TLA$^+$ specifikacijų išskyrimas iš Elixir programos

Deividas Bražėnas, Karolis Petrauskas

Vilniaus universitetas, Matematikos ir informatikos fakultetas,
Informatikos institutas, Didlaukio g. 47, LT-08303, Vilnius
deividas.brazenas@mif.stud.vu.lt, karolis.petrauskas@mif.vu.lt

Santrauka. Šiame tyrime yra nagrinėjamas metodas, padedantis užtikrinti Elixir programos atitikimą programinės įrangos inžinieriaus kurtai TLA$^+$ specifikacijai. Kuriant metodą apibrėžtas vertimo taisyklių rinkinys, skirtas TLA$^+$ specifikacijų išskyrimui iš nuosekliosios išskirstyto Elixir algoritmo dalies. Naudojant sudarytas taisykles, buvo įgyvendintas vertimo įrankis, Elixir kodą paverčiantis į TLA$^+$ specifikaciją. Sugeneruotos specifikacijos teisingumas tikrinamas modelio tikrinimu ir tikslinimu, o teisingas vertimo įrankio veikimas užtikrinamas konvertuojant sugeneruotą specifikaciją atgal į Elixir kodą ir vykdant pirminės programos vienetų testus.

Raktiniai žodžiai: TLA$^+$, PlusCal, Elixir, Formalūs metodai, Išskirstytos sistemos

1 Įvadas

Atsirandant vis daugiau didelio masto išskirstytų sistemų, jas eksploatuojant dažnai kyla subtilios, tačiau rimtos problemos. Tai yra įprastas reiškinys, kadangi išskirstytos sistemos yra itin sudėtingos – net keletas sąveikaujančių agentų gali sukurti tūkstančius ar net milijonus unikalių sistemos būsenų. Tokiu mastu nėra įmanoma ištestuoti ar net žinoti apie kiekvieną galimą ribinę situaciją (angl. *edge case*). Būtent tai nutiko ir vienam didžiausių debesijos kompiuterijos paslaugų teikėjui *Amazon*, kai dėl lenktynių sąlygos (angl. *race condition*) Elastic Compute Cloud (EC2) paslauga nustojo veikti ir sukėlė dideles prastovas pagrindinėms interneto sistemoms [16].

Formali verifikacija yra vienas iš būdų, padedantis užtikrinti išskirstytų sistemų teisingumą. Programos formalusis verifikavimas yra matematinis įrodymas, kad ji daro tai, kas iš jos tikimasi. Formalaus verifikavimo metu yra sudaroma specifikacija, atitinkanti algoritmą, apibrėžiamos savybės, kurias algoritmas turi tenkinti ir įrodoma, kad specifikacija tenkina tas savybes. Toks tikrinimas leidžia užkirsti kelią įvairioms spragoms pasiekti vykdomąją aplinką, kai kita technika jų rasti būtų beveik neįmanoma [14].

DOI: https://doi.org/10.15388/LMITT.2023.1

Viena iš formalios verifikacijos kalbų, sėkmingai naudojama industrijoje, yra TLA$^+$, 1999 metais sukurta *Leslie Lamport* [13]. TLA$^+$ yra kalba, skirta programinės įrangos modeliavimui abstraktesniame lygyje, nei kodas, remiasi matematika ir nesisieja su jokia programavimo kalba [6].

Turint sistemos formaliąją specifikaciją, ją galimą patikrinti modelio tikrinimu (angl. *model checking*). Modelio tikrinimas yra kompiuterizuotas metodas dinaminių sistemų analizei, modeliuojant jų būsenų perėjimus (angl. *state transition*) [4] ir tikrinant gyvavimo (angl. *liveness*) ir saugos (angl. *safety*) savybes visose galimose baigtinės būsenų sistemos (angl. *finite-state*) būsenose [6]. TLA$^+$ specifikacijos gali būti patikrintos *TLC* įrankiu [5].

Kuriant sistemas atsiranda problema, jog modelio tikrinimui dažniausiai yra naudojamos programuotojų kurtos formaliosios specifikacijos, o kritinės klaidos gali būti padarytos programavimo metu. Gerai, kai šios klaidos yra aptinkamos per kodo peržiūrą (angl. *pull request*) ar testavimą, tačiau kartais jos pasiekia vykdomąją aplinką ir gali neigiamai paveikti sistemos naudotojų darbą. Formalios specifikacijos ir programos įgyvendinimo skirtumas gali būti sumažintas naudojant tam tikras programavimo kalbas, kurios leidžia formalią verifikaciją (*ADA* [1], *Spec#* [12]), generuoja kodą iš specifikacijų (*APTS* [9], *TLA*$^+$ [10]) arba išgauna specifikacijas iš kodo. Šiame tyrime nagrinėjame pastarąjį metodą, kadangi tai leidžia programinės įrangos inžinieriams sutelkti dėmesį į programavimą, o formaliuosius metodus naudoti kaip papildomą įrankį sistemos teisingumui užtikrinti.

Viena iš populiarių programavimo kalbų, dažnai naudojama išskirstytų sistemų kūrime, yra Elixir. Ji naudoja Erlang virtualiąją mašiną *BEAM*, žinomą dėl mažos delsos ir aukšto atsparumo klaidoms [17]. Be to, Elixir kalboje plačiai išvystytas metaprogramavimas (angl. *metaprogramming*), leidžiantis pasiekti ir manipuliuoti programos abstrakčiu sintaksės medžiu (AST) [11], kuris yra naudingas išgaunant TLA$^+$ specifikacijas iš kodo.

Kaip tarpinę kalbą Elixir kodo vertimui į formaliąją TLA$^+$ specifikaciją pasirinkome PlusCal. Tai yra išskirstytų algoritmų kūrimo kalba, kuri yra kompiliuojama į TLA$^+$, sukurta *Leslie Lamport*, norint padaryti formalius metodus patrauklesnius programinės įrangos inžinieriams [8]. Dėl panašumo į imperatyviąją programavimo kalbą ir aukštesnio abstrakcijos lygmens konstrukcijų (pvz. sąlyginių sakinių, rekursyvaus operatorių iškvietimo), kodo vertimo į specifikaciją taisyklės yra paprastesnės bei lengviau plečiamos. Dėl šių priežasčių PlusCal yra plačiai naudojama TLA$^+$ bendruomenėje [18].

2 Elixir vertimo į PlusCal taisyklės

Tyrimo metu apibrėžėme Elixir vertimo į PlusCal taisyklių rinkinį. Apibrėžėme tik dalį visų galimų taisyklių – jos pasirinktos taip, kad būtų įmanoma išversti eksperimente nagrinėjamą algoritmą į PlusCal ir padengti skirtingas, dažniausiai naudojamas Elixir kalbos konstrukcijas. Kitos, darbe neapibrėžtos, taisyklės turėtų būti kuriamos tokiu pat principu.

Iš viso darbe apibrėžėme 35 vertimo taisykles ir jas suskirstėme į šias grupes:

1. Įprastinių matematinių operatorių (suma, daugyba, dalyba, ir kt.) vertimas.
2. Sąlyginių operatorių (*if/else*, *cond, pin*) vertimas.
3. Funkcijų iškvietimo vertimas.
4. Standartinių Elixir tipų (skaičiaus, eilutės, žemėlapio, struktūros ir kt.) vertimas.
5. *Enumerables* modulio operacijų (*into*) vertimas.
6. *Map* modulio operacijų (*get*, *put, size,* dekonstravimo ir atnaujinimo) vertimas.
7. Reikšmės grąžinimo vertimas – PlusCal operatoriai nepalaiko reikšmės grąžinimo, tad grąžinama reikšmė turi būti išsaugota kintamajame.
8. Tuščios reikšmės priskyrimo vertimas – kai kuriems PlusCal tipams (pvz. žemėlapiui) negalima priskirti tuščios reikšmės, tad jiems priskiriamos tam tikros, konfigūracijoje apibrėžtos reikšmės.

Šios vertimo taisyklės apibrėžia, kaip Elixir kalbos abstraktaus sintaksės medžio elementai yra atvaizduojami į PlusCal kalbos konstrukcijas. Elixir kalbos AST elementas yra sudarytas iš trijų dalių – žymos (angl. *marker*), metaduomenų ir vaikinių elementų (angl. *children*).

Vertimo taisyklės viršutinėje dalyje, virš linijos, yra apibrėžta taisyklės įvestis – AST mazgas arba standartinio tipo kintamasis (pvz. skaičius, žemėlapis ir kt.). Taisyklės įvestis turi tenkinti kontekste (kairėje $\vdash$ simbolio pusėje) apibrėžtas išankstines sąlygas (angl. *preconditions*), atitikti šabloną (angl. *pattern matching*) ir tenkinti papildomas sąlygas, sujungtas $\wedge$ simboliu. Kintamieji vertimo taisyklėse yra pažymėti *pasviruoju* šriftu. Jiems rekursyviai bus taikomos vertimo taisyklės, iki kol bus gauta galutinė PlusCal išraiška. Reikšmės, kurios vertimo įvestyje yra ignoruojamos, yra pažymėtos _ simboliu. Po linija yra apibrėžta taisyklės išvestis PlusCal kalboje. Šalia linijos yra

pateiktas vertimo taisyklės pavadinimas. 1 pav. pateikiama *IF-ELSE* sąlyginio sakinio vertimo taisyklė, o 2 pav. – *ENUM-INTO* operacijos vertimo taisyklė.

$$\frac{\vdash \{:\text{if}, \_, [condition, [\text{do}: \{: \_\text{block}\_, [], do\_block\}, \text{else}: \{: \_\text{block}\_, [], else\_block\}]]\}}{\text{if } condition \text{ then } do\_block \text{ else } else\_block \text{ end if;}} \ (IF-ELSE)$$

1 pav. IF-ELSE sąlyginio sakinio vertimo taisyklė.

IF-ELSE sąlyginio sakinio vertimo taisyklėje šablono atitikimo būdu ieškomo abstrakčios sintaksės medžio elemento, kurio žyma būtų *:if* atomas (konstanta, kurios reikšmė yra jos vardas), o vaikiniuose elementuose būtų apibrėžta sąlyga *condition* bei *do* ir *else* Elixir kodo blokai, kuriems toliau bus taikomos vertimo taisyklės.

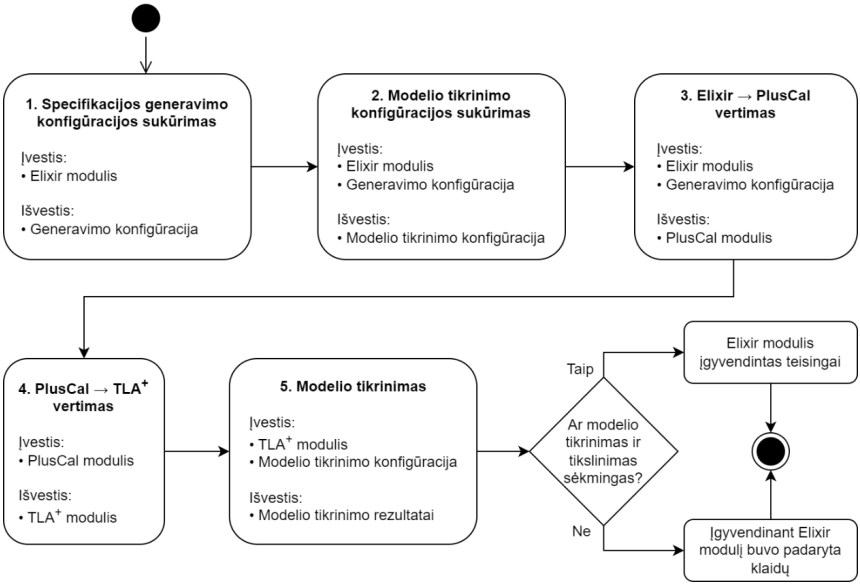
$$\frac{\vdash \begin{array}{l}\{:=, \_, [\{variable, \_, \text{nil}\}, \{\{:., \_, [\{: \_\text{aliases}\_, \_, [:\text{Enum}]\}, :\text{into}]\}, \_, \\ [\{enumerable, \_, \text{nil}\}, \{:\%\{\}, \_, []\}, \{:\text{fn}, \_, [\{:->, \_, [[iterator], assignment]\}]\}]\}]\}\end{array}}{variable \ := \ [iterator \ \backslash in \ enumerable \ |-> \ assignment]} \ (ENUM-INTO)$$

2 pav. ENUM-INTO operacijos vertimo taisyklė.

ENUM-INTO operacijos vertimo taisyklėje, Elixir *Enumerables* tipo operacija *Enum.into/3* yra paverčiama į PlusCal išraišką. Ši funkcija yra skirta elementų įterpimui iš vieno rinkinio į kitą, jiems pritaikant apibrėžtą transformaciją. PlusCal algoritmų kalboje gautas rinkinys atitinka struktūros (angl. *struct*) tipą, kuris atvaizduoja raktus į jiems priskirtas reikšmes. Taisyklės įvestyje galima matyti, jog Elixir operacijos *Enum.into/3* rezultatas priskiriamas kintamajam *variable*. Operacija turi tris argumentus – rinkinį *enumerable*, kuriuo yra iteruojama, tuščią žemėlapio tipo elementą, kuris atnaujinamas iteruojant ir transformacijos funkciją. Transformacijos funkcija yra sudaryta iš kintamojo *iterator* bei AST mazgo *assignment*, atliekančio priskyrimą. Priskyrime *assignment* gali būti naudojamas kintamasis *iterator*, tačiau vertimo taisyklėje tai nėra nurodoma, kadangi AST mazge *assignment*, kuriam ir toliau bus taikomos vertimo taisyklės, jis yra naudojamas kaip paprastas kintamasis. Taisyklės PlusCal išvestyje kintamajam *variable* priskiriama struktūra, naudojanti kintamuosius *iterator* ir *enumerable* bei į PlusCal išverstą priskyrimo išraišką *assignment*.

3 Elixir vertimo į TLA+ metodas

Elixir kodo vertimo į PlusCal kalbos konstrukcijas yra esminė kodo susiejimo su specifikacija dalis, tačiau jų taikymas turi būti integruotas į programų kūrimo procesą. Tyrimo metu sukūrėme metodą, nuoseklųjį Elixir programinį kodą išverčiantį į TLA$^+$ specifikaciją ir patikrinantį jos teisingumą. Metodo schema pateikta 3 pav.



3 pav. Elixir vertimo į TLA+ metodo schema.

Pasiūlytame metode pirmiausia apibrėžiama konfigūracija, skirta verčiamam Elixir moduliui. Joje saugoma informacija, kuri būtina specifikacijos išgavimui, bet negali būti išgauta iš kodo. Konfigūracijoje apibrėžiami verčiamų funkcijų pavadinimai, konstantos, kintamųjų pradinės reikšmės (kai kurie TLA$^+$ kintamųjų tipai neveikia teisingai modelio tikrinimo metu, kai jie neturi pradinių reikšmių [18]) bei savybių apibrėžimai.

Antrasis metodo žingsnis yra TLA$^+$ modelio tikrinimo konfigūracijos apibrėžimas. Joje yra informacija, reikalinga modelio tikrinimui, tokia kaip konstantos, invariantai (savybė, kuri turi būti teisinga kiekvienoje pasiekiamoje būsenoje) ir laiko savybės. Ši dalis negali būti išgauta iš kodo, kadangi žmogus turi nuspręsti, kaip sugeneruota specifikacija turi būti tikrinama.

Kai turime generavimo konfigūraciją, vykdome vertimą iš Elixir į PlusCal. Tam naudojame 2 skyriuje pateiktas vertimo taisykles.

Sugeneravus PlusCal specifikaciją, ji verčiama į TLA$^+$ specifikaciją, naudojant *SANY* įrankį, kuris yra standartinio TLA$^+$ įrankio dalis [7]. Po generavimo gauta TLA$^+$ specifikacija naudojama modelio tikrinimui ar savybių įrodymui, kaip ir programinės įrangos inžinieriaus kurta TLA$^+$ specifikacija. Remiantis modelio tikrinimo bei tikslinimo (angl. *refinement*) rezultatais, galima patikrinti, ar kodas įgyvendina abstrakčią specifikaciją ir veikia teisingai.

Modelio tikslinimo metu yra tikrinama ar sugeneruota specifikacija patikslina programuotojo kurtą abstrakčią specifikaciją ir turi jai būdingas savybes [3]. TLA$^+$ formaliuosiuose metoduose tikslinimas yra atliekamas apibrėžiant tikslinimo sąryšį (angl. *refinement map*) tarp abstrakčios ir konkrečios specifikacijų. Jis susieja sugeneruotos specifikacijos būsenas ir perėjimus su programinės įrangos inžinieriaus sudarytos specifikacijos būsenomis ir perėjimais. Jei modelio tikslinimas sėkmingas, sugeneruota specifikacija įgyvendina programuotojo kurtą abstrakčią specifikaciją.

4 Vertimo metodo teisingumas ir pilnumas

Vertimo metodo teisingumą parodėme dviem būdais – tikslinimu bei vertimo patikrinimu (angl. *translation validation*).

Tikrinant vertimą tikslinimo metodu, tikrinama ar sugeneruotoje specifikacijoje nėra vykdomi veiksmai, kurie nėra įmanomi programuotojo kurtoje specifikacijoje. Taip pat, tikrinamos laiko savybės, galinčios eliminuoti nieko nedarančias specifikacijos, kurios nėra teisingos (pvz. ar nėra generuojama specifikacija, kiekviename žingsnyje kurianti atvaizdą į visada teisingą pradinę būseną).

Naudojant vertimo patikrinimo metodą, sugeneruota TLA$^+$ specifikacija paverčiama atgal į Elixir kodą. PlusCal algoritmas, esantis sugeneruotoje TLA$^+$ specifikacijoje, yra naudojamas kaip įvestis vertimui atgal į Elixir. Vertimo metu PlusCal algoritmas yra konvertuojamas į nuosekliąją Elixir funkciją, naudojant vertimo taisykles grįstas šablono atitikimu. Taip gaunamas pilnai funkcionuojantis sugeneruotas Elixir modulis.

Turint sugeneruotą Elixir kodą, tikrinama, ar vertimo metu buvo išsaugotos jo kritinės veikimo savybės [15] (pvz. vertimo metu kodo komentarai yra ignoruojami, kadangi jie neturi įtakos algoritmo veikimui). Tam savo tyrime naudojame esamos kodo bazės funkcinius vienetų testus, užtikrinančius,

jog tiek pradinis, tiek sugeneruotas Elixir moduliai veikia identiškai. Šiais testais yra testuojamos atskiros algoritmo funkcijos, o ne visas algoritmas. Norint užtikrinti, jog testavimo tikrinamų funkcijų įvestis bei gauti rezultatai yra vienodi, pradinio ir sugeneruoto Elixir modulių funkcijos yra perduodamos į testo apibrėžimą kaip parametrai.

Vertimo taisyklių pilnumą apskaičiavome dviem būdais. Pirmajame pilnumo vertinimo būde bendrą įgyvendintų vertimo taisyklių skaičių padalijome iš visų galimų abstrakčios sintaksės medžio mazgų konstrukcijų naujausioje Elixir versijoje (v1.14.3)[1] skaičiaus. Gautas pilnumo procentas yra 47 %, kuris parodo, jog vertimo taisyklėmis yra padengta beveik pusė visų galimų AST mazgų konstrukcijų. Nors šis vertinimo būdas ir parodo standartinių AST konstrukcijų padengimą, nėra aišku, koks yra visos Elixir kalbos padengimas. Dėl šios priežasties pilnumą įvertinome ir antruoju būdu – bendrą vertimo taisyklių skaičių padalijus iš visų funkcijų, prieinamų naujausios Elixir versijos standartinėje bibliotekoje (v1.14.3)[2], skaičiaus. Apytikslis išverstų standartinių Elixir funkcijų procentas yra 3,4 %. Šis procentas yra gana mažas dėl to, kad Elixir yra gana plati kalba, kurioje kuriami ne tik išskirstytieji algoritmai, tačiau net ir toks mažas procentas leidžia išversti eksperimente naudotą išskirstytąjį algoritmą.

5 Eksperimentas

Siekiant patikrinti tyrimo metu sukurto metodo veikimą, atlikome eksperimentą. Eksperimentui naudojome Bracha patikimo transliavimo (angl. *reliable broadcast*) protokolą [2], kuris yra vienas iš svarbiausių asinchroninio modelio bizantiškiems gedimams atsparių (angl. *Byzantine fault tolerant*) protokolų.

Bracha patikimo transliavimo protokolas veikia esant tokiam sistemos modeliui: kai sistemoje *n* mazgų, iš kurių vienas yra *lyderis*, bendras sutarimas bus pasiektas, kai sistemoje yra mažiau nei *n/3* pažeistų mazgų [2].

Bracha patikimo transliavimo algoritmo pseudo kodas pateikiamas 4 pav. Protokolas yra suskirstytas į žingsnius, atitinkančius pranešimų tipus. Kiekviename žingsnyje mazgas laukia, kol gaus pakankamai pranešimų, leidžiančių išsiųsti kito tipo pranešimą ir išsiuntus jį pereina į kitą žingsnį. Protokolą sudaro šie žingsniai – *transliavimas* (4 pav. 2-3), *pasiūlymo pranešimo*

[1] https://hexdocs.pm/elixir/1.14.3/syntax-reference.html#the-elixir-ast
[2] https://hexdocs.pm/elixir/1.14.3

gavimas (4 pav. 5-6), *atgarsio pranešimo gavimas* (4 pav. 7-8), *pasiruošimo pranešimo gavimas* (4 pav. 9-10) ir *sutarimas* (4 pav. 11-12). Simboliu *t* yra žymimas pažeistų mazgų kiekis.

```
1.  // broadcaster node:
2.  input M
3.  send «PROPOSE, M» to all
4.  // all nodes:
5.  after receiving «PROPOSE, M» message from the broadcaster do
6.    send «ECHO, M» to all
7.  after receiving 2t + 1 «ECHO, M» messages and not having sent a READY message do
8.    send «READY, M» to all
9.  after receiving t + 1 «READY, M» messages and not having sent a READY message do
10.   send «READY, M» to all
11. after receiving 2t + 1 «READY, M» messages do
12.   output M
```

4 pav. Bracha patikimo transliavimo algoritmo pseudo kodas [2].

Eksperimento metu sukūrėme algoritmo TLA$^+$ specifikaciją, įgyvendinome jį Elixir programavimo kalba bei naudojantis tyrimo metu sukurtu metodu sugeneravome jo žingsnių TLA$^+$ specifikacijas. Sugeneruotas specifikacijas patikrinome modelio tikrinimu bei tikslinimu.

Sugeneruotas TLA$^+$ specifikacijas konvertavome atgal į Elixir kodą, kurį ištestavome pirminės programos vienetų testais. Vienetų testuose tikrinome Bracha protokolo atitikimą funkciniams reikalavimams (pvz. sulaukus reikalingo kiekio *READY* pranešimų, sistemoje yra pasiekiamas bendras sutarimas). Taip užtikrinome, jog tiek programuotojų sukurtas, tiek iš TLA$^+$ specifikacijos sugeneruotas Elixir moduliai veikia teisingai ir atitinka funkcinius reikalavimus.

Sėkmingi eksperimento metu atlikti patikrinimai parodo, jog patikimo transliavimo protokolas buvo teisingai įgyvendintas Elixir programavimo kalboje, o vertimo metodas išsaugo kritiškai svarbią algoritmo informaciją, reikalingą teisingam veikimui.

Išvados

Šio tyrimo metu sukūrėme metodą, leidžiantį patikrinti Elixir kodo atitikimą abstrakčiai, programinės įrangos inžinieriaus kurtai specifikacijai. Atlikę tyrimą nustatėme, jog iš nuosekliosios Elixir kodo dalies galima išgauti TLA$^+$

specifikacijas, o sugeneruotų specifikacijų teisingumą patikrinti taikant modelio tikrinimą, specifikacijų tikslinimą ir vertimo patikrinimą.

Tarpinės PlusCal kalbos naudojimas turi tam tikrų trūkumų, tokių kaip sudėtingumas įrodinėjant specifikacijas TLA$^+$ įrodymų sistema TLAPS [8] ar lėtesnis modelio tikrinimas [8]. Nepaisant to, PlusCal leidžia vertimo taisyklėms naudoti matematinę kalbą, kuri yra panaši į imperatyviąją programavimo kalbą. Dėl šios priežasties taisyklių rinkinio plėtimas yra paprastesnis bei patrauklesnis programinės įrangos inžinieriams.

Vieną kartą apibrėžus generavimo ir modelio tikrinimo konfigūracijas, automatinį TLA$^+$ specifikacijų generavimo ir modelio tikrinimo metodą galima įtraukti į sistemos kūrimo procesą. Tai gali padėti pastebėti nepageidaujamus algoritmo elgsenos pokyčius, kol jie dar nesukėlė problemų sistemos naudotojams.

Literatūra

[1] AdaCore & Thales (2020). Implementation Guidance for the Adoption of SPARK.

[2] Bracha, G. (1987). Asynchronous Byzantine agreement protocols. In Information and Computation, 75.2, 130–143.

[3] Cansell, D., Gibson J. P. & Mery, D. (2007). Refinement: A Constructive Approach to Formal Software Design for Secure e-voting Interface. In Electronic Notes in Theoretical Computer Science, 39–55.

[4] Clarke, E. M., Henzinger, T. A, Veith, H. & Bloem, R. (2018). Handbook of Model Checking. Springer.

[5] Yu, Y., Manolios, P. & Lamport, L. (1999). Model Checking TLA+ specifications. In Correct Hardware Design and Verification Methods, 1703, 54–66.

[6] Lamport, L, (2002). Specifying Systems. The TLA+ Language and Tools for Hardware and Software Engineers. Microsoft Research.

[7] Lamport, L. A PlusCal User's Manual. P Syntax. https://lamport.azurewebsites.net/tla/p-manual.pdf [žiūrėta 2023-04-11].

[8] Lamport, L. The PlusCal Algorithm Language. https://lamport.azurewebsites.net/pubs/pluscal.pdf [žiūrėta 2023-04-11].

[9] Leonard, E. I. & Heitmeyer, C. L. (2008). Automatic Program Generation from Formal Specifications using APTS. In Automatic Program Development, 93–113.

[10] Mafra, G. M. (2019). TLA+ Transmutation. https://github.com/bugarela/tla-transmutation [žiūrėta 2023-04-11].

[11] McCord, C. (2015). Metaprogramming Elixir. Write Less Code, Get More Done (and Have Fun!). The Pragmatic Bookshelf.

[12] Microsoft (2004). Spec#. https://www.microsoft.com/en-us/research/project/spec/ [žiūrėta 2023-04-11].

[13] Newcombe, C. (2014). Why Amazon Chose TLA+. In Abstract State Machines Alloy, B, TLA, VDM, and Z: 4th International Conference, 25–38.

[14] Newcombe, C., Rath. T., Zhang, F., Munteanu, B., Brooker, M. & Deardeuff, M. (2015). How Amazon Web Services Uses Formal Methods. In Communications of the ACM, 58, 66–73.
[15] Patterson, D. & Ahmed, A. (2019). The Next 700 Compiler Correctness Theorems (Functional Pearl). In Proceedings of the ACM on Programming Languages, 3, 85–114.
[16] AWS Team (2011). Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. https://aws.amazon.com/message/65648/ [žiūrėta 2023-04-11].
[17] Elixir Team. Elixir Home Page. https://elixir-lang.org/ [žiūrėta 2023-04-11].
[18] Wayne, H. (2018). Practical TLA+: Planning Driven Development. Apress Media.