
Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Tržaška 25, Ljubljana

seminarska naloga

ZANESLJIVOST
PROGRAMSKE OPREME

zmogljivost in vrednotenje računalniških sistemov

Ljubljana, januar 2007

Avtor: Črtomir Gorup
Vpisna številka: 63030160

Kazalo

1	Uvod	3
2	Zanesljivost programske opreme	3
2.1	Kaj je zanesljiva programska oprema?	4
2.2	Razvojni cikel zanesljive programske opreme	4
2.3	Študije primerov nezanesljive programske opreme	5
2.3.1	informacijski sistem Virtual Case File	5
2.3.2	sistem za razvrščanje prtljage na letališču Denver	6
2.3.3	obsevalna naprave THERAC-25	7
2.4	Zakaj je programska oprema nezanesljiva?	7
2.5	Zagotavljanje zanesljivosti programske opreme	8
3	Programski paket Alloy Analyzer	9
3.1	Splošno o programu	9
3.2	Delovanje	10
3.2.1	Definicija abstraktnega modela	10
3.2.2	Kako deluje analiza	12
3.3	Primer analize datotečnega sistema	13
4	Zaključek	15
5	Viri in literatura	15

1 Uvod

V današnjem svetu računalniki vodijo letala, upravljajo z bančnimi transakcijami, skrbijo za komunikacijo ter opravljajo veliko podobnih kritičnih del. Napake programske opreme v zgornjih primerih so katastrofalne. Neizbežna je velika finančna škoda, zelo pogoste so tudi človeške žrtve. Zaradi tega bi morali več pozornosti nameniti zanesljivosti programske opreme.

Seminarska naloga je razdeljena na dva dela. V prvem bom pokazal, da je nezanesljivost skoraj vedno posledica konceptualnih napak že v sami zasnovi programske opreme. Predstavil bom nekaj znanih primerov neuspešnih informacijskih projektov ter analiziral vzroke za njihov neuspeh. V drugi polovici bom predstavil programski paket Alloy Analyzer, ki služi za analizo logičnih modelov programske opreme. Podal bom primer modela ter njegovo analizo z programom Alloy Analyzer.

2 Zanesljivost programske opreme

Napake v programski opremi niso nič novega. Pojavljati so se začele z razvojem računalništva v petdesetih letih, stopnjevale s krizo programske opreme v šestdesetih letih ter dočakale novo tisočletje. V začetku so nekateri raziskovalci upali, da so napake v programski opremi posledica neznanja programerjev in da se bo situacija z leti normalizirala. Stanje danes kaže, da je bila hipoteza napačna. Če na razvoj programske opreme pogledamo širše hitro opazimo, da je najpogostejši vzrok napak sama kompleksnost programske opreme. Izdelava programske opreme se od gradnje hiše razlikuje po tem, da zanjo ne obstaja točno predpisan postopek izdelave ali načrt. V primerjavi s hišo je programska oprema abstrakten objekt, zato tudi zelo težko točno ovrednotimo njeno kvaliteto. Tako kot pri izdelavi in vrednotenju se tudi pri testiranju pojavljajo razlike med programsko opremo ter ostalimi objekti. Samo delovanje in testiranje programske opreme je močno odvisno od strojne opreme ter vhodnih podatkov. Problem predstavlja tudi dejstvo, da na podlagi uspešnega testiranja z nekim naborom vhodnih podatkov brez natančne analize ne moremo trditi, da se programska oprema podobno obnaša za podoben nabor vhodnih podatkov. Vzpodbudo nam dajejo formalni načini dokazovanja pravilnosti, enega si bomo ogledali v nadaljevanju.

Glavni vzrok krize programske opreme je, da so računalniki postali nekajkrat zmogljivejši! Naj povem drugače: dokler ni bilo računalnikov nismo imeli problemov s programiranjem; ko smo imeli nekaj manj zmogljivih računalnikov, je programiranje predstavljalo srednje težek problem, in sedaj, ko imamo gigantske računalnike, je programiranje postalo gigantski problem! Edsger Dijkstra, 1972

2.1 Kaj je zanesljiva programska oprema?

Zanesljivost je ena izmed zelo zaželenih lastnosti programske opreme. Zanesljiva programska oprema nam tudi ob neznanih vhodnih podatkih zagotavlja pravilno in deterministično delovanje, tako kot je določeno v njeni specifikaciji. Načrtovanje in razvoj zanesljive programske opreme ni lahko delo. Vemo, da programska oprema opravlja kritična dela ponavadi neprimerna za ljudi. Kljub težavnosti pa od nje pričakujemo opravljeno delo v krajšem času z nič ali manj napakami kot bi jih pri enakem delu storil človek. Problem se pojavi tudi pri ocenjevanju zanesljivosti programske opreme, saj tudi tukaj ne obstaja noben metrični sistem. Na zanesljivost moramo paziti v vseh razvojnih fazah programske opreme. Med načrtovanjem moramo predvideti vse faktorje, ki bodo med delovanjem vplivali na obnašanje programske opreme. Žal pa lahko njeno dejansko zanesljivost vidimo, ko je programska oprema v uporabi. Za mnoge primere je takrat že prepozno.

2.2 Razvojni cikel zanesljive programske opreme

Kljub temu, da za razvoj zanesljive programske opreme nimamo točno določenega postopka, obstajajo smernice katerih se velja držati. Smernice sem razdelil po posameznih fazah razvoja, v nadaljevanju jih bom na kratko predstavil.

- analiza
Od programske opreme ne moremo pričakovati zanesljivega delovanja v vsaki situaciji, če še njeni načrtovalci ne poznajo vseh situacij ter željenih odzivov nanje. Cilj analize je torej specifikacija ciljev in nalog programske opreme ter določitev njenega pravilnega odziva na vsako možno situacijo.
- načrtovanje
V tej fazi že vemo kaj od programske opreme pričakujemo, potreben je še načrt kako naj programska oprema cilje uresniči. Velike probleme in naloge obvladujemo tako, da jih pretvorimo v bolj abstraktno obliko. S tem ločimo dejanski problem od samega kodiranja. Naloge, cilje in probleme razbijemo na manj manjše enote ter tako zagotovimo modularnost in povečamo preglednost in kvaliteto.
- kodiranje
Ob dobrem poznavanju načrta iz prejšnje faze je kodiranje rutinskega značaja. Nerazumevanje delovanja programske opreme ter njenih funkcij lahko privede do pojava, ko se konceptualne napake iz analize in načrtovanja prenesejo na končni produkt. Kljub temu, da računalnik ne pozna koncepta "lepo napisane kode" in zanj oblika kode ni pomembna je s človeškega stališča drugače. Zaradi potreb po vzdrževanju in spreminjanju programske kode so se oblikovala pravila lepega kodiranja. Splača se jih držati.
- testiranje

Testiranje posameznih modulov se lahko izvaja že med programiranjem, najboljše testiranje pa sledi na koncu. Dandanes se zelo uveljavljajo automatizirani testi, saj so hitrejši in učinkovitejši. V grobem ločimo dva tipa testov. Prvi so testi modulov kjer preverjamo delovanje posameznih delov programske opreme. V naslednjem koraku sledijo integracijski testi, ki preverjajo skladnost delovanja že prej preverjenih modulov.

2.3 Študije primerov nezanesljive programske opreme

Načrtovanju in izdelavi programske opreme je posvečeno veliko pozornosti, vendar še vedno prihaja do napak v končnih izdelkih. Nekaj takih z velikimi posledicami si bom predstavil v nadaljevanju.

2.3.1 informacijski sistem Virtual Case File

Septembra 2000 so pri ameriški vladni agenciji FBI začeli z 380 milijonov dolarjev vrednim informacijskim projektom Trillogy. Projekt je bil zasnovan v treh delih. Prvi del je predvideval posodobitev računalniške opreme, drugi posodobitev omrežnega sistema in telekomunikacij, tretji pa je vključeval posodobitev programske opreme. Takrat so na agenciji uporabljali informacijski sistem ACS iz leta 1970, ki je kljub omejenim funkcionalnostim in starosti je deloval stabilno. V posodobitvi je bil predviden prehod na podatkovno bazo Oracle, dodatne funkcionalnosti ter vmesnik za dostop preko spleta. Po terorističnih napadih 11 septembra so vodilni možje v FBIju spremenili svoje želje. Za povečanje državne varnosti so si zamislili popolnoma nov informacijski sistem imenovan Virtual Case File na kratko VCF. Informacijski sistem VCF naj bi po zmogljivosti močno prekašal sistem ACS. Agentom bi nudil popolno informacijsko podporo v vseh fazah njihovih primerov. Vgrajeno naj bi imel podporo za rudarjenje podatkov ter avtomatsko povezovanje podobnih primerov med sabo. Agencija FBI in podjetje ACIS sta skupaj sestavili 800 strani dolgo specifikacijo zahtev za katero se je kasneje izkazalo, da je nejasna in polna napak. Po dveh letih dela so pri podjetju ACIS ugotovili, da so v zamudi, ameriški kongres je problem rešil z dodatnimi 123 milijoni ameriških dolarjev. Podjetje je najelo dodatne programerje in poskušalo zamujeno nadoknaditi. Vmes se na agenciji FBI zamenjalo tudi nekaj vodilnih mož. Njihove želje in potrebe glede novega informacijskega sistema so se zato velikokrat spremenile.

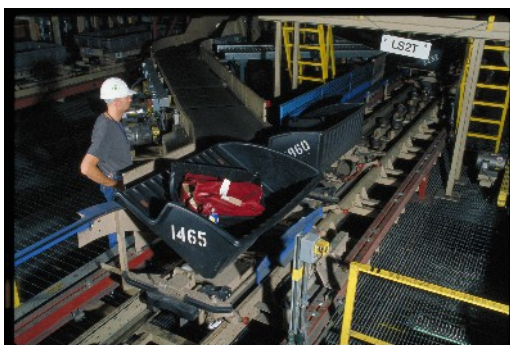
Decembra 2003 je podjetje agenciji sicer dobavilo nov informacijski sistem, vendar izkazal se je, da je neuporaben. V iskanju krivca za polomijo so se obrnili na neodvisno arbitražo, ki je našla napake na obeh vpletenih straneh. Informacijski sistem so sicer poskušali popraviti, vendar se je vodstvo agencije januarja 2005 odločilo za dokončno opustitev projekta.

Prva dva dela projekta Trillogy sta bila kljub precej višjim stroškom dokončana. Agencija v povezavi z informacijskim sistemom VCF uradno priznava materialno škodo v

višini 104 milijona dolarjev. Neodvisni analitiki zatrjujejo naj bi bila ta vsaj dvakrat višja. Za neuspeh projekta je več razlogov. Zagotovo je eden največjih nejasne in spreminjajoče se specifikacije zahtev agencije FBI. Po drugi strani je podjetje ACIS izdelalo slab in pomankljiv načrt. Sprotne konceptualne napake so reševali z dodajanjem programerjev.

2.3.2 sistem za razvrščanje prtljage na letališču Denver

Če bi bilo vse po načrtih bi morale biti letališče v Denverju eno izmed najmodernejših na svetu. Zasluge bi imel sistem za avtomatsko razvrščanje prtljage, ki naj bi zanj skrbel od prihoda letala do izročitve potniku. Med načrtovanjem je bil poudarek na popolni avtonomnosti ter hitrosti sistema. Najdaljši načrtovani čas za premik vse prtljage določenega leta med različnimi izhodnimi vrati je bil 6 minut. Kljub temu, da je bil sistem izdelan ni daloval po pričakovanih. Prtljago je po letališču premikalo 3100 tirnih vozičkov navadne velikosti ter 450 večjih vozičkov. Dolžina vseh tirov je bila 25 kilometrov, za sortiranje je bilo uporabljenih še 9 kilometrov tekočih trakov. Sistem je krmililo 300 računalnikov 486 distribuiranih v osmih kontrolnih centrih. Med seboj so bili povezani z hitrim optičnim omrežjem. Vozički so bili opremljeni z RFID transponderji, prtljaga pa z črtnimi kodami. Za sledenje je sistem uporabljal 400 RFID ter 56 laserskih čitalcev. Med nalaganje in razlaganjem prtljage se vozički niso ustavljali, njihova hitrost se je le znižala. Normalna delavna hitrost za premikanje po tirih je bila 30 km/h.



Slika 1 Del sistema za avtomatsko razvrščanje prtljage

Problemi so se pojavljali že med gradnjo, vendar so projekt vseeno dokončali. Kljub temu so programske in strojne napake povzročale velike preglavice in sistem je bil skoraj neuporaben. Prtljaga je bila velikokrat narobe razvrščena, poškodovana ali pa je padala z vozičkov. Prtljava na tirih je povročala hujše okvare vozičkov in celotnega sistema. Pokazalo se je tudi, da omrežje in računalniki niso vedno sposobni obdelati vseh ključnih informacij v realnem času. Zaradi tega je prihajalo do napak tudi med dinamičnim nalaganjem prtljage na vozičke. Vozički so velikokrat zaostajali ali prehitevali in prtljaga je padala po tleh ali v druge vozičke. Zakasnitve v procesiranju podatkov in omrežju so ob konicah povzročile zaletavanje vozičkov in dodatno zmedo.

Nikoli popolnoma delujoc avtomatskega razvrščanja prtljage je investitorje stal 234 milijona dolarjev. Napovedana otvoritev Denverskega letališča je bila zaradi nedelujočega sistema prestavljena iz marca 1994 na februar 1995. Ocenjena škoda zaradi zaprtega letališča znaša milijon dolarjev na dan. Tudi po otvoritvi sistem avtomatskega razvrščanja prtljage ni deloval zanesljivo. Na letališču ga je uporabljala samo letalska družba United Airlines, pa še to samo za odhodne lete. Podjetje BAE Automated Systems of Carrollton je po tem projektu šlo v stečaj, letalska družba United Airlines, njihova največja stranka, pa je bankrotirala. Lani poleti je vodstvo letališka dokončno odpovedalo projekt ter prešlo na ročen način razvrščanja prtljage.

2.3.3 obsevalna naprave THERAC-25

Leta 1982 je začelo podjetje Atomic Energy of Canada Limited prodajati računalniško vodene obsevalne naprave Therac-25 za zdravljenje rakavih obolenj. Naprave so omogoče dve vrsti in različne stopnje obsevanj. Prvi problem se je pojavil leta 1985, ko je pacientka iz neznanega vzroka dobila preveliko dozo sevanja. Incidenti so se nadaljevali, kasneje so ugotovili, da so bili nekateri pacienti obsevani tudi z 17000 radi. Normalna doza znaša 200 radov. Obsevalne naprave so v redkih okoliščinah sevale z ogromno močjo. V dobrem letu so obsevalne naprave Therac-25 ranile oziroma povzročile smrt šestih ljudi.

Po vsakem incidentu so bolnišnice kontaktirale podjetje AECL, ki je v začetku trdno zavračalo možnost napake v svojem izdelku. Po drugem incidentu so se v podjetju vseeno odločili, da v bolnišnico pošljejo svojega tehnika. Tehnik napake ni uspel ponoviti ponoviti, zato je še vedno veljajo da so obsevalne naprave brez napake. Podjetje je sicer preventivno izvedlo minimalne posodobitve strojne opreme, vendar so se napake še vedno pojavljale. Po petem incidentu v bolnišnici v Texasu sta se fizik Fritz Hager in njegov pomočnik, oba zaposlena v bolnišnici, odločila poiskati smrtno kombinacijo ukazov in nastavitvev. Po mnogo urah preiskovanja sta ugotovila, da je delovanje obsevalne naprave odvisno od hitrosti vnašanja ukazov. Obstajal je primer, ko vnešene vrednosti na zaslonu niso bile skladne s dejanskimi parametri v računalniku. Dejanski prikaz napačnega delovanja je bil za podjetje AECL dovoljšen dokaz, začeli so s prenovo programske opreme.

Kasneje so ugotovili, da je programsko kodo razvijal en sam človek. Testiranje so izvajali 2700 ur, kar je za tak kritičen sistem premalo. Očitamo jim lahko tudi napačen pristop do testiranja, saj so sistem testirali kot celoto in ne po modulih. Programsko opremo so popravili in obsevalne naprave so bile v uporabi še mnogo let.

2.4 Zakaj je programska oprema nezanesljiva?

Napake v programski opremi niso nov pojav. Pojavljati so se začele z začetki računalništva v 50. letih, ko je bilo napak zaradi manjšega števila računalnik in enostavnejše

programske opreme manj. Od takrat dalje se število računalnikov hitro povečuje. Prav tako narašča kompleksnost programske opreme. Posledica tega je veliko več napak ter hujše posledice. Napak poznamo več vrst, odkrijemo jih lahko v vseh fazah razvoja programske opreme ali pa celo med samim delovanjem. Kasneje kot napako odkrijemo, težje jo odpravimo. Včasih odprava ene napake povzroči druge neskladnosti in posledično več novih napak. Natančen pogled pokaže, da lahko za vse kasneje odkrite večje napake najdemo vzrok v slabem načrtovanju pred kodiranjem. Take napake imenujemo konceptualne napake, njihovo odpravljanje je najbolj zahtevno.

FBIjev informacijski sistem VCF je tipičen slabega načrtovanja. Problemi so se začeli že v osnovi, saj so bile zahteve naročnikov nejasne. Posledica tega je bil slabo izdelan in nedorečen načrt programske opreme. Taka osnova ni je skorajšnje zagotovilo za slab končni produkt. Malo drugače je bilo v primeru projekta avtomatskega razvrščanja prtljage na letališču Denver. Naročnik in podjetje BAE Automated Systems so si že v osnovi zamislili zelo kompleksen sistem. Na svetu do danes še ni narejenega takega sistema za avtomatsko razvrščanje prtljage. Na treh manjših letališčih po svetu danes uporabljajo podoben sistem, le da je ta veliko manjši in enostavnejši. Kljub kompleksnosti problema bi verjetno leta 1993 s takratno tehnologijo in z mnogo več časa vloženega v načrtovanje in analizo dosegli zastavljene cilje. Pri obsevalnih napravah Therac-25 je zanimivo, da je najzahtevnejši del programske opreme deloval pravilno. Napake, ki so bile v razmeroma enostavnem uporabniškem vmesniku lahko pripisemo površnosti pri načrtovanju, izdelavi in testiranju.

2.5 Zagotavljanje zanesljivosti programske opreme

S problemom zanesljive programske opreme se raziskovalci ukvarjajo že veliko let. Začetnik je Alan Turing, ki je v 40. letih dokazal da je problem zaustavljivosti programa neodločljiv. Program, ki bi za vse ostale programe določil ali se kdaj ustavijo ali ne torej ne obstaja. V 60. letih je veliko znanstvenikov in raziskovalcev začelo posvečati več časa formalnemu dokazovanju pravilnosti programov. Kmalu se je izkazalo, da je za obvladovanje večjih problemov potrebna obravnava na višjem, bolj abstraktnem nivoju. Ta ideja je danes osnova UML modeliranja in objektnega programiranja. Zadnje čase se vse več pozornosti namenja programom, ki znajo analizirati abstraktne modele drugih programov in tako ugotoviti neskladnosti še pred kodiranjem. Pri večini mora uporabnik najprej izdelati model programa ter podati pravila ter zakonitosti za posamezne spremenljivke. Analizator v naslednjem koraku simulira različne poteke delovanja modela programa ter poišče stanja, v katerih spremenljivke kršijo katero izmed pravil. Analizatorji take vrste ne preverjajo programske kode, zato nimamo nobenega zagotovila, da program dejansko deluje. Prepričani smo le, da v abstraktni zasnovi programa ni naskladij. Orodij za analizo zasnove programa je več, jaz bom v nadaljevanju na kratko predstavil programski paket Alloy. V spodnji tabeli je navedenih nekaj podobnih orodij.

programski jezik	orodje	avtor
Zing	Zing	Microsoft Research
Z	Jaza	University of Waikato
PROMELA	Spin	Bell laboratories
FSP	LTSA	Imperial College London
OCL	USE	University of Bremen
B	Pro-B	University of Southampton

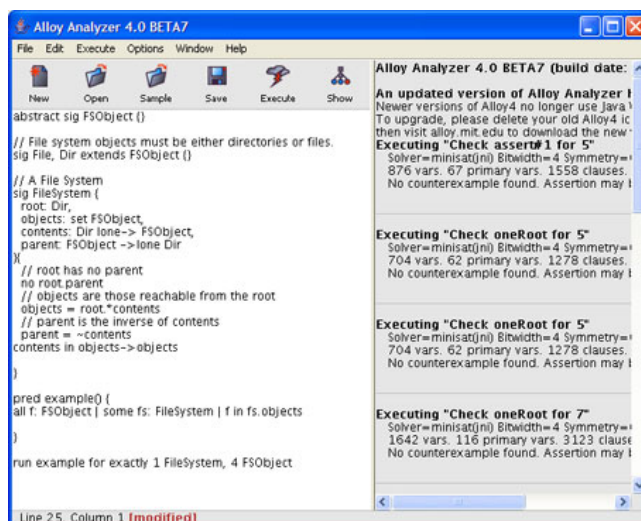
Tabela 1 Orodja za preverjanje zasnove programske opreme

3 Programski paket Alloy Analyzer

3.1 Splošno o programu

Leta 1999 je skupina raziskovalcev univerze Massachusetts Institute of Technology pričela z razvojem analizatorja abstraktnih modelov Alloy Analyzer. Program je do danes desegel četrto različico. Njegovi popularnosti zagotovo pripomore dejstvo, da je uporaben na veliko različnih področjih. Lahko ga uporabljamo za analizo telekomunikacijskih protokolov, testiranje zasnove programske opreme, reševanje logičnih nalog.

Sam izgled programa daje uporabniku videz enostavnosti. Na osnovnem oknu najdemo gumba za shranjevanje in odpiranje modelov, vnosno polje za urejanje definicije modela in gumb za začetek analize. V programu obstaja še eno okno, to je okno za pregled rezultatov analize. Rezultati analize so lahko primeri ali protiprimeri izpeljani iz konkretnega abstraktnega modela, ogledamo si jih lahko v obliki grafa ali v format XML.



Slika 2 Osnovno okno programa Alloy Analyzer

Za analizo mora načrtovalec najprej s posebnim strukturnim jezikom izdelati model. Definirati mora razrede, ki v njem nastopajo, povezave med njimi ter dejstva, ki v modelu vedno veljajo. Podati je potrebno tudi maksimalno število generiranih objektov oziroma zgornjo mejo preiskovanja prostora ter predpostavke katere bomo kasneje preverili. Analizator omogoča dva načina delovanja. V prvem načinu analizator poišče vse primerke, ki ustrezajo zgornjim omejitvam. Drugi način pa služi za dokazovanje nepravilnosti predpostavk. V tem primeru nam analizator poišče protiprimerke do vnaprej določene velikosti, ki ustrezajo dejstvom ne ustrezajo pa tudi predpostavkam.

3.2 Delovanje

3.2.1 Definicija abstraktnega modela

Pri analizi abstraktnega modela je zagotovo najpomembnejša sama definicija modela. V njej določimo različne tipe razredov, interakcije med njimi in omejitve. Za iskanje primerov in protiprimerov so v abstraktnem modelu nujno potrebne še predpostavke in predikati. V nadaljevanju bom na kratko opisal vsakega izmed gradnikov abstraktnega modela.

- razred (Signature)

Razredi so v abstraktnem modelu osnovni gradniki. Njihova definicija je sestavljena iz imena in telesa. V telesu so deklarirane spremenljivke razreda in relacije, ki jih razred uporablja. Celotna zasnova razredov močno spominja na samo objektno programiranje. Tudi tukaj imamo možnost abstraktnih razredov ter dedovanja.

V spodnjem okvirčku je primer razreda `FileSystem`. Vsaka instanca `FileSystem` razreda ima vsebuje spremenljivko `root` tipa `Dir`, množico objektov tipa `FSObject` ter dve množici povezav. Prva povezuje mape z datotekami, druga pa datoteke in mape z mapami.

```
sig FileSystem {
  root: Dir,
  objects: set FSObject,
  contents: Dir lone-> FSObject,
  parent: FSObject ->lone Dir
}
```

- dejstvo (Fact)

Eni izmed gradnikov mimo katerih ne moremo so tudi dejstva. Vsako dejstvo je sestavljeno iz imena in množice logičnih trditev, ki si jih lahko predstavljamo kot

omejitve objektov. Poudariti je potrebno, da dejstva veljajo vedno in za vse objekte ne glede ali iščemo primere ali protiprimere. Med analizo Alloy zavrže vse objekte, ki ne ustrezajo dejstvom. Tukaj lahko naredimo napako in definiramo dejstvo, ki je vedno trivialno napačno. Kljub temu, da je predpostavka mogoče napačna Alloy v tem primeru ne najde nobenih protiprimerov.

V spodnjem primeru z dejstvom zagotovimo povezavo med datotekami in mapami. Za vsako mapo d in posameznim elementom njene vsebine o mora veljati starševska zveza.

```
fact defineContents {
  all d: Dir, o: d.contents | o.parent = d
}
```

- predpostavka (Assertion)

Predpostavke so množice logičnih trditev, ki naj bile posledica dejstev. Z ukazom `Check ime_predstavke For n` lahko preverimo ali obstaja kakšen protiprimer, ki ustreza dejstvom in ne ustreza predpostavki. Velikost morebitnega protiprimera je največ n .

V spodnjem primeru predpostavka `acyclic` predpostavlja, da ne obstaja nobena mapa, ki bila sama v sebi. Z drugimi besedami, v strukturi datotečnega sistema ni ciklov.

```
assert acyclic {
  no d: Dir | d in d.^contents
}
```

- predikat (Predicate)

Predikati so sestavljeni iz imena, argumentov ter seznama logičnih trditev. Vresnost predikata je lahko samo `true` ali `false`. Med trditvami v predikatu velja konjunkcija.

Spodnji predikat preveri, če je datotečni sistem FS' res enak datotečnemu sistemu FS z odstranjeno datoteko f . Preverjanje poteka z dvema trditvama. Prva zahteva sploh vsebovanost datoteke f v datotečnem sistemu FS . Druga pa preverja enakost vsebine datotečnega sistema FS in FS' .

```
pred rm (fs, fs': FileSystem, f: File) {
  f in fs.objects
  fs'.contents = fs.contents - f.(fs.parent)->f
}
```

- funkcija (Function)

Funkcije so v abstraktnih modelih precej podobne metodam. Razlikujejo se samo v tipih vrednostih, ki jih vračajo. Za razliko od predikatov, ki so lahko le pravilni ali napačni funkcije vračajo vse podatkovne tipe. Vračanje ne poteka tako kot pri navadnih proceduralnih jeziki, vrednosti se vračajo skozi argumente.

V spodnjem primeru funkcija na datotečnem sistemu FS izvede premik datoteke ali mape f v mapo d . Rezultat je nov datotečni sistem shranjen v spremenljivki FS' . Sprememba tudi tukaj poteka v dveh delih. Najprej se preveri, ali sta objekt f in mapa d sploh na voljo v datotečnem sistemu FS. V naslednjem koraku se novemu datotečnemu sistemu FS' priredi starega z odstranjeno staro relacijo med mapo in objektom ter dodano novo.

```
fun move (fs, fs': FileSystem, f: FSObject, d: Dir) {
  (f + d) in fs.objects
  fs'.contents = fs.contents - f.(fs.parent)->f + d->f
}
```

- ukaz (Run ali Check)

V vsakemu modelu je natanko eden ukaz za analizo. Z uporabo ukaz `Check ime_predpostavke` `For n` sprožimo iskanje protiprimera do velikosti n . Če tak protiprimer bo ustrezal vsem dejstvom, ne pa predpostavki. V določenih primerih nas zanimajo primeri, ki ustrezajo določenim zahtevam. Z ukazom `Run ime_predikaza` `For n` dobimo vse tiste primere, ki ustrezajo dejstvom in izbranemu predikatu. Število objektov v primeru pa je manjše ali enako n .

```
Check acyclic For 5
```

3.2.2 Kako deluje analiza

Bistvo programa Alloy Analyzer se skriva v reševanju problema valuacije Boolovih enačb (ang. SAT problem). Analiza modela se začne z pretvorbo modela v sistem Boolovih enačb prvega reda. Pred pričetkom reševanja se enačbe poenostavijo, ter pretvorijo

v konjunktivno normalno obliko. V primeru da želimo z analizo odkriti protiprimerse se enačbe predpostavk dodatno negirajo ter dodajo k preostalim enačbam. Tako je problem iskanja protiprimerse preveden v problem valuacije Boolovih enačb. Enačbe v naslednjem koraku obdela poseben algoritem imenovan SAT analizator, ki poišče take vrednosti spremenljivk, da so vse Boolove enačbe pozitivne. Ko imajo vse spremenljivke prirejene prave vrednosti jih program pretvori v nam razumljivejšo obliko.

Že veliko let se ve, da je SAT problem eden tipičnih predstavnikov NP-polnih problemov. Kljub temu, da je za SAT problem NP-poln in da je v zadnjih letih vedno več hitrejših in učinkovitejših algoritmov njegova točna zgornja meja za časovno zahtevnost ostaja še vedno eksponentna. Zaradi tega so načrtovalni programa v Alloy implementirali več najbolj znanih SAT analizatorjev, ki jih lahko izbiramo v nastavitvah. Kljub temu se ob začetku analize z dodatnim ukazom priporoča omejitev števila objektov v modelu. Tipične analize modelov trajajo od nekaj mikro sekund do nekaj ur.

3.3 Primer analize datotečnega sistema

Poglejmo si praktični primer uporabe analizatorja Alloy na poenostavljenem modelu datotečnega sistema. Model je v grobem sestavljen iz treh razredov, dveh predikatov in ene predpostavke. V nadaljevanju bom najprej podal podroben opis zatem pa še dejansko kodo modela. Model datotečnega sistema je precej poenostavljen, omogoča samo eno operacijo to je premik map. Z analizo bomo ugotovili, da tukaj obstaja možnost napake. Nikjer namreč implicitno prepovedali premika mape same vase. Alloy Analyzer nas bo na to opozoril.

Osnovna gradnika datotečnega sistema sta razreda File in Dir. Oba sta izpeljana iz abstraktnega razreda Object. Največji razred je razred FS, ki vsebuje tudi tri notranje spremenljivke. Spremenljivki dirs in files sta množici objektov tipa Dir oziroma File. Zanimivejša je spremenljivka contains, ki je pravtako množica. Elementi množice so preslikave, ki slikajo mape v mape ali datoteke. S pomočjo te množice preslikav dobimo strukturo datotečnega sistema.

Naslednja zanimiva stvar je predikat move_dir, ki sprejme štiri argumente. Spremenljivki fs in fs' sta tipa FS, spremenljivki d in to pa tipa Dir. Predikat drži kadar je datotečni sistem fs' enak datotečnemu sistemu fs po premiku mape d v mapo to. Pogoji za tako situacijo so opisani s tremi logičnimi trditvami. Prva pravi, da morata biti ciljna in končna mapa sploh v fs datotečnem sistemu. V drugi trditvi vsebini novega datotečnega sistema priredimo vsebini prejšnjega z nekaj spremembami. Potrebno je spremeniti mapo katera se preslika v mapo d. Z zadnjo vrstico zagotovimo, da se število map in datotek v fs in fs' datotečnih sistemih enako. Za veljavnost predikata morajo biti izpolnjeni vsi trije pogoji. Naslednji predikat reachable velja ko v datotečnem sistemu fs obstaja taka mapa, ki vključuje vse ostale mape in datoteke. To pomeni, da ne obstaja nobena mapa ali datoteka do katere ne mogli priti.

Preostane nam še predpostavka `assert_OK`, ki predpostavlja povezanost datotečnega sistema tudi po spremembi lokacije katerekoli mape. Formalen zapisan pravi, da če datotečni sistem `fs` zadošča predikatu `reachable` in če v njem premaknemo katerokoli mapo dobimo datotečni sistem `fs'`, ki mora tudi zadoščati predikatu `reachable`.

V zadnji vrstici analiziramo predpostavko `assert_OK`, katera se v nadaljevanju izkaže za napačno.

```
module filesystem

abstract sig Object {}
sig File, Dir extends Object {}

sig FS {
  dirs: set Dir,
  files: set File,
  contains: dirs -> (dirs+files)
}

pred move_dir(fs,fs':FS,d,to:Dir){
  d+to in fs.dirs
  fs'.contains = fs.contains - Dir->d + to->d
  fs'.files = fs.files and fs'.dirs = fs.dirs
}

pred reachable(fs:FS){
  some root:fs.dirs | fs.(dirs+files) in root.*(fs.contains)
}

assert move_OK{
  all fs,fs':FS,d,to:Dir | reachable[fs] and
  move_dir[fs,fs',d,to] implies reachable[fs']
}

check move_OK
```

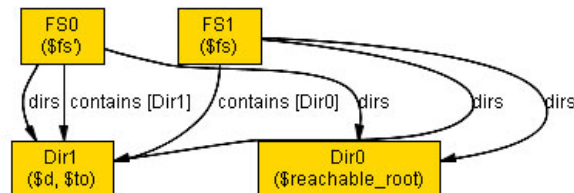
Analizator po 50ms vrne naslednji odgovor. Iz njega lahko razberemo da je našel protiprimer, naša predpostavka o doseglivosti datotek in map ne drži. Za reševanje je analizator uporabil minisat algoritem.

```

Executing "Check move_OK"
  Solver=minisat(jni) Bitwidth=4 Symmetry=ON
  785 vars. 69 primary vars. 1537 clauses. 2183ms.
  Assertion is invalid. 50ms.

```

Poglejmo si protiprimer v obliki grafa.



Slika 3 Rezultat analize predpostavke v obliki grafa

Na grafu vidimo da imamo dve stanji označeni z FS. To si lahko predstavljamo kot dva datotečna sistema. Eden(desni) je pred premikom mape, drugi(levi) je po premiku mape. Iz slike lahko razberemo nastali problem. V modelu nismo nikjer prepovedali premika mapa same vase. V tem primeru mapa ni več dosegljiva in datotečni sistem ne zadošča predikatu `assert_OK`.

4 Zaključek

Kljub temu, da se Alloy Analyzer dandanes večinoma uporablja v akadamske name- ne lahko vidimo da je korak v pravo smer. Uporaba programske opreme za kritična opravila narašča iz dneva v dan in kmalu bomo postali od nje tako odvisni, da si niti najmanjših napak ne bomo smeli privoščiti. Zagotovo je, da bo treba za odpravo napak storiti nekaj pri sami zasnovi programske opreme. Ravno tukaj bodo orodja za formalno dokazovanje pravilnosti delovanja pokazala svojo moč. Analizatorji, ki so danes na voljo kot eksperimentalni programi bodo čez nekaj let lahko že našli svoj prostor v praksi. S hitrejšimi računalniki in učinkovitejšimi algoritmi se bo njihov obseg modeliranja povečal in tako postal dovoljšen za realne kompleksnejše primere. Mogoče bodo sanje o programski opremi brez napak enkrat postale resničnost.

5 Viri in literatura

Literatura

- Dependable Software by Design, Daniel Jackson, Scientific American, June 2006
- Vodenje projektov, prof. Franc Solina, Založba FE in FRI
- Who Killed the Virtual Case File?, Harry Goldstein, IEEE Spectrum, September 2005

Spletni viri

- http://en.wikipedia.org/wiki/Software_reliability
- http://ardent.mit.edu/airports/ASP_papers/Bag_System_at_Denver.PDF
- <http://alloy.mit.edu>
- http://en.wikipedia.org/wiki/Software_crisis
- http://en.wikipedia.org/wiki/Virtual_Case_File
- http://en.wikipedia.org/wiki/Boolean_satisfiability_problem