VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Srautinio CIF parserio (sintaksinio analizatoriaus) sukūrimas

Creating a streaming CIF parser

Magistro baigiamasis darbas

Atliko: Mindaugas Legeckas (parašas)

Darbo vadovas: prof. dr. Saulius Gražulis (parašas)

Recenzentas: (parašas)

Vilnius – 2023

Santrauka

Esami nesrautiniai CIF parseriai nesugeba apdoroti didelės apimties CIF duomenų srautų arba tai daro neefektyviai, bandydami išskleisti visą dokumento medį operatyviojoje atmintyje, taip užkirsdami kelią *shell* konvejerių grandinės procesų vienalaikiui ir lygiagrečiam vykdymui. Mes sukūrėme srautinį CIF parserį, kuris panaudodamas Ada kalbos vienalaikes užduotis leidžia ne tik apdoroti didelius duomenų srautus su itin mažu rezidentinės aibės dydžiu, bet ir per dalinį duomenų apdorojimą nesustabdo konvejerio proceso vykdymo. Sukurtas sintaksinis analizatorius panaudoja esamą *cod-tools* įrankių rinkinį ir jame naudojamą nesrautinį parserį ir naudojant Ada kalbos teikiamas priemones apdoroja jam pateiktus duomenis po vieną duomenų bloką. Darbe pirmiausia apžvelgiame esamą kontekstą ir literatūrą. Tuomet, aprašome suplanuotą parserio architektūrą bei principinius techninius sprendimus. Galiausiai apibūdiname įgyvendinimo proceso organizaciją ir eigą bei rezultatus.

Summary

Existing CIF parsers are not able to process large sets of CIF data or are doing it inefficiently by attempting to unfold the full DOM tree in operating memory. This hinders the executing of piped shell processes both concurrently and in parallel. We created a streaming CIF parser that utilizes concurrent tasks provided by Ada programming language and not only allows processing of large data sets with a small resident set, but also enables concurrency and parallelism in piped processes through partial data processing. Created streaming CIF parser uses existing *cod-tools* tool set including its non-streaming parser and processes data by single data block by using tools provided by Ada programming language. In this work we first describe the context of the research, then move on to the planned architecture of the parser and principal technical choices. Finally we describe the organisation of the development process, its course and results.

Turinys

1. Įvadas

1.1. Darbo aktualumas

Vienas iš pamatinių šiuolaikinio mokslo aspektų yra gebėjimas struktūruotai aprašyti duomenis. Pažanga komunikacijos technologijose atvėrė naujus mokslininkų bendradarbiavimo horizontus, o tai, savo ruožtu, sudarė poreikį turėti ne tik struktūruotus duomenis, bet ir bendrus duomenų struktūravimo standartus, idant būtų įmanoma keistis šiais duomenimis. Kristalografijos mokslas šią problemą išsprendė sukurdamas standartizuotą kristalografinės informacijos failų (angl. *crystallographic information file*, toliau – CIF) [HAB91] formatą. Griežta sintaksė, formaliai aprašyta gramatika ir žodynuose aprašyta semantika leido verifikuoti kristalų aprašymus, užkertant kelią klaidoms, kurios pasitaikydavo šią informaciją perteikiant kitais formatais, pvz., lentelėmis. Minėtosios savybės ir ilgainiui dėl sudėtingėjančių kristalų struktūrų išaugusios CIF failų apimtys sukėlė poreikį atsirasti mašininiams sintaksiniams analizatoriams.

Duomenys Vilniaus universiteto prižiūrimoje, vystomoje ir visame pasaulyje aktyviai naudojamoje atviroje kristalografinėje duomenų bazėje COD (angl. *Crystallography Open Database*) yra saugomi minėtų CIF failų formatu [GCD+09; GDM+12]. Dėl sparčiai augančios duomenų bazėje esančių duomenų apimties tapo sudėtinga naudotis CIF failus apdoroti leidžiančia esama *cod-tools* sistema, kuri naudoja tradicinį, t. y. bandantį išskleisti visą dokumento medį operatyviojoje atmintyje, sintaksinį analizatorių.

Tokia prieiga neišnaudoja „Unix“ pagrindu sukurtų operacinių sistemų architektūrinių pranašumų. Šiose sistemose įvesties ir išvesties elementai (pvz.: failai, katalogai ir kt.) yra baitų srautai, pasiekiami per failų deskriptorius. Tai galioja ir vienam iš itin paplitusių ir reikšmingų šių sistemų funkcionalumų – konvejeriui (angl. *piping*) [RT74]. Būtent dėl šios ypatybės konvejerio funkcionalumas leidžia naudotojui sudaryti komandų grandines, nukreipiant vienos komandos (t. y. jos proceso) standartinę išvestį į po jos grandinėje einančios komandos standartinę įvestį.

Modernūs procesoriai turi daugiau negu vieną branduolį, todėl užduotys gali būti išskaidomos ir atliekamos lygiagrečiai, taip pagreitinant jų išpildymą [UDR+16, p. 343–344]. Operacinei sistemai kuriant konvejerį, visai procesų grandinei yra paskiriamas nedidelis atminties kiekis – buferis – kurį procesai naudoja serializuoto tekstinio duomenų srauto perdavimui. Šis buferis naudojamas tiek įvesties, tiek išvesties operacijoms. Gautąjį duomenų srautą procesai išskleidžia savo vidinėje atmintyje. Konvejeriai, dėl minėtojo jiems paskirto buferio, yra itin tinkami lygiagretinimui. Juose sujungti procesai geba veikti lygiagrečiai, kadangi jie neturi bendrų duomenų, o duomenų srauto perdavimas procesams gali vykti nenutrūkstamai. Pati konvejerio grandinė taip pat gali būti dar labiau išlygiagretinama, panaudojant vien numatytąją *shell* aplinką [HKV+21; VKM+21]. Tačiau, jei konvejerio jungiamoje procesų grandinėje atsiranda procesas, kuris gali operuoti tik pilnu duomenų srautu (pvz., nesrautinis sintaksinis analizatorius) ir tas srautas yra didelis (tarkime, dokumento medžio konstravimas), tai duomenų išskleidimas bus vykdomas operatyviojoje atmintyje ir jos pritrūkus bus iššaukiamas duomenų iškėlimas į ilgalaikę atmintį (angl. *swapping*). Be to, kol toks procesas nuskaito ir apdoroja visą jam teikiamų duomenų srautą, kiti, toliau grandinėje esantys procesai, sustoja, nes negauna jokio duomenų srauto, su kuriuo galėtų dirbti. Vadinasi, srautinis

duomenų apdorojimas yra itin svarbus, siekiant išlaikyti imanentinį konvejerių lygiagretumą bei gebėti išskirstyti procesų išpildymą į skirtingus procesoriaus branduolius.

Sumažintas perduodamų duomenų kiekis leistų išlaikyti „locality of reference“ principą, kadangi mažas duomenų kiekis galėtų būti kešuojamas. Kadangi kešas atminties hierarchijoje yra aukštesnėje vietoje, tai reiškia, kad proceso našumas žymiai išaugtų, nes procesoriui duomenys būtų pateikiami žymiai greičiau nei iš sisteminės atminties [UDR+16, p. 72–73].

Tiek „Unix“ operacinė sistema, tiek konvejeriai nuo pat jų atsiradimo iki šių dienų yra naudojami srityse, reikalaujančiose efektyvaus darbo su dideliais duomenų kiekiais. *Boeing* aviacijos kompanija pasitelkė konvejerius komunikacijai su jos geometrijos ir vizualizacijos sistema – „AGPS“ [Dic92]. Nacionalinė aeronautikos ir kosmoso administracija (angl. *National Aeronautics and Space Administration*, toliau – NASA) kurdama kosminio skrydžio sekimo ir telemetrijos dvimačio ir trimačio vizualizavimo sistemą, naudojo konvejerius komunikacijai tarp subprocesų [Jel89]. Taipogi, kurdama trimates animacijas iš surinktų didelių duomenų kiekių, NASA naudoja tiek srautinę duomenų apdorojimo prieigą, tiek ir pačius konvejerius – informacija į *renderer'į* perduodama pakadriui per konvejerį [ES19]. Jungtinių Amerikos Valstijų karinių jūrų pajėgų antrosios pakopos mokyklos studentų vystytoje autonominio povandeninio laivo sistemoje tarpprocesinė komunikacija taip pat buvo įgyvendinta panaudojant „Unix“ konvejerius [BHM+96].

Poreikis efektyviai apdoroti didelius duomenų srautus itin aktualus ir mikrovaldiklių srityje. Dėl gero efektyvumo ir sąnaudų santykio, jie dažnai naudojami kaip duomenų agregatoriai, surenkantys duomenis iš sensorių. Neretai kartu atliekamas ir pradinis duomenų apdorojimas prieš persiunčiant duomenis toliau. Kadangi mikrovaldiklių resursai yra itin riboti, o duomenų srautas gali būti ne tik didelės apimties, bet ir nenutrūkstamas, kyla poreikis gebėti apdoroti duomenis efektyviai ir be trikdžių.

Kadangi CIF protokolo failai naudojami Vilniaus universiteto mokslinėje veikloje bei vis dar neturi jiems skirto srautinio sintaksinio analizatoriaus, nusprendėme dirbti su šios struktūros failais. Be to, vienas iš didžiausių darbo su CIF protokolu aktualumų yra tai, kad yra prieinamas itin didelis šios struktūros duomenų kiekis. CIF failai itin lengvai jungiasi tarpusavyje (angl. *concatenation*), todėl nesudėtingai galima sukurti dar didesnio dydžio failus. Mes darome prielaidą, kad itin didelių struktūruotų failų apdorojimo ir panaudojimo „Unix“ operacinių sistemų konvejeriuose problematika galėtų būti išspręsta pritaikant srautinį informacijos iš failų nuskaitymą. Srautinių analizatorių paklausumą įrodo tai, kad plačiai naudojami struktūruotų failų formatai – „XML“ ir „JSON“ – jau turi jiems sukurtų srautinių tekstinių analizatorių [Dol18; Maz13].

Esamų parserių architektūros dažnai yra paremtos objektinio programavimo paradigma bei atgalinėmis (angl. *callback*) funkcijomis. Abi šios architektūrinės savybės sukelia klaidų ir nestabilaus funkcionavimo pavojų. Objektinio programavimo atveju, kai yra taikomas klasių paveldėjimas, tai įvardijama kaip trapios bazinės klasės problema. Nors yra mokslininkų, kvestionuojančių šios problemos poveikį realiame gyvenime [SGA+17], vis tik egzistuoja galimybė su ja susidurti, kai objektinio programavimo paradigmoje vyksta paveldėjimas tarp didelio klasių skaičiaus ir/arba yra paveldimos svetimos klasės bei jų funkcionalumas yra modifikuojamas arba išplečiamas paveldėjusiose klasėse [Bud98; SGA+17]. Esant tokioms sąlygoms atsiranda pavojus, jog kažku-

riuo momentu bazinė klasė iškvies modifikuotuosius metodus, taip sukeldama nenumatytą elgesį. Atgalinių funkcijų atveju taip pat kyla nenumatytos elgsenos pavojus. Naudodami atgalines funkcijas mes pateikiame jas kitiems metodams kaip argumentus idant jos būtų iškviestos kažkuriuo momentu priimančiajame metode. Be to, priimantysis metodas pats gali būti paverstas atgaline funkcija, taip sudarant sudėtingas atgalinio kvietimo grandines – „*callback'ų* pragarą“ [GMB15]. Tokia architektūra yra sunkiai suvaldoma, iškyla pavojus, jog atgalinės funkcijos bus iškviestos nenumatytose vietose, taip sukeliant neprognozuoto funkcionalumo pavojų. Be to, tokia architektūra yra sunkiai modifikuojama bei palaikoma, itin sudėtingas tampa klaidų nustatymas ir jų taisymas.

Siekdami išvengti aukščiau aptartų problemų, kylančių naudojant objektinio programavimo paradigmą, mūsų planuojamam parseriui nusprendėme įgyvendinti architektūrą, derinančią tiek objektinio, tiek funkcinio programavimo paradigmas. Vietoje viso dokumento medžio gamybos operatyviojoje atmintyje, operacijos bus vykdomos su ribotu duomenų kiekiu, kuris bus atvaizduojamas programoje objektu, laikančiu tik vieno duomenų bloko duomenis iš kurių bus gaminamas mažesnis dokumento medis, taip apsisaugant nuo jau aptarto duomenų srauto iškėlimo į ilgalaikę atmintį. Viena iš galimybių būtų panaudoti pratęsimo perdavimo programavimo stilių (angl. *continuation-passing style*). Tai reiškia, kad kartu su funkcijos pabaigoje grąžinama reikšme, bus paduodamas ir tęsinys – šiuo atveju funkcijos forma – kurį galima kviesti, norint gauti kitą duomenų bloką, o funkcijos būsena yra saugoma parserio objekte. Be to, pabaigus darbą su šiuo duomenų bloku, operatyvinė atmintis yra atlaisvinama tolimesniam darbui. Tokia prieiga, skaidant informaciją į blokus, leidžia iš esmės apdoroti neribotus informacijos kiekius, su itin mažais atminties kiekiais. Šią architektūrą sąlygoja ir egzistuojanti *cod-tools* sistemos architektūra – dabartinėje sistemoje visi CIF failo duomenų objektai yra grąžinami kaip struktūrų masyvas, per kurį yra iteruojama ir kiekviena struktūra yra apdorojama išoriniame cikle, panaudojant jau egzistuojančias bibliotekas. Tokiai architektūrai dėkingesnis mūsų pasirinktas reenterabilios funkcijos modelis, kadangi, priešingai nei naudojant atgalines funkcijas, mes vis dar galime naudoti esamas *cod-tools* bibliotekas. Taip yra todėl, kad keičiamas tik išorinis ciklas, t. y. vietoj iteravimo per masyvą, kurio visos struktūros yra atmintyje, iteruojame per failo srautą, kviesdami pratęsimo funkciją ir kiekvieną kartą gaudami vienos struktūros aprašymą.

1.2. Darbo tikslai

Šio darbo tikslas yra nustatyti, kiek srautinis teksto analizatorius leis padidinti pralaidumą ir greitaveiką bei sumažins delsą ir rezidentinės aibės dydį. Remdamiesi šia informacija patobulinsime esamą *cod-tools* apdorojimo sistemą, naudojamą *Crystallography Open Database* terpėje.

1.3. Darbo uždaviniai

1. Apžvelgti konvejerių funkcionalumą ir jų naudą apdorojant didelius duomenų kiekius.
2. Pasiūlyti operacijų lygiagretinimo būdą, apdorojant *Crystallogaphy Open Database* duomenų bazę, panaudojant konvejerius.

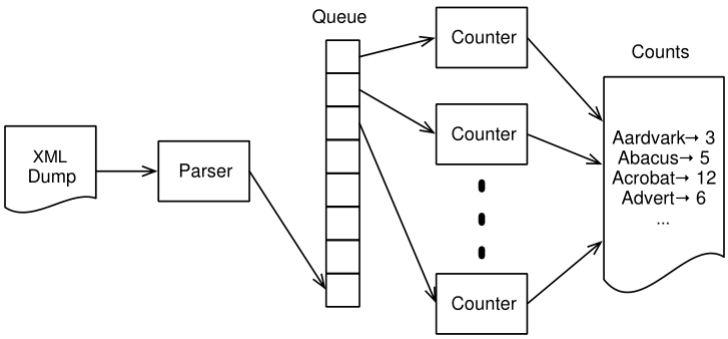
3. Apžvelgti egzistuojančius parserius.
4. Sukurti srautinį sintaksinį analizatorių, gebantį apdoroti neriboto dydžio CIF struktūros srautus ribotoje operatyviojoje atmintyje.
5. Pasitelkiant įrankius, leidžiančius apriboti rezidentinės aibės ir virtualios atminties dydžius (pvz., *ulimit* komanda Linux operacinėje sistemoje), palyginti gauto parserio bei egzistuojančių parserių našumą, greitaveiką ir delsą.
6. Palyginti gauto parserio ir egzistuojančių parserių rezidentinių aibių dydžius.

2. Literatūros apžvalga

2.1. Įvadas

Svarstant galimus srautinio sintaksinio analizatoriaus įgyvendinimo būdus, išsiskiria keli šablonai (angl. *patterns*), atitinkantys planuojamo sintaksinio analizatoriaus architektūrinius reikalavimus.

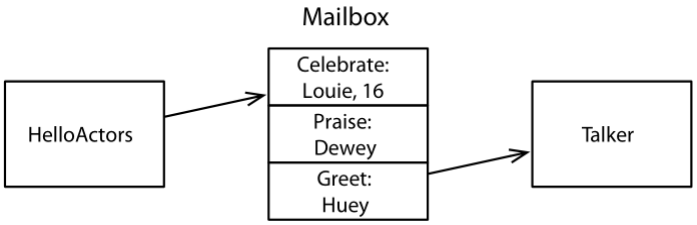
Itin tinkantis atrodo „gamintojo-vartotojo“ (angl. *producer-consumer*) modelis (1 pav.). Naudojant šį modelį vietoj vienos gijos, atliekančios tiek sintaksinę analizę, tiek tolimesnius veiksmus su išanalizuotomis pradinių duomenų dalimis, atskiriame šias atsakomybes: viena gija – gamintojas – atlieka sintaksinę analizę ir gautus rezultatus deda į eilę, o kita gija – vartotojas – ima po reikšmę iš eilės ir ją apdoroja [But14, p. 36]. Šis procesas yra vykdomas ligi tol, kol pasiekiama pradinių duomenų pabaiga. Eilę, į kurią rezultatus deda gamintojas ir iš kurios blokus ima vartotojas, galima apriboti, taip užtikrinant sklandų abiejų gijų darbą. Vienas iš didesnių šio šablono trūkumų yra tas, kad net esant daugiau nei vienai gamintojų ar vartotojų gijai, jos yra statiškai sukuriamos proceso pradžioje. Tai padaro sistemą nelanksčia, t. y. sistema nesugebėtų padidinti vienos ar kitos pusės pajėgumų, priklausomai nuo rezultatų.



1 pav. „Gamintojo-vartotojo“ modelis [But14, p. 36]

Dėl šios priežasties patrauklesnis atrodo „pašto dėžutės“ (angl. *mailbox*) šablonas (2 pav.), besiremiantis vadinamuoju aktorių programavimu (angl. *actor programming*) [But14, p. 115]. Šis šablonas panaudoja panašius į „gamintojo-vartotojo“ šablono principus: bent viena gija apdoroja pradinius duomenis ir deda juos į eilę – „pašto dėžutę“ – iš kurios, bent viena skaitanti gija – „aktorius“ – ima duomenų blokus ir juos apdoroja paeiliui. Vienas iš esminių skirtumų tarp „gamintojo-vartotojo“ šablono ir „pašto dėžutės“ šablono yra tai, kad „pašto dėžutės“ šablono skaitančios gijos saugo savo būseną, kuri prieinama tik nuskaitymui [But14, p.120], taip sukuriant sąlygas bendrai sistemos koordinacijai ir reaktyvumui vykdymo metu (angl. *runtime*). „Pašto dėžutės“ skaitančių gijų objektai yra itin dinamiški – atsiradus poreikiui galima susikurti (o tai reiškia kartu ir sunaikinti) papildomų gijų. Sujungiant su jau prieš tai minėtuoju būsenos saugojimu, sistema galėtų koordinuotai sunaikinti gijas, kurios yra rimties būsenoje, nes pradinius duomenis analizuojanti gija ar gijos nespėja pakankamai greitai dėti duomenų blokų į eilę. Be abejo, tai galėtų būti ir rodiklis, kad reikia papildyti analizatorių gijas objektais, idant paspartėtų eilės užpildymas. Be to, „pašto dėžutės“ eilėje patalpinti blokai neša ne tik pradinių duomenų dalį, bet ir nurodymą, ką su

tais duomenimis daryti, todėl skaitančios gijos, kuriamos pagal poreikį ir informaciją eilės bloke, gali būti skirtingų tipų.



2 pav. „Pašto dėžutės“ modelis [But14, p. 118]

Aprašytųjų modelių panaudojimas reikalauja vienalaikiško (angl. *concurrent*) užduočių vykdymo. Tiek C programavimo kalba, tiek alternatyvios jai kalbos, tokios kaip C++ ar Rust, neturi kalbos lygmenyje įgyvendinto vienalaikiškumo – tai yra pasiekiama tik per lygiagrečias gijas ir rankinį jų valdymą. Todėl svarstydami technologijas, kuriomis įgyvendinsime srautinį sintaksinį analizatorių kaip vieną iš pagrindinių reikalavimų kėlėme vienalaikiškumą kalbos lygmenyje.

2.2. Technologijos

Kartu su jau minėtuoju vienalaikiškumu kalbos lygmenyje nustatėme papildomus reikalavimus, kuriuos turėtų atitikti technologijos-kandidatės. Šie reikalavimai, mūsų manymu, yra būtini siekiant sėkmingo sintaksinio analizatoriaus išpildymo:

1. Lygiagretumas (angl. *parallelism*) ir vienalaikiškumas (angl. *concurrency*) kalbos lygmenyje.
2. Galimybė panaudoti esamas C kalba parašytas bibliotekas (*cod-tools*).
3. Programavimo kalbos greitaveika – kompiliuojama, o ne interpretuojama.
4. Atviro kodo įrankių grandinė.
5. Kodo perkeliamumas.

Be šių privalomų reikalavimų nustatėme papildomus neprivalomus reikalavimus, kurie nėra kritiškai svarbūs srautinio analizatoriaus įgyvendinimui, tačiau, manome, kad jį pagerintų:

1. Galimybė verifikuoti sistemos modelį ir/arba invariantus.
2. Kuo mažesni papildomi naudojimo poreikiai (angl. *overhead*).

Nors C programavimo kalba iš pirmo žvilgsnio atrodo patraukli, nes ja yra parašytas *cod-tools* įrankių rinkinys, ji yra itin netinkama vienalaikiškumo atžvilgiu. Joje, kaip ir C++ ar Rust, vienalaikiškumas įgyvendinamas per rankiniu būdu kontroliuojamas gijas. Tokia prieiga tinkamesnė lygiagretinimui, o ne vienalaikiškumui. Be to, dėl rankinės gijų kontrolės iškyla klaidų pavojus. Dėl šių priežasčių atmetėme C, C++ ir Rust kalbas ir ėmėmės ieškoti kalbų su stipriu vienalaikiškumo įgyvendinimu. Kaip dvi realiausios kandidatės buvo pasirinktos C# ir Ada.

C# – tai „Microsoft“ kompanijos sukurta, palaikoma ir vystoma kalba. Jos autorius – Anders Hejlsberg, prieš tai dirbęs su TurboPascal ir Delphi. Ši kalba buvo pristatyta 2000-aisiais metais ir standartizuota tarptautiniu ECMA standartu 2002-aisias metais. Kalba yra multiparadigminė, statiškos griežtos tipizacijos, naudojanti objektinio programavimo paradigmą [ECM17]. C# kalba savo sintakse ir principais turi itin daug bendro su C, C++ ir Java. C# pasižymi automatiniu „šiukšlių surinkėju“ (angl. *garbage collector*), automatiškai atlaisvinančiu atmintį nuo nepasiekiamo ar nebenaudojamo kodo. Svarbu paminėti, kad ši kalba kompiliuojama į tarpinę kalbą – *Common Intermediate Language* (toliau – CIL), kuri vykdymo metu yra įkeliama į *Common Language Runtime* (toliau – CLR) terpę. Joje *Just-in-time* (toliau – JIT) kompiliatorius paverčia CIL į mašininį kodą.

Ada – programavimo kalba, kurta pagal Jungtinių Amerikos Valstijų gynybos departamento kontraktą. Jos kūrimui vadovavo Jean Ichbiah. Viešai pristatyta ši kalba buvo 1980-aisiais, standartizuota 1987-aisiais metais. 1995 m. standartizuota Ada 95 revizija buvo pirmoji ISO standartu standartizuota objektinio programavimo paradigmos kalba. Kalba ligi šiol yra vystoma, tačiau konservatyviai, siekiant išsaugoti kodo suderinamumą. Šiuo metu yra ruošiama nauja kalbos revizija [AXE21]. Ada yra statiškos itin griežtos tipizacijos bei palaikanti objektinio programavimo paradigmą [Int12; SGH92]. Ši programavimo kalba skirta ilgalaikių, tikralaikių (angl. *real-time*), itin patikimų sistemų, kuriose klaidos galėtų sukelti tragiškas pasekmes [Int12, p. 677], kūrimui. Todėl ją itin plačiai naudoja JAV gynybos departamentas – net 30% kritinių karybos programinės įrangos komponentų (apie 50 mln. eilučių kodo) buvo parašyta Ada [Nat97, p. 37]. Priešingai nei C#, Ada neturi automatinio atminties atlaisvinimo, nes tai galėtų sukelti nenumatytą funkcionalumą tikralaikėse sistemose. Jei nėra iškviečiamas objekto finalizavimas per pateiktą procedūrą, pavadinimu „Unchecked_Deallocation“, objektas gyvuos ligi pagrindinė procedūra baigs vykdymą [Int12, p. 173].

Tiek Ada, tiek C# pasižymi stipriu vienalaikiško programavimo palaikymu. Abejose programavimo kalbose egzistuoja „užduočių“ (angl. *task*) tipai, skirti asinchroniniam loginių gijų vykdymui [ECM17, p. 171; Int12, p. 189]. Svarbu pabrėžti, kad C# kalboje sukurtos užduotys yra vykdomos iškart, nebent yra pavėlinamos (angl. *delay*) arba susiejamos su kita logine gija, idant būtų išprovokuojamas šių užduočių paleidimas, kai tuo tarpu Ada reikalauja, jog po užduoties deklaracijos ji būtų aktyvuota vėlesniu metu.

2.2.1. Lygiagretumas ir vienalaikiškumas

Vienalaikės programos pasižymi keliomis loginės kontrolės gijomis [But14, p. 1; BW07, p. 15; LKN97]. Tačiau vienalaikis užduočių vykdymas nebūtinai reiškia, kad jos bus vykdomos lygiagrečiai ir *vice versa*. Vienalaikiškumas yra kelių vienu metu vykstančių įvykių suvaldymas, o lygiagretumas – tai skirtingų programos dalių vykdymas lygiagrečiai [But14, p. 2]. Tai reiškia, kad vienalaikiai įvykiai nebūtinai vykdomi nenutrūkstamai, kadangi vykdymo metu (angl. *runtime*) kurio nors vieno įvykio vykdymas gali būti pristabdomas, idant būtų progresuojama su kito įvykio vykdymu. Nepaisant to, tai yra vienalaikės loginės kontrolės gijos.

Vienalaikiškumas svarbus siekiant išlaikyti programą reaktyvią (angl. *responsive*) [But14, p.

6], t. y. kad nebūtų užrakinta pagrindinė aplikacijos gija. Tai yra aktualu planuojamo srautinio CIF sintaksinio analizatoriaus kontekste, kadangi analizatorius, esantis konvejerio komandų grandinės viduryje, turi gebėti priimti standartinę įvestį, pateikiamą jam iš anksčiau grandinėje einančios komandos. Vadinasi, siekiant aplikacijos lygiagretumo *shell* lygmenyje, analizatorius privalo gebėti apdoroti dalinį įvesties srautą, kad po jo konvejerio grandinėje einantys procesai neturėtų laukti įvesties srauto.

Vienalaikiškumo kontekste iš minėtųjų kalbų kandidačių išsiskiria C# ir Ada. Jei C, C++ ir Rust vienalaikiškumą įgyvendina per gijas, tai tiek Ada, tiek C# turi vienalaikes gijų abstrakcijas, kurias vadina užduotimis (angl. *task*). Abiem atvejais pagrindinė šių abstrakcijų paskirtis – asinchroninis (tiek pagrindinės, tiek kitų užduočių atžvilgiu), o tai reiškia kartu ir vienalaikiškas, procesas. Tačiau Ada šiuo atžvilgiu vis tik yra pranašesnė, kadangi užduotys joje yra įgyvendintos ne bibliotekos, o kalbos lygmenyje [LKN97], priešingai nei C#.

Lygiagretumo įgyvendinimui įtakos turi ne tik programavimo kalba, bet ir apdorojamų duomenų struktūra. Priešingai nei CIF duomenų struktūra, nei XML, nei JSON neturi standartizuoto srautinio formato. Bandant įgyvendinti sintaksinės analizės proceso lygiagretumą programos lygmenyje XML duomenų struktūrai, iškyla duomenų lygiagretumo (angl. *data parallelism*) problema. Kadangi visos gijos naudotųsi tuo pačiu vientisu duomenų šaltiniu, iškyla poreikis nustatyti, kurie duomenys jau buvo apdirbti. Tai reiškia, jog programa turi gebėti paskirstyti duomenis dalimis gijoms. Vienas iš šios problemos sprendimo būdų yra preliminari sintaksinė analizė (angl. *pre-parsing*). Tai yra procedūra, kurios metu sukonstruojamas struktūruotų duomenų „skeletas“ – loginis medis [YW11; LCP06]. Nustačius atskirų teksto dalių skaičių, tolimesnė sintaksinė analizė būtų perduota atskiroms gijoms. Kadangi tai yra papildoma procedūra, nebūtina standartinei sintaksinei analizei ir kuri privalo būti nuosekli (angl. *sequential*), vadinasi preliminari analizė turi būti atliekama minimaliomis sąnaudomis, pvz., netikrinant dokumento sintaksės. Minimizuota preliminari analizė (pvz., XML struktūros atveju išlaikanti vienintelį reikalavimą, kad atidarančioji žymė privalo turėti uždarančiąją žymę) bus įvykdymo žymiai greičiau, nei pilno dokumento medžio konstravimas pilnos sintaksinės analizės metu [LCP06].

Po tokios preliminarios analizės, prasidėjus pagrindiniam sintaksinės analizės procesui, yra sukuriamas numatytas gijų skaičius ir vienai iš jų yra paskiriama šakninė viršūnė (angl. *root node*), o likusios gijos lieka rimties būsenoje (angl. *idle*). Rimties būsenoje esančios gijos prideda savo užklausą į užklausų eilę [LCP06], taip pranešdamos, kad yra pasiruošusios darbui. Bet kuri aktyvi gija vykdo įprastinę analizę, iki susiduria su pradine žyme. Tuo atveju aktyvi gija patikrina užklausų eilę, ar joje yra gijų, laukiančių užduoties. Jei taip, laukiančiajai gijai paskiriama viena seserinė viršūnė (jei tokia egzistuoja) ir laukusioji gija tampa aktyvia. Rekursyviai kartojant šį procesą, sintaksinė analizė yra išlygiagretinama.

Pirminės analizės kritikai akcentuoja papildomus naudojimo poreikius, kurie mažina efektyvumą, ir nenumato kaip tvarkytis su XML struktūroje esančiais išskirtiniais (angl. *exception*) elementais, pvz., komentarais [YW11]. Komentaro viduje gali būti tiek pradžios, tiek pabaigos žymių, kurios turėtų būti ignoruojamos, tačiau pirminės analizės metu bus laikomos įprastomis žymėmis. Pašalinus pirminės analizės procesą duomenimis grįstas lygiagretumas tampa nebeįmanomas, tad

kaip alternatyvą autoriai siūlo tiesioginį lygiagretinimą. Visų pirma visoms gijoms yra nurodomas poslinkis dokumente, taip suskirstant visą dokumentą į dalis. Kiekviena gija tuomet pradeda sintaksinę analizę nuo pirmo sutikto skirtuko (angl. *delimiter*). XML struktūros atveju tai yra simbolis „<“. Jei analizės metu gija susiduria su „užuomina“ (angl. *clue*), pvz., simboliais „-->“, XML struktūroje žyminčiais komentaro pabaigą, tuomet galima įtarti jog analizės pradžios skirtukas nebuvo viršūnės pradžia [YW11]. Tokiu atveju yra aktyvuojama reanalizės (angl. *reparsing*) procedūra, kuri iki trijų kartų kartoja analizę, bandydama nustatyti, ar pradinis skirtukas yra išimtinio elemento sferoje.

Komandos, sujungtos tarpusavyje į vientisą *shell* konvejerį, komunikuoja ženklų srautu. Individualūs šio srauto elementai yra atskirti simboliu LF (*line feed*), žyminčiu esamos eilutės pabaigą ir naujos eilutės pradžią. Paties srauto pabaigą žymi failo pabaigos simbolis EOF (*end-of-file*). Segmentuotas srauto pobūdis sudaro sąlygas konvejeriui apdoroti duomenis skirtingose jo dalyse vienalaikiškai, o UNIX branduolys (angl. *kernel*) planuoja, komunikuoja ir sinchronizuoja minėtąsias dalis [VKM+21].

Pagrindinis vienalaikio užduočių vykdymo tikslas, priešingai nei lygiagrečių procesų, nėra efektyvumas sunaudojamų kompiuterio resursų atžvilgiu, bet užtikrinti, kad sistema veikia korektiškai ir valdomai. Tai reiškia, jog vienalaikės sistemos gali būti įgyvendintos tiek lygiagrečiose, tiek sekvencinėse platformose [AP11].

Vadinasi, siekdami užtikrinti mūsų planuojamo sintaksinio analizatoriaus, kaip UNIX konvejerio grandies, gebėjimą būti išlygiagretintam, privalome užtikrinti vienalaikiškumą programos lygmenyje.

2.2.2. Galimybė panaudoti esamas C kalba parašytas bibliotekas (*cod-tools*)

Jau turimas *cod-tools* įrankių rinkinys kelia poreikį naujajam sintaksiniam analizatoriui gebėti naudotis jau esamu C kalbos kodu. Vadinasi, įgyvendinimo kalba turi turėti API – *application programming interface* – sąsają, galinčią iškviesti esamas C kalba parašytas funkcijas.

Ada turi interfeisus kelioms programavimo kalboms, įskaitant C. Tam yra skirtas „Interfaces.C“ modulis, savyje talpinantis bazinius tipus, konstantas ir subprogramas, leidžiančias perduoti skaitines ir eilučių reikšmes C ar C++ kalba parašytoms funkcijoms [Int12, p. 537].

Šiuo atžvilgiu, kur kas sudėtingesnė situacija su C# programavimo kalba. Kaip jau minėjome anksčiau, C# išeities kodą kompiliuoja į tarpinę kalbą – CIL, kurią vykdymo metu apdoroja JIT kompiliatorius. Tokiu būdu sugeneruojamos dinamiškos bibliotekos. Tai yra nesuderinama su statiškai kompiliuotu C kodu, tad norėdami kviesti C kodo funkcijas iš C# kodo, privalome specialiai perkompiliuoti į dinamines bibliotekas. Tam atlikti galima pasitelkti Microsoft integruotos kūrimo aplinkos „Visual Studio“ plėtinį „C++/CLI“, skirtą atlikti jungiamosios grandies tarp C++ ir C# vaidmenį, bet kartu panaudojamą ir C kalbai. Tačiau, tai reikštų, jog kodo vystymas būtų susietas su „Visual Studio“, o tai neatitiktų tiek kodo perkeliamumo, tiek atviro kodo įrankių grandinės reikalavimų.

2.2.3. Atviro kodo įrankių grandinė

Viena iš JAV gynybos departamento gairių, renkantis programavimo kalbą yra ta, jog pirmenybė teikiama standartizuotoms programavimo kalboms be nuosavybės teisės (angl. *non-proprietary*), kadangi tai padidina tiek kodo, tiek programuotojų portabilumą ir sumenkina „prisirakinimo“ prie vieno šaltinio tikimybę [Nat97, p. 55]. Planuojamam sintaksiniui analizatoriui iškėlėme reikalavimą, jog naudojama įrankių grandinė būtų atviro kodo. Tai leistų užtikrinti ilgalaikį sistemos naudojimą bei nepriklausomumą nuo išorinių komercinių ar kitokių sprendimų.

C# kompiliavimui naudoja .NET kompiliatorių platformą (angl. *.NET Compiler Platform*), dar žinomą kaip „Roslyn“. Šie įrankiai yra atviro kodo, tačiau jų vystymas yra koordinuojamas vienos įmonės – „Microsoft“. Ada taip pat turi atviro kodo kompiliatorių – „GNAT (GNU Ada)“, kuris yra GCC kompiliatorių sistemos dalis, o tai reiškia, kad yra palaikomas „Free Software Foundation, Inc.“. Šios aplinkybės rodo, jog Ada, šiuo atveju, būtų patrauklesnė technologija.

Verta paminėti, kad iki 2014–ųjų metų, kai „Microsoft“ atvėrė tiek .NET platformos, tiek „Roslyn“ kompiliatoriaus išeities kodą ir ėmė labiau koncentruotis į tarpplatforminį suderinamumą, didelį pasisekimą turėjo atviro kodo kompiliatorius „Mono“, atitinkantis ECMA C# standartą ir leidęs kompiliuoti „.NET Framework“ platformos, suderinamos tik su „Microsoft Windows“ operacine sistema, kodą į vykdomuosius Linux failus. Tačiau kaip jau minėjome, „Microsoft“ ėmus palaikyti tarpplatforminį kompiliavimą, bei „Mono“ tapus „.NET Foundation“ dalimi ir „Microsoft“ panaudojus dalį „Mono“ kompiliatoriaus savybių „Roslyn“ kompiliatoriuje, pirmieji ėmė daryti įtaką tolimesniam „Mono“ kompiliatoriaus vystymui, tad pastarasis ėmė koncentruotis į nešiojamuosius įrenginius.

2.2.4. Sistemos modelio ir/arba invariantų verifikacija

Ankstyvosioms C# versijoms 2004-aisiais metais (1.0, 2.0) „Microsoft“ mokslininkai buvo sukūrę sistemų modelių ir invariantų verifikacijos sistemą „Spec#“. Ji susidėjo iš trijų dalių: C# kalbos dialekto, atskiro kompiliatoriaus, integruoto į integruotą kūrimo aplinką „Visual Studio“, ir statiško programų verifikatoriaus. Tačiau nuo 2005-ųjų metų šis projektas buvo apleistas, tad, turint omenyje smarkius C# kalbos pokyčius per pastarąsias 8-as versijas (šiuo metu naujausia yra 10.0 versija), net ir atradus išeities kodą, šios sistemos panaudojimas verifikacijai yra praktiškai neįmanomas.

Kaip jau minėjome anksčiau, Ada programavimo kalba skirta kritinių sistemų kūrimui, kuriose bet kokia klaida gali turėti itin reikšmingas pasekmes, tad tokių sistemų verfikavimas yra būtinas. Tam tikslui yra skirta SPARK sistema. Ši sistema sukuria galimybę statiškai verifikuoti tiek sekvencines, tiek vienalaikiškas programas [HM03]. SPARK naudoja apribotą Ada kalbos dialektą statinei analizei, bet kartu turi ir įrankių, kuriais galima statiškai įrodyti, jog, pvz., programoje nėra vykdymo laiko klaidų ar kad programa atitinka specifikacijas, kurias nurodė programuotojas per pridėtas pirmines ir galutines sąlygas [BC17]. SPARK palaiko ne tik formalią verifikaciją, bet ir testavimo metodus ir įrankius. Tai reiškia, jog naudojantis ta pačia sistema galima tiek verifikuoti, tiek atlikti, pvz., „unit“ testus.

Bene didžiausias SPARK privalumas yra tas, jog šiuo dialektu parašytas kodas yra supran-

tamas standartiniams Ada kompiliatoriams. Vadinasi, ta pati kodo bazė gali būti naudojama tiek verifikacijai, tiek testavimui, tiek vykdymui. Ši savybė vadinama vykdomaisiais kontraktais (angl. *executable contracts*). Kita vertus, naudojant SPARK sistemą, vietoje klasikinės Ada kalbos, gali nukentėti kodo skaitomumas. Itin svarbu pabrėžti, jog SPARK dialektas yra Ada poaibis, o tai reiškia, jog ne visos Ada kalbos savybės ir konstrukcijos yra prieinamos SPARK sistemoje.

Kartu dėl Ada ir SPARK architektūrinių savybių, daug silpnybių, tipiškų kitoms programavimo kalboms, nėra įmanomos. Pavyzdžiui, kadangi SPARK disciplinuoja krūvos (angl. *heap*) atminties panaudojimą bei dinamišką atminties išskyrimą, dėl to išvengiama krūvos perpildymo (angl. *heap overflow*) arba atminties nutekėjimo (angl. *memory leak*) [TJB+11].

SPARK sistema neretai naudojama kaip efektyvus būdas su minimaliais kaštais užtikrinti reikalavimų atitikimą kompleksiškose sistemose. Pvz., karinio orlaivio Lockheed C130J „Super Hercules“ misijų kompiuteryje (angl. *mission computer*), didelė dalis – apie 80% – programinės įrangos parašyta SPARK dialektu, kadangi vienas iš tikslų buvo užtikrinti, jog programa veiks „iš pirmo karto“ [Cha00].

2.3. Išvados

Iš dviejų svarstytų technologijų – C# ir Ada, tinkamesnė mūsų planuojamo sintaksinio analizatoriaus įgyvendinimui pasirodė Ada. Tai lėmė ne tik itin stiprus vienalaikiškumo palaikymas kalbos lygmenyje, bet ir egzistuojantis C kalbos interfeisas, leisiantis efektyviau ir paprasčiau susieti analizatorių su *cod-tools* įrankių rinkiniu. Be to, Ada kompiliavimas tiesiai į mašininį kodą, o ne į tarpinę kalbą, užtikrins spartesnį programos veikimą bei gi sumažins programos papildomus naudojimo poreikius.

Sintaksiniam analizatoriui planuojame panaudoti vienalaikes užduotis. Esame numatę, jog programoje bus bent viena gija, kuri skaidys ateinantį ženklų srautą ir kiekvieną gautą bloką dės į eilę. Ši eilė veikiausiai bus apribota, t. y. gebanti priimti ribotą blokų skaičių. Be to, bus viena arba daugiau gijų (užduočių), kurios ims minėtus blokus iš eilės ir apdoros juos *cod-tools* įrankių rinkinio pagalba.

3. Detalusis planas

3.1. Įvadas

Siekdami įvertinti Ada tinkamumą srautinio CIF parserio įgyvendinimui atlikome eibę empirinių bandymų tiek profesinės praktikos apimtyje, tiek už jos ribų. Šiais bandymais siekėme nustatyti šiuos esminius punktus:

1. Ada ir C suderinamumą per standartinį Ada interfeisą bei pastarojo panaudojamumą, jungiant Ada su esamu „cod-tools“ įrankių rinkiniu
2. Įvertinti praėjusiame skyriuje aptartus sintaksinio analizatoriaus įgyvendinimo modelius ir šablonus
3. Išnagrinėti Ada vienalaikiškumo modelį ir jo praktinį panaudojimą

3.2. Empiriniai bandymai

3.2.1. Ada ir C interfeisas

Ada ir C tarpusavio kodo sąveika yra pamatinis planuojamo srautinio CIF parserio aspektas, todėl yra itin svarbu, jog Ada ir C interfeisas veiktų sklandžiai bei sudarytų sąlygas esamam „cod-tools“ įrankių rinkiniui būti kviečiamam iš ir pačiam kviesti Ada kodą. Interfeiso patikra vyko dviem etapais: pirmiausia atlikome bandymus su baziniu Ada ir C kodo sujungimu, o sėkmingai pavykus tai atlikti, perėjome prie „cod-tools“ kodo sujungimo su Ada. Bandymai buvo sėkmingi ir abiem atvejais pavyko tiek kviesti C kodą Ada pusėje, tiek *vice versa*.

Be to, bandymų metu išryškėjo svarbus svarstytinas dizaino aspektas – laisvoji jungtis (angl. *loose coupling*). Kadangi vienas iš tikslų, kuriant srautinį CIF parserį, yra padidinti kodo perkeliamumą bei sudaryti sąlygas naudoti srautinį parserį kaip bazę kitoms programoms, interfeiso jungtis tarp Ada programinio kodo ir „cod-tools“ įrankių bazės turėtų būti kuo laisvesnė. Negana to, numatoma, kad „cod-tools“ ir srautinio sintaksinio analizatoriaus vystymas vyks atskirai, o tvirta jungtis daug artimiau sujungtų Ada ir C kodą. Šis sunkumas kyla iš to, kaip veikia Ada ir C interfeisas: Ada paketo (angl. *package* lygmenyje yra aprašoma kokios struktūros (plačiąja prasme) yra importuojamos (kintamieji, funkcijos, struktūros (siaurąja C terminologijos prasme) ar eksportuojamos (kintamieji, funkcijos, procedūros, ar įrašai (angl. *records*))). Šiam aprašymui generuoti galima panaudoti GCC kompiliatoriaus funkciją „dump-ada-spec“, kuriai pateikus *header* failą yra automatiškai sugeneruojamas Ada paketas. Problema kyla su C struktūromis (angl. *struct*), kurios Ada pusėje yra atvaizduojamos įrašų – „record“ – struktūros pavidalu. Kai C struktūra yra tik deklaruojama *header* faile, bet nėra aprašomi jos nariai (angl. *members*), naudojant automatinį specifikacijos generavimą Ada pakete gauname tuščią įrašą – „null record“ bei komentarą, kad struktūra yra nebaigta – „incomplete struct“. Tai reiškia, kad esant poreikiui pasiekti struktūros narius, jie turės būti aprašyti rankiniu būdu ir taisomi kiekvieną kartą struktūrai pakitus C pusėje.

Tvirtos sąsajos interfeiso lygmenyje galime išvengti dviem būdais. Vienu atveju galėtume visų struktūrų aprašymus C pusėje perkelti į *header* failus ir naudoti automatinį generavimą. Tai reikštų,

kad struktūrų nariai visuomet turėtų būti prieinami viešai. To išvengti galėtume antruoju būdu – Ada pusėje nesikreipdami į narius tiesiogiai, o per C kalbos funkcijas. Šiuo atveju Ada kodui yra nebūtina žinoti tikslios įrašo (struktūros) sandaros.

Dar vienas aspektas į kurį dera atkreipti dėmesį yra C kodo generavimas iš gramatikų. Jungtinio Ada ir C kodo kompiliavimas vykdomas sukuriant projekto failą, kuriame yra nurodomi išeities kodo katalogai, naudojamos kalbos bei pasirinktinai nurodomos įvairių programos paruošimo vykdomajai aplinkai stadijų opcijos. Ruošiant programą kompiliatorius automatiškai randa priklausomybių failus pagal projektiniame faile nurodytus katalogus bei sukompiliuoja tiek Ada, tiek C kodą. Tačiau šie įrankiai nesugeneruoja C failų iš formaliai aprašytų gramatikų, pvz., panaudojant *bison* įrankį. Nors galima panaudoti egzistuojančius Ada įrankius ir XML struktūra aprašyti instrukcijas, kurios turi būti išpildytos prieš programos paruošimo proceso pradžią, žymiai paprastesnis ir daugiau lankstumo suteikiantis atrodo variantas, kuriame tiek prerekvizitai, tiek programos paruošimas būtų aprašytas atskirame „make“ faile.

3.2.2. Architektūriniai modeliai

Praėjusiame skyriuje išskyrėme du architektūrinius modelius, kurie iš pirmo žvilgsnio atitinka srautinio CIF parserio reikalavimus. Abiejų modelių kertinė dalis yra eilė, į kurią būtų dedami ir iš kurios būtų imami paskiri CIF duomenų blokai. Be to, tokia eilė turi gebėti dirbti su daugiau nei viena vienalaike užduotimi (*task*). Bazinės Ada bibliotekos pateikia bendruosius (angl. *generic*) saugyklų (angl. *container*) paketus, kurių tipus galima panaudoti konstruojant naujus tipus paveldėjimo metodu. Mūsų aptartiems architektūriniams modeliams itin tinkamas atrodo sinchronizuotos apribotos eilės tipas (angl. *synchronized bounded queue*) [AXE12]. Paveldint šį tipą privalome nurodyti eilėje būsiančių elementų tipą ir eilės dydį. Be to, kadangi šis tipas yra sinchronizuotas, nuskaitymo užduotys, kai eilė tuščia, o užrašymo užduotys, kai ji yra užpildyta, yra pervedamos į laukimo režimą.

Taipogi, praktiniame atitikties įrodyme išbandėme minėtąjį eilės tipą su daugiau nei vienu vartotoju, šiuo atveju daugiau nei viena užduotimi (*task*), imančia CIF duomenų blokus iš eilės. Mūsų ankstesniame skyriuje aptartų modelių vartotojų esminis skirtumas yra tas, kad vienu atveju vartotojai seka savo būseną ir gali suteikti informaciją išoriniams užklausėjams, o kitu – ne. Empirinių bandymų metu nustatėme, kad užduoties būsenos sekimas nėra būtinas, kol architektūra leidžia analizuoti tik vieną failą. Tačiau, kadangi vienas iš iškeltų tikslų yra galimybė apdoroti begalinį srautą arba pateikti parseriui sąrašą failų, kuriuos reikia apdoroti, susidaro poreikis užduoties būsenos sekimui. Be to, bandymų metu pastebėjome, jog tiek sklandžiai parserio darbo pabaigai, tiek daugiau negu vieno failo apdorojimui, būtina žinoti, kada į eilę yra įdedamas paskutinis viso srauto ar failo duomenų blokas. Šiuo tikslu atlikome pakeitimus „cod-tools“ C kode, kad paskutinis failo duomenų blokas būtų pažymėtas, taip perduodant šią informaciją į Ada pusę.

3.2.3. Atminties valdymas

Ypatingai svarbus aspektas išryškėjęs praktinio aititkties įrodymo metu yra tinkamas atminties valdymas. Kaip žinia, C programavimo kalba pasižymi dinamišku atminties išskyrimu bei

galimybe operuoti nuorodomis (angl. *pointers*) į konkrečius atminties adresus, kuriuose yra saugoma informacija. Ada aplinkoje šias nuorodas atitinka prieigos tipas (angl. *access type*), suteikiantis prieigą prie atminties vietos, kurioje saugoma kito tipo informacija. Esamame nesrautiniame „cod-tools“ parseryje išskirta atmintis likdavo C kodo kontrolėje, todėl ji būdavo atlaisvinama pabaigus darbą su informacija. Naujojo srautinio parserio atveju nuorodos į išskirtas atminties vietas palieka C kontekstą ir yra perduodamos į Ada pusę, o C pusė nesužino, kada darbas su šiomis atminties vietomis yra baigtas. Dėl šios priežasties iškyla atminties nutekėjimo (angl. *memory leak*) pavojus.

Kad išvengtume atminties nutekėjimo praktiniame atitikties įrodyme suderinome kelių sluoksnių pagalbinę struktūrą bei pakeistas (angl. *overriden*) standartines Ada derinimo (angl. *adjust*) ir finalizacijos (angl. *finalization*) procedūras. Derinimo procedūra yra kviečiama, kai jos tipo kintamiesiems yra priskiriama reikšmė aktyviame kontekste (tai yra nuoroda į atminties vietą yra papildomai panaudojama – dubliuojama), o finalizacijos – kai šio tipo kintamieji palieka aktyvų kontekstą, t.y. darbas su tam tikra nurodos kopija yra baigtas. Baziniame struktūros, dedamos į eilę, lygmenyje yra duomenų blokas (*datablock*) bei prieiga (nuoroda C terminologijoje) prie jo (*datablock access*). Ši struktūra yra apvelkama (angl. *wrapped*) į skaičiuojamo duomenų bloko tipą (*counted datablock*), kuris savyje talpina patį prieigos tipą, t.y. nuorodą, bei jo panaudojimų skaičių. Galiausiai, įprastoms operacijoms, pvz., sąveikai su eile, yra naudojamas skaičiuojamojo duomenų bloko prieigos tipas (*counted datablock access*), taip užtikrinant, kad visuomet bus panaudojama ta pati nurododa į konkrečią atminties vietą bei nuorodų skaitiklis atitiks realų nuorodų panaudojimo skaičių. Šiuo būdu derinimo ir finalizacijos procedūros, kurioms priskirtas skaičiuojamo duomenų bloko prieigos tipas, atitinkamai padidina arba sumažina nuorodų skaitliuką. Be to, finalizacijos procedūra seka, ar yra bent viena aktyviai naudojama nuoroda (t.y. skaitliuko reikšmė yra daugiau negu 0). Jei taip nėra, tuomet finalizacijos procedūra incijuoja atminties atlaisvinimą, panaudojant jau esamas „cod-tools“ funkcijas. Tokiu būdu, Ada kodas tampa duomenų blokų savininku bei yra atsakingas už tinkamą atminties valdymą.

3.3. Planas

Po empirinių bandymų ir praktinio atitikties įrodymo darbus tęsime dviem fazėmis. Pirmoji fazė skirta galutiniam srautinio parserio įgyvendinimui. Kertinė parserio dalis turės būti apribota eilė, kuri turės būti apsaugota nuo lenktynių tarp vienalaikių loginės kontrolės gijų. Architektūrinis modelis remsis „pašto dėžutės“ šablonu: skaitančios loginės kontrolės gijos – užduotys – turi gebėti informuoti išorės esybes apie savo būseną, kad galėtume sudaryti sąlygas nenutrūkstamo srauto arba kelių failų sintaksinei analizei. Šis bazinis funkcionalumas privalės būti atskirtas į Ada biblioteką, kurią būtų galima naudoti tiek Ada, tiek C kode per aiškiai apibrėžtą ir aprašytą interfeisą. Tai sudarys galimybę tolimesniam parserio vystymui bei patogiam perpanaudojimui su jau esamu „cod-tools“ funkcionalumu – egzistuojančios programos bus aprašomos Ada pusėje atskiruose programų failuose. Be to, kadangi Ada bibliotekos viduje kurs ir naudos vienalaikes užduotis, tai suteiks galimybę naudoti vienalaikę terpę C kalboje. Be to, kaip minėjome anksčiau, vienalaikės užduotys, skaitančios atskirus duomenų blokus iš eilės atskirai, sudaro galimybes išlygiagretinti visą sintaksinės analizės procesą per UNIX konvejerius.

Saugiam ir tvarkingam atminties valdymui planuojame naudoti aptartą kelių lygmenų duomenų bloko struktūrą, kuri leis Ada pusei automatiškai sekti, kada atmintis turi būti atlaisvinta ir užtikrinti, kad ji nebūtų atlaisvinta anksčiau nei baigtas darbas su konkrečia atminties vieta.

Be to, būtina paminėti, kad „cod-tools“ įrankių rinkinys yra aktyviai vystomas ir naudojamas, o naujasis srautinis CIF parseris reikalauja pokyčių ir šio įrankių rinkinio kodo bazėje, todėl nusprendėme, kad sieksime srautinio CIF parserio funkcionalumo įgyvendinimo kaip pasirinktinos funkcijos tiek kompiliavimo, tiek vykdymo metu.

Antruoju etapu analizuosime įgyvendinto srautinio CIF parserio veikimą ir našumą. Pirmiausia išmatuosime rezidentinės aibės dydį ir palyginsime jį su esamo parserio rezidentinės aibės dydžiu. Kartu matuosime ir abiejų analizatorių duomenų apdorojimo trukmę bei įvertinsime našumo pokyčius.

4. Įgyvendinimo eiga

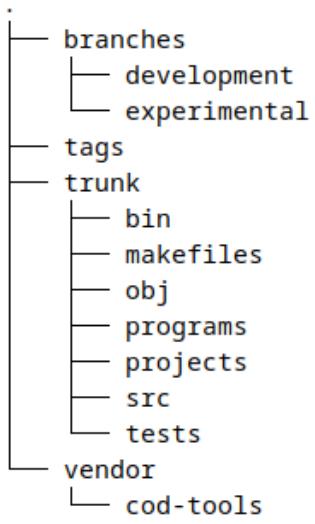
4.1. Įgyvendinimo aplinka

Įgyvendinimo darbai buvo vykdomi naudojant Lenovo Thinkpad x220 modelio nešiojamąjį kompiuterį. Šio kompiuterio komplektacijoje yra 8 GB operatyviosios atminties ir Intel i5-2450M procesorius, turintis du fizinius branduolius ir keturias gijas. Bazinis šio procesoriaus dažnis yra 2,5 GHz, o maksimalus – 3,1 GHz. Procesoriaus adreso plotis yra 64 bitai. Vystymo darbams buvo naudojama Linux operacinė sistema, „Arch Linux“ distribucija, pasižyminti pastovios laidos (angl. *rolling release*) modeliu, Kodas buvo rašomas naudojat *Neovim* teksto redagavimo programą. Kodo versijavimas atliktas per *Apache Subversion* versijavimo sistemą atskiroje repozitorijoje.

Kodą kompiliavome nauddami *Free Software Foundation* įrankiais: C kodo kompiliavimui naudojome *GCC*, o Ada kodui – *GNAT* kompiliatorių, integruotą į *GCC* kompiliatorių sistemą.

4.2. Įgyvendinimo organizacija

Repoztorijoje sukūrėme keturis pirmo lygmens katalogus: *branches*, *tags*, *trunk* ir *vendor* (3 pav.). *Trunk* (arba kamieno) katalogas savyje talpina stabilų naujausią programos kodą. *Branches* katalogas skirtas vystymo atsišakojimams nuo kamieno. Mūsų analizatoriaus įgyvendinimo pradžioje kūrėme šakas kertinių Ada ir C interfeisinimo bei pačios Ada kalbos ypatybių nagrinėjimui ir išbandymui. *Tags* katalogas skirtas ateityje, tęsiant įrankio vystymą, užkonservuotoms įrankio versijoms – tolesnis tobulinimas jose negalimas ir yra vykdomas šakose, vėliau prijungiant prie kamieno. *Vendors* katalogas skirtas išoriniams resursams. Mūsų atveju tai yra *cod-tools* analizatoriaus išeities kodas. Mūsų įrankio vystymo pradžioje *cod-tools* kamieninė šaka buvo dubliuota, kad galėtume atlikti būtinus pakeitimus *cod-tools* pusėje. Tolesnio vystymo planuose yra šio kodo prijungimas prie pagrindinės *cod-tools* repozitorijos, papildant selektyvios kompiliacijos funkcionalumu.



3 pav. Repozitorijos struktūra (du aukščiausi lygmenys)

4.3. Įgyvendinimo darbai

Sintaksinio analizatoriaus įgyvendinimo darbai buvo vykdomi pakopomis, pagal darbų prioritetą ir priklausomybes. Kaip jau minėjome, praktinės atitikties įrodymo metu išsiaškinome atminties nutekėjimo pavojų, kai duomenų bloko kontrolė perleidžiama iš C į Ada pusę ir šiuo tikslu sukūrėme kontroliuojamo duomenų bloko tipą Ada pusėje bei nurodėme jam specifines finalizacijos (t.y. atminties atlaisvinimo) instrukcijas. Šis tipas tapo baziniu elementu su kuriuo vykdomos operacijos Ada pusėje.

Taip pat, kaip ir buvo planuota pradžioje, panaudojome Ada pateikiamą sinchronizuotos apribotos eilės paketą, nustatydami kontroliuojamo duomenų bloko tipą kaip eilės elementą (t.y. tipą, kuris bus dedamas į eilę), o eilės dydį apribodami iki 5–ų. Kadangi elementų rašymui į eilę ir nuskaitymui iš jos panaudojome Ada vienalaikes užduotis (4 pav.), o eilė yra sinchronizuota, skaitanti užduotis, užsipildžius eilei (pasiekus jau minėtą nustatytą eilės elementų skaičiaus apribojimą) yra sustabdoma ir pervedama į laukimo būseną, ligi eilėje atsiras vietą patalpinti elementą. Tokiu pačiu principu skaitanti užduotis (ar užduotys, kaip matysime tolimesniame funkcionalumo aprašyme) eilei esant tuščiai, yra pristabdoma ir laukia, kol eilėje atsiras elementų, kuriuos galėtų nuskaityti.

```
task Cif_Parser_Task is
  entry Begin_Parsing (Cif_Filename : Unbounded_String;
                       Cif_Options : Cif_Option_T;
                       Error_Code_Access : Error_Code_T_Access);
end Cif_Parser_Task;

task body Cif_Parser_Task is
  Task_Cif_Filename : Unbounded_String;
  Task_Cif_Options : Cif_Option_T;
  Task_Error_Code_Access : Error_Code_T_Access;
begin
  -- On some OSes, notably on Debian-10 and Ubuntu-20.04, the BSS
  --  segment seems to be uninitialised, and the parser fails with
  --  Assertion "`!cif_cc' failed" in
  --  src/externals/codcif/cif_grammar.y:502. To avoid this, the
  --  static global variables in the C code are now initialised
  --  explicitly:
  Cif1_Init_Parser;
  Cif2_Init_Parser;
  loop
    select
      accept Begin_Parsing (Cif_Filename : in Unbounded_String;
                            Cif_Options : in Cif_Option_T;
                            Error_Code_Access : in Error_Code_T_Access) do
        Task_Cif_Filename := Cif_Filename;
        Task_Cif_Options := Cif_Options;
        Task_Error_Code_Access := Error_Code_Access;
      end Begin_Parsing;
      declare
        Filename_Char_Array :  char_array := To_C (To_String (Task_Cif_Filename), Append_Nul => TRUE);
      begin
        Parse_Cif_From_File_With_Error_Code
          (Filename_Char_Array, Task_Cif_Options, Task_Error_Code_Access);
      end;
      or
        terminate;
    end select;
  end loop;
end Cif_Parser_Task;
```

4 pav. Rašančiosios Ada vielanaikės užduoties deklaracija ir įgyvendinimas.

Idant suplanuotas funkcionalumas veiktų korektiškai, koregavome esamus *cod-tools* YACC failus, kad sintaksinės analizės metu kiekvienas išanalizuotas duomenų blokas nebūtų dedamas į CIF failo struktūrą, apibrėžtą C kode, o būtų iškviečiama iš Ada kodo eksportuota procedūra *enqueue_datablock*, kuriai pateikiama nuoroda į duomenų bloko struktūrą, o Ada pusėje ji yra apvelka-

ma į kontroliuojamo duomenų bloko tipą ir įdedama į eilę.

Vystymo metu iškilo paskutinio duomenų bloko problema: Ada kodui būtina žinoti, kada yra pasiekiamas paskutinis CIF failo duomenų blokas. Tai yra būtina norint gebėti tiek analizuoti kelis failus iš eilės. Šią problemą išsprendėme papildydami C kode apibrėžtą duomenų bloko struktūrą logine reikšme (angl. *boolean value*) – *last_datablock_in_stream* – kuri yra pažymima kaip teisinga tuomet, kai *cod-tools* parseris kelia į eilę paskutinį srauto duomenų bloką. Kartu, papildėme Ada kodą logine reikšme, žyminčia, ar sintaksinė analizė yra aktyvuota (naudojama C kode, kaip tikrinama sąlyga, ar sintaksinė analizė turėtų būti tęsiama), procedūromis *Enable_Parsing* ir *Stop_Parsing* (keičia minėtosios loginės reikšmės teisingumą), procedūra *Flush_Queue* (išvalo eilę nuo bet kokių užsilikusių elementų, kai analizės procesas yra sustabdytas, pvz., kai iš vieno failo yra gautas norimas duomenų blokas ir reikia pradėti kito failo sintaksinę analizę) bei funkcija *Is_Parsing_Stopped* (grąžina minėtąją loginę reikšmę). Kartu, pastaroji funkcija yra eksportuojama, kad tiek *cod-tools* mūsų atveju, tiek kitos programos, kuriamos ateityje, galėtų gauti informaciją apie dabartinę sintaksinės analizės būseną.

Įgyvendinus eilės funkcionalumą kilo poreikis gebėti jį ištestuoti. Šiuo tikslu pradėjome kurti ir *trunk/programs* kataloge patalpinome dvi Ada programas – *cif_print_datablock_names*, į standartinę išvestį išvedančią visus duomenų blokų vardus iš jai pateikto CIF duomenų srauto, ir *cif_print_datablock*, kuri jai pateikus duomenų bloko ar kelių blokų vardus ir CIF duomenų srautą, susidūrusi su jais sraute, į standartinę išvestį išveda visų užklaustų duomenų blokų aprašus. Tokiu būdu galėjome testuoti ne tik patį eilės funkcionalumą, bet kartu ir analizės proceso stabdymą bei atnaujinimą.

Testavimus atlikome su trimis skirtingos apimties duomenų rinkiniais – 4-ų, 28-ų bei 38518-os duomenų blokų CIF failais. Pastarasis tapo kertiniu testu tiek mūsų kurtam analizatoriui, tiek vėliau tyrimo metu lygintiems jau esamiems sintaksiniams analizatoriams. Dėl didelio duomenų kiekio, šio failo apdorojimas reikalauja itin didelio operatyviosios atminties kiekio, kadangi esami parseriai, įskaitant *cod-tools* įrankių rinkinį, bando joje išskleisti visą dokumento medį.

Tolimesnėse vystymo pakopose pridėjome testų katalogą, kuriame talpinome programoms skirtus testus (magistro baigiamojo darbo vystymo apimties pabaigoje jų buvo 43), kad atliekant pakeitimus jau sukurtose programose galėtume efektyviai patikrinti, ar numatytas funkcionalumas nebuvo sulaužytas. Kartu, tai pridėjo vystymo per testavimą (angl. *test driven development* arba sutrumpintai – TDD) elementą visam procesui.

Galų gale, pridėjome trečią programą, bandančią nustatyti visų duomenų blokų žymų (angl. *datablock tag*), sutiktų pateiktame duomenų sraute, ribines reikšmes. To pasiekėme sukurdami papildomą duomenų žymos įrašo tipą (5 pav.), savyje talpinantį minimalią ir maksimalią reikšmes bei duomenų bloko vardą. Pačioje programoje šio tipo elementus talpiname į maišos (angl. *hash*) žodyną, prieš tai patikrindami, ar tokio vardo elemento jau nėra jame. Jei tokį elementą randame, lyginame jo reikšmes su tomis, kurios gautos iš iš eilės nuskaityto duomenų bloko. Kadangi duomenų bloko žymų reikšmės gali būti daugybės įvairių tipų (simbolių eilutė, skaitinė reikšmė, masyvas, žodynas ir pan.), detalus palyginimas pagal duomenų tipą reikalautų didelių pakeitimų esamame *cod-tools* kode ir laiko sąnaudų. Be to, programos tikslas yra nustatyti ribines reikš-

mes, kad vartotojas galėtų išsiaiškinti, kurios iš šių reikšmių yra nevalidžios. Todėl, šiuo atveju nusprendėme lyginti reikšmes kaip simbolių eilutes. Todėl skaitinė reikšmė, kuri tokiu būdu nustatoma kaip pvz., mažesnė, nebūtinai bus mažesnė lyginamas reikšmes pavertus į skaitinę išraišką. Tačiau, mūsų programos tikslą toks lyginimas išpildo.

```
type Datablock_Tag is record

  Name      : Unbounded_String;
  Min_Value : Unbounded_String;
  Max_Value : Unbounded_String;

end record;

package Datablock_Tags_Map is new Ada.Containers.Indefinite_Hashed_Maps
  (Key_Type => String,
   Element_Type => Datablock_Tag,
   Hash => Ada.Strings.Hash,
   Equivalent_Keys => "=");
```

5 pav. Duomenų bloko žymos įrašo tipo ir maišos žodyno deklaracijos.

Užbaigus pagrindinį suplanuotą funkcionalumą, kaip paskutinį vystymo darbą įgyvendinimo galimybę parseriui teikti duomenis standartinės įvesties būdu (pvz., konvejeriu). Kadangi *cod-tools* parseris geba skaityti standartinę įvestį, Ada kodas patikrina, ar yra pateiktas bent vienas failas ir neradus nė vieno duomenų failo signalizuoja C kodui skaityti standartinę įvestį.

4.4. Įgyvendinimo metu iškilusios problemos

Įgyvendinimo metu su didesnėmis problemomis nesusidūrėme. Pasiruošimo stadijoje atlikti empiriniai bandymai, kuriant praktinės atitikties įrodymą gana anksti išryškino jau anksčiau aptartas problematiškas vietas, tad įgyvendinimą pradėjome jau su esamais surastais sprendimais arba numatytais galimais sprendimais. Tačiau, nemažą iššūkį kėlė darbas su *cod-tools* kodo baze. Daugybės metų ir žmonių įdirbis kėlė sunkumų ne tik savo apimtimi, bet ir persipynusiomis, kartais itin komplikuotomis struktūromis ir ryšiais tarp atskirų kodo dalių. Tokiais atvejais tekdavo taikyti savo vystomą įrankį prie šių egzistuojančių kodo dalių bei ieškoti būdų apeiti iškilusias problemas, neprarandant funkcionalumo ar našumo. Tam teko skirti itin daug laiko, kuris galėjo būti panaudotas papildomų funkcionalumų vystymui.

Taipogi, būdavo atvejų, kai Ada kalbos ypatybės, bandant panaudoti egzistuojančius jos įrankius (pvz., įvairių tipų konteinerių struktūras – vektorius ar kt.), būdavo ne itin aiškios. Ada bendruomenė yra itin maža, kalba nėra plačiai naudojama, tad informacijos trūkumas kartais prilėtindavo vystymo procesą. Tokiais atvejais naudodavomės oficialiu Ada žinynu, tačiau detalesni veikimo principai ar išaiškinimai jame būtų pagelbėję dirbti efektyviau ir įgyvendinti papildomus funkcionalumus.

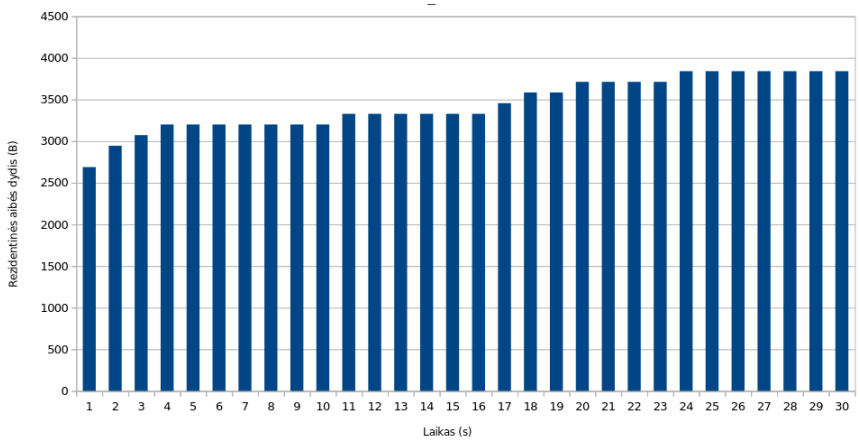
5. Testavimas ir palyginimas su esamais analizatoriais

5.1. Pasiruošimas

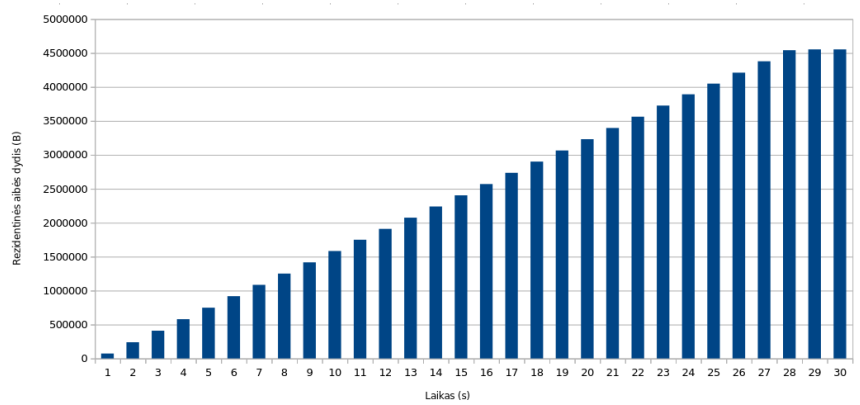
Siekdami palyginti mūsų ir esamų CIF parserių [MVB+16] rezidentinių aibių dydžius ir veikimą bendrai, atsirinkome kelis egzistuojančius ir stabiliai veikiančius analizatorius. Visų pirmą, atmetėme visus parserius, sukurtus *Python* 2.x versijomis. Šios kalbos versijos jau pasiekė savo gyvavimo ciklo pabaigą (angl. *end-of-life*), tad yra ne tik keblesnės kompiliuoti, bet ir yra nesaugios naudoti, kadangi nebėra aktyviai palaikomos ir jų saugumo spragos yra netaisomos. Iš žinomų analizatorių sąrašo atrinkome dokumentuotus, prieinamus ir stabiliai veikiančius du parserius: *vcif* ir *crystcif*, sukurtus atitinkamai C ir Javascript kalbomis bei palyginome juos su *cod-tools* įrankių rinkinyje esančia *cifparse* programa bei mūsų pačių sukurta *cif_print_datablock_names* programa.

Palyginimui naudojome CIF komponentų žodyno failą, savyje talpinantį 38518-ką duomenų blokų. Pradinių testų su egzistuojančiais parseriais metu nustatėme, kad abu esami analizatoriai veikia itin neefektyviai: *vcif* sintaksinės analizės procesas yra itin lėtas, o *crystcif* nesugebėjusi apdoroti testinio failo išveda nenumatytą Javascript klaidą, jog nebėra erdvės krūvos atminties išskyrimui: angl. *JavaScript heap out of memory*. Todėl nusprendėme paleisti kiekvieno parserio procesą su komponentų žodynu kaip pradiniais duomenis ir naudodami Linux operacinės sistemos teikiamus įrankius (*ps*, *grep*, *sleep*) fiksuoti pirmųjų 30-ies sekundžių procesų rezidentinių aibių dydžius.

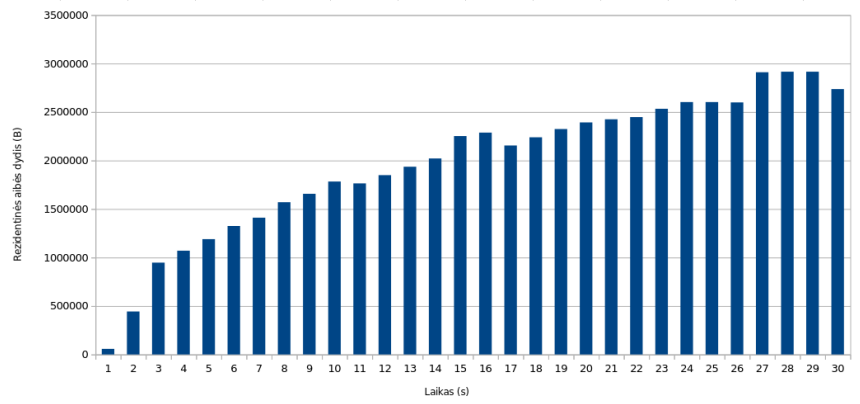
Be to, atlikome ir apkrovos testą naudodami mūsų parserį – *cif_print_datablock_names* programai kaip įvesties duomenis pateikėme penkis tuos pačius komponentų žodyno failus, taip padidindami apdorojamų duomenų blokų kiekį iki 192590-ies. Šio testo metu irgi fiksavome rezidentinės aibės dydžius kiekvieną programos vykdymo trukmės sekundę.



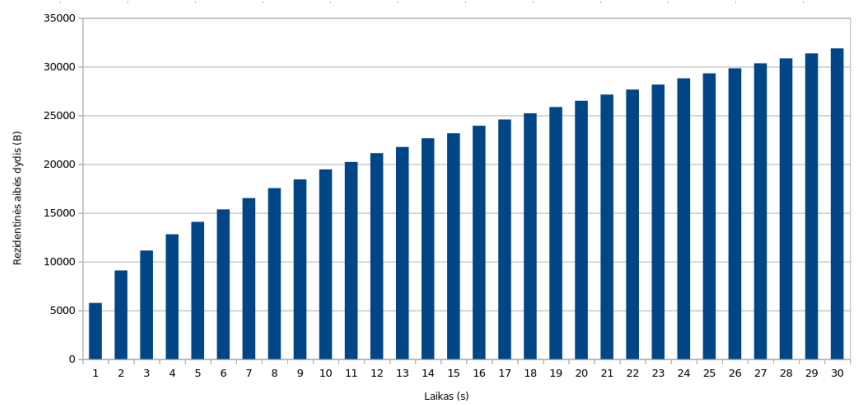
(a) cif_print_datablock_names



(b) cifparse

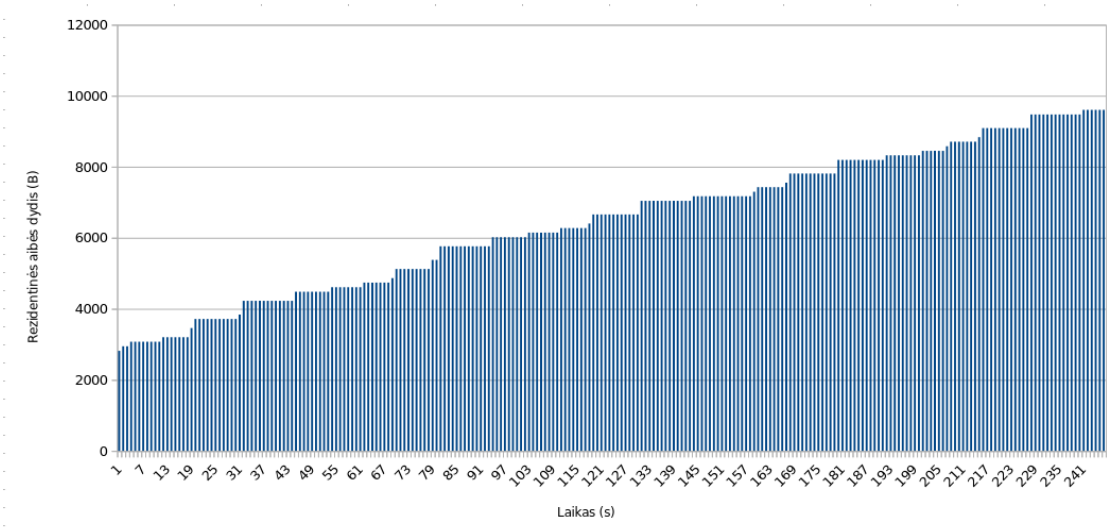


(c) crystcif



(d) vcif

6 pav. Rezidentinės aibės testo rezultatai.



7 pav. Apkrovos testo rezultatai.

5.2. Testavimo rezultatai

Atlikus testavimą nustatėme, kad tiek *vcif*, tiek *crystcif* nesugeba sėkmingai apdoroti komponentų žodyno failo – pirmasis dėl itin ilgos parsinimo proceso trukmės, o pastarasis dėl vykdymo laiku kylančio krūvos atminties trūkumo klaidos. Tačiau net jei šie analizatoriai ir sugebėtų tai padaryti sėkmingai, kaip matome iš rezultatų (6 pav.), rezidentinės aibės dydis net ir pradiniame analizės etape yra ženkliai didesnis nei mūsų įrankio.

Iš testavimo metu naudotų parserių, sėkmingai apdoroti komponentų žodyną pavyko tik su mūsų sukurtu srautiniu ir *cod-tools* parseriais. Nors pastebėjome išaugusią apdorojimo trukmę (*cod-tools* parseris duomenis apdorojo per ~29 sekundes, o mūsų analizatorius per ~46 sekundes), tačiau palyginus rezidentinių aibių dydžių aukščiausias reikšmes, matome, jog mūsų parserio rezidentinės aibės dydis yra net 1200 kartų mažesnis nei *cod-tools* parserio. Tai reiškia, kad mūsų sukurtas parseris yra žymiai efektyvesnis ir gali būti naudojamas mažus operatyviosios atminties resursus turinčiose sistemose.

Apkrovos testas (7 pav.) buvo sėkmingas. Mūsų analizatoriaus programa sėkmingai apdorojo visą duomenų srautą per ~242 sekundes. Nors testas parodė išaugusį rezidentinės aibės dydį, jis vis tiek yra ženkliai mažesnis nei kitų lygintų parserių.

6. Tolimesnio vystymo ir panaudojimo perspektyvos

Pradėdami vystyti srautinį sintaksinį analizatorių jam numatėme du panaudojimo būdus. Pirmasis būdas yra remtis parašytu Ada įrankiu, kaip pagrindu, kurį panaudojant kuriamos nuosavos programos. Šiam panaudojimo būdui sudarytos visos sąlygos, kadangi jis remiasi esminiu įgyvendintu parserio funkcionalumu. T.y. vartotojas naudos tas procedūras ir funkcijas, kurios pačios sukuria reikalingas struktūras korektiškam sistemos veikimui. Kitas panaudojimo būdas būtų naudoti mūsų sintaksinį analizatorių kaip biblioteką, kurios pagalba būtų kuriamos kur kas sudėtingesnės programos. Siekdami užtikrinti šią galimybę dalį funkcijų ir procedūrų pavertėme daugiareikšmėmis (angl. *overload*). Vietoj įprastai automatiškai kuriamos eilės, kurią naudoja pirmuoju atveju kviečiamos funkcijos ir procedūros, šiuo atveju vartotojui paliekama galimybė pačiam apsibrėžti reikiamą eilę ir ją perduoti minėtosioms funkcijoms arba procedūroms. Šios, naudodamos esamas standartines Ada kalbos priemones gebės atlikti reikalingus veiksmus, kad numatytas įrankio funkcionalumas būtų išpildytas. Be to, kadangi bazinės funkcijose ir procedūrose su eile dirba (tiek rašo, tiek skaito) po vieną vienalaikę užduotį, naudojant ką tik aptartas funkcijas galima didinti užduočių skaičių, taip dar efektyviau apdorojant duomenis.

Vienalaikių užduočių panaudojimas Ada kode ir su eilės operacijomis susijusių elementų eksportavimas sudaro galimybes šį įrankį naudoti kaip biblioteką, leidžiančią naudoti vienalaikius procesus, kalboms, kuriose vienalaikiškumas neįgyvendintas arba jo panauda yra itin komplikuota, pvz., C. Taipogi, kadangi sukurtas srautinis parseris geba apdoroti dalinį duomenų srautą vienalaikiu būdu, tai sudaro sąlygas *shell* veikimo vienalaikiškumui ir lygiagretumui. Sukurtas parseris, atsidūręs konvejeriu sujungtų komandų grandinės viduryje, nestabdys visos grandinės, o gebės perduoti duomenis tolimesniam apdorojimui.

Kaip jau minėjome anksčiau, vienoje iš bandomųjų programų, ieškančio duomenų žymų ribinių reikšmių, šios reikšmės lyginamos kaip simbolių eilutės. Vienas iš prioritetinių darbų tolimesnio vystymo pakopose būtų įgyvendinti pilną palyginimą pagal atitinkamą reikšmės duomenų tipą.

Taipogi, dėl laiko stokos, nespėjome įgyvendinti CIF validacijos ir kitų klaidų perkėlimo iš C į Ada pusę. Vystymo metu vykusiuose pasitarimuose kaip vienas iš sprendimo būtų buvo svarstytas variantas pridėti klaidų pranešimus prie duomenų bloko, kad Ada pusėje skaitant duomenų blokus iš eilės būtų galima gauti šiuos pranešimus. Tačiau, toks funkcionalumas reikalautų didelių pokyčių *cod-tools* kode, tad jie palikti įgyvendinimui ateityje. Šios jungties trūkumas šiuo metu kelia problemas tam tikrose vietose. Kaip jau minėjome, mūsų sukurtas parseris geba priimti duomenis per standartinę įvestį. Tai jis padaro pasiųsdamas signalą *cod-tools* parseriui, jog šis skaitytų duomenis iš standartinės įvesties. Tačiau Ada kodas nevaliduoja nei tokiu būdu pateiktų duomenų validumo, nei jų egzistavimo apskritai. Tad jei standartine įvestimi paduodama, pvz., tuščia simbolių eilutė, Ada negauna klaidos pranešimo iš C pusės, tad programa lieka amžiname cikle, laukdama, kol eilėje atsiras bent vienas elementas, kurį galėtų nuskaityti.

7. Išvados

1. Esami CIF parseriai nesugeba apdoroti didelės apimties CIF failų arba tai daro itin neefektyviai, bandydami išskleisti visą dokumento medį operatyviojoje atmintyje.

2. Sintaksinis analizatorius, kuris nesugeba apdoroti dalinių duomenų ir esantis konvejeriu sujungtos procesų grandinės viduryje, užkerta kelią vienalaikiškam ir lygiagrečiam šios grandinės procesų vykdymui.

3. „Gamintojo-vartotojo“ eilės architektūrinis modelis leidžia efektyviai dalimis apdoroti didelius duomenų kiekius.

4. Sėkmingai sukurtas srautinis sintaksinis analizatorius, gebantis apdoroti didelės apimties duomenis su ženkliai mažesniu rezidentinės aibės dydžiu.

5. Sukurtas srautinis sintaksinis analizatorius gali būti naudojamas tolimesniam darbui su CIF struktūros duomenimis tiek Ada, tiek kitose kalbose, panaudojant Ada teikiamą vienalaikiškumo funkcionalumą.

Literatūra

[AP11] H. I. Ali ir L. M. Pinho. A parallel programming model for Ada. *Proceedings of the 2011 ACM annual international conference on special interest group on the Ada programming language*, SIGAda '11, p.p. 19–26, Denver, Colorado, USA. Association for Computing Machinery, 2011. ISBN: 9781450310284. DOI: 10.1145/2070337.2070350. URL: https://doi.org/10.1145/2070337.2070350.

[AXE12] AXE Consultants. *Annotated Ada Reference Manual. 2012 Edition*. 2012, 1270 p.

[AXE21] AXE Consultants. *Ada Reference Manual. 202x Edition, Draft 32*. 2021, 1148 p.

[BC17] C. Brandon ir P. Chapin. The use of SPARK in a complex spacecraft. *ACM SIGAda Ada Letters*, 36(2):18–21, 2017-05. DOI: 10.1145/3092893.3092896.

[BHM+96] D. Brutzman, T. Healey, D. Marco ir B. McGhee. The *Phoenix* autonomous underwater vehicle. *Technology and the Mine Problem Symposium*, tom. 1, skyr. 5, p.p. 79–99. Naval Postgraduate School, 1996.

[Bud98] T. A. Budd. Functional programming and the fragile base class problem. *ACM SIGPLAN Notice*, 33(12):66–71, 1998. ISSN: 0362-1340. DOI: 10.1145/307824.307881.

[But14] P. Butcher. *Seven concurrency models in seven weeks: when threads unravel*. Pragmatic Bookshelf, 1-as leid., 2014. ISBN: 1937785653.

[BW07] A. Burns ir A. Wellings. *Concurrent and real-time programming in Ada*. Cambridge University Press, USA, 3rev ed leid., 2007. ISBN: 0521866979.

[Cha00] R. Chapman. Industrial experience with SPARK. *ACM SIGAda Ada Letters*, XX(4):64–68, 2000-12. DOI: 10.1145/369264.369270.

[Dic92] T. P. Dickens. Technique for using a geometry and visualization system to monitor and manipulate information in other codes. *Software Systems for Surface Modeling and Grid Generation*, numeris 3143 NASA Conference Publication, p.p. 501–508. National Aeronautics and Space Administration, National Aeronautics ir Space Administration, 1992. URL: https://ntrs.nasa.gov/citations/19920015154.

[Dol18] S. Dolan. Jq 1.6 manual. 2018-11. URL: https://stedolan.github.io/jq/manual/v1.6/. Tikrinta 2021-11-09.

[ECM17] ECMA International. Standard ECMA-334 - C# Language Specification. Standard. Versija 5, ECMA International, 2017-12. 494 p. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-334_5th_edition_december_2017.pdf.

[ES19] K. Elkins ir G. Shirah. How NASA uses render time procedurals for scientific data visualization. *SIGGRAPH Asia 2019 Technical Briefs*, SA '19, p.p. 103–105, Brisbane, QLD, Australia. Association for Computing Machinery, 2019. ISBN: 9781450369459. DOI: 10.1145/3355088.3365169.

[GCD+09] S. Gražulis, D. Chateigner, R. T. Downs, A. F. T. Yokochi ir k.t. Crystallography Open Database – an open-access collection of crystal structures. *Journal of Applied Crystallography*, 42:726–729, 2009. DOI: 10.1107/S0021889809016690. URL: http://dx.doi.org/10.1107/S0021889809016690.

[GDM+12] S. Gražulis, A. Daškevič, A. Merkys, D. Chateigner ir k.t. Crystallography Open Database (COD): an open-access collection of crystal structures and platform for world-wide collaboration. *Nucleic Acids Research*, 40:D420–D427, 2012. DOI: 10.1093/nar/gkr900. URL: http://nar.oxfordjournals.org/content/40/D1/D420.abstract.

[GMB15] K. Gallaba, A. Mesbah ir I. Beschastnikh. Don't call us, we'll call you: characterizing callbacks in Javascript. *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, p.p. 1–10, Beijing, China. IEEE, 2015. ISBN: 978-1-4673-7899-4. DOI: 10.1109/ESEM.2015.7321196.

[HAB91] S. R. Hall, F. H. Allen ir I. D. Brown. The crystallographic information file (CIF): a new standard archive file for crystallography. *Acta Crystallographica Section A*, 47:655–685, 1991. DOI: 10.1107/S010876739101067X. URL: http://dx.doi.org/10.1107/S010876739101067X.

[HKV+21] S. Handa, K. Kallas, N. Vasilakis ir M. C. Rinar. An order-aware dataflow model for parallel Unix pipelines. *Proceedings of the ACM on Programming Languages*, 5(ICPF):1–28, 2021. DOI: 10.1145/3473570.

[HM03] D. J. Howe ir S. Michell. An approach to formal verication of real time concurrent Ada programs. *Proceedings of the 12th international workshop on real-time Ada*, IRTAW '03, p.p. 87–92, Viana do Castelo, Portugal. Association for Computing Machinery, 2003. ISBN: 9781450374460. DOI: 10.1145/959222.959238. URL: https://doi.org/10.1145/959222.959238.

[Int12] International Organization for Standardization. ISO/IEC 8652:2012 Information technology — Programming languages — Ada. Standard. Versija 3, International Organization for Standardization, 2012-12. 832 p.

[YW11] C. You ir S. Wang. A data parallel approach to XML parsing and query. *2011 IEEE International Conference on High Performance Computing and Communications*, p.p. 520–527, 2011-09. DOI: 10.1109/HPCC.2011.74.

[Jel89] J. F. Jeletic. Operational computer graphics in the flight dynamic environment. *Graphics Technology in Space Applications (GTSA 1989)*, numeris 3045 Nasa Conference Publication, p.p. 121–128. National Aeronautics and Space Administration, 1989.

[LCP06] W. Lu, K. Chiu ir Y. Pan. A parallel approach to XML parsing. *2006 7th IEEE/ACM International Conference on Grid Computing*, p.p. 223–230, 2006-09. DOI: 10.1109/ICGRID.2006.311019.

[LKN97] H. Loeper, A. Khattab ir P. Neubert. Concurrent objects in Ada 95. *Ada Lett.*, XVII(6):47–64, 1997-11. ISSN: 1094-3641. DOI: 10.1145/264934.264941. URL: https://doi.org/10.1145/264934.264941.

[Maz13] P. Mazurkiewicz. Streaming XML parser in C++. 2013-06. URL: https://www.codeproject.com/articles/587488/streaming-xml-parser-in-cplusplus. Tikrinta 2021-11-02.

[MVB+16] Andrius Merkys, Antanas Vaitkus, Justas Butkus, Mykolas Okulič-Kazarinas, Visvaldas Kairys ir Saulius Gražulis. *COD::CIF::Parser*: an error-correcting CIF parser for the Perl language. *Journal of Applied Crystallography*, 49(1):292–301, 2016-02. DOI: 10.1107/S1600576715022396. URL: http://dx.doi.org/10.1107/S1600576715022396.

[Nat97] National Research Council. *Ada and beyond: software policies for the Department of Defense*. The National Academies Press, Washington, DC, 1997. 101 p. ISBN: 978-0-309-05597-0. DOI: 10.17226/5463. URL: https://nap.nationalacademies.org/catalog/5463/ada-and-beyond-software-policies-for-the-department-of-defense.

[RT74] D. M. Ritchie ir K. Thompson. The Unix time-sharing system. *Commun. ACM*, 17(7):365–375, 1974. ISSN: 0001-0782. DOI: 10.1145/361011.361061.

[SGA+17] A. Sabané, Y. Guéhéneuc, V. Arnaoudova ir G. Antoniol. Fragile base-class problem, problem? *Empirical Software Engineering*, 22:2612–2657, 5, 2017. ISSN: 1573-7616. DOI: 10.1007/s10664-016-9448-2.

[SGH92] E. Schonberg, M. Gerhardt ir C. Hayden. A technical tour of Ada. *Communications of the ACM*, 35(11):43–52, 1992-11. DOI: 10.1145/138844.138846.

[TJB+11] J. L. Tokar, D. F. Jones, P. E. Black ir C. E. Dupilka. Software vulnerabilities precluded by Spark. *Proceedings of the 2011 ACM annual international conference on special interest group on the Ada programming language*, SIGAda '11, p.p. 39–46, Denver, Colorado, USA. Association for Computing Machinery, 2011. ISBN: 9781450310284. DOI: 10.1145/2070337.2070356. URL: https://doi.org/10.1145/2070337.2070356.

[UDR+16] E. Upton, J. Duntemann, R. Roberts, T. Mamtora ir B. Everard. *Learning computer architecture with Raspberry Pi*. John Wiley & Sons, Inc., 2016. 507 p. ISBN: 978-1-119-18394-5.

[VKM+21] N. Vasilakis, K. Kallas, K. Mamouras, A. Benetopoulos ir L. Cvetković. PaSh: light-touch data-parallel shell processing. *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, p.p. 49–66, Online Event, United Kingdom. ACM, 2021. ISBN: 9781450383349. DOI: 10.1145/3447786.3456228.