LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

GALĪGU DETERMINĒTU AUTOMĀTU BŪVES UZDEVUMU AUTOMĀTISKA VĒRTĒŠANA

BAKALAURA DARBS

Autors: **Artūrs Jānis Pētersons**
Studenta apliecības Nr. ap14050
Darba vadītājs: Dr.sc.comp. Agris Šostaks

RĪGA 2018

ANOTĀCIJA

Studējot datorzinātņu teorētiskos pamatus, neatņemama sastāvdaļa ir galīgu determinētu automātu apgūšana, kas noved pie Tūringa mašīnu teorijas un datora darbības pamatprincipiem. Studentiem automātu apgūšanas procesā viena no komponentēm ir šādu automātu konstruēšana un attiecīgi pasniedzējiem – šo automātu vērtēšana. Tā kā datorzinātnes ir ļoti populāras un ir daudz studentu, tad pasniedzējiem jāvelta daudz darba, lai šos risinājumus novērtētu. Determinētu automātu vērtēšanu varētu veikt automatizēti, ietaupot pasniedzēju laiku un samazinot kļūdu iespējamību.

Bakalaura darba mērķis ir izstrādāt automātisku vērtēšanas sistēmu. Darba ietvaros tika analizēti esošie risinājumi automātu vērtēšanai kā arī tika strādāts pie praktiskas vērtēšanas sistēmas izstrādes.

Atslēgas vārdi: Galīgs determinēts automāts, uzdevumi, vērtēšana, optimizācija.

ABSTRACT

AUTOMATED GRADING OF DETERMINISTIC FINITE AUTOMATA.

Deterministic finite automata are crucial in studying theoretical foundations computer science. It eventually leads to Turing machines and basic principles of computers. For students a part of learning about automata is constructing such automata, and the counterpart for teachers – grading the solutions. Since computer sciences are very popular and there are many students pursuing them, it is a lot of effort to grade all these solutions. In case of deterministic automata this process could be automated thus saving time and avoiding errors.

Goal of this bachelor thesis is to develop an automated grading system. To achieve this goal existing solutions were evaluated as well as development was done on implementing an automated grading system.

Keywords: Deterministic finite automata, problems, grading, optimization.

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

API	–	funkciju kopums, kas ļauj sazināties ar programmatūru vai sistēmu
BFS	–	grafu apstaigāšanas algoritms meklējot plašumā
GDA	–	galīgs determinēts automāts
GNA	–	galīgs nedeterminēts automāts
GoLang	–	*Google* izstrādāta programmēšanas valoda
HTTP	–	hiperteksta pārraides protokols
JSON	–	datu struktūras pieraksts
MOSEL	–	MSO paplašinājums, kas ļauj ieviest dažādas komponentes un rīkus
MSO	–	otrās pakāpes loģika ar operācijām pār kopām
TED	–	izmaiņu skaits, kas nepieciešams, lai pārveidotu vienu koku par otru (*tree edit distance*)
YAML	–	datu struktūras pieraksts

IEVADS

Datorzinātnes mūsdienās attīstās ļoti strauji. Zinātnieku un programmētāju komandas veido jaunas tehnoloģijas, ko pēc tam izmantos citas programmētāju komandas savā ikdienas darbā. Pēdējā laikā ļoti daudz tiek runāts par programmētāju darba kvalitāti – piegādātā programmatūra mēdz būt nekvalitatīva vai netiek nodota termiņos. Lai gan šajā darbā apskatītā problēma tiešā veidā nerisina šīs problēmas, tā ir svarīga datorzinātnes sastāvdaļa. Datoru teorētiskie pamati balstās uz 20.gs. 30.gados izstrādāto Tūringa mašīnas matemātisko modeli [1]. Izpratne par Tūringa mašīnām balstās automātu teorijā, kas mūsdienās tiek pasniegta datorzinātņu studentiem.

Lai trenētu un pārbaudītu studentu izpratni par tēmu, neatņemama šādu kursu sastāvdaļa ir praktiskie automātu būves uzdevumi. No studenta perspektīvas ir ļoti būtiski uzkrāt pēc iespējas vairāk zināšanu un prasmju par tēmu aktīvi trenējoties būvēt automātus. Bieži teorētiskos konceptus ir vieglāk izprast pēc praktiskas to izmēģināšanas. Bet, tā kā studiju kursi mēdz būt diezgan pilni un viena automāta novērtēšana var būt ļoti laikietilpīga, rodas ļoti daudz problēmas kursu pasniedzējiem. Pirmkārt, visu darbu novērtēšana ir ļoti laikietilpīga. Otrkārt, ilglaicīgi veicot vienu un to pašu procesu, palielinās iespēja, ka vērtējumā būs kāda kļūda neuzmanības vai noguruma dēļ. Automatizētas vērtēšanas sistēmas ieviešana risinātu abas šīs problēmas. Programmatūra varētu novērtēt daudzus darbus ievērojami ātrāk kā to dara viens cilvēks, kā arī programmatūra nenogurst – vērtējumu precizitāte un objektivitāte nemainīsies atkarībā no darbu daudzuma.

Šī bakalaura darba mērķis ir pētīt praktiskās problēmas, kādas rodas galīgu determinētu automātu vērtēšanas gaitā, un censties izstrādāt strādājošu vērtēšanas sistēmu, kas atvieglotu pasniedzēja darbu un dotu papildu iespēju studentiem apgūt automātu teoriju. Šī mērķa sasniegšanai vispirms nepieciešams veikt teorētisko izpēti par iespējamiem algoritmiem, ko varētu izmantot GDA automātiskai vērtēšanai, sākot ar līdzīgu pētījumu izpēti un alternatīvu iespēju izskatīšanu. Pēc iepazīšanās ar teorētiskajiem principiem šādas sistēmas izstrādei, jāveic programmēšanas darbs, lai realizētu algoritmus.

Šis bakalaura darbs ir balstīts uz Artūra Jāņa Pētersona 2017.gadā izstrādāto kursa darbu "*Galīgu determinētu automātu būves uzdevumu automātiska vērtēšana*". Iepriekšējā darbā tika apskatīts viens iepriekš veikts pētījums – "*Automated grading of DFA constructions*" [2], kas risina šo pašu problēmu, kā arī kursa darbā izteikti novērojumi un komentāri par izmantotajām metodēm. Balstoties uz pētījumu un kursa darbu bakalaura darbā tika veikta vērtēšanas sistēmas izstrāde.

Izstrādātās vērtēšanas sistēmas pamatā ir trīs metožu apvienojums. Katra no šīm metodēm vērtē konkrētu iespējamo kļūdu veidu. Šo trīs metožu apvienojums labi spēj novērtēt gandrīz korektus risinājumus. Pirmā veida kļūdas, kādas tiek meklētas ir sintakses kļūdas studenta iesniegtajā risinājumā. Iespējams, ka ideja risinājuma pamatā ir pareiza un tas novests līdz korektam risinājumam, taču pieļauta neliela sintakses kļūda risinājuma pierakstā. Otrā no trim metodēm cenšas noskaidrot, vai students nav pārpratis uzdevuma nosacījumus un pareizi izpildījis līdzīgu, bet nedaudz atšķirīgu uzdevumu. Pēdējā metode novērtē studenta iesniegtā automāta un sagaidāmā automāta akceptētās valodas un veic salīdzinājumu šīm valodām. Ja valodas ir ļoti līdzīgas, tad tiek piešķirts adekvāts vērtējums. Pēc visu vērtējumu iegūšanas, tiek aprēķināts gala vērtējums studenta iesniegtajam risinājumam.

Sistēmas kvalitatīvai darbībai bez algoritmiskajām prasībām jāapmierina arī dažas papildu prasības, kas nodrošinās sistēmas lietojamību. Pirmā no prasībām ir API, kas ļauj novērtēt automātu būves uzdevumu risinājumus. Izmantojot šīs sistēmas piedāvāto API, jābūt iespējai nākotnē izveidot tīmekļa vietni, kuras lietotāji varētu pārskatāmā veidā norādīt automātu definīcijas un veikt uzdevumu vērtēšanu. Otrā sistēmas prasība ir lai tā būtu konfigurējama. Konfigurācijas iespēja ļaus pielāgot aprēķinātos vērtējumus gadījumā, ja pēc sistēmas ieviešanas tiktu konstatēts, ka vērtēšanas sistēmā ir kādas neprecizitātes. Sistēmas izstrādē arī tiek pieņemts, ka pamatā tiks vērtēti mazi automātu risinājumi. Lielāku automātu gadījumā apstrādes laiks varētu strauji pieaugt.

Bakalaura darba pirmajā daļā sniegts teorētiskais pamats un apraksts vērtēšanas algoritmiem. Otrajā daļā aprakstīta algoritmu implementācija un tās sarežģītības novērtējums. Tika veikta arī praktiska sistēmas implementācija, ko gan, diemžēl, neizdevās līdz galam pabeigt bakalaura darba ietvaros.

1. AUTOMĀTISKA GDA NOVĒRTĒŠANA

Darba labākai uztveramībai, daļa no šīs nodaļas teksta ir balstīta uz Artūra Jāņa Pētersona 2017.gadā izstrādātā kursa darba "*Galīgu determinētu automātu būves uzdevumu automātiska vērtēšana*" rezultātiem. Teorētiskais apraksts izmantots tikai bakalaura darba praktiskās daļas labākai saprotamībai un lasāmībai.

Līdz šim nav veikti daudz pētījumi par galīgu determinētu automātu salīdzināšanu būves uzdevumu kontekstā. Automātu ekvivalences noteikšana determinētā gadījumā ir relatīvi vienkārša problēma, taču atšķirību noteikšana ne. It īpaši, ja tas tiek apskatīts kā būves uzdevumu vērtēšana – tur stājas spēkā papildu faktori, ko klasiski salīdzinot automātus nav iespējams ņemt vērā. Faktiski līdz šim ir bijis tieši viens pētījums, kurā ir aprakstīta daudz maz lietojama metode šādu gadījumu vērtēšanai [2]. Šajā nodaļā tiks aprakstīti teorētiskie algoritmi, kā implementēt GDA vērtēšanas sistēmu, balstoties uz [2] piedāvāto metodi.

Minētajā pētījumā tiek analizēts iesniegtais automāts un noteikts, cik korekts tas ir. Pašā pētījumā nav sniegts apraksts, kā iegūt pieļauto kļūdu aprakstu, izmantojot piedāvāto metodi. Balstoties uz pētījumu ir arī izstrādāts tīmekļa rīks *Automata Tutor* [3], kas gan piedāvā salīdzinoši limitētu funkcionalitāti, lai to varētu praktiski izmantot.

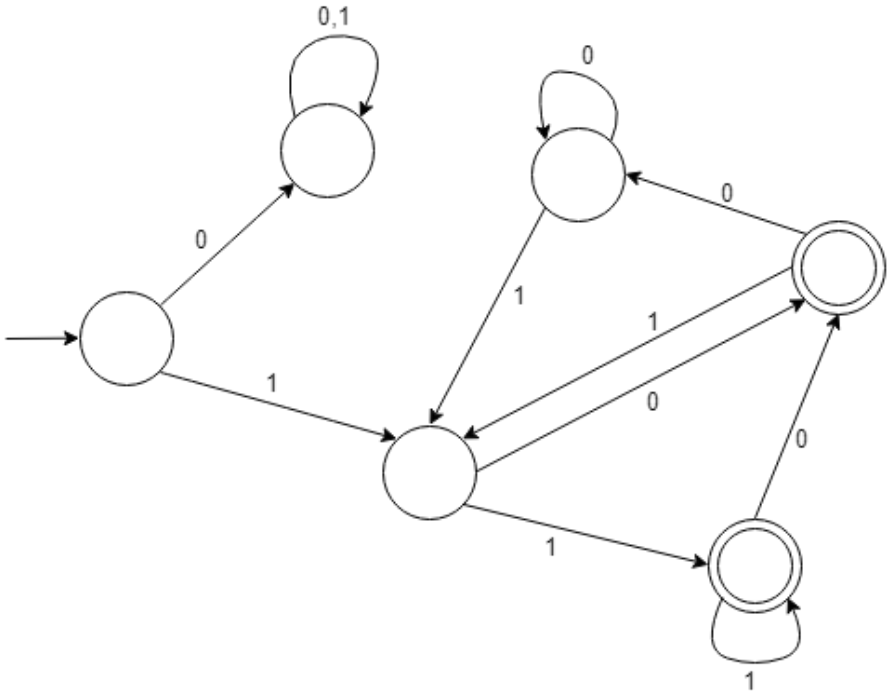
Autori pēc teorētiskās izpētes veica arī vairākus salīdzinošos izmēģinājumus, izmantojot studentu veidotus risinājumus. Rīka aprēķinātais vērtējums tika salīdzināts ar divu neatkarīgu pasniedzēju vērtējumiem. Izmēģinājumos empīriski tika pierādīts, ka rīks spēj vērtēt ar lielāku konsistenci kā cilvēki, kā arī vērtējums ir godīgāks – eksperimentā iegūtā vērtējuma atšķirību gadījumā, cilvēks piekrita rīka piešķirtajam vērtējumam pēc atkārtotas GDA izpētes.

Vērtējuma noteikšanai tiek lietots trīs metožu apvienojums. Šīs metodes ir:

- GDA pārveidošana par angļu valodai līdzīgas gramatikas izteiksmi MOSEL sintaksē;
- nelielu sintakses kļūdu meklēšana;
- dažādu vārdu piederības salīdzināšana GDA akceptētajai un sagaidāmajai valodai.

Katrai no šīm metodēm ir sava grupa ar risinājumiem, kurus tā spēj korekti identificēt un novērtēt. Ja risinājums nav korekts un neietilpst grupā, ko metode atpazīst, tad šādam risinājumam visticamāk konkrētā metode dod zemu vērtējumu. Šāds efekts novērojams, jo, pat ja metode spētu novērtēt šādu risinājumu, tas varētu prasīt daudz laika un arī būt nekorekts. Piemēram, nelielu sintakses kļūdu meklēšanu varētu paplašināt, lai šī metode meklē ceļu no iesniegtā uz korekto risinājumu. Taču tas varētu prasīt pārāk daudz laika, lai būtu lietderīgi. Citā gadījumā, izveidojot GDA aprakstošu MOSEL gramatikas izteiksmi, var iegūt rezultātu,

kas ir tālu no sagaidāmā rezultāta un metode piešķirtu zemu vērtējumu. Taču kļūda varētu būt aizmirsts akceptējošais stāvoklis, tātad vērtējumam būtu jābūt salīdzinoši augstam.

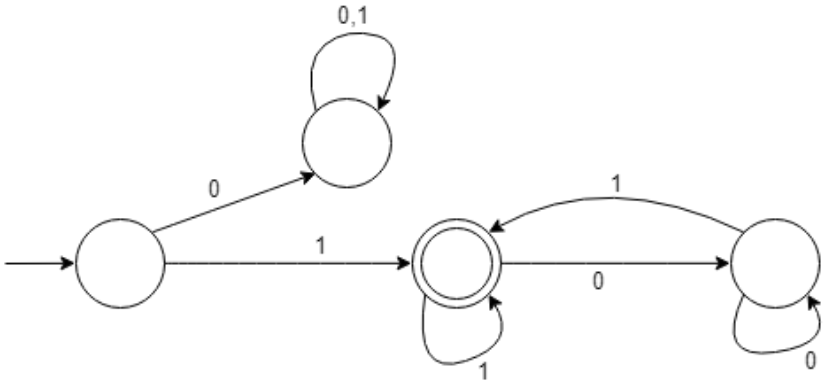


1.1. att. **Korekta GDA piemērs A_1. *L = {s | s sākas ar 1 un priekšpēdējais s simbols ir 1}***

Šeit un turpmāk par piemēru izmantota sekojoša valoda: *L = {s | s sākas ar simbolu 1, un s priekšpēdējais simbols ir 1}* (skatīt 1.1. attēlu). Automātu piemēri, kuros pieļautās kļūdas atbilst katrai no vērtēšanas metodēm, parādīti attiecīgajās nodaļās.

1.1. Uzdevuma nosacījumu nepareiza interpretācija – princips

Pietiekami regulāri studenti neizlasa uzdevuma nosacījumus līdz galam vai tos neievēro, parasti gan sasniedzot sagaidāmo rezultātu. Var gadīties arī, ka sarežģītāku uzdevuma nosacījumu gadījumā, students pavisam vienkārši kādu daļu nosacījumu būs piemirsis. Situācijas, kad students izpilda uzdevumu līdzīgiem nosacījumiem, vajadzētu vērtēt tuvu maksimālajai atzīmei, varbūt izņemot gadījumus, kad ir būtiski ievērot nosacījumus precīzi (pierādījuma uzdevumi) vai uzdevums, ko students atrisinājis, ir krietni vienkāršāks kā dotais uzdevums. Lai šādus risinājumus automātiski atrastu un novērtētu, ir nepieciešams veids, kā salīdzināt uzdevumu nosacījumus, pirms tam no studenta iesniegta risinājuma ģenerējot potenciālos aprakstus viņa izpildītajam uzdevumam tādejādi secinot, kur students pieļāvis kļūdu [2].



1.2. att. **Nekorekta GDA piemērs A_2. Automāts akceptē valodu, kur pēdējais (nevis priekšpēdējais) simbols ir '*1*'**

1.2. attēlā redzamajā piemērā automāts akceptē valodu kuras pirmais un pēdējais simbols ir '*1*'. Šī valoda pēc apraksta ir ļoti tuvu prasītajai valodai, taču automāta konstrukcija ir krietni vienkāršāka. Galu galā studentam vajadzētu prast arī izpildīt tieši uzdoto uzdevumu nevis labi izpildīt kaut kādu uzdevumu.

1.1.1. MOSEL valodas definēšana

Uzdevuma formulējuma aprakstīšanai datoriem viegli uztveramā valodā tiek izmantota speciāli konstruēta MOSEL gramatika (skatīt 1. un 2. pielikumus). MOSEL ir MSO atvasinājums, kas sākotnēji tika prezentēts [4]. Pētījuma [2] autori pielāgojuši sintaksi regulāru valodu aprakstīšanai tieši automātu būves uzdevumu kontekstā. Tā kā MOSEL ir tikai atvasinājums, tā formulas ir iespējams pārrakstīt par MSO izteiksmēm, tādejādi radot iespēju izmantot visas MSO iespējas darbojoties ar izteiksmēm. Šīs gramatikas pielāgojuma izstrādē bija divi galvenie mērķi: tai jābūt pietiekami izsmeļošai, lai ar to varētu izteikt daudzas bieži sastopamas konstrukcijas GDA būves uzdevumos; kā arī tai jābūt ļoti līdzīgai dabiskajai angļu valodai. Otrais nosacījums garantē, ka MOSEL gramatikas izteiksmes ir līdzīgas to dabiskās valodas ekvivalentiem, tāpēc arī dažādu uzdevumu formulējumu apraksti būtu pietiekami līdzīgi (vai pietiekami atšķirīgi) gan MOSEL gramatikā, gan dabiskajā valodā –, kā arī pietiekami konkrētai, lai šīs gramatikas izteiksmes nebūtu pārāk garas un cilvēkam nesaprotamas.

Ar gramatiku vien nepietiek, lai vērtētu studentu darbus, nepieciešams aprakstīt arī kā no iesniegtā GDA iegūt atbilstošo MOSEL izteiksmi. Ir acīmredzami, ka šādu pāreju nepieciešams definēt, jo iegūt MOSEL izteiksmi no GDA nav nemaz tik vienkārši. Faktiski nav zināms konkrēts determinēts algoritms, ar kura palīdzību to varētu paveikt. Nepieciešams ģenerēt īsas MOSEL izteiksmes un tās pārveidot par GDA, ko var salīdzināt ar iesniegto automātu.

1.1.2. MOSEL iegūšana no GDA

Veidojot MOSEL aprakstu no automāta, to var darīt divos soļos – vispirms iegūt MSO izteiksmi, kas apraksta GDA, no kuras pēc tam tiktu iegūta MOSEL izteiksme. Problēmas rodas abās pārejās. MSO izteiksmes iegūšana ir zināma jau diezgan sen [5], taču šis ir tikai viens veids kā iegūt derīgu MSO izteiksmi. Ir iespējamas daudz dažādas izteiksmes, kas atbilst dotam automātam. Līdz ar to, pat izveidojot MOSEL izteiksmi, nav garantijas, ka tā atbildīs automāta akceptētās valodas aprakstam, nevis automāta struktūras aprakstam [2].

Šīs problēmas dēļ [2] autori nevis no GDA konstruē valodu aprakstošu MOSEL izteiksmi, bet cenšas ar pilno pārlasi uzminēt šim automātam atbilstošo MOSEL izteiksmi. Tā kā MOSEL gramatika tika veidota pietiekami konkrēta, tad attiecīgajai izteiksmei nevajadzētu būt pārāk garai un to varētu atrast sākot pārlasīt iespējamās izteiksmes sākot ar īsāko un pamazām izteiksmi pagarinot līdz iepriekš noteiktam ierobežojumam. Formulas garums tiek aprakstīts sekojoši: tas ir maksimālā vērtība no

- mezglu skaita MOSEL izveduma kokā;
- maksimālā skaitļa, kurš parādās izteiksmēs, kas apraksta fragmenta garumu vai parādās citās skaitļu operācijās;
- simbolu virknes, kura parādās izteiksmē, garuma.

Pamazām palielinot MOSEL formulas garumu tiek meklēta izteiksme, kas atbilst iesniegtajam GDA. Lai noteiktu, ka izteiksme atbilst GDA, šai izteiksmei nepieciešams uzkonstruēt atbilstošo automātu *A'* ar augstāk aprakstīto metodi. Ja iesniegtā GDA un *A'* akceptētās valodas sakrīt, tad arī var droši apgalvot, ka atrastā izteiksme apraksta iesniegto GDA.

Izteiksmju ģenerēšanas paātrināšanai tika izmantotas arī matemātiskās loģikas optimizācijas, kā $\neg\neg E = E$; $E \wedge E = E$ u.tml. Šo loģikas optimizāciju starpā ir arī tādas optimizācijas, kas ietver konjunkciju un disjunkciju; šīs optimizācijas krietni spēja uzlabot algoritma darbības laiku.

1.1.3. Vērtējuma noteikšana

Atliek noteikt atšķirības starp dažādām MOSEL izteiksmēm, lai noteiktu, cik atšķirīgs ir studenta iesniegtais risinājums no sagaidāmā risinājuma. Tā kā MOSEL valoda ir formāla gramatika, tai ir iespējams izveidot izveduma koku (skatīt 1.3. un 1.4. attēlus) un, ja šis koks ir kreisais vai labais izveduma koks, tad tas ir arī viennozīmīgs. Kad no studenta iesniegtā GDA un sagaidītā GDA ir izveidoti attiecīgie izveduma koki, atliek tos salīdzināt, ko var paveikt aprēķinot TED – koka izmaiņu attālumu [6]. Šis lielums ir aprēķināms saskaitot nepieciešamās izmaiņas, ko jāveic izveduma kokā lai no viena koka iegūtu otru. Par izmaiņām tiek uzskatītas

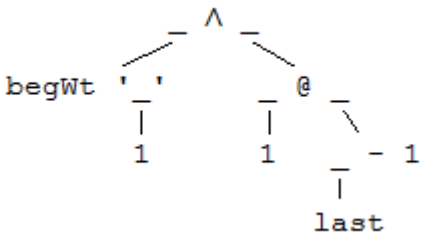
3 viedu darbības: virsotnes pārsaukšana, virsotnes dzēšana un virsotnes pievienošana. Lai šis vērtējums būtu objektīvs, TED tiek salīdzināts pret virsotņu skaitu sagaidāmā automāta MOSEL izveduma kokā. Tādejādi pie vienas nepieciešamās izmaiņas lieliem izveduma kokiem tiek piešķirts augstāks vērtējums kā vienkāršākiem izveduma kokiem. Tā kā katrai valodai pastāv dažādi MOSEL izvedumi, autori pārlasa vairākus iespējamos izveduma kokus, un beigās izvēlās augstāko piešķirto vērtējumu [2].

Automātus A_1 un A_2 aprakstošās MOSEL izteiksmes ir attiecīgi:

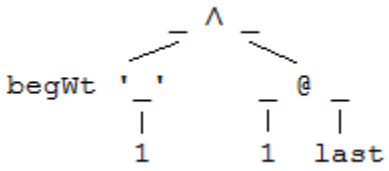
$$L_1 = begWt\ '1' \wedge '1'\ @\ (last - 1)$$

$$L_2 = begWt\ '1' \wedge '1'\ @\ last$$

Viegli pamanīt, kur abās izteiksmēs ir atšķirība – pirmajā izteiksmē simbols '*1*' ir atrodams priekšpēdējā pozīcijā, taču otrajā izteiksmē tas ir atrodams pēdējā pozīcijā. Atšķirība izveduma kokos būtu viens papildu mezgls otrajā izteiksmē:



1.3. att. **L_1 MOSEL izveduma koks**



1.4. att. **L_2 MOSEL izveduma koks**

Arī izveduma kokos ir viegli pamanāma atšķirība starp abām valodām. Izveduma pirmajā daļā (pirms loģiskā 'un') atšķirību nav. Taču pēc loģiskā 'un' ir viegli pamanāma atšķirība – parādās vēl viens mezgls, kas veic aritmētisku darbību.

Protams, dažādas sarežģītības uzdevumiem vērtējums par vienu pieļautu kļūdu būs dažāds. Tāpēc nepieciešams kaut kādā veidā salīdzināt pieļauto kļūdu skaitu attiecībā pret izteiksmes izmēru, kas sarežģītākiem uzdevumiem būs lielāks. Šī salīdzināšana tiek veikta tiešā veidā:

$$WTED(A_1, A_2) = \frac{TED(A_1, A_2)}{|T_{A_1}|}$$

Automāta A_2 MOSEL izveduma kokā pietrūkst divi mezgli, lai šis izveduma koks sakristu ar A_1 izveduma koku. Tā kā A_1 ir atskaites punkts (tas ir korekts automāts), tad gala vērtējums WTED tiks rēķināts pret A_1 mezglu skaitu. Šajā konkrētajā piemērā tiktu piešķirts vērtējums $\frac{2}{7}$.

1.1.4. Trūkumi

Pētījumā piedāvātā metode nav arī bez trūkumiem – netiek novērtēts, cik faktiski sarežģīti bija risināt uzdevumu, ko paveicis students. Ja studenta iesniegtais GDA risinājums ir

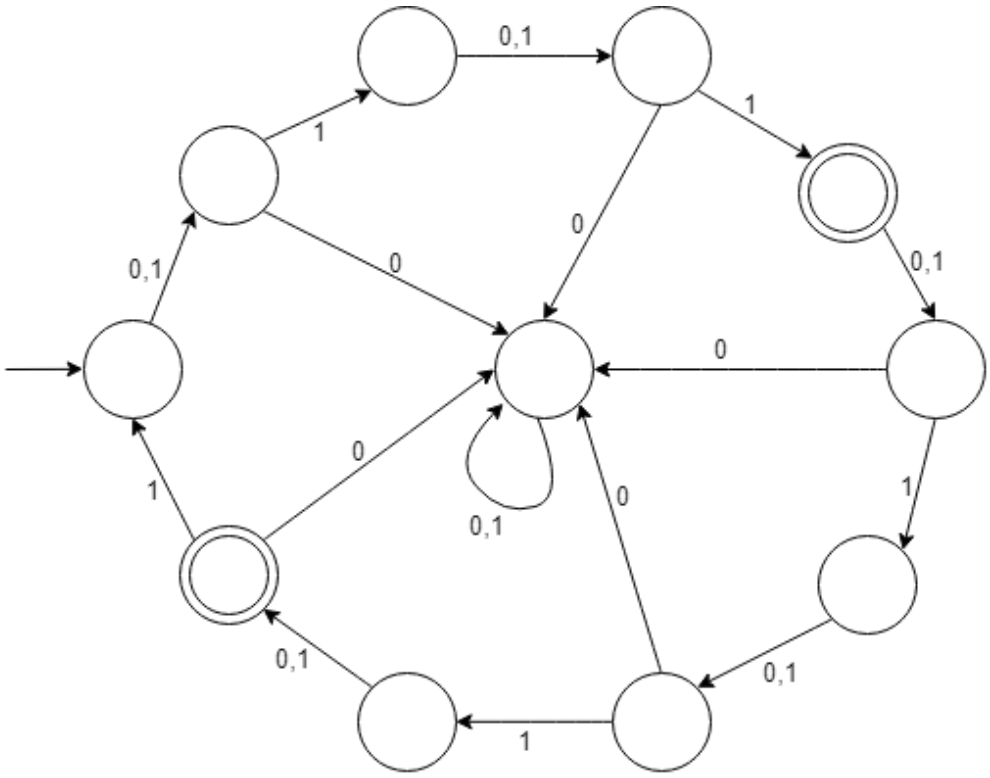
komplicētāks par sagaidīto, tad tā vēl nebūtu problēma. Taču ja komplicētāks ir sagaidītais GDA risinājums, tad var rasties problēmas. Nevar droši apgalvot, ka problēmas sarežģītība ir tieši proporcionāla MOSEL izteiksmes sarežģītībai, taču šis ir vienīgais kritērijs, pēc kura tiek pielāgots vērtējums. Biežāk pat būs tieši pretēji tam, kā tiek piedāvāts rēķināt vērtējumu – sarežģītāku valodu pārveidojot, var nejauši iegūt valodu, kurai uzkonstruēt GDA ir krietni vienkāršāk. Šādā gadījumā tiktu piešķirts augsts vērtējums, ja valodu apraksti atšķirtos ļoti nedaudz. Patiesībā risinājums nebūtu pelnījis pārāk augstu vērtējumu, jo atrisinātā problēma neatbilst prasībām un ir krietni vienkāršāk risināma.

Kā piemēru šādai situācijai, aplūkosim sekojošu uzdevumu:

$$L = \{s \mid s\ garums\ dalot\ ar\ 5\ dod\ atlikumu\ 4,$$
$$un\ visās\ pāra\ pozīcijās\ atrodas\ simbols\ '1'\}$$

Atbilstošā MOSEL izteiksme (izveduma koku skatīt 1.7. attēlā):

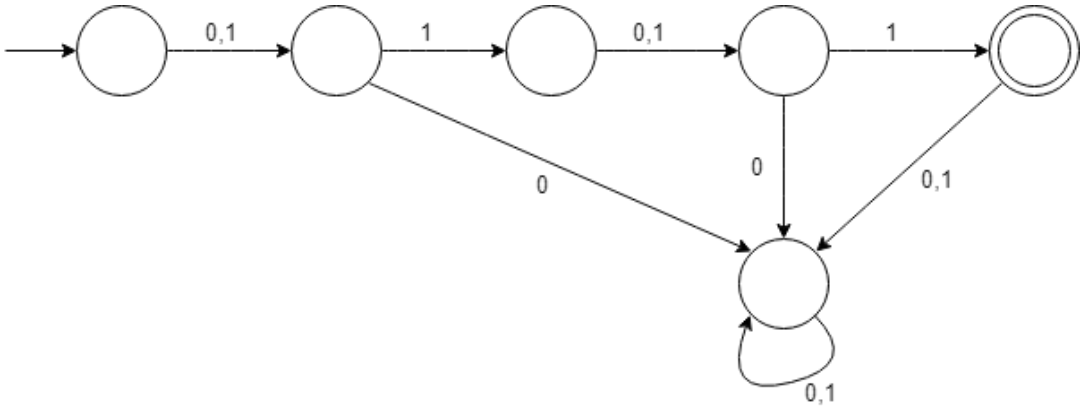
$$'1' @ \{x \mid |psLe\ x| \ \%\ 2 = 0\} \wedge |all| \ \%\ 5 = 4$$



1.5. att. **Valodu *L = {s | s garums dalot ar 5 dod atlikumu 4, un visās pāra pozīcijās atrodas simbols 1}* akceptējošs automāts**

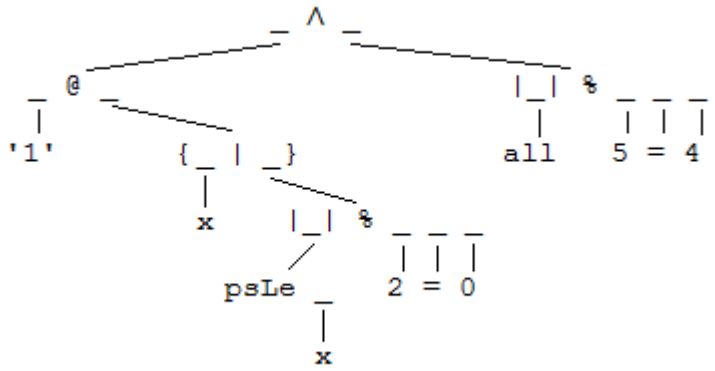
Šai valodai automāts izveidojas relatīvi liels, jo ir jāatceras gan nolasītā vārda garums pēc moduļa 5, gan tas ka katram otrajam simbolam ir jābūt '*1*'. Pieļaujot kļūdu uzdevuma nosacījumu interpretācijā, students varētu būt iesniedzis automātu, kura vienīgā atšķirība no aprakstītā automāta ir tāda, ka tas akceptē vārdus garumā 4:

$$'1' @ \{x \mid |psLe\ x| \ \%\ 2 = 0\} \wedge |all| = 4$$



1.6. att. **Valodu *L = {s | s garums ir vienāds ar 4, un visās pāra pozīcijās atrodas simbols 1}* akceptējošs automāts**

Atšķirība starp abām MOSEL izteiksmēm ir divi mezgli, taču otrajai izteiksmei atbilstošo automātu uzkonstruēt ir krietni vieglāk. Pirmās valodas izveduma kokā ir 14 mezgli, un šī koka pārveidošanai par otrās valodas izveduma koku, vienam mezglam ($|all| \% 5 = 4$) jāmaina mezgla informācija, lai šis mezgls kļūtu par ($|all| = 4$), kā arī nepieciešams arī izmest vienu mezglu. Tātad rīka piešķirtais vērtējums būtu $\frac{2}{16}$, kas pārveidojot uz atzīmi 10 baļļu skalā ir $\left(1 - \frac{2}{16}\right) * 10 = \frac{7}{8} * 10 = 8{,}75 \approx 9$. Salīdzinot konstruētā automāta sarežģītību, šāds vērtējums nav adekvāts.



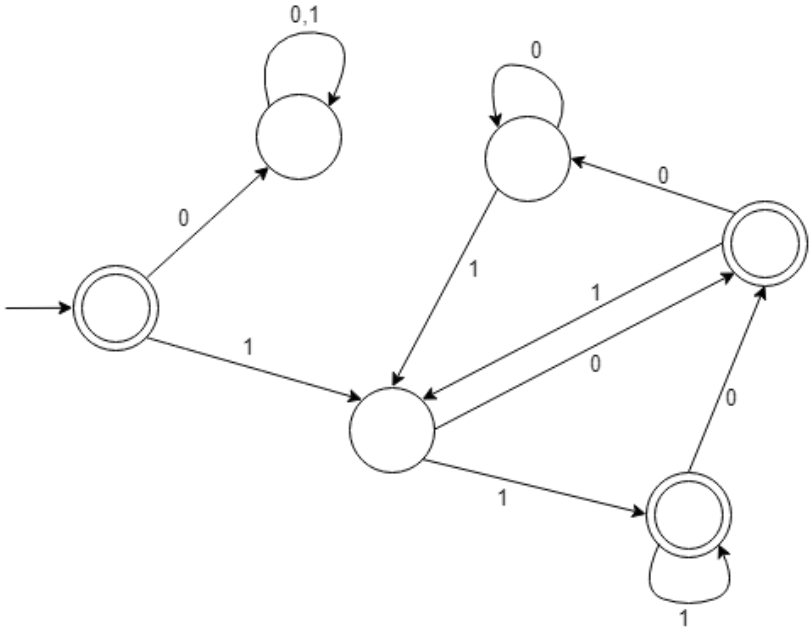
1.7. att. ***Valodas L = {s | s garums dalot ar 5 dod atlikumu 4, un visās pāra pozīcijās atrodas simbols 1}* MOSEL izveduma koks**

Jāpiemin, ka konkrētajā gadījumā kļūda dabiskās valodas interpretācijā ir diezgan ievērojama. Ir liela atšķirība, vai pieļautā kļūda ir "garums pēc moduļa *m* ir vienāds ar *n*" aizvietošana ar "garums ir vienāds ar *n*", vai arī "garums ir vienāds ar *n*" aizvietošana ar "garums ir vismaz *n*". Otrajā gadījumā cilvēciska kļūda ir daudz ticamāka.

1.2. Sintakses kļūdas – princips

Šis kļūdu paveids ietver neuzmanības kļūdas korektam risinājumam. Ja students ir pareizi atrisinājis uzdevumu, taču, pierakstot savu risinājumu, pieļāvis kādu neuzmanības

kļūdu, tad šādas kļūdas tiks atrastas šajā solī. Uzdevuma izmēram pieaugot, iespēja, ka tiks pieļauta kāda neuzmanības kļūda, palielinās. Kļūdīties ir tikai cilvēcīgi, un agrāk vai vēlāk kāda kļūda neizbēgami tiek pieļauta, it īpaši mācoties. Tā kā neuzmanības kļūdas, kas ietver sintakses kļūdas atrisinājumā, ir salīdzinoši vieglāk izlabot kā nepareizu risinājuma ideju, tad par šādām neprecizitātēm arī tiek noņemts mazāk punktu. Visticamāk, students sākotnēji bija atrisinājis uzdevumu pareizi.



1.8. att. **Nekorekta GDA piemērs A_3. Automāts akceptē arī ε, kas neatbilst valodas aprakstam**

1.8. attēlā redzamajā piemērā risinājuma autors ir izveidojis korektu automātu, kas gandrīz precīzi akceptē prasīto valodu. Izmainot vienu stāvokli (izņemot sākuma stāvokli no akceptējošo stāvokļu kopas) ir iespējams izlabot automātu tā, lai tas pilnīgi precīzi akceptētu prasīto valodu.

1.2.1. Kļūdu meklēšana

Lai atrastu sintakses kļūdas, iesniegtais GDA tiek pārveidots tā, ka tas akceptē prasīto valodu L. Pēc nepieciešamo pārveidojumu atrašanas, vērtējums tiek aprēķināts ņemot vērā veikto pārveidojumu skaitu. Studenta iesniegtais automāts var būt ar citu algoritmu nekā to iedomājies uzdevuma sastādītājs. Tāpēc netiek meklēts automāts, kas sintaktiski precīzi atbilstu kādam iepriekš zināmam korektam automātam (kas akceptē prasīto valodu), bet gan automāts, kas akceptē prasīto valodu un ir ekvivalents iepriekš zināmajam automātam [7]. Šis nosacījums arī vienkāršo iespējamos pārveidojumus – nav nepieciešams dzēst kādu stāvokli, pietiek pārvietot pārejas, kas uz to ved. Ja šīs pārejas tiktu dzēstas, tad kaut kādam stāvoklim q', no kura pāreja veda uz stāvokli q, ko gatavojamies dzēst, trūks izejošā pāreja un vēl viens

pārveidojums būs šādas pārejas ieviešana. Šos divus soļus var un ir vēlams apvienot vienā solī – pārejas izmainīšana. Visi pieejamie pārveidojumi automātam ir šādi:

- pārejas izmainīšana;
- stāvokļa ievietošana, kur uz jauno stāvokli neved neviena pāreja, un visas pārejas, kas iziet no šī stāvokļa ved uz šo pašu stāvokli;
- stāvokļa pievienošana vai izņemšana no akceptējošo stāvokļu kopas.

1.2.2. Vērtējuma noteikšana

Vērtējuma aprēķināšanai nepieciešams zināt, cik sarežģīts ir uzdevums. Jo sarežģītāks uzdevums, jo augstāks vērtējums par risinājumu ar vienu sintakses kļūdu – vienu labojumu automātā. Noteikt uzdevuma sarežģītību no tā apraksta var būt grūti, kā arī ja to darīs cilvēks, tad tas būtu subjektīvs vērtējums. Taču šī rīka mērķis ir pievienot fiksētu daudzumu subjektīvisma – veikt vērtēšanu maksimāli objektīvi. Tāpēc vērtēšanai tiek izmantots vēl viens GDA – tāds, kas akceptē prasīto valodu un nesatur lieku informāciju, ko varētu saturēt studenta iesniegtais automāts (piemēram, daudz nesasniedzamu stāvokļu). Šīs metodes vērtējums ir veikto pārveidojumu skaits dalīts ar korektā automāta stāvokļu un pāreju skaita summu. Acīmredzami, automātam, kam ir mazāks sintakses kļūdu skaits, šis skaitlis būs zemāks [2].

$$WDFA\text{-}D(A_1, A_2) = \frac{DFA\text{-}D(A_1, A_2)}{v + e}$$

Automāta A_3 gadījumā DFA-D vērtība būs 1, jo jāpārveido tikai 1 stāvoklis (jāizņem no akceptējošo stāvokļu kopas). Šis skaitlis jādala ar automāta A_1 kopējo stāvokļu un pāreju skaita summu: 6+11=17. WDFA-D šajā piemērā ir $\frac{1}{17}$.

Praksē, nepieciešamo pārveidojumu atrašanai, tiek pārbaudīti visi iespējamie pārveidojumi, līdz tiek sasniegts laika (pieļauto pārveidojumu) limits vai atrasts automāts, kas akceptē prasīto valodu [8].

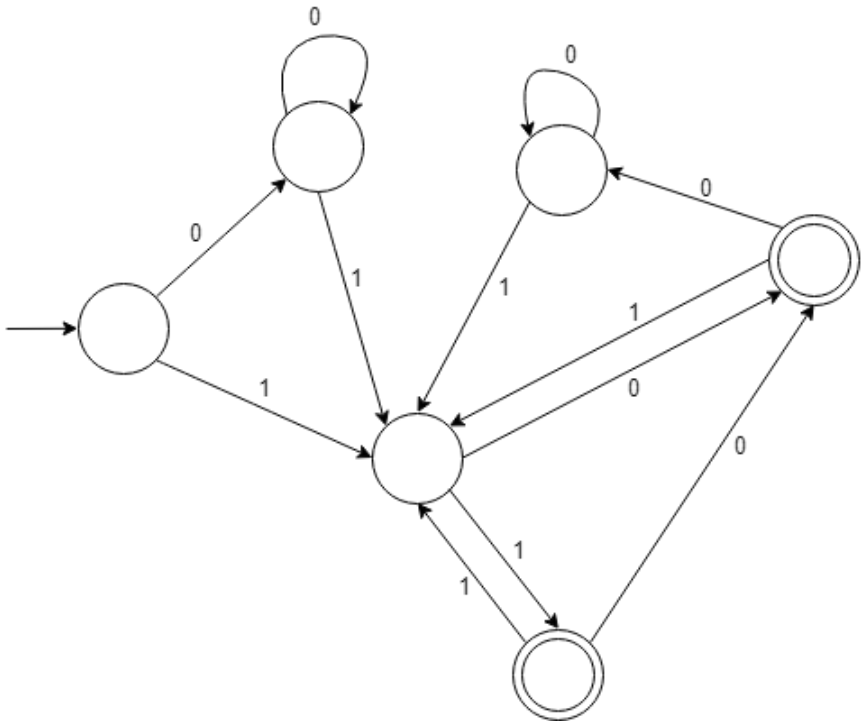
1.2.3. Metodes paplašināšana

Dažu sintakses kļūdu atrašanai metode ir izveidota pietiekami efektīva, taču praksē tā darbojas tikai uz maziem automātiem un nelielam kļūdu skaitam. Ja būtu zināms automāts, ko students bija vēlējies izveidot, tad varētu salīdzināt abus automātus un noteikt to atšķirības, taču, ja students spētu uzrādīt šo automātu, metodei nebūtu pielietojuma. Iesniegto automātu nevar minimizēt, jo tādejādi tiktu izveidots automāts, kas akceptē to pašu valodu ko sākotnējais GDA, taču sintakses kļūdas vairs nebūtu iespējams atrast. Vienīgais veids dažu sintakses kļūdu atrašanai ir pārveidojumu pārbaudīšana, kas arī tiek piedāvāts.

Šī metode liek domāt par potenciāliem pielietojumiem nepabeigtu risinājumu novērtēšanā. Pārveidojumu pielietošana un salīdzināšana dažās iterācijās ir izdarāma efektīvā laikā, taču iespējamais pārveidojumu skaits (līdz GDA tiktu papildināts līdz korektam), var būt ļoti liels, maksimāli $|Q| * |\Sigma|$, ja visas pārejas jāizveido no jauna. Šis jau uzreiz liek secināt, ka ar vienkāršām metodēm tas nav izdarāms, jo prasa eksponenciālu laiku.

1.3. Nelielas valodas atšķirības – princips

Iepriekšējās divas metodes apskatīja risinājumus, kuri ir tuvu pareizajam risinājumam, taču ir ar nelielām kļūdām. Visi pārējie risinājumi vai nu saturēs vairāk kļūdu vai arī būs nepilnīgi. Mēģināt aprakstīt visu iespējamo veidu kļūdas un formalizēt tās varētu būt pārāk grūti un gari. Tāpēc pārējo risinājumu novērtēšanai var salīdzināt iesniegto GDA akceptēto valodu ar sagaidīto akceptējamo valodu. Šīs salīdzināšanas vajadzībām valodas netiek nekādi aprakstītas, bet vienkārši tiek pārbaudīti dažādi vārdu un salīdzināts, vai viņi pieder sagaidāmajai un iesniegtā GDA akceptētajai valodai. Šī bieži būs tā metode, pēc kuras rezultāta tiek noteikts vērtējums iesniegtajam automātam, ja risinājums netiks korekti novērtēts kādā no iepriekšējiem gadījumiem [2].



1.9. att. **Nekorekta GDA piemērs A_4. Akceptētā valoda ir tuvu prasītajai valodai, taču ir pieļautas vairākas kļūdas**

1.9. attēlā parādīts automāts, kas akceptē līdzīgu valodu prasītajai. Salīdzinot ar automātu A_1, ir pieļautas divas sintakses kļūdas, tāpēc iespējams, ka šim automātam piešķirtais vērtējums nāktu no tās metodes. Taču piemēra vajadzībām tas netraucē. Ir grūti izveidot saprātīgu aprakstu šī automāta akceptētajai valodai, bet ir redzams, ka tā satur vārdus, ko A_1

neakceptē (piemēram, ‘*011*’), kā arī dažus vārdus, ko akceptē A_1, šis automāts neakceptēs (piemēram, ‘*111*’).

1.3.1. Vērtējuma noteikšana

Vērtējuma aprēķināšanas algoritms kaut kādā mērā ir ļoti līdzīgs iepriekšējo metožu algoritmiem. Tiek saskaitīts atšķirību skaits abās valodās, un dalīts ar kopējo vārdu skaitu sagaidītajā valodā. Šī darbība tiek veikta konkrētam vārdu garumam, tādejādi iegūstot proporciju, kas norāda atšķirīgo vārdu skaitu pret sagaidīto vārdu skaitu. Ideālā gadījumā tiktu aprēķināta robeža, vārda garumam tiecoties uz bezgalību, taču praksē tas būtu pārāk sarežģīti un faktiski tiek aprēķināta proporcija daļai vārdu. Aprēķins tiek veikts sākot ar vārda garumu 0 un tiek pamazām palielināts. Visbiežāk kļūdas tiks pieļautas tieši gadījumos, kad vārdi nav pārāk gari. Citos gadījumos būs iespējams novērtēt aptuveno kļūdaino vārdu proporciju.

$$A\text{-}DEN\text{-}DIF(L_1, L_2) = \frac{\sum_{n=0}^{2k} \frac{|((L_1 \setminus L_2) \cup (L_2 \setminus L_1)) \cap \Sigma^n|}{\max(|L_2 \cap \Sigma^n|, 1)}}{2k+1}$$

Jāņem vērā, ka ne visām valodām robeža tieksies uz konkrētu skaitli. Šī iemesla dēļ tiek aprēķināta vidējā kļūda: tiek apskatīti visi n no 0 līdz $2k$, kur k ir L_2 (sagaidītā valoda) akceptējoša GDA stāvokļu skaits. Katram šādam n tiek aprēķināta šo valodu pārklāšanās proporcija un no visiem rezultātiem tiek paņemts vidējais. Iegūtais skaitlis tad arī ir metodes sniegtais vērtējums studenta iesniegtajam risinājumam. Līdzīgi kā iepriekš – jo mazāks piešķirtais vērtējums, jo labāks risinājums.

1.3.2. Trūkumi

Metode acīmredzami ir paredzēta, lai noteiktu vērtējumu tiem risinājumiem, kam tas tāpat būs salīdzinoši zems. Visbiežāk nelielās kļūdas automātos ir atrastas ar kādu no iepriekšējām divām metodēm. Izmantotā vērtēšanas metode šajā gadījumā ir pamatota, taču tā ne vienmēr būs objektīva. Ja students ir izstrādājis risinājumu līdzīgai valodai, bet automāta konstrukcijas nekādā veidā neatbilst korekta automāta konstrukcijām, tad valodas atšķirību meklēšana būs efektīva metode – salīdzinoši precīzi varēs noteikt, cik tuvu studenta risinājums ir vēlamajam. Citos gadījumos studentam var tikt piešķirts nepamatoti zems vērtējums. Par piemēru ņemsim valodu, kas sastāv no divām loģiskām sastāvdaļām, kuras abas ir pietiekami sarežģītas, lai tās novērtētu neatkarīgi. Ja kāda no šīm daļām ir absolūti korekta, bet otra – absolūti nekorekta, visticamāk neviena no vērtēšanas metodēm nevarēs sniegt pienācīgu vērtējumu darbam.

Uzdotā metode labi pilda savu uzdevumu – novērtēt risinājumus, kuri vēl nav novērtēti salīdzinoši korektā veidā. Problēma šajā risināšanas metodē paliek tā, ka ir dažas risināšanas metodes un uzdevumi, kuri nekādā veidā netiek korekti novērtēti.

Lai noteiktu atšķirību starp automātu akceptētajām valodām, tiek ģenerēti vārdi līdz iepriekš fiksētam garumam. Galvenā problēma šajā varētu būt lieli alfabēti. Atsevišķos gadījumos kopējais alfabēta izmērs varētu nebūt pārāk būtisks, ja ir maz stāvokļi. Tad ļoti iespējams varētu grupēt simbolus pa grupām, kuras uzvedas līdzīgi. Piemēram, valoda L_1 = {*s | s visi simboli izņemot pēdējo ir 0, alfabētā 0,1}* un valoda L_2 = {*s | s visi simboli izņemot pēdējo ir 0, alfabētā 0,.., 9}* ir ļoti līdzīgas automāta konstruēšanas ziņā, mainās tikai pāreju nosacījumi. Akceptēto vārdu proporcija pret kopējo vārdu skaitu gan krasi izmainās – no aptuveni $\frac{1}{2^n}$ uz $\frac{1}{10^n}$. Šajā gadījumā algoritms darbosies optimāli, taču sarežģītākos gadījumos atšķirības būs lielākas.

Vēl viens būtisks trūkums ir vērtējuma piešķiršana daļēji korektiem automātiem, kam ir liela daļa vārdu, kurus šis automāts nepareizi apstrādā. Ja iesniegtais automāts akceptē daudzus vārdus pareizi, taču ir kāda neliela kļūda, kuras dēļ daudzi vārdi tiek nepareizi akceptēti vai noraidīti, tas var strauji pasliktināt vērtējumu. Piemēram, ja tiek sagaidīts, ka vārdi garumā x netiks akceptēti nemaz, taču iesniegtais automāts akceptē y šādus vārdus, tad kopējais automāta vērtējums palielināsies par $\frac{y}{2k+1}$. Pietiekami lieliem y (vai pietiekami maziem k) ar šo var pietikt, lai piešķirtais vērtējums par automātu nebūtu pozitīvs. Šī metode var būt arī nevienlīdzīga gadījumos, kad sagaidītā valoda ir liela vai tieši ļoti maza. Ja valodai pieder daudz vārdu, tad šī vērtēšanas metode lēnāk pazeminās vērtējumu kā gadījumā, ja valodai pieder maz vārdu.

1.4. Gala vērtējuma aprēķināšana

Katra no aprakstītajām 3 metodēm dod savu vērtējumu, no kura ir jāiegūst viens gala vērtējums ar kādu novērtēt iesniegto automātu. Lai gan varētu mēģināt apvienot visus vērtējumus vienā un izdot to kā gala vērtējumu, praksē šāda metode nebūtu efektīva un par rezultātu tiek pieņemts augstākais no trim vērtējumiem. Šāds lēmums pieņemts galvenokārt divu iemeslu dēļ: esošo vērtēšanas rīku varētu uzlabot, piedāvājot atsauksmju iespēju, lai koriģētu nekorektos vērtējumus; kā arī galvenokārt tikai viena no trim metodēm GDA novērtē objektīvi. Piemēram, ja ir aizmirsts akceptējošais stāvoklis, MOSEL apraksts un akceptētā valoda var ļoti mainīties, kaut gan ir kļūdaini norādīts šis viens stāvoklis. Šādam darbam nevajadzētu noņemt daudz punktu, kā to piedāvātu darīt divas no trim metodēm [2].

Pirms gala vērtējuma piešķiršanas, katras metodes iegūtais vērtējums tiek normalizēts, izmantojot iepriekš zināmus koeficientus. Šie koeficienti sākotnējā [2] pētījumā tika noskaidroti

empīriskā veidā – veicot izmēģinājumus ar dažādiem automātiem un analizējot iegūtos rezultātus. Koeficientu pielietošana pārveido metodes rezultātu atbilstošā vērtējumā. Tas nepieciešams, jo visas metodes izdod kaut kāda veida proporciju, no kuras iespējams secināt par iesniegtā GDA korektumu, nevis galīgu vērtējumu. Kad vērtējumiem veikta normalizācija, vienkārši tiek izvēlēts augstākais vērtējums.

1.4.1. Korekts automāts

Vērtēšanai nepieciešami lieli datora resursi, tāpēc to taupības nolūkos vispirms tiek pārbaudīts vai risinājums nav absolūti korekts. Tādā gadījumā kļūdas nemaz netiek meklētas un uzreiz tiek piešķirts maksimālais vērtējums. Tomēr šim solim ir arī viens negaidīts blakusefekts – tie automāti, kas šo pārbaudi neiztur, vairs *nevar* saņemt maksimālo vērtējumu. No vienas puses tas šķiet loģiski – ja automāts saņem maksimālo vērtējumu, tad tam būtu jābūt absolūti korektam. Automāts, kas saņem atzīmi 9,6 no 10, pēc atzīmes noapaļošanas iegūs maksimālo vērtējumu. Tāpēc, neatkarīgi no visiem apstākļiem, ja pēc noapaļošanas iznāk maksimālais vērtējums, tad no tā tiek noņemts viens punkts [8]. Situācijā, kad maksimālais punktu skaits ir norādīts kā 100 punkti nevis 10, iepriekš minētais automāts saņems 96 no 100 punktiem, kas ļauj daudz objektīvāk uzlūkot saņemto vērtējumu un automāta korektumu, arī nav nepieciešamības mākslīgi samazināt piešķirto vērtējumu.

Punktu noņemšana vieš šaubas par risinājuma efektivitāti vairāku iemeslu dēļ. Pirmkārt, šī loģika varētu būt apšaubāma, jo pie maza kopējo punktu skaita (piemēram, 5) par dažām mazām kļūdām arī varētu piešķirt maksimālo vērtējumu. Otrkārt, šī loģika pie lielākiem punktu skaitiem varētu norādīt uz kaut kādu algoritma nekorektu konfigurāciju – algoritmam vajadzētu atrast kādu kļūdu automātā, ja jau tas nebija pilnīgi korekts. Treškārt, šādas manipulācijas parasti tiek pievienotas pēc kāda precedenta.

```
//If rounding yields maxgrade deduct 1 point by default
if (scaledGrade == maxGrade)
    scaledGrade = maxGrade - 1;
```

1.10. att. **Koda fragments (automatatutor-backend/AutomataPDL/DFA/DFAGrading.cs rindas 196-198) [8]**

Šīs manipulācijas ar vērtējumu liek domāt, ka, veicot risinājuma testēšanu, ir novērotas problēmas pie diezgan korektiem darbiem, kas lika ieviest šo loģiku. Iespējams, ka tas ir tikai piesardzībai, taču spriežot pēc koda komentāriem diez vai tā ir. Un ja ir – kāpēc tas nav novērsts pašā algoritmā?

1.4.2. Alternatīvas rezultātu apvienošanai

Rezultāti starp dažādām metodēm nevienā brīdī netiek apvienoti, ņemot vērā izmantotās metodes, šāda rezultātu apvienošana nav objektīvi realizējama. Ir diezgan skaidri aprakstīts, ka parasti tikai viena no visām metodēm spēs dot labvēlīgu rezultātu [2]. No metožu aprakstiem arī izriet, ka konkrētās kļūdas atrašanai, lietotā metode ir diezgan optimāla kā arī var nojaust, ka šie kļūdu veidi visbiežāk nepārklāsies. Ir daži gadījumi, kuros varētu kombinēt piešķirtos vērtējumus individuāli metodēm. Piemēram, ja algoritms atrod nelielu sintakses kļūdu, taču izrādās, ka tā nav metode, kas piešķir visaugstāko vērtējumu, bet visaugstāko piešķir trešā metode, kas meklē nelielas valodas atšķirības. Šajā gadījumā pie pietiekami liela automāta varētu ņemt vidējo vērtējumu no abām metodēm. Ir skaidrs, ka pirmā metode dos rezultātu tikai tad, ja risinājums sintaktiski būs līdzīgs vēlamajam risinājumam, tātad risinājumā ir atrodama arī kāda sintakses kļūda. Taču trešās metodes – valodas atšķirību – rezultāts norāda, ka šī sintakses kļūda (kļūdas) ir maznozīmīga un tikai nedaudz ietekmē rezultātu. Tad attiecīgi par šādu darbu varētu piešķirt augstāku vērtējumu kā par darbu, kur ir līdzīgs sintakses kļūdu apjoms, bet šīs kļūdas ir smagākas. Diemžēl praksē nav tik vienkārši noteikt gadījumus, kad kādu divu vērtēšanas metožu aprēķinātais vērtējums ir uzskatāms par korektu un ka šos vērtējumus varētu kombinēt.

2. PRAKTISKA VĒRTĒŠANAS SISTĒMAS REALIZĀCIJA

Bakalaura darba ietvaros tika veikta [2] aprakstīto algoritmu implementācija. Tika izstrādāts vienkāršs tīmekļa serviss, kurš spēj vērtēt iesniegtu automāta risinājumu. Tīmekļa risinājuma galvenā funkcionalitāte fokusējās uz datu apstrādi, tāpēc netika veidota lietotāju saskarne, bet gan API serviss, kas saņem un atgriež datus JSON formātā. Pieprasījums tiek veikts sinhroni, tāpēc lietotājs saņem vērtējumu pieprasījuma atbildē. Papildus galējam vērtējumam, lietotājam tiek atgriezti arī katras atsevišķās metodes aprēķinātie rezultāti, kas gan nav paredzēti parādīšanai lietotāja interfeisā, bet gan vairāk risinājuma autoriem atkļūdošanas nolūkos. Viena no pamata prasībām šim risinājumam bija iespēja to viegli konfigurēt, tāpēc izstrādājot servisu tika domāts par konfigurējamu parametru ieviešanu.

Izstrādes gaitā izmantota versiju kontroles sistēma *GitHub*. Izstrādes gaitā pirmkods regulāri saglabāts versiju kontroles sistēmā saglabājot integritāti. Izstrādātā servisa pirmkods ir pieejams https://github.com/ajpetersons/dfa-grader.

2.1. Tehnoloģijas izvēle

Algoritmu realizācija tika veikta GoLang programmēšanas valodā. GoLang radās 2007.gadā, kad *Google* inženieri izlēma radīt jaunu programmēšanas valodu, kas būtu ērta lietošanā, kā arī ātra izpildes un kompilācijas ziņā. 2012.gadā valoda tika oficiāli izlaista lietošanā plašākai publikai. GoLang, kā jau daudzas citas programmēšanas valodas, zemā līmenī balstās uz C valodas izpildāmo kodu. Valodas priekšrocības iekļauj:

- statisku tipu definēšana, kas nodrošina augstāku pārskatāmību un samazina kļūdas iespējamību;
- atkritumu savākšana ("*garbage collection*") – lietotājam nav jāuztraucas par atmiņas piešķiršanu vai atbrīvošanu;
- vienlaicīga izpilde – valoda ir paredzēta daudzkodolu procesoriem un paredz procesu vienlaicīgu izpildi;
- atvērtais pirmkods – GoLang implementācija ir pieejama atvērtajā pirmkodā, kā arī visu publiski pieejamo bibliotēku pirmkods ir ērti pieejams [9].

Valoda ir strauji guvusi popularitāti, un daudzi ievērojami uzņēmumi to jau lieto, sākot ar *Google*, kas šo valodu izstrādā. Daži no uzņēmumiem, kas izmanto GoLang: *Adobe, BBC, Dropbox, Netflix, SpaceX, Uber* [10].

2.1.1. GoLang priekšrocības mūsdienīga servisa izstrādē

Konkrētā servisa izstrādei GoLang tika izvēlēts vairāku apsvērumu dēļ. Viens no pirmajiem iemesliem jāmin autora pieredze darbā ar šo valodu. Šis faktors ļāva ātri uzsākt

darbu, netērējot papildu laiku, lai iepazītos ar valodu un tās konstrukcijām. Vēl viens iemesls ir jau valodā iebūvētās konstrukcijas, kas atvieglo tīmekļa servisu izstrādi, kā piemēram, tīmekļa serveris, kas nav papildus jāuzstāda vai jākonfigurē [11]. Atliek tikai izveidot izpildāmo failu, kas pēc tam darbojas uz operētājsistēmas, kurai tas kompilēts.

Primārais iemesls, kāpēc izvēlēta šī valoda, tomēr ir vienlaicīga izpilde. Šī konstrukcija ir iekļauta GoLang programmēšanas valodā jau no paša sākuma ar mērķi ļaut programmētājiem izmantot daudzkodolu procesoru dotās iespējas. Kopš 2004.gada Mūra likums turpinās nevis palielinot takts frekvenci vienā kodolā, bet gan palielinot kodolu skaitu procesorā [12]. Taču, tas arī nozīmē, ka viens process vairs nespēs izmantot visu procesora jaudu. Attiecīgi arī programmētājiem nākas pielāgoties un veidot risinājumus, kas spēj izmantot daudzkodolu procesora piedāvātās iespējas.

Lielākā daļa populāro programmēšanas valodu ir radušās pirms jau vairāk kā 20 gadiem. Laikā, kad šīs valodas radās, daudzkodolu skaitļošana vēl nebija attīstījusies, tāpēc programmēšanas valodas tika veidotas sinhronas – visas darbības notiek secīgi. Taču mūsdienās, šī paradigma ir mainījusies, un daudzkodolu skaitļošana ir iespējama, tāpēc programmēšanas valodām ir jāpielāgojas. Vairākās programmēšanas valodās tas vai nu nav iespējams, vai nav izdarāms vienkāršā veidā. Taču valoda GoLang jau no paša sākuma tika veidota ar daudzkodolu skaitļošanu padomā [9]. Šāda arhitektūra ļauj programmētājam nedomāt par vairāku kodolu izmantošanu, bet koncentrēties uz programmas pirmkoda izstrādi. GoLang vienlaicīga izpilde tiek sākta ar vienkāršu komandu (2.1. attēlā redzamajā piemērā funkcijas *myFunc2* un *myFunc3* izpildīsies vienlaicīgi), kā arī ir pieejama iebūvēta bibliotēka, kas palīdz dažādiem apakšprocesiem sazināties un saglabāt datu integritāti [13].

```
myFunc1()
go myFunc2()
myFunc3()
```

2.1. att. **Koda fragments – GoLang vienlaicīgo procesu sākšana**

Vēl viens no valodas pozitīvajiem aspektiem, kas gan nav īpaši būtisks tieši šī servisa izstrādē, ir iebūvēta funkcionalitāte tīmekļa servisu izstrādei [9]. Parasti, kad ir izstrādāts tīmekļa serviss, tas tiek darīts ar kaut kādu trešās puses bibliotēku palīdzību un izmantojot atsevišķu servera programmatūru. GoLang gadījumā ir iespējams iztikt bez trešās puses bibliotēkām un atsevišķiem tīmekļa serveriem. Šīs programmēšanas valodas standarta bibliotēkā jau ir iebūvēta funkcionalitāte, kas ļauj apstrādāt ienākošus HTTP pieprasījumus, kā arī ir iekļauta vienkārša šablonu apstrādes bibliotēka, kas ļauj atgriezt lietotājam pilnvērtīgu tīmekļa lapu [14]. Lielākā priekšrocība, izmantojot GoLang tīmekļa servisa implementācijai, ir jau iebūvētais tīmekļa serveris [11]. Parasti būtu vajadzīgs konfigurēt papildu tīmekļa serveri kā *Apache* vai *Nginx*, taču GoLang spēj klausīties konkrētos portos ienākošus savienojumus.

Rezultātā ir izveidojama komandrindas lietotne, ko atliek tikai palaist un tīmekļa serveris uzreiz darbojas.

Profesionāla programmētāja darbs nav iedomājams bez izmantoto bibliotēku pārzināšanas. Ļoti bieži nepieciešams saprast kāpēc konkrētais koda fragments strādā tieši tā nevis savādāk un dažreiz nākas veikt vai vismaz ierosināt kādas izmaiņas bibliotēkas kodā. Bibliotēku pārvaldība, kā tā ir ieviesta GoLang programmēšanas valodā, atvieglo programmētāja darbu tieši meklējot un pētot bibliotēkas pirmkodu vai dokumentāciju. GoLang bibliotēku struktūra nosaka, ka trešās puses bibliotēkām ceļš uz bibliotēku atbilst *GitHub* repozitorijai, kur atrodas bibliotēkas pirmkods. Tas nozīmē, ka nav nepieciešams uzturēt kādu atsevišķu bibliotēku pārvaldības repozitoriju, kā arī programmētājiem ir viegli piekļūt pirmkodam sev pazīstamā veidā. Dokumentācijai tiek uzturēts atsevišķs serviss - *GoDoc*. Šajā servisā ir pieejama dokumentācija visām bibliotēkām, kas atrodas kādā no atbalstītajām versiju kontroles sistēmām [15]. Šāds serviss nodrošina programmētājam pārskatāmu dokumentāciju ar visu funkciju aprakstiem.

2.2. Algoritmu realizācija

Pirms algoritmu realizācijas nepieciešams implementēt dažāda veida palīgfunkcionalitāti, kas strādā tieši ar automātiem. Tā kā risinājums paredzēts tieši determinētu automātu vērtēšanai, tad tika pieņemts lēmums automātu implementāciju kodā veikt tikai determinētiem automātiem. Pēc tam visas funkcijas, kas darbojas tieši ar automāta vērtēšanu, kā parametrus saņem divus automātus aprakstošus objektus.

Automātu objekta pamata struktūra tika iegūta no atvērtā pirmkoda bibliotēkas [16], kas ļauj aprakstīt un darbināt GDA. Šajā bibliotēkā automāts implementēts izmantojot iebūvēto vārdnīcas struktūru. Glabājot visus stāvokļus vai alfabēta simbolus šādā struktūrā nevis masīvā ir dažas priekšrocības. Netiek pazaudēta nekāda papildu loģika, ko būtu iespējams veikt izmantojot masīvus, piemēram, elementu skaita noteikšana. Vienīgā problēma ar šo risinājumu, kas varētu rasties ir, ka vārdnīcai nepieciešama atslēga un saturs. Par atslēgu jau tiek lietots konkrētais elements, taču jānorāda arī saturs. Šajā implementācijā par saturu tiek izmantota patiesuma vērtība, kam ir vairākas priekšrocības. Elements, kas atrodams vārdnīcā satur pozitīvu patiesuma vērtību. Attiecīgi visi elementi, kas nav atrodami vārdnīcā saturēs negatīvo vērtību. GoLang piedāvā konstrukciju, kas ļauj pārbaudīt atslēgas esamību vārdnīcā [17] (skatīt

```
if val, ok := myMap["key"]; ok {
    // work with val
}
```

2.2. att. **Koda fragments – GoLang vārdnīcas elementa esamības pārbaude**

2.2. attēlu), kas ir ļoti ērts veids, kā ātri noskaidrot vai elements atrodams vārdnīcā. Implementējot šādu struktūru ar masīvu palīdzību, nāktos pārlasīt visus masīva elementus. Taču, šajā gadījumā noskaidrot elementa piederību masīvam var vēl vienkāršākā veidā – uzreiz lasot elementu no vārdnīcas. Gadījumā, ja elements nebūs atrodams, tad tiks atgriezta *nulles* vērtība [17], šajā implementācijā nepatiesā patiesuma vērtība, ko uzreiz var lietot dažādās izteiksmēs.

Tā kā MOSEL struktūra pati par sevi jau atgādina koku, tad implementācijai paredzēts izmantot kokam līdzīgu datu struktūru. Veiksmīgai implementācijai nepieciešams nodefinēt dažas funkcijas – konvertāciju uz tekstu un no teksta, kā arī konvertāciju uz MSO izteiksmi –, kas arī noteiks datu struktūras laukus un atribūtus. Konvertācijas uz un no teksta tiek veiktas izmantojot rekursīvas funkcijas. No MOSEL veidojot tekstu, vispirms notiek izsaukums uz bērnu mezglu teksta veidošanas funkcijām, un pēc tam pievienotas nepieciešamās daļas, kas būtiskas tikai šai funkcijai. Ja bērnu mezgli nav iespējami (piemēram, teksta konstantei), tad uzreiz tiek atgriezts teksta saturs. Konvertācija no teksta notiek līdzīgi. Vispirms tiek atpazīts kaut kāds atomārs teksta fragments un pēc tam izmantojot saturu rekursīvi izveidots konkrētais elements. Pārveidošana par MSO izteiksmi notiek pārveidojot MOSEL termus par konkrētiem MSO termiem. Tā kā MSO arī tiek veidots ar koka struktūru, tad iespējams pielietot rekursīvas funkcijas līdzīgi kā ar MOSEL. No šiem funkciju aprakstiem arī izriet MOSEL un MSO datu tipu struktūra – ir konkrēti zināmi termi un to apstrādes funkcijas, šiem termiem ir fiksēts tekstuālais attēlojums un bērnu mezgli, ja tādi ir iespējami.

2.2.1. Uzdevuma nosacījumu nepareiza interpretācija

Šī vērtēšanas metode praksē ir vissarežģītākā no visām trim metodēm, jo iekļauj dabiskās valodas apstrādi. Precīza uzdevuma nosacījumu interpretācija prasa jau iepriekš zināmu precīzu sagaidītās valodas aprakstu. Ņemot vērā, ka vienam GDA var atbilst vairāki MSO apraksti, tad no GDA ģenerējot MOSEL aprakstu, var iegūt aprakstus, kas neatbilst uzdotajam uzdevumam. Šī iemesla dēļ, darba autors ir izvēlējies pieprasīt jau iepriekš zināmu konkrētā uzdevuma MOSEL aprakstu, lai izvairītos no potenciālām problēmām, kas rodas ģenerētu aprakstu dēļ. Strādājot pie pārējām vērtēšanas metodēm radās dažādi neparedzēti sarežģījumi, to skaitā algoritmu optimizācijas problēmas un nepieciešamība precīzāk izpētīt teorētiskos pamatus vairākām darbībām. Šo sarežģījumu dēļ neizdevās praksē implementēt vērtēšanas metodi, kas aprakstīta šajā nodaļā.

Dotā MOSEL funkcija vispirms tiek validēta, lai algoritms pēc tam ar viņu varētu darboties. Zinot korektās izteiksmes izmēru, varam noteikt ierobežojumus meklēšanas algoritmam. Nav jēgas meklēt risinājumus, kuros MOSEL apraksts ir ļoti tālu no sagaidītā

apraksta, jo tajā gadījumā iegūtais vērtējums būs ļoti zems. Tā vietā jau no paša sākuma tiek meklēti risinājumi, kuriem MOSEL izteiksmes izmērs ir maksimāli tuvu meklētā GDA aprakstošas izteiksmes izmēram. Pēc tam ar jau zināmiem algoritmiem [6] var viegli aprēķināt attālumu starp iegūtajām izteiksmēm. Tā kā katrs izteiksmes elements sevī ietver 0 vai vairāk citus MOSEL elementus, šo izteiksmi var uzskatīt par koku, kur attiecīgi elements ir lapa, vai kāds iekšējs mezgls. Tādā gadījumā var pielietot *Tree-Edit-Distance* algoritmus, lai noteiktu atšķirību starp izteiksmēm, kas izteiktas kā koki. Teorētiski šis meklēšanas algoritms darbojas laikā $O(|T_1| * |T_2|)$, kas ir ļoti pieņemama sarežģītība [6]. Faktiski kodā plānots izmantot jau esošu implementāciju šim pašam algoritmam [18].

Lai vispār varētu salīdzināt MOSEL izteiksmes, tās vispirms nepieciešams ģenerēt. Teorētiski būtu iespējams pārveidot studenta iesniegto automātu par to aprakstošu izteiksmi, taču ir iespējamas daudz MSO izteiksmes ar vienādu nozīmi [5]. Šī iemesla dēļ, nav vērts konstruēt izteiksmi no dota automāta apraksta. Praksē vērtīgāk ir konstruēt dažādas, pareizajai izteiksmei līdzīgas izteiksmes un pārbaudīt, vai kāda no šīm izteiksmēm apraksta studenta iesniegto automātu.

MOSEL izteiksmju atšķirība tiek vērtēta pēc 3 kritērijiem, palielināt (vai samazināt) izteiksmes sarežģītību var attiecīgi izmainot kādu no šiem kritērijiem. Apskatīsim dažādās iespējamās izmaiņas MOSEL izteiksmes izveduma kokam T:

- skaitļa konstantes maiņa $O(2|T|)$ – izteiksmē var parādīties dažādas skaitliskas konstantes, katru no tām var izmainīt par ±1, skaits nepārsniedz koka mezglu skaitu;
- mezgla ievietošana, izņemšana vai pārsaukšana $O(c|T| + 2|T|)$ – var gadīties, ka students kādu nosacījumu ir pārpratis un jāmaina koka struktūra. Atkarīgais lielums ir koka pašreizējais izmērs, iespējamo MOSEL izteiksmju skaits ir fiksēts – c;
- teksta konstantes maiņa $O(|T| * |\Sigma|^k)$, k ir konstantes garuma ierobežojums – meklējot tekstu, vairākās vietās parādās teksta konstantes, kas var būt kļūdaini pārrakstītas. Tiek pārbaudītas līdzīgas konstantes.

Kopējā sarežģītība veidojas $O\left((4 + c)|T| + |T| * |\Sigma|^k\right) = O(|T| * |\Sigma|^k)$. Lai meklēšana neaizņemtu pārāk daudz laika, tiek ierobežots maksimālais izmaiņu skaits, kas attiecīgi arī ierobežo teksta konstanšu garumu. Pie vairāk izmaiņām jau paliek neobjektīvi uzskatīt atrasto kļūdu skaitu par ticamu. Praksē bieži ir novērojamas mazas izteiksmes un alfabētu izmēri, kas šo meklēšanu padara pietiekami optimālu.

Kad ir izveidota jauna MOSEL izteiksme, pirmā lieta ko nepieciešams izdarīt ir pārbaudīt, vai tā apraksta to pašu valodu, ko iesniegtais GDA. Šī procesa realizācijai no MOSEL apraksta jāizveido automāts, kas akceptē to pašu valodu. Vispirms notiek konvertācija no MOSEL uz MSO, lai pēc tam varētu izmantot jau zināmas konvertācijas no MSO uz GDA [5]. Konvertācijas uz MSO implementācijas pamatā ir rekursīva funkcija, kas meklē termus dotajā izteiksmē un pārveido tos par attiecīgajām MSO izteiksmēm. Katram MOSEL termam ir konkrēti definēta pāreja uz MSO, bez cikliem. Tāpēc šī pāreja ir realizējama lineārā laikā.

Darbā tika izvēlēts variants implementēt jau iepriekš zināmu un lietotu metodi [2], taču tika izskatīti un analizēti arī citi varianti, kā varētu formāli aprakstīt studentiem uzdoto problēmu tā, lai pēc tam no šī apraksta varētu uzkonstruēt regulāru valodu un automātu, kas akceptē šādu valodu. Šai sintaksei būtu jāapmierina divas prasības – jau minētā īpašība, ka no apraksta iespējams uzkonstruēt regulāru valodu, kā arī meklētajai sintaksei jābūt pietiekami tuvai dabiskai valodai, lai tajā varētu izsmeļoši, precīzi un nepārprotami aprakstīt uzdevumus, ko studenti risina.

Pirmais analogs MOSEL valodai, kas tika aplūkots bija pati MSO sintakse. Tā kā MOSEL ir tikai MSO paplašinājums, tad pastāvēja pamatots iemesls, lai izsvērtu šādu sintakses izvēli. Par MSO ir zināms, ka tā apraksta regulāras valodas [19], tāpēc nebūtu problēmu uzkonstruēt atbilstošu automātu, taču pieraksts šajā sintaksē ir diezgan sarežģīts un tas neatspoguļo dabiskā valodā uzdotu problēmu. MOSEL sintakse gūst ievērojamu priekšrocību pār tīru MSO ieviešot konkrētus un izsmeļošus termus. Piemēram:

$$begWt\,'00' \rightarrow \exists x \exists y \left(Q_0(x) \wedge Q_0(y) \wedge x < y \wedge \neg \exists z \left(z < x \vee (z > x \wedge z < y) \right) \right)$$

$$begWt\,'0' \rightarrow \exists x \left(Q_0(x) \wedge \neg \exists z (z < x) \right)$$

Vēl viens pamatots kandidāts, ar ko varētu aizstāt MOSEL ir regulārās izteiksmes. Ar regulārajām izteiksmēm var aprakstīt jebkādu regulāro valodu, kā arī eksistē algoritmi kas no GDA spēj konstruēt regulāro izteiksmi un pretēji [20, pp. 92-107]. Regulārās izteiksmes ir arī ar salīdzinoši (pret MSO) vienkāršām komponentēm, kas dod priekšrocību izteiksmes lasāmībā. Līdzīgi kā iepriekš, problēmas ir ar dabiskās valodas aprakstīšanu. Tiešā veidā izmantojot regulārās izteiksmes, lai aprakstītu uzdevumus, tiek iegūtas izteiksmes, kurās viena izmaiņa ne obligāti atspoguļos korektu izmaiņu dotā uzdevuma definīcijā. Piemēram:

$$begWt\,'01' \wedge |indOf\,'01'| = 2 \rightarrow 011^*00^*11^*0^*$$

$$begWt\,'1' \wedge |indOf\,'01'| = 2 \rightarrow 11^*00^*11^*00^*11^*0^*$$

Ņemot piemēru no MOSEL valodas, varētu papildināt regulārās izteiksmes tā, ka ar šīm izteiksmēm var sekmīgi aprakstīt dabisko valodu un viņas joprojām korekti translētos uz

regulāro izteiksmi, un attiecīgi uz regulāru valodu. Translācija uz regulāro izteiksmi spētu nodrošināt izteiksmes regularitāti un konvertējamību par automātu. Ietvars, kas translētu dabisko valodu uz regulāro izteiksmi, varētu tikt veidots līdzīgi kā MOSEL valoda, lai ar uzdotās sintakses palīdzību varētu konkrēti aprakstīt uzdevumus, kā arī ērti noteikt šo uzdevumu nepareizu interpretāciju. Par sintakses pamatu varētu ņemt pašu MOSEL valodu, jo tā jau ir pierādījusi sevi kā labu variantu dabiskās valodas izteiksmju aprakstīšanai. Atliek nodefinēt pāreju no MOSEL uz regulārajām izteiksmēm, kas gan šī bakalaura darba ietvaros netika darīts.

2.2.2. Sintakses kļūdas

Praktiskā algoritma realizācijā tiek veikti iepriekš aprakstītie pārveidojumi. Papildus tiek mēģināti arī citi sākuma stāvokļi, pieļaujot šāda veida sintakses kļūdas. Algoritms ir realizēts rekursīvi, vispirms pārbaudot, vai dotais GDA jau neatbilst meklētajam automātam. Gadījumā, kad automāti nesakrīt, tiek izmēģināti dažādi potenciālie pārveidojumi, rekursīvi izsaucot funkciju un palielinot rekursijas dziļumu, kas arī sakrīt ar veikto pārveidojumu skaitu. Automātu sakritības gadījumā, tiek izmantoti GoLang kanāli, lai nodotu informāciju par atrasto risinājumu.

Teorētiski aprēķinot algoritmisko sarežģītību tiek iegūta ļoti liela sarežģītība – $O(n^m)$. Katrā algoritma iterācijā tiek veiktas darbības, kas atbilst vienai izmaiņai automātā:

- stāvokļa pievienošana $O(1)$ – nav objektīvi mēģināti pievienot vairākus stāvokļus reizē un uzskatīt to par vienu pārveidojumu;
- sākuma stāvokļa maiņa $O(|Q|)$ – lai gan var izvēlēties tikai vienu jaunu sākuma stāvokli, ir $|Q|$ varianti, kas visi jāpārbauda;
- kāda stāvokļa pievienošana vai izņemšana no akceptējošo stāvokļu kopas $O(|Q|)$ – līdzīgi kā iepriekš, viena pārveidojuma ietvaros tiek mainīts tikai viens stāvoklis, taču jāpārbauda ir visi stāvokļi;
- pārejas pārvirzīšana $O(|Q|^2|\Sigma|)$ – no katra stāvokļa katram simbolam pāreju mēģina pārvirzīt uz katru citu stāvokli.

Summējot visas darbības tiek iegūta izteiksme $O(|Q|^2|\Sigma| + 2|Q| + 1)$. Tik daudz darbības tiek veiktas, lai pārbaudītu vienu pārveidojumu. Minētajās sarežģītībās ir aprēķināts atkārtojumu skaits, faktiski, lai implementētu šos pārveidojumus ir nepieciešams lielāks darbību skaits ar konstantu kārtu C. Analogi varam aprēķināt maksimālo rekursīvo izsaukumu dziļumu:

- stāvokļa pievienošana $O(|Q|)$ – tiek izveidoti $|Q|$ stāvokļi, kas atbilst stāvokļu skaitam korektā, minimizētā automātā;

- sākuma stāvokļa uzstādīšana $O(1)$ – praksē nevar būt gadījumi, kad ir vairāki sākuma stāvokļi. Algoritms kādā no iterācijām pārbaudīs korekto sākuma stāvokli lieki neizmēģinot citus stāvokļus;
- akceptējošo stāvokļu uzstādīšana $O(|F|)$, maksimāli $O(|Q|)$ – akceptējoši stāvokļi jāuzstāda atbilstoši korektam, minimizētam GDA;
- pāreju izveidošana $O(|Q| * |\Sigma|)$ – no katra stāvokļa ir izejoša pāreja katram alfabēta simbolam.

Summējot, iegūstam $O(|Q| * |\Sigma| + 2|Q| + 1)$. Faktiski šīs izteiksmes nozīme ir nepieciešamais darbību skaits, lai uzkonstruētu pilnīgi jaunu automātu, kas atbilst korektam, minimizētam automātam. Uzreiz varam veikt pieņēmumu, ka tik daudz darbības veikt nevajadzēs, jo kaut kāda daļa no automāta būs korekta.

Faktiski šī problēma pieder klasei NP. Šī piederība gan nenozīmē, ka neeksistē polinomiāla laika algoritms, kas spētu atrast nepieciešamos pārveidojumus, lai no studenta iesniegta automāta uzkonstruētu korektu automātu, kas risina doto problēmu. Iespējams, pārveidojumu atrašana ir pat NP-pilna problēma, taču šī darba ietvaros netika meklēts pierādījums, kāpēc šī būtu NP-pilna problēma. Praktiski algoritmiskā sarežģītība tiek novērsta ar vairāku paņēmienu palīdzību.

Pirmkārt, tiek samazināts koeficients pie pakāpes. Realizējot algoritmu ar pilnās pārlases metodi (augstāk aprakstītais aprēķins atbilst tieši pilnās pārlases metodei), tas faktiski ir krietni uzlabojams. Piemēram, stāvokļa pievienošana vai izņemšana no akceptējošo stāvokļu kopas vienam stāvoklim var tik veikta divas reizes pēc kārtas, uzskatot šīs darbības par diviem pārveidojumiem, kaut gan faktiski nav ticis veikts neviens pārveidojums. Analogi divu pāreju izmainīšana var tikt veikta dažādā secībā. Acīmredzami, ir iespējami dažādi šādi pārveidojumi, kas praksē nepavisam nav nepieciešami. Tādēļ algoritms tika pielāgots, lai pārveidojumi netiktu izmēģināti atkārtoti. Praktiski tas tiek realizēts ierobežojot pieļaujamos pārveidojumus divos veidos. Viena veida pārveidojumi tiek veikti leksikogrāfiskā secībā. Piemēram, veicot vairākas secīgas stāvokļa pievienošanas vai izņemšanas no akceptējošo stāvokļu kopas, šie pārveidojumi noteikti tiks veikti leksikogrāfiski augošā secībā. Tā rezultātā viens un tas pats pārveidojums netiks veikts un pārbaudīts divas reizes, samazinot rekursijas izsaukumu skaitu. Papildus leksikogrāfiskajai secībai viena pārveidojuma ietvaros, tiek veikta ierobežošana uz pārveidojumu savstarpējo secību, kaut kādā mērā ieviešot leksikogrāfisku secību dažādu pārveidojumu secībā. Piemēram, ja ir ticis veikts kādas pārejas pārveidojums, turpmāk netiks izmēģināta sākuma stāvokļa maiņa. Šādu pārveidojumu ieviešana īpaši neuzlabo algoritmisko sarežģītību, taču samazina konstanti, pie aprēķina, kas maziem skaitļiem ir pietiekami pamanāms rezultāts.

Papildus algoritmiskām optimizācijām, varam ierobežot meklēšanas dziļumu. Meklēšanas mērķis nav atrast nepieciešamos pārveidojumus vispārīgajā gadījumā, bet pārbaudīt gadījumus, kad students ir pieļāvis dažas sintakses kļūdas. Dažu kļūdu meklēšanu varam veikt ierobežojot meklēšanas dziļumu līdz konstantei, vai vērtībai, kas atkarīga no stāvokļu skaita. Pirmajā gadījumā algoritmiskā sarežģītība faktiski pat tiek samazināta līdz polinomiālai sarežģītībai $O(n^c)$, kur c ir konstante un n ir iespējamo pārveidojumu skaits, atkarīgs no stāvokļu skaita un ieejas alfabēta izmēra.

No minētajiem pārveidojumiem vienu varētu veikt jau iepriekš – jauna stāvokļa pievienošana [8]. Šai pieejai efekts būtu tādā gadījumā, kad tiek meklēts absolūti pareizais risinājums. Tad ir izdevīgi iepriekš pievienot stāvokļus, lai lieki netiktu veidoti jauni stāvokļi. Taču ierobežojot maksimālo pārveidojumu skaitu uz jau iesāktiem risinājumiem, šāda pieeja nav izdevīga. Iegūtais uzlabojums vienas iterācijas darbību skaitā ir mazāks, nekā papildu sarežģītība, ko veido uzreiz palielināts stāvokļu skaits. Arī empīriski izmēģinot abas pieejas, praksē efektīvāka izrādījās metode pievienot jaunus stāvokļus rekursīvā algoritma ietvaros, nevis pirms tā.

2.2.3. Nelielas valodas atšķirības

Algoritms realizēts divās daļās. Vispirms tiek uzģenerēti visi vārdi garumā līdz N, ko akceptē katrs automāts. Skaitlis N ir konfigurējama vērtība, taču vispārīgajā gadījumā tas ir atkarīgs no korektā automāta stāvokļu skaita. Lai izvairītos no neprecizitātēm, ir ieviesta arī N minimālā vērtība, kas tiek izmantota maziem automātiem. Skaitlis N tiek veidots atkarībā no ieejas alfabēta izmēra, jo tas ir galvenais faktors pēc kura aug valodas izmērs. Pieaugot alfabēta izmēram, tiek izmantots attiecīgi mazāks pārbaudāmo vārdu maksimālais garums N. Pēc akceptēto vārdu ģenerēšanas, tiek izpildīts algoritms, kas aprakstīts iepriekš. Tiek aprēķināta proporcija – cik ir vārdu garumā n, kas pieder tieši vienai no valodām L_1 un L_2, pret vārdu skaitu garumā n, kas pieder sagaidāmajai valodai.

Algoritma sarežģītība teorētiski ir diezgan augsta. Lai aprēķinātu visus akceptētos vārdus, kuru garums nepārsniedz N, tiek izpildīts cikls, konstruējot masīvu ar stāvokļiem, kādos GDA var atrasties pēc konkrēta soļu skaita. Attiecīgi pēc iterācijas, var arī noteikt, kurus vārdus automāts akceptē. Tā kā nepieciešams uzturēt informāciju ne tikai par stāvokļiem, kas akceptē, bet gan visiem stāvokļiem, tad faktiski katrā iterācijā tiek veiktas $|\Sigma|^n$ darbības. Kopā tātad tiek veiktas $\lim_{n \to N} |\Sigma|^n = |\Sigma|^{N+1}$ darbības. Analogs darbību skaits tiek veikts arī koeficienta aprēķināšanai. Tātad varam secināt, ka kopējā algoritma sarežģītība ir $O(m^n)$, kas ir ļoti augsta sarežģītība.

Sarežģītības ierobežošanai tiek samazināts meklējamo vārdu garums, to samazinot, līdz tas sasniedz 4. praksē tas ļoti neietekmē atrastā rezultāta precizitāti, taču var samazināt sarežģītības kārtu vairākas reizes. Pie maziem alfabētiem tiek izmantots liels koeficients N, pie lielākiem alfabētiem samazinātais vārda garums tiek kompensēts ar paša alfabēta izmēra pieaugumu. Lielākajā daļā gadījumu valodas izmērs uzdevumos studentiem nebūs pārāk liels, kā arī automātu izmērs nebūs ļoti liels, kas nozīmē augstu koeficientu N.

Faktiski arī šī vērtēšanas metode nav paredzēta absolūtai gadījumu pārlasei – metodes mērķis ir konstatēt gadījumus, kad students ir pieļāvis dažas kļūdas attiecīgi nekorekti akceptējot (vai neakceptējot) konkrētus vārdus, visbiežāk speciālgadījumus.

Potenciāli šīs metodes ātrdarbību vēl varētu mēģināt uzlabot. Pieņemsim, ka students iesniedzis automātu M_1 uzdevumam, kurā prasīto valodu akceptē M_2. Šobrīd, lai noteiktu, vai vārds pieder neviena, viena vai abu automātu akceptētajai valodai, jāatceras šī informācija par katru vārdu. Pie alfabēta Σ, šādu vārdu skaits ir $|\Sigma|^n$, kur n ir vārda garums. Tā vietā mēs varētu konstruēt divus automātus M_a un M_s – automātus, kas akceptē attiecīgi M_1 un M_2 akceptēto valodu apvienojumu un šķēlumu. M_s akceptē visus tos vārdus, ko akceptē abi automāti M_1 un M_2. M_a akceptē visus tos vārdus, kurus akceptē M_s, kā arī tos vārdus, kurus akceptē tieši viens no M_1 un M_2. Tagad, lai noskaidrotu, cik ir tādu vārdu garumā n, kurus akceptē tieši viens no M_1 un M_2, nav nepieciešams atcerēties, kādus vārdus akceptē katrs automāts. Varam tikai izskaitīt, cik ir tādu vārdu garumā n, ko akceptē M_s un M_a, un atņemt šos skaitļus. Tā kā M_s akceptē vārdus, ko akceptē *abi* automāti, un M_a akceptē tos vārdus, ko akceptē *vismaz viens* no M_1 un M_2, tad atņemot no M_a akceptēto vārdu skaita M_s akceptēto vārdu skaitu, iegūsim meklēto vērtību. Akceptēto vārdu skaitu var meklēt ar dinamiskās programmēšanas metodēm, kas nozīmē laika sarežģītību $O(n * |Q|)$. Mums jāatceras, cik vārdu garumā n tiek akceptēti, ja ieejas stāvoklis ir q. Pēc tam šos skaitus var summēt, lai iegūtu datus par vārdiem garumā $n+1$.

2.2.4. GDA apstrādes algoritmi

Protams, algoritmu, kas apstrādā automātus, darbība nav iedomājama bez dažādiem palīgalgoritmiem. Ar dažādu algoritmu palīdzību tiek veiktas darbības ar iesniegtajiem GDA, kā arī darbības, kas palīdz vērtēšanas algoritmiem. Šie algoritmi nodrošina, ka automāts vienmēr ir zināmā formā – tas noteikti ir determinēts, attiecīgajos algoritma darbības punktos ir izdevīgi, ka automāts ir minimāls kā arī vairākās vietās nepieciešams noskaidrot vai automāti ir ekvivalenti. Kaut arī visas šīs ir palīgdarbības, to realizācijas algoritmus nepieciešams noprogrammēt un ņemt vērā.

Viens no pirmajiem šādiem algoritmiem, kas ir nepieciešams, ir GDA determinizācijas algoritms. Lai gan darba mērķis ir vērtēt determinētus automātus, nav garantijas, ka students

iesniegs determinizētu risinājumu. Taču, iesniedzot nedeterminizētu risinājumu, šī rīka darbība ne obligāti būs korekta. Datu struktūra, kurā tiek glabāts automāta apraksts, nepieļauj situāciju, kad no viena stāvokļa ir vairākas izejošas pārejas ar vienu un to pašu alfabēta simbolu. Taču ir iespējams automāts, kam no kāda stāvokļa neeksistē visas iespējamās pārejas. Šāda situācija var rasties ne tikai pēc studenta iesniegta automāta, bet arī algoritmu darbības laikā, piemēram, meklējot sintakses kļūdu kāda pāreja tiek dzēsta. Šo īpatnību dēļ determinizācijas algoritms ir implementēts salīdzinoši vienkāršā veidā. Tiek pārbaudīts, vai visas iespējamās pārejas ir nodefinētas, un gadījumā ja nav, tad tās tiek pievienotas un novirzītas uz jaunu "miskastes" stāvokli. Šīs operācijas laika sarežģītība ir $O(|Q| * |\Sigma|)$, jo vienīgais, ko nepieciešams izdarīt, ir pārlasīt visas pārejas.

Konvertējot MSO uz GDA, viens no pamata principiem ir pārveidošana uz GNA. Tajā gadījumā arī būs vajadzīgs ieviest atbilstošu datu struktūru, kā arī funkciju, kas determinizē GNA. Šāds algoritms ir zināms jau pietiekami sen. Lielākā problēma ar šo algoritmu ir sliktākā gadījuma sarežģītība. Pastāv iespēja, ka determinizējot GNA stāvokļu skaits pieaugs eksponenciāli [21]. Alternatīvu risinājumu diemžēl nav – ja ir nepieciešams strādāt ar GDA, tad jāņem vērā šis palielināšanās risks.

Lai efektīvi darbotos ar GDA, tos vispirms var un vajag minimizēt. Minimizēšana potenciāli samazina automāta izmēru, nemainot akceptēto valodu. Minimizēšanas algoritmi ir zināmi jau pietiekami sen un tiek plaši lietoti praksē. Šim algoritmam ir divi soļi: nesasniedzamo stāvokļu dzēšana un neatšķiramo stāvokļu apvienošana. Pirmais tiek realizēts ar BFS algoritmu apstaigājot visus sasniedzamos stāvokļus un pārējos dzēšot. Pēc šī algoritma izpildes nepieciešams apvienot neatšķiramos stāvokļus. Šī soļa realizācija veikta ar diezgan zināmu algoritmu – *table filling* algoritmu. Vispirms katrs akceptējošais stāvoklis tiek atzīmēts kā atšķirams no katra neakceptējošā stāvokļa, visi pārējie stāvokļu pāri uzskatāmi par neatšķiramiem. Pēc tam iteratīvi tiek pārlasīti visi neatšķiramie stāvokļu pāri, un, ja ar kādu alfabēta simbolu no šiem stāvokļiem iespējams nonākt atšķiramos stāvokļos, tad arī šie stāvokļi tiek atzīmēti kā atšķirami. Kad vairs neeksistē tāds neatšķiramu stāvokļu pāris, no kura varētu nokļūt uz kādu atšķiramu stāvokļu pāri, algoritms beidz darbu [20, pp. 155-163].

Atliek pārbaudīt divu automātu ekvivalenci. Šo ekvivalenci var pārbaudīt veicot viena automāta apstaigāšanu ar BFS algoritma palīdzību attiecīgi atzīmējot vienādos stāvokļus abos automātos. Vispirms abu automātu sākuma stāvokļi tiek atzīmēti kā ekvivalenti, ja abi ir vai nu akceptējoši vai arī neakceptējoši. Abos automātos vienlaikus tiek veikta pāreja ar kādu konkrētu simbolu. Nonākot stāvoklī, kas iepriekš nav bijis aplūkots, tiek salīdzināti akceptēšanas nosacījumi, un sakritības gadījumā stāvokļi abos automātos atzīmēti kā ekvivalenti. Ja tiek atrasts kāds stāvoklis, kas jau bijis sastapts iepriekš, tad tiek pārbaudīts, vai attiecīgais stāvoklis

otrā automātā atbilst iepriekš atzīmētajam. Nepieciešams pārlasīt visas pārejas, tāpēc sarežģītība ir $O(|Q| * |\Sigma|)$

2.2.5. GDA konstruēšana no MSO formulas

Ļoti būtiska no "uzdevuma nosacījumu nepareizas interpretācijas" vērtēšanas metodes algoritma ir GDA uzbūvēšana no MSO formulas. Šis solis ir nepieciešams pārbaudot, vai uzģenerētā MOSEL izteiksme atbilst dotam automātam. MSO izteiksmes izveidošana no MOSEL apraksta nav pārāk sarežģīta, taču GDA iegūšana nav tik vienkārša, tāpēc šajā nodaļā tiks aprakstīts algoritms, ko varētu izmantot šīs darbības veikšanai.

Algoritms pārejai no MSO uz GDA balstās uz rekursīvu algoritmu, kas katru MSO termu pārveido par attiecīgu automātu. Ir daži pamata termi, no kuriem uzreiz tiek veidots konkrētas konstrukcijas automāts, pārējiem termiem tiek izmantota rekursija, ar kuras palīdzību šī terma apakšizteiksmes tiek pārveidotas par attiecīgajiem automātiem un tad šie automāti tiek attiecīgi apvienoti. Lielākā problēma šajā konstrukcijā ir, ka galā tiek iegūts nevis GDA, bet gan GNA, ko vēl nepieciešams determinizēt. Šī pēdējā operācija nosaka augstu konstrukcijas teorētisko sarežģītību, vēl pat nerēķinot sarežģītību paša GNA izveidošanai. Šo sarežģītību gan nav iespējams objektīvi samazināt – ja ir nepieciešams izmantot šo vērtēšanas metodi, tad jārēķinās ar potenciāli augstu algoritma sarežģītību.

Automātu konstruēšanas algoritmam ir pāris īpašības, kas atvieglo šī algoritma uztveramību un darbību. Pirmkārt, MSO formulās tiek aizstāti visi pirmās pakāpes mainīgie. MSO formulas sastāv no pirmās un otrās pakāpes mainīgajiem, kā arī dažādām operācijām. Lai vienkāršotu automātu konstruēšanu, dotā MSO formula tiek pārveidota par MSO_0 formulu, kas

$$\begin{aligned} MSO_0 \coloneqq\ & S(X, Y) \mid X \subseteq Y \mid X \subseteq Q_a \mid X < Y \mid Sing(X) \mid \\ & \mid \psi \mid \psi \vee \phi \mid \\ & \mid \exists X\ \psi \end{aligned}$$

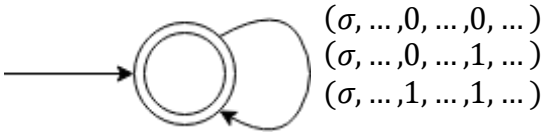
2.3. att. **MSO_0 semantika**

satur tikai otrās pakāpes mainīgos un ir ekvivalenta iepriekš dotajai MSO formulai [22, p. 19]. Tādā veidā nav jāuztraucas par pirmās pakāpes mainīgajiem un konstrukcijā var izmantot tikai kopas. 3. pielikumā ir redzama konkrētu MSO termu pāreja uz MSO_0 termiem. Ar uzdotajām operācijām pirmās pakāpes mainīgie tiek aizstāti par attiecīgiem otrās pakāpes mainīgajiem. Lai nodrošinātu, ka vēl var izmantot pirmās pakāpes mainīgos, tiek ieviesta papildu operācija, kas pārbauda, ka kopa (otrās pakāpes mainīgais) satur tieši vienu elementu – *Sing(X)*. Vēl viena īpašība, kas nepieciešama korektu automātu konstruēšanai ir alfabēta pielāgošana. Tā kā tagad darbības notiek ar kopām, katrai pārejai nepieciešama papildu norāde par simbola piederību konkrētām kopām. Ja MSO izteiksmē parādās *n* kopas, tad papildinātais MSO_0 ieejas alfabēts

būs $\Sigma \times \{0,1\}^n$. Šāds ieejas alfabēts ļauj konkrēti norādīt, kurām kopām katrs simbols pieder. Varētu rasties jautājums, kā pēc tam iegūt automātu ar jau zināmo ieejas alfabētu, bet tā nav problēma – vienkārši tiek veidota projekcija uz Σ. Rezultātā uzreiz izveidojas GNA, kurš prot uzminēt īstās pārejas [5], [22, p. 21].

Kad MSO izteiksme ir pārveidota par MSO_0 izteiksmi, atliek to pārveidot par GNA. 2.3. attēlā dots MSO_0 sintakses apraksts, visas pārējās darbības ir izsakāmas izmantojot dotos termus un dažādas loģikas operācijas. No šī arī izriet, ka nepieciešams nodefinēt precīzu GNA konstrukcijas shēmu katram no 2.3. attēlā dotajiem termiem. Dotie termi vai nu veido konkrētu GNA, vai rekursīvi konstruē apakšformulu automātus un tos pārveido vai apvieno. Tālākos piemēros tiek pieņemts, ka kopējais otrās pakāpes mainīgo skaits ir n.

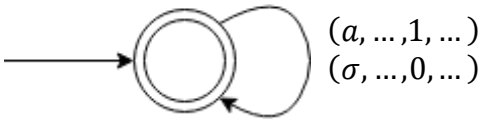
- $X_i \subseteq X_j$:



2.4. att. **GNA, kas atbilst MSO izteiksmei $X_i \subseteq X_j$**

Attiecība norāda, ka X_i ir X_j apakškopa. Tātad nepieciešams konstruēt automātu, kas akceptē tos simbolus, kam piederība kopām X_i un X_j apmierina šo prasību. Automāta pārejai vienīgais nosacījums ir, ka tā akceptē tad un tikai tad ja nolasītais elements ir pieder kopai X_i tikai tad, ja tas pieder arī kopai X_j. 2.4. attēlā dots atbilstošs automāts pie nosacījuma, ka $i < j$. Pārejā simbola σ vietā var atrasties jebkurš simbols, kas pieder ieejas alfabētam Σ. Daudzpunktes vietā atrodas jebkāda *0* un *1* kombinācija, kas ir atbilstošajā garumā, lai fiksētie simboli atrastos attiecīgi pozīcijās i un j. Viegli pārliecināties par attiecības korektumu. Piemēram, nolasītais simbols pieder kopai X_i, bet nepieder kopai X_j. Tad pozīcijās i un j pārejas vektorā atradīsies attiecīgi 1 un 0: $(\sigma, \dots, 1, \dots, 0, \dots)$. Tā kā X_i nav X_j apakškopa, tad šādai pārejai nevajadzētu būt iespējamai, kas tā arī ir.

- $X_i \subseteq Q_a$:



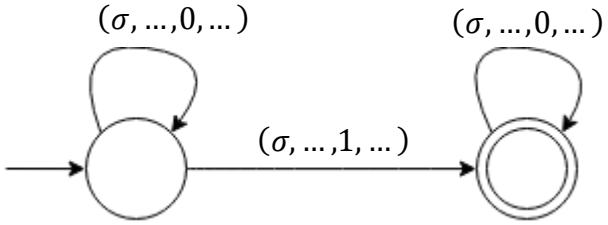
2.5. att. **GNA, kas atbilst MSO izteiksmei $X_i \subseteq Q_a$**

Gadījums, kad nepieciešams pārliecināties, ka nolasītais simbols atbilst konkrētam ieejas alfabēta simbolam, ir ļoti līdzīgs apakškopu meklēšanai. Izteiksme $X_i \subseteq Q_a$ semantiski nozīmē, ka visi simboli kopā X_i ir 'a'. Tātad nepieciešams pārliecināties, ka visi simboli, kas pieder kopai X_i, ir 'a'. Pārējie simboli drīkst būt patvaļīgi ieejas alfabēta simboli, tajā skaitā 'a'. Rezultējošais automāts ir parādīts 2.5. attēlā. Analogi kā iepriekšējā gadījumā, tiek fiksēta

pozīcija i un ieejas simbols. Simbols σ tiek aizstāts ar visiem Σ simboliem, arī 'a'. Mums neinteresē piederība pārējām kopām X_j ($j \neq i$), tāpēc tajās pozīcijās ir pieļaujams patvaļīgs simbols.

- $Sing(X_i)$:

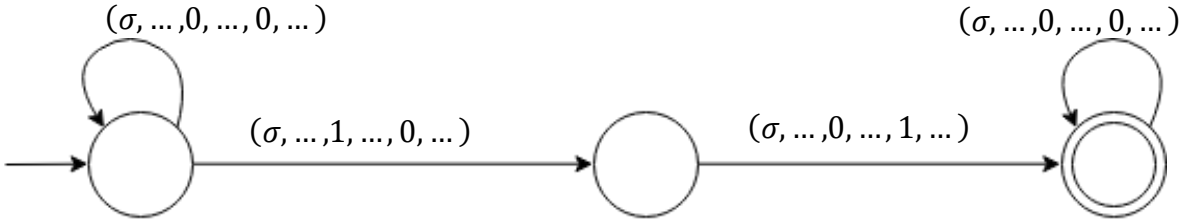
Izteiksme ir patiesa gadījumos, kad kopa X_i satur tieši vienu elementu. Zinot ieejas alfabēta simbolu nozīmi ir viegli saprast, kādam jāizskatās attiecīgajam automātam, kas apmierinās šo izteiksmi. Nepieciešams pārbaudīt, vai konkrētajai kopai ir tieši viens elements, kas tai pieder. 2.6. attēlā redzamais automāts apmierina šo prasību. Nonākt akceptējošajā stāvoklī ir iespējams tikai tad, ja kopai pieder elements, un gadījumā, ja kopai pieder vēl kāds elements, neeksistē atbilstoša pāreja un automāts pamet akceptējošo stāvokli.



2.6. att. **GNA, kas atbilst MSO izteiksmei *Sing(X_i)***

- $S(X_i, X_j)$:

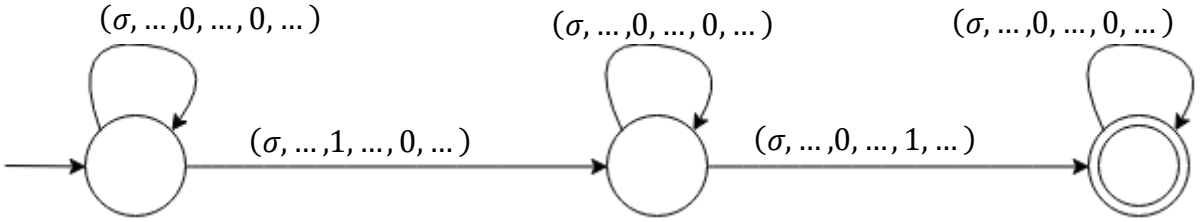
Ir būtiski vispirms izprast šī terma semantisko nozīmi, lai labāk izprastu konstrukcijas, kas izmantotas reprezentējošā automāta būvē. Sākotnēji šī operācija ir mantota no pirmās pakāpes mainīgo operācijas $S(x,y)$, tāpēc pierādījumā tiks izmantotas singulāras kopas. Tomēr tas nenozīmē, ka šie principi nedarbotos arī uz nesingulārām kopām. Terma semantiskā nozīme ir, ka pēc kopas X_i elementa seko kopas X_j elements. Singulāru kopu gadījumā viegli ievērot, ka tas faktiski nozīmē, ka pēc elementa x seko elements y. Izprotot semantisko nozīmi, ir arī viegli saprast, kā izveidot atbilstošu automātu, kas akceptēs šādu konstrukciju. Gadījumos, kad nolasītais simbols nepieder nevienai no kopām X_i vai X_j, automāts paliek sākuma stāvoklī. Automāta darbība virzās uz priekšu vispirms nolasot simbolu, kas pieder X_i, pēc tam nolasot simbolu, kas pieder X_j. Nonākot akceptējošā stāvoklī, automāts akceptēs tikai simbolus, kas neietilpst nevienā no kopām X_i un X_j.



2.7. att. **GNA, kas atbilst MSO izteiksmei *S(X_i, X_j)***

- $X_i < X_j$:

Šis gadījums ir ļoti līdzīgs iepriekšējam gadījumam. Vienīgā atšķirība no iepriekšējā gadījuma ir, ka X_j elements var būt ne uzreiz aiz X_i elementa. Līdz ar to nedaudz jāpielāgo iepriekš uzdotais automāts, lai tas pieļautu citu elementu esamību starp X_i un X_j. Atbilstošais automāts ar papildu pāreju ir uzdots 2.8. attēlā.



2.8. att. **GNA, kas atbilst MSO izteiksmei $X_i < X_j$**

- $\neg\psi$:

Šajā gadījumā automāta konstrukcija ir viegli uztverama, bet ar lielu skaitļošanas sarežģītību. Dotā izteiksme semantiski nozīmē, ka rezultējošais automāts akceptēs ψ akceptētās valodas papildinājumu. Šo darbību ir viegli veikt ar GDA – atliek tikai pārdefinēt akceptējošo stāvokļu kopu $F = Q \setminus F$. Tādejādi visi vārdi, kas iepriekš tika akceptēti vairs netiek akceptēti un otrādi. Taču veikt analogu operāciju GNA nav tik vienkārši, atliek vien konstruēt atbilstošo GDA.

- $\psi \vee \phi$:

Nepieciešams, lai izpildās vismaz viena no abām formulām. Uzbūvējot formulām atbilstošos automātus A_ψ un A_ϕ, nepieciešams, lai kāds no automātiem akceptē ieejas vārdu. GNA gadījumā nav sarežģīti izveidot automātu, kas ir divu citu automātu apvienojums un akceptē abu automātu akceptēto valodu apvienojumu. Atliek izveidot jaunu stāvokli, un ar divām ε-pārejām pāriet uz attiecīgi A_ψ un A_ϕ sākuma stāvokļiem. Sākot darbu, apvienojuma automāts "uzminēs", kurš no abiem automātiem akceptētu ieejas vārdu un veiks attiecīgo pāreju uz to automātu.

- $\exists X_n\ \psi$:

Izteiksme semantiski nosaka, ka eksistē tāda kopa X_n, ka n argumentu ψ izpildās. Rekursīvi varam uzkonstruēt automātu, kas darbojas ar ieejas alfabētu $\Sigma \times \{0,1\}^n$ un apmierina ψ. Apzīmēsim $\phi = \exists X^n\ \psi$. Tādā gadījumā ϕ ir *n-1* argumenta, un ir patiesa tad, ja tās argumenti der kā ψ pirmie *n-1* argumenti. Eksistences kvantors nosaka to, ka eksistē kaut kāds X_n, kas apmierina ψ, šo X_n nekādā veidā nevaram ietvert ϕ argumentu sarakstā. Šīs atomārās funkcijas pārveidošanai par ekvivalentu GNA nav nepieciešams veikt nekādus pārveidojumus automātam, kas apmierina ψ. Ja eksistē mainīgo kopa, kas apmierina ψ, tad eksistē X_n, kas apmierina ψ. Lai iegūtu korektu GNA atliek tikai samazināt ieejas alfabētu uz

$\Sigma \times \{0,1\}^{n-1}$ un koriģēt pārejas. Pāreju koriģēšana var tikt veikta ar projekciju izmantošanu izslēdzot attiecīgo indeksu visām ψ esošajām pārejām [5], [22, pp. 20-23].

2.3. Izstrādātā servisa saskarne

Lai nodrošinātu pilnvērtīgu servisa darbību, nepietiek tikai ar algoritmu implementāciju, bet arī jānodrošina lietotāja saskarne ar šo servisu. Šī bakalaura darba ietvaros nebija paredzēts izstrādāt tīmekļa vietni ar vizuālu lietotāja saskarni, kas arī netika darīts. Taču bija paredzēts izstrādāt API piekļuvi servisam. Vēl viena no prasībām, kas tika izvirzītas darba sākumā, bija konfigurējama servisa izstrāde. Darba gaitā abas šīs prasības tika izpildītas.

2.3.1. Tīmekļa serveris

Pateicoties GoLang jau iebūvētajām tīmekļa servera funkcijām, tīmekļa servera izstrāde nebija problemātiska. Kā jau iepriekš aprakstīts, GoLang ir salīdzinoši jauna programmēšanas valoda, tāpēc tajā ir piedāvātas mūsdienīgas iespējas arī bez papildu bibliotēkām. Faktiski tika izmantota autoram jau iepriekš zināma bibliotēka, kas sniedz plašākas iespējas salīdzinot ar iebūvētā tīmekļa servera funkcionalitāti, piemēram, ļaujot norādīt atļautās pieprasījuma metodes. Servisa arhitektūra, kurā jau iekļauts tīmekļa serveris, ļoti atvieglo servisa uzstādīšanu un darbināšanu. Nav nepieciešams izmantot vēl kādu papildu programmatūru, lai novirzītu datu plūsmu no tīmekļa uz servisu, tas pats sāk klausīties norādīto portu. Izstrādātajā risinājumā lietotājs padod informāciju par automātiem JSON formātā, un atbildē saņem automātam piešķirto vērtējumu.

Tiek sagaidīti divi automāti – studenta risinājums un korekts automāts, pret kuru tiks veikta salīdzināšana. Pēc uzdevuma nosacījumu nepareizas interpretācijas metodes implementācijas datu formāts varētu tikt mainīts – korektā automāta vietā saņemot šo automātu aprakstošu MOSEL izteiksmi. Pēc tam no šīs izteiksmes varētu uzkonstruēt atbilstošo GDA ar jau iepriekš aprakstītajām metodēm un algoritmiem.

Saņemto datu formāts ir veidots pēc iespējas tuvāks dabiskam GDA aprakstam. Katram automātam nepieciešams norādīt:

- stāvokļu kopu – katrs stāvoklis tiek identificēts ar teksta mainīgo, teksta masīvs;
- sākuma stāvokli – teksta mainīgais;
- ieejas alfabētu – masīvs ar teksta mainīgajiem, kur katrs apzīmē vienu ieejas simbolu;
- akceptējošo stāvokļu kopu – masīvs ar teksta mainīgajiem;
- pārejas funkciju – masīvs ar vienu pāreju raksturojošiem objektiem:
    - pārejas simbols (teksts);

- pārejas izejas stāvoklis (teksts);
- pārejas mērķa stāvoklis (teksts).

Automāta vērtēšanas laikā lietotājs gaida atbildi. Atkarībā no konfigurācijas, atbilde un vērtējums par automāta risinājumu tiek atgriezts dažu sekunžu laikā. Servera logu failos tiek drukāti dažādi starprezultāti, kuri var tikt izmantoti atkļūdošanas vajadzībām. Pēc automāta novērtēšanas tiek atgriezts iegūtais rezultāts kā arī daži papildu parametri:

- statuss – norāde par veiksmīgu vai neveiksmīgu datu apstrādi;
- ziņa – īss rezultātu aprakstošs teksts;
- kļūda – kļūdas apraksts, ja bijusi kļūda;
- iegūtais rezultāts – reāls skaitlis ar automātam piešķirto vērtējumu;
- maksimālais rezultāts – reāls skaitlis ar maksimāli iegūstamo vērtējumu, kas norādīts servera konfigurācijā;
- nelielu valodas atšķirību metodes rezultāts – vērtējums, kas piešķirts konkrētajā vērtēšanas metodē;
- sintakses kļūdu metodes rezultāts – vērtējums, kas piešķirts konkrētajā vērtēšanas metodē.

Datu struktūru aprakstus JSON formātam līdzīgā pierakstā iespējams aplūkot pie izstrādātā risinājuma repozitorijas vietnē *GitHub*. Šajā repozitorijā, kā arī 4. pielikumā redzami ieejas datu piemēri.

2.3.2. Konfigurācija

Otra no pamata prasībām servisam bija konfigurācijas iespēja. Iepriekš zināmais serviss nepiedāvāja nekādu konfigurācijas iespēju, visi nepieciešamie pielāgojumi bija jāveic uzreiz kodā [8]. Šāda arhitektūra praksē ir ļoti neizdevīga un mūsdienu prasībām neatbilstoša. Tāpēc šajā darbā izstrādātajā risinājumā tika paredzētas konfigurācijas iespējas saprātīgā apjomā. Konfigurācija pieejama tādām lietām kā maksimālais iegūstamais vērtējums, vērtēšanas metožu parametri (piemēram, valodu atšķirību metodei – vārdu garums, laika ierobežojums, gala pielāgošanas koeficienti, u.tml.). Konfigurācijas faili pieļaujami JSON vai YAML formātā, un, protams, lietotājam iespējams norādīt šo failu atrašanās vietu. Konfigurācijas failu neesamības gadījumā, tiek izmantotas standarta vērtības, kas gan ir norādītas pirmkodā un ir konfigurējamas tikai caur koda izmaiņām. Servisā paredzēta arī iespēja no jauna ielasīt konfigurācijas failu saturu nepārstartējot pašu servisu. Šī funkcionalitāte nodrošina maksimāli augstu sistēmas pieejamību, mainoties prasībām [23].

REZULTĀTI UN SECINĀJUMI

Nav iespējams pilnvērtīgi apgūt datorzinātņu pamatus neapgūstot teoriju par Tūringa mašīnām, ko savukārt nav iespējams apgūt bez mācīšanās par galīgiem determinētiem automātiem. Kā jebkurai tēmai, praktiski uzdevumi ir nozīmīga sastāvdaļa no GDA apguves, attiecīgi nozīmīgs darbs ir arī šos risinājumus vērtēt. Par spīti automātu determinētībai, līdz šim ir bijis tikai viens nozīmīgs pētījums par automatizētu risinājumu novērtēšanu.

Bakalaura darba ietvaros tika analizēti dažādi informācijas avoti, gūstot plašāku izpratni par automātu teorijas pamatprincipiem kā arī sarežģītākiem modeļiem, kā aprakstīt regulāras valodas. Analizējot iepriekš veiktu pētījumu par GDA automātisku novērtēšanu, tika secināts, ka līdz šim nav bijis daudz progresa šīs tēmas izpētē kā arī tika novērtētas iespējamās metodes kā izstrādāt praksē strādājošu vērtēšanas rīku. Pēc esošo risinājumu izpētes, tika noskaidroti trīs iespējamo kļūdu veidi kā arī metodes, ar kuru palīdzību šādas kļūdas varētu konstatēt. Šie trīs kļūdu veidi ir: sintakses kļūdas korektā risinājumā; korekts risinājums līdzīgam risinājumam; korekts risinājums, kas akceptē nedaudz atšķirīgu valodu. Piedāvātā metode optimāli darbojas uz gandrīz korektiem automātiem, daļēji korektu automātu vērtēšanai nepieciešama papildu izpēte.

Pēc teorētiskās izpētes tika veikts darbs pie praktiskas vērtēšanas sistēmas izstrādes. Faktiski tika izstrādātas divas no trim vērtēšanas metodēm – sintakses kļūdas korektā risinājumā un korekts risinājums līdzīgai valodai –, trešās metodes implementācijai tika veikta teorētiskā sagatavošanās ar algoritmu izpēti, bet nepietika laika, lai izstrādātu darbojošos implementāciju. Galvenās problēmas izstrādes gaitā bija saistītas ar algoritmu optimizāciju, jo daudzas no problēmām, kas tika risinātas, ir ar eksponenciālu sarežģītību. Šo ierobežojumu dēļ risinājums optimāli darbojas uz maziem automātiem.

Bakalaura darba laikā mērķis tika sasniegts un tika izstrādāta GDA vērtēšanas sistēma, kas izmantojama kā tīmekļa serviss. Tika veikta teorētiska izpēte un noskaidrotas metodes, kā varētu vērtēt GDA būves uzdevumus. Izstrādātā sistēma spēj atpazīt divu veidu kļūdas, kā arī to ir iespējams attīstīt, lai tas spētu atšķirt vēl citus kļūdu veidus. Darba izstrādes gaitā autors pilnveidoja savas zināšanas par automātu teoriju un tajā izmantojamajiem algoritmiem, kā arī uzlaboja savas programmēšanas prasmes izmantotajā valodā.

Šajā darbā piedāvāto risinājumu būtu iespējams attīstīt tālāk ar vairāku metožu palīdzību. Darbā tika pieminēta specifiskas sintakses izveide, ar kuras palīdzību varētu translēt dabiskajā valodā izteiktus uzdevuma nosacījumus par regulārajām izteiksmēm. Varētu izstrādāt paņēmienu kā vērtēt vairāku kļūdu vienlaicīgu konstatēšanu, jo viena no konstatējamām nepilnībām ir servisa nespēja konstatēt vairāku kļūdas tipu vienlaicīgu atrašanos risinājumā.

Rīka funkcionalitāti varētu uzlabot izmantojot mašīnmācīšanās metodes, tādejādi apgūstot vēl jaunus kļūdu paveidus. Kā arī nākotnē varētu papildināt praktisko implementāciju ar uzdevuma nosacījumu nepareizas izpratnes konstatēšanu un papildu funkcionalitāti, kas ļautu, piemēram, darbināt rīku meklējot tikai konkrēta veida kļūdas. Protams, praktisko implementāciju varētu uzlabot optimizējot izmantotos algoritmus.

IZMANTOTĀ LITERATŪRA UN INFORMĀCIJAS AVOTI

[1] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London mathematical society,* vol. 2, no. 1, pp. 230-265, 1937.

[2] R. Alur, L. D'Antoni, S. Gulwani, D. Kini and M. Viswanathan, "Automated Grading of DFA Constructions," *IJCAI,* vol. 13, pp. 1976-1982, 2003.

[3] «Automata Tutor,» [Tiešsaiste]. Available: http://www.automatatutor.com. [Piekļūts 15 04 2018].

[4] P. Kelb, T. Margaria, M. Mendler and C. Gsottberger, "MOSEL: A flexible toolset for monadic second-order logic," in *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Springer, 1997, pp. 183-202.

[5] R. D. Wehr, *Monadic Second Order Logic and Automata on Infinite Words: Büchi's Theorem,* 2007.

[6] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problem," *SIAM journal on computing,* vol. 18, pp. 1245-1262, 1989.

[7] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM journal of research and development,* vol. 3, no. 2, pp. 114-125, 1959.

[8] «Automata Tutor · GitHub,» [Tiešsaiste]. Available: https://github.com/AutomataTutor. [Piekļūts 05 04 2017].

[9] «Frequently Asked Questions (FAQ) - The Go Programming Language,» [Tiešsaiste]. Available: https://golang.org/doc/faq#Origins. [Piekļūts 04 04 2018].

[10] «GoUsers · golang/go Wiki · GitHub,» [Tiešsaiste]. Available: https://github.com/golang/go/wiki/GoUsers. [Piekļūts 10 05 2018].

[11] «net - GoDoc,» [Tiešsaiste]. Available: https://golang.org/pkg/net/. [Piekļūts 10 04 2018].

[12] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, "Power challenges may end the multicore era," *Communications of the ACM,* vol. 56, no. 2, pp. 93-102, 2013.

[13] «sync - GoDoc,» [Tiešsaiste]. Available: https://godoc.org/sync. [Piekļūts 20 04 2018].

[14] «html - GoDoc,» [Tiešsaiste]. Available: https://godoc.org/html/. [Piekļūts 30 04 2018].

[15] «About - GoDoc,» [Tiešsaiste]. Available: https://godoc.org/-/about. [Piekļūts 03 04 2018].

[16] "GitHub - lytics/dfa: Deterministic Finite Automata to define computation with labeled states and explicit transitions," [Online]. Available: https://github.com/lytics/dfa. [Accessed 15 04 2018].

[17] «Effective Go - The Go Programming Language,» [Tiešsaiste]. Available: https://golang.org/doc/effective_go.html#maps. [Piekļūts 15 04 2018].

[18] «GitHub - c4e8ece0/goted: Tree Edit Distance,» [Tiešsaiste]. Available: https://github.com/c4e8ece0/gote. [Piekļūts 10 05 2018].

[19] L. Libkin, "MSO on Strings and Regular Languages," in *Elements of Finite Model Theory*, Springer, 2012, pp. 124-126.

[20] J. E. Hopcroft, Introduction to automata theory, languages, and computation, Pearson Education India, 2008.

[21] R. Van Glabbeek and B. Ploeger, "Five determinisation algorithms," in *International Conference on Implementation and Application of Automata*, Springer, 2008, pp. 161-170.

[22] W. Thomas, "Applied automata theory," *Course Notes, RWTH Aachen*, 2005.

[23] «GitHub - spf13/viper: Go configuration with fangs,» [Tiešsaiste]. Available: https://github.com/spf13/viper. [Piekļūts 13 05 2018].

PIELIKUMI

1. Pielikums
MOSEL konkrētā sintakse

```
ID: (a..z)(a..z | A..Z | | . | 0..9)*
CID: (A..Z)(a..z | A..Z | | . | 0..9)*
x: ID
X: CID
a: (a..z)
s: (a..z)*
m, n: (0..9)*
φ ::= φ C φ | ¬ φ | Q x.φ | Q X.φ | true | false | P CMP P | a@P
    | a@S | P∈S | |S|% m CMP n | |S| CMP n | begWt 's' | endWt
    's' | isEmpty
P ::= x | fst | last | P + 1 | P - 1 | fstOc 's' | lastOc 's'
S ::= X | indOf 's' | S SC S | all | | psLt P | psLe P | psGt P
    | psGe P
C ::= ∧ | ∨
Q ::= ∀ | ∃
CMP ::= ≤ | ≥ | = | < | >
SC ::= ∩ | ∪
```

ID apzīmē identifikatorus, *CID* vārda identifikatorus. *x* un *X* apzīmē attiecīgi pirmās un otrās pakāpes mainīgos. *a* ir simbola konstante, *s* – vārda konstante, *m* un *n* – skaitļa konstantes.

Predikātu semantika

$s, I \models \varphi_1\ C\ \varphi_2$	ja $(s, I \models \varphi_1)\ C\ (s, I \models \varphi_2)$
$s, I \models \neg\varphi$	ja $s, I \not\models \varphi$
$s, I \models Q\ x.\varphi$	ja $Qj \in [1..\lvert s\rvert].\ (s, I[j/x] \models \varphi)$
$s, I \models Q\ X.\varphi$	ja $QJ \subseteq [1..\lvert s\rvert].\ (s, I[J/X] \models \varphi)$
$s, I \models$ true	
$s, I \not\models$ false	
$s, I \models P_1\ CMP\ P_2$	ja $P_1 \Rightarrow_{s,I} i_1 \wedge P_2 \Rightarrow_{s,I} i_2 \wedge i_1\ CMP\ i_2$
$s, I \models a@P$	ja $\exists i,\ P \Rightarrow_{s,I} i \wedge s_i = a$
$s, I \models a@S$	ja $\exists J,\ S \Rightarrow_{s,I} J \wedge \forall_j \in J,\ s_j = a$
$s, I \models P \in S$	ja $P \Rightarrow_{s,I} j \wedge S \Rightarrow_I J \wedge I(j) \in I(J)$
$s, I \models \lvert S\rvert \%\ m\ CMP\ n$	ja $\exists J,\ S \Rightarrow_{s,I} J \wedge \lvert J\rvert \%\ m\ CMP\ n$
$s, I \models \lvert S\rvert\ CMP\ n$	ja $\exists J,\ S \Rightarrow_{s,I} J \wedge \lvert J\rvert\ CMP\ n$
$s, I \models$ begWt ‘s_1’	ja $s_1 \in PR(s)$
$s, I \models$ endWt ‘s_1’	ja $s_1 \in SUF(s)$
$s, I \models$ isEmpty	ja $\lvert s\rvert = 0$

Pozīciju semantika

$x \Rightarrow_{s,I} j$	ja $I(x) = j$
$fst \Rightarrow_{s,I} 1$	ja $\lvert s\rvert > 0$
$last \Rightarrow_{s,I} \lvert s\rvert$	ja $\lvert s\rvert > 0$
$P + 1 \Rightarrow_{s,I} j + 1$	ja $P \Rightarrow s,I\ j \wedge j < \lvert s\rvert$
$P - 1 \Rightarrow_{s,I} j - 1$	ja $P \Rightarrow s,I\ j \wedge j > 1$
$fstOc\ `s`' \Rightarrow_{s,I} j$	ja $s` \in PR(s_{j,\lvert s\rvert}) \wedge \nexists j` < j.s` \in PR(s_{j`,\lvert s\rvert})$
$lastOc\ `s`' \Rightarrow_{s,I} j$	ja $s` \in PR(s_{j,\lvert s\rvert}) \wedge \nexists j` > j.s` \in PR(s_{j\,0,\lvert s\rvert})$

Kopu semantika

$X \Rightarrow_{s,I} J$	ja $I(x) = J$
indOf ‘s_1’ $\Rightarrow_{s,I} J$	ja $J = \{j \mid s_1 \in PR(s_{j,\lvert s\rvert})\}$
$S_1\ SC\ S_2 \Rightarrow_{s,I} J_1\ SC\ J_2$	ja $S_1 \Rightarrow_{s,I} J_1 \wedge S_2 \Rightarrow_{s,I} J_2$
$all \Rightarrow_{s,I} [1..\lvert s\rvert]$	
$psLt\ P \Rightarrow_{s,I} [1..j - 1]$	ja $P \Rightarrow_{s,I} j$
$psLe\ P \Rightarrow_{s,I} [1..j]$	ja $P \Rightarrow_{s,I} j$
$psGt\ P \Rightarrow_{s,I} [j + 1..\lvert s\rvert]$	ja $P \Rightarrow_{s,I} j$
$psGe\ P \Rightarrow_{s,I} [j..\lvert s\rvert]$	ja $P \Rightarrow_{s,I} j$

$x < y$	$X_x < X_y$
$S(x,y)$	$S(X_x, X_y)$
$x = y$	$X_x \subseteq X_y \wedge X_y \subseteq X_x$
$Q_a(y)$	$Sing(X_y) \wedge X_y \subseteq Q_a$
$y \in X$	$Sing(X_y) \wedge X_y \subseteq X$
$\psi \rightarrow \phi$	$\neg\psi \vee \phi$
$\psi \wedge \phi$	$\neg(\neg\psi \vee \neg\phi)$
$\forall x\ \psi$	$\neg\exists x\ (\neg\psi)$
$\forall X\ \psi$	$\neg\exists X\ (\neg\psi)$
$\exists x\ \psi$	$\exists X_x\ (Sing(X_x) \wedge \psi)$

X_x ir injektīvi izveidota kopa, kura satur tieši vienu mainīgo $-x$.

Sing(X) ir funkcija, kas ir patiesa tad un tikai tad, ja kopa *X* satur tieši vienu elementu.

```
{
    "target": {
        "alphabet": ["0", "1"],
        "start_state": "0",
        "final_states": ["3"],
        "states": ["0", "1", "2", "3", "4", "5"],
        "transitions": [
            { "from": "0", "symbol": "0", "to": "1" },
            { "from": "0", "symbol": "1", "to": "2" },
            { "from": "1", "symbol": "0", "to": "1" },
            { "from": "1", "symbol": "1", "to": "1" },
            { "from": "2", "symbol": "0", "to": "3" },
            { "from": "2", "symbol": "1", "to": "2" },
            { "from": "3", "symbol": "0", "to": "1" },
            { "from": "3", "symbol": "1", "to": "4" },
            { "from": "4", "symbol": "0", "to": "5" },
            { "from": "4", "symbol": "1", "to": "4" },
            { "from": "5", "symbol": "0", "to": "1" },
            { "from": "5", "symbol": "1", "to": "2" }
        ]
    },
    "attempt": {
        "alphabet": ["0", "1"],
        "start_state": "0",
        "final_states": ["3"],
        "states": ["0", "1", "2", "3", "4"],
        "transitions": [
            { "from": "0", "symbol": "0", "to": "1" },
            { "from": "0", "symbol": "1", "to": "2" },
            { "from": "1", "symbol": "0", "to": "1" },
            { "from": "1", "symbol": "1", "to": "1" },
            { "from": "2", "symbol": "0", "to": "3" },
            { "from": "2", "symbol": "1", "to": "2" },
            { "from": "3", "symbol": "0", "to": "1" },
            { "from": "3", "symbol": "1", "to": "4" },
            { "from": "4", "symbol": "0", "to": "2" },
            { "from": "4", "symbol": "1", "to": "4" }
        ]
    }
}
```

Bakalaura darbs „***Galīgu determinētu automātu būves uzdevumu automātiska vērtēšana***" izstrādāts Latvijas Universitātes Datorikas fakultātē.

Ar savu parakstu apliecinu, ka darbs izstrādāts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: ***Artūrs Jānis Pētersons*** _____ **___.05.2018**.

Rekomendēju darbu aizstāvēšanai
Darba vadītājs: ***docents, Dr.sc.comp. Agris Šostaks*** _____ **___.05.2018.**

Recenzents: ***profesors, Dr.sc.comp. Juris Vīksna***

Darbs iesniegts Datorikas fakultātē 28.05.2018.

Dekāna pilnvarotā persona:
vecākā metodiķe Ārija Sproģe_____

Darbs aizstāvēts kursa darbu pārbaudījuma komisijas sēdē
___.06.2018. prot. Nr. _____

Komisijas sekretārs(-e): _____