VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS
FUNDAMENTINIŲ MOKSLŲ FAKULTETAS
INFORMACINIŲ SISTEMŲ KATEDRA

Karolis Geležauskas

„SALESFORCE“ VERSLO SISTEMOS VERSIJŲ KONTROLĖS UŽDAVINIŲ AUTOMATIZAVIMAS „GITHUB“ APLINKOJE
AUTOMATION OF SALESFORCE BUSINESS SYSTEM VERSION CONTROL PROCESSES ON THE GITHUB PLATFORM

Baigiamasis magistro darbas

Informacinių sistemų programų inžinerijos studijų programa
valstybinis kodas 6211BX017
Informatikos inžinerijos studijų kryptis

Vilnius, 2023

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS
FUNDAMENTINIŲ MOKSLŲ FAKULTETAS
INFORMACINIŲ SISTEMŲ KATEDRA

Karolis Geležauskas

„SALESFORCE“ VERSLO SISTEMOS VERSIJŲ KONTROLĖS UŽDAVINIŲ AUTOMATIZAVIMAS „GITHUB“ APLINKOJE
AUTOMATION OF SALESFORCE BUSINESS SYSTEM VERSION CONTROL PROCESSES ON THE GITHUB PLATFORM

Baigiamasis magistro darbas

Informacinių sistemų programų inžinerijos studijų programa
valstybinis kodas 6211BX017
Informatikos inžinerijos studijų kryptis

Vadovas dr. Algirdas Laukaitis
(Moksl. laipsnis/pedag. vardas, vardas, pavardė)

Lietuvių kalbos konsultantas dr. Vaida Buivydienė
(Moksl. laipsnis/pedag. vardas, vardas, pavardė)

Vilnius, 2023

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS
FUNDAMENTINIŲ MOKSLŲ FAKULTETAS
INFORMACINIŲ SISTEMŲ KATEDRA

Informatikos inžinerijos studijų kryptis

Informacinių sistemų programų inžinerijos studijų programa, valstybinis

kodas 6211BX017

Informacinių sistemų specializacija

MAGISTRO BAIGIAMOJO DARBO
UŽDUOTIS

…….....Nr. .....
Vilnius

Studentui Karoliui Geležauskui

Baigiamojo darbo tema: „Salesforce“ verslo sistemos versijų kontrolės uždavinių automatizavimas „GitHub“ aplinkoje

Baigiamojo darbo užbaigimo terminas 2023 m. gegužės 28 d.

BAIGIAMOJO DARBO UŽDUOTIS:

1. Ištirti probleminę sritį ir pasiūlyti versijų kontrolės uždavinių ir konfliktų sprendimo automatizavimo metodą;
2. Atlikti versijų kontrolės sistemų, versijų kontrolės uždavinių, jų automatizavimo būdų ir versijų kontrolės taikymo verslo sistemose analizę;
3. Suprojektuoti prototipą, atlikti versijų kontrolės uždavinių automatizavimo metodo eksperimentinį tyrimą ir įvertinti atlikto tyrimo rezultatus.

Baigiamojo darbo rengimo konsultantai: dr. Vaida Buivydienė
(Moksl. laipsnis/pedag. vardas, vardas, pavardė)

Vadovas dr. Algirdas Laukaitis
(Moksl. laipsnis/pedag. vardas, vardas, pavardė)

Užduotį gavau

Karolis Geležauskas
……..
(Vardas, pavardė)

……..…....
(Data)

Vilniaus Gedimino technikos universitetas
Fundamentinių mokslų fakultetas
Informacinių sistemų katedra

ISBN ISSN
Egz. sk. ……..
Data ….-….-….

Antrosios pakopos studijų **Informacinių sistemų programų inžinerijos** programos magistro baigiamasis darbas

Pavadinimas **„Salesforce“ verslo sistemos versijų kontrolės uždavinių automatizavimas „GitHub“ aplinkoje**

Autorius **Karolis Geležauskas**

Vadovas **dr. Algirdas Laukaitis**

Kalba: lietuvių

Anotacija

Baigiamajame magistro darbe nagrinėjamas „Salesforce“ verslo sistemos versijų kontrolės uždavinių automatizavimas „GitHub“ aplinkoje. Licencijuojamoms, uždarojo kodo verslo sistemoms nėra paprasta pritaikyti versijų kontrolę. Todėl yra svarbu sukurti metodą vykdyti versijų kontrolę verslo sistemai ir pritaikyti versijų kontrolės uždavinių automatizaciją.

Analitinėje dalyje išnagrinėtos versijų kontrolės sistemos, jų uždaviniai ir galimi automatizavimo būdai ir apžvelgiamas versijų kontrolės naudojimas „Salesforce“ verslo sistemoje.

Pagal probleminės srities analizę apibrėžiamas konkrečios verslo srities kūrimo procesas ir nustatomos galimos automatizacijos: automatinis pakeitimų perkėlimas tarp versijų kontrolės sistemos ir „Salesforce“ aplinkų, automatinis testavimas, automatinis testavimo užduočių kūrimas ir automatinis konfliktų sprendimas versijų kontrolės sistemoje. Šios automatizacijos įgyvendinamos „GitHub“ versijų kontrolės aplinkoje taikant „GitHub Actions“ platformą. Šioms automatizacijoms yra atliekami du eksperimentiniai tyrimai: kūrimo proceso automatizacijos tyrimas ir automatinio konfliktų sprendimo realizacijos tyrimas. Gauti rezultatai parodė, kad versijų kontrolės uždavinių automatizavimas sumažino laiko sąnaudas dviejų savaičių laikotarpyje 81,48 %, o automatinio konfliktų sprendimo realizacijos tikslumas yra 34,78 %.

Išnagrinėjus literatūrą, probleminę sritį, praktiškai pritaikius prototipą ir ištyrus jį eksperimentiškai pateikiamos baigiamojo darbo išvados.

Darbą sudaro 6 dalys: įvadas, literatūros analizė, siūlomo metodo aprašymas, prototipo aprašymas ir eksperimentinis tyrimas, išvados, literatūros sąrašas.

Darbo apimtis – 80 puslapiai teksto be priedų, 33 iliustracijų, 7 lentelės, 28 bibliografiniai šaltiniai.

Prasminiai žodžiai: Salesforce verslo sistema, versijų kontrolė, versijų kontrolės uždaviniai, versijų kontrolės uždavinių automatizavimas, versijų kontrolės konfliktai, automatinis versijų kontrolės konfliktų sprendimas

Vilnius Gediminas Technical University
Faculty of Fundamental Sciences
Department of Information Systems

ISBN ISSN
Copies No. ……..
Date ….-….-….

Master degree Studies **Information Systems Software Engineering** study programme Master Graduation Thesis

Title **Automation of Salesforce Business System Version Control Processes on the GitHub Platform**

Author **Karolis Geležauskas**

Academic supervisor **dr. Algirdas Laukaitis**

Thesis language: Lithuanian

Annotation

The master's thesis investigates the automation of version control processes for the Salesforce business system on the GitHub platform. It is not easy to apply version control to licensed, closed-code business systems. Therefore, it is important to develop a method to implement version control for a business system and to apply the automation of version control processes.

The literature analysis part of this study examines version control systems, the processes that use them and possible automation methods, and investigates the use of version control in a Salesforce business system.

Based on the problem domain analysis, the development process for a specific business system is defined and possible automations are identified: the automatic migration of changes between version control system and Salesforce environments, the automatic testing, the automatic creation of testing tasks and the automatic conflict solution in the version control system. These automations are implemented in the GitHub version control platform using GitHub Actions. Two experimental studies are applied to implementation of these automations: a study on the automation of the development process and a study on the implementation of automatic conflict solution. The results obtained show that the automation of the version control tasks reduced the time wastage over a two-week period 81,48 % and the accuracy of the automatic conflict solution is 34,78%.

After the literature review, the analysis of the problem domain, the practical application of the prototype and the experimental testing, the conclusions of the master's thesis are presented.

The thesis consists of 6 parts: introduction, literature analysis, description of the proposed method, description of the prototype and the experimental research, conclusion, references.

Thesis consists of 80 p. text without appendixes, 33 pictures, 7 tables and 28 bibliographical entries.

Keywords: Salesforce business system, version control, version control processes, automation of version control processes, CI/CD, version control conflicts, automating the solution of version control conflicts

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

	Karolis Geležauskas, 20163928	
	(Studento vardas ir pavardė, studento pažymėjimo Nr.)	
	Fundamentinių mokslų fakultetas	
	(Fakultetas)	
	Informacinių sistemų programų inžinerija, ISIfm-21	
	(Studijų programa, akademinė grupė)	

BAIGIAMOJO DARBO (PROJEKTO)
SĄŽININGUMO DEKLARACIJA

2023 m. gegužės 27 d.

Patvirtinu, kad mano baigiamasis darbas tema „Salesforce verslo sistemos versijų kontrolės uždavinių automatizavimas GitHub aplinkoje“ yra savarankiškai parašytas. Šiame darbe pateikta medžiaga nėra plagijuota. Tiesiogiai ar netiesiogiai panaudotos kitų šaltinių citatos pažymėtos literatūros nuorodose.

Mano darbo vadovas daktaras Algirdas Laukaitis.

Kitų asmenų indėlio į parengtą baigiamąjį darbą nėra. Jokių įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs (-usi).

			Karolis Geležauskas
	(Parašas)		(Vardas ir pavardė)

SUTIKIMAS DĖL ASMENS DUOMENŲ NAUDOJIMO

2023-05-28
(Data)

Šiuo sutikimu aš, Karolis Geležauskas (toliau – Duomenų subjektas) sutinku, kad Vilniaus Gedimino technikos universitetas, juridinio asmens kodas 111950243, adresas Saulėtekio al. 11, LT-10223 Vilnius (toliau – Duomenų valdytojas), tvarkytų mano asmens duomenis kitų studentų mokymosi tikslu. T. y. tvarkytų (*pažymėkite tinkamą*):

- ☐ vardą, pavardę, bakalauro baigiamąjį darbą;
- ☐ bakalauro baigiamąjį darbą, nenurodant vardo, pavardės;
- ☒ vardą, pavardę, magistro baigiamąjį darbą;
- ☐ magistro baigiamąjį darbą nenurodant vardo, pavardės.

Šiuo tikslu tvarkomų asmens duomenų Duomenų valdytojas neperduos jokiems tretiesiems asmenims, studentams su baigiamaisiais darbais bus leidžiama susipažinti vidinėje informacinėje sistemoje. Duomenų subjekto asmens duomenys šiuo tikslu bus naudojami ne ilgiau nei 5 metai.

Šiuo sutikimu Duomenų subjektas patvirtina, kad yra supažindintas su šiomis teisėmis:

- Susipažinti su savo duomenimis ir kaip jie yra tvarkomi (teisė susipažinti);
- Reikalauti ištaisyti arba, atsižvelgiant į asmens duomenų tvarkymo tikslus papildyti asmens neišsamius asmens duomenis (teisė ištaisyti);
- Savo duomenis sunaikinti arba sustabdyti savo duomenų tvarkymo veiksmus (išskyrus saugojimą) (teisė sunaikinti ir teisė „būti pamirštam“);
- Reikalauti, kad asmens duomenų valdytojas apribotų asmens duomenų tvarkymą (teisė apriboti);
- Teise į duomenų perkėlimą (teisė perkelti);
- Nesutikti, kad būtų tvarkomi asmens duomenys, kai šie duomenys tvarkomi ar ketinami tvarkyti kitais tikslais;
- Pateikti skundą Valstybinei duomenų apsaugos inspekcijai;

Duomenų subjektas turi teisę bet kuriuo metu atšaukti savo sutikimą.

Karolis Geležauskas K Geležauskas

[Duomenų subjekto vardas, pavardė, parašas]

Duomenų valdytojo rekvizitai:
Vilniaus Gedimino technikos universitetas
Juridinio asmens kodas: 111950243
Adresas: Saulėtekio al. 11, LT-10223 Vilnius
Tel. (8 5) 274 5030
Faks. (8 5) 270 0112
El. paštas: vgtu@vgtu.lt
PVM mokėtojo kodas: LT119502413
Duomenų apsaugos pareigūno tel. (8 5) 251 2191, el. paštas: dap@vgtu.lt

TURINYS

1. ĮVADAS 12

2. ANALITINĖ DALIS 15

2.1 Versijų kontrolės sistemos 15

2.2 Versijų kontrolės uždaviniai 20

2.3 Versijų kontrolės uždavinių automatizavimo būdai 22

2.4 Versijų kontrolės taikymas verslo sistemose 26

2.5 Analitinės dalies apibendrinimas ir pagrindiniai rezultatai 27

2.6 Analitinės dalies išvados 28

3. SIŪLOMAS BŪDAS ARBA METODAS 29

3.1 Probleminės srities analizė 29

3.1.1 Verslo sistemos kūrimo proceso dalykinėje srityje analizė 32

3.1.2 Verslo sistemos kūrimo proceso siūlomos automatizacijos 34

3.2 Sistemai keliami reikalavimai 35

3.2.1 Funkciniai reikalavimai 35

3.2.2 Nefunkciniai reikalavimai 36

3.3 Aukšto lygio užduočių diagrama 37

3.3.1 Užduočių aprašymai 38

3.4 Automatinis pakeitimų perkėlimas tarp versijų kontrolės sistemos ir skirtingų aplinkų 41

3.4.1 Automatinis pakeitimų perkėlimas iš versijų kontrolės sistemos į aplinkas 41

3.4.2 Automatinis pakeitimų perkėlimas iš aplinkų į versijų kontrolės sistemą 43

3.5 Automatinis testavimas 44

3.6 Automatinis testavimo užduočių kūrimas 45

3.7 Konfliktų sprendimas versijų kontrolės sistemoje 47

3.7.1 Konfliktų atsiradimas versijų kontrolės sistemose 47

3.7.2 Aukšto lygmens konfliktų automatinio sprendimo proceso modelis 49

3.7.3 Konfliktų sprendimo kategorijos ... 50
3.7.4 Konfliktų automatinių sprendimų sistemos veiksmų seka ... 51
3.8 Trečiojo skyriaus apibendrinimas ir pagrindiniai rezultatai ... 52
3.9 Trečiojo skyriaus išvados ... 53
4. PROJEKTINĖ DALIS ... 54
4.1 Prototipo architektūra ... 54
4.2 Projektuojamos sistemos realizacija ... 56
4.2.1 Automatinio pakeitimų perkėlimo į „Salesforce“ aplinkas realizacija ... 56
4.2.2 Automatinio pakeitimų perkėlimo į versijų kontrolės sistemą realizacija ... 58
4.2.3 Automatinio testavimo realizacija ... 59
4.2.4 Automatinio testavimo užduočių kūrimo realizacija ... 61
4.2.5 Automatinio konfliktų sprendimo realizacija ... 63
4.3 Prototipo tyrimas ... 68
4.3.1 Kūrimo proceso automatizacijos tyrimas ... 68
4.3.2 Automatinio konfliktų sprendimo realizacijos tyrimas ... 72
4.4 Ketvirtojo skyriaus apibendrinimas ir pagrindiniai rezultatai ... 75
4.5 Ketvirtojo skyriaus išvados ... 76
IŠVADOS ... 77
LITERATŪRA ... 78

ILIUSTRACIJŲ SĄRAŠAS

2.1 pav. Versijų kontrolės sistemos užduotys ... 16

2.2 pav. Centralizuotosios versijų kontrolės sistemos struktūra ... 17

2.3 pav. Paskirstytosios versijų kontrolės sistemos struktūra ... 18

2.4 pav. Genetinio programavimo pagrindinis ciklas ... 23

2.5 pav. Pateikto kodo abstrakčiosios sintaksės medis ... 25

2.6 pav. „Salesforce“ CRM architektūra ... 27

3.1 pav. Dalykinės srities kūrimo procesas ... 29

3.2 pav. „Salesforce“ programinės įrangos poveikis dalykinės srities kūrimo procesui ... 32

3.3 pav. Funkcionalumo įgyvendinimo būsenos ... 33

3.4 pav. Verslo sistemos kūrimo proceso automatizacijos informacinės sistemos užduotys ... 37

3.5 pav. Automatinio pakeitimų perkėlimo iš verslo sistemos į aplinkas procesas ... 42

3.6 pav. Automatinio pakeitimų perkėlimo iš aplinkų į atšakas procesas ... 44

3.7 pav. Automatinio testavimo į atitinkamas aplinkas procesas ... 45

3.8 pav. Automatinis techninio testavimo užduoties sukūrimo procesas ... 46

3.9 pav. Automatinis funkcinio testavimo užduoties sukūrimo procesas ... 47

3.10 pav. Konflikto atsiradimas versijų kontrolės sistemoje ... 48

3.11 pav. Konfliktų automatinių sprendimų procesas ... 49

3.12 pav. Konfliktų sprendimo naudojant konfliktų sprendimo sistemą sekos diagrama ... 52

4.1 pav. „Salesforce“ sistemos versijų kontrolės uždavinių automatizacijos „GitHub“ aplinkoje prototipo paketų diagrama ... 54

4.2 pav. Automatinės validacijos pseudokodas ... 57

4.3 pav. Automatinio pakeitimų perkėlimo pseudokodas ... 58

4.4 pav. Automatinio pakeitimų perkėlimo į „GitHub“ versijų kontrolės sistemą pseudokodas ... 59

4.5 pav. Automatinio testavimo pseudokodas ... 61

4.6 pav. Automatinės techninio testavimo užduoties kūrimo pseudokodas ... 62

4.7 pav. Automatinės funkcinio testavimo užduoties kūrimo pseudokodas ... 62

4.8 pav. Automatinio konfliktų sprendimo pseudokodas 63

4.9 pav. Konfliktų sprendimų sistemos klasių diagrama 65

4.10 pav. Konfliktuojančio failo turinys failo turinio papildymo metu 66

4.11 pav. Konfliktuojančio failo turinys kompleksinio konflikto metu; 67

4.12 pav. Konfliktų sprendimo nusiuntimo į kitą saugyklą pseudokodas 68

4.13 pav. Laiko sąnaudos kūrimo proceso veiksmuose 71

4.14 pav. Bendros laiko sąnaudos kūrimo proceso veiksmuose 72

4.15 pav. Kompleksinio konflikto supaprastintas atvejis 74

4.16 pav. Bendras konfliktų sprendimo tikslumas pagal kategoriją 75

LENTELIŲ SĄRAŠAS

2.1 lentelė. Centralizuotosios ir paskirstytosios versijų kontrolės sistemų modelių palyginimas 19

3.1 lentelė. Kūrimo proceso automatizacijos sistemos užduočių aprašymai 38

4.1 lentelė. Prototipo komponentų aprašymai 55

4.2 lentelė. „Salesforce“ kūrimo proceso automatizavimo apžvalga. 69

4.3 lentelė. Laiko sąnaudos kūrimo proceso veiksmuose 71

4.4 lentelė. Sistemoje aptikti ir automatiškai išspręsti konfliktai pagal kategoriją 73

4.5 lentelė. Automatinių konfliktų sprendimo realizacijos tikslumas 73

SANTRUMPOS

VCS – (angl, Version Control System) versijų kontrolės sistema;

CVCS – (angl. centralized Version Control System) centralizuotoji versijų kontrolės sistema;

DVCS – (angl. distributed Version Control System) paskirstytoji versijų kontrolės sistema;

CI – (angl. continuous integration) nuolatinė integracija;

CD – (angl. continuous delivery) nuolatinis tiekimas;

AST – (angl. Abstract Syntax Tree) abstrakčiosios sintaksės medis;

GP – genetinis programavimas;

UAT – (angl. user acceptance testing) naudotojo priėmimo testavimas.

1. ĮVADAS

Versijų kontrolės sistemos yra neatsiejama programinės įrangos kūrimo dalis. Kiekvienas kūrimo procesas tampa paprastesnis, kai galima sekti kūrimo istoriją. Versijų kontrolė palengvina bendradarbiavimą su komandos nariais ir padeda atstatyti sistemą po klaidingo kodo įkėlimo į sistemą. Tačiau susiduriama su problema, kad versijų kontrolės sistemose iškeliamų uždavinių sprendimas nėra idealus. Didelė dalis laiko yra skiriama konfliktams spręsti. Taip pat uždarojo kodo, perkamoms kaip produktas arba licencijuojamos debesų kompiuterijos verslo sistemoms, nėra paprasto būdo pačios verslo sistemos viduje spręsti versijų kontrolės uždavinius. Dėl to labai svarbu sukurti metodą, kuris gali automatizuoti tokios verslo sistemos versijų kontrolės uždavinius.

Tyrimo objektas

„Salesforce“ verslo sistemos versijų kontrolės uždavinių automatizavimas „GitHub“ aplinkoje.

Darbo tikslas

Pasiūlyti „Salesforce“ verslo sistemai versijų kontrolės uždavinių automatizaciją ir sukurti metodą automatiškai spręsti versijų kontrolės konfliktus „GitHub“ aplinkoje.

Darbo uždaviniai

1. Ištirti probleminę sritį ir pasiūlyti versijų kontrolės uždavinių ir konfliktų sprendimo automatizavimo metodą;
2. Atlikti versijų kontrolės sistemų, versijų kontrolės uždavinių, jų automatizavimo būdų ir versijų kontrolės taikymo verslo sistemose analizę;
3. Suprojektuoti prototipą ir atlikti versijų kontrolės uždavinių automatizavimo metodo eksperimentinį tyrimą ir įvertinti atlikto tyrimo rezultatus.

Temos naujumas

Uždarojo kodo, perkamoms kaip produktas arba licencijuojamos debesų kompiuterijos verslo sistemoms, nėra paprasto būdo verslo sistemos viduje spręsti versijų kontrolės uždavinius ir šių uždavinių sprendimas tokioms sistemoms yra mažai išanalizuotas. Taip pat versijų kontrolės sistemos konfliktų automatinis sprendimas nėra plačiai nagrinėta sritis ir failų jungimo algoritmų pritaikymas spręsti konfliktams nėra išsamiai ištirtas.

Temos aktualumas

Šiuo metu versijų kontrolės sistemos yra naudojamos beveik visuose programinės įrangos kūrimo projektuose. Tačiau licencijuojamoms, uždarojo kodo verslo sistemoms, tokioms kaip „Salesforce“, nėra sukurto metodo automatiškai spręsti versijų kontrolės uždavinius. Taip pat

automatinis konfliktų sprendimas nėra dažnai automatizuojamas ir jo sprendimas yra labai sudėtingas procesas. Dalis šių konfliktų gali būti išspręsti taikant modernius failų jungimo algoritmus.

Tyrimo metodika

Analitinėje darbo dalyje, atliekant mokslinės literatūros, susijusios su verslo sistemos versijų kontrolės uždavinių automatizacija, naudojami bibliotekinio tyrimo ir lyginamosios analizės metodai. Analizuojamos versijų kontrolės sistemos, jų uždaviniai, automatizacijos būdai ir egzistuojantys įrankiai tai įgyvendinti. Praktinei versijų kontrolės uždavinių automatizacijai buvo naudojami konstravimo ir eksperimentinės analizės metodai.

Mokslinė darbo vertė

Pasiūlytas ir įgyvendintas naujas metodas „Salesforce“ verslo sistemos versijų kontrolės uždaviniams automatizuoti „GitHub“ aplinkoje. Įgyvendintas metodas automatizuoti dalį versijų kontrolės sistemoje aptinkamų konfliktų.

Darbo rezultatai

1. Atlikus versijų kontrolės sistemų, versijų kontrolės uždavinių, jų automatizavimo būdų ir versijų kontrolės taikymo verslo sistemose analizę buvo nustatyta, kad egzistuoja daug įrankių versijų kontrolės uždaviniams automatizuoti, bet „Salesforce“ verslo sistemai konkretaus pritaikyto įrankio nėra. Taip pat versijų kontrolės uždavinių automatizacija nėra galima, jei sistemoje yra aptinkamas konfliktas failų jungimo metu, o konfliktų sprendimas nėra plačiai išnagrinėta tema.
2. Ištyrus probleminę sritį buvo nustatyta, kad „Salesforce“ verslo sistemai galima automatizuoti pakeitimų perkėlimą tarp versijų kontrolės sistemos ir „Salesforce“ aplinkų, testavimą, testavimo užduočių kūrimą ir konfliktų sprendimą. Taip pat nustatytos galimos konfliktų kategorijos, kurioms buvo pasiūlyti galimi sprendimai.
3. Įgyvendintas automatizacijų prototipas, kuriam atliktas eksperimentinis tyrimas ir nustatyta, kad kūrimo proceso laiko sąnaudos dviejų savaičių laikotarpyje sumažėjo 81,48 %, o automatinių konfliktų sprendimo realizacijos išsprendžiamų konfliktų tikslumas yra 34,78 %.

Darbo struktūra

Darbas sudarytas iš 6 skyrių: įvado, analitinės dalies, siūlomo metodo dalies, projektinės dalies, išvadų ir literatūros.

1. Įvade yra aprašoma nagrinėjama sritis, tyrimo objektas, suformuoti tikslai, uždaviniai. Pateikiamas temos naujumas, aktualumas, metodika ir mokslinė darbo vertė. Išskiriami darbo rezultatai.
2. Analitinės dalies skyriuje aprašomos egzistuojančios versijų kontrolės sistemos, ištiriami versijų kontrolės sistemų uždaviniai, jų automatizavimo būdai ir apžvelgiamas versijų kontrolės naudojimas „Salesforce“ verslo sistemose.
3. Siūlomo būdo arba metodo skyriuje pateikiamas konkrečios analizuojamos srities kūrimo procesas ir nustatomos galimos automatizacijos. Iškeliami kuriamai sistemai funkciniai ir nefunkciniai reikalavimai, sudaroma užduočių diagrama. Išskiriami keturi pagrindiniai procesai ir pateikiamos siūlomos automatizacijos.
4. Projektinės dalies skyriuje aprašomas sukurtas sistemos prototipas, pateikiamos automatinio pakeitimų perkėlimo tarp versijų kontrolės sistemos ir aplinkų, automatinio testavimo, automatinio testavimo užduočių kūrimo, automatinių konfliktų sprendimų realizacijos ir ištiriamas automatizacijų sutaupomas laikas ir išspręstų konfliktų tikslumas.

Darbo aprobacija

2023 m. balandžio 19 d. „Vilnius Tech“ universitete vykusioje 26-oje Lietuvos jaunųjų mokslininkų konferencijos sekcijoje „Informacinės sistemos“ baigiamojo magistro darbo tematika pristatytas pranešimas „Salesforce“ verslo sistemos versijų kontrolės uždavinių automatizavimas „GitHub“ aplinkoje.

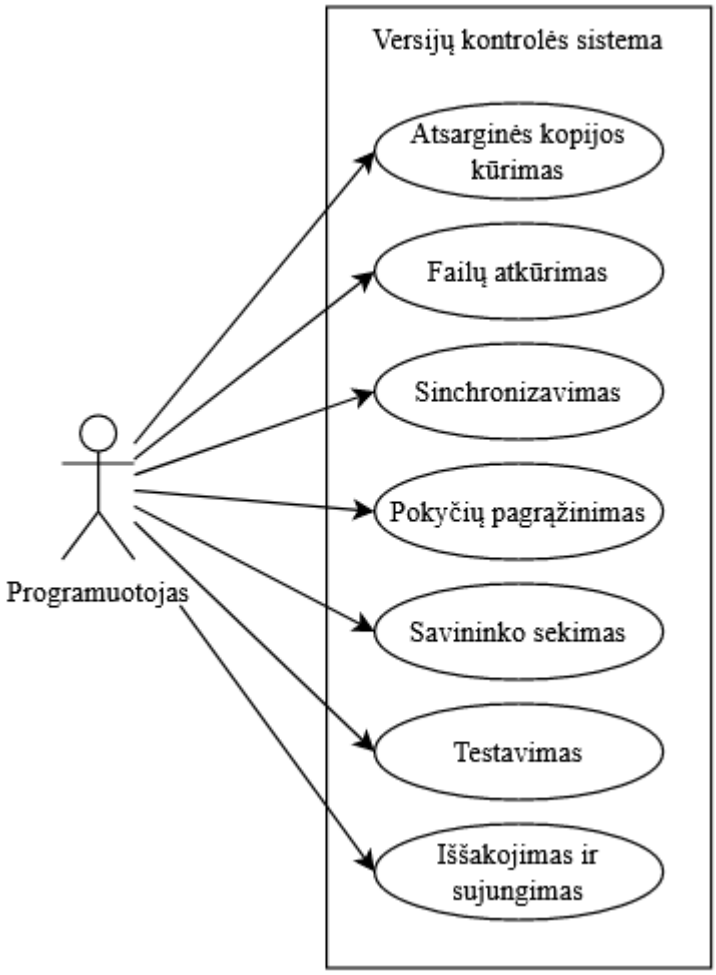
2. ANALITINĖ DALIS

Šioje baigiamojo darbo dalyje apžvelgiama versijų kontrolės sistemos, nustatomi uždaviniai, jų galimi automatizavimo būdai ir versijų kontrolės taikymas „Salesforce“ verslo sistemoje. Versijų kontrolės sistemų apžvalgoje nustatoma versijų kontrolės sistemų esmė, šių sistemų funkcijos ir egzistuojantys modeliai. Versijų kontrolės uždavinių apžvalgoje nustatomi pagrindiniai versijų kontrolės uždaviniai, jų panaudojimas ir privalumai. Versijų kontrolės uždavinių automatizavimo apžvalgoje nustatomi galimi automatizavimo metodai versijų kontrolės sistemoms, ištiriami įrankiai ir nustatomi galimi metodai spręsti uždaviniams. Versijų kontrolės taikymo verslo sistemose apžvalgoje tiriamos „Salesforce“ taikomos versijų kontrolės sistemos, su kokiais apribojimais susiduriama ir kokie metodai egzistuoja pritaikyti versijų kontrolę.

2.1 Versijų kontrolės sistemos

Versijų kontrolė yra procesas, kurio metu yra valdomos informacijos skirtingos versijos. Paprasčiausia versijų kontrolės forma yra naudojama kiekvieno žmogaus. Tai yra kiekvieno dokumento modifikavimas, saugojimas nauju pavadinimu, prirašant skaičių, kad būtų atskiriama versija. Tačiau rankinis daugelio dokumentų versijų valdymas linkęs į klaidas, dėl to versijų kontrolės sistemos palengvina ir paspartina šį procesą (O’Sullivan, 2009).

Versijų kontrolės sistemos valdo vystomo objekto kūrimą. Vystomas projektas yra įkeliamas į versijų kontrolės sistemą. Po to programinės įrangos kūrėjas privalo išsitraukti projekto versiją į savo darbinę aplinką, kad būtų galima dirbti prie projekto ir daryti kodo pakeitimus. Kai pakeitimai yra tenkinami, jie gali būti užfiksuojami į saugyklą paaiškinant, kas buvo pakeista. Toliau programinės įrangos kūrėjas privalo sinchronizuoti savo asmeninę versiją su kitų komandos narių pakeitimais, kad nebūtų prarandami kitų naudotojų pakeitimai. Galiausiai, išleidimo pavadinimas bus pažymėtas, kad būtų išleidžiama nauja versija (Zolkifli ir kt., 2018). Pagrindinės versijų kontrolės sistemos užduotys pateiktos 1 paveiksle.



2.1 pav. Versijų kontrolės sistemos užduotys (Ruparelia, 2010)

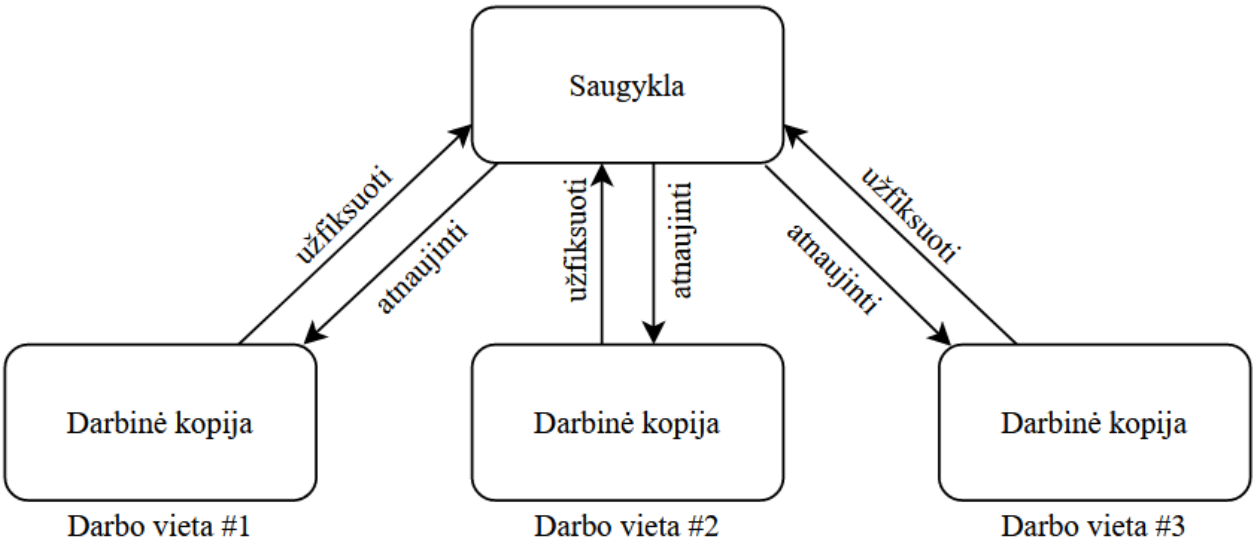
Šių užduočių detalesnė informacija pateikta toliau (Ruparelia, 2010):

1. Atsarginės kopijos kūrimas ir failų atkūrimas. Tai leidžia išsaugoti failus, kai jie yra keičiami ir suteikia galimybę grįžti į ankstesnę versiją.
2. Sinchronizavimas. Tai leidžia dalintis modifikuojamais failas ir atnaujinti kodą į naujausią versiją.
3. Pokyčių pagrąžinimas. Nenorimi kodo pakeitimai gali būti grąžinami į senesnę užfiksuotą versiją.
4. Pokyčių sekimas. Kai failai yra atnaujinami, galima palikti reikalingą informaciją, kurioje paaiškinama, kodėl įvyko pakeitimas. Tai leidžia matyti, kaip ir kodėl keičiasi kodas.
5. Savininko sekimas. Tai leidžia sekti, kas atliko failų pokyčius.
6. Testavimas. Galima atlikti trumpalaikius pokyčius izoliuotoje aplinkoje, juos ištestuoti ir patikrinti prieš užregistruojant pakeitimus.
7. Iššakojimas ir sujungimas. Kaip ir testavime, tai yra kodo iššakojimas į atskirą aplinką ir ją modifikuoti izoliuotai, pakeitimai yra sekami atskirai. Vėliau šie iššakojimai sujungiami į originalią saugyklą.

Nors visos versijos kontrolės sistemos atlieka tokias pat pagrindines funkcijas, bet jos yra kuriamos dviejų modelių (Otte, 2009):

1. centralizuotasis modelis, kurį naudoja centralizuotosios versijų kontrolės sistemos (CVCS);
2. paskirstytasis modelis, kurį naudoja paskirstytosios versijų kontrolės sistemos (DVCS).

Centralizuotasis modelis buvo pats populiariausias būdas valdyti failų versijas. Centralizuotosios versijų kontrolės sistemos struktūra pateikta 2 paveiksle.



2.2 pav. Centralizuotosios versijų kontrolės sistemos struktūra (Lubański, 2019b)

Kaip matoma 2 paveiksle, centralizuotoji versijų kontrolės sistema turi vieną pagrindinę saugyklą, kuri iš esmės yra serveris (De Alwis & Sillito, 2009). Visi failai yra saugomi saugykloje ir visi turi prieigą prie jų savo kompiuteriuose. Visi programinės įrangos kūrėjai vykdo pakeitimus savo darbinėje kopijoje. Kai failai saugomi sistemoje ir programinės įrangos kūrėjas užfiksuoja pakeitimus, tada visi pakeitimai yra pasidalinami su kitais programinės įrangos kūrėjais, kurie gali atnaujinti savo darbines kopijas. Tačiau rašymo prieiga į saugyklą yra suteikiama tik keliems programinės įrangos kūrėjams – aktyviems naudotojams (De Alwis & Sillito, 2009). Visi kiti naudotojai yra priskiriami prie pasyvių naudotojų, kurie turi tik peržiūros ir savo darbinės kopijos atnaujinimo funkcijas (Zolkifli ir kt., 2018). Pagrindiniai centralizuotosios versijų kontrolės sistemos privalumai (Lubański, 2019b; Malmsten, 2010):

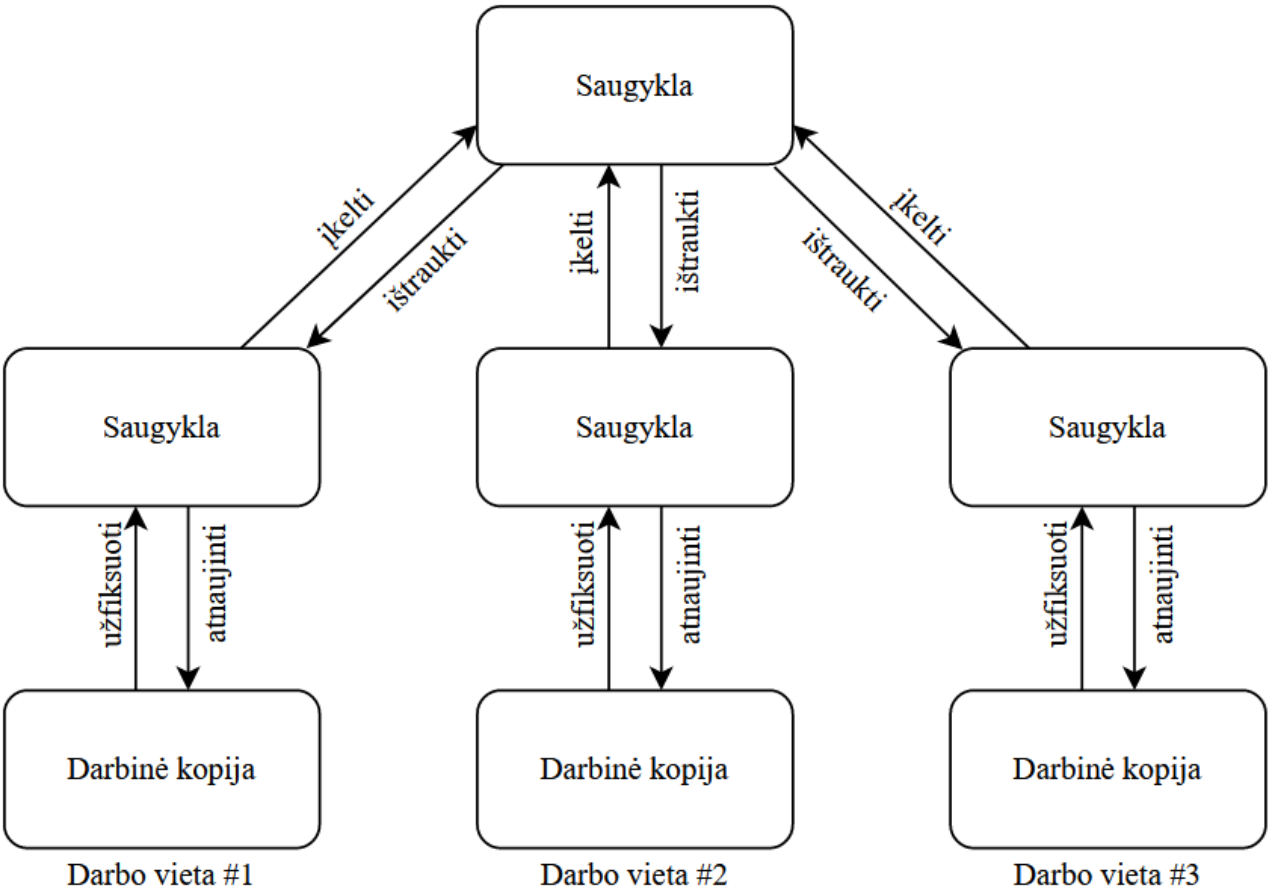
- centralizuotąją sistemą paprasta suprasti;
- galima suteikti prieigą kataloginiu lygiu;
- veikia greičiau su binariniais failais;
- privaloma peržiūrėti kitų darbus;
- tik viena saugykla, kurią reikia sekti;

- saugykla neleidžia konfliktų.

Tačiau egzistuoja ir trūkumai, dėl kurių šiuo metu vyrauja paskirstytosios sistemos (Otte, 2009):

- nėra galimybės dirbti neprisijungus prie tinklo;
- tinklas visada yra duomenų kamštis;
- ribotos šakojimo ir sujungimo galimybės;
- naudotojai turi būti skirstomi į pasyvius ir aktyvius.

Šiuo metu paskirstytasis versijų kontrolės modelis yra populiariausias būdas valdyti projektų versijų kontrolę (Brindescu ir kt., 2014). Kai projekto kūrimui naudojamas centralizuotasis modelis, tai reiškia, kad visa projekto istorija yra saugoma vienoje vietoje ir programinės įrangos kūrėjai neturi galimybės dirbti kartu naudodamiesi vietine versijų kontrole. Jei naudotojas neturi prieigos prie serverio, tada jis nebegali atlikti jokių versijų kontrolės funkcijų (Zolkifli ir kt., 2018). Tai yra pagrindinė priežastis, dėl ko dauguma projektų valdomi paskirstytosios versijų kontrolės sistemos modeliu. Paskirstytosios versijų kontrolės sistemos struktūra yra pateikta 3 paveiksle.



2.3 pav. Paskirstytosios versijų kontrolės sistemos struktūra (Lubański, 2019b)

Kaip matoma 3 paveiksle, paskirstytoji sistema yra sukurta taip, kad kiekviena lokali darbo vieta turi savo atskirą lokalią saugyklą. Kai naudotojas atlieka pakeitimus, jis gali sinchronizuoti

lokalią saugyklą su serveriu ir taip pasidalinti pakeitimais su komanda (Somasundaram, 2013). Naudotojai gali pasiimti pakeitimus iš kitų saugyklų ir nereikia jų ištraukti iš centrinio serverio. Sinchronizavimas gali būti vykdomas tarp visų naudotojų. Jie išsirenka, kokie pakeitimai turi būti paliekami. Naudotojai ne tik mato paskutinę failų versiją, bet jie gali nukopijuoti saugyklą. Jeigu serveris tampa neprieinamas, tada bet kuri naudotojo saugykla gali būti naudojama atkurti serveriui. Kiekviena lokali saugykla yra atsarginė kopija (Zolkifli ir kt., 2018). Pagrindiniai paskirstytosios versijų kontrolės sistemos privalumai (Lionetti, 2012):

- versijų kontrolės sistemos veiksmai, išskyrus įkėlimą ir ištraukimą iš pagrindinės saugyklos – serverio, yra labai greiti, nes sistemai reikia prieigos tik prie lokalių duomenų;
- informacijos iškėlimas į saugyklą gali būti atliekamas be interneto;
- kiekvienas naudotojas turi pilną projekto kopiją ir gali ja dalintis su kitais naudotojais.

Tačiau paskirstytosios versijų kontrolės sistemos turi ir trūkumų (Lionetti, 2012; Malmsten, 2010):

- jei projektas yra didelis, versijų kontrolės sistemos istorija gali užimti daug vietos ir gali ilgai užtrukti ją parsisiųsti;
- jei projektas susideda iš daug binarinių failų, kurie negali būti suspausti, tada visos projekto versijos gali užimti daug vietos;
- naudotojų atliekamas darbas gali būti nematomas, nes neprivalo sukelti informacijos į saugyklą;
- daug saugyklų sunkina projekto sekimą.

Nors abu modeliai turi tokias pačias pagrindines funkcijas, bet reikia atsižvelgi į prieš tai pateiktus kiekvieno modelio privalumus ir trūkumus ir pagal kūrimo procesą pasirinkti tinkamą modelį. Pagrindiniai skirtumai tarp centralizuotosios versijų kontrolės sistemos ir paskirstytosios versijų kontrolės sistemos yra pateikti 1 lentelėje (Zolkifli ir kt., 2018).

2.1 lentelė. Centralizuotosios ir paskirstytosios versijų kontrolės sistemų modelių palyginimas

Versijų kontrolės sistema	**Saugykla**	**Saugyklos prieiga**	**Programų pavyzdžiai**	**Kūrimo proceso charakteristikos**
Centralizuotoji versijų kontrolės sistema	Egzistuoja tik viena centrinė saugykla – serveris	Kiekvienas naudotojas, kuriam reikalinga prieiga prie saugyklos, turi būti prisijungęs prie tinko.	• CVS; • Subversion; • Perforce Revision Control System	• Programinės įrangos pakeitimai projektams vykdomi tik kelių naudotojų; • Komanda yra vienoje vietoje

		Skaityti gali visi, bet rašyti tik tam tikri paskirti naudotojai		
Paskirstytoji versijų kontrolės sistema	Kiekvienas naudotojas turi atskirą pilną vietinę saugyklą kompiuteryje	Kiekvienas naudotojas gali dirti neprisijungęs prie tinko, bet kiekvienam naudotojui reikia dalintis savo saugykla su kitais naudotojais	• Git; • Mercurial; • Bazaar; • BitKeeper;	• Paskirstytoji sistema tinkama vienam ar daugiau programinės įrangos kūrėjui, nes projekto saugykla yra paskirstyta tarp visų kūrėjų; • Gali būti naudojama prie mažų ir prie didelių projektų, nes lengva parastam naudotojui prisidėti prie kūrimo proceso; • Komanda gali būti keliose vietose

Versijų kontrolės sistemos yra svarbi programinės įrangos kūrimo proceso dalis ir reikia suprasti, kas yra versijų kontrolės valdymo sistema ir kokio tipo modelis yra tinkamas. Egzistuoja du sistemų modeliai ir kiekvienas turi savo privalumus ir trūkumus. Priklausomai nuo kūrimo proceso reikia pasirinkti tinkamą modelį, nes jie skirtingai paveikia programinės įrangos kūrimo procesą.

2.2 Versijų kontrolės uždaviniai

Versijų kontrolės sistemos atlieka daug funkcijų įvairių sistemų priežiūroje ir kūrimo procese. Šios sistemos yra viena iš pagrindinių nuolatinio sistemos integravimo dalių. Šio proceso funkcijos (Shahin ir kt., 2017):

- sumažinti kūrimo proceso ir testavimo trukmę;
- padidinti kūrimo ir testavimo rezultatų matomumą;
- palaikyti pusiau automatinį nuolatinį testavimą;
- aptikti pažeidimus, trūkumus ir klaidas;
- atsižvelgti į saugumo ir plečiamumo problemas kūrimo procese;
- padidinti kūrimo proceso patikimumą.

Versijų kontrolės sistemos suteikia didelį kiekį įrankių ir funkcionalumų, kad būtų valdomas kuriamos sistemos kūrimo procesas. Viena iš priežasčių naudotis versijų kontrolės sistema yra tai, kad keli žmonės gali modifikuoti to pačio failo turinį vienu metu. Jei keli naudotojai pakeičia tą patį failą, sistema sujungia pakeitimus į vieną arba perspėja dėl konfliktų (Spinellis, 2005). Taigi naudotojai gali laisvai dirbti ties tuo pačiu projektu. Jei informacija keičiama, paskutinė failo versija ar visas projektas visada bus versijų kontrolės sistemoje.

Tačiau, jei automatinis failo turinio sujungimas nėra įmanomas, nes pakeitimai vykdomi toje pačioje failo vietoje, tada atsiranda failo turinio konfliktas. Konfliktų atsiradimas naudojantis versijų kontrolės sistemomis vidutiniškai įvyksta tarp 7,6 ir 19,3 % visų kėlimų metu (Kasi & Sarma, 2013). Kai įvyksta sujungimo konfliktai, jie sutrukdo kūrimo procesą, ypač kai pakeitimai daug skiriasi tarp versijų. Tai atsitinka todėl, kad naudotojas privalo sustabdyti kūrimo procesą ir išspręsti sujungimo konfliktą. Dėl to naudotojas susiduria su tokiais trikdžiais (Brindescu ir kt., 2020):

1. sutrikdoma naudotojo darbų eigą;
2. naudotojas turi suprasti, kokie pakeitimai yra vykdomi kito programinės įrangos kūrėjo;
3. naudotojas reikalaujamas daryti kodo pakeitimus, kurie gali sukurti naujas klaidas ypač jei tai atlieka pradedantysis naudotojas.

Taigi konfliktų sprendimai reikalauja daug pastangų iš naudotojo, o tai dažnai sukelia produktyvumo smukimą.

Kadangi versijų kontrolės sistemos reikalauja įvesti aprašymą kiekvieną kartą, kai išsaugoma nauja projekto versija, jos palengvina suprasti, kokie projekto pakeitimai vykdomi (Lubański, 2019a). Taip pat, jei tai yra kodas arba tekstinis failas, tai dauguma versijų kontrolės sistemų suteikia galimybę matyti, kuriose vietose buvo vykdomi pakeitimai. Taigi kitas svarbus versijų kontrolės sistemos uždavinys yra pakeitimų dokumentavimas.

Kita nauda naudojantis versijų kontrolės sistemas yra galimybė gražinti keičiamą informaciją į ankstesnę versiją. Dėl to naudotojai visada gali grįžti į specifinę failo versiją. Dėl šios priežasties versijų kontrolės sistemos apsaugo naudotojus nuo netyčinių ištrynimų ar pakeitimų. Pavyzdžiui, naudojantis versijų kontrolės sistema galima pagrįžti į visiškai veikiančią senesnę projekto versiją ir palyginti pakeitimus su naujausia versija, tokiu būdu nustatant tam tikras problemas. Taigi versijų kontrolės sistemų saugyklos laiko savyje daug naudingos informacijos, kuri gali būti panaudota kūrimo procesuose (Greene & Fischer, 2015). Dėl to svarbu versijų kontrolės sistemoms kaupti visą projekto istoriją saugykloje ir turėti galimybę naudotojams peržiūrėti, palyginti visus pakeitimus.

Nors prieš tai minėti privalumai lengvina ir spartina kūrimo procesą, tačiau susiduriama su problema, kad šis pranašumas reikalauja nemažai laiko. Kiekvieno pakeitimo aprašymas, versijų saugojimas, konfliktų sprendimas reikalauja supratimo ir pastangų iš naudotojo. Taip pat versijų kontrolės sistemos neapsaugo projektų, kuriuose naudojamos techninės priemonės yra atnaujinamos ir sugadina tam tikras funkcijas. Šiuo metu nėra paprasto būdo automatizuoti šiuos uždavinius, tačiau egzistuoja tam tikri metodai, kurie naudojami kitokio tipo sistemose panašius uždavinius vykdyti.

2.3 Versijų kontrolės uždavinių automatizavimo būdai

Nors versijų kontrolės sistemos dar neturi įrankio automatizuoti visus susijusius uždavinius, tačiau egzistuoja metodai ir priemonės, kurios supaprastina ar pagreitina versijų kontrolės sistemas. Galimi įrankiai versijų kontrolės sistemų automatizavimui (Henschel & Di Francesco, 2020; Virtanen, 2021):

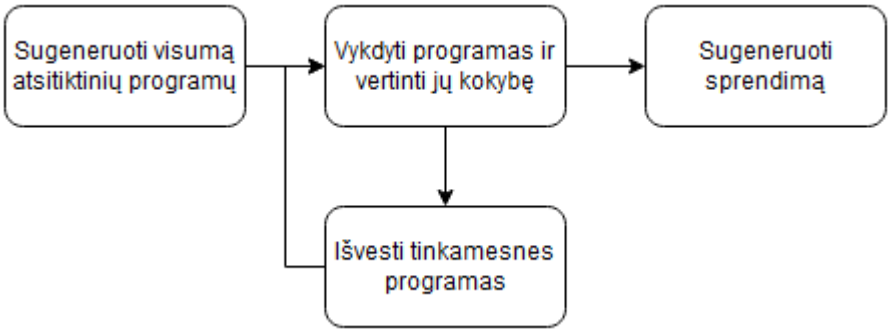
- „Jenkins“ – atviro kodo automatizavimo serveris, kuris yra labai lengvai išplečiamas ir pritaikomas. Turi didelę bendruomenę ir plačią įskiepių ir integracijų biblioteką, todėl ją galima lengvai pritaikyti daugeliui versijų kontrolės sistemų. Šis įrankis taip pat palaiko daug operacinių sistemų ir programavimo kalbų. Tačiau „Jenkins“ įdiegimas reikalauja daug konfigūracijos ir sąrankos.
- „Travis CI“ – debesijos pagrindo CI/CD platforma, kuri integruojasi su „GitHub“ ir „Bitbucket“ versijų kontrolės aplinkomis. Lengva įdiegti, naudotis ir turi paprastą „YAML“ tipo konfigūracijos failą. Palaiko didelį kiekį programavimo kalbų ir karkasų. Įrankis daug išplečiamas ir gali palaikyti didelio dydžio failus. Tačiau „Travis CI“ nėra toks greitas kaip kiti įrankiai ir jame negalima paskirstyti resursų.
- „CircleCI“ – debesijos pagrindo CI/CD platforma, kuri taip pat integruojasi su „GitHub“ ir „Bitbucket“ versijų kontrolės aplinkomis. Naudoja „YAML“ tipo konfigūracijos failą ir šį įrankį paprasta įdiegti. Palaiko daug programavimo kalbų ir karkasų ir įrankis lengvai išplečiamas. Tačiau ribojamas programų sukūrimo kiekis nebent perkami papildomi resursai.
- „GitLab CI/CD“ – integruotas įrankis į „GitLab“ versijų kontrolės aplinką. Palaiko „Docker“ ir „Kubernetes“, todėl lengva įdėti automatizacijas į konteinerius ir kelti programas. Palaiko daug programavimų kalbų ir karkasų. Tačiau įrankiu galima naudotis tik „GitLab“ versijų kontrolės aplinkoje.
- „GitHub Actions“ – yra plati ir lanksti platforma įdiegta į „GitHub“ versijų kontrolės aplinką. Įrankis integruotas su „GitHub“ ir tai suteikia paprastą darbo eigą kodo valdyme, nuolatiniame integravime ir diegime. Lengva įdiegti, naudotis ir turi „YAML“ tipo konfigūracijos failą. Palaiko didelį kiekį programavimo kalbų ir karkasų. Tačiau įrankiu galima naudotis tik „GitHub“ versijų kontrolės aplinkoje.

Visi prieš tai paminėti įrankiai įgalina automatizuoti daugelį versijų kontrolės uždavinių, tačiau šie įrankiai negali pradėti darbo, jei pačioje versijų kontrolės aplinkoje failų jungimo metu yra aptinkamas failų turinio jungimo konfliktas. Dalis šių konfliktų atsiranda dėl „GIT“ naudojamo „diff3“ failo jungimo algoritmo. „diff3“ algoritmas sujungia pakeitimus iš dviejų modifikuotų failo versijų ir pradinės versijos. Taigi algoritmas lygina tris failų versijas ir pateikia failą su visais

pakeitimais ir įspėjimais dėl konfliktų (*GNU Software Manual*, 2021). Tačiau „diff3“ algoritmas negali išspręsti konfliktų, kurie atsiranda toje pačioje failo vietoje. Dėl to šis algoritmas turi ribotą panaudojimą. Vidutiniškai apie 24,3% - 26,2% visų konfliktų sprendimų nereikalauja įterpti jokio naujo turinio į failą, reikia tik pasirinkti tinkamą sprendimą konflikto vietoje (Ji ir kt., 2020). Dėl šios priežasties galima tam tikriems konfliktų sprendimams taikyti kitokius algoritmus ar metodus.

Norint automatizuoti konfliktų sprendimą versijų kontrolės sistemose, gali reikėti taikyti automatinį programavimą. Automatinis programavimas – mechanizmas sugeneruoja programinį kodą pagal atitinkamas sąlygas (O'Neill & Spector, 2020). Nors automatinis kodo rašymas nėra toks ištobulintas, kad jo taikymas parašytų visą programinį kodą, tačiau panaudojus tam tikrus automatinio programavimo metodus galima automatizuoti sistemų kodą siauroje srityje.

Vienas iš būdų automatizuoti kodą yra genetinio kodo naudojimas. Genetinis programavimas (GP) – tai metodas skirtas kompiuteriui automatiškai išspręsti aukštu lygiu apibrėžtą problemą (Burke & Kendall, 2014). Genetinis programavimas transformuoja programų visumą į naują programų kartą pritaikant analogiškas funkcijas natūraliems genetiniams veiksmams. Šis procesas pavaizduotas 4 paveiksle.



2.4 pav. Genetinio programavimo pagrindinis ciklas

Taikydami genetinį programavimą ne su generuojamomis programomis, bet su programos skirtingomis versijomis ir atrenkant tinkamumo sąlygas pagal konfliktų pašalinimą galima automatizuoti kai kuriuos versijų kontrolės sistemos konfliktus.

Galimas sprendimas konfliktų sprendimui – duomenimis grindžiamas mašininis mokymas (Dinella ir kt., 2021). Neuroninis tinklas yra apmokomas konfliktuojančio failo pradine versija ir abejomis modifikuotomis failo versijomis. Pagal šiuos duomenis sistema gali išspręsti konfliktus, kurie atsiranda failo turinio viduje ir atrinkti tinkamus failo turinio elementus. Tačiau šiam sprendimui egzistuoja nemažai trūkumų:

- nėra išsaugota pakankamai konfliktų sprendimui reikiamų duomenų, kad neuroninis tinklas būtų išmokytas pakankamai tiksliai spręsti konfliktus;

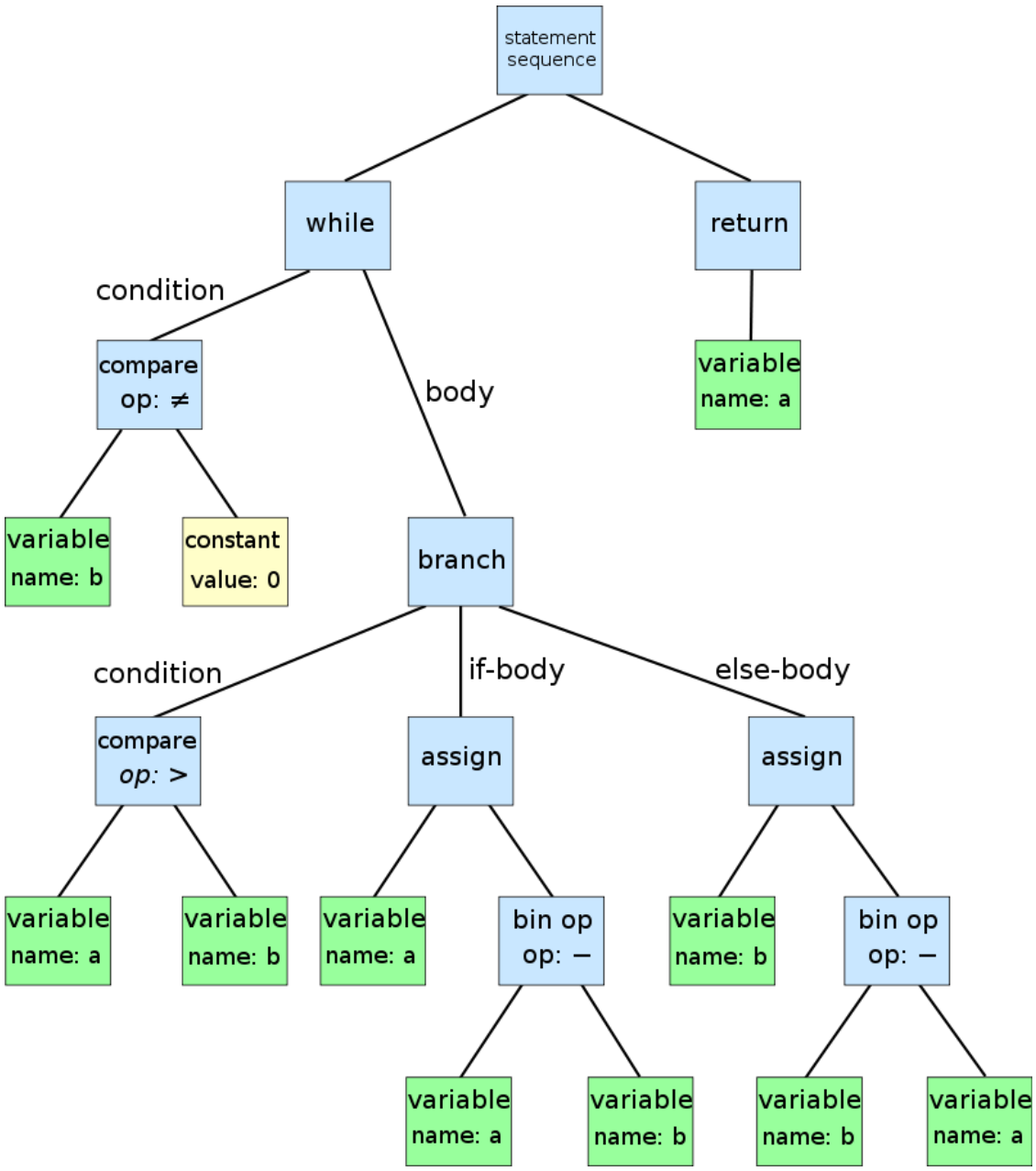
- konfliktų sprendimas galimas tik failo turinio papildymui ar šalinimui ir jei keli darbuotojai modifikuoja tą patį failo turinio vietą ir turi išlikti tik vienas sprendimas, tada mašininis mokymas automatiškai negali išspręsti konflikto;
- esamų konfliktų sprendimo neuroninių tinklų modelių tikslumas nėra didelis, šešių ir daugiau eilučių teisingo išsprendimo tikslumas siekia tik iki 37,04 %;
- neuroninio tinklo naudojimas reikalauja daug resursų ir debesijoje laikomų programų konfliktų sprendimui gali jų nepakakti;
- konfliktų sprendimas yra pritaikomas tik tai programavimo kalbai, pagal kurią modelis yra išmokytas.

Taigi šio metodo taikymas reikalauja kaupti didelį kiekį konfliktų sprendimų, kad būtų galima išmokyti sistemą spręsti konfliktus konkrečiai verslo sistemai.

Kitas būdas automatizuoti versijų kontrolės sistemų konfliktų sprendimą – meta programavimas. Meta programavimas yra kompiuterių programų – meta programų rašymas, kuris kitas programas gali laikyti informacijos šaltiniu. Dėl to jis gali kurti naujas programas arba modifikuoti esamas (Lilis & Savidis, 2019). Manipuliuoti šaltinio kodą reikalauja transformuoti informaciją į struktūrą – abstrakčiosios sintaksės medį (AST). Abstrakčiosios sintaksės medžio pavyzdys atitinkamam kodui pateiktas 5 paveiksle.

```
while b ≠ 0
    if a > b
       a := a - b
    else
        b := b - a
    return a
```

a



b

2.5 pav. Pateikto kodo (a) abstrakčiosios sintaksės medis (b)

Meta programavimo naudojimas suteikia atitinkamų privalumų. Tai yra našumas, šaltinio kodo daugkartinis panaudojimas ir informacijos apie programą suteikimas (Lilis & Savidis, 2019). Meta programavimas pagerina našumą sukurdamas efektyvią specializuotą programą pagal specifikacijas, o ne pagal bendras, bet neefektyvias programas. Informacija apie programą yra suteikiama, kai meta programa išanalizuoja programos struktūrą ir charakteristikas, pritaiko patobulintą optimizaciją, patikrina programos logika ir validuoja pačią programą.

Keičiant versijų kontrolės sistemose naudojamą kodą į abstrakčiosios sintaksės medį galima automatizuoti konfliktus, kurie atsiranda dėl „diff3“ algoritmo neefektyvaus konflikto aptikimo. „diff3“ algoritmas gali sujungti skirtingus metodus į vieną ir aptikti didelį konfliktą, nes metodai turi

panašią struktūrą, tačiau abstrakčiosios sintaksės medis gali išskirti metodus į atskirus dalis ir juos įterpti į failą be konflikto.

Nors automatinis programavimas atrodo gali išspręsti daug problemų, tačiau reikia išskirti tai, kad visiška automatizacija reikalauja dirbtinio intelekto sprendimo. Genetinis programavimas tobulėja, bet jis dar nėra pasiekęs visiškai automatizuotą programavimą. Taip pat taikyti meta programavimą yra labai sudėtinga, norint automatizuoti sudėtingų verslo sistemų versijų kontrolę. Norint pritaikyti automatinį konfliktų sprendimą gali reikėti taikyti kodo semantikos įrankius, kurie papildytų algoritmus informacija, kuri failo modifikacija yra tinkama.

2.4 Versijų kontrolės taikymas verslo sistemose

Šiuo metu didžioji dalis verslų naudoja tam tikras uždarojo kodo, perkamas kaip produktas arba licencijuojamas debesų kompiuterijos informacines sistemos savo versluose. Viena populiariausių informacinių sistemų tipų yra CRM – santykių su klientais valdymo sistema. Šios sistemos yra žmonių, procesų ir technologijų kombinacija, kuria siekiama suprasti įmonės klientus. Tai yra integruotas būdas valdyti santykius sutelkiant dėmesį į klientų išlaikymą ir santykių kūrimą. CRM atsirado iš informacinių technologijų tobulinimo ir organizacinių pokyčių į klientus centruotuose procesuose. Įmonės, kurios sėkmingai įdiegia CRM, gauna klientų lojalumą ir ilgai trunkantį pelningumą. Tačiau tinkamas įdiegimas yra sudėtingas daugeliui įmonių, nes įmonės nesupranta, kad CRM reikalauja visos įmonės, visų funkcijų ir į klientus centruotų verslo procesų perkūrimo. Nors didelė dalis CRM yra technologija, bet žvelgiant į CRM kaip į tik technologijomis grįstą sprendimą nėra teisinga ir tikėtina, kad CRM diegimas žlugs. Valdyti sėkmingą CRM įdiegimą reikalauja integruoto ir balansuoto metodo į technologiją, procesus ir žmones (Chen & Popovich, 2003).

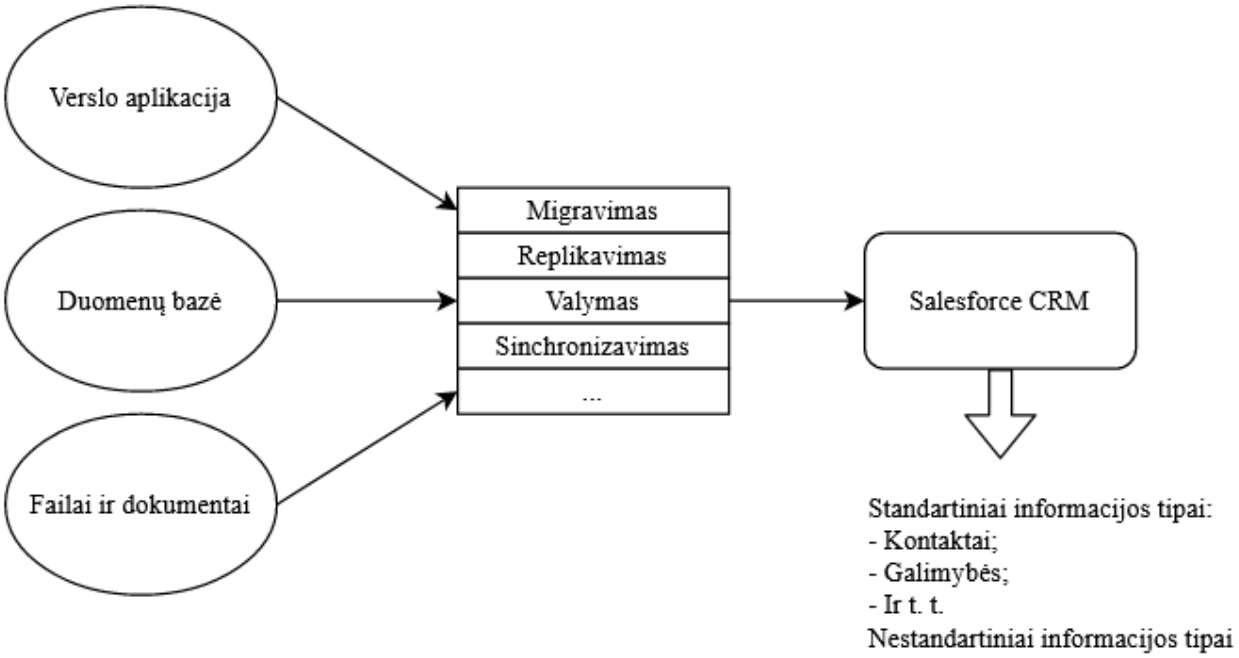
Iš šių CRM sistemų viena populiariausių yra „Salesforce“ CRM sistema. „Salesforce“ yra ypatinga tuo, kad ji egzistuoti tik debesijoje. Dėl to „Salesforce“ produktas yra teikiamas kaip paslauga. Ši sistema padeda tinkamai apdoroti santykius su klientais, apjungti tai su skirtingomis sistemos struktūromis ir sukurti verslui unikalią aplikaciją (Salesforce, s.a.-b). Tai reiškia, kad nereikia turėti jokios aplikacijos pačiame kompiuteryje. Tereikia turėti prieigą prie interneto ir naršyklę ir galima prisijungti prie „Salesforce“ iš bet kurios vietos. „Salesforce“ padeda sudėti svarbias asociacijas su klientais, geriau suprastų klientų norus, nustatyti naujus būdus padėti ir greičiau atkreipti dėmesį į klientui ištinkančias problemas (Salesforce, s.a.-a).

„Salesforce“ architektūra yra daugialypė ir susideda iš trijų pagrindinių dalių:

- Verslo aplikacija. Ji susideda iš skirtingų mažesnių aplikacijų, kurios yra naudojamos įmonių kaip „SAP“, „PeopleSoft“ ir t. t.

- Duomenų bazė. Sistema gali integruotis su rinkiniu skirtingų duomenų bazių pvz.: „Oracle“, „Sybase“ ar „DB2“.
- Failai ir dokumentai. „Salesforce“ CRM architektūra apjungia organizacijos failus ir dokumentus.

Šie trys komponentai yra bendrai paimami ir perduodami per meta procesus kaip migravimas, replikacija, valymas, sinchronizavimas ir t. t. Komponentų ir procesų visuma yra „Salesforce“ CRM, kuri saugo klientų informaciją ir seka progresą per įvairius informacijos tipus kaip galimybės ar kampanijos (Manohar, 2017). Šią architektūrą galima matyti 6 paveiksle.



2.6 pav. „Salesforce“ CRM architektūra (Manohar, 2017)

„Salesforce“ CRM sistema neturi savyje nuolatinio integravimo / nuolatinio diegimo funkcionalumo ir dėl to šiai sistemai reikia pritaikyti versijų kontrolės sistemą su automatizavimo įrankiais. Su versijų kontrolės sistema verslo sistemoje atsiranda galimybė kūrimo procese pritaikyti CI/CD. Sistemoje galima atlikti automatinį testavimą, aptikti sistemos klaidas ir problemas, sumažinti kūrimo ir testavimo trukmę ir pagerinti kitus procesus.

2.5 Analitinės dalies apibendrinimas ir pagrindiniai rezultatai

Verslo sistemos versijų kontrolės uždaviniams automatizuoti reikia pasitelkti atitinkamą versijų kontrolės sistemą, jai tinkamą aplinką ir įrankius. Buvo nustatyti du pagrindiniai versijų kontrolės modeliai: centralizuotasis ir paskirstytasis. Šie modeliai buvo palyginti ir nustatyta, kad sistemos versijų kontrolės uždaviniams automatizuoti reikia kuo dažniau kelti pakeitimus į saugyklą, tai paskirstytasis versijų kontrolės modelis yra tinkamesnis.

Apžvelgus versijų kontrolės sistemos uždavinius buvo nustatyti pagrindiniai privalumai naudotis versijų kontrolės sistemomis. Galimas failų skirtingų versijų saugojimas ir pakeitimų istorijos sekimas. Taip pat galima keliems žmonėms dirbti su tuo pačiu failu, grąžinti pakeitimus ir kurti atsargines kopijas.

Buvo ištirti galimi automatizacijos įrankiai pagal atitinkamus kriterijus. Taip pat nustatyta, kad automatinis versijų kontrolės uždavinių sprendimas nėra įmanomas, jei naudojantis versijų kontrolės sistemomis yra nustatomas pakeitimų sujungimo konfliktas. Dėl to buvo išanalizuoti pagrindiniai metodai automatiniam konfliktų sprendimui. Visų konfliktų sprendimas nėra įmanomas, nes tai reikalauja automatinio kodo rašymo, tačiau yra algoritmų ir metodų, kurie gali automatizuoti dalį konfliktų. Galimi konfliktų sprendimo metodai – genetinis programavimas, duomenimis grindžiamas mašininis mokymas ir abstrakčiosios sintaksės medžio algoritmas.

Ištirta „Salesforce“ verslo sistema ir buvo nustatyta, kad nors ši sistema neturi nuolatinio integravimo ir nuolatinio diegimo funkcionalumo, tačiau galima taikyti versijų kontrolės sistemą su atitinkamais automatizavimo įrankiais. Verslo sistemoje su versijų kontrolės sistema galima atlikti automatinį testavimą, aptikit sistemos klaidas, sumažinti kūrimo proceso trukmę ir pagreitinti kitus susijusios procesus.

2.6 Analitinės dalies išvados

Apibendrinus atliktą analizę formuluojamos tyrimo išvados:

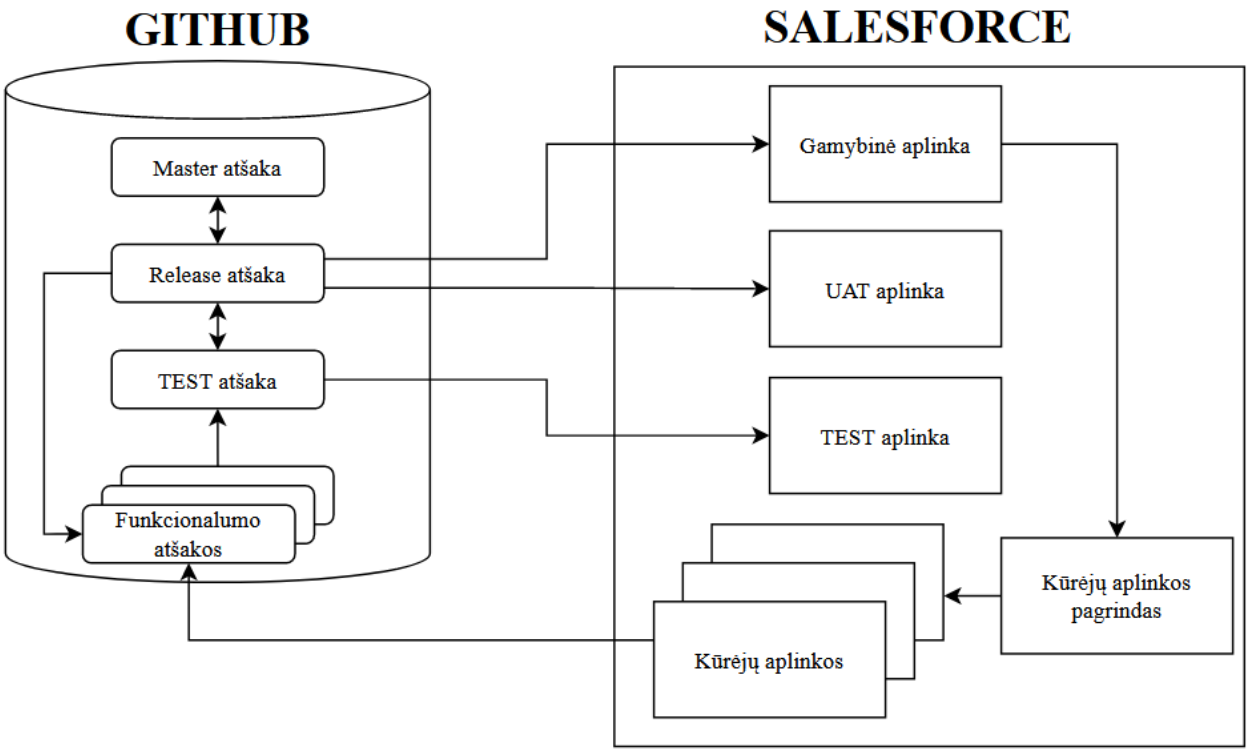
1. Apžvelgus verslo sistemų modelius ir jų privalumus ir trūkumus, buvo nustatyta, kad verslo sistemų versijų kontrolės automatizavimui reikia kuo dažnesnių kėlimų į saugyklą, o tai dažniau atliekama su paskirstytosiomis versijų kontrolės sistemomis.
2. Buvo ištirti pagrindiniai versijų kontrolės uždaviniai ir nustatyta, kad konfliktų sprendimas užtrunka daugiausiai laiko naudojantis šiomis sistemomis.
3. Nustatyta, kad egzistuoja daug įrankių versijų kontrolės uždaviniams automatizuoti, tačiau automatizacija nėra galima, jei sistemoje yra aptinkamas konfliktas failų jungimo metu.
4. Nors automatiniam konfliktų sprendimui nėra sukurto įrankio, tačiau egzistuoja genetinis programavimas, duomenimis grindžiamas mašininis mokymas ir meta programavimas ir jie gali būti pritaikyti verslo sistemos versijų kontrolės sistemai, kad būtų sprendžiami kai kurie konfliktai.

3. SIŪLOMAS METODAS

Šioje baigiamojo darbo dalyje pateikiamas konkrečios analizuojamos srities kūrimo procesas ir nustatomas galimas siūlomas metodas spręsti versijų kontrolės sistemos uždavinius. Iškeliami kuriamai sistemai funkciniai ir nefunkciniai reikalavimai ir sudaroma užduočių diagrama. Išskiriami keturi pagrindiniai procesai ir pateikiamos jiems siūlomos automatizacijos.

3.1 Probleminės srities analizė

Šiuo metu dalykinėje srityje yra naudojama „GitHub“ versijų kontrolės sistemos aplinka. Ši aplinka yra susieta su „Salesforce“ verslo sistemos platformos gamybine aplinka. Dalykinės srities versijų kontrolės sistemos ir „Salesforce“ aplinkų kūrimo proceso diagrama pateikta 1 paveiksle.



3.1 pav. Dalykinės srities kūrimo procesas

Pagal pateiktą 1 paveikslą galima matyti, kad kūrimo procese yra naudojama „GitHub“ versijų kontrolės sistemos aplinka, kuri susideda iš 4 tipų atšakų. Šios atšakos atitinkamai susietos su 4 rūšių skirtingomis „Salesforce“ aplinkomis. „GitHub“ aplinkoje yra išskiriami 4 tipai atšakų:

- „Master“ atšaka, kuri šiuo atveju yra naudojama kaip visos dalykinėje srityje naudojamos sistemos atsarginė kopija;
- „Release“ atšaka, kuri yra susieta su gamybine ir „UAT“ (vartotojo priėmimo testavimo) aplinkomis. Po pakeitimų patvirtinimo ši atšaka yra atnaujinama pagal „TEST“ atšakos gaunamą naują informaciją. Kai pakeitimai yra patvirtinami, jie perkeliami į „Master“ atšaką

vienodos kopijos turėjimui. Šioje atšakoje laikoma informacija, kuri atitinka gamybinę aplinką, t. y. joje yra visa sistema. Taip pat pagal šią atšaką yra kuriamos kūrėjų atšakos, kuriose vykdomi pakeitimai;

- „TEST“ atšaka, kuri yra susieta su „Release“ atšaka ir „TEST“ aplinka. Į šią atšaką yra pirmiausia perkeliami pakeitimai iš kūrėjų atšakų, kai įvykdomi lokalūs testavimai ir sprendžiami sujungimo konfliktai. Iš šios atšakos perkeliama informacija į „TEST“ aplinką;
- Funkcionalumo atšakos, kurių vienu metu gali egzistuoti kelios, priklausomai, kiek kūrėjų dirba su sistema. Jos yra sukuriamos pagal „Release“ atšaką ir pakeitimai jungiami į „TEST“ atšaką, kur sprendžiami jungimo konfliktai, jei tokių būna. Šios atšakos nėra pastovios, t. y. atlikus pakeitimus jos yra uždaromos ir pagal vartotojų istorijas yra vis sukuriamos naujos. Šių atšakų pakeitimų testavimai yra atliekami kūrėjų aplinkose, t. y. informacija tarp kūrėjo aplinkos ir atšakos yra vienoda.

Kūrimo procese yra naudojamos 5 tipų aplinkos:

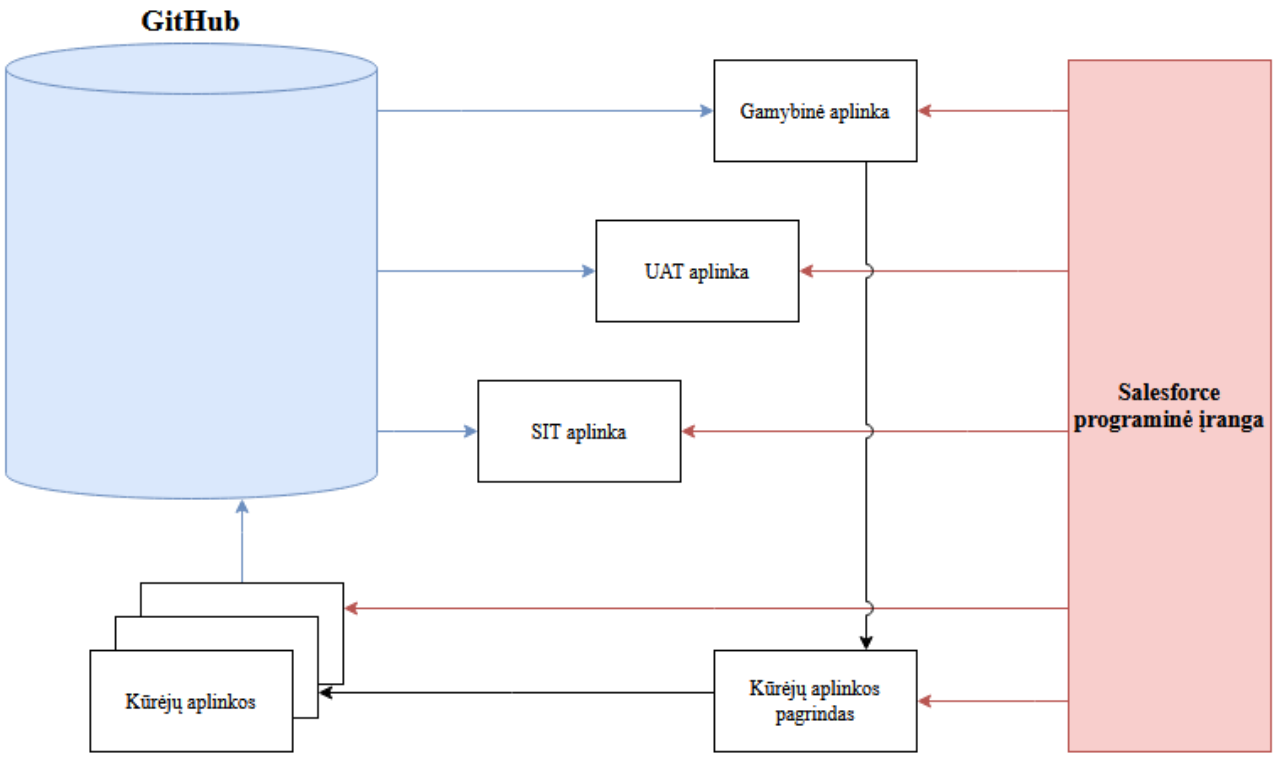
- Gamybinė aplinka – šia aplinka naudojasi tiesioginiai vartotojai pelnui generuoti. Joje testavimas nėra atliekamas, joje veikia visos integracijos ir vykdomi su verslu susiję procesai. Pagal šią aplinką yra vis atnaujinama kūrėjų aplinkos pagrindas, kuriame yra visas su sistema susijęs kodas, bet pati verslo informacija nėra saugoma. Gamybinės aplinkos atnaujinimai vyksta pagal „GitHub“ aplinkos „Release“ atšaką. Ši aplinka atnaujinama, kai „Release“ atšakos pakeitimai yra patvirtinami „UAT“ aplinkoje;
- „UAT“ aplinka – šioje aplinkoje yra vykdomi vartotojų priėmimo testavimai pakeitimams, kurie yra įkeliami į „Release“ atšaką. Šia aplinka naudojasi vartotojai, o kūrėjai pagal vartotojų komentarus keičia savo aplinkose pakeitimus. Šioje aplinkoje yra tik testavimo informacija, o tikros klientų informacijos nėra. Ji yra atnaujinama, kai iš „TEST“ aplinkos patvirtinti pakeitimai perkeliami iš „TEST“ atšakos į „Release“ atšaką;
- „TEST“ aplinka – šioje aplinkoje yra vykdomi visi kūrėjų pakeitimų testavimai, įskaitant sistemos integracijas. Jei pakeitimai nesusiję su integracijos kodu, tada tikrinama ar nėra kokių nepastebimų pokyčių su egzistuojančiomis integracijomis. Ši aplinka yra atnaujinama pagal „GitHub“ aplinkoje esančią „TEST“ atšaką. Šioje aplinkoje testavimą atlieka tam įdarbinti sistemų testuotojai;
- Kūrėjų aplinkos pagrindas – naudojama kurti sistemos kopijas kūrėjams. Šioje aplinkoje neveikia integracijų kodas ir su ja tiesiogiai niekas nedirba. Ji yra nuolat atnaujinama, kai atsiranda nauji pakeitimai gamybinėje aplinkoje;
- Kūrėjų aplinkos – naudojamos kurti naujiems sistemos funkcionalumams, spręsti klaidas, grįžti į šią aplinką, kai atsiranda netinkamų atvejų testavimo metu. Šių aplinkų gali būti tiek

pat, kiek yra dirbančių programuotojų su pačia sistema. Jos yra atnaujinamos pagal poreikį, t. y. jei daromi pakeitimai naujausiam sistemos funkcionalumui, kuris neegzistuoja kūrėjo aplinkoje, tada kūrėjas kreipiasi dėl aplinkos atnaujinimo į sistemos kūrimo administraciją. Ši aplinka yra tiesiogiai susijusi su „GitHub“ aplinkoje sukuriamomis kūrėjų atšakomis. Nuo šios aplinkos į „GitHub“ aplinką yra perkeliami patys pirmieji pakeitimai.

Kai kurie sistemos elementai neegzistuoja „GitHub“ versijų kontrolės sistemoje. Tai yra todėl, kad „Salesforce“ programinė įranga neturi galimybės ištraukti kai kurių elementų meta duomenų. Dėl šios priežasties yra naudojamas kūrimo sekimo dokumentas, kuriame yra pildoma informacija, kai įvykdomi pakeitimai kūrėjų aplinkose, kurių neįmanoma perkelti į „GitHub“ aplinką. Pagal šiame dokumente supildytą informaciją yra įvykdomi tokie pat pakeitimai „TEST“ aplinkoje. Kai testuotojai patvirtina pakeitimus, tada rankiniu būdu pakeitimai yra perkeliami į „UAT“ aplinką ir vartotojams patvirtinus perkeliami galutiniai pakeitimai į gamybinę aplinką. Po perkėlimo į gamybinę aplinką kūrėjų pagrindo aplinka yra atnaujinama.

Kaip matoma sistemoje nėra taikomos visos versijų kontrolės sistemos galimos automatizacijos. Sistemoje neegzistuoja automatinis atšakų integravimas, kuris automatiškai sujungtų skirtingų atšakų laikomą informaciją į bendras atšakas. Taip pat sistemoje nėra automatinio diegimo. Sistemą patobulinti galima naudojantis suplanuotais automatiniais testavimais, automatiniais pakeitimų kėlimais į aplinkas ir automatiniais informacijos ištraukimais iš aplinkų į versijų kontrolės sistemą.

Nors dalykinėje srityje yra apibrėžtas toks kūrimo procesas, tačiau nėra atsižvelgiama į vieną iš pagrindinį šaltinių, iš kurio gali atsirasti sistemos pokyčiai. Visa sistema yra kuriama ir vystoma „Salesforce“ programinėje įrangoje. Dėl šios priežasties visos pateiktos 1 paveiksle aplinkos yra veikiamos „Salesforce“ programinės įrangos atnaujinimais. Šis poveikis pateiktas 2 paveiksle.

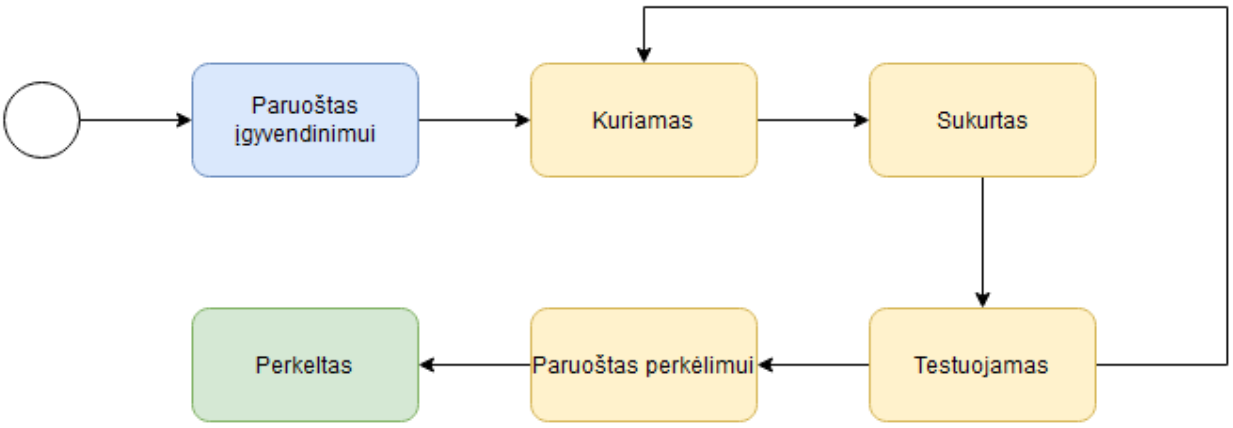


3.2 pav. „Salesforce“ programinės įrangos poveikis dalykinės srities kūrimo procesui

Kadangi „Salesforce“ programinė įranga keičia pačios programinės įrangos technologijas, prideda naujas funkcijas ir pašalina jau pasenusias, tai „GitHub“ versijų kontrolės sistemos aplinka nepadeda iš anksto išvengti pokyčių, kuriuos sukuria pačios programinės įrangos kūrėjai. Pavyzdžiui, jei pašalinamas senas metodas ar įvykdomi saugos pakeitimai, tai sistemos funkcionalumas gali nebeveikti ir gali reikėti rankiniu būdu tikrinti visus sistemos pokyčius ir nustatyti ar jie neigiamai nepaveikia verslo naudotojų darbo.

3.1.1 Verslo sistemos kūrimo proceso dalykinėje srityje analizė

Šiame darbe susikoncentruojama į verslo sistemos kūrimo procesą. Projekto ir užduočių valdymo sistemoje „Jira“ naujam funkcionalumui įgyvendinti yra naudojamos skirtingos būsenos, pagal kurias yra atliekami skirtingi veiksmai. Funkcionalumo būsenos yra pateiktos 3 paveiksle.



3.3 pav. Funkcionalumo įgyvendinimo būsenos

Pagal pateiktą 3 paveikslą matoma, kad egzistuoja 6 funkcionalumo įgyvendinimo būsenos:

1. Paruoštas įgyvendinimui – funkcionalumas yra aprašytas ir paruoštas būti kuriamas. Analitikai, išanalizavę funkcionalumą ir jį aprašę projektų valdymo sistemoje, pakeičia funkcionalumo būseną į „Paruoštas įgyvendinimui“. Toliau šis funkcionalumo įgyvendinimas yra priskiriamas programuotojui, jis pradeda darbą ir funkcionalumo statusas pakeičiamas į „Kuriamas“;
2. Kuriamas – programuotojas vykdo nurodytą užduotį ir atlikęs darbą pakeičia funkcionalumo būseną į „Sukurtas“. Į šią būseną funkcionalumas patenka po to, kai jis jau yra „Paruoštas įgyvendinimui“ būsenoje ir programuotojas pradeda darbą arba kai „Testuojamas“ būsenos funkcionalume aptinkami reikalingi pakeitimai ir jo būsena pagrąžinama;
3. Sukurtas – funkcionalumas yra įgyvendintas ir būsena pakeičiama iš „Kuriamas“ į „Sukurtas“. Kai testavimui yra priskiriamas kitas komandos narys, būsena pakeičiama į „Testuojamas“ ;
4. Testuojamas – funkcionalumo testavimas susideda iš dviejų dalių: funkcinio testavimo ir techninio testavimo. Funkcinio testavimo metu priskirtas žmogus ištestuoja testuojamoje aplinkoje skirtingus funkcionalumo scenarijus, o techninio funkcionalumo metu priskirtas žmogus įvertina įgyvendinto funkcionalumo techninę kokybę „GitHub“ versijų kontrolės aplinkoje. Šie testavimai gali būti atliekami skirtingų asmenų, bet funkcionalumą įgyvendinęs žmogus projektų valdymo sistemoje sukuria dvi užduotis skirtingiems testavimams. Priskirtas žmogus atlieka testavimą, pakomentuoja ir, jei viskas tinkamai įgyvendinta, pakeičia būseną į „Paruoštas perkėlimui“ arba grąžina į būseną „Kuriamas“;
5. Paruoštas perkėlimui – funkcionalumas yra patvirtinamas perkėlimui ir kai jis yra perkeltas, būsena pakeičiama į „Perkeltas“.
6. Perkeltas – funkcionalumas yra įvykdytas ir perkeltas į gamybinę aplinką.

Pagal prieš tai paminėtas projektų valdymo sistemoje apibrėžtas funkcionalumo būsenas kūrimo procesas susideda iš šių žingsnių:

1. Funkcionalumo įgyvendinimas programuotojo asmeninėje „sandbox“ aplinkoje;
2. Perkėlimas į versijų kontrolės aplinką „GitHub“. Kuriamam funkcionalumui yra sukuriama „GIT“ atšaka ir į ją perkeliamas funkcionalumas;
3. Techninis testavimas. Kitas programuotojas peržiūri pakeitimus ir atlieka techninį testavimą. Jei reikia pakeitimų, pateikiami komentarai, tada pagal juos funkcionalumas pataisomas ir po to programuotojas vėl juos peržiūri;
4. Funkcionalumas perkeliamas į bendrą „TEST“ versijų kontrolės atšaką ir testavimo aplinką. Pakeitimai yra sujungiami su „TEST“ atšaka, jei reikia išsprendžiami atsiradę konfliktai, ir perkeliami į „TEST“ aplinką.
5. Funkcinis testavimas. Testavimui priskirtas asmuo atlieka funkcinį testavimą ir aprašo jį „Jira“ projektų planavimo ir valdymo sistemoje. Jei testavimo rezultatai teigiami, funkcionalumas perkeliamas į „UAT“ aplinką, ištestuojamas verslo žmogaus ir pakeičiamas užduoties statusas į „Paruoštas perkėlimui“.
6. Pakeitimai perkeliami į „Release“ atšaką ir atitinkamu laiku yra perkeliami į gamybinę aplinką.

3.1.2 Verslo sistemos kūrimo proceso siūlomos automatizacijos

Pagal prieš tai aprašytą verslo sistemos kūrimo procesą ir pagal su sistema dirbančių asmenų apklausą buvo nustatyti kūrimo etapai, kurie reikalauja daug laiko ir gali būti optimizuoti:

1. Pakeitimų perkėlimas iš versijų kontrolės sistemos į aplinkas arba pakeitimų perkėlimas tarp skirtingų aplinkų. Verslo sistemos kūrimo sistemoje yra priskirtas asmuo, kuris turi perkelti pakeitimus iš „GitHub“ versijų kontrolės aplinkos skirtingų atšakų į atitinkamas „Salesforce“ aplinkas. Šis darbas trunka vieną darbo dieną ir tai vykdoma kas dvi savaites. Iš versijų kontrolės sistemos ištraukiama naujausia informacija ir ji su atitinkamomis komandomis perkeliama į tam tikras aplinkas. Šis procesas gali būti automatizuotas naudojantis skriptu ir „GitHub“ versijų kontrolės aplinkos teikiamu „GitHub Actions“ įrankiu, kuris gali įvykdyti tam tikra nustatytą skriptą po atitinkamų versijos kontrolės sistemoje atliekamų veiksmų.
2. Rankinis funkcinio ir techninio testavimo užduočių kūrimas projektų planavimo ir valdymo sistemoje „Jira“. Kiekvienas programuotojas atlikęs užduotį ir pakeitęs funkcionalumo būseną „Jira“ projektų valdymo sistemoje iš „Kuriamas“ į „Sukurtas“ turi sukurti dvi naujas užduotis: „Techninis testavimas“ ir „Funkcinis testavimas“. Šis užduočių generavimas yra susijęs su versijų kontrolės sistemoje perkeliamais

pakeitimais, todėl tai gali būti automatizuoti su „GitHub Actions“ suteiktu įrankiu iškviečiant sukurtą skriptą, kuris sugeneruotų užduotis.

3. Konfliktų sprendimas versijų kontrolės sistemoje, kai pakeitimai yra perkeliami tarp skirtingų versijų kontrolės sistemos atšakų. Kai pakeitimai perkeliami iš vienos versijų kontrolės atšakos į kitą, kartais atsiranda konfliktai. Šiuos konfliktus kiekvienas programuotojas turi spręsti rankiniu būdu ir tai gali užtrukti nemažai laiko. Kai kurie atsiradę kodo konfliktai gali būti automatiškai sprendžiami iškviečiant parašytą skriptą, kuris paprastas problemas gali išspręsti pats.
4. Automatinių testų paleidimas skirtingose aplinkose. Aplinkose keliami pakeitimai turi būti vis testuojami ir jiems yra rašomi testai. Retkarčiais vieni pakeitimai gali sugadinti kitą sistemos funkcionalumą, į kurį programuotojas neatsižvelgė. Naudojantis versijų kontrolės sistema galima suplanuoti automatinius testų paleidimus aplinkose ir rezultatus perduoti komandai.
5. Kūrėjų aplinkų atnaujinimas po funkcionalumų perkėlimo į gamybinę aplinką. Kas dvi savaites pagrindinė gamybinė aplinka yra atnaujinama su naujausiais funkcionalumais. Po šio veiksmo kiekviena programuotojo aplinka turi būti atnaujinta su naujausia informacija. Šis veiksmas gali būti automatizuojamas naudojantis versijų kontrolės sistema, kai į pagrindinę atšaką yra perkeliami naujausi pakeitimai, tuo pačiu tie pakeitimai būtų perkeliami ir į kitas aplinkas.

Pagal prieš tai pateiktas siūlomas automatizacijas yra projektuojama sistema.

3.2 Sistemai keliami reikalavimai

Pagal prieš tai pateiktas verslo sistemos kūrimo proceso siūlomas automatizacijas verslo sistemai galima pritaikyti CI/CD procesus, kad sistemos kūrimo ir testavimo procesas taptų daug greitesnis, efektyvesnis, būtų išvengiama klaidų ir greičiau aptinkamos sistemos problemos.

Versijų kontrolė gali būti pritaikyta kodo konfliktų sprendimui. Kuriama sistema naudosis daug vartotojų ir jai turi būti nustatyti funkciniai ir nefunkciniai reikalavimai. Reikalavimų nustatymas – tai procesas, kurio metu yra nustatoma, kokias paslaugas turi atlikti kuriama sistema ir kokie ribojimai bus taikomi. Pagal numatomus naudotojų poreikius ir sistemos teikiamas paslaugas sudaromi funkciniai ir nefunkciniai reikalavimai.

3.2.1 Funkciniai reikalavimai

Funkciniai reikalavimai yra sistemos funkcijos, kurios apibūdina sistemas elgseną tam tikromis sąlygomis. Sistemai keliami tokie funkciniai reikalavimai:

- Automatinis pakeitimų perkėlimas iš versijų kontrolės sistemos į dirbamas sistemos aplinkas

    - Sistema automatiškai nustato ir perkelia į atitinkamą aplinką pakeitimus iš reikiamos versijų kontrolės atšakos;
- Automatinis testavimo užduočių kūrimas;
    - Sistema automatiškai sukuria techninio testavimo užduotį projektų valdymo sistemoje „Jira“, kai pakeitimams perkelti yra sukuriamas prašymas versijų kontrolės sistemoje „GitHub“;
    - Sistema automatiškai sukuria funkcinio testavimo užduotį projektų valdymo sistemoje „Jira“, kai pakeitimai yra perkeliami į „TEST“ aplinką.
- Automatinis pakeitimų perkėlimas iš dirbamų aplinkų į versijų kontrolės sistemą;
    - Sistema automatiškai perkelia pakeitimus iš gamybinės aplinkos į versijų kontrolės sistemą, kai juos atlieka administratorius rankiniu būdu nesinaudodamas versijų kontrolės sistema.
- Automatinis testavimas ir rezultatų išvedimas;
    - Sistema automatiškai paleidžia testavimo scenarijus ir rezultatus išveda į atitinkamus kanalus, kad programuotojai greitai sureaguotų, kai testavimo rezultatai neigiami.
- Konflikto sprendimas atitinkamiems scenarijams:
    - Sistema automatiškai paima iš versijų kontrolės serverio konfliktuojančius failus;
    - Sistema automatiškai sulygina konfliktuojančio failo versijas tarp versijų kontrolės atšakų, kurios yra jungiamos;
    - Sistema pateikia konfliktų sprendimo rezultatą su tikimybe, kad sprendimas yra teisingas;
    - Sistema išsaugo konfliktų sprendimus į atitinkamus failus tolesniems darbams.

3.2.2 Nefunkciniai reikalavimai

Nefunkciniai reikalavimai yra susiję ne su sistemos funkcionalumu, bet apibūdina kaip sistema turi veikti. Šių reikalavimų galimi atributai yra saugumas, patikimumas, sistemos priežiūra, plečiamumas ar tinkamumas naudotis. Sistemai keliami tokie nefunkciniai reikalavimai:

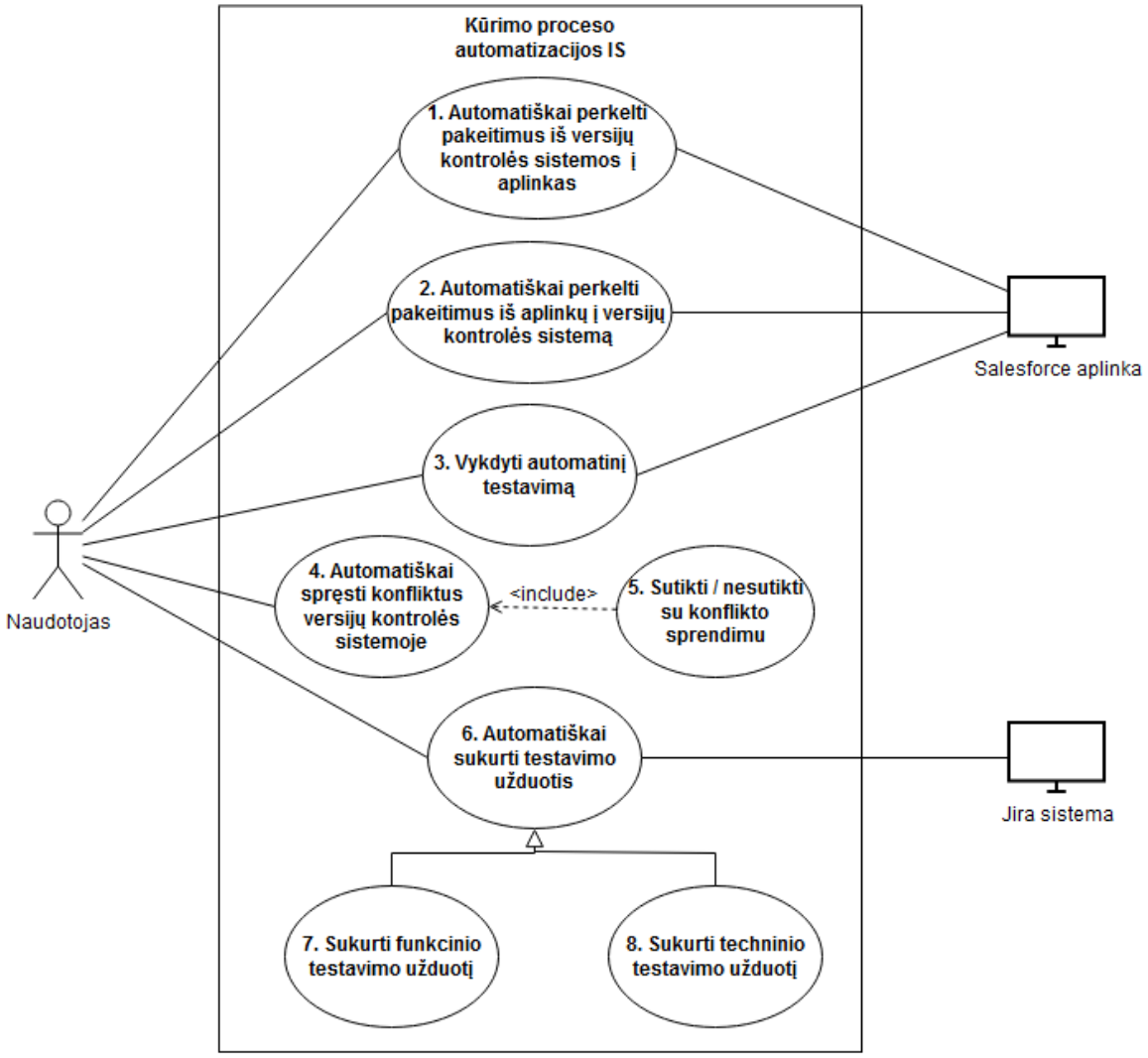
- Reikalavimai tinkamumui naudotis:
    - Naudotojo sąsajoje pateikiama informacija turi būti lengvai įskaitoma;
    - Automatinių pakeitimų perkėlimo rezultatai pateikiami versijų kontrolės sistemoje su pranešimu apie rezultatus;
    - Automatiniai testavimo rezultatai pateikiami atitinkamuose komunikacijos kanaluose;
    - Automatinis konflikto sprendimas pateikiamas versijų kontrolės sistemos prašyme perkelti pakeitimus į kitą atšaką;
    - Failo skirtingos versijos turi būti lengvai pasiekiamos.

- Sistemos veikimas turi nestabdyti naudotojo darbo ir jis gali toliau dirbti su failais, kol vykdomi automatiniai procesai.

- Reikalavimai saugumui:
  - Konfliktuojančių failų versijos turi būti saugiai persiunčiamos iš versijų kontrolės sistemos į konfliktų sprendimų sistemą ir atgal.

3.3 Aukšto lygio užduočių diagrama

Pagal prieš tai pateiktus sistemos keliamus reikalavimus yra sudaromo aukšto lygio užduočių diagrama. Diagrama pateikta 4 paveiksle.



3.4 pav. Verslo sistemos kūrimo proceso automatizacijos informacinės sistemos užduotys

Verslo sistemos kūrimo proceso automatizacijos informacinė sistema suteikia naudotojui galimybę automatiškai perkelti pakeitimus iš versijų kontrolės sistemos į skirtingas aplinkas arba perkelti pakeitimus iš skirtingų aplinkų į versijų kontrolės sistemą, sukurti techninio ir funkcinio

testavimo užduotis projektų valdymo sistemoje ir automatiškai spręsti konfliktus versijų kontrolės sistemoje pasirenkant sutikti ar nesutikti su konflikto sprendimu.

Visus prieš tai paminėtus automatizuotus procesus naudotojas gali matyti versijų kontrolės sistemoje. Pakeitimų perkėlimas iš versijų kontrolės sistemos į aplinkas, testavimo užduočių kūrimas ir konfliktų sprendimas yra susijęs su prašymo perkelti pakeitimus į kitą atšaką sukūrimu. Kai prašymas sukuriamas, automatiškai susigeneruoja techninio testavimo užduotis, jei aptinkamas konfliktas, tada sistema automatiškai pradeda spręsti konfliktus, kai pakeitimai yra patvirtinami ir perkeliami į kitą atšaką, susikuria funkcinio testavimo užduotis ir pakeitimai automatiškai perkeliami į atitinkamą aplinką. Pakeitimų perkėlimas iš aplinkų į versijų kontrolės sistemą ir automatinio testavimo vykdymas yra atliekamas nustatytu laiku kiekvieną dieną.

3.3.1 Užduočių aprašymai

Pagal aukšto lygio užduočių diagramą sudaroma kūrimo proceso automatizacijos sistemos užduočių aprašymų lentelė.

3.1 lentelė. Kūrimo proceso automatizacijos sistemos užduočių aprašymai

Nr.	**Užduočių aprašymas**
1	Pavadinimas – Automatiškai perkelti pakeitimus iš versijų kontrolės sistemos į aplinkas. Tikslas – automatiškai perkelti pakeitimus iš atitinkamos versijų kontrolės sistemos atšakos į atitinkamą aplinką. Prieš sąlyga – naudotojas sujungia savo pakeitimus su atitinkamoje atšakoje esančiu turiniu. Aktoriai: • Naudotojas; • „Salesforce“ aplinka. Sėkmės veiksniai: • Perkelti pakeitimus į aplinką. Ypatingos situacijos: • Nutrūksta ryšys tarp versijų kontrolės sistemos ir tam tikros aplinkos. Variantai: • Pakeitimai sėkmingai perkeliami; • Pakeitimai neperkeliami ir pateikiama klaidos informacija.

	Po sąlyga – Pakeitimai perkelti į aplinką.
2	Pavadinimas – Automatiškai perkelti pakeitimus iš aplinkų į versijų kontrolės sistemą. Tikslas – automatiškai perkelti pakeitimus iš atitinkamos aplinkos į atitinkamą versijų kontrolės sistemos atšaką. Prieš sąlyga – naudotojas atlieka pakeitimus atitinkamoje aplinkoje. Aktoriai: • Naudotojas; • „Salesforce“ aplinka. Sėkmės veiksniai: • Perkelti pakeitimus į versijų kontrolės sistemos atšaką. Ypatingos situacijos: • Nutrūksta ryšys tarp versijų kontrolės sistemos ir tam tikros aplinkos. Variantai: • Pakeitimai sėkmingai perkeliami; • Pakeitimai neperkeliami ir pateikiama klaidos informacija. Po sąlyga – pakeitimai perkelti į versijų kontrolės sistemos atšaką.
6	Pavadinimas – Automatiškai sukurti testavimo užduotis. Tikslas – automatiškai sukurti testavimo užduotis projektų valdymo sistemoje, kai atliekamas atitinkamas veiksmas pakeitimų jungimo metu. Prieš sąlyga – naudotojas sukūrė prašymą perkelti pakeitimus į kitą atšaką. Aktoriai: • Naudotojas; • „Jira“ sistema. Sėkmės veiksniai: • Sukurti techninio testavimo užduotį projektų valdymo sistemoje; • Sukurti funkcinio testavimo užduotį projektų valdymo sistemoje. Ypatingos situacijos:

	• Nutrūksta ryšys tarp versijų kontrolės sistemos ir projektų valdymo sistemos. Variantai: • Techninio testavimo užduotis sėkmingai sukuriama projektų valdymo sistemoje; • Funkcinio testavimo užduotis sėkmingai sukuriama projektų valdymo sistemoje; • Testavimo užduotis nesukuriama ir pateikiama klaidos informacija. Po sąlyga – projektų valdymo sistemoje egzistuoja funkcinio ir techninio testavimo užduotys.
4	Pavadinimas – Automatiškai spręsti konfliktus versijų kontrolės sistemoje. Tikslas – pagal pateiktą konfliktuojančių failų informaciją, išspręsti konfliktą. Prieš sąlyga – sistema priėmė iš versijų kontrolės serverio pateiktus konfliktuojančius failus, juos perskaitė ir pritaikė algoritmą. Aktoriai: • Naudotojas. Sėkmės veiksniai: • Pateikti tinkamą konflikto sprendimą; • Pateikti tikimybę dėl teisingo konflikto sprendimo; • Perduoti informaciją versijų kontrolės sistemai. Ypatingos situacijos: • Sistemai konflikto išspręsti nepavyko. Variantai: • Sprendžiamas konfliktas, kai pateikiamas failo papildymo konfliktas; • Sprendžiamas konfliktas, kai pateikiamas failo turinio šalinimo konfliktas; • Sprendžiamas konfliktas, kai pateikiamas failo turinio modifikavimo konfliktas. Po sąlyga – sistema perduoda konflikto sprendimą į versijų kontrolės sistemą ir naudotojas gali jį peržiūrėti.
3	Pavadinimas – Vykdyti automatinį testavimą. Tikslas – atitinkamose aplinkose vykdyti automatinį testavimą pagal atitinkamų versijų kontrolės sistemos atšakų esamą informaciją. Prieš sąlyga – pakeitimai perkelti į atitinkamą versijų kontrolės sistemos atšaką.

	Aktoriai: • Naudotojas; • „Salesforce“ aplinka. Sėkmės veiksniai: • Paleisti automatinį testavimą atitinkamu laiku tam tikroje aplinkoje; • Išvesti testavimo rezultatus versijų kontrolės sistemoje; • Pateikti testavimo rezultatus komandos nariams atitinkamais kanalais. Ypatingos situacijos: • Nutrūksta ryšys tarp versijų kontrolės sistemos ir aplinkos. Variantai: • Testavimo rezultatai teigiami ir pateikiami teigiami rezultatai atitinkamuose kanaluose ir versijų kontrolės sistemoje; • Testavimo rezultatai neigiami ir pateikiami neigiami rezultatai atitinkamuose kanaluose ir versijų kontrolės sistemoje. Po sąlyga – atitinkama aplinka ištestuojama pagal atšakoje esančius atitinkamus pakeitimus.

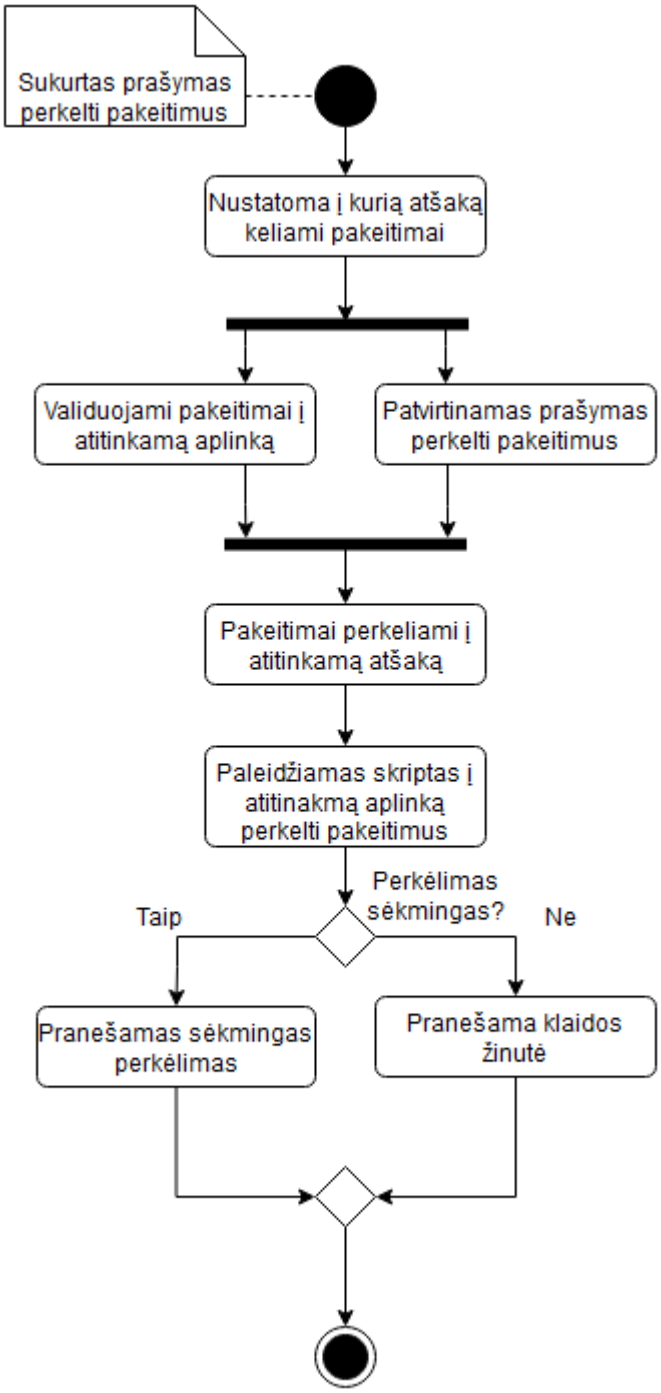
3.4 Automatinis pakeitimų perkėlimas tarp versijų kontrolės sistemos ir skirtingų aplinkų

Vykstant kūrimo procesui į versijų kontrolės sistemą ir į skirtingas aplinkas turi būti vis perkeliami pakeitimai. Šis procesas vyksta kiekvieną darbo dieną ir kiekvienas darbuotojas turi perkelti viską rankiniu būdu. Pakeitimai turi atsidurti versijų kontrolės sistemoje, nes ten yra saugomas tikrasis sistemos kodas su pakeitimų istorija ir visais sujungtais skirtingų darbuotojų pakeitimais. Taip pat pakeitimai turi būti perkeliami į skirtingas aplinkas, kad būtų galimybė juos testuoti ir naudotis sukurtais funkcionalumais gamybinėje aplinkoje. Šie abu procesai gali būti automatizuoti ir tokiu būdu būtų sutaupoma daug darbuotojų laiko pakeitimų perkėlimui ir išvengiama vienu metu pakeitimų kėlimo problema pasinaudojant pakeitimų kėlimo eilėmis.

3.4.1 Automatinis pakeitimų perkėlimas iš versijų kontrolės sistemos į aplinkas

Kaip buvo aprašyta praeitame poskyryje verslo sistemos kūrimo procese egzistuoja keturių tipų atšakos: „Master“, „UAT“, „TEST“ ir funkcionalumo atšakos. Programuotojas pirmus pakeitimus atlieka funkcionalumo atšakoje ir kūrėjo aplinkoje. Šie pakeitimai atliekami dažnai ir nuolatos, kol

sukuriamas funkcionalumas ir automatizacija šiame etape nėra būtinas. Kai funkcionalumas įgyvendinamas ir turi būti perkeliamas į kitas atšakas ir aplinkas, tada proceso automatizacija suteikia daugiau naudos – išvengiama rizika, kad keli programuotojai kels pakeitimus vienu metu, juos užkeis, taip pat sutaupomas laikas keliant kiekvieną funkcionalumą rankiniu būdu. Automatinio pakeitimų perkėlimų iš versijų kontrolės sistemos į aplinkas procesas pateiktas 5 paveiksle.



3.5 pav. Automatinio pakeitimų perkėlimo iš verslo sistemos į aplinkas procesas

Kaip matyti diagramoje pakeitimai yra validuojami ir perkeliami į atitinkamą aplinką:

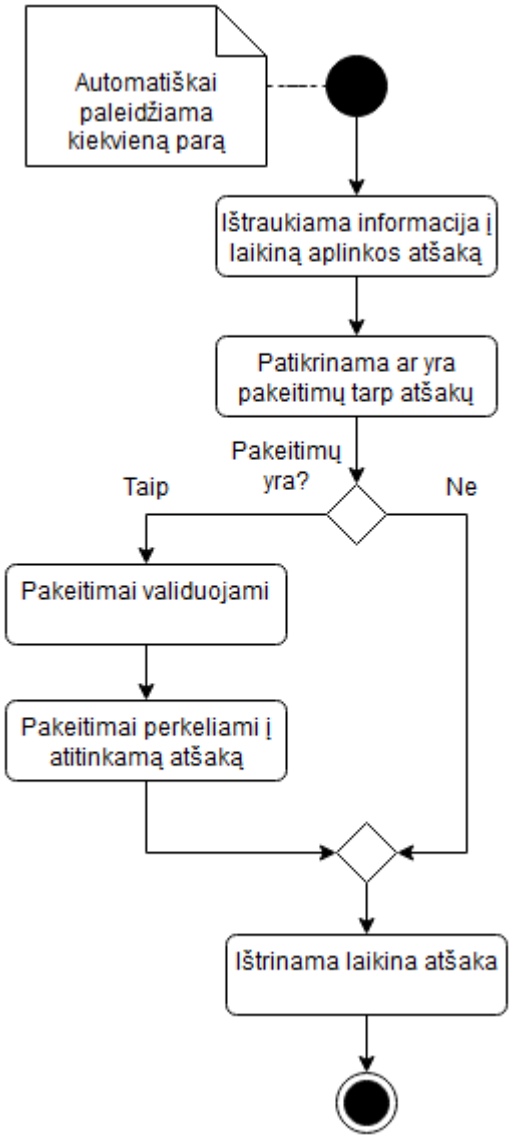
- „TEST“ atšakos pakeitimai perkeliami į „TEST“ aplinką;

- „Release“ atšakos pakeitimai perkeliami į „UAT“ aplinką;

Procesas prasideda, kai sukuriamas prašymas perkelti pakeitimus į „TEST“ arba „Release“ aplinką. Sukūrus šį prašymą sistema automatiškai validuoja pakeitimus į atitinkamą aplinką. Jei validacija ir pakeitimų perkėlimas patvirtinami, tada automatiškai sistema sukelia pakeitimus į tai atšakai skirtą aplinką.

3.4.2 Automatinis pakeitimų perkėlimas iš aplinkų į versijų kontrolės sistemą

Ne visi pakeitimai yra atliekami versijų kontrolės sistemoje. Sistemos administracijai su sistema susijusius pakeitimus atlieka pačioje aplinkoje ir su versijų kontrolės sistema veiksmų neatlieka. Tačiau šie mažesni pakeitimai taip pat turi būti perkelti į versijų kontrolės sistemą, nes kitu atveju administratorius turi tuos pačius pakeitimus atlikti keletą kartų per visas aplinkas. Taip pat atsiranda rizika, kad administratorius gali neperkelti visko į kitą aplinką ar padaryti ne tokius pat veiksmus. Dėl šių priežasčių pakeitimai iš aplinkų turi būti perkelti ir į versijų kontrolės sistemą, iš kurios, jei reikia, pakeitimai taip pat būtų perkeliami į kitas aplinkas pagal prieš tai pateiktą procesą. Automatinio pakeitimų perkėlimo iš aplinkų į versijų kontrolės sistemą procesas pateiktas 6 paveiksle.



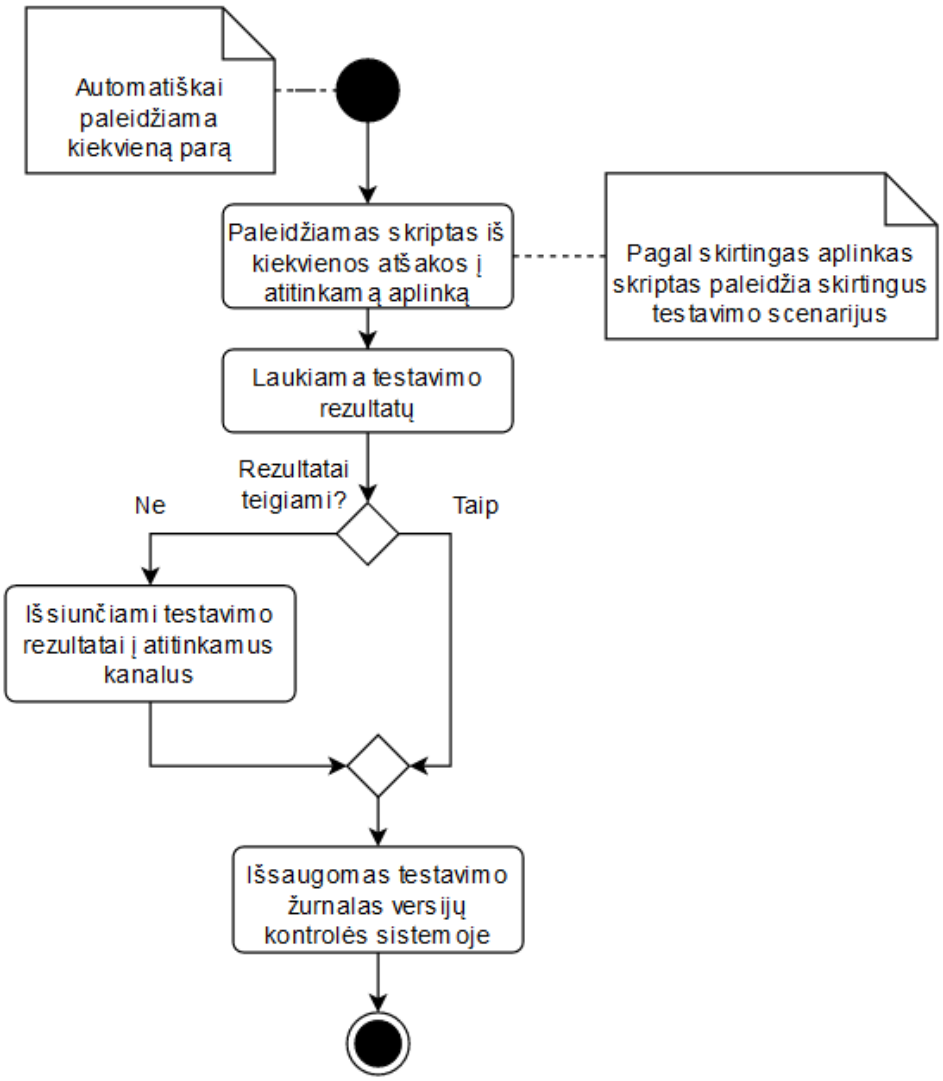
3.6 pav. Automatinio pakeitimų perkėlimo iš aplinkų į atšakas procesas

Versijų kontrolės sistema negali automatiškai aptikti pakeitimų aplinkoje ir į ją juos perkelti. Taip pat verslo sistemos aplinka neturi įrankio, kuris leidžia automatiškai nustatyti, kas pasikeitė tarp aplinkos ir atitinkamos atšakos ir perkelti tuos pakeitimus į verslo sistemos atšaką. Dėl to automatinis pakeitimų ištraukimas iš aplinkos į versijų kontrolės sistemą turi būti atliekamas tam tikru nustatytu laiku kiekvieną dieną iškviečiant skriptą, kuris paimtų visą sistemos informaciją, ją sulygintų su atitinkamoje atšakoje esančiais duomenimis ir perkeltų pasikeitusią informaciją į atšaką.

3.5 Automatinis testavimas

Skirtingose aplinkose kiekvieną dieną atsiranda daug pokyčių. Į aplinkas įkelia pakeitimus administratoriai, programuotojai įdiegia naujus funkcionalumus ir projektuojama automatiškai perkelti pakeitimus iš aplinkų į versijų kontrolės atšakas ir iš atšakų į aplinkas. Visi šie veiksmai sukelia riziką, kad sukurti testavimo scenarijai gali grąžinti neigiamą rezultatą. Dėl to reikalingas automatinis sistemos testavimas. Tačiau, automatinis testavimas negali būti atliekamas automatinių

pakeitimų metu, nes tai trunka daug laiko, apkrauna aplinkas ir sulėtėja aplinkų veikimas. Dėl to automatinis testavimas turi būti atliekamas tam tikru nustatytu laiku kiekvieną parą. Automatinio testavimo į atitinkamas aplinkas procesas pateiktas 7 paveiksle.



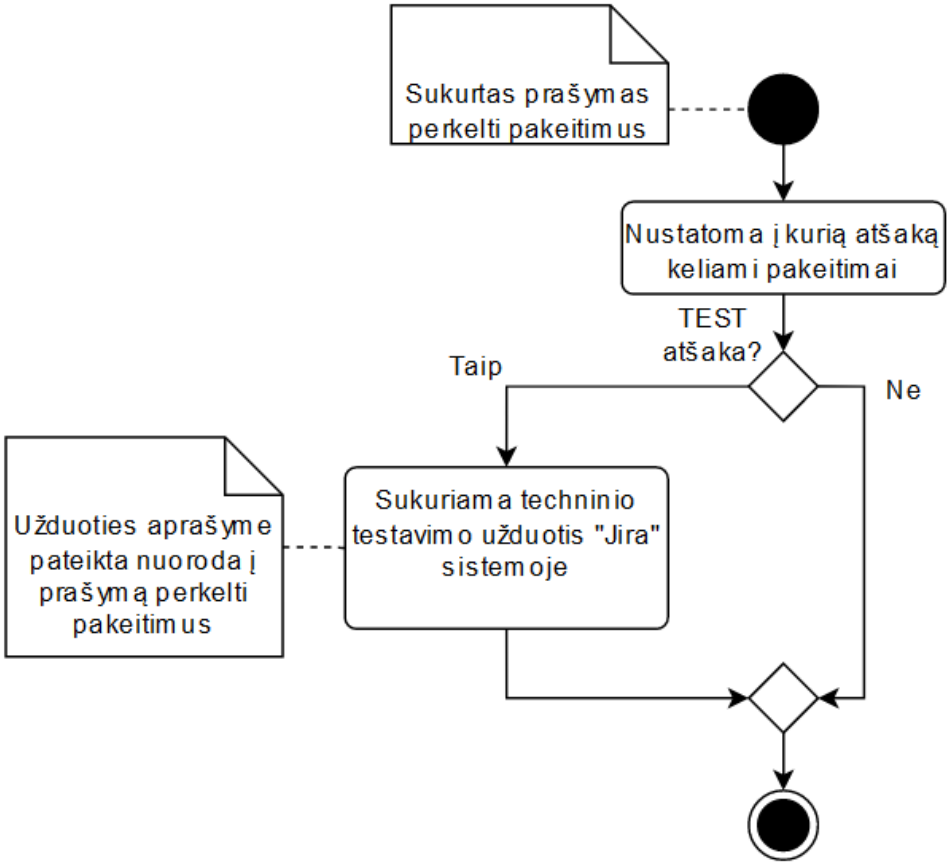
3.7 pav. Automatinio testavimo į atitinkamas aplinkas procesas

Iš versijų kontrolės sistemos galima paleisti nustatytu laiku skriptą. Skriptas iš kiekvienos atšakos į atitinkamą aplinka įvykdo komandą, kuri paleidžia aplinkai sukurtus testavimo scenarijus. Pagal gautus rezultatus skriptas juos perduoda atitinkamais kanalais, kuriuos stebi komandos nariai. Tokiu būdu neveikiantys testai gali būti greitai pastebėti, o paleisti testavimo scenarijai ne darbo metu nesulėtina aplinkų.

3.6 Automatinis testavimo užduočių kūrimas

Visos sistemos kūrimo procesui yra naudojama projektų ir užduočių valdymo sistema „Jira“. Kiekvienam kuriamam funkcionalumui reikalingas techninis ir funkcinis testavimas. Prieš automatizaciją, po funkcionalumo įgyvendinimo programuotojo asmeninėje atšakoje ir sukūrus

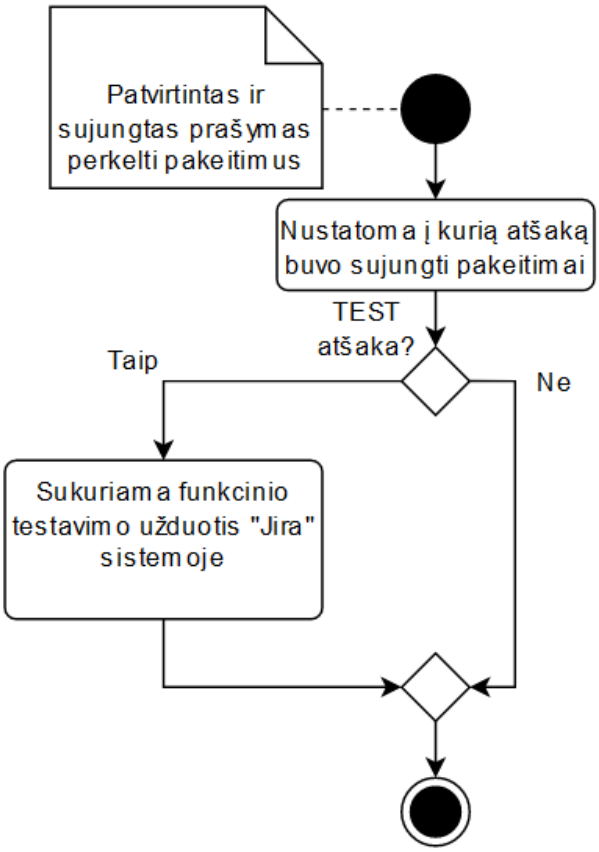
prašymą perkelti pakeitimams į „TEST“ aplinka, programuotojas rankiniu būdu prisijungia prie projektų ir užduočių valdymo sistemos, suranda funkcionalumo aprašymą ir prie jo sukuria techninio testavimo užduotį, kurios aprašymas yra nuoroda į versijų kontrolės sistemoje prašymą perkelti funkcionalumą. Šis procesas gali būti automatizuotas. Automatinis techninio testavimo užduoties sukūrimo procesas pateiktas 8 paveiksle.



3.8 pav. Automatinis techninio testavimo užduoties sukūrimo procesas

Kaip matyti 8 paveiksle, kai susikuria prašymas perkelti pakeitimus į „TEST“ atšaką, paleidžiamas skriptas, kuris pagal funkcionalumo atšakos pavadinimą suranda projektų ir užduočių valdymo sistemoje funkcionalumo aprašą ir sukuria techninio testavimo užduotį su nuoroda į prašymą perkelti pakeitimus versijų kontrolės sistemoje.

Po pakeitimų perkėlimo programuotojas sukuria funkcinio testavimo užduotį. Ši užduotis aprašo, kaip turi būti ištestuojamas funkcionalumas ir priskiriamas žmogus tai atlikti. Šis procesas taip pat gali būti automatizuotas. Automatinis funkcinio testavimo užduoties sukūrimo procesas pateiktas 9 paveiksle.



3.9 pav. Automatinis funkcinio testavimo užduoties sukūrimo procesas

Kaip matyti 9 paveiksle, kai į „TEST“ atšaką yra patvirtinamas prašymas perkelti pakeitimus ir jis sujungiamas su šia atšaka, tada sukuriama funkcinio testavimo užduoties. Tai pašalina būtinumą programuotojui po kiekvieno perkėlimo sukurti funkcinio testavimo užduotį ir tai leidžia matyti, kad pakeitimai jau yra perkelti, nes sukurta užduotis.

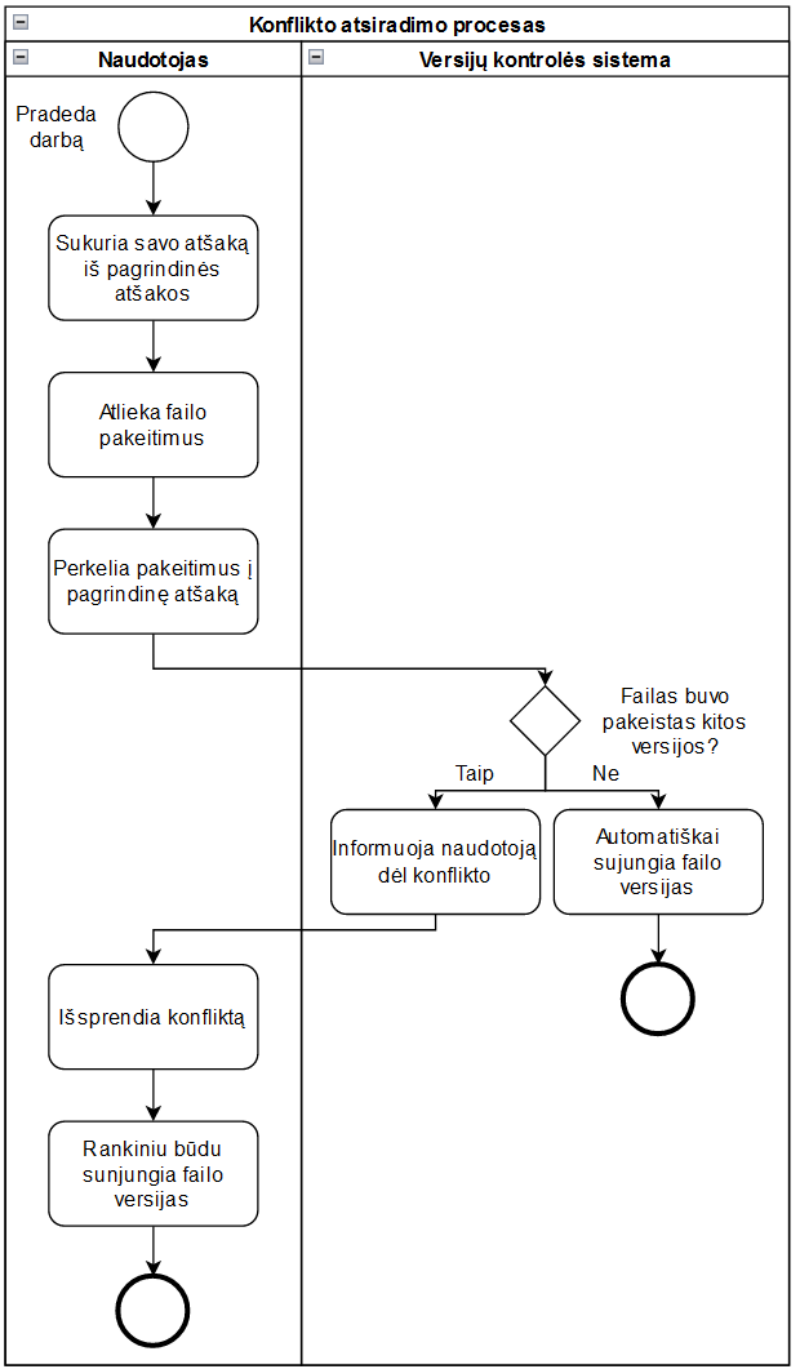
3.7 Konfliktų sprendimas versijų kontrolės sistemoje

Konfliktų sprendimas versijų kontrolės sistemose yra sudėtingas procesas ir visiškas šio proceso automatizavimas šiuo metu nėra įmanomas. Visų konfliktų automatiniam sprendimui reikia automatiškai kodą rašančios sistemos, o tai šiuo metu dar nėra įgyvendinta. Tačiau galima automatizuoti tam tikras konfliktų rūšis, kurios atsiranda versijų kontrolės sistemoje, kai naudojamas „diff3“ standartinis failų jungimo algoritmas. Automatiniam konfliktų sprendimui tam tikrais scenarijais apžvelgiamas konfliktų atsiradimas versijų kontrolės sistemose, konfliktų automatinio sprendimo procesas, galimos sprendimo kategorijos ir sprendimo veiksmų seka.

3.7.1 Konfliktų atsiradimas versijų kontrolės sistemose

Versijų kontrolės sistemose konfliktas atsiranda, kai keli arba vienas naudotojas sukuria savo atskiras atšakas nuo pagrindinės atšakos ir padaro pakeitimus failo tose pačioje vietose. Kai pakeitimai yra atliekami ir pirmasis naudotojas perkelia savo failo versiją į pagrindinę atšaką,

pakeitimai išsisaugo automatiškai, tačiau sekantiems to pačio failo versijų kėlimams yra pateikiamas automatinis konflikto aptikimas ir naudotojas turi rankiniu būdu šį konfliktą išspręsti. Šio proceso diagrama pateikiama 10 paveiksle.

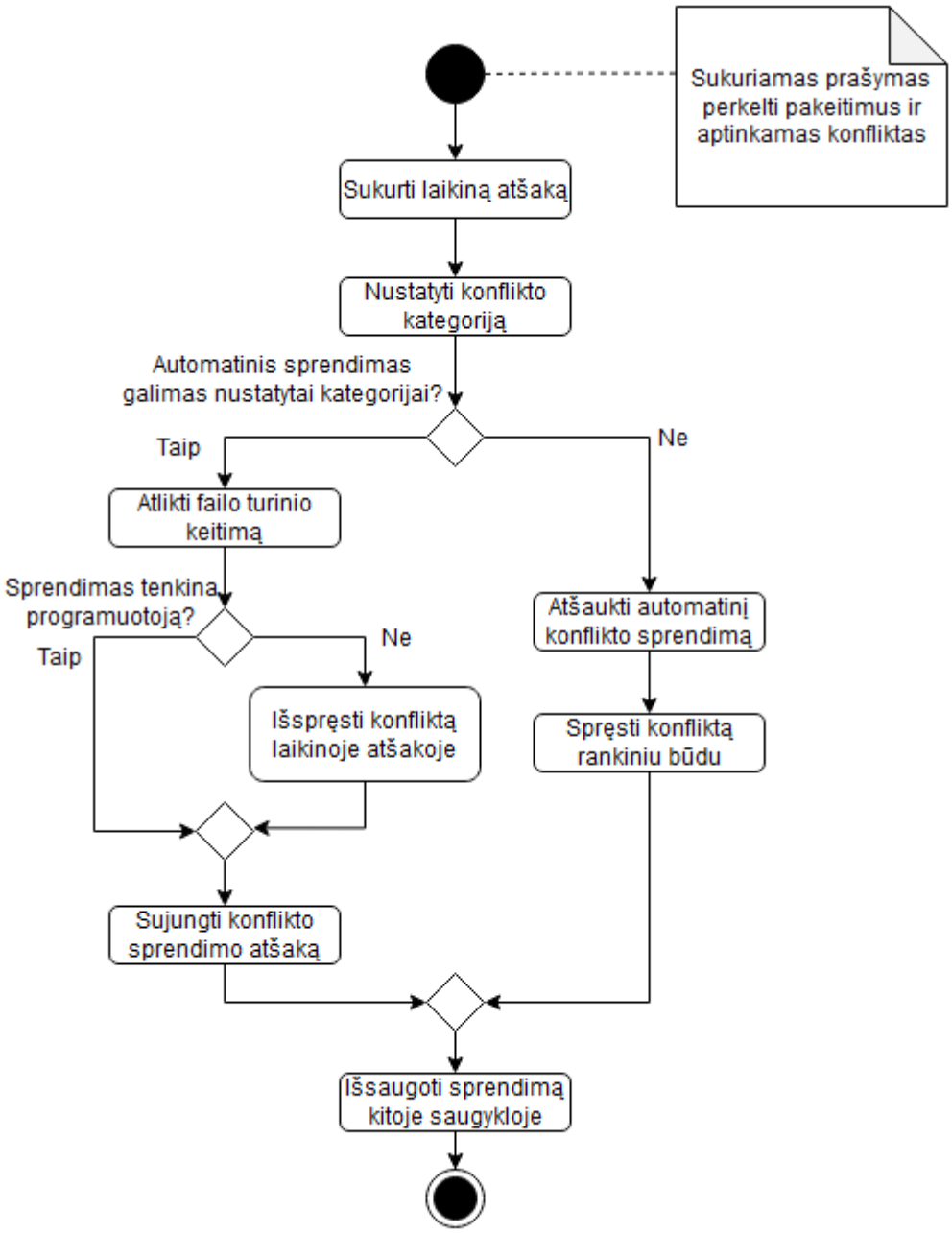


3.10 pav. Konflikto atsiradimas versijų kontrolės sistemoje

Automatinių konfliktų sprendimo sistema proceso žingsnyje, kuriame informuojamas naudotojas dėl konflikto sprendimo, automatiškai išsprendžia konfliktą ir naudotojas vietoj rankinio sprendimo pasirenka sistemos pateiktą sprendimą ir versijų kontrolės sistemoje automatiškai susijungia failų versijos.

3.7.2 Aukšto lygmens konfliktų automatinio sprendimo proceso modelis

Konfliktų automatinių sprendimų sistemoje pagrindinė funkcija yra konflikto sprendimas, kuris vyksta „Automatiškai spręsti konfliktus versijų kontrolės sistemoje.“ užduotyje. Šio konfliktų automatinio sprendimų proceso veiklos diagrama pateikiama 11 paveiksle.



3.11 pav. Konfliktų automatinių sprendimų procesas

Konfliktų sprendimo procesas sistemoje yra sudarytas iš 5 pagrindinių veiksmų:

- Konflikto priėmimas. Šiame veiksme automatinio konfliktų sprendimų sistema priima iš versijų kontrolės sistemos konfliktuojančių failų informaciją. Ši informacija susideda iš skirtingų konfliktuojančių failų versijų, iš kurių viena pažymėta kaip naudotojo atlikti pakeitimai, o kita kaip pagrindinė versija, į kurią bandoma įkelti naudotojo pakeitimus. Taip

pat pateikiami konfliktuojančių failų pavadinimai. Sistema išskiria konfliktuojančią vietą, suranda konflikto pradžios ir pabaigos vietą faile.

- Konflikto kategorijos nustatymas. Šiame veiksme sistema nustato konflikto kategoriją. Pagal šią kategoriją sistema gali taikyti jau aprašytus algoritmus. Jei kategorija nenustatoma, atšaukiamas automatinis konflikto sprendimas, nes konfliktui nėra sukurtas algoritmas automatiniam sprendimui.
- Konflikto sprendimas. Šiame veiksme sistema pritaiko reikiamą algoritmą failų apjungimui ir turinio išsaugojimui.
- Sprendimo pateikimas laikinoje atšakoje. Sistema automatiškai išsaugo sprendimą į laikiną atšaką, kurioje galima įvertinti sprendimo kokybę. Jei sprendimas tenkinamas, jis sujungiamas su reikiama atšaka ir sprendimas išsaugomas kitoje saugykloje sprendimų kaupimui. Kitu atveju programuotojas atšaukia sprendimą ir pateikia rankinį sprendimo variantą.
- Sprendimo išsaugojimas kitoje saugykloje. Šis veiksmas atliekamas papildomai, kad būtų saugomi failai prieš konfliktą, po konflikto ir pats sprendimas. Kitos saugyklos sprendimai bus naudojami ateityje apmokyti neuroninį tinklą.

3.7.3 Konfliktų sprendimo kategorijos

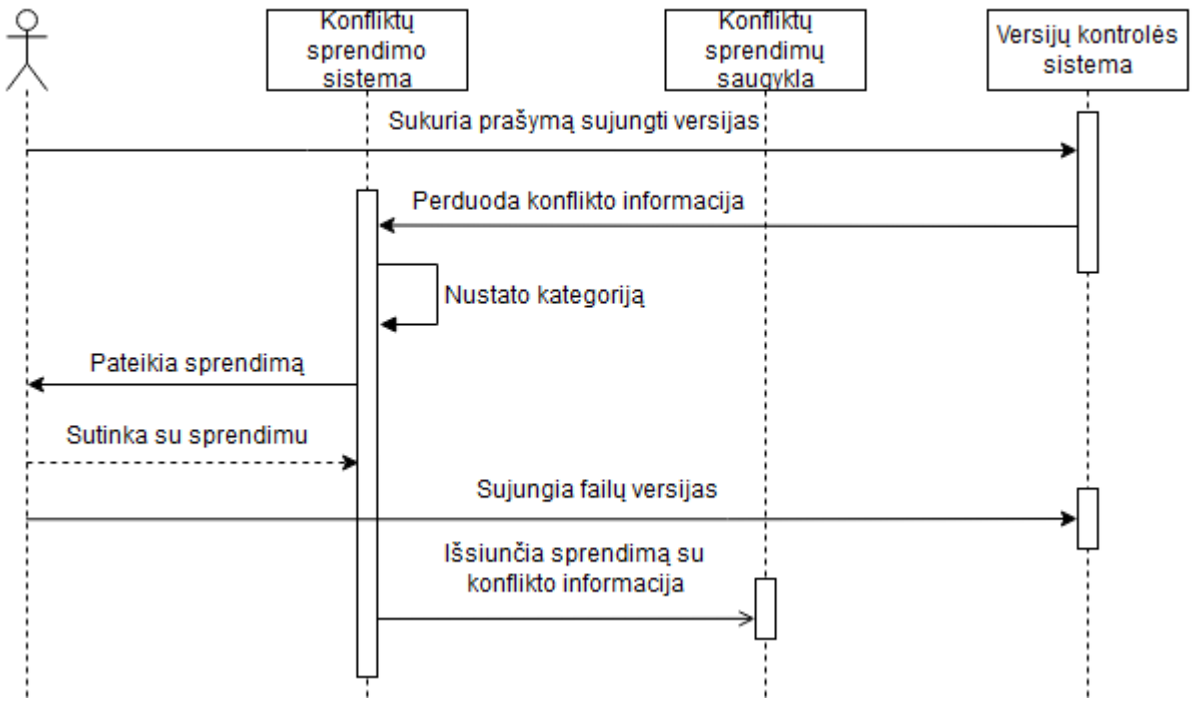
Konfliktų sprendimams yra nustatytos kategorijos, pagal kurias sistema žino, kokius veiksmus turi atlikti, kad būtų tinkamai išspręstas konfliktas. Šios kategorijos yra:

- Konflikto sprendimas, kai įterpiamas papildomas turinys dokumento pabaigoje. Versijų kontrolės sistemos automatiškai neišsprendžia konflikto, jei pridedamas papildomas turinys failo gale, nes esamo turinio kodo pabaigos simbolis yra pakeičiamas naujame dokumente ir aptinkamas konfliktas.
    - Jei įterpiamas turinys failo pabaigoje, tai dokumento pabaigos simbolis yra perkeliamas į po naujai įterptu dokumento turiniu ir failas yra išsaugomas.
    - Kitu atveju, perkeliamas pastraipos ar metodo pabaigos simbolis po įterpiamu metodu.
- Konflikto sprendimas, kai pašalinamas failo turinys. Kaip ir įterpime, versijų kontrolės sistema negali automatiškai išspręsti konflikto, nes esamo turinio dokumento pabaiga yra pakeičiama naujame dokumente.
    - Jeigu trinama dokumento pabaiga, tada dokumento pabaigos simbolis yra perkeliamas prieš trinamą turinį ir dokumentas yra išsaugojamas;
    - Jeigu trinamas dokumentu turinio pradžia ar vidurys, tada dokumento pastraipos ar metodo pabaigos simbolis yra perkeliamas prieš trinamą turinį ir dokumentas yra išsaugojamas.

- Kompleksinis konflikto sprendimas, kai modifikuojamas failo turinys. Šis konflikto sprendimas yra sudėtingiausias, nes tai reiškia, kad keičiama jau egzistuojanti failo turinio dalis ir reikia taikyti sudėtingesnius algoritmus, tačiau galima išskirti keletą subkategorijų, kurios palengvina sprendimą:
    - Pervadinimas. Tai reiškia, kad dokumento turinyje yra dalis, kuri pervadinama. Šiam konfliktui spręsti galima paimti naudotojo modifikuotą dokumento versiją, išsaugoti joje esamą pervadintą turinio dalį į kintamąjį, tą patį padaryti su dokumento versija, kurioje yra turinys prieš pervadinimą ir paleisti apjungimo algoritmą, kuris pervadina visame turinyje esančią informaciją pagal naują pavadinimą.
    - Įterpimo ir šalinimo kombinacija. Šioje subkategorijoje pritaikomi abu algoritmai iš įterpimo ir šalinimo kategorijų.
    - Sudėtingas konflikto sprendimas. Kai turinys yra ne tik pervadinimas, bet ir keičiamas, trinamas, įterpiamas ir konfliktas atsiranda daugelyje turinio vietų. Jei kompleksinis konfliktas įvyksta dėl kelių panašių metodų įterpimo vienoje vietoje, tada jis gali būti išspręstas pritaikant abstrakčios sintaksės medžio algoritmą. Šis metodas atskiria panašius metodus ir klaidingas konfliktas panaikinamas. Tačiau daugeliu atvejų sistema konflikto automatiškai neišsprendžia ir prašoma naudotojo rankiniu būdu išspręsti konfliktą, kurio sprendimas išsaugomas sprendimų saugykloje.

3.7.4 Konfliktų automatinių sprendimų sistemos veiksmų seka

Konfliktų automatinių sprendimų sistemos veikimas susideda iš informacijos gavimo iš versijų kontrolės sistemos, konflikto sprendimo, duomenų saugojimo ir rezultatų išvedimo. Sistemos veiksmų sekai nustatyti yra pateikiama konfliktų automatinių sprendimų sistemos sekos diagrama 13 paveiksle.



3.12 pav. Konfliktų sprendimo naudojant konfliktų sprendimo sistemą sekos diagrama

Prieš tai pateiktame paveiksle yra pateikta veiksmų seka, kai automatinio konfliktų sprendimo sistema nustato kategoriją, kurią gali spręsti ir pateikia sprendimą, su kuriuo naudotojas sutinka. Tokiu atveju naudotojas pagal sukurtą laikiną atšaką sujungia failų versijas ir sistema išsiunčia sprendimą su konflikto informacija į atskirą saugyklą. Yra ir kitų atvejų, kai konfliktų sprendimo sistema nustato kategoriją, kurios negali spręsti arba naudotojas nesutinka su sprendimu. Tokiu atveju naudotojas išsprendžia konfliktą rankiniu būdu, o konfliktų sprendimo sistema tik išsiunčia sprendimą su konflikto informacija į konfliktų sprendimo saugyklą.

3.8 Trečiojo skyriaus apibendrinimas ir pagrindiniai rezultatai

Šiame skyriuje buvo apžvelgtas konkrečios analizuojamos srities kūrimo procesas, nustatytos projektuojamai sistemai užduotys ir galimos automatizacijos: automatinis pakeitimų perkėlimas tarp versijų kontrolės sistemos ir skirtingų aplinkų, automatinis testavimas, automatinis testavimo užduočių kūrimas ir automatinis konfliktų sprendimas.

Buvo pasiūlyta vykdyti automatinį kėlimą iš versijų kontrolės sistemos į „Salesforce“ aplinkas, kai pakeitimai yra sujungiami su aplinkai skirta atšaka. Kai pakeitimai sujungiami su „TEST“ atšaka, jie automatiškai perkeliami į „TEST“ aplinką, kai sujungiami su „Release“ atšaka – į „UAT“ aplinką. Automatinis pakeitimų perkėlimas iš aplinkų vykdomas kiekvieną parą nustatytu laiku, nes nėra galimybės automatiškai aptikti įvykusių pakeitimų pačioje verslo sistemoje.

Automatiniam testavimui įgyvendinti buvo pasiūlyta kviesti kiekvienai atitinkamai aplinkai jos atšakoje laikomus testus iš versijų kontrolės sistemos aplinkos. Kai testavimas įvykdomas, rezultatai išsaugomi atšakoje ir perduoda informacija į komunikacijos kanalus.

Buvo nustatyta, kad automatinis testavimo užduočių sukūrimas galimas prašymo sujungti pakeitimus sukūrimo ir sujungimo metu. Kai sukuriamas prašymas sujungti pakeitimus versijų kontrolės aplinkoje į „TEST“ atšaką, „Jira“ projektų valdymo sistemoje yra sukuriama techninio testavimo užduotis. Kai prašymas sujungiamas, sukuriama funkcinio testavimo užduotis. Kiekvienos užduotys sukūrimas yra pranešamas komunikacijos kanalais, kad būtų matomos sukurtos užduotys.

Visiškas automatinis konfliktų sprendimas nėra įmanomas, bet buvo nustatytos trys konfliktų kategorijos, kurioms galima sukurti algoritmą konfliktų sprendimui: turinio papildymo konfliktas, turinio šalinimo konfliktas ir kompleksinis konfliktas. Daliai šių konfliktų yra aprašomi galimi konfliktų sprendimai.

3.9 Trečiojo skyriaus išvados

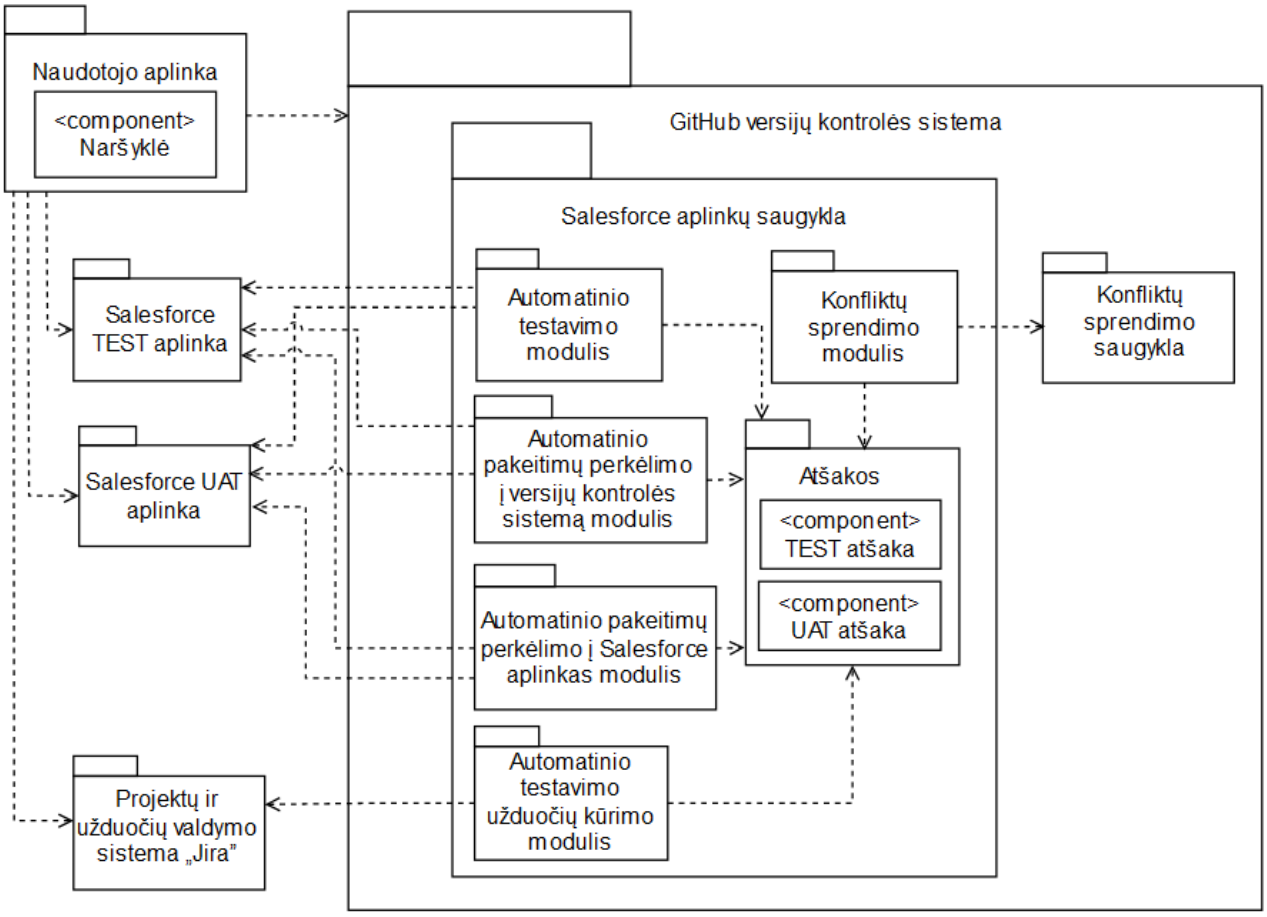
1. Apžvelgus probleminę sritį buvo nustatytos galimos versijų kontrolės uždavinių automatizacijos: automatinis pakeitimų perkėlimas iš versijų kontrolės sistemos į aplinkas, automatinis pakeitimų perkėlimas iš aplinkų į versijų kontrolės sistemas, automatinis testavimas, automatinis testavimo užduočių kūrimas ir automatinis konfliktų sprendimas.
2. Nustatytų užduočių automatizavimui buvo pasiūlyti galimi sprendimo metodai, kurie iškviečiami, kai versijų kontrolės sistemoje aptinkami tam tikri įvykiai arba nustatomi atitinkamas laikas paleisti automatizacijas tam tikru laiku.
3. Automatiniam konfliktų sprendimui buvo nustatyta, kad visiškas automatinis konfliktų sprendimas nėra įmanomas, bet nustatytos galimos konfliktų kategorijos, kurioms pasiūlyti galimi sprendimai. Apibrėžtos konfliktų kategorijos yra failo turinio papildymo konfliktas, failo turinio šalinimo konfliktas ir kompleksinis konfliktas.

4. PROJEKTINĖ DALIS

Šioje darbo dalyje aprašomas sukurtas sistemos prototipas, pateikiami automatinio pakeitimų perkėlimo tarp versijų kontrolės sistemos ir aplinkų, automatinio testavimo, automatinio testavimo užduočių kūrimo ir automatinių konfliktų sprendimų realizacijos ir ištiriamas automatizacijų efektyvumas ir tikslumas.

4.1 Prototipo architektūra

Pagal iškeltus sistemos reikalavimus ir siūlomus automatizacijos būdus buvo sukurtas prototipas. „Salesforce“ sistemos versijų kontrolės uždavinių automatizacijos „GitHub“ aplinkoje prototipo paketų diagrama pateikta 1 paveiksle.



4.1 pav. „Salesforce“ sistemos versijų kontrolės uždavinių automatizacijos „GitHub“ aplinkoje prototipo paketų diagrama

Pagal pateikta prototipo paketų diagramą pateikiamas paketų aprašymas 1 lentelėje.

4.1 lentelė. Prototipo komponentų aprašymai

Paketas	**Aprašymas**
Naudotojo aplinka	Per naudotojo aplinką naudotojas gali pasiekti versijų kontrolės sistemą ir skirtingas „Salesforce“ aplinkas.
„Salesforce“ TEST aplinka	„Salesforce“ aplinka, kurioje atliekamas pakeitimų perkėlimas iš versijų kontrolės sistemos ir į versijų kontrolės sistemą pagal atitinkamus modulius.
„Salesforce“ UAT aplinka	„Salesforce“ aplinka, kurioje atliekamas naudotojų priėmimo testavimas ir pakeitimų perkėlimas iš versijų kontrolės sistemos ir į versijų kontrolės sistemą pagal atitinkamus modulius.
„Salesforce“ aplinkų saugykla	Saugykla, kurioje yra laikomi visi versijuojami duomenis, saugomi visų darbuotojų pakeitimai ir laikomos visos automatizacijos.
Projektų ir užduočių valdymo sistema „Jira“	Sistema, kurioje yra aprašomi funkcionalumai, kuriamos testavimo užduotys ir valdomas visas projektas.
Automatinio pakeitimų perkėlimo į versijų kontrolės sistemą modulis	Modulis, kuris vykdo nustatytu laiku iš TEST ir UAT aplinkų pakeitimų perkėlimą į „Salesforce“ aplinkų saugyklą.
Automatinio pakeitimų perkėlimo į „Salesforce“ aplinkas modulis	Modulis, kuris vykdo pakeitimų perkėlimą iš „Salesforce“ aplinkų saugyklų į atitinkamas aplinkas, kai patvirtinami prašymai perkelti pakeitimus į atitinkamas versijų kontrolės atšakas.
Automatinio testavimo modulis	Modulis, kuris vykdo nustatytu laiku automatinius testavimus paimdamas naujausią informaciją iš atitinkamų „Salesforce“ aplinkų saugyklos atšakų ir ją testuodamas atitinkamoje aplinkoje.

Automatinio testavimo užduočių kūrimo modulis	Modulis, kuris sukuria funkcinio ir techninio testavimo užduotis projektų ir užduočių valdymo sistemoje. Kai sukuriamas prašymas perkelti pakeitimus į TEST aplinką, automatiškai sugeneruojama techninio testavimo užduotis, kai prašymas patvirtinamas, sugeneruojama funkcinio testavimo užduotis.
Konfliktų sprendimo modulis	Modulis, kuris sprendžia atitinkamos rūšies versijų kontrolės konfliktus, kai sukuriami prašymai perkelti pakeitimus. Konfliktai sprendžiami, kai konfliktas atsiranda dėl turinio papildymo, turinio pašalinimo ir metodų / kintamųjų pervadinimo.
Konfliktų sprendimo saugykla	Saugykla, kurioje konfliktų sprendimo modulis saugo sprendimus tolimesniems tyrimams.

Iš diagramos galima matyti, kad suprojektuotos automatizacijos yra „GitHub“ versijų kontrolės sistemos „Salesforce“ aplinkų saugykloje. „Salesforce“ aplinkų saugykloje esančios automatizacijos susijusios su atitinkamomis „Salesforce“ aplinkomis, projektų ir užduočių valdymo sistema „Jira“ ir konfliktų sprendimo saugyklą.

4.2 Projektuojamos sistemos realizacija

Pagal pateiktą prototipo paketų diagramą ir siūlomus automatizacijos metodus realizuojama sistema. „Salesforce“ aplinkų, „Jira“ projektų ir užduočių valdymo sistemos ir „GitHub“ versijų kontrolės sistemos integravimas atliekamas naudojantis suteikta „GitHub Actions“ platforma. Toliau pateikiamos penkių paketų: automatinio pakeitimų perkėlimo į „Salesforce“ aplinkas, automatinių pakeitimų perkėlimo į versijų kontrolės sistemą, automatinio testavimo, automatinio testavimo užduočių kūrimo ir konfliktų sprendimo realizacijos, kuriose aprašomas jų veikimas ir pateikiamas pseudokodas.

4.2.1 Automatinio pakeitimų perkėlimo į „Salesforce“ aplinkas realizacija

„GitHub“ versijų kontrolės sistemos „Salesforce“ aplinkų saugykloje yra sukuriamas „YAML“ programavimo kalba skriptas, kuris iškviečiamas kiekvieną kartą, kai saugykloje yra sukuriamas prašymas perkelti pakeitimus į „TEST“ arba „UAT“ atšaką ir kai pakeitimai yra sujungiami su „TEST“ arba „UAT“ atšaka. Kai yra sukuriamas prašymas perkelti pakeitimus, iškviečiama automatinė pakeitimų validacija, be kurios teigiamo rezultato pakeitimai negali būti sujungiami. Ši validacija aptinka pakeitimus, kurie yra nepilni, klaidingi arba sugadina kitus failus, kurie galimai yra susieti su įgyvendintu funkcionalumu. Validacijos pseudokodas pateiktas 2 paveiksle.

```
JEI veiksmas == 'Sukurtas prašymas perkelti pakeitimus' IR (atšaka == TEST
ARBA atšaka == UAT)
    PALEISTI 'ubuntu-latest'
    PALEISTI 'actions/checkout@v3'
    'Failu sąrašas' = '/manifest/$MANIFEST.xml'
    JEI atšaka == TEST
        'Aplinkos raktas' = 'TEST aplinkos raktas'
        'Aplinkos nuoroda' = 'TEST aplinkos nuoroda'
    JEI atšaka == UAT
        'Aplinkos raktas' = 'UAT aplinkos raktas'
        'Aplinkos nuoroda' = 'UAT aplinkos nuoroda'
    ATŠIFRUOTI 'Aplinkos raktas'
    INSTALIUOTI 'Salesforce CLI'
    'Validacijos skriptas' = {
        'Testų lygis' = 'Be testų',
        'Įvykių fiksavimas' = 'Fiksuoti įvykius',
        'Raktas' = 'Aplinkos raktas',
        'Nuoroda' = 'Aplinkos nuoroda',
        'Keliami failai' = 'Failu sąrašas'
    }
    PALEISTI 'Validacijos skriptas'
    IŠVESTI 'Validacijos rezultatai'
```

4.2 pav. Automatinės validacijos pseudokodas

Kaip matoma iš 2 paveikslo skriptas yra sukonfigūruotas būti paleistas, kai sukuriamas prašymas perkelti pakeitimus. Toliau paleidžiama „Ubuntu“ virtuali mašina, prisijungia prie „Salesforce“ aplinkų saugyklos, ištraukia užšifruotą prisijungimo raktą atitinkamai aplinkai, atšifruoja raktą, instaliuoja „Salesforce“ komandinių eilučių paketą ir paleidžia validacijos skriptą į „Salesforce“ aplinką. Validacijos skriptas yra nustatytas, kad testai nebūtų paleidžiami, validuojami failai atrenkami pagal prie prašymo perkelti pakeitimus pridėtą failų sąrašą ir nustatomas žurnalinių įvykių fiksavimas. Po validacijos rezultatai yra išvedami į prašymo perkelti pakeitimus puslapį.

Kai pakeitimai yra patvirtinami perkėlimui, paleidžiamas automatinio pakeitimų perkėlimo skriptas. Automatinio pakeitimų perkėlimo skripto pseudokodas pateiktas 3 paveiksle.

```
JEI veiksmas == 'Pakeitimai sujungti su atšaka' IR (atšaka == TEST ARBA
atšaka == UAT)
    PALEISTI 'ubuntu-latest'
    PALEISTI 'actions/checkout@v3'
    'Failu sąrašas' = '/manifest/$MANIFEST.xml'
    JEI atšaka == TEST
        'Aplinkos raktas' = 'TEST aplinkos raktas'
        'Aplinkos nuoroda' = 'TEST aplinkos nuoroda'
    JEI atšaka == UAT
        'Aplinkos raktas' = 'UAT aplinkos raktas'
        'Aplinkos nuoroda' = 'UAT aplinkos nuoroda'
    ATŠIFRUOTI 'Aplinkos raktas'
    INSTALIUOTI 'Salesforce CLI'
```

```
'Perkėlimų skriptas' = {
    'Testų lygis' = 'Be testų',
    'Įvykių fiksavimas' = 'Fiksuoti įvykius',
    'Raktas' = 'Aplinkos raktas',
    'Nuoroda' = 'Aplinkos nuoroda',
    'Keliami failai' = 'Failu sąrašas'
}
PALEISTI 'Perkėlimų skriptas'
IŠVESTI 'Perkėlimo rezultatai'
```

4.3 pav. Automatinio pakeitimų perkėlimo pseudokodas

Šis skriptas panašiai kaip validacijos skriptas nustato veiksmą, aplinką, ištraukia atitinkamus raktus, nustato skripto parametrus ir iškviečia pakeitimų perkėlimą. Perkėlimo rezultatai yra pateikiami prie prašymo perkelti pakeitimus. Šie skriptai palengvina pakeitimų perkėlimą tarp aplinkų, nes automatinė validacija nustato problemas dar prieš pakeitimų sujungimą, o automatinis perkėlimas išsprendžia didelių pakeitimų perkėlimo problemą, nes jie yra išskaidomi mažesniais ir sumažėja tikimybė po perkėlimo į gamybinę aplinką atsirasti kėlimų problemų, kurios gali būti ilgai sprendžiamos ir daug kainuoti.

4.2.2 Automatinio pakeitimų perkėlimo į versijų kontrolės sistemą realizacija

Kaip ir automatiniam pakeitimų perkėlimui iš „Salesforce“ sistemos yra sukuriamas „YAML“ programavimo kalba skriptas. Šis skriptas yra iškviečiamas nustatytu laiku kiekvieną parą. Tokiu būdu kiekvieną parą atlikti administratorių ar kitų darbuotojų pakeitimai, kurie nėra išsaugomi versijų kontrolės sistemoje, yra perkeliami į versijų kontrolės sistemą. Kadangi administratoriai atlieka pakeitimus tik „TEST“ ir pagrindinėje darbinėje aplinkoje, šių dviejų aplinkų atitinkamoms atšakoms yra nustatomas automatinis failų ištraukimas. Automatinio pakeitimų perkėlimo iš aplinkų į versijų kontrolės atšakas pateikiamas 4 paveiksle.

```
NUSTATYTI 'Vykdymo grafikas' = '0 24 * * *'
KAI 'Vykdymo laikas'
    PALEISTI 'ubuntu-latest'
    PALEISTI 'actions/checkout@v3'
    'Failu sąrašas ištraukti' = '/manifest/RETRIEVE_MANIFEST.xml'
    'Aplinkos raktas TEST' = 'TEST aplinkos raktas'
    'Aplinkos nuoroda TEST' = 'TEST aplinkos nuoroda'
    'Aplinkos raktas PROD' = 'PROD aplinkos raktas'
    'Aplinkos nuoroda PROD' = 'PROD aplinkos nuoroda'
    INSTALIUOTI 'Salesforce CLI'
    SUKURTI 'Laikina TEST atšaka'
    PRISIJUNGTI_PRIE 'Laikina TEST atšaka'
    ATŠIFRUOTI 'Aplinkos raktas TEST'
    'Ištraukimo skriptas' = {
        'Testų lygis' = 'Be testų',
        'Įvykių fiksavimas' = 'Fiksuoti įvykius',
```

```
        'Raktas' = 'Aplinkos raktas TEST',
        'Nuoroda' = 'Aplinkos nuoroda TEST',
        'Keliami failai' = 'Failu sąrašas ištraukti'
    }
    PALEISTI 'Ištraukimo skriptas'
    JEI 'Pakeitimų yra'
        SUJUNGTI 'TEST atšaka' IR 'LAIKINA TEST atšaka'
        IŠVESTI 'Sujungimo rezultatai TEST'
    IŠTRINTI 'LAIKINA TEST atšaka'
    SUKURTI 'Laikina PROD atšaka'
    PRISIJUNGTI_PRIE 'Laikina PROD atšaka'
    ATŠIFRUOTI 'Aplinkos raktas PROD'
    'Ištraukimo skriptas' = {
        'Testų lygis' = 'Be testų',
        'Įvykių fiksavimas' = 'Fiksuoti įvykius',
        'Raktas' = 'Aplinkos raktas PROD',
        'Nuoroda' = 'Aplinkos nuoroda PROD',
        'Keliami failai' = 'Failu sąrašas ištraukti'
    }
    PALEISTI 'Ištraukimo skriptas'
    JEI 'Pakeitimų yra'
        SUJUNGTI 'PROD atšaka' IR 'LAIKINA PROD atšaka'
        IŠVESTI 'Sujungimo rezultatai PROD'
    IŠTRINTI 'LAIKINA PROD atšaka'
```

4.4 pav. Automatinio pakeitimų perkėlimo į „GitHub“ versijų kontrolės sistemą pseudokodas

Kaip matoma iš 4 paveikslo skriptas yra iškviečiamas pagal nustatytą vykdymo grafiką. Šiuo atveju grafikas nustatytas kiekvieną parą 24 val. Kai pasiekiamas nustatytas laikas, skriptas sukurią „Ubuntu“ virtualią mašiną, prisijungia prie saugyklos, ištraukia versijų kontrolės sistemoje saugomą failų sąrašą, atitinkamus aplinkų raktus ir nuorodas, instaliuoja „Salesforce CLI“, sukuria laikinas atšakas „TEST“ ir gamybinės aplinkų atšakoms, prisijungia prie atitinkamos atšakos ir paleidžia ištraukimo skriptą. Ištraukimo skriptas išsaugo visą informaciją laikinoje atšakoje ir palyginą ją su atitinkamos aplinkos atšaka. Jei pakeitimai aptinkami, sujungiamos atšakos, išvedami sujungimo rezultatai ir ištrinama laikina atšaka. Tokiu būdu kiekvieną dieną „GitHub“ atšakose saugoma informacija yra atnaujinama į naujausią ir programuotojai gali dirbti su naujausiomis failų versijomis.

4.2.3 Automatinio testavimo realizacija

Automatinio testavimo realizacijai yra sukuriamas „YAML“ skriptas, kuris yra patalpinamas „Salesforce“ aplinkų saugykloje. Šis skriptas yra iškviečiamas nustatytu laiku „TEST“, „UAT‘ ir gamybiniai aplinkoms. Kiekvienai aplinkai yra paleidžiami testai iš atitinkamų aplinkų atšakų. Jei po testavimo aptinkami neigiami rezultatai, jie išsiunčiami atitinkamais kanalais komandos darbuotojams. Automatinio testavimo pseudokodas pateiktas 5 paveiksle.

```
NUSTATYTI 'Vykdymo grafikas' = '0 2 * * *'
```

```
KAI 'Vykdymo laikas'
    PALEISTI 'ubuntu-latest'
    PALEISTI 'actions/checkout@v3'
    PALEISTI 'slackapi/slack-github-action@v1.23.0'
    PALEISTI 'send-emails-action'
    'Roboto testai' = '/Robot_Tests/*'
    'Aplinkos raktas TEST' = 'TEST aplinkos raktas'
    'Aplinkos nuoroda TEST' = 'TEST aplinkos nuoroda'
    'Aplinkos raktas UAT' = 'UAT aplinkos raktas'
    'Aplinkos nuoroda UAT' = 'UAT aplinkos nuoroda'
    'Aplinkos raktas PROD' = 'PROD aplinkos raktas'
    'Aplinkos nuoroda PROD' = 'PROD aplinkos nuoroda'
    INSTALIUOTI 'Salesforce CLI'
    PRISIJUNGTI_PRIE 'TEST atšaka'
    ATŠIFRUOTI 'Aplinkos raktas TEST'
    'Testavimo skriptas' = {
        'Testų lygis' = 'Visi testai',
        'Įvykių fiksavimas' = 'Fiksuoti įvykius',
        'Raktas' = 'Aplinkos raktas TEST',
        'Nuoroda' = 'Aplinkos nuoroda TEST',
        'Roboto testai' = 'Roboto testai'
    }
    PALEISTI 'Testavimo skriptas'
    JEI 'Yra neigiamų rezultatų'
        IŠVESTI 'Testavimo rezultatai TEST'
        IŠSIŲSTI 'Testavimo rezultatai TEST' Į 'slack kanalą'
        IŠSIŲSTI 'Testavimo rezultatai TEST' Į 'outlook el. paštų sąrašą'
    PRISIJUNGTI_PRIE 'UAT atšaka'
    ATŠIFRUOTI 'Aplinkos raktas UAT'
    'Testavimo skriptas' = {
        'Testų lygis' = 'Visi testai',
        'Įvykių fiksavimas' = 'Fiksuoti įvykius',
        'Raktas' = 'Aplinkos raktas UAT',
        'Nuoroda' = 'Aplinkos nuoroda UAT',
        'Roboto testai' = 'Roboto testai'
    }
    PALEISTI 'Testavimo skriptas'
    JEI 'Yra neigiamų rezultatų'
        IŠVESTI 'Testavimo rezultatai UAT'
        IŠSIŲSTI 'Testavimo rezultatai UAT' Į 'slack kanalą'
        IŠSIŲSTI 'Testavimo rezultatai UAT' Į 'outlook el. paštų
sąrašą'
    PRISIJUNGTI_PRIE 'PROD atšaka'
    ATŠIFRUOTI 'Aplinkos raktas PROD'
    'Testavimo skriptas' = {
        'Testų lygis' = 'Visi testai',
        'Įvykių fiksavimas' = 'Fiksuoti įvykius',
        'Raktas' = 'Aplinkos raktas PROD',
        'Nuoroda' = 'Aplinkos nuoroda PROD',
        'Roboto testai' = 'Roboto testai'
    }
```

```
PALEISTI 'Testavimo skriptas'
JEI 'Yra neigiamų rezultatų'
    IŠVESTI 'Testavimo rezultatai PROD'
    IŠSIŲSTI 'Testavimo rezultatai PROD' Į 'Slack kanalą'
    IŠSIŲSTI 'Testavimo rezultatai PROD' Į 'Outlook el. paštų sąrašą'
```

4.5 pav. Automatinio testavimo pseudokodas

Kaip matoma iš 5 paveikslo skriptas yra iškviečiamas pagal nustatytą vykdymo grafiką. Šiuo atveju skriptas yra iškviečiamas kiekvieną parą 2 val. nakties. Tokiu būdu testavimo scenarijai patikrina testus po sujungtų pakeitimų iš aplinkų į versijų kontrolės sistemą. Skriptas sukuria virtualią mašiną, prisijungia prie saugyklos, įrašo „Slack“ komunikacijų programos komandas ir el. laiškų siuntimo komandas. „TEST“, „UAT“ ir gamybinėms aplinkoms atitinkamai iš jų atšakų „TEST“ ir „Release“ yra paleidžiami testai. Jei testavimo rezultatai neigiami, tada išvedami testavimo rezultatai į „GitHub“, išsiunčiama žinutė į atitinkamą „Slack“ programos kanalą, kuriame yra visi komandos programuotojai ir administratoriai, ir išsiunčiamas el. laišką į sukurtą sąrašą, kuriame yra tie patys darbuotojai. Tokiu būdu kiekvieną dieną darbuotojai mato naujausius testavimo scenarijų rezultatus ir problemos gali būti greitai išspręstos.

4.2.4 Automatinio testavimo užduočių kūrimo realizacija

Automatinio testavimo užduočių kūrimo realizacijai sukuriami du „YAML“ skriptai. Vienas skriptas yra skirtas techninių užduočių kūrimo automatizacijai, o kitas – skirtas funkcinių užduočių kūrimui. Abu šie skriptai patalpinami į „Salesforce“ aplinkų „GitHub“ saugyklą. Abu skriptai yra iškviečiami, kai pakeičiamas prašymo perkelti pakeitimus statusas. Techninio testavimo užduotis sukuriama, kai pirmą kartą sukuriamas prašymas perkelti pakeitimus į „TEST“ atšaką, o funkcinio testavimo užduotis sukuriama, kai prašymas perkelti pakeitimus yra patvirtinimas. Abu šie veiksmai išsiunčia žinutę į „Jira“ projektų ir užduočių valdymo sistemą. Automatinio techninių užduočių kūrimo pseudokodas pateiktas 6 paveiksle.

```
JEI veiksmas == 'Sukurtas prašymas perkelti pakeitimus' IR atšaka == TEST
    PALEISTI 'ubuntu-latest'
    PALEISTI 'actions/checkout@v3'
    PALEISTI 'atlassian/gajira'
    PALEISTI 'slackapi/slack-github-action@v1.23.0'
    'Raktas' = 'Jira sistemos raktas'
    ATŠIFRUOTI 'Raktas'
    'Techninio testavimo žinutė' = {
        'Pavadinimas' = 'Techninis testavimas'
        'Funkcionalumo pavadinimas' = 'Funkcionalumo atšakos pavadinimas',
        'Aprašymas' = 'Prašymo perkelti aprašymas',
        'Priskiriamas asmuo' = 'Prašymo perkelti pakeitimus tikrintojas',
        'Kūrėjas' = 'Prašymo perkelti pakeitimus kūrėjas',
        'Tipas' = 'Užduotis',
        'Projektas' = 'Salesforce'
```

```
    }
    IŠSIŲSTI 'Techninio testavimo žinutė'
    IŠVESTI 'Techninio testavimo žinutės rezultatą'
```

4.6 pav. Automatinės techninio testavimo užduoties kūrimo pseudokodas

Kaip matoma iš 6 paveikslo skriptas yra iškviečiamas, kai sukuriamas prašymas perkelti pakeitimus į „TEST“ atšaką. Skriptas sukuria virtualią mašiną, prijungia prie saugyklos ir įrašo „Jira“ veiksmų paketą. Toliau sugeneruojama į „Jira“ sistemą žinutė su informacija iš prašymo perkelti pakeitimus. Funkcionalumo „Jira“ sistemoje ir funkcionalumo atšakos pavadinimas „GitHub“ sistemoje sutampa, techninės užduoties aprašymas ištraukiamas iš prašymo perkelti pakeitimus aprašo. Pagal priskirto asmens tikrinti kodą el. paštą yra atrinkamas asmuo „Jira“ sistemoje, o pagal kūrėjo el. paštą yra nustatomas techninės užduoties kūrėjas. Pagal visus šiuos parametrus išsiunčiama žinutė su prašymu sukurti „Subtask“ tipo užduotį, kuri priskirta prie funkcionalumo.

Labai panašiu principu yra sukuriamas automatinio funkcinių užduočių kūrimo pseudokodas. Jis pateiktas 7 paveiksle.

```
JEI veiksmas == 'Pakeitimai sujungti su atšaka' IR atšaka == TEST
    PALEISTI 'ubuntu-latest'
    PALEISTI 'actions/checkout@v3'
    PALEISTI 'atlassian/gajira'
    PALEISTI 'slackapi/slack-github-action@v1.23.0'
    'Jira-Raktas' = 'Jira sistemos raktas'
    ATŠIFRUOTI 'Jira-Raktas'
    'Funkcinio testavimo žinutė' = {
        'Pavadinimas' = 'Funkcinis testavimas'
        'Funkcionalumo pavadinimas' = 'Funkcionalumo atšakos pavadinimas',
        'Aprašymas' = 'Prašymo perkelti aprašymas',
        'Kūrėjas' = 'Prašymo perkelti pakeitimus kūrėjas',
        'Tipas' = 'Užduotis',
        'Projektas' = 'Salesforce',
        'Raktas' = 'Jira-Raktas'
    }
    IŠSIŲSTI 'Funkcinio testavimo žinutė'
    IŠVESTI 'Funkcinio testavimo žinutės rezultatas'
    JEI 'Funkcinio testavimo žinutės rezultatas' == "Teigiamas"
        'Nuoroda' = 'Projekto pavadimo nuoroda ' + 'Funkcionalumo
pavadinimas'
        'Aprašymas' = 'Reikia atlikti funkcinį testavimą ' + 'Nuoroda'
        IŠSIŲSTI 'Aprašymas' Į 'Slack kanalą'
```

4.7 pav. Automatinės funkcinio testavimo užduoties kūrimo pseudokodas

Kaip matoma iš 7 paveikslo skriptas yra iškviečiamas, kai prašymas perkelti pakeitimus yra sujungiamas su „TEST“ atšaka. Tai reiškia, kad po automatinio pakeitimų perkėlimo į „Salesforce“ aplinką susikuria funkcinio testavimo užduotis ir jau gali būti atliekamas testavimas. Skriptas sukuria virtualią mašiną, prijungia prie „GitHub“ saugyklos, įrašo „Jira“ veiksmų paketą ir „Slack“

komunikacijų programos komandas. Kaip ir techninio testavimo užduoties kūrimui yra sukuriama žinutė į „Jira“ sistemą. Tačiau priskirtas asmuo atlikti techninį testavimą nebūtinai atlieka funkcinį testavimą. Dėl to yra naudojamos „Slack“ komandos, kuriomis sukuriama žinutė į bendrą kanalą su prašymu atlikti testavimą su nuoroda į funkcionalumą „Jira“ sistemoje.

Šių automatizacijų realizavimas sumažina laiko sąnaudas, nes nebereikia po kiekvieno prašymo perkelti pakeitimus rankiniu būdu kurti testavimo užduotis. Taip pat šių užduočių automatinis sukūrimas užtikrina, kad pakeitimai jau egzistuoja prašyme perkelti pakeitimus arba aplinkose, o automatinis informavimas apie testavimą užtikriną, kad darbuotojai gali greitai pamatyti ir ištestuoti funkcionalumą.

4.2.5 Automatinio konfliktų sprendimo realizacija

Automatinis konfliktų sprendimas realizuojamas naudojantis du sukurtus „YAML“ skriptus. Pirmas skriptas yra paleidžiamas kiekvieną kartą, kai sukuriamas prašymas perkelti pakeitimus į „TEST“ atšaką ir aptinkamas konfliktas. Konfliktų sprendimui yra iškviečiamas „Python“ skriptas, kuris nustato konflikto kategoriją ir, jei konflikto kategorijai yra sukurtas sprendimo algoritmas, jį pritaiko. Po konflikto sprendimo yra sukuriama konflikto sprendimo atšaka ir į ją patalpinamas atnaujintas failas su sprendimu. Jei programuotoją tenkina sprendimas, jį priima ir sujungia su „TEST“ atšaka. Kitu atveju programuotojas rankiniu būdu išsprendžia konfliktą. Automatinio konfliktų sprendimo pseudokodas pateiktas 8 paveiksle.

```
JEI veiksmas == 'Sukurtas prašymas perkelti pakeitimus' IR atšaka == TEST
    PALEISTI 'ubuntu-latest'
    PALEISTI 'actions/checkout@v3'
    PALEISTI 'actions/setup-python@v4'
    PALEISTI 'action-conflict-finder@v4.0'
    'Konfliktų sprendimo modulis' = '/PythonScripts/ConflictSolver'
    JEI konfliktas = 'Aptiktas'
        SUKURTI 'Funkcionalumo konflikto atšaka'
        'Konflikto informacija' = {
            'Failai' = [
                'Failo pavadinimas',
                'Failo pradinė versija',
                'Failo TEST atšakos versija',
                'Failo funkcionalumo atšakos versija'
            ]
            'Saugyklos pavadinimas' = 'Salesforce aplinkų saugykla',
            'Saugyklos raktas' = 'Salesforce aplinkų saugyklos raktas'
        }
        PERDUOTI 'Konflikto informacija' Į 'konfliktų sprendimo modulis'
        IŠVESTI 'Konflikto sprendimo rezultatai'
```

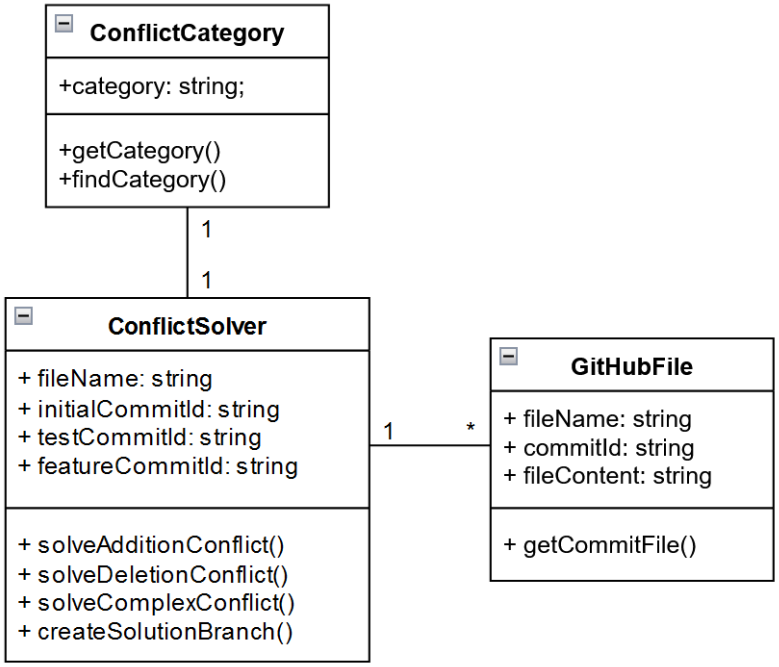
4.8 pav. Automatinio konfliktų sprendimo pseudokodas

Iš 8 paveikslo galima matyti, kad skriptas yra iškviečiamas, kai sukuriamas prašymas perkelti pakeitimus į „TEST“ atšaką ir aptinkamas konfliktas. Sukuriama virtualia „Ubuntu“ mašina, prisijungiama prie „GitHub“ saugyklos ir įrašomas „Python“. Išsiunčiama konflikto informacija į išsaugotą „Python“ konflikto sprendimo modulį, kuriame yra nustatoma konflikto kategorija, pritaikomi sprendimo būdai ir sukuriama atšaka „Salesforce“ aplinkų saugykloje su sprendimu. Skriptas praneša žinutę su nuoroda į konflikto sprendimą. Jei sprendimas nepateikiamas, tada programuotojas prisijungia prie konflikto sprendimo atšakos, jį išsprendžia ir sujungia su TEST atšaka. Šis skriptas sumažina funkcionalumo įgyvendinimo trukmę ir gali išspręsti konfliktus, kurie atsiranda dėl paprasto konfliktų sprendimui taikomo „GitHub“ algoritmo. Taip pat ši automatizacija išsprendžia kitų automatizacijų nutraukimo riziką dėl atsiradusio konflikto.

Kai iškviečiamas „Python“ skriptas, į jį perduodami keturi parametrai:

- Failo pavadinimas – naudojamas ištraukti tinkamą failo versiją iš „Salesforce“ aplinkų saugyklos atitinkamos atšakos;
- Failo pradinė versiją – failo versijos identifikavimo kodas, kuris nurodo versijų kontrolės sistemoje į pradinį failą, nuo kurio buvo pradėti darbai „TEST“ atšakos failo versijoje ir funkcionalumo atšakos versijoje. Į šią failo versiją skriptas sujungia „TEST“ ir funkcionalumų atšakos failų versijas ir pateikia sprendimą;
- Failo „TEST“ atšakos versija – failo versijos identifikavimo kodas, kuris nurodo versijų kontrolės sistemoje į „TEST“ atšakoje esantį failą. Šio failo turinys yra naudojamas skripte bandant sujungti pakeitimus tarp „TEST“ ir funkcionalumo atšakų;
- Failo funkcionalumo atšakos versija – failo versijos identifikavimo kodas, kuris nurodo versijų kontrolės sistemoje į funkcionalumo atšakoje esantį failą. Šio failo turinys yra naudojamas skripte bandant sujungti pakeitimus tarp „TEST“ ir funkcionalumo atšakų.

Šie kintamieji yra naudojami konfliktų sprendimo „Python“ skripte, kuriame sugeneruojamas konfliktuojančio failo turinio sprendimas, sukuriama nauja sprendimo atšaka ir perkeliamas išspręstas failas į tą atšaką. Konfliktų sprendimo skripto klasių diagrama pateikta 9 paveiksle.



4.9 pav. Konfliktų sprendimų sistemos klasių diagrama

Kaip matoma iš 9 paveikslo konfliktų sprendimo modulis sudarytas iš trijų klasių: „ConflictSolver“, „ConflictCategory“ ir „GitHubFile“. Konfliktų sprendimo klasėje yra vykdomi konfliktų sprendimai pagal nustatyto konflikto kategoriją, konfliktų kategorijos klasėje yra vykdomas konflikto kategorijos nustatymas ir jos išvedimas, o „GitHub“ failų klasėje yra ištraukiamas failo turinys pagal failo pavadinimą ir atitinkamą failo versijos identifikavimo kodą.

Pagal konfliktuojančių failų turinį yra išskirtos trys kategorijos: failo turinio papildymo konfliktas, failo turinio šalinimo konfliktas ir kompleksinis konfliktas. Konfliktuojančio failo turinys failo turinio papildymo konflikto metu pateikiamas 10 paveiksle.

```
public class Test1 {
    public void method1(){
        System.debug('test');
    }
Accept Current Change | Accept Incoming Change
<<<<<<< HEAD (Current Change)
    public void method2(){
        Integer x;
        x = 1;
        System.debug(x);
=======
    public void method3(){
        Integer y;
        y = 2;
        System.debug(y);
>>>>>>> feature1 (Incoming Change)
    }
}
```

a)

```
public class Test1 {
    public void method1(){
        System.debug('test');
    }
    public void method2(){
        Integer x;
        x = 1;
        System.debug(x);
    public void method3(){
        Integer y;
        y = 2;
        System.debug(y);
    }
}
```

b)

```
public class Test1 {
    public void method1(){
        System.debug('test');
    }
    public void method2(){
        Integer x;
        x = 1;
        System.debug(x);
    }
    public void method3(){
        Integer y;
        y = 2;
        System.debug(y);
    }
}
```

c)

4.10 pav. Konfliktuojančio failo turinys failo turinio papildymo metu; a) konfliktuojančios vietos; b) konflikto sprendimas taikant abiejų pakeitimų sujungimą; c) išspręstas konfliktas taikant turinio papildymo metodą

Kaip matoma iš 10 paveikslo konfliktas aptinkamas, nes abu metodai yra įterpiami failo pabaigoje. Jeigu abu metodai sujungiami sutinkant su abejais pakeitimais, tada klasėje yra užfiksuojama klaida, kad trūksta „method2“ uždarymo simbolio „}“. Taikant turinio papildymo algoritmą, pakeitimai yra sujungiami ir tarp metodų yra įterpiamas papildomas „}“ uždarymo simbolis. Tokiu pat būdu sprendžiamas ir turinio šalinimo konfliktas.

Kai konfliktas yra aptinkamas keliose vietose arba konfliktas užfiksuojamas persidengdamas per metodą, tada reikia taikyti kompleksinio konflikto kategorijos sprendimą. Konfliktuojančio failo turinys kompleksinio konflikto metu pateikiamas 11 paveiksle.

```
public class Test1 {
    public void method1(){
        System.debug('test');
    }
Accept Current Change | Accept Incoming Change | Acc
<<<<<<< HEAD (Current Change)
    public string getString1(){
        return 'Test1';
=======
    public String getString2(){
        return 'Test2';
>>>>>>> feature2 (Incoming Change)
    }
    public void method2(){
        Integer x;
        x = 1;
Accept Current Change | Accept Incoming Change | Acc
<<<<<<< HEAD (Current Change)
        String string1 = getString1();
=======
        String string2 = getString2();
>>>>>>> feature2 (Incoming Change)
        System.debug(x);
    }
    public void method3(){
        Integer y;
        y = 2;
        a = 'Test';
        System.debug(y);
    }
Accept Current Change | Accept Incoming Change | Acc
<<<<<<< HEAD (Current Change)
    public method4(){
        Integer z;
        z = 3;
        System.debug(z);
=======
    public void method5(){
        Integer q = 1;
        System.debug(q);
>>>>>>> feature2 (Incoming Change)
    }
}
```

a)

```
public class Test1 {
    public void method1(){
        System.debug('test');
    }
    public string getString1(){
        return 'Test1';
    public String getString2(){
        return 'Test2';
    }
    public void method2(){
        Integer x;
        x = 1;
        String string1 = getString1();
        String string2 = getString2();
        System.debug(x);
    }
    public void method3(){
        Integer y;
        y = 2;
        a = 'Test';
        System.debug(y);
    }
    public method4(){
        Integer z;
        z = 3;
        System.debug(z);
    public void method5(){
        Integer q = 1;
        System.debug(q);
    }
}
```

b)

```
public class Test1 {
    public void method1(){
        System.debug('test');
    }
    public string getString1(){
        return 'Test1';
    }
    public String getString2(){
        return 'Test2';
    }
    public void method2(){
        Integer x;
        x = 1;
        String string1 = getString1();
        String string2 = getString2();
        System.debug(x);
    }
    public void method3(){
        Integer y;
        y = 2;
        a = 'Test';
        System.debug(y);
    }
    public method4(){
        Integer z;
        z = 3;
        System.debug(z);
    }
    public void method5(){
        Integer q = 1;
        System.debug(q);
    }
}
```

c)

4.11 pav. Konfliktuojančio failo turinys kompleksinio konflikto metu; a) konfliktuojančios vietos; b) konflikto sprendimas taikant abiejų pakeitimų sujungimą; c) išspręstas konfliktas taikant kompleksinio konflikto sujungimo metodą

Kaip matoma iš 11 paveikslo konfliktas aptinkamas, nes metodai arba metoduose esančios funkcijos yra įterpiamos tose pačiose vietose. Jeigu visi konfliktai sujungiami sutinkant su abejais pakeitimais, tada klasėje užfiksuojamos sintaksės klaidos. Tokiam konfliktų sprendimui galima taikyti abstrakčiosios sintaksės medžio algoritmą. Šis algoritmas atskiria iš skirtingų versijų metodus ir sujungia juos kaip atskirus kodo blokus. Tokiu būdu konfliktas yra automatiškai išsprendžiamas.

Po konflikto atšakos sujungimo su „TEST“ atšaka yra iškviečiamas kitas „YAML“ skriptas, kuris išsaugo konfliktų sprendimus į kitą „GitHub“ saugyklą. Saugykloje išsaugomas kiekvienas konflikto sprendimas atskirame aplanke penkiuose failuose. Pirmas failas yra pradinė bendra failo versija, nuo kurios išsiskyrė funkcionalumų daromi pakeitimai, antras failas yra „TEST“ atšakoje esantis failas, trečias failas yra iš funkcionalumo atšakos modifikuotas failas, kuris konfliktuoja su „TEST“ atšakoje esančiu failu, ketvirtas failas yra su konflikto išsprendimu, o penktame faile yra

pateikiama konflikto kategorija, kuri yra įrašoma rankiniu būdu, jei programuotojas turėjo išspręsti konfliktą pats. Konfliktų sprendimo nusiuntimo į kitą saugyklą pseudokodas pateiktas 12 paveiksle.

```
JEI veiksmas == 'Pakeitimai sujungti su atšaka' IR atšaka == TEST IR
jungimo-atšaka = 'Funkcionalumo konflikto atšaka'
    PALEISTI 'ubuntu-latest'
    PALEISTI 'actions/checkout@v3'
    'Failai' = [
                'Failo pradinė versija',
                'Failo TEST atšakos versija',
                'Failo funkcionalumo atšakos versija',
                'Failas su konflikto išsprendimu',
                'Konflikto kategorijos failas'
    ]
    'Aplankas' = 'Funkcionalumo pavadinimas' + 'failo pavadinimas'
    PRISIJUNGTI PRIE 'Konfliktų sprendimo saugykla'
    SUKURTI 'Aplankas'
    ATIDARYTI 'Aplankas'
    ĮKLIJUOTI 'Failai'
    IŠVESTI 'Konflikto išsaugojimo rezultatai'
```

4.12 pav. Konfliktų sprendimo nusiuntimo į kitą saugyklą pseudokodas

Kaip matoma iš 12 paveikslo, skriptas prisijungia prie kitos saugyklos, su „Linux“ komandomis sukuria aplanką ir į jį patalpina visus failus. Šis skriptas yra skirtas tik konfliktų saugojimui. Šis failų saugojimas bus panaudotas tolimesnėje analizėje nustatyti daugiau konfliktų kategorijų ir konfliktų sprendimų algoritmo tobulinimui.

4.3 Prototipo tyrimas

Suprojektuotas versijų kontrolės uždavinių automatizacijos prototipas yra ištiriamas eksperimentiškai. Tuo yra siekiama išsiaiškinti ar siūlomas metodas yra efektyvus versijų kontrolės uždaviniams spręsti. Šios automatizacijos yra pritaikomos realiai „Salesforce“ verslo sistemai ir buvo stebimi rezultatai – prieš automatizacijos pritaikymą ir po pritaikymo. Tyrimas yra išskaidomas į dvi dalis:

- Kūrimo proceso laiko sąnaudų tyrimas dviejų savaičių laikotarpyje prieš automatizacijos pritaikymą ir dviejų savaičių laikotarpyje po pritaikyto versijų kontrolės uždavinių automatizavimo;
- Automatinių konfliktų sprendimo realizacijos tikslumo tyrimas.

4.3.1 Kūrimo proceso automatizacijos tyrimas

„Salesforce“ verslo sistemai yra taikomas lankstus projektų valdymo ir programinės įrangos kūrimo metodas. Prieš automatizacijų realizavimą sistemos kūrimo procese buvo identifikuoti pagrindiniai veiksmai, kurie gali būti patobulinti:

- Pakeitimai keliami tarp aplinkų retai ir dažniausiai apima daug funkcionalumų vienu metu. Tai reiškia, kad pakeitimų kėlimas yra lėtas, apima daug sistemos elementų ir testavimas turi būti atliekamas visiems susijusiems funkcionalumams. Kėlimas į gamybinę aplinką užtrunka apie 6 valandas;
- Kiekvienas kėlimas į „TEST“ aplinką yra atliekamas rankiniu būdu. Tai reiškia, kad pakeitimų kėlimui susidaro eilė, nes kiekvienas žmogus turi kelti pakeitimus vienas paskui kitą ir tai vidutiniškai užtrunka 30 minučių kiekvienam kėlimui;
- Visos sistemos bendras testavimas prasideda tik dvi dienos prieš paskirtą perkėlimo į gamybinę aplinką dieną. Visų funkcionalumų automatinis testavimas užtrunka vidutiniškai 6 valandas ir po testavimo rezultatų dažnai nustatoma daug problemų, nes daug pakeitimų keliami vienu metu;
- Kiekvienas programuotojas keldamas pakeitimus į „TEST“ aplinka turi sukurti techninio ir funkcinio testavimo užduotis „Jira“ projektų valdymo sistemoje. Šis užduočių kūrimas vidutiniškai trunka 5 minutes ir tai dažnai kartojasi, nes testavimo užduotyse ir prašyme perkelti pakeitimus reikia aprašyti tokią pat informaciją;
- Sistemos administratoriai dažnai pakeitimus atlieka tiesiogiai „Salesforce“ aplinkose. Iš to atsiranda rizika praleisti tam tikrus žingsnius kėlimo į kitą aplinką metu arba gali būti atlikti nevienodi pakeitimai tarp aplinkų. Šis rankinis pakeitimų perkėlimas vidutiniškai gali trukti apie 10 minučių.

Pagal prieš tai pateiktus sistemos kūrimo veiksmus pateikiama „Salesforce“ sistemos kūrimo proceso automatizavimo apžvalgos lentelė.

4.2 lentelė. „Salesforce“ kūrimo proceso automatizavimo apžvalga.

Procesas	**Prieš automatizaciją**	**Po automatizacijos**
Pakeitimų kėlimas testavimui į „Salesforce“ aplinkas	Pakeitimai keliami retai ir apima daug funkcionalumų vienu metu. Susidaro kėlimo eilė. Vidutinė trukmė darbuotojui atlikti kėlimą - 30 min.	Pakeitimai keliami automatiškai po vieną funkcionalumą, susidaro automatinė eilė. Darbuotojui nebereikia atlikti kėlimo.
Sistemos testavimas	Testavimas atliekamas retai ir dvi dienos prieš kėlimą į gamybinę aplinką atliekamas visos sistemos testavimas, kuris gali trukti 4 – 8 val.	Testavimas atliekamas kiekvieną naktį, nesutrukdo aplinkų naudojimosi ir rezultatai pateikiami

		komandai atitinkamais kanalais.
Tiesioginis pakeitimų darymas aplinkose	Administratoriai kelia pakeitimus tiesiogiai į visas aplinkas, turi identiškai atkartoti veiksmus, kurie gali užtrukti apie 10 minučių.	Pakeitimai automatiškai kiekvieną parą perkeliami į versijų kontrolės sistemą ir sujungus atšakas, tokie pat pakeitimai perkeliami į kitas aplinkas.
Funkcionalumo valdymas „Jira“ projektų valdymo sistemoje	Komandos nariai turi sukurti techninio ir funkcinio testavimo užduotis kiekvienam funkcionalumui ir tai užtrunka 5 min.	Užduotys automatiškai susikuria sistemoje, kai sukuriamas ir sujungiamas prašymas perkelti pakeitimus ir komanda informuojama apie galimą testavimą.
Pakeitimų išleidimas į gamybinę aplinką	Bendras visų pakeitimų kėlimas trunka 6 val. ir dažnai reikalauja papildomų veiksmų, nes atsiranda konfliktų tarp skirtingų funkcionalumų.	Pakeitimų išleidimas į gamybinę aplinką trunka apie 2 val., nes išankstinis testavimas ir problemų taisymas sumažina riziką komplikacijoms.

Iš lentelės galima matyti, kad po automatizacijų realizacijos kūrimo procesas tapo spartesnis. Pakeitimai į „TEST“ aplinką Administratorių pakeitimai automatiškai perkeliami į versijų kontrolės sistemą, kurie automatiškai yra ištestuojami ir juos galima kelti per „GitHub“ į kitas aplinkas. Sistemos automatinis testavimas atliekamas kiekvieną naktį ir rezultatai pateikiami komandai. Taip pat programuotojai mažiau laiko užtrunka „Jira“ sistemoje, nes nebereikia rankiniu būdu kurti užduotis. Nauji funkcionalumai ir pokyčiai išleidžiami į gamybinę aplinką greičiau, sumažėjo rizika atsirasti komplikacijoms kėlimo metu.

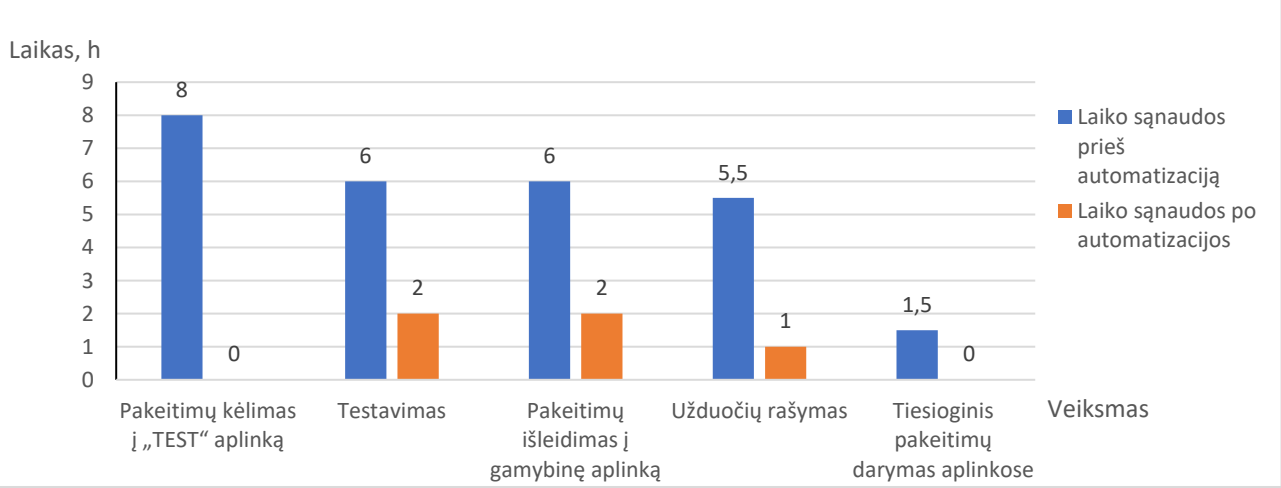
Sistemos kūrimo procesas yra vykdomas pagal „Agile“ metodiką, todėl funkcionalumų išleidimas į gamybinę aplinką yra atliekamas kartą į dvi savaites. Dėl šios priežasties tyrimas buvo atliekamas dviejų savaičių laikotarpiais. Komanda susideda iš 10 narių, 8 iš jų – programuotojai, o 2 – administratoriai. Buvo sekamas laikas, kiek užtruko komandos nariai 2 savaičių laikotarpyje prieš

automatizaciją ir 2 savaičių laikotarpyje po automatizacijos. Pagal praleidžiamą visų darbuotojų laiką kūrimo proceso veiksmuose sudaroma laiko sąnaudų lentelė.

4.3 lentelė. Laiko sąnaudos kūrimo proceso veiksmuose

Veiksmas	**Praleistas laikas prieš automatizaciją, val.**	**Praleistas laikas po automatizacijos, val.**
Pakeitimų kėlimas į „TEST“ aplinką	8	0
Testavimas	6	2
Pakeitimų išleidimas į gamybinę aplinką	6	2
Užduočių rašymas	5,5	1
Tiesioginis pakeitimų darymas ne pagrindinėje aplinkoje	1,5	0
Iš viso	27	5

Pagal gautus rezultatus sudaroma laiko sąnaudų kūrimo proceso veiksmuose diagrama, kuri pateikta 13 paveiksle.

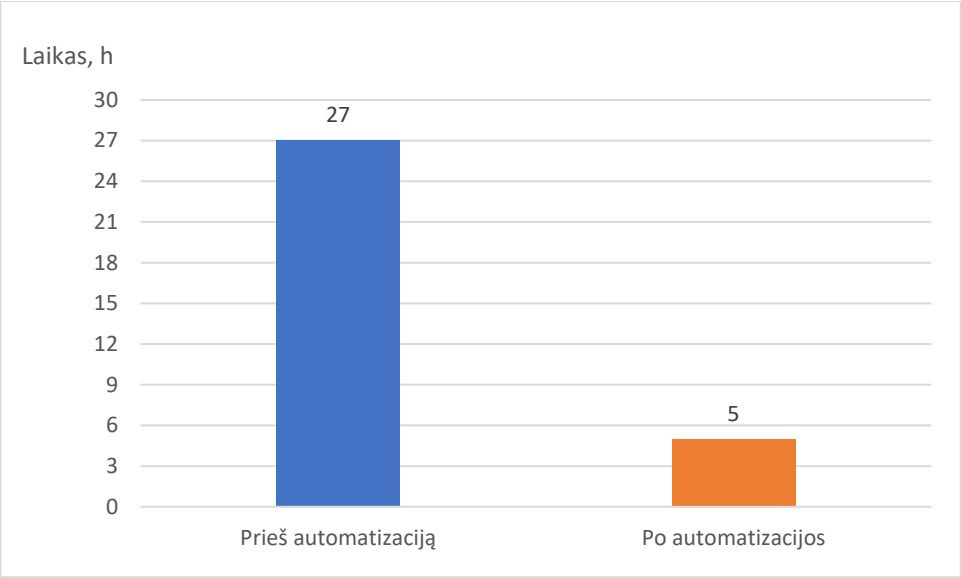


4.13 pav. Laiko sąnaudos kūrimo proceso veiksmuose

Kaip matoma iš 3 lentelės ir 13 paveikslo po automatizacijos daugiausiai laiko buvo sutaupyta pakeitimų kėlime į „TEST“ aplinką. Šis veiksmas buvo pašalintas, nes pakeitimai po patvirtinto prašymo perkelti pakeitimus į „TEST“ atšaką yra automatiškai perkeliami į aplinką. Kiti du veiksmai, kurie žymiai sutrumpėjo yra testavimas ir pakeitimų išleidimas į gamybinę aplinką. Šių veiksmų praleidžiamas laikas sutrumpėjo iš po 6 valandų į po 2 valandas. Tai nulėmė automatinis testavimas, nes atsiradusios problemos sistemoje pranešamos kiekvieną parą ir greitai ištaisomos. Dėl to neatsiranda atvejų, kai prieš kėlimą testavimo metu ir pačių pakeitimų išleidimo metu aptinkami konfliktuojantys funkcionalumų atvejai, kurie reikalauja daug laiko juos išspręsti. Užduočių rašymas sutrumpėjo iš 5,5 val. į 1 valandą. Šis veiksmas sutrumpėjo dėl automatiškai sukuriamų testavimo

užduočių po prašymo perkelti pakeitimus sukūrimo ir sujungimo. Pats veiksmas neišnyko, nes funkciniam testavimui reikia papildomai parašyti testavimo scenarijų, kuris nėra pateikiamas versijų kontrolės sistemos prašyme perkelti pakeitimus. Tiesioginis pakeitimų darymas ne pagrindinėje aplinkoje buvo pašalintas, nes pakeitimai automatiškai perkeliami į versijų kontrolės sistemą, iš kurios jie yra automatiškai perkeliami, kai sujungiamas prašymas juos perkelti.

Pagal prieš tai pateiktą laiko sąnaudų kūrimo proceso veiksmuose lentelę ir diagramą sudaroma bendrų laiko sąnaudų kūrimo proceso veiksmuose diagrama. Ji pateikta 13 paveiksle.



4.14 pav. Bendros laiko sąnaudos kūrimo proceso veiksmuose

Iš diagramos matyti, kad kūrimo procese išskirtų veiksmų laiko sąnaudos dviejų savaičių laikotarpyje sumažėjo iš 27 valandų į 5 valandas. Taigi praleidžiamas laikas prie nustatytų uždavinių sumažėjo 81,48 %. Komandoje yra 10 narių, tai vidutiniškai vienam darbuotojui sutaupomas laikas dviejų savaičių laikotarpyje – 2,2 val.

4.3.2 Automatinio konfliktų sprendimo realizacijos tyrimas

Automatinių konfliktų sprendimo realizacijos efektyvumui nustatyti buvo atliktas eksperimentinis efektyvumo tyrimas. 3 mėnesius buvo sekami „GitHub“ versijų kontrolės sistemoje atsiradę konfliktai kėlimo į „TEST“ atšaką metu, pritaikytas automatinių konfliktų sprendimo skriptas ir nustatytas išspręstų ir neišspręstų konfliktų kiekis pagal atitinkamą konflikto kategoriją. Aptikti konfliktai pagal kategoriją pateikti 4 lentelėje.

4.4 lentelė. Sistemoje aptikti ir automatiškai išspręsti konfliktai pagal kategoriją

Konflikto kategorija	**Konfliktų kiekis**	**Išspręsti konfliktai**	**Neišspręsti konfliktai**
Failo turinio papildymas	3	3	0
Failo turinio šalinimas	2	2	0
Kompleksinis konfliktas	18	3	15
Iš viso	23	8	15

Kaip matoma iš lentelės sistemoje per tyrimo laikotarpį iš viso buvo 23 konfliktai, iš kurių 3 buvo failo turinio papildymo, 2 failo turinio šalinimo ir 18 kompleksiniai konfliktai. Išspręstų konfliktų iš viso buvo 8, neišspręstų – 15.

Pagal gautus rezultatus yra nustatomas konfliktų sprendimų tikslumas bendram konfliktų kiekiui ir kiekvienai konfliktų kategorijai. Sprendimo tikslumas yra dydis, kuris nusako teisingų sprendimų ir visų konfliktų santykį išreikštu procentais. Jis skaičiuojamas taip:

$$T = \frac{k_s}{k} \times 100 \tag{1}$$

kur T – tikslumas, k_s – teisingų sprendimų skaičius iš konfliktų sprendimo sistemos, k – konfliktų skaičius.

Pagal formulę gaunamas konfliktų sprendimo tikslumas kiekvienai kategorijai. Jis pateiktas 4 lentelėje.

4.5 lentelė. Automatinių konfliktų sprendimo realizacijos tikslumas

Kategorija	Failo turinio papildymas	Failo turinio šalinimas	Kompleksinis konfliktas	**Bendras**
Tikslumas	100,00%	100,00%	16,67%	34,78%

Kaip matoma iš lentelės visi failo turinio ir failo šalinimo konfliktai yra automatiškai išsprendžiami. Tačiau šie konfliktai sudaro tik 21,7 % visų atsiradusių konfliktų per 3 mėnesių laikotarpį. Kompleksinių konfliktų buvo išspręsti 16,67 % pritaikius abstrakčiosios sintaksės medžio algoritmą. Didelė dalis kompleksinių konfliktų atsidaro, nes keičiama logika pačiuose metoduose vienu metu ir tokių konfliktų nėra įmanoma automatiškai išspręsti, nes tai reikalauja automatinio kodo rašymo. Pavyzdžiui, jei konfliktas atsiranda to pačio metodo toje pačioje kodo eilutėje, kurioje du

programuotojai pakeičia logiką skirtingai, tada nėra aišku ar abiejų pakeitimų išsaugojimas yra teisingas sprendimas. Neišsprendžiamo konflikto supaprastintas pavyzdys pateikiamas 15 paveiksle.

```
<<<<<<< HEAD (Current Change)
    public static Integer returnCalculation(Integer x, Integer y, Integer z){
        return x + y + z;
=======
    public static Integer returnCalculation(Integer x, Integer y, Integer a){
        return x + y + a;
>>>>>>> origin/testtwo (Incoming Change)
    }
```

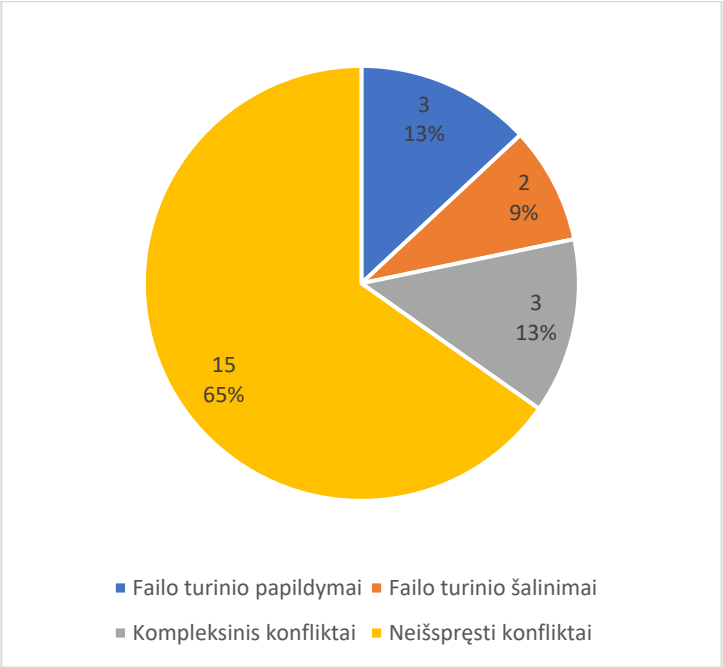
4.15 pav. Kompleksinio konflikto supaprastintas atvejis

Iš 15 paveikslo galima matyti, kad šiam konfliktui yra galima pritaikyti daug skirtingų sprendimų:

- Jei kintamieji z ir a reiškia tą patį, tada teisingas sprendimas yra x + y + z arba x + y + a;
- Jei kintamieji z ir a yra skirtingi koeficientai, tada galimas sprendimas yra x + y + z + a;
- Jei metodo logika įgyvendina sudėtingą verslo taisyklę ir abu koeficientai negali egzistuoti arba šie koeficientai turi būti atimami vienas iš kito, arba kita komplikuota logika, tada galimas sprendimas gali būti įgyvendinamas tik rankiniu būdu.

Taigi nepateikus papildomo konteksto automatinis konfliktų sprendimas nėra paprastai įgyvendinamas ir egzistuoja daug panašių scenarijų, kurių prototipas negali išspręsti.

Pagal prieš tai pateiktą realizacijos tyrimą pateikiama bendra konfliktų sprendimo tikslumo diagrama 16 paveiksle.



4.16 pav. Bendras konfliktų sprendimo tikslumas pagal kategoriją

Kaip matoma iš 16 pav. diagramos ir 5 lentelės bendras automatinių konfliktų sprendimų tikslumas yra 34,78 %. Norint padidinti konfliktų sprendimo tikslumą reikia išskaidyti kompleksinius konfliktus į daugiau kategorijų ir nustatyti galimus algoritmus jų sprendimui. Taip pat kai kurių kompleksinių konfliktų sprendimas nėra įmanomas vien iš pateikiamo kodo. Tokiam sprendimui reikalingas metodų kontekstas, kad būtų galima nustatyti tinkamą konfliktų sprendimą.

4.4 Ketvirtojo skyriaus apibendrinimas ir pagrindiniai rezultatai

Šiame skyriuje buvo suprojektuotas versijų kontrolės uždavinių automatizacijos, jos realizuotos ir ištirta kūrimo proceso automatizacija ir automatinis konfliktų sprendimas. Pateiktos automatizacijos buvo realizuotos naudojantis „GitHub“ teikiama „GitHub Actions“ platforma ir „Python“ skriptu.

Automatinis pakeitimų perkėlimas iš versijų kontrolės sistemos realizuotas prašymo perkelti pakeitimus sukūrimo ir sujungimo įvykio metu. Kai sukuriamas prašymas, pakeitimai validuojami, o kai prašymas sujungiamas ir validacija yra teigiama, pakeitimai yra sujungiami.

Automatinis pakeitimų perkėlimas į versijų kontrolės sistemą realizuotas paleidžiant skriptą kiekvieną parą 24 val. nakties. Šis skriptas sukuria naują atšaką su visais pakeitimais ir juos automatiškai sujungia su „TEST“ atšaka.

Automatinis testavimas realizuotas paleidžiant skriptą kiekvieną parą 2 val. nakties. Šis skriptas atitinkamai „TEST“ ir „UAT“ aplinkoms paleidžia jų atšakų automatinius testus ir išveda rezultatus į komunikacijos kanalą „Slack“ ir el. paštu. Kadangi testavimas vyksta po automatinio pakeitimų perkėlimo, tai ištestuojami ir tiesioginiai aplinkų pakeitimai.

Automatinis testavimo užduočių kūrimas realizuotas paleidžiant skriptą prašymo perkelti pakeitimus sukūrimo ir sujungimo įvykio metu. Kai sukuriamas prašymas, automatiškai sukuriama techninio testavimo užduotis „Jira“ projektų valdymo sistemoje ir pranešama į komunikacijos kanalą „Slack“, o kai pakeitimai sujungiami, tada automatiškai sukuriama funkcinio testavimo užduotis „Jira“ projektų valdymo sistemoje ir pranešama į „Slack“.

Tiriamojoje dalyje buvo atliktas eksperimentinis tyrimas kūrimo proceso automatizacijai ir automatiniam konfliktų sprendimui. Kūrimo proceso automatizacijos tyrimo metu buvo nustatyta, kiek laiko dviejų savaičių komanda praleidžia prie nustatytų versijų kontrolės uždavinių prieš automatizaciją ir kiek laiko praleidžia kitame dviejų savaičių laikotarpyje po automatizacijos. Laiko sąnaudos sumažėjo iš 27 val. į 5 val. Automatinių konfliktų sprendimo realizacijai buvo sekami 3 mėnesius versijų kontrolės sistemoje atsiradę konfliktai ir pritaikyta automatizacija konfliktų sprendimui. Buvo aptikti 23 konfliktai, iš kurių 8 buvo automatiškai išspręsti. Konfliktų sprendimo realizacijos tikslumas 34,78 %. Norint padidinti tikslumą, reikia nustatyti daugiau konfliktų sprendimo kategorijų ir joms pritaikyti atitinkamus algoritmus.

4.5 Ketvirtojo skyriaus išvados

1. Pagal siūlomus automatizuoti versijų kontrolės uždavinius buvo suprojektuotas automatinis pakeitimų perkėlimas į „Salesforce“ aplinkas, automatinis pakeitimų perkėlimas į versijų kontrolės sistemą, automatinis testavimas, automatinis testavimo užduočių kūrimas ir automatinis konfliktų sprendimas.
2. Atlikus kūrimo proceso automatizacijos eksperimentinį tyrimą, kūrimo proceso laiko sąnaudos sumažėjo iš 27 val. į 5 val. dviejų savaičių laikotarpyje. Automatizacijos sumažino praleidžiamą laiką prie kūrimo procese vykdomų veiksmų 81,48 %.
3. Atlikus automatinių konfliktų sprendimo realizacijos eksperimentinį tyrimą, nustatyta, kad konfliktų sprendimo tikslumas yra 34,78 % Iš aptiktų 5 failo turinio papildymo ir turinio šalinimo konfliktų visi konfliktai buvo išspręsti automatiškai, tačiau šie konfliktai tyrimo metu sudarė tik 21,7 % visų sistemoje atsiradusių konfliktų. Iš aptiktų 18 kompleksinių konfliktų 3 buvo išspręsti automatiškai taikant abstrakčiosios sintaksės medžio algoritmą.

IŠVADOS

1. Atlikus su versijų kontrolės uždaviniais susijusių mokslinių darbų literatūros analizę, nustatyta, kad egzistuoja daug įrankių versijų kontrolės uždaviniams automatizuoti, tačiau automatizacija nėra galima, jei sistemoje yra aptinkamas konfliktas failų jungimo metu, o konfliktų automatizacija nėra plačiai ištirta tema. Todėl automatiniam konfliktų sprendimui įgyvendinti reikia gilesnių tyrimų ir metodų kūrimo.
2. Probleminės srities analizė parodė, kad „Salesforce“ verslo sistemai galima automatizuoti pakeitimų perkėlimą tarp versijų kontrolės sistemos ir „Salesforce“ aplinkų, testavimą, testavimo užduočių kūrimą ir konfliktų sprendimą. Buvo nustatyta, kad visiškas automatinis konfliktų sprendimas nėra įmanomas, bet nustatytos galimos konfliktų kategorijos, kurioms pasiūlyti galimi sprendimai: failo turinio papildymas, failo turinio šalinimas ir kompleksinis konfliktas.
3. Realizavus automatizacijas „Salesforce“ verslo sistemai, buvo atliktas kūrimo proceso automatizacijos eksperimentinis tyrimas ir nustatyta, kad kūrimo proceso laiko sąnaudos dviejų savaičių laikotarpyje sumažėjo iš 27 val. į 5 val. Praleidžiamas laikas prie nustatytų uždavinių sumažėjo 81,48 %.
4. Atlikus automatinių konfliktų sprendimo realizacijos eksperimentinį tyrimą, nustatyta, kad konfliktų sprendimo tikslumas yra 34,78 %. Iš aptiktų 5 failo turinio papildymo ir turinio šalinimo konfliktų visi konfliktai buvo išspręsti automatiškai, tačiau šie konfliktai tyrimo metu sudarė tik 21,7 % visų sistemoje atsiradusių konfliktų. Iš aptiktų 18 kompleksinių konfliktų 3 buvo išspręsti automatiškai taikant abstrakčiosios sintaksės medžio algoritmą.

Realizavus kūrimo proceso automatizacijas ir automatinį konfliktų sprendimą buvo pastebėtos tolimesnių tyrimų kryptys:

1. Išskaidyti kompleksinių konfliktų kategoriją į siauresnes kategorijas, kurioms galima būtų pritaikyti konfliktų sprendimo algoritmus.
2. Didelei daliai kompleksinių konfliktų automatinis konfliktų sprendimas nėra įmanomas, nes tai reikalauja daugiau funkcionalumo konteksto, kad būtų galima sistemai suprasti, kaip turi būti tinkamai išspręstas konfliktas. Todėl reikia nustatyti galimus įrankius kontekstui pateikti.
3. Pagal konfliktų sprendimų saugyklą nustatyti kitas galimas konfliktų sprendimų kategorijas, sukurti kitus algoritmus ir sukaupus pakankamą kiekį konfliktų sprendimų pritaikyti mašininį mokymą jų sprendimui.

LITERATŪRA

Brindescu, C., Codoban, M., Shmarkatiuk, S., & Dig, D. (2014). How do centralized and distributed version control systems impact software changes? *Proceedings - International Conference on Software Engineering, CONFCODENUMBER*, 322–333. https://doi.org/10.1145/2568225.2568322

Brindescu, C., Ramirez, Y., Sarma, A., & Jensen, C. (2020). Lifting the Curtain on Merge Conflict Resolution: A Sensemaking Perspective. *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, 534–545. https://doi.org/10.1109/ICSME46990.2020.00057

Burke, E. K., & Kendall, G. (2014). Search methodologies: Introductory tutorials in optimization and decision support techniques. *Springer US*. https://doi.org/10.1007/0-387-28356-0

Chen, I. J., & Popovich, K. (2003). Understanding customer relationship management (CRM) People , process and technology. *Business Process Management Journal*, *9*. https://doi.org/10.1108/14637150310496758

De Alwis, B., & Sillito, J. (2009). Why are software projects moving from centralized to decentralized version control systems? *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering, CHASE 2009*, 36–39. https://doi.org/10.1109/CHASE.2009.5071408

Dinella, E., Mytkowicz, T., Svyatkovskiy, A., Bird, C., Naik, M., & Lahiri, S. K. (2021). *DeepMerge: Learning to Merge Programs*. http://arxiv.org/abs/2105.07569

GNU Software Manual. (2021).

Greene, G. J., & Fischer, B. (2015). Interactive tag cloud visualization of software version control repositories. *2015 IEEE 3rd Working Conference on Software Visualization, VISSOFT 2015 - Proceedings*, 56–65. https://doi.org/10.1109/VISSOFT.2015.7332415

Henschel, J., & Di Francesco, M. (2020). *A comparison study of managed CI/CD solutions*. https://git.cubieserver.de/jh/cs-e4000-seminar/raw/commit/a9b2cba6ec2c187f7a9b34c1f4a9ab8dc5097f71/cs-seminar_2020-04-11.pdf

Ji, T., Chen, L., Yi, X., & Mao, X. (2020). Understanding merge conflicts and resolutions in git rebases. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE, 2020-October*, 70–80. https://doi.org/10.1109/ISSRE5003.2020.00016

Kasi, B. K., & Sarma, A. (2013). Cassandra: Proactive conflict minimization through optimized task scheduling. *Proceedings - International Conference on Software Engineering*, 732–741. https://doi.org/10.1109/ICSE.2013.6606619

Lilis, Y., & Savidis, A. (2019). A survey of metaprogramming languages. *ACM Computing Surveys*, *52*(6). https://doi.org/10.1145/3354584

Lionetti, G. (2012). *What is Version Control: centralized vs. DVCS*. Atlassian Products & News. https://doi.org/10.1007/978-1-4842-6353-2_1

Lubański, M. (2019a). *3 reasons why you should use Version Control System by Mateusz Lubański Medium*. FAUN Medium. https://medium.com/@m.lubanskii/3-reasons-why-you-should-use-version-control-system-82ae27add187

Lubański, M. (2019b). *Centralized vs Distributed Version Control Systems*. FAUN Medium. https://medium.com/faun/centralized-vs-distributed-version-control-systems-a135091299f0

Malmsten, C. F. (2010). Evolution of Version Control Systems. *Comparing CENTRALIZED against DISTRIBUTED Version Control models*, 1–16.

Manohar, A. (2017). Relationship in cloud environment. *IEEE*, 1–4.

O'Neill, M., & Spector, L. (2020). Automatic programming: The open issue? *Genetic Programming and Evolvable Machines*, *21*(1–2), 251–262. https://doi.org/10.1007/s10710-019-09364-2

O'Sullivan, B. (2009). *Mercurial: The Definitive Guide* (1st leid., T. 4, Numeris 1). O'Reilly Media.

Otte, S. (2009). Version control systems. *Computer Systems and Telematics*. https://doi.org/10.1109/MS.2005.140

Ruparelia, N. B. (2010). The history of version control. *ACM SIGSOFT Software Engineering Notes*, *35*(1), 5–9. https://doi.org/10.1145/1668862.1668876

Salesforce. (s.a.-a). *Learn About the Salesforce Platform Advantage Unit Salesforce*.

Salesforce. (s.a.-b). *What is Salesforce - What does Salesforce do*. https://www.salesforce.com/products/what-is-salesforce/?d=70130000000i7zF

Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* (T. 5, p. 3909–3943). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/ACCESS.2017.2685629

Somasundaram, R. (2013). *Git: Version control for everyone. The non-coder's guide to everyday version control for increased efficiency and productivity*.

Spinellis, D. (2005). Version Control Systems. *IEEE*, *22*(5), 108–109.

Virtanen, J. (2021). *Comparing Different CI/CD Pipelines*. https://www.theseus.fi/bitstream/handle/10024/511026/Opinnaytetyo_Joni_Virtanen.pdf?sequence=2&isAllowed=y

Zolkifli, N. N., Ngah, A., & Deraman, A. (2018). Version Control System: A Review. *Procedia Computer Science*, *135*, 408–415. https://doi.org/10.1016/j.procs.2018.08.191