazne zadovoljivosti. Posebna poglavlja su posvećena modernim modifikacijama DPLL algoritma, uključujući i rešavanje vođeno konfliktima, optočinjanje iznova i slično. Takođe, posebno poglavlje je posvećeno i heuristikama izbora literala. Posebno je razmatran problem rešavanja formula sa pravilnom strukturom (simetričnih formula) kao i problem slučajno generisanih k-SAT formula. Drugi deo monografije je u potpunosti posvećen primenama SAT rešavača.

Iako svi navedeni opisi detaljno opisuju mnoge važne aspekte SAT rešavača, način izlaganja je uglavnom neformalan i opisan, i algoritmi su predstavljeni ili kao nedovoljno razrađen pseudokod ili u obliku opisa na prirodnom jeziku. Mnogo formalniji opisi SAT rešavača zasnovanih na DPLL proceduri su dati kroz apstraktne sisteme opisane u [KG07, NOT06], međutim, oni ne pokrivaju mnoge značajne aspekte implementacije. Smatramo da ova teza prvi tekst koji daje opis mnogih SAT tehnika na precizno definisan, uniforman i formalan način.

Pregled verifikacije SAT rešavača. Prvi koraci u verifikaciji SAT rešavača su napravljeni relativno skoro.

Autori dva sistema promena stanja za SAT koji su opisani u poglavlju 4.4.4 su neformalno dokazali njihovu korektnost [KG07, NOT06]. Unapređenja ovih sistema, takođe opisana u [KG07, NOT06] opisuju i dokazuju korektnost rada SMT rešavača. Žang i Malik (eng. Zhang i Malik) su veoma neformalno dokazali korektnost modernog SAT rešavača [ZM03]. Baret (eng. Barrett) je u svojoj tezi je formalno, ali ne i mehanički, dokazao korektnost Stanfordskog okvira za kombinovanje procedura odlučivanja koji se može koristiti u okviru SMT rešavača [Bar03].

Što se tiče mehaničke formalne verifikacije u okviru sistema za dokazivanje teorema, Leskije i Konšon (fr. Lescuyer i Conchon) su u okviru sistema Coq formalizovali i verifikovali SAT rešavač zasnovan na klasičnoj DPLL proceduri, bez modernih modifikacija [LS08]. Šankar i Voše (eng. Shankar) i (fr. Vaucher) su u okviru sistema PVS formalizovali i verifikovali opis najvišeg nivoa modernog SAT rešavača zasnovanog na DPLL proceduri [SV09]. Iako neki od navedenih radova uključuju opis svih tehnika korišćenih u modernim SAT rešavačima, implementacione tehnike nižeg nivoa nisu pominjane ni u jednom od njih. Ova

teza je prvi rad koji pruža potpuno formalizovane i mehanički proverene dokaze korektnosti modernih SAT rešavača, izvedene na raznim nivoima apstrakcije korišćenjem raznih paradigmi za verifikaciju. Koliko nam je poznato, ova teza je jedini rad u kojoj su verifikovani i svi tehnički detalji implementacije SAT rešavača, od kojih je najzančajnija shema dva posmatrana literala.

Pregled primena SAT rešavača Izvanredan pregled primena SAT rešavača može da se nađe u drugom delu monografije [BHMW09]. Primene uključuju oblasti ograničene provere modela (eng. bounded model checking), planiranja, verifikacije sofvera, dizajna kombinatornih objekata (eng. combinatorial design), statističke fizike. Takođe, opisane su i primene alata koji modifikuju i proširuju SAT rešavače, kao što su SMT rešavači, rešavači za formule opšte iskazne strukture (neklauzalne formule), Max-SAT rešavači, Pseudo-bulovski rešavači i sl.

U okvirima ove teze, najznačajnije su primene SAT rešavača u rešavanju problema raspoređivanja. Čin-a-fat i Hartog (hol. Chin-A-Fat i Hartog) su koristili SAT rešavač za generisanje rasporeda časova. Međutim, sistem koji je u ovoj tezi predložen daje mogućnost specifikovanja mnogo šireg spektra zahteva. Na primer, pristup koji je predložen u [Har07] ne daje mogućnost raspoređivanja časova koji traju više od dva časa, ne daje mogućnosti zadavanja ograničenja na pauze u rasporedu, problem raspoređivanja sala nije obrađen na pravi način, itd. Vreme potrebno samo za fazu SAT kodiranja koje je prijavljeno u [Har07] je nekoliko stotina minuta, dok je u našem sistemu svaka pojedinačna instanca kodirana u vremenu manjem od jednom minuta.

17 Dalji rad

Iako je *formalizacija sistema pravila* urađena prilično iscrpno, određeni aspekti su ostavljeni za dalji rad. Na primer, nije data precizna karakterizacija zaustavljanja sistema koji uključuje i zaboravljanje i otpočinjanje iznova. Za razliku od jednostavnijih sistema za koje je zaustavljanje važilo bez ikakvih dodatnih uslova, dalji rad na istraživanju ovih sistema bi trebalo da obuhvati formulisanje potrebnih i dovoljnih uslova za zaustavljanje ovih komplikovanijih sistema. Takođe, interesantno pitanje je i formalizacija SMT rešavača.

Kako bi se produžio životni vek *rešavača ArgoSAT*, neophodno je konstantno ulaganje u njegov razvoj. Osnovni pravci daljeg rada su implementacija još heurističkih komponentni kao i razvoj grafičkog korisničkog interfejsa.

Jedan od osnovnih pravaca daljeg rada na *implementaciji u okviru logike višeg reda* je poboljšanje efikasnosti generisanog koda.

Prvo, postoji nekoliko algoritamskih unapređenja koje je potrebno sprovesti. Na primer, u tekućoj implementaciji, provera da li je literal tačan u označenoj valuaciji M zahteva linearni obilazak liste. Većina realnih rešavača keširaju istinitosne vrednosti svih literala u nizu što dopušta proveru u konstantnom vremenu. Takođe, implementacija faze analize konflikata je u našoj implementaciji data mala apstraktnije nego što je to slučaj u realnim rešavačima (npr. MiniSAT).

Dalje, neke heuristike moraju da budu pažljivije implementirane. Na primer, testovi koje smo radili su koristili isključivo trivijalnu heuristiku izbora literala (onu koja bira slučajni, do tada nedefinisani, literal). Kako bi se dobio upotrebljiv rešavač, neophodno je koristiti zahtevnije heuristike (npr. VSIDS). Takođe, korisno bi bilo implementirati i tehnike zaboravljanja i otpočinjanja iznova.

Iako ove modifikacije zahtevaju ulaganje dodatnog napora, verujemo da se one mogu izvesti veoma pravolinijski. Sa druge strane, ozbiljniji problem predstavlja činjenica da zbog čisto-funkcionalne prirode logike višeg reda nisu dozvoljeni bočni efekti i nije moguće vršiti *destruktivno ažuriranje* struktura podataka. Kako bi se ovo prevazišlo, neophdno je instruirati generator koda da generiše monadički Haskell, odnosno imperativni ML kôd, za šta je podrška u Isabelle tek skoro implementirana [BKH+08].

Nadamo se da bi kada se ove izmene unesu, generisani kôd mogao postati uporediv po efikasnosti sa nekim modernim SAT rešavačima.

Što se *primena* tiče, jedan od mogućih pravaca daljeg rada je korišćenje SAT rešavača za rešavanje još nekih optimizacionih problema, pored problema raspoređivanja časova. Takođe, verovatno je da bi se bolji rezultati postigli ukoliko bi se umesto SAT koristili SMT rešavači koji bi koristili teorije specijalno prilagođene domenu koji se rešava.

18 Zaključci

Osnovni ciljevi ove teze su da se na sistematičan način opišu moderni SAT rešavači zasnovani na DPLL algoritmu, formalno dokaže njihova korektnost i opišu putevi za efikasnu i fleksibilnu implementaciju, kao i neke njihove praktične primene. U nastavku će biti diskutovano kako ova teza doprinosi realizaciji ovih ciljeva.

Opisi SAT rešavača. Jedan od ciljeva teze, bio je da pruži metodičan opis modernih SAT tehnologija. Kako bi ovaj cilj mogao biti ispunjen, korišćen je pristup izlaganja *„od opšteg ka pojedinačnom" (eng. top-down)* i SAT tehnologije su uvođene postupno, na nekoliko različitih nivoa, sve vreme povećavajući nivo formalnosti i detaljnosti u izlaganju. U uvodnoj glavi 4, prikazani su uobičajeni načini opisivanja SAT rešavača. Na neformalan, ali razumljiv, način uveden je klasični DPLL algoritam, kao i njegove savremene modifikacije. U toj glavi, navedeni su i prigodni primeri koji bi čitaocima trebalo da olakšaju razumevanje rada rešavača. U nastavku teze, nivo formalnosti i detaljnosti u izlaganju je povećavan, zadržavajući metodološki način izlaganja u svim delovima teze, uključujući i one koji su se bavili verifikacijom rešavača. Opisi rešavača dati kroz tezu bi trebalo da budu u potpunosti dovoljni da bi čitaoc bez prethodnog predznanja mogao da se upozna sa svim relevantnim aspektima savremenih SAT rešavača. Prateći pseudokod bi trebalo da pruža sasvim dovoljan osnov za izgradnju i implementaciju modernog SAT rešavača.

Izolovanje koncepata u okviru implementacije SAT rešavača Osnovni cilj o kome je vođeno računa prilikom implementacije SAT rešavača, je da su različite tehnike podržane, koliko je to moguće, nezavisno jedna od druge. U okviru implementacije, poželjna je razdvojenost algoritamskih koraka višeg nivoa od tehničkih detalja implementacije. Ovo se najjasnije postiže ukoliko implementacija striktno prati apstraktni formalni okvir dat kroz sisteme promena stanja. Takođe, sve heuristički aspekti rešavača bivaju identifikovani i izdvojene u nezavisne komponente. Ovo dovodi do kôda koji je jasan, lak za razumevanje, održavanje i korišćenje i o čijim se svojstvima može precizno rezonovati. Dalje, ovakva fleksibilna arhitektura omogućava i istovremeno postojanje više različitih tehnika (posebno heuristika) u okviru istog SAT rešavača.

Koristi od ovoga su višestruke. Prvo, moguće je praviti fer poređenja između različitih tehnika što čini rešavač pogodnim za izvođenje eksperimentalnih istraživanja na polju iskazne zadovoljivosti. Dalje, rešavač je u mogućnosti da između više podržanih tehnika odabere onu koja je najpogodnija za rešavanje date ulazne formule čime se značajno može popraviti efikasnost rešavanja.

Ovakav pristup implementaciji rešavača je sproveden u okviru ove teze. Navedeni principi su predstavljali osnovu za opis implementacije jezgra rešavača koji je dat u glavi 9. Definisano jezgro je zatim nadgrađeno i u glavi 11 je opisana fleksibilna objektno-orijentisana arhitektura realnog SAT rešavača, koja i dalje poštuje sve navedene principe.

Implementacija realnog softverskog sistema. Cilj teze je je i implementacija realnog SAT rešavača. Kao odgovor na ovaj zadatak razvijen je rešavač ArgoSAT, implementiran u jeziku C++koji je direktno zasnovan na fleksibilnoj arhitekturi opisanoj u glavi 11. Arhitektura definiše jasan API rešavača i dopušta njegovu jednostavnu nadgradnju komponentama kao što je (grafički) korisnički interfejs, generator dokaza nezadovoljivosti i slično. Iako postizanje velike brzine rešavanja nije bio prioritetni cilj prilikom izgradnje rešavača ArgoSAT, rezultati testiranja prikazani u glavi 12 ukazuju na to da on jeste uporediv sa današnjim vodećim rešavačima.

Verifikacija SAT rešavača. Cilj teze je i da se formalno dokaže korektnost SAT rešavača, što je i urađeno na nekoliko nivoa, prateći date opise SAT rešavača.

Kao prvi i najznačajniji korak ka ovom cilju, u delu *Formalizacija*, urađena je formalizacija *apstraktnih sistema promena stanja za SAT* i dokazana je njihova korektnost. Osnovna prednost korišćenja apstraktnog opisa je da su rešavači predstavljeni kao matematički objekti tako je da je rezonovanje o njima moguće korišćenjem uobičajenog matematičkog aparata. Takođe, verifikacija apstraktnih opisa predstavlja temelj svih narednih pristupa verifikaciji. Osnovni nedostatak korišćenja apstraktnih opisa je to što postoji značajna razlika između formalizovanih sistema i konkretnih rešavača. Sistemi promena stanja nisu direktno izvršivi i veliki broj detalja je neophodno precizirati kako bi se na njihovim osnovama mogli nadgraditi realni, izvršivi SAT rešavači.

Mnogo precizniji opis rešavača u odnosu na apstraktne sisteme promena stanja daje imperativni pseudokod prikazan u glavi 9. U okviru rada na ovoj tezi, sprovedena je njegova verifikacija u *Horovom stilu*, kao što je to opisano u poglavlju 9.10. Osnovna prednost korišćenja ovakvog načina verifikacije je to što omogućava dokazivanje korektnosti imperativnog koda koji se koristi u većini realnih implementacija SAT rešavača. Međutim, i dalje postoji značajan raskorak između sprovedenog dokaza u Horovom stilu i korektnosti realne C/C++ implementacije. Naime, ne postoji formalna veza između prikazanog pseudokoda i realnog programskog koda. Takođe, tokom same verifikacije postojao je veliki broj ručno izvedenih koraka prilikom formulisanja uslova korektnosti i njihovog međusobnog spajanja, što narušava pouzdanost same verifikacije.

Kako bi se postigao najveći mogući nivo pouzdanosti u verifikaciji SAT rešavača, u glavi 10 izvršena je njihova verifikacija tehnikom *plitkog utapanja (eng. shallow embedding)* u logiku, tj. njihova definicija u okviru logike višeg

reda i dokaz korektnosti tako izgrađenih definicija. Ovaj pristup je prvo sproveden za klasični DPLL algoritam, a zatim i za moderni SAT rešavač koji je zasnovan na imperativnom pseudokodu prikazanom u glavi 9. Osnovna prednost koju donosi verifikacija putem utapanja u logiku je to što je, kada se SAT rešavač definiše unutar dokazivača teorema, moguće izvršiti njegovu verifikaciju direktno unutar logike dokazivača teorema i nije potrebno koristiti bilo kakvu vrstu operacione ili denotacione semantike programskog jezika. Takođe, na osnovu formalne definicije rešavača je moguće generisati izvršni kôd u nekom od funkcionalnih programskih jezika i za generisani kôd se sa veoma visokim stepenom pouzdanosti može tvrditi da je korektan. Nedostaci ovog pristupa verifikaciji su to što zahteva kreiranje nove implementacije (odnosno definicije) SAT rešavača u logici dokazivača teorema. Takođe, neophodno je koristiti specijalne tehnike kako bi se omogućilo korišćenje struktura podataka čiju je vrednost moguće menjati i time omogućilo generisanje efikasnog koda.

Iako postoje i drugi pokušaji verifikacije SAT i SMT rešavača (opisani u glavi 16), verifikacija urađena u sklopu ove teze je otišla najdalje u tom pravcu. Kao najznačajnije aspekte ovde opisane verifikacije izdvajaju se činjenica da su svi dokazi provereni u okviru pomoćnika pri dokazivanju, kao i činjenica da ova teza prva daje dokaze korektnosti za mnoge implementacione tehnike niskog nivoa, među kojima je najznačajnija shema dva posmatrana literala. Celokupna verifikacija je dostupna u [Mar08] i kao zanimljivost vredi navesti da dokazi sadrže preko 30.000 linija Isabelle kôda, što čini oko 700 strana u PDF formatu.

Primene SAT rešavača. Jedan od ciljeva teze bio je i da se demonstrira moć savremenih SAT rešavača kroz neke od njihovih praktičnih primena.

U glavi 13 opisana je tehnika za konstruisanje rasporeda časova zasnovana na kodiranju uslova korektnosti i udobnosti rasporeda u okviru iskazne logike. Iako je naknadno ustanovljeno da je već bilo pokušaja primene ove tehnike, naš sistem daje značajno bolje rezultate od svih ranije predloženih sličnih sistema. Činjenica da je tokom prethodne dve godine na ovaj način konstruisano desetak rasporeda časova za 3 fakulteta i jednu srednju školu u Beogradu, pokazuje da je ova tehnika primenljiva u praksi.

U glavi 15, dat je opis upotrebe SAT rešavača u okviru SMT rešavača koji služe za ispitivanje zadovoljivosti uslova izraženih na bogatijem jeziku od jeziku iskazne logike — SMT rešavači ispituju zadovoljivost formula logike prvog reda u odnosu na određene odlučive teorije prvog reda. Rešavač ArgoSAT uklopljen je u biblioteku ArgoLib, implementirana je procedura odlučivanja za linearnu aritmetiku nad racionalnim brojevima zasnovana na simpleks metodi i prikazana je mogućnost praktičnog ispitivanja zadovoljivosti formula ove teorije.

Glavna poruka ove teze je da je moguće izgraditi SAT rešavač tako da njegova implementacija eksplicitno prati apstraktni formalni opis, ali sadrži i sve implementacione tehnike niskog nivoa. Ovako izgrađen rešavač je moguće formalno verifikovati, a sa druge strane on može biti dovoljno efikasan i imati mnogobrojne praktične primene.

Bibliografija

[BMS00] L. Baptista and J. P. Marques-Silva. Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability. In *CP '00*, LNCS 1894, pp. 489–494, Singapore, 2000.

[Bar03] C. Barrett. Checking validity of quantifier-free formulas in combinations of first-order theories. *Ph.D. thesis, Stanford University*, 2003.

[BS97] R. J. Jr. Bayardo and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *14th AAAI*, pp. 203–208, Providence, 1997.

[BKS04] P. Beame, H. Kautz and A. Sabharwal Towards Understanding and Harnessing the Potential of Clause Learning *Journal of Artificial Intelligence Research 22*, pp. 319–351, 2004.

[Bie08] A. Biere. PicoSAT essentials. *JSAT 4*, pp. 75–97, 2008.

[BHMW09] A. Biere, M. Heule, H. van Maaren and T. Walsh. *Handbook of Satisfiability* IOS Press, Amsterdam, 2009.

[Bla77] R. G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research 2(2)*, pp. 104–107, 1977.

[BRJ05] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, 2005.

[BHZ06] L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional Satisfiability and Constraint Programming: A Comparative Survey. *ACM Comput. Surv. 38(4)*, 2006.

[BKH+08] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkok, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *TPHOLs '08*, LNCS 5170, Montreal, 2008.

[Cla08] E. Clarke. The Birth of Model Checking. In *25 Years of Model Checking*, LNCS 5000, pp. 1–26, 2008.

[CE82] E. Clarke, E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic*

of Programs, Workshop, LNCS 131, pp. 52–71, Yorktown Heights, 1982.

[Coo71] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *3rd STOC*, pp. 151–158, New York, 1971.

[Dan63] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[Dav94] M. Davis, R. Sigal, E. Weyuker. *Computability, Complexity, and Languages (Fundamentals of Theoretical Computer Science).* Morgan Kaufmann/Academic Press, 1994.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Commun. ACM 5(7)*, pp. 394–397, 1962.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM 7(3)*, pp. 201–215, 1960.

[DM06a] B. Dutertre and L. De Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV 2006*, LNCS 4144, Seattle, 2006.

[DM06b] B. Dutertre and L .De Moura. Integrating Simplex with DPLL(T). *Technical Report SRI-CSL-06-01*, SRI International, 2006.

[ES04] N. Een and N. Sorensson. An Extensible SAT Solver. In *SAT '03*, LNCS 2919, pp. 502–518, S. Margherita Ligure, 2003.

[Eme08] E. A. Emerson. The Beginning of Model Checking: A Personal Perspective. In *25 Years of Model Checking*, LNCS 5000, pp. 27–45, 2008.

[Flo67] R. Floyd. Assigning Meanings to Programs. In *Sympos. Appl. Math. Vol. XIX*, pp. 19–32, 1967.

[FGW02] A. Forsgren, P. E. Gill, and M. H. Wright. Interior methods for nonlinear optimization. *SIAM Rev 44*, pp. 525–597, 2002.

[FB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. *Refactoring: Improving the Design of Existing Code* Addison-Wesley, 1999.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras and C. Tinelli. DPLL(T): Fast Decision Procedures. In *CAV'04*, LNCS 3114, pp. 175–188, Boston, 2004.

[Gel07] A. van Gelder. Verifying Propositional Unsatisfiability: Pitfalls to Avoid. In *SAT '07*, LNCS 4501, pp. 328–333, Lisbon, 2007.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A Rast and Robust SAT Solver, In *DATE'02*, pp. 142–149, Paris, 2002.

[GKSS07] C P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability Solvers. In *Handbook of Knowledge Representation*, Elsevier, 2007.

[GSK98] C. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search through Randomization. In *15th AAAI*, pp. 431–437, Madison, 1998.

[Gor00] M. Gordon. From LCF to HOL: a short history In *Proof, Language, and Interaction, Essays in Honour of Robin Milner.*, pp. 169–186, The MIT Press, 2000.

[Haf08] F. Haftmann. Code generation from Isabelle/HOL theories. `http://isabelle.in.tum.de/documentation.html`, 2008.

[HLP07] F. Harmelen, V. Lifschitz, B. Porter. *Handbook of Knowledge Representation*, Elsevier, 2007.

[Har08] J. Harrision. Formal Proof —- theory and practice. In *Notices of the AMS 55(11)*, 2008.

[Har07] J. Hartog. Timetabling on Dutch High Schools: Satisfiability versus gp-Untis. *Msc. thesis, Technical University Delft*, The Netherlands, 2007.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM 12(10)*, pp. 576–580, 1969.

[Hua07a] J. Huang. A Case for Simple SAT Solvers. In *CP '07*, LNCS 4741, pp. 839–846, Providence, 2007.

[Hua07b] J. Huang. The Effect of Restarts on the Efficiency of Clause Learning. In *IJCAI '07*, pp. 2318–2323, Hyderabad, 2007.

[HHH+06] F. Hutter, Y. Hamadi, H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *CP'06*, pp. 213–228, 2006.

[KMS72] V. Klee, G. J. Minty, and O. Shisha. How good is the simplex algorithm? *Inequalities 3*, pp. 159–175, 1972.

[Kra08] A. Krauss. Defining recursive functions in Isabelle/HOL. `http://isabelle.in.tum.de/documentation.html`, 2008.

[KS08] D. Kroening, O. Strichman. *Decision Procedures*. Springer, 2008.

[KG07] S. Krstić and A. Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *FroCos '07*, LNCS 4720, pp. 1–27, Liverpool, 2007.

[LM92] J. L. Lassez and M. J. Maher. On Fourier's algorithm for linear arithmetic constraints. *Journal of Automated Reasoning 9*, pp. 373–379, 1992.

[LJ94] J. Launchbury and S. Peyton Jones. Lazy Functional State Threads. In *PLDI'94*, Sigplan notices 29(6), pp. 24–35, Orlando, 1994.

[Lem54] C. E. Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, pp. 36–47, 1954.

[LS08] S. Lescuyer and S. Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *TPHOLs'08: Emerging Trends*, Montreal, 2008.

[Mar05] F.Marić. Implementacija shema za ugradnju procedura odlučivanje u automatske dokazivače teorema. *Magistarska teza, Matematički fakultet, Univerzitet u Beogradu*. 2005.

[Mar08] F. Marić, SAT Solver Verification. *The Archive of Formal Proofs,* `http://afp.sf.net/entries/SATSolverVerification.shtml`.

[VJMT08] M. Vujošević-Janičić, F. Marić, D. Tošić. Using Simplex Method in Verifying Software Safety. In *Balcor'07*, pp. 305–319, Belgrade-Zlatibor, 2008.

[Mar09a] F. Marić. Formalization and Implementation of SAT solvers. *Journal of Automated Reasoning 43(1)*, pp. 81–119, Springer, 2009.

[Mar09b] F. Marić. Flexible Implementation of SAT solvers. Manuscript in preparation.

[Mar09c] F. Marić. Formal Verification of a Modern SAT Solver. Manuscript submitted.

[Mar09d] F. Marić. Timetabling Based on SAT Encoding: a Case Study. Manuscript submitted.

[Mar09e] F. Marić. SAT Based Link-Grammar Parser. Report of the Google Summer of Code Project.

[MJ04] F. Marić, P. Janičić. ARGO-LIB: a generic platform for decision procedures. In IJCAR'04, LNCS 3097, Cork, 2004.

[MJ09a] F. Marić and P. Janičić. Formal Correctness Proof for DPLL Procedure. *Informatica*. To appear. 2009.

[MJ09b] F. Marić, P. Janičić. SAT Verification Project. Manuscript submitted.

[MJ09c] F. Marić, P. Janičić. Formalization of Abstract State Transition Systems for SAT. In preparation.

[MSS99] J P. Marques-Silva and K A. Sakallah. Grasp: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Computers 48(5)*, pp. 506–521, 1999.

[SS96] J. P. Marques Silva and K. A. Sakallah. Conflict Analysis in Search Algorithms for Satisfiability. In *8th ICTAI*, pp. 467–469, Toulouse, 1996.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC '01*, pp. 530–535, Las Vegas, 2001.

[NO79] G. Nelson, D. Oppen. Simplification by cooperating decision procedures. In *ACM Transactions on Programming Languages and Systems 1*, pp. 245–257, 1979.

[NOT06] 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 ACM 53(6)*, pp. 937–977, 2006.

[Nik08] M. Nikolić. Metodologija izbora pogodnih vrednosti parametara SAT rešavača. *Magistarska teza, Matematički fakultet, Univerzitet u Beogradu*, 2008.

[NMJ09] M. Nikolić, F. Marić, P. Janičić. Instance-Based Selection of Policies for SAT Solvers. In *SAT'09*, LNCS 5584, Swansea, Springer, 2009.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS 2283, Springer, 2002.

[NW99] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.

[Pau89] L. C. Paulson. The foundation of a generic theorem prover. In *Journal of Automated Reasoning 5*, pp. 363—397, 1989.

[PD07] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *SAT '07*, LNCS 4501, pp. 294–299, Lisbon, 2007.

[SV09] N. Shankar and M. Vaucher. The mechanical verification of a DPLL-based satisfiability solver. In preparation.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *CP'05*, pp. 827–831, Sitges, 2005.

[ST91] D. Sleator, D. Temperley. Parsing English with a Link Grammar. *Carnegie Mellon University Computer Science technical report CMU-CS-91-196*. October 1991.

[Wie06] F. Wiedijk. *The Seventeen Provers of the World*. Foreword by Dana S. Scott, Springer, 2006.

[XHH+08] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-Based Algorithm Selection for SAT. *JAIR 32*, pp. 565–606, 2008.

[Zha97] H. Zhang. SATO: An Efficient Propositional Prover. In *CADE-14*, LNCS 1249, pp. 272–275, Townsville, 1997.

[ZM02] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *CAV '02*, LNCS 2404, pp. 17–36, Copehagen, 2002.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using independent resolution-based checker. In *DATE '03*, p10880, Washington DC, 2003.

[ZMMM01] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *ICCAD '01*, pp. 279–285, San Jose, 2001.

[VSTTE] *Verified Software: Theories, Tools, Experiments.* Conference. `http://vstte.ethz.ch/`

A Terminološki rečnik

A.1 Englesko-srpski

accepting state	prihvatajuće stanje
to assert a literal	dodati literal u valuaciju
backjumping	povratni skokovi
backjump level	nivo povratnog skoka
backjump clause	klauza povratnog skoka
backtrack search	pretraga sa vraćanjem
branching	grananje
cardinality constraint	uslov kardinalnosti
completeness	potpunost
conflict	konflikt
conflict-driven learning	učenje vođeno konfliktima
conflict clause	konfliktna klauza
conjunctive normal form (CNF)	konjunktivna normalna forma (KNF)
consistent valuation	neprotivrečna valuacija
constraint	ograničenje
constraint satisfaction problem (CSP)	problem zadovoljavanja ograničenja
clause	klauza
dead lock	kružna blokada
decision literal	pretpostavljeni literal
decision strategy	strategija za izbor literala
elimination rule	pravilo eliminacije
feasible solution	mogućno rešenje
final state	završno stanje
floating point	pokretni zarez
formula	formula
greedy algorithm	gramzivi algoritam
hard constraint	tvrdo ograničenje
heap	hip
implied literal	izvedeni literal
idle period	pauza u rasporedu
inference	zaključivanje
initial state	početno stanje
interface	interfejs
literal	literal
lexicographic extension	leksikografsko proširenje relacije
monadic programming	monadičko programiranje
multiset extension	multiskup proširenje relacije

opposite literal	suprotan literal
phase caching	keširanje polariteta
pivot	pivot, pivotiranje
proof assistant	asistent u dokazivanju teorema
proof-checking	provera dokaza
pure literal	literal jedinstvenog polariteta
rejecting state	odbacujuće stanje
restarting	otpočinjanje iznova
SAT solver	SAT rešavač
satisfiability	zadovoljivost
satisfiable formula	zadovoljiva formula
search	pretraga
shallow embedding	plitko utapanje
side effect	bočni efekat
soft constraint	meko ograničenje
soundness	saglasnost
split rule	pravilo isprobavanja
state	stanje
state transition system	sistem promene stanja
subsumption	obuhvatanje
termination	zaustavljanje
transition rule	pravilo prelaska
trusted kernel	poverljivo jezgro
two-watch literal scheme	shema dva posmatrana literala
unit clause	jedinična klauza
unit literal	jedinični literal
unit propagation queue	red jediničnih literala
unique implication point	tačka jedinstvene implikacije
unsatisfiable formula	nezadovoljiva formula
valuation	valuacija
variable	promenljiva
well-founded relation	dobro zasnovana relacija

A.2 Srpsko-engleski

bočni efekat	side effect
dobro zasnovana relacija	well-founded relation
dodati literal u valuaciju	assert literal
formula	formula
gramzivi algoritam	greedy algorithm
grananje	branching
hip	heap
interfejs	interface
izvedeni literal	implied literal
jedinična klauza	unit clause
jedinični literal	unit literal
keširanje polariteta	phase caching
klauza	clause
klauza povratnog skoka	backjump clause
konflikt	conflict
konfliktna klauza	conflict clause
konjunktivna normalna forma (KNF)	conjunctive normal form (CNF)
kružna blokada	deadlock
leksikografsko proširenje relacije	lexicographic extension
literal	literal
literal jedinstvenog polariteta	pure literal
meko ograničenje	soft constraint
mogućno rešenje	feasible solution
monadičko programiranje	monadic programming
multiskup proširenje relacije	multiset extension
neprotivrečna valuacija	consistent valuation
nezadovoljiva formula	unsatisfiable formula
nivo povratnog skoka	backjump level
obuhvatanje	subsumption
odbacujuće stanje	rejecting state
ograničenje	constraint
otpočinjanje iznova	restarting

pauza u rasporedu	idle period
pivot	pivot
plitko utapanje	shallow embedding
početno stanje	initial state
pomoćnik pri dokazivanju teorema	proof assistant
potpunost	completeness
povratni skokovi	backjumping
poverljivo jezgro	trusted kernel
pokretni zarez	floating point
pravilo eliminacije	elimination rule
pravilo isprobavanja	split rule
pravilo prelaska	transition rule
prihvatajuće stanje	accepting state
pretpostavljeni literal	decision literal
pretraga	search
pretraga sa vraćanjem	backtrack search
problem zadovoljavanja ograničenja	constraint satisfaction problem (CSP)
promenljiva	variable
provera dokaza	proof-checking
red jediničnih literala	unit propagation queue
saglasnost	soundness
SAT rešavač	SAT solver
shema dva posmatrana literala	two-watch literal scheme
sistem promene stanja	state transition system
stanje	state
strategija za izbor literala	decision strategy
suprotan literal	opposite literal
tačka jedinstvene implikacije	unique implication point
tvrdo ograničenje	hard constraint
učenje vođeno konfliktima	conflict-driven learning
uslov kardinalnosti	cardinality constraint
valuacija	valuation
zaključivanje	inference
zadovoljiva formula	satisfiable formula
zadovoljivost	satisfiability
zaustavljanje	termination
završno stanje	final state