УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Срђан Попић

Прилог рјешењу ефикасне верификације функционалних захтјева помоћу програмских језика

ДОКТОРСКА ДИСЕРТАЦИЈА

Ментор:

проф. др Никола Теслић

Нови Сад, 2023

УНИВЕРЗИТЕТ У НОВОМ САДУ **ОБРАЗАЦ – 5а**

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА[1]

Врста рада:	Докторска дисертација
Име и презиме аутора:	Срђан Попић
Ментор (титула, име, презиме, звање, институција)	Др Никола Теслић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду
Наслов рада:	Прилог рјешењу ефикасне верификације функционалних захтјева помоћу програмских језика
Језик публикације (писмо):	Српски (ћирилица)
Физички опис рада:	Унети број: Страница_____124 Поглавља_____13 Референци_____142 Табела_____5 Слика_____17 Графикона_____0 Прилога_____0
Научна област:	Електротехника и рачунарско инжињерство
Ужа научна област (научна дисциплина):	Рачунарска техника и рачунарске комуникације
Кључне речи / предметна одредница:	Инжињерство програмских захтјева, доменски-специфични програмски језици, Верификација и Валидација, Тестирање
Резиме на језику рада:	Докторска теза анализира предлог за имплементацију верификације функционалних програмских захтева. Предмет истраживања је проналажење свих релевантних стандарда, препорука и најбољих пракси, а затим особина и функционалности прате дате стандарде и препоруке у области верификације. Истраживање потом проналази постојећа релевантна решења и њихову усклађеност са датим особинама и функционалностима. Резултат истраживања је развој доменски-специфичног језика за верификацију функсионалних програмских захтјева који имплементира све особине и функционалности чиме потврђује исправност концепта.
Датум прихватања теме од стране надлежног већа:	06.03.2023
Датум одбране: (Попуњава одговарајућа служба)	

[1] Аутор докторске дисертације потписао је и приложио следеће Обрасце:
5б – Изјава о ауторству;
5в – Изјава о истоветности штампане и електронске верзије и о личним подацима;
5г – Изјава о коришћењу.
Ове Изјаве се чувају на факултету у штампаном и електронском облику и не кориче се са тезом.

Чланови комисије: (титула, име, презиме, звање, институција)	Председник: др Мирослав Поповић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду Члан: др Богдан Павковић, ванредни професор, Факултет техничких наука, Универзитет у Новом Саду Члан: др Иван Каштелан, ванредни професор, Факултет техничких наука, Универзитет у Новом Саду Члан: др Михајло Савић, доцент, Електротехнички факултет, Универзитет у Бањој Луци Ментор: Др Никола Теслић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду
Напомена:	

UNIVERSITY OF NOVI SAD
FACULTY OF TECHNICAL SCIENCES

KEY WORD DOCUMENTATION[2]

Document type:	Doctoral dissertation
Author:	Srđan Popić
Supervisor (title, first name, last name, position, institution)	PhD Nikola Teslić, full professor, Faculty of Technical Sciences, University of Novi Sad
Thesis title:	Simple Framework for Efficient Development of the Functional Requirement Verification-specific Language
Language of text (script):	Serbian language (cyrilic)
Physical description:	Number of: Pages_____124 Chapters_____13 References_____142 Tables_____5 Illustrations_____17 Graphs_____0 Appendices_____0
Scientific field:	Electrical and Computer Engineering
Scientific subfield (scientific discipline):	Computer engineering and communications
Subject, Key words:	Requirement Engineering, Domain-specific languages, Verification & Validation, Testing
Abstract in English language:	The doctoral thesis analyzes the proposal for implementing the verification of functional software requirements. The subject of the research is to find all relevant standards, recommendations, and best practices, and then to examine the features and functionalities that follow the given standards and recommendations in the field of verification. The research then identifies existing relevant solutions and their compatibility with the given features and functionalities. The result of the research is the development of a domain-specific programming language for the verification of functional requitements that implements all the features and functionalities, thus confirming the correctness of the concept.
Accepted on Scientific Board on:	06.03.2023
Defended: (Filled by the faculty service)	
Thesis Defend Board: (title, first name, last name, position, institution)	President: PhD Miroslav Popović, full professor, University of Novi Sad Member: PhD Bogdan Pavković, adjunct professor, University of Novi Sad Member: PhD Ivan Kaštelan, adjunct professor, University of Novi Sad Member:PhD Mihajlo Savić, associate professor, University of Banja Luka Mentor: PhD Nikola Teslić, fullprofessor, Faculty of Technical Sciences, University of Novi Sad

[2] The author of doctoral dissertation has signed the following Statements:
5б – Statement on the authority,
5в – Statement that the printed and e-version of doctoral dissertation are identical and about personal data,
5г – Statement on copyright licenses.
The paper and e-versions of Statements are held at he faculty and are not included into the printed thesis.

Note:	

Захваљујем се Институту за Рачунарску технику и рачунарске комуникације, због пружене прилике за докторске студије, а нарочито младим колегама запосленим у бањалучком дијелу Института, који су са својом енергијом и индуковали у мени жељу и вољу да се упустим у ову авантуру.

Захваљујем се ментору др Николи Теслићу, као и др Богдану Павковићу и др Ивану Каштелану, који су својим савјетима довели ову сагу до срећног краја. Посебно се захваљујем др Милану Бјелици чије су мудре, стратешке одлуке довеле до објављивања рада у часопису, као и професорима са Електротехничког Факултета у Бањалуци на свесрдној подршци.

На крају, највише се захваљујем породици, а првенствено супрузи Драгани, која је била уз мене и вјеровала у позитиван исход овог путовања и у оним тренуцима када ни ја нисам вјеровао у успјех.

Сажетак

Када је у питању процес спецификације, валидације и верификације функционалних програмских захтјева, постоји читав низ алата за моделовање, анализу и валидацију. Животни циклус развоја софтверског производа и подразумјева коришћење великог броја различитих алата у разним фазама развоја. Међутим, када је у питању верификација програмских захтјева, алати који је подржавају или су превише комплексни или имају изузетно узак домен примјене. Такође, са становишта наручиоца потребна је и независна провјера програмских захтјева.

У овој дисертацији се истражују особине и функционалности потребне при спецификацији и верификацији функционалних програмских захтјева, затим могућности за креирање произвољног верификационог програмског језика таквог да кориснику, био он наручилац или испоручилац циљаног програма, омогућује верификацију свих функционалних програмских захтјева, без обзира на алате којима су ти захтјеви специфицирани, моделовани, на крају, и имплементирани. Истраживање прати IEEE препоруке, стандарде и *најбољу праксу* када је у питању област инжењеринга програмских захтјева. Ово посебно обухвата особине као што су: тачност, комплетност, сљедивост, зависност, битност (тежину односно стабилност) као и јединственост – које нису у фокусу процеса валидације.

Најзад, у циљу потврде концепта, истражују се могући правци развоја језика за верификацију функционалних програмских захтјева, како би се доказало да је језик који прати све потребне препоруке, стандарде и *најбољу праксу* остварив. Концепт се може користити за развој програмског језика за верификацију функционалних програмских захтјева насталих у било којем животном циклусу развоја производа, и без обзира на саму репрезентацију програмских захтјева. Ово ће бити презентовано у пар одабраних примјера.

Abstract

When it comes to the process of specification, validation and verification of functional software requirements, there exists a whole range of tools for modeling, analysis, and validation. The life cycle of software product development includes the use of many different tools at various stages. However, when it comes to the verification of program requirements, the tools that support it are either too complex or have a very narrow scope of application. Also, from the customer's point of view, an independent verification of the program requirements is necessary.

In this dissertation the properties and functionalities needed for specification and verification of the functional software requirements, together with the possibilities of the creation of an arbitrary verification programming language. This verification programming language is such that the user, whether he is the customer or the supplier of the target program, can verify all of the functional program requirements, regardless of the tools with which these requirements are specified, modeled, and finally implemented. The research follows IEEE recommendations, standards, and best practices when it comes to the field of software requirements engineering. This especially includes features such as accuracy, completeness, traceability, dependency, severity as well as uniqueness - which are not the focus of the validation process.

Finally, in order to proof the concept, the dissertation researches the options to a solution that implements verification-specific languages in order to prove that a language that follows all the necessary recommendations, standards and best practices is achievable. The presented concept can be used to develop a programming language for the verification of functional programming requirements created in any product development life cycle, regardless of the representation of the programming requirements itself. This is presented in select examples.

Садржај

Листа слика

Листа табела

Листа скраћеница

AADL – Architecture Analysis and Design Language

ACE – Attempto Controlled English

APE – Attempto Parsing English

API – Application Programming Interface

AST – Abstract Syntax Tree

ATAP – Accelerating Test Automation Platform

AUTOSAR – Automotive Open System Architecture

CAST – Computer-Aided Specification and Testing

CFG – Control Flow Graph

CNL – Controlled Natural Languages

CPL – Computer-Processable Language

DRS – Discourse Representation Structure

DSL – Domain-Specific Language

EBNF – extended Bakus-Naur Form

ETSI – European Telecommunications Standardization Institute

GPL – General Purpose Language

GRL – Goal-oriented Requirement Language

IEEE – Institute of Electrical and Electronics Engineers

ISO/IEC – International Organization for Standardization/International Electrotechnical Commission

LPDA – Logic Programming with Defaults and Argumentation Theory

NLP – Natural Language Processing

PMBOK – Project Management Body of Knowledge

RAD – Rapid Application Development

RE – Requirement Engineering

RM-RNL – Requirement Modeling with Restricted Natural Language

SAE – Society of Automotive Engineers

SCS – Safety-Critical Systems

SWRL – Semantic Web Rule Language

TPTP – Thousands of Problems for Theorem Provers

TSL – Test Specification Language

TTCN-3 – Testing and Test Control Notation v3

UCM – Use Case Maps

URN – User Requirement Notation

V&V – Validation and Verification

VHDL – VHSIC-HDL – Very High Speed Integrated Circuit Hardware Description Language

VSL – Verification-Specific Language

XML – eXtensible Markup Language

XPath – XML Path language

YAML – Yet Another Markup Language

1 Увод

A good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language. Guy Steele [1].

Животни циклус развоја софтверског производа је концептуални оквир који дефинише задатке који се требају извршити у свакој фази датог животног циклуса [2]. Сврха животног циклуса развоја софтверског производа је таква да испоручи програм високог квалитета. Неке од методологија које се користе у развоју софтвера су сљедеће:

- *Agile* развој.
- *Continuous integration* – континуална интеграција.
- Итеративни и инкрементални развој.
- Rapid application development – RAD – рапидни развој апликације.
- Спирални модел развоја
- *Waterfall* модел, са изведеним моделима:
    - V-модел развоја
    - Структурна анализа система и метод дизајна
- Развој вођен понашањем
- Модел хаоса ...

Свака од наведених методологија животног циклуса развоја укључује скуп процеса који се најчешће зову једним именом: Верификација и Валидација (V&V – Validation and Verification) [3]. Основна сврха процеса V&V јесте да обезбиједи да се користе исправни процеси како би се изградио исправан и квалитетан систем [4].

Различити животни циклуси, имајући различите приоритете развоја (нпр. прво трошковни ризик, па тек онда квалитет) имају и различите приоритете процеса V&V као и различите везе између V&V и других процеса. У било којем случају, они су неизбјежан дио развоја софтвера без обзира на њихово мјесто у самом животном циклусу развоја.

Термини верификације и валидације се често користе како би се описао исти скуп процеса, међутим, постоји суптилна разлика између ова два појма дефинисана у PMBOK (*Project Management Body of Knowledge*) водичу [5] :

- **Валидација** представља скуп процеса који обезбјеђују да производ, услуга или систем одговарају потребама корисника-наручилаца, као и осталих заинтересованих страна. Често, валидација, укључује и процес прихватања и провјере да ли производ одговара кориснику.
- **Верификација**, насупорт томе, је **евалуација** да ли су производ, услуга или систем усклађени са прописима, захтјевима, осталим спецификацијама, као и наметнутим условима.

Чињеница да је верификација у ствари евалуација говори у прилог томе да је извршавање линија програмског кода најприближнији, али не и једини начин да се верификација захтјева и оствари.

Друга, корисна дефиниција верификације каже да је верификација програмских захтјева дио инжењеринга програмских захјтева који се састоји од тестирања стварне имплементације наспрам *прецизне спецификације* [6]. Термин *прецизна спецификација* се дефинише као спецификација која има упориште и објашњење у циљевима, функцијама али и ограничењима стварног свијета [7]. То значи да се **верификација** одвија **тестирањем** производа како би се потврдило оно што је специфицирано програмским захтјевом. Да би се оваква верификација извршила потребно је да се и систем извршава. Иако је на први поглед верификација тестирањем је исто што и тестирање, то је само дјелимично истина, јер је верификација окренута према програмским захтјевима и проистиче само из програмских захтјева. Верификација своју покривеност заснива искључиво на програмским захтјевима а никако на програмском коду. Сама верификација је много формалнија од тестирања. Тестирање представља процес који се користи да би се утврдила исправност, правилност у функционисању и тиме обезбиједио квалитет крајњег производа, а верификација је скуп систематичних процедура анализе прегледа и тестирања у свим фазама животног циклуса [8].

Верификација програмских захтјева се може постићи на још један начин: анализирањем модела система [9] или било којег његовог дијела. Оваква верификација

се зове **верификација анализирањем**, а неријетко и **провјера модела** (*Model Checking*) [10], јер је потребно изградити модел система а затим над таквим моделом вршити верификацију. Примарно, овај тип верификације је кориштен за верификацију хардвера, који се може представити апстрактним моделом коначне машине стања, међутим, користи се и за верификацију софтверских система [11] а посебно руковаоца, уграђених програмских система за рад у реалном времену, као и сигурносних алгоритама. Овај начин верификације је погодан за системе који још нису дефинисани или за системе који нису доступни. Системи, чије извршавање није једноставно или није јефтино, чија употреба захтијева доста времена, ризика или неких других ресурса су такође погодни за овакву врсту верификације [12]. Потешкоће које произилазе из верификације анализирањем су потреба за додатним знањима о алатима за моделирање, специфичним језицима који су по правилу комплексни за кориштење, посебно наручиоцима. Додатна ограничења верификације анализирањем, као што су симулациона ограничења, биће објашњена касније. Ово истраживање ће бити фокусирано на верификацију тестирањем.

Верификација програмских захтјева припада области која се назива инжењеринг захтјева (енг. *Requirement Engineering – RE*). Инжењеринг захтјева је шири, мултидисциплинарни концепт, усресређен на човјека, који посредује између свих заинтересованих страна које учествују у процесу креирања, управљања, анализе и одржавања програмских захтјева и као такав обухвата и верификацију програмских захтјева. Инжињеринг захтјева спаја многе дисциплине као што су когнитивна психологија, антропологија, социологија, лингвистика, онтологија, али ипак верификација програмских захтјева припада софтверском инжињерингу.

Дисертација истражује које од особина и функционалности је потребно да посједује верификациони програмски језик (Verification-Specific Language – VSL) способан за верификацију функционалних програмских захтјева, затим који су постојећи програмски језици и алати кориштени за ову врсту верификације, као и колико су они усклађени са датим особинама и функционалностима. На крају, истражено је да ли је могуће развити програмски језик за верификацију функционалних захтјева који је у потпуности усклађен са траженим особинама и функционалностима. У ову сврху, а у циљу потврде концепта, приложено је рјешење

окружења које ће бити способно да подржи креирање текстуалног верификационо-специфичног програмског језика интерпретерског типа.

У наставку, послије представљања мотивације и доприноса докторске дисертације, биће изложени резултати истраживања из области инжењеринга захтјева, како би се прво уочило који програмски захтјеви улазе у истраживање, а затим како би се препознале особине и функционалности препоручене или стандардизоване од стране IEEE и других тијела. Након тога, биће представљене особине DSL програмских језика, њихове различите репрезентације, са посебним освртом на предности, али и трошкове који произилазе из развоја и кориштења оваквих програмских језика. Из ова два истраживања уочава се метрика која ће бити кориштена у овом раду, а која је потребна како би се изградио VSL. На основу ове метрике, изведена је и теза докторског истраживања.

Допринос докторске дисертације је нарочито видљив у поглављу у којем се истражују алати који се користе за верификацију функционалних програмских захтјева, као и за тестирање софтвера, а развијени су као програмски језици. Проучавају се и програмски језици за тестирање, програмски језици за специфицирање функционалних програмских захтјева, као и контролисани природни језици који су додатно специјализовани како би могли послужити у сврху верификације.

Резултати овог дијела истраживања су сумирани у сљедећем поглављу у облику упоредне табеле. Осим што је видљив обзор науке и технике у овој области, поређењем различитих рјешења, уочавају се и заједничке особине и функционалности свих истражених алата, али и недостаци у односу на особине и функционалности наведене у метрици, а које би морао посједовати један верификациони програмски језик.

Имајући у виду резултате овог дијела истраживања, у наставку се даје прилог рјешењу развоја **Окружења** које би се користило за развој различитих VSL, а у сврху доказивања да је могуће развити верификациони програмски језик који је у потпуности усаглашен са IEEE стандардима, препорукама и *најбољим праксама*. Предлажу се основни појмови и операције неопходне за имплементирање како би се подржали сви потребни аспекти програмског језика. Такође се објашњавају процеси верификације, као и различити нивои имплементације, пошто је могуће неке функционалности и особине оставити имплементиране на уобичајен начин, али и прилагодити их другачијим верификационим процесима и метамоделима специфичним за дати домен у

којем се функционални програмски захтјеви верификују. Како су међу кључним особинама комплексност и проширивост, у рјешењу се предлажу и начини и могућности функционалних проширења, укључивањем већ готових библиотека. Ту су, такође и четири имплементације VSL, које на пластичан начин приказују све предности оваквог приступа. Након тога, даје се преглед резултата другог дијела истраживања у виду четири имплементације VSL и успоредна мјерења која су произашла из ових имплементација. На крају представљен је закључак цијелог истраживања.

2 Мотивација и допринос докторске дисертације

Процес верификације програмских захтјева је процес који укључује најмање двије заинтересоване стране: наручиоца и добављача. Добављач има своје начине и алате да верификује дате захтјеве. Различити животни циклуси развоја софтвера нуде многе алате или препоруке за то, а пошто су добављачи вични у програмирању, готово је сигурно да посједују рјешење које је у складу са њиховом изабраном филозофијом развоја. Наручилац, са друге стране, углавном није упознат са овим алатима, а упознавање би сигурно одузело много времена. Такође, како би обје стране биле сигурне да је производ у складу са захтјевима, укључивање треће , независне стране би било неопходно. Многе релевантне компаније, као што су *Accenture, IBM,* неке од највећих банака, осигуравајућих компанија и државних агенција у Сједињеним Америчким Државама, су користиле независну трећу страну за верификацију и тестирање [13]. Независна, трећа страна – у неким случајевима је и сертификована – би извршила верификацију производа по датим захтјевима дајући своју крајњу оцјену. Овако нешто би готово сигурно драстично повећало крајњу цијену производа.

Наручилац би сигурно прихватио верификацију помоћу програмског језика или окружења који познаје. Ово би убрзало и развој алата јер не би било потребе за додатним упознавањем. Такође, уколико би се могла искористити технологија која се користила при спецификацији захтјева, цијели систем развоја би био и робуснији (измјене на захтјевима би се одмах пресликале и на верификацију) а технологија позната наручиоцу.

Ипак, наручилац би са сваким новим производом који наручује, највјероватније морао развијати нови модел или језик за верификацију. Често, нови пројекат са собом носи и нову терминологију, због чега је немогуће поново користити већ једном створени програмски језик за верификацију у неком другом окружењу. Наручилац, такође мора бити увјерен да алат који је развио подржава све препоруке, стандарде и *најбољу праксу*. Сви ови аргументи иду у прилог томе да се истражи:

- Које су то особине и функционалности које треба да посједује неки програмски језик како би задовољио препоруке, стандарде и најбољу праксу у области верификације функционалних захтјева? Овом скупу

додати и ниво комплексности, али само као референтну особину, јер је то полазна тачка: *направити што једноставније окружење за наручиоца*.

- Који су постојећи алати или програмски језици који се користе за верификацију функционалних захтјева?
- Да ли је уопште могуће развити програмски језик за верификацију функционалних захтјева који садржи дате особине и функционалности? У сврху потврде концепта, развити интерпретерски аутомат који подржава креирање верификационих програмских језика уз обавезну имплементацију датих особина и функционалности.

Истраживању области верификације програмских захтјева посвећено је цијело једно поглавље овог документа које детаљније описује тренутне домете технологије и у области развоја DSL језика за програмске захтјеве (били они само за спецификацију захтјева, за верификацију, па и тестирање), а дјелимично и у области развоја алата за верификацију програмских захтјева.

Изоловање скупа обавезних функционалности и особина верификационог процеса, истраживање алата који се користе у овој области као и упоредна анализа заједно са потврдом концепта да је могуће развити DSL за верификацију функционалних програмских захтјева који имплементира све битне функционалности и особине је допринос ове докторске дисертације.

3 Инжињеринг програмских захтјева

Инжињеринг програмских захтјева је, као што је раније наведено мултидисциплинарна област која је медијатор између заинтересованих страна у процесима везаним за програмске захтјеве. Овај концепт је процес у којем се формулише, документује и одржава (проширује и мијења) скуп захтјева [14] у ширем значењу, не само програмских захтјева. Ово истраживање области инжињеринга програмских захтјева, биће фокусирано искључиво на спецификацију и верификацију програмских захтјева са аспетка софтверског инжењеринга.

Најгрубља подјела програмских захтјева је подјела на функционалне и не-функционалне [15]. Нефункционални захтјеви су захтјеви који описују особине и ограничења система, као што су брзина, корисност, безбиједност, поузданост, робусност, брзину опоравка, могућности одржавања, итд... Са друге стране, како је функција у ствари понашање обезбијеђено од стране једне или више компоненти система [16], тако су функционални захтјеви они захтјеви који дефинишу понашање система или његових елемената. Функционални захтјеви специфицирају релацију понашања између улаза (стимуланса) и излаза (одговора). Иако функционални, ови захтјеви не смију помињати било какву технологију и морају бити независни од имплементације и дизајна [17].

Функционални захтјеви могу бити класификовани по својим особинама на семантичке и несемантичке [18]. Несемантичке особине су структуралне (захтјеви који описују структуру), синтаксичне (које описују правила и сам језик) и прагматичке (које описују стварне односе са конкретним Окружењем) [19]. Семантички захтјеви су захтјеви који описују идеално знање о самом проблему [19]. Када се узме у обзир дефиниција валидације, очигледно је да се семантичке особине функционалних захтјева провјеравају валидацијом, а несемантичке верификацијом.

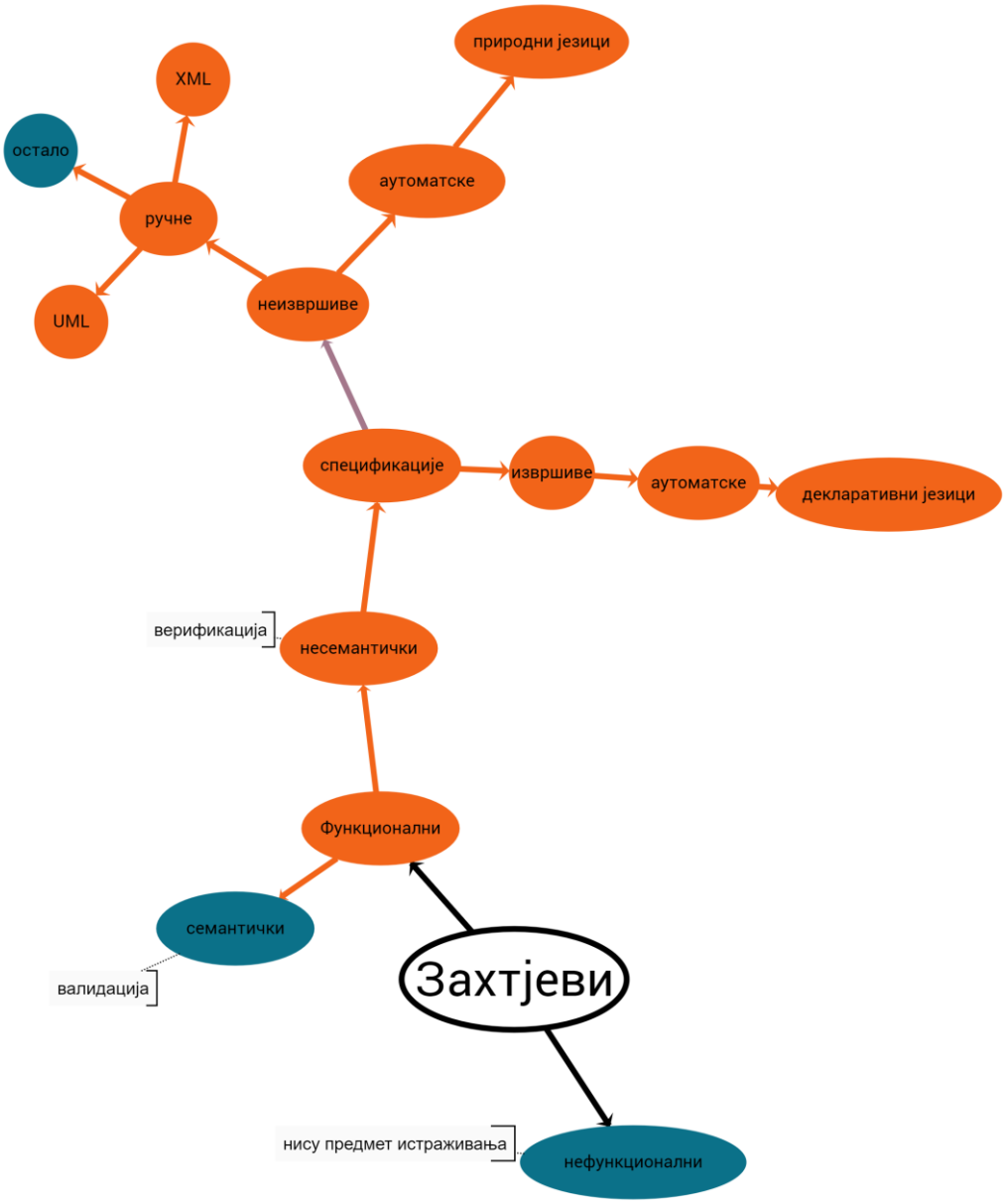
Документовање програмских захтјева јесте њихова спецификација. Формална дефиниција спецификације програмског захтјева јесте, да је то спецификација одређеног софтверског производа, програма или чак скупа програма који извршавају одређене, тачно утврђене функције у специфичном окружењу. Спецификација програмског захтјева може бити написана од стране једног или више представника добављача, као и једног или више представника наручилаца [20].

Иако се првобитно сматрало да су све спецификације функционалних несемантичких захтјева извршиве, постоје истраживања која доказују да постоје и спецификације које не могу да се изврше, а специфицирају несемантичке функционалне захтјеве [21]. Све извршиве спецификације захтјева морају бити аутоматизоване, а такође могуће је аутоматизовати и неизвршиве спецификације [18].

Начини на које је могуће аутоматизовати спецификације јесте да се напишу помоћу:

- Декларативног или формалног језика – језика који је формалан и математички, добро дефинисане семантике и дозвољава висок ниво апстракције. Декларативни језици могу бити:
    - Логички језици [22].
    - Функционални језици [23].
- Природног (контролисаног) језика – постојећи језик, на примјер енглески који је доведен у оквире строго дефинисаног језика [24].

Чак и неизвршиве спецификације, могуће је дефинисати на сљедеће начине: помоћу описних језика као што су XML и њему сличних, или језика моделирања као што су UML и слични, па и ручно, читањем, помоћу листи за провјеру и слично [25] . Слика 1 приказује ову подјелу програмских захтјева у потпуности, затим алата за спецификацију и верификацију. Очигледно је да се само за један мали дио, неизвршивих, ручних, сцпецификација несемантичких програмских захтјева, не може користити неки језик у неком капацитету.



Слика 1 - Подјела програмских захтјева и спецификација у односу на могућности верификације

3.1 Структура, аспекти и и особине захтјева и спецификација

Сваки језик који се користи за спецификацију мора се користити на такав начин да програмском захтјеву обезбиједи структуру и особине које су препорука, *најбоља пракса* или стандард.

Захтјеви имају своје карактеристике које се могу подијелити у двије перспективе: технички изводљиве и технички неизводљиве. Због своје мултидисциплинарности инжињерство захтјева уноси у саме захтјеве и карактеристике

које су лингвистичке или социолошке природе, и потребно их је изоловати како би се издвојиле карактеристике помоћу којих је могуће аутоматизовати верификацију функционалних програмских захтјева. Техника које се најчешће користи за аутоматско раздвајање карактеристика захтјева су NLP (Natural Language Processing)[26].

3.1.1 Структура спецификације програмског захтјева

Многа истраживања указују на структуру као битну компоненту спецификације али и валидације и верификације програмског захтјева, из сљедећих разлога:

- Када се користи природни језик, који није формализован, па стога, уноси доста двосмислености у саме захтјеве који воде до неспоразума у пракси, стриктно структуирана спецификација помаже да се овакве ситуације минимализују [27].
- Обрасци и шаблони су структуре које омогућују имплементацију правила и тиме омогућују да се програмски захтјеви квалитетније и коректније и валидирају и верификују [28].

Структура функционалног програмског захтјева, када се изолује нетехничка компонента програмског захтјева, може имати три варијанте, од најкомпликованије до најједноставније [20]:

- <**услов**> <субјекат> <**акција**> <објекат> <**ограничење**>

ПРИМЈЕР: *Када је примљен сигнал X* (услов)*, систем* (субјекат) *ће подесити* (акција) *трећи бит регистра* (објекат) *за 2 милисекунде* (ограничење)

- <**услов**> <**акција** или **ограничење**> <вриједност>

ПРИМЈЕР: *Уколико је подешен трећи бит регистра* (услов) *биће детектоване вриједности сензора* (акција или ограничење) *у опсегу од 1 до 10 њутна*.(вриједност)

- <**субјекат**> <**акција**> <вриједност>

ПРИМЈЕР: *Билинг систем* (субјекат) *ће приказати неплаћене рачуне* (акција) *опадајућем редослиједу датума доспјећа* (виједност)

Сваки од ових дијелова структуре је у суштини много комплекснији од овог једноставног приказа, ипак неке особине се могу уочити из њих.

Услови или тачније, **предуслови** морају бити мјерљиви атрибути, а у зависности од услова, захтјев ће у одређеним случајевима бити потребан, док у другим неће, што значи да и приликом верификације, уколико одређени услови нису задовољени, неће бити потребно верификовати одређене захтјеве. Услови, такође могу да **групишу одређене захтјеве**, иако, као што ће бити више ријечи касније, и декомпозиција ствара **релацију груписања**.

Акција – описује понашање система, дијела система или веза између дијелова система. Акција садржи и лексички квантификатор **нивоа озбиљности** самог захтјева. Када су у питању препоруке за спецификацију на енглеском језику, ови квантификатори су сљедећи: *shall, should, will, may*. У питању је честа препорука али могуће је повећати број нивоа на такав начин да ови лексички квантификатори буду само класа озбиљности помоћу које се долази до коначног нивоа озбиљности [29].

Ограничења се односе и на дизајн и на имплементацију, али и на саме процесе у систему. Ограничења се могу примијенити на групу програмских захтјева, или бити у вези са неким постојећим програмским захтјевом. Она стварају и везу **зависности** између два или више програмских захтјева. У таквим случајевима, уколико један захтјев није прошао верификацију, остали који зависе од њега не могу бити верификовани.

Посљедица чињенице да програмски захтјеви могу зависити једни од других, доводи верификацију у **проблем цикличне зависности** [30], у којој, на примјер, захтјев А зависи од захтјева Б, који опет зависи од захтјева В, који зависи од захтјева А и слично. Верификација у овом случају није могућа, стога програмски језик за верификацију функционалних програмских захтјева мора бити у стању да провјери да ли релације између захтјева могу проузроковати цикличну зависност и да је примијете прије извршавања верификације. Детаљније о проблему цикличности у поглављу *3.4* Усмјерени ациклични граф зависности.

3.1.2 Аспекти функционалног програмског захтјева

Иако структура програмског захтјева даје осјећај да је и сам програмски захтјев једнодимензионалан, он је у ствари тродимензионалан и може да се посматра из три аспекта [27]:

1. *Аспект података*: Са аспекта података, статичка и структурална перспектива програмског захтјева долази у фокус. Дефинисање скупова и структура улазних и излазних података, као и начина коришћења података и међусобних **релација зависности**.

2. *Функционални или акциони аспект*: Функционални аспект се фокусира на динамичку обраду података од стране система или неког његовог дијела (нпр. функције). У питању је ток самих података као и редослијед процесирања података, односно **конзистентност** самог процеса.

3. *Аспект понашања*: Аспект понашања је фокусиран на информације о систему, његовом позиционирању у околини, као и стањима у којима је могуће да се нађе. Описане су све реакције на све побуде из околине у било којем стању да се систем налази и транзиције приликом ових побуда, као и ефекти које ће систем или неки његов елеменат имати на окружење. Покривањем свих могућих стања, побуда, транзиција и повратних ефеката овај аспект се фокусира на **комплетност**.

3.1.3 Особине спецификације програмског захтјева

И структура, и аспекти обликују програмски захтјев, и на одређени начин, дају темеље за особине програмског захтјева. Особине техничке перспективе које треба посједовати добро специфициран програмски захтјев су сљедеће:

- **Тачност** – како би захтјев био тачан, мора бити специфициран на начин да је сљедив у свим правцима [20]. Ово значи да верификација програмских захтјева мора обезбиједити **сљедивост**. Уколико се креира програмски језик који може да извршава спецификацију програмских захтјева, он у својој интерпретацији мора бити у могућности да провјери тачност програмског захтјева [31], [32].

- **Недвосмисленост** – ова особина је у самој основи програмских језика, који су по својој дефиницији недвосмислени и структуирани. Без језика који је довољно формалан са једне стране, а опет недвосмислен и разумљив и за наручиоца (експерта у домену) и за добављача (софтверског експерта) са друге стране, не може бити добро

специфицираног захтјева [33]. Добро дефинисан језик премошћава јаз између ове двије стране. Ипак и добро дефинисан програмски језик не значи и могућност провјере двосмислености изворног кода [31], [34], [35]. Најбоља потврда овога јесте чињеница да компајлери дозвољавају *shadowing* (постојање истоимених, али различитих референци и у унутрашњем и у ванјском опсегу програмског кода). Ово значи да и у случају да спецификација функционалних захтјева постане извршива помоћу развијеног интерпретера или генератора, исти **неће бити у могућности да провјери недвосмисленост функционалног захтјева**. Правило је да су дозвољене недвосмислености често извор багова и једино што је могуће јесте упозоравање. Управо због овога, сви програмски језици који се користе за спецификацију и/или верификацију програмских захтјева покушавају да на неки додатни начин одговоре на појаве двосмислености [36]–[40].

- **Комплетност –** скуп програмских захтјева мора бити такав да покрива све дијелове система, као и све типове улаза (како исправних, тако и неисправних) у систем. Осим комплетности цијелог скупа захтјева, постоји и комплетност једног програмског захтјева: Захтјев је комплетан када му се ништа не може додати нити одузети јер је мјерљив и довољно добро описује дату потребу наручиоца [20], [41]. Анализу покривености је једноставно извршити над кодом програмског језика, као и провјеру синтаксе. Ниво комплетности неког програмског кода, зависи од тога колико је потребно додатних информација унијети, у виду конфигурације, улазних података или додатног програмског кода, односно проширења програмског захтјева [31]. Ипак, није увијек случај да овај додатни дио кода (или текста програмског захтјева) стварно припада функционалном захтјеву, него је више довођење система у стање у којем се обезбјеђује валидност захтјева. Уколико унутар програмског језика обезбиједимо постојање **задатака прије и послије верификације** захтјева, обезбиједићемо могућност да захтјеви буду комплетни [42]. Оно чему ови задаци служе је управо да се систем који се верификује доведе у стање у коме је могућа провјера датог захтјева,

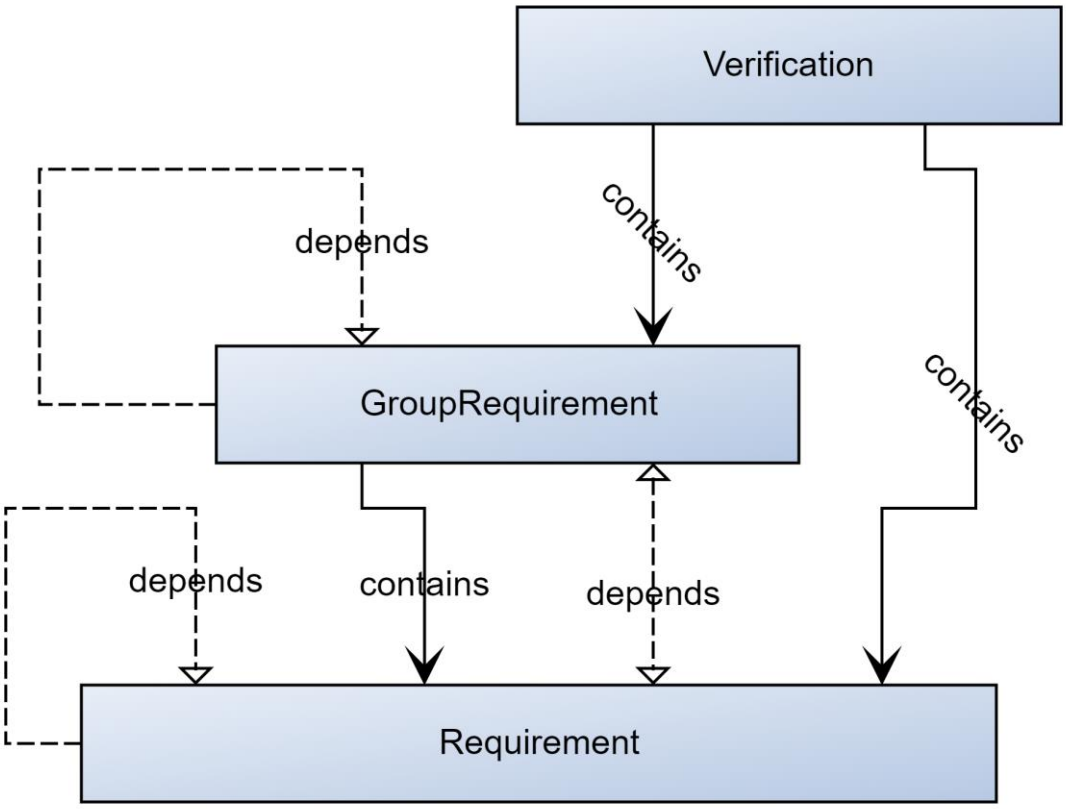
као и да се пониште посљедице верификације једног програмског захтјева на други уколико оне не постоје. Ово се често назива и компензација некомплетности [31].

- **Конзистентост –** значи да не постоји конфликт нити међу захтјевима, као ни унутар једног захтјева. Многи природни језици имају логичку неконзистентност, што је познат и чест проблем који се појављује приликом специфицирања захтјева [16]. Проблем неконзистентности унутар програмског захтјева се најчешће рјешава коришћењем контролисаних природних језика (*Controlled Natural Languages – CNL*), природних језика, са ограниченом граматиком, појмовима и ријечницима [37]. Неконзистентност између појединих програмских захтјева се рјешава увођењем механизма **предуслова**, односно, **претпоставке** као и **релације зависности** [20]. Предуслов служи да обезбиједи окружење у којем програмски захтјев постоји, а релација зависности између програмских захтјева обезбјеђује да се ограничи домен једног захтјева другим захтјевом. Користећи ове механизме избјегава се неконзистентност на начин да два наизглед конфликтна захтјева не постоје у истом домену. Са становишта верификације програмских захтјева, ово значи да уколико предуслови неког захтјева нису испуњени, захтјев се ни не верификује, као и да уколико један захтјев није испуњен или верификација није извршена, ни други захтјев који од њега зависи неће бити верификован. И структура програмског захтјева описана у претходном поглављу, са условима и ограничењима сугерише везу између програмских захтјева. Провјера конзистентности се често обезбјеђује имплементацијом шаблона [35].

- **Ниво озбиљности –** често се користи и појам **тежински фактор**, као и **фактор ризика** или **ниво битности** (енг. *severity*). Сваки захтјев посједује одређену тежину, можда и по више основа. У питању је механизам потребан при управљању ризицима [43] и јако је битан приликом укупне евалуације процеса верификације – резултата верификације. Алати за верификацију захтјева морају имплементирати ниво озбиљности као особину захтјева и укључити исту у израчунавање

укупне оцјене верификованости система. Такође препорука кориштења специфичних кључних ријечи [20] (*will, may, should, shall*), које описују утицај захтјева, као и да ли је захтјев обавезан, пожељан или необавезан указује на обавезну могућност дефинисања нивоа битности. Ниво битности биће детаљније дефинисан у наредном поглављу.

- **Провјерљивост –** мора бити могуће провјерити да ли је програмски захтјев испуњен или није. Захтјев мора да посједује начин да се докаже да је испуњен. Провјерљивост је повећана ако је функционални захтјев мјерљив [20]. Да би се неки функционални захтјев могао провјерити – мора бити могуће извршити оно што представља његову спецификацију. Ово је један од разлога због којег се и тежи креирању верификационог DSL.
- **Могућност измјене –** како би захтјев могао сигурно да се измијени, са становишта спецификације и верификације програмских захтјева, он мора бити **јединствен** и имати **могућност референцирања**. Ово постаје још битније када је у питању ситуација у којој један захтјев зависи од другог [43]. Уколико овога не би било, измјена захтјева не би могла да се изведе. Слика 2 приказује све врсте релација између захтјева, а испрекиданом линијом представљена је релација зависности. Истраживања су показала да релација зависности повећава и комплетност, конзистентност и стабилност програмског захтјева [44].



Слика 2 - Примјер различитих релација између захтјева [2]

- **Сљедивост** – најбитнија особина програмског захтјева на коју се дјелимично своде скоро све друге особине. Сљедивост је могућност остваривања и праћења различитих типова веза између програмских захтјева. Слика 2 приказује различите врсте релација између различитих врста програмских захтјева. Захтјеви са слике означени као *GroupRequirement* су захтјеви настали на вишим нивоима итерације спецификирања захтјева. При процесу декомпозиције, програмски захтјеви типа *GroupRequirement* се рашчлањују на скуп мањих, специјалнијих, међусобно искључивих захтјева (типа *Requirement*). Овим путем настају везе представљене релацијом *contains*. Појединачне активности верификације су означене као *Verification*. Слика 2, такође приказује и релације зависности.

Остале препоруке и критеријуми релевантних тијела траже од програмских захтјева да не користе уопштене, неограничене и двосмислене изразе, као што су:

суперлативи, субјективне синтагме, двосмислени придјеви (нпр: минималан, значајан, оптималан... уколико нису исти појмови стриктно дефинисани), изрази који се не могу верификовати, компаративне и негативне фразе [20]. Очигледно је да и ове препоруке иду на руку алатима за верификацију програмских захтјева.

Такође ту су и сљедеће препоруке нетехничких особина, које не могу утицати директно на верификацију, али посредно утичу:

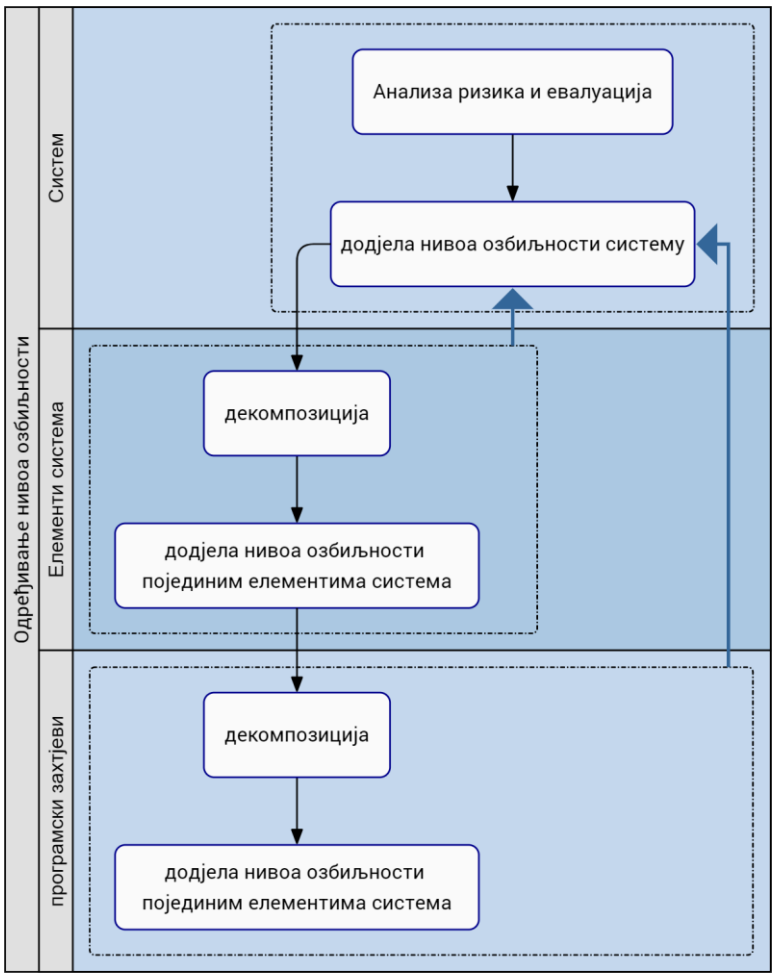
- **Неопходност** – је особина захтјева која се најбоље примјети уколико се дати захтјев изостави. Ако није могуће испунити друге захтјеве или процесе, уколико се захтјев изостави, онда је захтјев неопходан. Такође, уколико на захтјев не утиче проток времена, указује да је захтјев неопходан. Ова особина утиче на конзистентност и комплетност, односно на услове и релацију зависности.
- **Изводљивост** – није само у питању техничка изводљивост, него и правна и организациона изводљивост. Ово значи да између осталих и члан развојног тима мора бити укључен у оцјену изводљивости захтјева, како би указао на техничка ограничења имплементације.
- **Приуштивост** – у питању је финансијска изводљивост, која може да уведе додатна ограничења у одређене захтјеве.
- **Разумљивост –** захтјеви морају бити разумљиви објема странама. Због ове особине, приликом дефинисања захтјева, креирају се рјечници појмова – појмовници, који су касније корисни у креирању метамодела програмског језика за верификацију.

3.2 Нивои озбиљности програмских захтјева

Озбиљност програмског захтјева је везана за ризик. Ризик се у ISO/IEC стандарду дефинише као комбинација вјероватноће догађаја и његове посљедице [43]. Иако је ризик најочигледнији разлог за увођење овог појма, ниво озбиљности осим што може значити ограничавање губитака или опасности, једнако је примјењив на достизање бенефита [45].

Нивои озбиљности се почетно процјењују у системским захтјевима, јер је ризик, односно озбиљност особина цијелог система [46]. Након анализе ризика и бенефита,

прво се дефинише скуп нивоа, а затим се сваком системском захтјеву додијели одређени ниво озбиљности. Они се затим, приликом декомпозиције захтјева, додјељују и програмским захтјевима [47]. На овај начин и функционални програмски захтјеви добијају свој ниво озбиљности.



Слика 3 - Одређивање нивоа озбиљности у процесу декомпозиције

Слика 3 илуструје активности дефинисања појединачних нивоа озбиљности приликом декомпозиције. При додјели нивоа озбиљности на нижим нивоима декомпозиције, може доћи до измјене нивоа озбиљности додијељених како функционалним, тако и системским захтјевима. Разлог томе је чињеница да захтјеви настали декомпозицијом *родитељског* захтјева, не могу имати виши ниво критичности од самог *родитељског* захтјева [3]. Процес је итеративан и завршава се када се није десила ни једна промјена нивоа озбиљности над свим захтјевима. Потребно је

напоменути да је могуће да се промијени и сам скуп могућих нивоа озбиљности, приликом декомпозиције, наравно уколико сам процес декомпозиције није стандардизован а скуп нивоа непромјењив.

Најбитнија два циља која се остварују помоћу нивоа озбиљности су:

- Комуникација између свих учесника у процесу, кроз све дијелове животног процеса развоја добављача [47]. На овај начин, сваки учесник у процесу постаје свјестан колико је значајан сваки програмски захтјев.
- Обезбјеђује се доношење коначне оцјене верификационог процеса у виду извјештаја верификације, који је обавезан документ верификационог процеса [48]. У циљу прецизног начина израчунавања резултата, препоручено је представљање нивоа озбиљности као тежинског фактора – бројем. Најстрожије доношење оцјене верификационог процеса би значило да је довољно да верификација једног програмског захтјева падне и да цијели резултат верификације буде негативан.

3.3 Сљедивост програмских захтјева

Још једна дефиниција сљедивости програмских захтјева гласи: „Сљедивост је способност праћења животног циклуса програмског захтјева у свим могућим смјеровима“ [49]. Као што је наведено, у питању је критична и најбитнија особина ригорозног процеса развоја софтвера [50]. Исто важи и приликом верификације функционалних захтјева безбједоносно-критичних система (*safety-critical systems – SCS*), који се раде по стандарду DO-178C [51]. Стандард захтјева документовање двосмјерних веза и изван домена животног циклуса програмских захтјева. Сљедивост обезбјеђује критичне аспекте подршке у процесу развоја SCS система, као што су безбједносна анализа, предвиђање ударца, итд…[52] . Због свега овога, обавезно је документовање веза програмских захтјева према изворном коду, као и према тестовима. Сљедивост програмских захтјева омогућује инжињерима да разумију екосистем софтвера, као и наизглед невидљиве везе између различитх модула у систему.

Ипак, много је разлога који објашњавају чињеницу да сљедивост у пракси није распрострањена и прихваћена. Неки од разлога су [53]:

- Различити језици се користе у различитим фазама животног циклуса развоја софтвера. Природни језик приликом спецификације захтјева, а програмски језик приликом развоја софтвера.
- Имплементација захтјева у софтвер и спецификација програмских захтјева описују проблем са различитих нивоа апстракције.
- Организациони разлози: организација, уколико није условљена стандардом као што је DO-178C не спроводи сљедивост, нити чак одржава евентуално наслијеђену сљедивост.
- Недостатак адекватних алата за креирање и одржавање сљедивости [54].

Имајући у виду особине које треба да посједује један VSL, управо је то алат који може да обезбиједи лакши прелаз између језика и нивоа апстракција. Овим би у великом мјери били елиминисани наведени разлози за избјегавањем развоја и одржавања сљедивости. У литератури постоје многе дефиниције и подјеле сљедивости, које се наводе у наставку, а затим ће бити именоване на начин на који ће се користити даље у раду.

3.2.1 Сљедивост унапријед и уназад

Сљедивост унапријед је било која врста везе ка логички сљедећем захтјеву. **Сљедивост уназад** је могућност праћења везе међу захтјевима у супротном смјеру [55]. Ово су основни типови сљедивости.

Најочигледнији примјери сљедивости унапријед и уназад се могу видјети код захтјева који су настали у процесу декомпозиције, односно који су произишли из истог захтјева вишег нивоа. Најчешће овакви захтјеви имају неки логичан редослијед, кретање по захтјевима *низ* логички редослијед је сљедивост унапријед, а кретање по овим захтјевима *уз* логички редослијед је сљедивост уназад. Могуће је чак и да постоји више путања сљедивости у оба правца.

3.2.2 Усправна и водоравна сљедивост

Водоравна сљедивост је сљедивост било које везе између програмских захтјева истог нивоа апстракције [56]. Сљедивости унапријед и уназад су водоравне сљедивости, али и не само оне. **Усправна сљедивост** је способност слијеђења и

захтјева који се налазе на различитим нивоима апстракције. Све релације *contains* са слике Слика 2 су примјер усправне сљедивости. Такође и релација *depends* између захтјева типа *Requirement* и *GroupRequirement* је усправна сљедивост.

3.2.3 Сљедивост изван контекста

Сљедивост изван контекста је најкомплекснија сљедивост, и представља могућност праћења сљедивости према артефактима животног циклуса развоја софтвера који нису унутар инжењеринга програмских захтјева. На примјер један тип сљедивости изван контекста је сљедивост која иде све до линије изворног кода (енг. *post-requirement traceability* [49]). Када су у питању *SCS* системи [51], ова врста сљедивости треба да сеже до тестова (јединичних, интеграционих... [57]). Гледајући на другу страну, сљедивост изван контекста је сљедивост према артефактима који су улаз за процес спецификирања захтјева (енг. *pre-requirement traceability* [49]). Сљедивост изван контекста није у фокусу овог истраживања.

3.2.4 Номенклатура сљедивости

Како би се у наставку рада могло јасније означавати одређена сљедивост, овдје ће бити коришћено другачије именовање, за све типове сљедивости. **Вертикална сљедивост** је могућност праћења по релацији *contains*, приказаној на слици Слика 2. која настаје када се апстрактни и општији захтјеви декомпонују у мање, конкретније захтјеве. **Хоризонтална сљедивост** је праћење по релацији *depends* између функционалних захтјева истог нивоа.

Сљедивост унапријед и уназад се имплементирају на начин да се прошири скуп особина програмског захтјева, са још једном особином која је јединствена (било која два различита програмска захтјева немају исту вриједност ове особине) и може да се упоређује [55], а затим се сортирањем добије релација сљедивости унапријед и уназад. Најчешће се за ово користи идентификатор програмског захтјева, али није риједак случај да се користи додатна јединствена (алфа)нумеричка особина, како би се сљедивост унапријед и уназад могла одржавати [54]. Због ове чињенице, да се релација остварује особином, ове двије сљедивости неће имати посебно име.

Све врсте сљедивости, које су у фокусу овог истраживања, се могу добити релацијама *contains* (релација садржавања, настала као посљедица декомпозиције и

сљедивости) и *depends* (релација зависности настала као посљедица конзистентности)**,** уз додатну јединствену особину програмског захтјева коју је могуће сортирати.

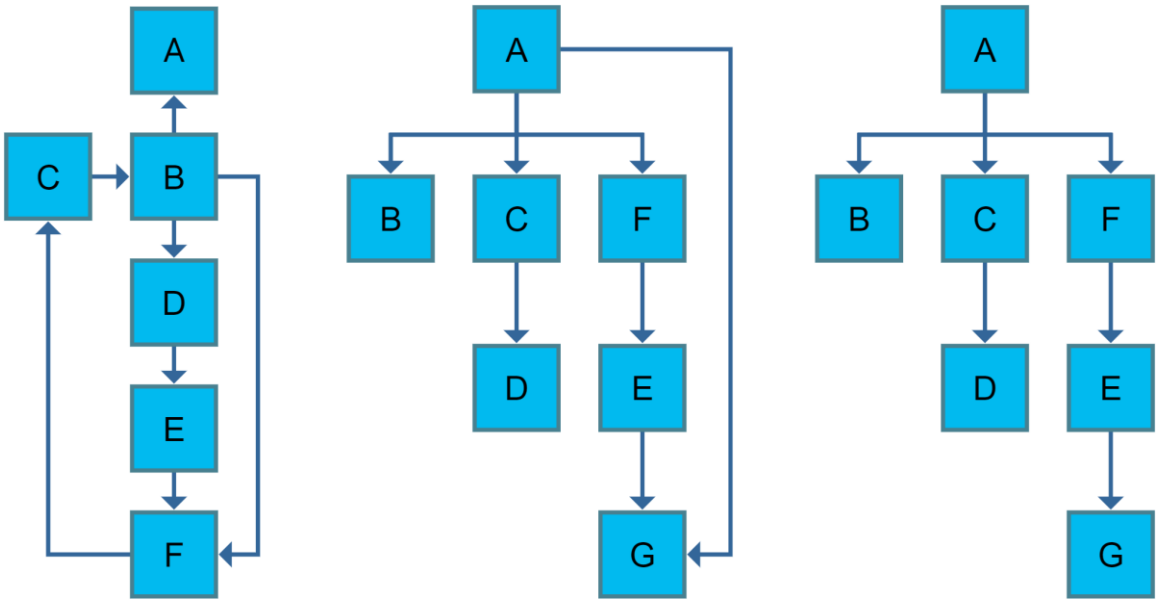
3.4 Усмјерени ациклични граф зависности

Постојање релације *depends*, повлачи за собом појаву усмјереног графа којег граде појединачни функционални програмски захтјеви и њихове међусобне релације зависности. У питању је, у ствари, *граф зависности* – усмјерени граф који представља међусобну зависност објеката, у овом случају програмских захтјева. За сваки граф зависности могуће је одредити присуство или одсуство могућности евалуације редослиједа. Најбољи примјер из области софтверског инжењеринга јесте увоз/импорт дефиниција (функције, промјењиве, структуре, класе, итд...) из других програмских модула, који такође могу увозити/импортовати друге дефиниције из других модула.

Дефиниција графа зависности:

За скуп објеката ***Z*** *и транзитивност* $\boldsymbol{R \subseteq Z \times Z}$ *гдје* $\boldsymbol{(a, b) \in R}$ *моделује зависности „a зависи од b“ (односно у случају верификације: „захтјев b мора бити верификован прије захтјева a“), граф зависности је граф* $\boldsymbol{G = (Z, T)}$ *са* $\boldsymbol{T \subseteq R}$ *гдје је* ***T*** *транзитивна редукција* ***R***.

Транзитивна редукција усмјереног ацикличног графа G је граф са минималним бројем грана али који има исту релацију као и оригинални граф. Како је релација зависности транзитивна, зависност се може остварити и транзитивним путем, тако да се неке гране могу укинути. Слика 4 приказује примјере цикличног (лијево) и нередукованог (у средини) и редукованог (десно) ацикличног графа. Средњи граф је нередукован јер грана AG може да се укине пошто је G зависно од A посредно преко објеката E и F. Транзитивно редуковани граф ацикличног графа је јединствен и минималан, а ово је битно како бисмо верификовали програмске захтјеве оптимално и увијек истим редослиједом.



Слика 4 - Примјери усмјереног цикличног и ацикличног графа

Циклични граф, приказан лијево на слици Слика 4 је цикличан на начин да између чворова B, F и C постоји зависност како непосредна, тако и посредна (преко чворова D и E). У графу зависности, појава цикличне зависности доводи до ситуације у којој није могућа евалуација редослиједа, односно у овом случају верификација захтјева, па је стога за успјешну верификацију функционалних програмских захтјева, од круцијалног значаја да релације зависности између захтјева творе усмјерени ациклични граф. У том случају редослиједевалуације је могућ помоћу тополошког сортирања. Тополошко сортирање је, када је у питању верификација, још поједностављено због сљедивости која инсистира на имплементацији јединственог својства програмског захтјева, који може да се сортира.

Управо из ових разлога, потребно је обезбиједити да систем верификације функционалних програмских језика има функционалност провјере цикличности графа зависности. Иако то није уобичајено, ипак због одређених специфичних детаља имплементације самог VSL, може се десити и да релација груписања такође може да формира циклични граф, тако да се ова функционалност треба звати **провјера цикличности**.

4. Доменски специфични језици

Процес креирања доменски-специфичног језика (*Domain-specific language - DSL*) подиже ниво апстракције изнад нивоа *регуларног* програмског језика. Ово се постиже уколико омогућимо специфицирање рјешења у терминима разумљивим домену проблема који се рјешава [58]. DSL користи познате концепте и правила из домена проблема као конструкте и елементе језика, а за то су задужени експерти датог домена од стране наручиоца. Експерти дефинишу области интереса, односно домен програмског језика. Ови домени могу да варирају, али постоји једна заједничка особина сваког од њих: сужавање фокуса. Сужени фокус омогућује да се ствари једноставније дефинишу, разумију и што је најбитније, изврше. Најчешће је фокус сужен на један скуп производа или пројеката унутар једне компаније. Углавном, доменски специфични језици су развијени *у кући* и нису нарочито изложени јавном приступу [32]. Провјерени рецепт треће генерације програмских језика који гласи: „понуди виши ниво апстракције и мапирај концепте вишег нивоа у ниже“ је подржан и у DSL филозофији.

Развој програмског језика није компликован процес. Двије главне ствари су омогућиле једноставнији развој DSL:

- Постојање алата за развој програмских језика
- Концепт метамодела – подиже ниво апстракције на ниво домена [32]

Доменски специфични програмски језици се примјењују у многим индустријским областима (потрошачка електроника, телекомуникације, жељезница, осигурање, авиоиндустрија, аутомобилска индустрија, компјутерско тестирање...). Креирани су углавном *унутар куће* без ванјске подршке [59], што указује да је знање експерата (у овом случају наручиоца) најбитније за креирање DSL.

Није мали број ни DSL језика који се користе за специфицирање или могу да се користе за специфицирање програмских захтјева [60], али такви језици ипак нису планирани да буду извршиви, него искључиво служе за међусобно разумијевање и смањење нејасноћа. Формални спецификациони језик, као и било који други програмски језик састављен је од три компоненте:

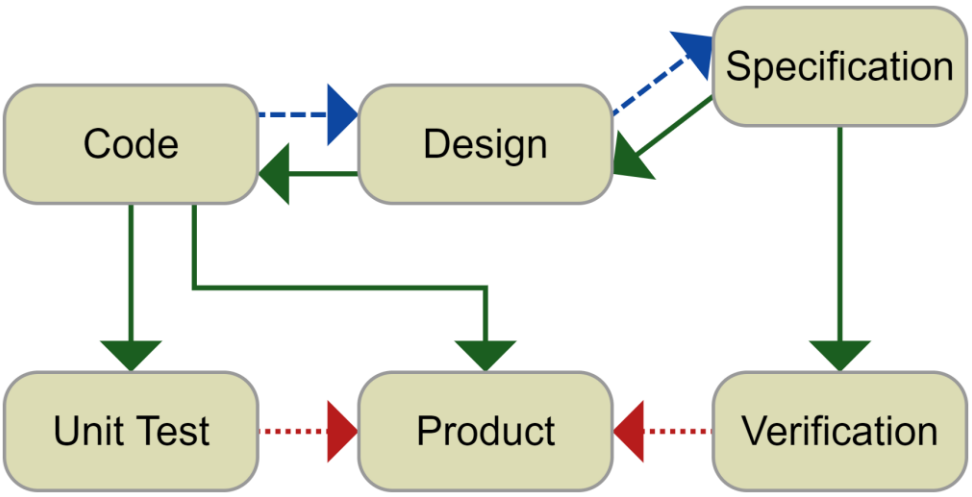
- Синтаксе која дефинише специфичну нотацију којом се специфицирају захтјеви.
- Семантике која помаже да се дефинише *универзум објеката* који ће бити кориштен за описивање.
- Скуп релација који дефинишу правила под којима неки објекат из *универзума* задовољава спецификацију [60].

Теорија која стоји иза развоја формалног спецификационог језика може бити или математичка (у литератури се још зове и алгебарска или орјентисана ка особинама) или базирана на моделу [61]. Основа математичке теорије је алгебра, логика или нека комбинација. Теорија заснована на моделу се састоји од језичких елемената као што су теорије скупова, релација и функција [61]. Када је у питању DSL који би се користио за спецификацију или верификацију за основу може да се користи било која од ове двије теорије.

Креирање DSL (заједно са програмима који га подржавају), умјесто коришћења неког већ постојећег програмског језика, се исплати уколико дати тип проблема или рјешења може бити јасније изражен новим језиком него постојећим. Такође, овај тип проблема би се требао довољно често појављивати. У наставку ће бити детаљније обрађена тема добити и трошкова имплементације DSL. Сваки DSL се специјализује за дати домен проблема, као и технику презентације проблема и рјешења, и остале аспекте домена.

4.1 DSL и животни циклус развоја софтвера

Без обзира какав је животни циклус развоја софтвера, неки процеси су обавезни. Слика 5 приказује поједностављен животни циклус развоја софтвера. Зелена линија представља примарни ток: креће се од спецификације захтјева (*Specification*), дизајна (*Design*), програмирања (*Code*), послије којег настаје производ (*Product*) као и *white-box* тестови (*Unit Test* - не нужно јединични). Када настане производ, тада се остварује могућност да се над њим (црвене линије) врши тестирање и верификација тестирањем. Тестирање као посљедица програмирања, а верификација као посљедица специфицирања захтјева.



Слика 5 - Поједностављени изглед животног циклуса у развоју софтвера[2]

Повратна спрега, на слици приказана плавом бојом од програмирања, преко дизајна до спецификације (у неким животним циклусима може да иде и од тестирања) преставља чињеницу да је могуће мијењати спецификацију програмског захтјева на основу дизајна или програмирања. Ако томе додамо да се спецификација захтјева може промијенити и по основама које су изван животног циклуса развоја софтвера (нпр. наручилац мијења захтјев), долазимо до проблема синхронизације процеса верификације, сваки пут када се промијени спецификација захтјева. У случају да нема директне и аутоматске везе између спецификација захтјева и верификације, неће бити ни користи од верификационог процеса [2], [20]. Проблем синхронизације може бити једноставно избјегнут уколико и спецификација програмског захтјева, а и верификација користе исти програмски језик [2]. Осим тога, DSL смањује могућност појаве концепата склоних грешкама (*error-prone*) у фази дизајна [32] и избјегава појаву нејасноћа између наручиоца и извођача.

4.2 Типови доменски специфичних језика

DSL језици по својој репрезентацији могу бити текстуални, као што је VHDL или XPath или графички као што је UML. Са становишта технике која је кориштена за имплементацију програмског језика, DSL се дијели на [62]:

- Интерне (*internal*) – који користе постојећи програмски језик високог нивоа за своју синтаксичку, а дјелимично и семантичку основу. Познаваоци програмског језика који интерни DSL користе као своју основу су у потпуности способни да користе овај програмски језик. Међутим, често овакви програмски језици буду *удаљени* од домена за који су развијани, а самим тим и од оних који би требало их користе.
- Екстерне (*external*) – који представљају у потпуности посебан програмски језик са сопственом синтаксом, семантиком и скупом релација. Често су прихватљивији експертима домена него интерни.
- Језички *workbench* – нови концепт који омогућава измјену основних постулата језика умјесто или заједно са изворним кодом. Нису дио разматрања у овом раду.

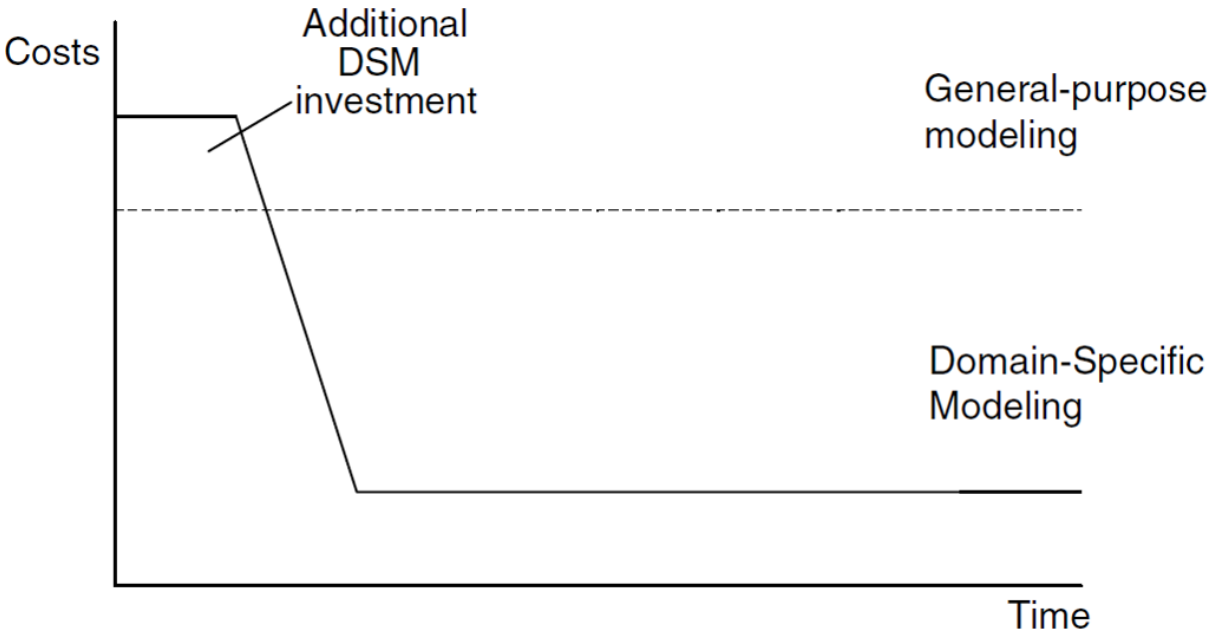
Трећи начин подјеле DSL потиче из раних година програмских језика. Подјела је према начину извршавања [31]:

- *Компајлер* (такође и *Генератор*) – који дати изворни код трансформишу у један од три облика:
  - у изворни код написан у неком другом вишем програмском језику (на примјер C, Java, Python…),
  - у изворни код писан у нижем програмском језику (на примјер Асемблер), или пак
  - у облику објектног или извршног програма. Што је крајњи случај и у прва два облика трансформације.
- *Интерпретер* (у литератури познат и као *Engine*) – који садржи *машину* која чита и интерпретира изворни код и на тај начин исти извршава.

Који год тип извршавања да се користи, DSL мора имати парсер и лексер који ће креирати апстрактно синтаксно стабло (AST) од изворног кода [34]. У случају интерпретера, ово стабло ће служити за извршавање, а у случају генератора, ово ће стабло служити прво за оптимизацију, а затим и за трансформацију у други програмски језик или извршни програм.

4.3 *Добити и трошкови имплементације доменски специфичних језика*

За DSL се каже да је добро дефинисан ако може описати понашање система и свих његових критичних дијелова на начин да експерт из датог домена може да разумије [62]. Међутим, ангажман доменских експерата у развоју програмског језика није јефтин. Ипак, у коначници, трошкови моделовања DSL ће бити нижи од трошкова моделовања општег програмског језика (*General Purpose Language – GPL*) вишег нивоа апстракције [35].

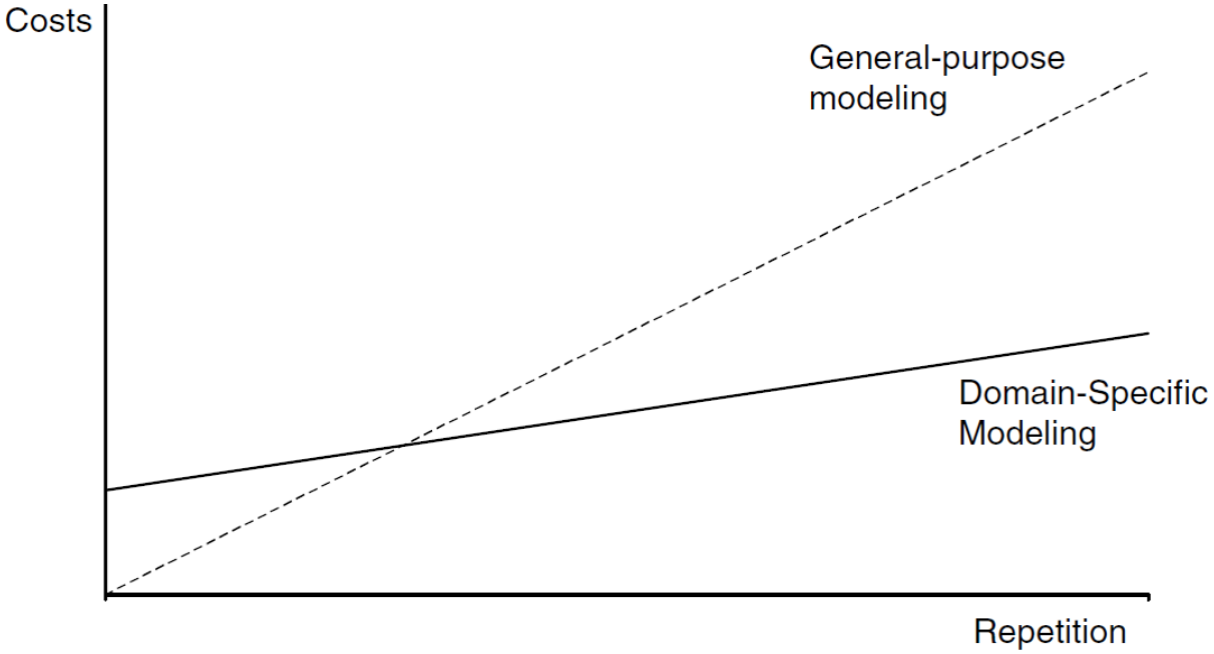


Слика 6 - Компаративни трошкови моделовања DSL и GPL [35]

Иницијални трошкови моделовања DSL су у потпуности састављени од трошкова људских ресурса: доменских експерата и искусних инжињера, али послије иницијалне фазе, трошкови развоја драстично опадају. Слика 6 приказује однос трошкова моделовања DSL и GPL. Разлози за ово смањење трошкова у моделовању DSL лежи у томе што постоје алати који помажу развој DSL, а такође послије иницијалног ангажмана експерата, они нису потребни у сљедећим фазама развоја. На крају, због ограниченог домена у односу на GPL смањен је и број елемената који морају бити дефинисани у DSL.

Са друге стране, уколико се искористи један DSL више пута, већа је вјероватноћа повратка уложеног. Према једном истраживању, уколико има више од три варијанте коришћења DSL уложено се исплати [63]. Слика 7 приказује однос трошкова

и поновног коришћења. Поновно коришћење укључује и ситуације у којима се користи друга верзија или друга конфигурација истог програмског језика.



Слика 7 - Компаративни приказ раста трошкова GPL и DSL приликом поновног коришћења [35]

Додајмо томе да су и трошкови одржавања и надоградње такође нижи, јер по обичају укључују само инжињере, али не и доменске експерте [35]. Трошкови чекања – губитак добити због чињенице да DSL још није доступан – су уобичајено већи од оригиналне цијене коштања DSL.

Осим смањених трошкова, моделирање DSL *у кући* доноси погодности када су у питању учесници на пројекту:

- што је више корисника DSL-а већа је и добробит, а повећава се и база знања [35],
- мање искусни инжињери су у могућности да развију комплексније апликације користећи DSL [64],
- искуснији инжињери, истраживања показују, перципирају DSL као ефикаснији алат [65],
- комуникација између свих учесника на пројекту је побољшана [34], [61], укључујући и учеснике из финансијског и пословног дијела пројекта [34].

4.4 Дефинисање доменски специфичног језика

Задатак дефинисања језика је утолико једноставнији уколико је проблем специфичан и јединствен. Неријетко се дешава да већ постоје неки алати, скрипте и неформални програми који тренутно помажу процесу за који се дефинише DSL. Уобичајено је и да постоји рјечник специфичан за дати домен који се колоквијално користи у контексту домена. Оно што се појављује као најбоља пракса при дефинисању DSL јесте креирање метамодела [32, pp. 227–229], [35], [39], [66]–[68], а постојећи артефакти домена су идеални за изградњу метамодела. Метамодел је највиши ниво апстракције. Ипак, колико се год тежи апстракцији, треба имати на уму да се и даље има могућност интерпретирања или компајлирања датог кода.

Основу за дефинисање метамодела се може наћи у:

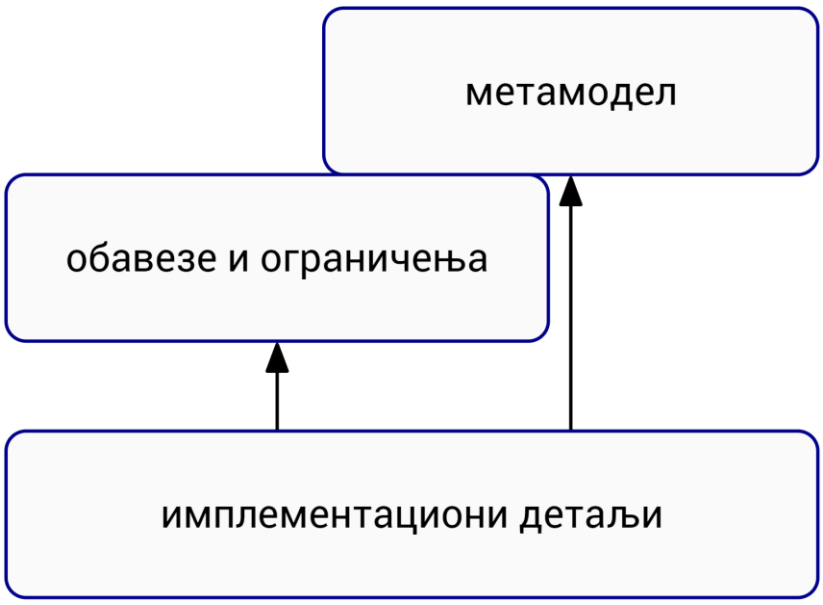
- архитектури или хијерархији ако постоји у домену. Уобичајено је да архитектура садржи и појмове и апстракције, погодне за креирање метамодела,
- Постојећим производима и апликацијама, са пратећим упутствима често садрже опис структуре понашања и семантике,
- Обрасцима који се налазе у домену ,
- Окружењу домена и њихова сучеља са *ваљским* свијетом,
- Програмском коду – уколико се користи развој одоздо – према горе, могуће је искористити постојећи програмски код за дефинисање метамодела.

Међутим, када су у питању програмски захтјеви најлогичнији извор за креирање метамодела јесу саме спецификације заједно са рјечницима-појмовницима који се обавезно креирају у процесу специфицирања како би обје заинтересоване стране знале тачно шта који појам представља. Многи верификациони алати користе ове појмовнике за конструкцију метамодела који ће се користити при верификацији [39], [40], [69], [70].

Препорука је да се послије креирања метамодела, истом касније додаје концепт кодирања. Концепт кодирања мора да садржи функционалности и ограничења која нису примјењива метамоделом [35] и у већини случајева нису у директној вези са

актуелним проблемом из датог домена. Ово је најбитнији разлог зашто се, концепт уводи у развој језика послије креирања метамодела. У области верификације програмских захтјева, управо функционалности и особине, које су у претходном поглављу уочене као круцијалне су оно што је што је потребно имплементирати као концепт кодирања. Могуће је да ће овај корак изазвати одређене контрадикције са метамоделом или да неће моћи да одговори на све захтјеве/структуре података које постоје у метамоделу.

Трећи корак у дефинисању DSL је додавање имплементационих детаља. Имплементациони детаљи ће приказати метамодел домена кроз правила и ограничења концепта кодирања, самом програмеру DSL. Детаљи имплементације служе да *помире* претходна два концепта на начин да евентуалне контрадикције ријеши, а недостатке надомјести. Управо ће детаљи имплементације рјешавати најситније детаље изведбе саме верификације.



Слика 8 - увезивање корака развоја DSL помоћу детаља имплементације

Слика 8 описује дату појаву: Колико год да се трудили, метамодел неће бити у потпуности усклађен са имплементираним ограничењима и обавезним имплементацијама, због тога су ту детаљи имплементације како би премостили ове неусклађености.

4.5 Функционалности алата за верификацију захтјева

Идеја развоја шаблона програмских захтјева, који могу бити поново искориштени у виду природног програмског језика није нова, то је најлакши начин да се имплементирају правила и ограничења. У поглављу о инжињерству програмских захтјева је изведен закључак о значају формалног програмског или природног језика у специфицирању захтјева. Велику већину функционалних програмских захтјева могуће је дефинисати помоћу језика, био то формални програмски језик, маркап језик или природни језик.

Такође, чињеница је да се исти језик може у одређеном капацитету користити и у сврху верификације [37], [58], [71], као и да се језици за спецификацију обогаћени додатним алатима [68] могу користити за верификовање, тих истих програмских захтјева. Са друге стране, постоји јасна веза између обавезних функционалности у процесу верификације и препоручених карактеристика програмских захтјева, приказана у Табели 1. На примјер, уколико процес верификације подржава јединственост програмског захтјева, међу осталим захтјевима, омогућује се сљедивост, тачност и могућност измјене. Уколико обезбиједимо да процес верификације може да подржи све набројане функционалности верификације, онда можемо са сигурношћу тврдити да ће тај процес обезбиједити све препоручене карактеристике које мора да садржи функционални програмски захтјев. Ово значи да, уколико у другом кораку креирања DSL (концепт кодирања) успијемо да имплементирамо дате функционалности и особине, бићемо у могућности да произведемо језик за верификацију функционалних програмских захтјева. И док се метамодел креира на основу рјечника спецификације програмских захтјева, концепт кодирања се развија имплементацијом функционалности и особина из ове табеле.

Табела 1 - Веза између функционалности процеса верификације и препоручених карактеристика програмског захтјева

Функционалности и особине верификације	**Препоручене карактеристике програмског захтјева**
Јединственост	Сљедивост, тачност, могућност измјене
Релација зависности	Конзистентност, могућност измјене, хоризонтална сљедивост
Провјера цикличности	Конзистентност и тачност
Релација груписања	Вертикална сљедивост
Предусловљавање	Конзистентност
Имплементација битности	Нивои битности
Могућност имплементације задатака прије и послије верификације	Комплетност, сљедивост
Синтаксна провјера	Комплетност, конзистентност, коректност (дјелимично)
Добро структуиран језик	Недвосмисленост, конзистентност
Могућност проширења	Могућност измјене, конзистентност

Са становишта DSL-а неке од ових особина морају бити задовољене, самим тим што имплементирамо програмски језик: могућност проширења, синтаксна провјера, структуираност језика. Са друге стране, иако многе функционалности верификације служе у подржавању сљедивости, оне је не покривају у потпуности. Такође, евидентно је из детаљније анализе сљедивости (*3.3 Сљедивост програмских захтјева*), да је сљедивост изнимно комплексна и излази изван оквира програмског захтјева. Због тога потребно је и **сљедивост уврстити међу особине верификације** које VSL мора да задовољи.

На крају, ниједан програмски језик, па ни DSL не може користи уопштене, неограничене и двосмислене изразе, као што су: суперлативи, субјективне синтагме, двосмислени придјеви, те други примјери из препорука релевантних тијела. Овоме треба додати и чињеницу да уколико користимо исти језик за спецификацију и верификацију, обезбјеђујемо аутоматску синхронизацију приликом измјене спецификације захтјева.

Када се гледа са аспекта имплементације ових особина и функционалности у одређени програмски језик, чини се јако компликованим. Међутим, управо је то посао професионалног инжињера захтјева – експерта домена: Развијање и специфицирање комплетних, недвосмислених, провјерљивих и конзистентних програмских захтјева на основу неформалних, двосмислених, некомплетних сценарија, информација и корисничких случајева [72].

5 Метрика коришћена у VSL са становишта спецификације и верификације функционалних пројектних захтјева

Наведена Табела 1 садржи функционалности и особине које је потребно да садржи програмски језик како би био способан да у потпуности подржи верификацију функционалних програмских захтјева. Уз констатацију да задње три особине, комплетан програмски језик већ подржава као и чињеницу да сљедивост, због своје комплексности, није у потпуности *покривена* и мора бити дио скупа обавезних особина VSL. Да би изградили DSL способан да на исправан начин специфицира и верификује функционалне захтјеве, он мора да:

- посједује сљедеће особине:
    - **Јединственост** – способност јединственог индексирања захтјева, како би се захтјев могао: пратити, измијенити и провјерити покривеност верификације захтјева. Јединственост се користи и за поређење захтјева како не би било *преклапања* између захтјева, као и за сљедивост. На крају, тачност која се огледа и у томе да се исти програм извршава на исти начин уколико су сви услови који утичу на њега исти, је подржана са овом особином.
    - **Битност** – подесити ниво битности сваког захтјева, од значаја за сигурносно критичне системе, јер омогућује мјерење тежине резултата верификације. Различити су видови битности неког програмског захтјева, они могу да се огледају у сигурносном или извршном аспекту [43]. Обезбјеђује мјерљивост програмског захтјева.
    - **Сљедивост** – могућност праћења животног циклуса програмског захтјева (од његове појаве, декомпозиције, остваривања у програмском коду, тестирања кода и потврде исправности) у свим могућим смјеровима је обавезна особина за сигурносно критичне производе [51]. Сљедивост обезбјеђује тачност програмског захтјева.

- **Релацију зависности –** свјесност програмског захтјева о томе на који начин зависи од осталих програмских захтјева. Зависност између програмских захтјева омогућује конзистентност програмског захтјева [20]. Релација зависности је битна и за особину сљедивости, јер омогућује хоризонталну компоненту ове особине.

- Понашања, односно функције:
  - **Груписање захтјева** – особина која је настала као посљедица декомпозиције захтјева. У питању је релација између више програмских захтјева различитог нивоа, као и релација програмским захтјевима истог нивоа који припадају истој групи. Груписање програмских захтјева омогућује вертикалну сљедивост.
  - **Провјера цикличности –** првенствено, као посљедица релације зависности али и груписања, језик (интерпретер или компајлер) мора примијетити присуство цикличне зависности [30] између програмских захтјева. Некад је сама конструкција програмског језика таква да онемогућује цикличност, на примјер: XML таг унутар којег се налази други XML таг је идеалан начин за представљање релације груписања. Исто важи и за графички приказ једног објекта унутар другог, што је чест случај код графичких DSL. Ипак, за представљање релације зависности, могу се десити цикличне појаве.
  - **Предусловљавање –** језик мора имати могућност дефинисања зависности неког програмског захтјева од нечега изван контекста верификације. Задаци прије и послије саме верификације који се препоручују при верификацији програмских захтјева, могу се поједноставити, јер су настали као посљедица потребе постављања система који се верификује у стање потребно да би могло доћи до верификације. Минимална функционалност која ипак мора постојати у овом случају је установљавање да ли

постоје услови да се програмски захтјев верификује или не. То је управо ова фунционалност.

Управо испуњеност овог скупа особина и функција се сматра метриком која ће бити коришћена у анализи сродних радова и истраживања из области верификације функционалних програмских захтјева. Такође, као додатна оцјена, користиће се и **комплексност** самог DSL из разлога што је могуће увођењем додатних алата највјероватније покрити све особине. Ипак оваква рјешења отежавају кориштење, нарочито када су у питању наручиоци и њихов домен интереса и знања.

6 Теза истраживања

Постојећи DSL који могу да се користе за спецификацију и верификацију функционалних програмских захтјева не испуњавају све обавезне особине и понашања која су наведена у метрици, иако је могућ DSL који испуњава све обавезне особине и понашања.

7 Сродни радови и истраживања из области верификације функционалних програмских захтјева

Истраживање је показало да се, када је у питању област верификације функционалних програмских захтјева помоћу програмских језика, алати могу подијелити у три групе:

- Језици за спецификацију захтјева проширени са додатним алатима који омогућују верификацију
- Контролисани природни језици шире примјене, који су кориштени за верификацију програмских захтјева
- Програмски језици за тестирање

Осим тога, како је наведено, верификација се може извршити на два начина: тестирањем и анализирањем. На крају, сваки од ових алата-језика има одређени ниво комплексности. При истраживању сродних радова, биће вођено рачуна о томе да се упореди колико су поједини радови у складу са особинама и функционалностима изолованим у претходном поглављу и табели Табела 1, затим и о томе да ли алат ради верификацију тестирањем или анализирањем, као и који је ниво комплексности коришћења датог алата.

7.1 User Requirement Notation

User Requirement Notation (URN) је визуални језик за моделовање који подржава анализу, спецификацију и валидацију захтјева [40]. Дефинисан је од стране тима *International Telecommunication Union* и представља први међународни стандард који експлицитно моделује у графичком окружењу сценарије, циљеве и везе између њих, у циљу моделовања система за који се специфицирају захтјеви. Настао је у жељи да се избори са нарастајућом комплексношћу захтијева унутар области телекомуникација. Намјера је била да се развије графички програмски језик за спецификације унутар националних, међународних стандардизационих тијела. Користи се и унутар комерцијалних организација за спецификацију захтјева изван стандарда.

URN је дефинисан како би имао способности да:

1. Дефинише захтјев корисника и у случајевима када има мали број детаља,
2. Опише сценарије на апстрактан начин, без референци према другим елементима система,
3. Омогући узимање у разматрање алтернативних рјешења,
4. Има могућност динамичког побољшавања,
5. Примјењив на дизајн протокола заснованих на преговарању,
6. Има могућност детекције и избјегавања нежељених интеракција,
7. Обезбиједи резоновање будуће стратегије у вези са самим захтјевом,
8. Обезбиједи начин да се дефинише и анализира нефункционални захтјев,
9. Обезбиједи начине да се изразе везе између захтјева и циљева,
10. Обезбиједи поновно коришћење
11. Обезбиједи сљедивост, укључујући и сљедивост изван контекста

Језик је углавном кориштен у телекомуникацијама и подијељен је у два под-језика:

- *Goal-oriented Requirement Language* (GRL) – који се користи за описивање нефункционалних захтјева. Као што име каже, служи за дефинисање циљева система и самог пословања, као и начина да се циљеви постигну, укључујући и алтернативе и образложења. GRL обезбјеђује горе наведене способности од 7 до 11.
- *Use Case Maps* (UCM) – који се користи за описивање сценарија и понашања алата/производа који је потребно изградити. Описивањем понашања, овај језик у ствари описује функционалне захтјеве. UCM апстракује интеракције међу компонентама, како би се оне дефинисале касније приликом декомпозиције. Овај под-језик обезбјеђује горе наведене способности од 1 до 7. Осим за спецификацију, UCM се користи и за валидацију захтјева, идејно рјешење архитектуре, као и генерисање тестова. Управо генерисање тестова је оно што је почетни корак за верификацију функционалних програмских захтјева.

Даља истраживања су показала да је UCM са својим дефинисањем сценарија и понашања способан и за верификацију функционалних захтјева:

- Проширењем основног UCM модела [73] додатним алатима, могуће је *полуаутоматски* генерисати верификационе тестове.
- Трансформацијом UCM модела у формалну спецификацију, чиме се, уз помоћ LOTOS-а аутоматски генеришу тестови за верификацију [74].
- Примјењивањем *тестних шаблона* [75] над UCM сценаријима могуће је извршти верификацију. Међутим, сами тестови се морају генерисати ручно, што чини ову технику најмање прихватљивом, из разлога синхронизације спецификације и верификације.

Сам URN имплементира **јединственост** и сваки елемент је јединствено идентификован унутар спецификације [40], [73], [76], а такође подржава имплементацију **нивоа озбиљности**, који се узима у обзир приликом евалуације укупног резултата [40], [73], [76]. Већ споменуте везе између сценарија и циљева су настале као резултат релација **сљедивости**, **зависности** или **декомпозиције** [38], [73], [76], међутим постоје озбиљни проблеми са одржавањем ових веза [38], [73]. Када је у питању сљедивост, постоје приједлози увођења додатних алата који омогућавају рјешавање датог проблема [77].

Пошто је URN дијелом језик моделирања, дозвољава постојање **цикличности**, а анализе синтаксе [78], [79] потврдиле су озбиљне проблеме који настају због постојања цикличности, а који онемогућује комплетну верификацију. **Предусловљавање** као функционалност не постоји, јер би то водило ка моделовању нечега што није дио система, али постоје истраживања која препоручују додатне алате који омогућују имплементацију предуслова [77], [80], [81].

7.2 Спецификација са *RSLingo* - верификација са *TSL*

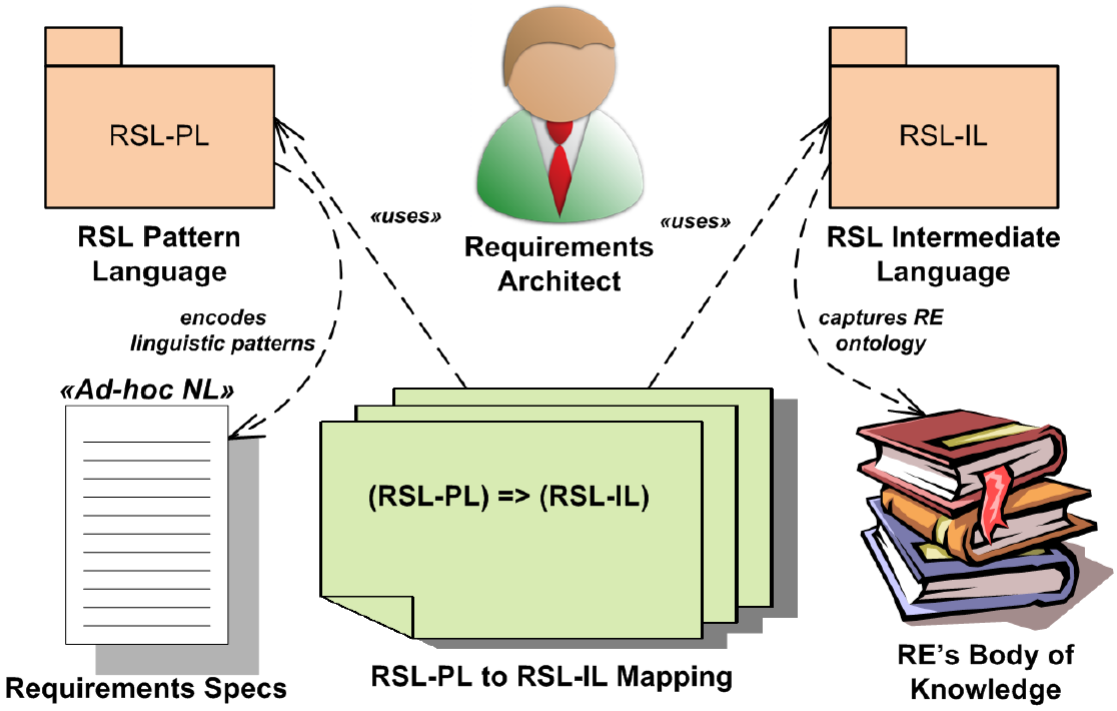
RSLingo је програмски језик који се користи за специфицирање захтјева [39], али је пројектован тако да омогућује проширења за верификацију, а чак је у могућности и да самостално верификује неке од захтјева, у већини случајева захтјеве из области квалитета – нефункционалне захтјеве. Представља резултат дугогодишњег истраживања [39], [82]. Основа, односно улаз у *RSLingo* су програмски захтјеви

написани у природном језику. Иако је природни контролисани језик, најуобичајенији и најприхватљивији облик специфицирања захтјева, склон је двосмисленостима и неконзистентности, а такође их је тешко аутоматски валидирати, верификовати или превести у друге форме (на примјер дијаграме, архитектуру или тестне случајеве). RSLingo је временом проширен на начин да посједује свој контролисани природни језик којим се креирају програмски захтјеви. Контролисани језик је такав да системски обезбјеђује ригорознију и конзистентнију спецификацију програмских захтјева. Овај програмски језик је назван RSL, и у дијелу верификације се не разликује од RSLingo па ће у истраживању бити третиран на исти начин. Његова је предност та што због сопственог природног језика има бољу екстракцију доменског знања [82].

Језик издваја појединачна знања домена и рјечник из програмских захтјева. Фокусиран је на бољу спецификацију захтјева и квалитетније разумијевање свих заинтересованих страна. *RSLingo* је практично састављен од два програмска језика која се међусобно мапирају [39]:

- *pattern language* (RSL-PL) – језик који служи за извлачење информација из захтјева написаног на природном језику, кодирањем језичких образаца. Овај дио језика користи технике процесирања природних језика како би се створили стабилни и недвосмислени појмови које ће користи други дио језика. Овај дио опонаша људски процес читања и разумијевања текста захтијева и снажно се ослања на синтаксичко-семантичко усклађивање са језичким обрасцима.
- *Intermediate language* (RSL-IL) – формални DSL језик са коначним скупом конструкција који представљају особине и функције из инжењеринга захтјева и представља онтологију програмских захтјева.

Заједно, ова два језика омогућују екстракцију доменског знања написаног у природном језику, парсирање и конвертовање у структуиранији формат. Цијели процес се надзире и користи од стране инжињера којег још зову и архитекта захтјева, како би се максимизовала екстракција са једне, али и исправно структуирање са друге стране (Слика 9 - Фаза дефинисања у RSLingo [39]).



Слика 9 - Фаза дефинисања у RSLingo [39]

Управо ова рашчлањеност језика омогућује верификацију програмских захтјева. Најпознатији језик који се користи као додатни алат за верификацију уз *RSLingo* је *Test Specification Language* – TSL [68]. TSL је снажно инспирисан граматиком, номенклатуром и стилом писања дефинисаним у *RSLingo* [82], користећи тестирање засновано на моделу, чиме је у могућности да тестира различите концепте који описују модел: и концепте из природног језика, али и из језика прилагођеног компјутерској обради. У могућности је да користи 4 стратегије за тестирање: анализа домена, use-case тестови, тестови прихватања (*acceptance testing*) и тестирање машине стања које се користи и за машину стања специфицирану са *RSLingo*.

Када су у питању особине и функционалности, **цикличности** су дозвољене у *RSLingo* и могу довести до непотпуне верификације система [83]. Захтјеви су **јединствено идентификирани**, користећи хијерархијску нотацију, слично као код нумерисања поглавља [39], [84]. Особина **сљедивости** се имплементира само између програмских захтјева и ванског свијета (сљедивост изван контекста), али не и унутар *универзума* програмских захтјева [39], [85], са друге стране ни TSL не употпуњава особину сљедивости [68], [82], тако да се може рећи да ова особина дјелимично задовољена. **Предусловљавање** постоји унутар TSL, док **зависност** од других захтјева

не постоји [82] осим као **групна зависност**, односно груписање које је посљедица декомпозиције у RSLingo [85]. Када је у питању **ниво озбиљности**, иницијално он није имплементиран, а како је то карактеристика која је изнимно битна за верификацију система базираних на сигурности и безбједности, постоји посебна надоградња (RSL4CYBERSECURITY) која рјешава овај проблем [83]. Сама чињеница да се користи доста концепата чини ово рјешење **комплексним** за коришћење. Ипак, примјећено је на примјер, да корисници лакше прихватају и брже уче RSLingo него URN, због перцепције да је UML компликованији [86].

7.3 Верификација захтјева комбинацијом језика Promela и SPIN

Promela је DSL за спецификацију програмских захтјева заснован на семи-формалној онтологији [87]. Језик се састоји од два нивоа. Први ниво дефинише процесе са учесницима и токовима послова и података. Сви појмови који се користе у описивању ових токова и процеса припадају контролисаној онтологији која је дизајнирана специјално за дати домен. За презентацију, овај ниво језика користи графичку презентацију дијаграма процеса тока (*Swimlane model*). Други ниво је моделирање акција унутар датих процеса и токова како би се у потпуности описао захтјев.

Promela описује системе који се називају и *P*-системи – модели са коначним бројем стања [88]. Идеја иза P-система јесте да се рачунања извршавају имитирајући биолошке процесе, на начин на који и ћелије међусобно комуницирају хемијским реакцијама кроз мембране ћелија. Управо зато се и користи дијаграм процеса тока који посједује визуелну представу процеса одвојених мембранама. Један од назива P-система је и *мембранско рачунарство.*

Са аспекта синтаксе, језик је формалан програмски језик и сличан је Це језику. Могуће је дефинисати да процеси раде конкурентно. Дефинисани процеси могу комуницирати преко дијељених глобалних промјењивих или преко канала комуникације. Унутар процеса постоје конструкције за недетерминистички избор, покретање других процеса, атомско извршавање и друго [89].

На основу овако дефинисаног програмског захтјева, генерише се *Promela* изворни код који ће бити извршен помоћу SPIN (*Simple Promela Interpreter*) како би се верификовали дати програмски захтјеви [69]. SPIN је верификатор модела (енг. *model*

checker) – аутоматизована техника за верификовање да ли дати модел одговара датим спецификацијама [10]. Један од бенефита коришћења SPIN верификатора модела је његова способност да се носи са комплексним моделима, великим доменима као и великом бројем промјењивих.

Сваки процес је јединствено идентификован, што значи да подржава **јединственост** [69], [87], [90]. Ова јединственост је искоришћена за имплементирање **зависности** између захтјева [87], [90], такође је имплементирана и зависност према догађајима изван система помоћу кључне ријечи *initP* и функције *initial* [69], [87], [90], чиме је остварена могућност постављања **предуслова**. Са становишта перформанси, SPIN се показао као најбољи у конкуренцији осталих *model checker* алата, а оцјена комплексности коришћења овог система од стране почетника је оцијењена као средња [91].

Комбинација ова два алата је изгледа прихватљива за корисника, јер је у питању графички алат, а и онтологија која је изграђена у процесима је одлична основа за комуникацију са извршиоцем, али и са машином. Међутим, **мјера нивоа озбиљности** постоји само дјелимично имплементирана и не може се користити код рачунања укупне оцјене верификације [69], [87], [90]. Истраживања такође показују да иако је P-систем такав да има коначан број стања, ипак могуће је грешком моделовати бесконачне процесе [90], [92], тако да провјера **цикличности** није имплементирана. Посљедица чињенице да је овај систем примјењив само на P-системе је и та да моделирање груписања у P-систему није могуће [88], па самим тим, није могуће ни имплементирање **групне зависности**. Приликом овог истраживања нису нађене потврде о постојању **сљедивости** ни на нивоу моделовања нити на нивоу верификације модела.

7.4 Моделовање и верификовање захтјева помоћу RM-RNL и AADL

Процес верификације у овом случају се одвија помоћу два програмска језика:

- **RM-RNL** (*Requirement Modeling with Restricted Natural Language*) је језик базиран на ограниченом природном језику који је способан да елимише двосмисленост из природног језика [93].

- Језик за анализу архитектуре и дизајна **AADL** (*Architecture Analysis and Design Language*) [94] је уједно и текстуални и графички DSL, а углавном се користи за спецификацију у безбједносно-критичном софтверу (SCS). Стандардизован је 2004 године од стране *Society of Automotive Engineers* (SAE) и де факто је стандард у авио и аутомобилским софтверским системима. Концептуално, AADL је језик за моделовање, чији се излаз користи и за спецификацију, анализу, и генерисање кода [94]. AADL омогућава да подаци и програмске компоненте буду организоване у пакете који представљају апстрактни изворни код, имплементиран у програмским језицима као што су Јава, Це или Ада, али и у графичким језицима. Могуће је конкурентно програмирање, као и интеракција преко портова, дијељење меморије и објеката.

Из перспективе анализе, односно верификације захтјева, AADL даје генерички улаз за различите алате за верификацију, као што су TASM (*Timed Abstract State Machine*) [95] , UPPAAL [96] и AGREE [97].

Кораци по којима ови алати сарађују:

1. RM-RNL креира једнозначну онтологију из природног језика којим су представљени захтјеви.
2. RM-RNL на основу онтологије, креира иницијалне моделе за AADL. Овдје су већ створене иницијалне везе које ће се користити у сврху остваривања сљедивости.
3. Дорађивање иницијалних модела креирањем спецификације захтјева, која представља улаз за алате за анализу и генерисање кода.
4. Анализирање – верификација неким од наведених алата.
5. Генерисање кода – кода који је дио будућег производа и није дио домена овог истраживања.

Битно је напоменути да током овог процеса, сви се аспекти **сљедивости** ажурирају. **Јединственост** захтјева је такође подржана коришћењем овог скупа алата [93]–[95], [98]. Исто важи и за везу **зависности**, а осим ове релације, модел има и хијерархијску структуру, како би приказао декомпозицију односно релацију

груписања [93], [94]. **Предусловљавање** укључује опште услове, као и временски зависне услове [93], [95], [98]. Најновија истраживања су примјетила проблем **цикличне зависности** и предложена су проширења овог рјешења помоћу вјештачке интелигенције [98]. Такође, није пронађена потврда да постоји могућност дефинисања **нивоа озбиљности**, нити постоји потврда да постојећа три система верификације користе ниво озбиљности, ни на који начин [95]–[97]. **Комплексност** датог система алата је изнимно висока [98] јер се практично користе три врсте алата.

7.5 Контролисани природни језици

Контролисани природни језици (CNL – *Controlled Natural Language*) су природни језици са ограниченом граматиком, појмовницима и рјечницима. Ограничења над језиком се примјењују како би се смањила или елиминисала и двосмисленост и сложеност датог језика. До сада обрађени језици су такође наметали проблем двосмислености, па је тако RSLingo прерастао у RSL, а AADL користи у ту сврху RM-RNL.

Традиционално, CNL се дијели у двије главне категорије: оне које побољшавају читљивост за људе, посебно за оне којима основа CNL није матерњи језик (формални језик); и оне које побољшавају рачунарску обраду текста (природни језик) [37]. Највећа мана CNL-а је то што их је лако читати, а тешко је писати [2], [36], [99].

Наравно, CNL не имплементира препоруке IEEE везане за верификациони процес, као ни оне везане за процес спецификације програмских функционалних захтјева. CNL иако контролисани, често постају узрок многих проблема са квалитетом програмског захтјева, као што су нетачност, неконзистентност, некомплетност и двосмисленост [27], [100]. Ипак, бројни су CNL које се користе у процесу верификације програмских захтјева [58]. У почетку, CNL је коришћен искључиво за јасну спецификацију захтјева, уз безброј потешкоћа, корисници CNL су, да би специфицирали захтјев, морали да се искључиво ослањају на текстуалну поруку грешке [36], [37] коју би добијали приликом валидирања креиране реченице. Корисници су требали да запамте сва ограничења CNL и да тек онда пишу реченицу. Када процес парсирања реченице не успије, CNL систем покушава да идентификује узрок грешке и обезбиједи разумно објашњење и сугестије како поправити грешку. Проблем са овим приступом је тај што је порука грешке, да би била корисна морала

садржати потпуно знање писања дате реченице [101]. Овај проблем је превазиђен коришћењем шаблона реченица који већ препоручују многи ауторитети из инжењеринга програмских захтјева [16], [20]. На тај начин, CNL су уз помоћ додатних алата прерасли у алате за спецификацију и верификацију.

Битно је нагласити, да је лоша страна употребе природног језика у сврху верификације програмских захтјева та да описи могу да садрже двосмислености. Ово се рјешава коришћењем хеуристике и додатних, специфичних правила. Ова правила могу да покрију ограничења провјере процеса специфицираног у програмском захтјеву.

7.5.1 Контролисани енглески језик – Attempto

Контролисани енглески језик Attempto - ACE (*Attempto Controlled English*) представља подскуп енглеског језика који може бити једнозначно преведен у логику првог реда [102] чиме је обезбијеђена прикладна замјена *ad hoc* захтјева са потпуно логичким изразима. Ово чини да иста представа програмског захтјева буде разумљива и машини, као и не-техничкој особи. Парсер (*Attempto Parsing Engine* - APE) уједно недвосмислено генерише два ентитета: синтаксно стабло - као синтаксну представу реченице, и структуру излагања (*discourse representation structure – DRS*) као семантичку представу реченице. Свака реченица се парсира у контексту претходних реченица [70]. DRS се може превести на друге логичке језике, чија су синтакса и семантика адекватно дефинисане, на примјер у стандардне, клаузалне форме логике првог реда [103].

Најпознатије логике првог реда које могу да се користе у овом случају су: TPTP (*Thousands of Problems for Theorem Provers*) [104], *OWL 2 Web Ontology Language* [105] и SWRL (*Semantic Web Rule Language*) [106]. Након што је програмски захтјев преведен у изворни код, а затим анализиран методом доказивања или провјере досљедности (зависно од тога која се логика користи), захтјев се може сматрати верификованим. Међутим, чињеница је да оваква логика може бити неодлучена, односно бесконачна [107]. Ово је превазиђено коришћењем алата за резоновање RACE, који користи аксиоме и теореме из ACE како би дошао до закључка. Закључак, односно резултат верификације је веома строг, због недостатка тежинског фактора и због самог резоновања [108].

Комплексност овог рјешења се огледа у томе што корисник мора разумјети цијелу онтологију како би могао да је користи [37]. **Јединствени идентификатори** програмских захтјева не постоје [42], [103], али постоји развијен додатни алат који их омогућује [42]. **Ниво озбиљности** није предвиђен као посебна особина али се може додавати логици ручно, прије самог резоновања [36], [103]. **Сљедивост** није особина ACE, као ни алата резоновања [109]. **Груписање** и **зависност** програмских захтјева није омогућено у ACE [110] али постоји додатни алат (*OWL Verbalizer*) који омогућује стварање зависности између реченица, односно програмских захтјева, као и њихово груписање [36]. Иако RACE обезбјеђује превенцију бесконачне петље при резоновању [103], сама провјера **цикличности** није имплементирана, ипак истраживања потврђују да је могуће имплементирати провјеру цикличности [111]. **Предуслови** су реализовани у самој спецификацији захтјева и као такви пренијети у логику [42].

7.5.2 Компјутерски обрадив језик – CPL

CPL (*Computer-Processable Language*) је природни контролисани језик који је способан да ријеши двосмислености у реченици и прецизно произведе тачну имплементацију [37]. Највише се користи у областима извлачења знања и вјештачке интелигенције. Приликом развоја овог контролисаног језика кориштен је натуралистички приступ, што је довело до тога да је језик течнији и разумнији за корисника, али да је тежак за контролу јер корисник није увијек у могућности да предвиди како ће процес разрјешења двосмислености завршити [37]. Потребно је доста вјештине како би се користио CPL за специфицирање програмских захтјева, узевши у обзир да интерпретер никада неће бити 100% савршен, него ће корисник морати да научи како да „контролише звијер“ како би се осигурало да систем разумије прави смисао програмског захтјева [112].

CPL је у могућности да конвертује текстуални програмски захтјев писан у стилу програмског језика [2], [37] и ту у доброј мјери прати препоручену структуру програмског захтјева у облику шаблона: *субјекат + глагол + допуне + прилози (мјесто, вријеме, начин)*. Постоје три типа реченица прихватљива од стране CPL: *основна чињеница, питање и правило*[37]. Када су у питању програмски захтјеви, *чињенице* и *правила* се користе за креирање програмског захтјева, док се *питање* користи као основа за верификацију програмског захтјева.

Чињеница је основни градивни елеменат програмског захтјева и могућа је у три облика (енглески):

- *There is|are NP*
- *NP глагол [NP] [PP]*
- *NP is|are пасивни глагол [by NP][PP]*

Гдје је: *NP* – именична фраза, а *PP* – приједлошка фраза

Питање има пет могућих форми, а при постављању верификације најчешће се користи: *Is it true that* ***реченица.*** Програмски захтјеви су у ствари *правила*, а *правила*, се граде као реченице помоћу *чињеница* и других *правила*, која се могу спајати са ријечима IF, THEN, AND, ABOUT, BEFORE, AFTER.

Истраживања показују да је CPL погоднији за спецификацију али и верификацију него други контролисани природни језици, јер не користи додатне алате, а и сам натуралистички приступ олакшава кориштење [37]. Када је у питању постављање **зависности** између захтјева, она може настати сама од себе уколико то тако систем схвати, али није могуће инсистирати на зависности између реченица. Ипак додатни алат *OWL Verbalizer* може помоћи да се ове везе зависности, као и **груписања** стварно створе [37]. Када је у питању **предусловљавање**, саме кључне ријечи омогућују условљавање изван контекста других захтјева [112], ипак могућа је појава и неких нежељених попратних ефеката услова, који у ствари не постоје. **Сљедивост** не постоји, јер је систем превише стохастичан да би могао користити везе за било какву сљедивост [37], [112]. Како не постоје опипљиве зависности између захтјева које се могу описати, нема могућности за **провјеру цикличности**. Уколико се и користи *OWL Verbalizer* којим би се ове додатне везе створиле, моугуће је да се неће моћи креирати логичко стабло извршавања и закључивања, што значи да провјера цикличности ни у том случају није могућа [37], [112], [113]. Постоје истраживања која указују на потребу **јединствене идентификације** унутар CPL, која тренутно није имплементирана [113], као и начине њене имплементације. **Ниво озбиљности** није могуће имплементирати у CPL језику [37], [112], [113], али је могуће проширити језик за додатним алатом резоновања [113] *Logic Programming with Defaults and Argumentation* (LPDA) [114], како би у оквиру њега могли да се приоритетизују правила и чињенице, а самим тим и цијели програмски захтјеви.

7.5.3 Аутоматизација тестова - ATA

Ово рјешење полази од становишта да тестирање почиње од програмских захтјева. Рјешење тежи да, кроз језик аутоматизује претварање програмских захтјева у тестове [115]. Циљ овог језика је да трансформише природни језик програмских захтјева у тестове писане у неком од GPL језика. У основи, у питању није контролисани природни језик, али проширује природни језик, на начин да аутоматски парсира, а затим на себи својствен начин резонује и доказује програмске захтјеве. У питању је рјешење које је синтеза контролисаног језика и тестног алата који може да се користи за верификацију програмских захтјева. Алат омогућује тестерима који нису програмери да помоћу њега аутоматизују тестове на основу програмских захтјева. Ово изгледа као огромна предност алата, јер корисници који нису програмери, могу на једноставан начин да верификују свој скуп програмских захтјева.

Процес аутоматизације се одвија у неколико корака у којим се прво ради сегментација цијеле реченице на неколико прелиминарних сегмената, а затим се сваки сегмент замјењује са најподеснијом n-торком (максимално: тројке), која у себи може садржавати: *акцију*, *циљ* и *податке*. Овим се одстрањују недвосмислености, али сам процес није једноставан, јер се из појединачног сегмента често може издвојити безброј n-торки, због чега је при одлуци која је n-торка најподеснија, учествује човјек. Ипак, овај корак одлучивања је уједно и корак учења, па у коначници, машина преузима одлучивање у највећем броју случајева. Сљедећи изазов представља начин на који се од датих n-торки креирају тестни случајеви. У овом кораку лежи допринос овог рјешења. У суштини се процес трансформације претвара у процес синтезе у којем се траже изводиве путање од дате n-торке. Изводљивост се провјерава на самој апликацији која се тестира. Могуће је да се деси да дата n-торка није изводива, па се систем враћа корак назад на избор друге n-торке, из чега даље учи.

Главне компоненте овог система укључују процесор природног језика, уз додатне интерпретере извршавања, учења и репродукције. ATA, као процесор природног језика користи двије алтернативне опције парсера природног језика: *Stanford Parser* [116] и *Link Grammar Parser* [117]. Интерпретери извршавања учења и репродукције су обједињени у компоненти која се назива *Runtime Interpreter*. Тренутно у ову сврху се могу користити сљедећи алати: *HtmlUnit* [118], *Selenium* [119], и комерцијални алат *Rational Functional Tester* [120].

Овај језик је **једноставан** за коришћење и не захтијева висок ниво експертизе корисника [115], [121]. Сам језик генерише идентификаторе, помоћу xPath алата који **јединствено идентификују** сваки тест [121]. **Ниво озбиљности** не постоји, тестови не носе тежинске факторе [115], [121]. **Сљедивост** и **груписање**, иницијално није могуће остварити [115], али уколико се ова техника комбинује са *Text2Test* [115], [122] алатом могуће их је остварити. Везе **зависности** постоје али падањем једног теста, сви тестови који зависе каскадно у стаблу тестова, неће се сматрати неизвршивим, него ће се сматрати да су пали [121]. Међутим, истраживања су показала да тестне путање које су настале на основу тестног стабла зависности, могу бити бесконачне и да ће у том случају бити погрешно протумачене као успјешно верификоване [115], [121]. Ово значи да је **цикличност** дозвољена, и погрешно протумачена. Није нађена потврда да је могуће условљавати тестове изван контекста другог теста.

7.6 Постојећи алати за тестирање на бази доменски-специфичних језика

Већина алата за тестирање на овај или онај начин може бити употријебљена за вериификацију тестирањем, стога се и истраживање фокусирало и на алате за тестирање на бази доменски-специфичних језика (*testing-specific language*). У наставку ће бити наведени сви истражени DSL-ови за тестирање, њихове могућности, као и област њихове примјене. Неки од њих не служе за верификацију функционалних захтјева, али су наведени из разлога како би се показала сличност у раду, концепту, особинама и функционалностима које покривају.

7.6.1 Photon

Photon је DSL за тестирање, који у позадини агрегира разне тестне алате, како би извршио тестирање [123]. Велики је индекс поновне искористивости чак и ако се промијени алат испод – изворни код писан у *Photon*-у остаје исти. *Photon* посједује прилагодни слој који преводи синтаксу овог језика у синтаксу разумљиву специфичном алату за сам процес тестирања. Овим се тесном инжињеру омогућује да користи безброј тестних алата, без стварног познавања истих. Осим тога, уколико се појави нови тестни алат, сами тесни случајеви не морају бити мијењани. Тестирање је сведено на тестне случајеве, као елементарну честицу тестирања, а сваки од њих мора

да имплементира три функције *init*, *run* и *report*, уз додатно дефинисање позадинских апликација.

7.6.2 Gatling

Gatling је интерни текстуални DSL, базиран на програмским језицима *Skala*, *Akka* и *Netty*, који кориснику даје могућност тестирања нефункционалних захтјева (провјера перформанси) [124] *web* апликација. Сам језик у позадини користи друге тестне алате. Подржава протоколе као што су HTTP, WebSockets, JMS, као и MQTT и ZeroMQ. Основна јединица су сценарији, слично тестним случајевима код *Photon*-а. Сценарио се такође, слично састоји од три функционалности које морају бити издефинисане: *exec*, *pause* и *report*.

7.6.3 Canopus

Canopus [125] је DSL који има своју и текстуалну и графичку представу, базиран на коришћењу *случајева* (слично као и у другим DSL). Сами програмски захтјеви се моделују у сценарије на основу којих се касније генеришу тестне скрипте у зависности од самог алата за тестирање. Служи за верификацију нефункционалних захтјева и сматра се савршенијим од *Gatling*-а. По типу тестирања, то је тестирање базирано на моделу - на основу модела се аутоматски генеришу тестни случајеви. Слично као *Gatling* и *Photon*, и *Canopus* користи додатне алате испод самог DSL.

7.6.4 coNCePTuaL

coNCePTuaL (*Network Correctness and Performance Testing Language*) [126] је програмски језик за верификацију нефункционалних захтјева, првенствено захтјева перформанси. Језик је портабилан – исти тестови се могу користити на било којем слоју, такође је читљив јер подражава енглески језик, и могуће је извршити исти тест више пута. Није шаблонизиран као претходна три језика, него је више као GPL и компајлерског је типа. Примјера ради, по проширеној Бакус-Наур форми (EBNF – extended Bakus-Naur Form) граматика овог језика садржи 56 форми, а граматика ANSI C програмског језика садржи 64 форме. Овај језик се не ослања на додатне алате јер је крајњи производ компајлера изворни код C програмског језика.

7.6.5 ATAP

ATAP (*Accelerating Test Automation Platform*) [127] је компајлерски тип тестног програмског језика, који омогућава тестеру аутоматизацију и извршавање тесних скрипти користећи (контролисан) природни енглески језик. DSL је и развијен узевши у разматрање да треба бити разумљив и не-техничким особама, чиме је омогућено да буде користан и за спецификацију, али и верификацију програмских захтјева [128]. Синтакса језика прати популарну граматику Gherkin [129]. Крајњи производ компајлирања је изворни код генерисан у програмском језику Јава који користи *Selenium WebDriver API*. Извршавањем генерисаног кода верификују се програмски захтјеви. Истраживања су показала да систем аутоматски покрива мало више од 70% свих потребних тестних случајева за верификацију, али и да су неки лексички конструкти прилично компликовани за не-техничке особе [127]. Ипак, примјена овог DSL може да умањи потребу за укључивањем човјека за 25%.

7.6.6 LiFT

Дизајнери програмског језика *LiFT* [130] су се при креирању водили истим циљевима као и креатор *ATAP* програмског језика: да писање тестова буде што је могуће више литерарно, а што мање програмерско. Ово би омогућило уплив експерата дате области који нису програмери у процес тестирања, односно аутоматско повезивање програмских захтјева, тестова и верификације. У питању је компајлерски текстуални програмски језик за тестирање веб апликација, који се у великој мјери ослања на *HtmlUnit* [118] при интеракцијама са интернетом. Користи исте концепте тестних случајева унутар којих се верификују жељене вриједности. Конструктивни дијелови језика су:

- *Action* – метод за интеракцију са системом,
- *Implicit Variable* – стање у којем се налази систем у облику скупа промјењивих,
- *Finder* – проналази графичке елементе који се тестирају,
- *Refinement* – други дио *Finder*-а – оно што буде пронађено од стране *Finder*-а може бити филтрирано,

- *Constraint* – конструкција која провјерава тачност неке тврдње – крајњи алат верификације.

7.6.7 TTCN-3

TTCN-3 (*Testing and Test Control Notation v3*) [131] је тестни текстуални DSL који се користи за тестирање у комуникационим системима, такође је заснован на тестним случајевима и дозвољава аутоматско генерисање тест скрипти. Развијен је и стандардизован кроз Европски Телекомуникациони Институт за стандардизацију ETSI, а изабран је од стране AUTOSAR (Automotive Open System Architecture) групе као званични језик за тестирање. Програмски језик је добро дефинисан, али није једноставан за учење. Проширив је и флексибилан највише због адаптера који одвајају систем који се тестира и платформу од самог тестног система. Језик такође укључује и листу језичких структура и објеката као што су компонента за паралелно тестирање, механизам за повезивање тестова, пресуда тестирања и управљање тајмером. У многоме личи на стандардне GPL па је ријетко кориштен директно за верификацију програмских захтјева, из разлога што свако ажурирање програмских захтјева повлачи за собом ручно ажурирање верификационог програмског кода.

7.6.8 Simulink

Simulink [132] је екстерни компајлерски DSL у обје представе: графичкој и текстуалној. Погодан је за тестирање вођено моделом, као и за системско тестирање. Углавном се *Simulink* користи како би се специфицирали програмски захтјеви у облику модела [133], а затим се помоћу додатних алата постиже тестирање, односно верификација. Оваквим приступом се остварује висока покривеност програмских захтјева тестовима, као и комплетност која је битна када је у питању SCS системи [134]. Језик није једноставан за учење и крива учења је веома равна те се од корисника захтијева да има доста експертизе из области моделовања. Животни циклуси за развој пословних апликација углавном избјегавају коришћење *Simulink*-а. Генератор извјештаја је задовољавајући и подесан је за тестирање вођено моделом, али у случају класичног животног циклуса развоја, извјештавање није од помоћи самом процесу тестирања. Као алат за верификацију, користи се за верификацију анализом, а како је то вођено моделом, генератор извјештаја је од користи за дати процес.

7.6.9 TESLA – програмски језик за спецификацију тестирања

TESLA (*Test Specification Language*) је екстерни, текстуални, интерпретерски DSL који се користи унутар тестне апликације *CAST* (*Computer-Aided Specification and Testing*) која је углавном фокусирана на тестирање уграђених софтверских система [135]. Интерпретерска машина је развијена помоћу програмског језика *SCALA*. TESLA изгледа више као формални програмски језик па је стога, једноставан програмерима, али не и експертима из других области. Спецификација програмских захтјева је могућа као и њихова аутоматска верификација, али експерти морају бити вични програмирању.

7.6.10 Задовољавање метрике

Оно што је заједничко за све ове тестне програмске језике јесте да на овај или онај начин користе концепт сценарија/случајева коришћења/тестних случајева, који су атомски кораци у тестирању. Један атомски корак углавном садржи три подкорака: постављање почетних услова, извршавање и провјеру добијеног понашања у односу на очекивано. Један скуп DSL-ова (*Canopus, coNCePTuaL, Photon, Gatling*) нису фокусирани на функционалне програмске захтјеве, док су други (*TTCN-3, Simulink* и *ATAP)* фокусирани на моделирање како би извршити тестирање у циљу верификације. На крају, скоро сви тестни програмски језици, имају веома равну криву учења (*TESLA, Simulink, TTCN-3, conceptual, ATAP*), а како је потребно укључити експерте домена у спецификацију програмских захтјева, ово онемогућава њихово масовније коришћење у сврху верификације функционалних програмских захтјева.

Оно што је типично за све тестне програмске језике који моделирају јесте да модели представљају машине коначног броја стања, нису Туринг-комплетне [136], па због тога није могуће представити бесконачну петљу. И остали језици имају проблем са рјешавањем проблема цикличне зависности, Тако, на примјер компајлер од *coNCePTuaL* увијек развија петљу у коначан број понављања корака петље [126]. Такође, варијабле не могу бити везане за један задатак, односно верификацију једног захтјева, па тако ни варијабла **јединствене идентификације** није могућа (осим у случају TTCN-3 [137]). **Нивои озбиљности** су генерално присутни у свим тестним програмским језицима, укључујући и различите начине израчунавања коначне пресуде [131]. Постављање почетних услова, први дио атомског корака се користи за

постављање услова извршења тестног случаја, односно **предусловљавања** [127], [130], [131]. Из истог корака могуће је направити да се тестни случај не извршава уколико неки други тестни случај није извршен – **зависност**. **Груписање** тестних случајева је уобичајено у тестним системима (*test suite*) и посједују га и тестни програмски језици. Не постоје покушаји да се постигне **сљедивост** ни у једном од тестних програмских језика, осим када је у питању *Simulink*, уз додатне алате који повећавају комплексност система [138].

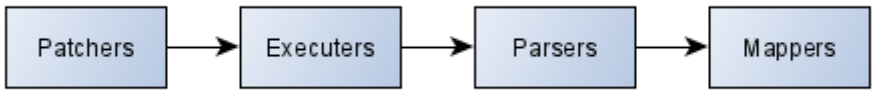
7.7 Програмски језик за тестирање у V-моделу животног циклуса развоја софтвера

Тестни програмски језик који је настао са циљем да имплементира сљедивост у процесу тестирања, коришћен је истовремено и у тестне и у сврху верификације програмских захтјева у V-моделу животног циклуса развоја софтвера [139]. Програмски језик је екстерни, текстуални, интерпретерског типа, као и сви други тестни програмски језици, састоји се од тестних случајева као основних елемената. У овом случају, тестни случајеви се описују помоћу XML. Језик је написан помоћу програмског језика Пајтон.

Дати програмски језик је специфичан јер је фокусиран на верификацију функционалних програмских захтјева за систем који обрађује фајлове или стримове. Из датог разлога DSL је фокусиран на подешавање улазних фајлова и провјеравање добијених и очекиваних резултата у фајловима. Тестни случајеви садрже атрибуте који омогућују сљедивост и ниво озбиљности.

Сам програмски језик је развијен у облику шаблона састављеног од 4 корака из два разлога:

- Због имплементирања правила и ограничења
- Због специфичности програма чије програмске захтјеве верификују



Слика 10 - Извршавање верификације[139]

Слика 10 приказује секвенцијалну рутину корака извршавања овог програмског језика. Такође приказује шта сваки од тестних случајева садржи. Четири врсте специфичних задатака, који могу бити:

- *Patcher* – задатак који се бави измјеном фајлова, односно постављање почетних услова
- *Executer* – извршава софтвер који се верификује на начин тражен у датом функционалном програмског захтјеву,
- *Parser* – извлачи из излазног фајла специфичне вриједности, користећи *xPath* и *regular expression*.
- *Mapper* – мапира извучене вриједности са очекиваним, и изводи закључак да ли је дати случај верификације прошао или није.

Слика 11 приказује примјер изворног кода једног тестног случаја писаног у овом програмском језику. Може се рећи да језик имплементира предуслове помоћу подешавања вриједности атрибута *input* чворова `<patch>`. На слици су два чвора, али их у суштини може бити произвољан број. Њихова сврха је да промијене изворни улазни фајл на начин да буде подешен у складу са траженом ситуацијом из програмског захтјева. Слика приказује манипулацију са XML фајлом, на начин да се изваци један чвор, дефинисан путањом, али овај чвор може изазвати било коју манипулацију над XML, обичним текстуалним фајлом као и бинарним фајлом.

Чвор `<execute>` дефинише извршавање команде, као и путање из које се извршава. Мора постојати један или више чворова овог типа. Чвор `<parse>` дефинише начин на који ће се подаци извући из фајлова или стримова. Као и *Patcher* ради на три типа података XML, текстуалном и бинарном. Подаци који се ваде се чувају у варијаблама именованим у овом чвору. Управо ове податке користи посљедњи тип чвора, `<expected>`. Овај тип чвора мапира вриједности добијене у претходном кораку парсирања са очекиваним вриједностима. Управо овај чвор извршава провјеру, односно верификацију програмског захтјева.

```
<system_tests>
    ...
    <test_case id="requirement_id_1.60"
            description="Correct reaction on missing port"
            name="missing_port_1">
        <patch format="xml" input="./prepared_input/network_1.xml"
            output="./input/network.xml" description="Remove port from switch">
            <remove>
                <switch name='SWITCH_1'>
                   <port name='PORT_16'/>
                </switch>
            </remove>
        </patch>
        <patch input="./prepared_input/specification_1.60.xml"
            output="./input/specification.xml"
            description="just copy from prepared values">
        </patch>

        <execute path = "./network_tools"
            command="check_network -S specification.xml -n network.xml -R network_report.xml"
            description="Execure check newtork with missing port on switch 1"/>

        <parse format="xml" input="./report/network_report.xml"
                description="find expected failed constraints">
            <find_attribute path=".//messages/error" attribute="check"
                            name="failed rule"/>
        </parse>

        <expected related="failed rule" minOccurrence="1">Link_Failed_1</expected>
        <expected related="failed rule" minOccurrence="1">Port_Missing_1</expected>
        <expected related="failed rule" minOccurrence="1">Edge_Failed_1</expected>
    </test_case>
    ...
</system_tests>
```

Слика 11 - Примјер тестног случаја[139]

Резултат рада програма писаног у овом програмском језику је излазни фајл у *JUnit* формату [140] приказан је на слици Слика 12. Извјештај се лако може претворити у формат који је више прикладан за људе, али и аутоматски учитати у дати систем програмских захтјева помоћу идентификатора тестних случајева. Ту је, такође, као успутна добит, и оцјена перформанси самог система, на основу временских података, као и кватификације резултата.

```
<testsuites id="20200512" name="SystemRequirementSpecification" tests="125"
            failures="12" time="1.241">
  ...
  <testsuite id="srs.tests" name="SRS Test" tests="35" failures="37" time="0.891">
    ...
    <testcase id="requirement_id_1.60" name="missing_port_1" time="0.021">
      <failure message="failed rule:Port_Missing_1:0 occurrences" type="ERROR"/>
      <failure message="failed rule:Link_Consistency_1:1 occurrences" type="ERROR"/>
    </testcase>
    ...
  </testsuite>
</testsuites>
```

Слика 12 - изглед извјештаја[139]

Лоше стране овог поројекта су:

- Превише нових појмова (*executor, patcher, parser…*) иако прилично добро одговарају уобичајеном скупу активности при верификацији.
- Потребно знање XML.
- Фокусираност на V-Модел циклус развоја софтвера.
- Фокусираност на специфичну врсту тестирања која укључује рад над фајловима. Улаз у систем који се верификује мора бити фајл или стрим и излаз из систем који се верификује мора бити фајл или стрим.
- Провјера синтаксе није комплетна, укључујући и недостатак провјере цикличности.

Јединственост постоји и везана је за извјештавање, **ниво озбиљности** је дефинисан у три нивоа, а могуће је и **слиједити** програмске захтјеве. **Груписање** се остварује помоћу припадности чвору *testSuite*. Остале особине нису подржане.

8 Резултати истраживања

У табели Табела 2 приказани су резултати истраживања сродних радова, а на основу предложене метрике. Дате су три врсте оцјена за особине и функционалности: подржава (П), дјелимично подржава (Д) не подржава (Н). Оцјене су грубе а уз додатна објашњења наведена помоћу фуснота су дјелимично боље прецизирана. Иако комплексност није дио метрике, налази се у табели из разлога што је за квалитетно писање програмских захтјева изнимно битно укључити експерте из дате области, а сама оцјена колико је једноставно људима који најчешће нису програмери да усвоје програмски језик је битна оцјена сваког DSL [35]. Постоје три нивоа оцјене за комплексност: висока, средња и ниска.

Оцјена да DSL дјелимично подржава неку особину или функционалност значи да уз додатне алате је ипак могуће подржати дату особину, или да је особина подржана под одређеним специфичним случајевима. Уколико би се користили додатни алати, комплексност система би се повећала [27], [58], [86]. Оцјена комплексности је дата у три нивоа, из разлога што постоје истраживања која пореде комплексност типова језика: формалних, натуралистичких и контролисаних [27], [37], док друга пореде комплексност појединачних језика [58], [86], [141], али у највећем броју три DSL, те је стога, довољно имати три оцјене за међусобно поређење.

Упоредна табела представља агрегацију оцјена свих алата наведених у претходном поглављу (*7 Сродни радови и истраживања из области верификације функционалних програмских захтјева*). Свака оцјена је проистекла из радова наведених у том поглављу, осим оцјена које су наведене за посљедњи алат: програмски језик за тестирање у V-моделу. Ту је извршена анализа на основу субјективне процјене.

Из упоредне табеле се јасно види да ни један од сродних алата није у могућности да у потпуности одговори на захтјеве проистекле из стандарда и *најбоље праксе* везане за DSL, верификацију и спецификацију функционалне програмских захтјева. Најочигледнији недостатак, по правилу, је провјера цикличности.

DSL	јединственост	Ниво озбиљности	слједивост	зависност	Провјера цикличности	груписање	предуслови	комплексност
URN[34]	П	П	П	П	Н	П	Н	висока
RSLingo	П	Д[5]	Д	Д	Н	П	П	средња
Promela	П	П[6]	Н	П	Н	Н	П	средња
RM-RNL	П	Н	П	П	Н[7]	П	П	висока
Attempto[8]	Д	Д	Н	Д	Н	Д	П	средња
CPL[9]	Н	Д	Н	Д	Н	Д	П	средња
ATA[10]	П	Н	Н	П	Н	Н	Н	ниска
Тестни програмски језици	Н[11]	П	Н[12]	П	Н	П	П	-
Претходно истраживање	П	П	П	Н	Н	П	Н	средња

Табела 2 - упоредна табела испуњености затхјева VSL

[3] Слједивост, зависност и груписање имају проблем са одржавањем веза

[4] Провјера цикличности и предуслови могући коришћењем додатних алата

[5] Помоћу RSL4CYBERSECURITY

[6] Не користи се код рачунања укупне оцјене

[7] Приједлог коришћења вјештачке интелигенције

[8] Постоје додатни алати за сваку дјелимично задовољену особину и функционалност

[9] За недостатке, предлаже се *OWL Verbalizer*

[10] Додатни алат *Text2Test* омогућује слједивост и груписање

[11] Осим TTCN-3

[12] Осим *Simulink*

9 Потврда концепта

На основу добијених резултата истраживања, јасно је да VSL мора бити разумљив без икаквих двосмислености за обје стране: наручиоце и добављаче, али у исто вријеме и извршив. Елементи језика (концепти из домена за који се прави софтвер и правила језика) који описују систем и специфицирају програмске захтјеве морају имати све неопходне особине како би се успјешно извршила верификација, наведене у таблели Табела 1.

У сврху потврде концепта биће предложено Окружење, базирано на принципу интерпретера, развијено у програмском језику Пајтон. Окружење нуди API (*Application Programming Interface*) за мапирање свих обавезних особина и подржава развој текстуалног интерпретерског VSL и извршавање изворог кода. Дијелови Окружења који су задужени за провјеру јединствености, индекса, груписања, релација зависности између програмских захтјева, већ су имплементирани. Одређени дијелови процеса се морају имплементирати, док је за друге могуће користити предложене функционалности. Ипак, ове предложене функционалности се могу подесити другачије. Окружење представља концепт кодирања (детаљније описан у поглављу 4.4

Дефинисање доменски специфичног језика), а за потпуну дефиницију DSL потребно је да се дефинише још и метамодел и детаљи имплементације. Како би описали цијело Окружење, сви дијелови система, као и принцип рада и подјеле одговорности ће бити описани у наставку.

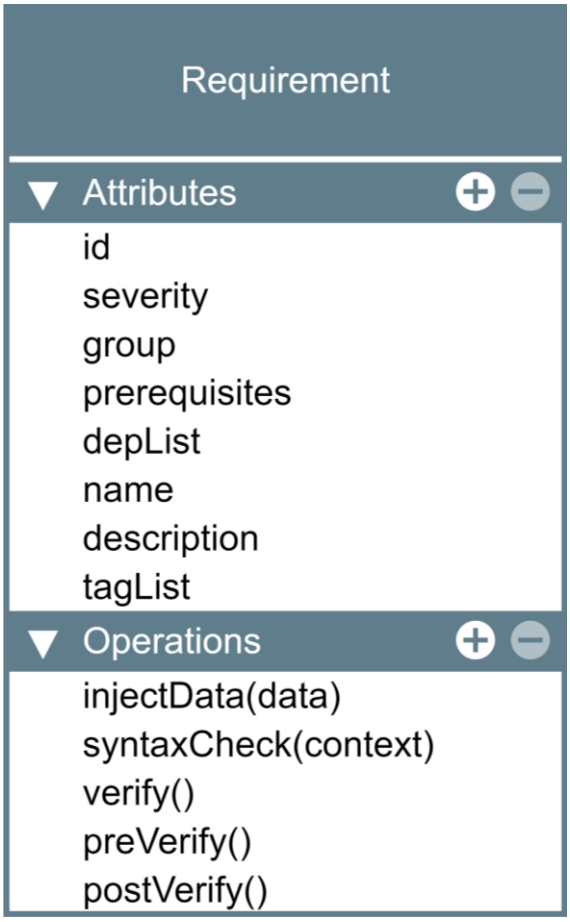
9.1 Шаблон програмског захтјева

Приликом истраживања инжењеринга програмских захтјева, али и приликом истраживања DSL, уочено је коришћење шаблона [16], [20], [37], [75]. Шаблони поједностављују ствари, али уједно и обавезују одређена дефинисања и имплементације, као и ограничења. Употреба шаблона проистиче из потребе дефинисања правила програмског језика, а уобичајени бенефити су [35]:

- Смањење броја грешака, јер поједине конструкције једноставно нису могуће,
- Пожељни обрасци дизајнирања

- Провјера комплетности – информисање о дијеловима који недостају. Иако није битно за извршавање, ова је особина иразито битна за инжињеринг захтјева.
- Минимализовање моделовања, увођењем уобичајених вриједности, функција и конвенција
- Омогућавање консистентности – уколико се неки програмски захтјев промијени, преко релација могуће је промијенити још неки програмски захтијев или промијенити ниво озбиљности.

Шаблон програмског захтјева се користи у **Окружењу**, као мост који повезује новоразвијени VSL за Окружењем. Он омогућује VLS-у да обезбиједи и учита све неопходне податке као што су идентификатори (`id`), тежински фактори (`severity`), начин груписања захтјева (`group`), предуслове (`prerequisite`) и међусобне зависности (`dependency`) у сам верификациони процес. Предуслови се дефинишу као референца на функционалност, чиме су омогућено дефинисање и комплексних предуслова.



Слика 13 - општа представа шаблона програмског захтјева

Шаблон такође омогућује и уношење података који нису обавезни као што је назив захтјева, опис и листа тагова који припадају захтјеву. Тагови нису обавезни нити су предложени у стандардима, али су се показали корисним за дефинисање додатних, неформалних веза између захтјева. Поједностављени приказ шаблона захтјева дат је на слици Слика 13. На слици се поред наведених атрибута виде и основне функционалности које су дјелимично имплементиране.

Функција `injectData` служи да учита конкретне вриједности које су потребне **Окружењу** у процесу верификације конкретног програмског захтјева. У суштини, сам процес парсирања изворног кода, који није дио функционалности **Окружења**, изолује потребне обавезне вриједности и затим их попуњава, користећи функцију `injectData` у једну инстанцу шаблона. Назив за овај процес је **учитавање контекста**.

Уобичајено је да процес парсирања изворног кода прати провјера синтаксе. Она је подијељена на два дијела: за прву је одговорна сама имплементација, а друга (функција `syntaxCheck`) је дио Окружења, која на основу претходно *убризганих* података провјерава синтаксу у оквирима у којима Окружење може да провјери синтаксу једног захтјева. Више о подјели одговорности ће бити ријечи у поглављу 9.3

Подјела одговорности при извршавању верификације. Функција `syntaxCheck` између осталог провјерава:

- да ли је сваки идентификатор стварно јединствен,
- да ли све референциране зависности, као и групе стварно постоје као други дефинисани програмски захтјеви и групе,
- да ли су тежински фактори наведени за сваки програмски захтјев из скупа (ре)дефинисаних тежинских фактора
- да ли постоји цикличка зависност, итд...

Функција `verify` референцира на функцију за верификацију која је такође дио детаља имплементације VSL. Имплементација VSL би требала бити таква да се парсирањем изворног кода добију све потребне информације за извршавање ове функције.

Функције `preVerify` и `postVerify` извршавају функционалност пред-задатака и пост-задатака предвиђених табелом Табела 1 чиме се у потпуности обезбјеђује комплетност и подржава сљедивост програмског захтјева. Функција `preVerify` подешава окружење пред верификацију једног програмског захтјева, на начин да се може извршити потребна верификација, а функција `postVerify` враћа цијело верификационо окружење у уобичајено стање, какво је било прије верификације програмског захтјева. Слично функцијама `setUp` и `tearDown` које се користе на тестним платформама.

9.2 Промјењиве

Синтакса промјењивих је једна од ствари која има своју уобичајену репрезентацију у Окружењу, али ју је могуће подесити другачије. Битно је примјетити да у случају поновног дефинисања синтаксе промјењивих, постоји могућност да ће се и

неке друге уобичајене функционалности морати додатно дефинисати. Ово важи и за предуслове, и њихова уобичајена имплементација ће морати бити предефинисана уколико се предефинише синтакса промјењивих. Све вриједности промјењивих у Окружењу, па самим тим и у VSL се асинхроно учитавају (*lazy loaded*).

9.2.1 Уобичајена нотација

Уобичајени приступ вриједности промјењиве је преко имена промјењиве. У случају комплекснијих структура као што су скупови, листе, структуре и мапе, приступ се дијели на два начина:

1. приступ преко индекса (листе, скупови и н-торке)
2. приступ помоћу имена (када су у питању мапе и структуре)

Примјер како се приступи десетом елементу низа који се зове `arr`:

```
arr[9]
```

Уколико се, на примјер дио структуре која се зове `struct` зове `thePart` и треба да се приступи вриједности тог дијела структуре то би изгледало овако:

```
struct.thePart
```

Наравно, ове се представе могу интерполирати, тако да ако је потребно ако приступити једном дијелу структуре који је низ, и неком члану овог низа, то би изгледало овако:

```
struct.thePart.arr[9]
```

Угласте заграде се могу користи и за софистицирано филтритање елемената низа или структуре. Постоје сљедећа два начина филтрирања:

```
struct.thePart[name="partName"].arr[9]

struct.[value<5, value>3].thePart
```

Прва линија филтрира само онај елеменат структуре којима је вриједност `struct.thePart.name` једнака `partName`, и приступа десетом елементу филтрираног дијела структуре званом `arr`. У другој линији, зарез унутар угластих заграда има улогу логичког И, тако да се у овој линији приступа дијелу структуре која има вриједности `struct.value` између 5 и 3 (искључујући ове двије граничне

вриједности). Када су у питању филтери, постоји једино ова релација логичког И изведена помоћу зареза унутар угластих заграда.

9.2.2 Тежински фактор програмског захтјева

Тежински фактор представља ниво озбиљности, и можесе дефинисати као фактор којим одређени програмски захтјев утиче на функционалност, сигурност и/или квалитет цијелог система. Сваки програмски захтјев има тежински фактор (назван: *severity*) као обавезну особину. Уобичајени скуп вриједности тежинског фактора је `CRITICAL`, `MAJOR`, `MEDIUM` и `LOW`. Сваки од ова четири члана скупа има и своју нумеричку вриједност, која се користи у процесу финалног израчунавања укупног резултата верификације, а уобичајени тежински фактор за сваки програмски захтјев је `MAJOR`. Вриједности, као и сами елементи скупа се могу предефиницати, а уобичајене нумеричке вриједности су:

- `LOW = 0`, што практично значи да уколико овај захтјев и није задовољен, неће утицати на резултат верификације,
- `MEDIUM = 33.33`,
- `MAJOR = 66.66` и
- `CRITICAL = 100`.

9.2.3 Резултати верификације

Резултат верификације појединачног програмског захтјева је такође специјална врста промјењиве која добија своју вриједност у току верификације. Уобичајене вриједности су `PASS`, `FAIL` и `NOT EXECUTED` (јер постоје ситуације, када због предуслова или зависности између програмских захтјева, верификација одређеног програмског захтјева није могућа). Понекад је потребно промијенити број могућих резултата верификације. Узмимо примјер два захтјева, гдје *захтјев1* зависи од *захтјев2*, и неколико различитих ситуација:

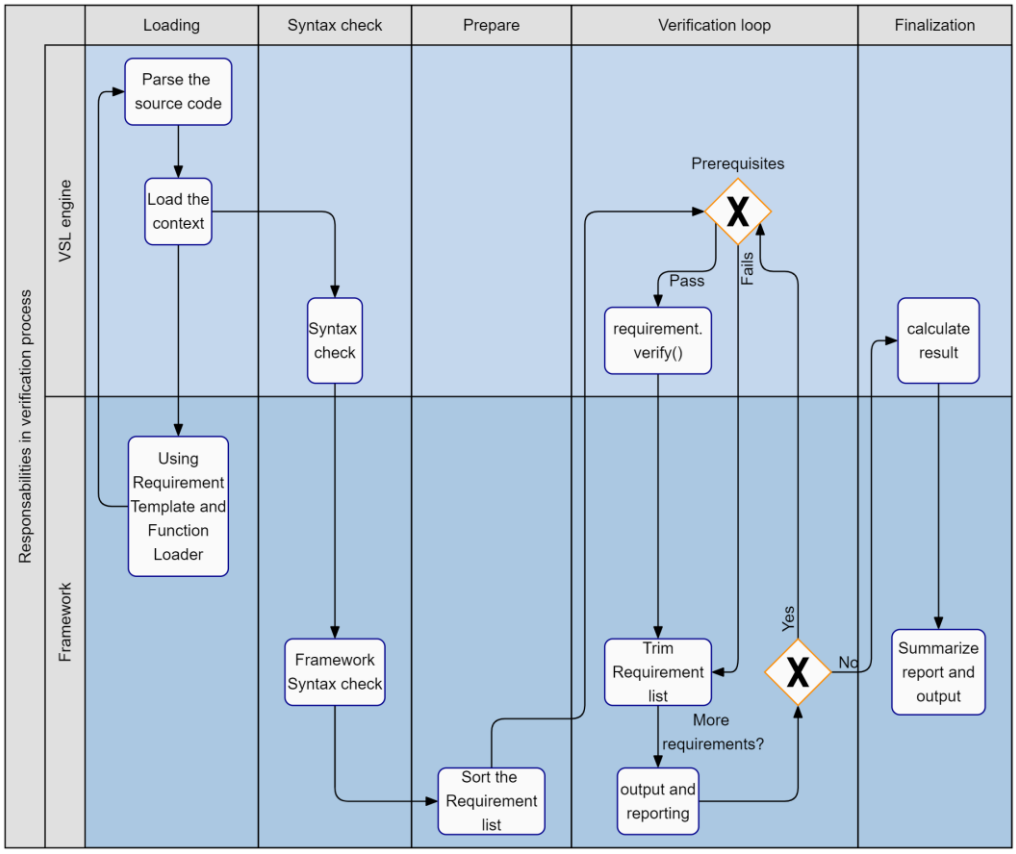
- Верификација захтјева *захтјев1* се не изврши зато што се није извршила верификација захтјева *захтјев2* (из било којег разлога)

- Верификација захтјева *захтјев1* се не изврши јер је верификација захтјева *захтјев2* пала
- Верификација захтјева *захтјев2* је извршена и захтјев је прошао, али верификација захтјева *захтјев1* није извршена због предуслова програмског захтјева који нису испуњени.

Могуће је да ће се ова три случаја другачије третирати приликом евалуације укупног резултата верификације. Ово би значило да је потребно имати још додатних резултата верификације појединачног програмског захтјева. Углавном, скуп резултата је могуће редефинисати уз обавезу да се нови скуп мапира на постојећи, као и да најмање један резултат мора бити мапиран на PASS и најмање један мапиран на FAIL.

9.3 Подјела одговорности при извршавању верификације

Окружење садржи све дијелове верификационог процеса, али неки дијелови су само *чувари мјеста* или апстрактне функције као што је парсирање кода или учитавање контекста (*universe of objects* [60] – скуп вриједности варијабли и показивача на функције који ће бити коришћен у процесу). Остали дијелови процеса, као што су извршавање синтаксне провјере су парцијално дефинисани и извршени у Окружењу, а дјелимично су у надлежности новоразвијеног VSL. Много јаснија и детаљнија подјела одговорности приказана је на слици Слика 14. *VSL engine* на слици представља зону одговорности новог VSL, док *Framework* представља одговорност Окружења.



Слика 14 - процес извршавања верификације са приказом подјеле одговорности[2]

9.3.1 Учитавање и синтаксна провјера

Процес извршавања почиње са учитавањем података, парсирањем самог изворног кода новог VSL. Овај дио процеса је специфичан за сваку представу програмског језика понаособ. Овдје се креира апстрактно синтаксно стабло – AST (*Abstract Syntax Tree*), а затим се учитава контекст на основу којег ће се вршити верификација, укључујући и програмске захтјеве, који настају као инстанце попуњених шаблона програмских захтјева (Слика 13).

Приликом учитавања и парсирања, провјеравају се све обавезне особине програмског захтјева, али је одговорност за овај процес и даље на страни новог програмског језика. Када су у питању предуслови, могуће је користити и синтаксу Пајтон језика, а уколико то није случај, парсер новог VSL је обавезан да парсира и предуслове. У току парсирања провјерава се и синтакса, која је дистрибуирана између

обје зоне одговорности. Када је у питању Окружење, оно, као што је речено, провјерава да ли су идентификатори јединствени, случајеве цикличности релација груписања и зависности. Такође се провјерава да ли су релације груписања и зависности стварне, односно да ли обје стране (оба програмска затјева) постоје.

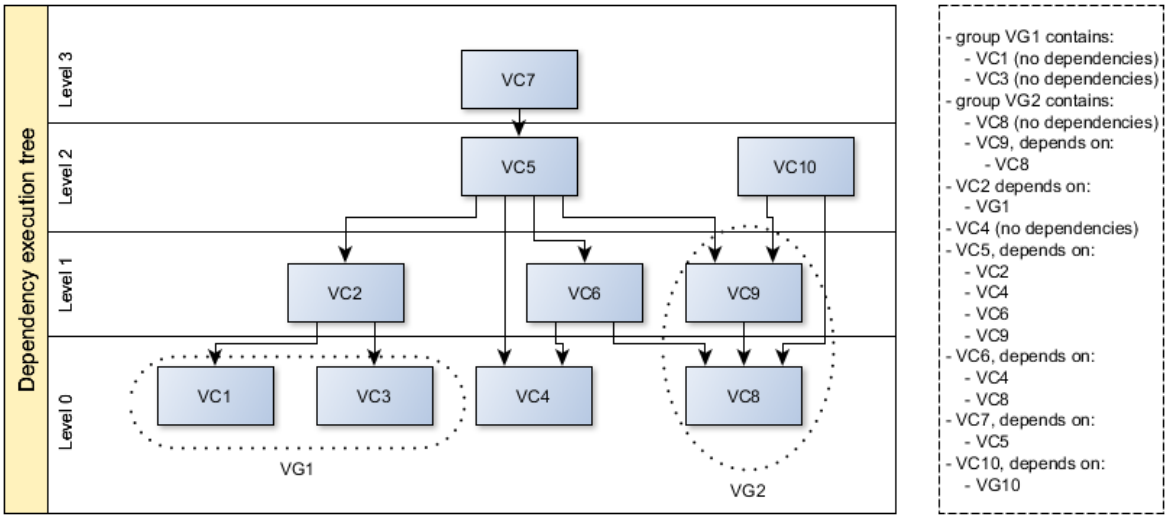
9.3.2 Припремање захтјева - сортирање

По завршетку синтаксне провјере све је спремно за сортирање програмских захтјева, и креирање *графа зависности програмских захтјева*. Граф одређује редослијед верификације функционалних програмских захтјева, јер релација зависности заједно са особином за сортирање успоставља јединствени редослијед верификације програмских захтјева. Од сортирања зависи којим редослиједом ће се извршавати верификације појединих програмских захтјева, а сортирање зависи од веза између појединих захтјева. Уколико постоји цикличност, захтјеви се не могу сортирати, а самим тим није могуће извршити верификацију. Управо због овога је било потребно онемогућити постојање цикличности. Овдје постаје јасно да редослијед захтјева у којем су они специфицирани у изворном коду, није релевантан извршење њихове верификације. Извршиће се прво они захтјеви који не зависе ни од једног програмског захтјева, па тек онда остали, опет по правилу да они којима су сви захтјеви од којих зависе постављени у редослијед извршавања. Како је могуће имати више захтјева на истом нивоу, особина која ће служити за сортирање ће омогућити увијек исти редослијед међу програмских захтјевима. На овај начин сваки пут када се буде извршавала верификација, она ће бити извршена истим редослиједом – чиме се обезбјеђује конзистентно извршавање саме верификације.

Слика 15 приказује један примјер сортирања програмских захтјева. Како би се на слици нагласила комплексност, над датим усмјереним ацикличним графом релације зависности, са слике није извршена транзитивна редукција. Оно што је јасно на крају процеса провјере синтаксе јесте како изгледа стабло зависности програмских захтјева. Један примјер је приказан на лијевој страни ове слике. Захтјеви су постављени на нивое, на основу тога од колико посредних нивоа захтјева зависе. Захтјеви који не зависе ни од кога су на нивоу 0. Ниво 1 садржи захтјеве који зависе од било којег броја захтјева са нивоа 0. Захтјеви са Нивоа 2 зависе од захтјева са нивоа 1 и нивоа 0.

Правило је да захтјеви са нивоа *n*, зависе од барем једног захтјева са нивоа *n*-1 и никако ни од једног захтјева са вишег нивоа од *n-1*.

Када се установи редослијед нивоа, остало је додатно правило које одређује редослијед верификације програмских захтјева унутар истог нивоа, а то је да се прво извршавају захтјеви који имају нижи идентификатор (идентификатор је у овом случају особина захтјева која се користи за сортирање). Редослијед са примјера са слике Слика 15 је сљедећи: VC1, VC3, VC4, VC8 (ниво 0), VC2, VC6, VC9 (ниво 1), VC5, VC10 (ниво 2), VC7 (ниво 3).



Слика 15 - примјер сортирања програмских захтјева[2]

9.3.3 Верификациона петља

Итерација којом се пролази кроз дату сортирану листу програмских захтјева се назива верификациона петља. Унутар ње се дјелимично извршавају фунционалности из Окружења, а другим дијелом функционалности дефинисане и развијене у новом VSL. Уколико захтјев посједује предуслове, исти ће бити провјерени, и уколико ови услови прођу, функција *verify* шаблона програмског захтјева ће бити извршена (прије ње *preVerify,* а послије ње *postVerify* са слике Слика 13). Уколико се деси да процес падне или на предусловима или на верификацији, долази до кљаштрења стабла извршавања. Процес кљаштрења стабла извршавања избацује све захтјеве који су зависили од

програмског захтјева чија је верификација управо пала (или је пао предуслов верификације). У случају да је скуп резултата верификације редефинисан, процес кљаштрења ће се фокусирати само на резултате који су мапирани на FAIL или NOT EXECUTED.

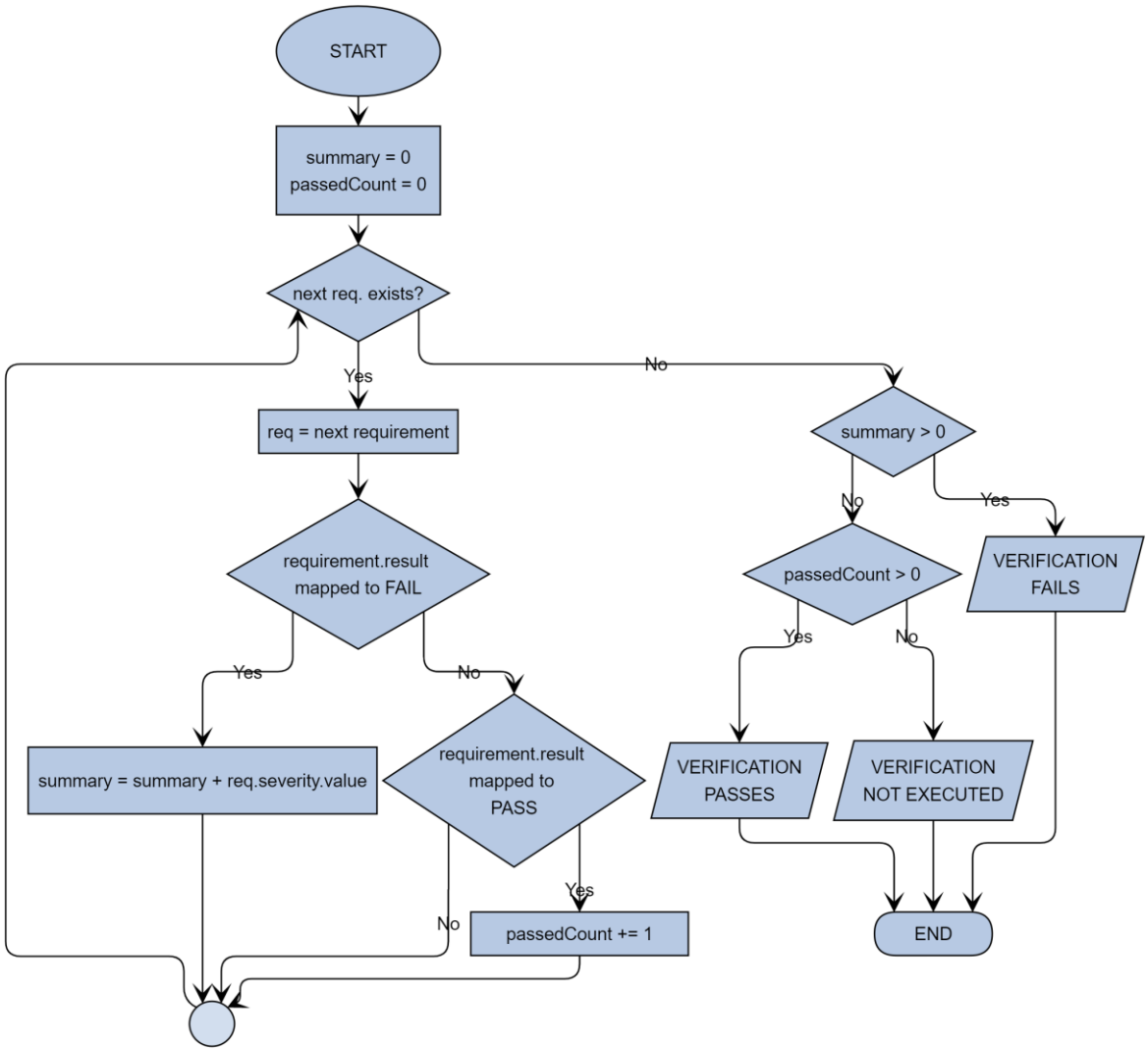
Сви захтјеви који су избачени из стабла, биће означени са једним од резултата дефинисаним у поглављу *Резултати верификације* на страни 84. Послије кљаштрења, сви резултати (и верификованих програмских захтјева, као и евентуално прескочених захтјева) ће бити процесирани помоћу функције (*output and reporting* – Слика 14) која служи за прихватање и обраду резултата који ће бити приказани на крају процеса у облику исписа на екрану или извјештаја. Ова функција олакшава израчунавање крајњег резултата, као и брже и концизније извјештавање на крају верификације. Када заврши ова функција, петља наставља од почетка, све док има захтјева у листи.

9.3.4 Финализација

Финализација почиње када се обраде сви пројектни захтјеви. У овом се кораку израчунава укупни резултат верификације, сумира у извјештај и испис на екран. Окружење имплементира уобичајено израчунавање резултата, које је приказано на слици Слика 16. Уобичајено израчунавање сумира све нумеричке вриједности тежинских фактора сваког појединачног програмског захтјева чија је верификација пала. Уколико је ова сума већа од нуле, цијели процес верификације се сматра да је пао, у супротном, ако постоји барем један програмски захтјев чија је верификација прошла, верификација се сматра успјешном.

Ово значи да уколико падне верификација било којег броја програмских захтјева чији је тежински фактор LOW, сам процес верификације неће бити сматран неуспјешним, јер је нумеричка вриједност LOW једнака нули. Уколико је потребно другачије израчунавање укупног резултата верификације, процес треба да буде додатно имплементиран, као што је то приказано на слици Слика 14.

Задњи корак у процесу финализације, искључиво у надлежности Окружења, јесте финализација извјештаја и исписа на стандардни излаз, на основу претходног израчунавања.



Слика 16 - Уобичајено израчунавање резултата[2]

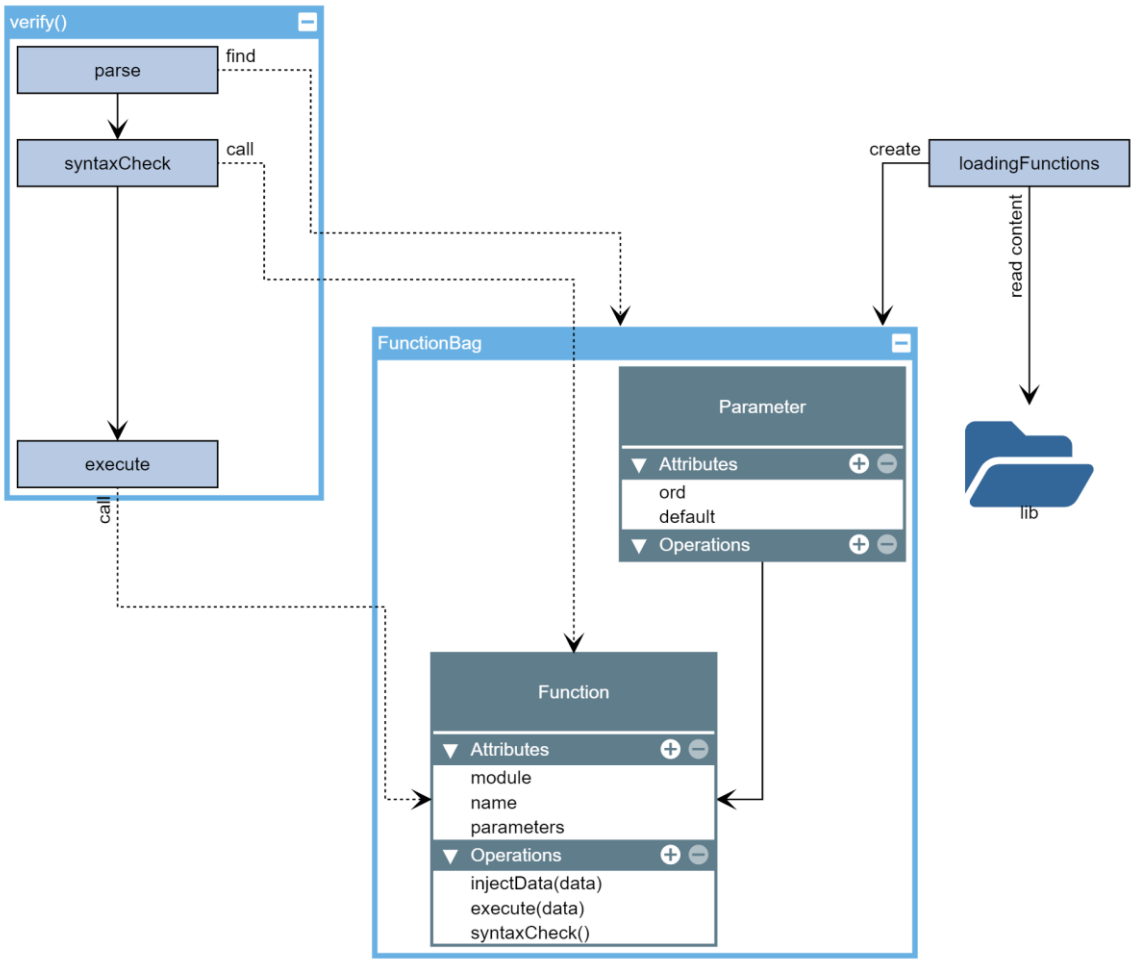
9.4 Могућности проширења

Како би се подржали и захтјеви за промјењивошћу и конзистентношћу (Табела 1), Окружење треба бити у могућности да се прошири додатним функционалностима као и да може да користи библиотеке функција. Могућности проширења, су засноване на начину на који програмски језик Пајтон проширује или обогаћује свој код. Слично Шаблону Програмског захтјева, који је искоршћен као комуникациони мост између онога што је већ развијено и постоји, и онога што је још потребно развити да би се

добио потпун VSL, Окружење користи Функцијски Шаблон, приказан на слици Слика 17.

У процесу учитавања приказаном на слици Слика 14, функције су учитане на начин да је фолдер `lib` скениран, заједно са потфолдерима, у циљу проналаска скрипти писаних у програмском језику Пајтон. Процес учитавања ће креирати инстанцу функције и скуп параметара за сваку функцију унутар сваке Пајтон скрипте на коју наиђе. Сваку ову појаву ће једнозначно чувати у Врећи Функција (*Function Bag*).

Приликом парсирања, уколико парсер наиђе на токен који не може да схвати, потражиће га у Врећи Функција, а синтаксна провјера ће провјерити да ли се функција у изворном коду користи на правилан начин. На крају, када се верификација буде извршавала, извршиће се и функција.



Слика 17 - Функцијски Шаблон

9.5 *Имплементирани примјери*

Како би се процес потврде концепта провео до краја, имплементирана су четири DSL која на различите начине користе Окружење. Први примјер имплементира DSL за тестирање у V-моделу животног циклуса развоја софтвера [139] наведен у поглављу 7.7 Програмски језик за тестирање у V-моделу животног циклуса развоја софтвера додајући му функционалности и особине које нису биле имплементиране. Други примјер помоћу Окружења, побољшава постојећи верификациони процес имплементацијом интерног верификационог програмског језика. Трећи примјер проширује постојећи програмски језик који је кориштен искључиво за спецификацију функционалних програмских захтјева како би могао да се користи и за верификацију.

На крају, четврти примјер користи Окружење за DSL који провјерава исправност за генерисаних података.

9.5.1 Програмски језик за тестирање у V-моделу животног циклуса развоја софтвера

Овај примјер имплементације понаша се на исти начин као и постојеће рјешење програмског језика за тестирање у V-моделу [139]. Циљ је био да се истраже предности и мане развоја програмског језика помоћу Оркужења мјерењем броја линија потребних за имплементацију, као и мјерењем и упоређивањем броја инжињер сати. Исти програмски језик презентован XML језиком је развијен помоћу Окружења, али са другим инжењерима упоредивих програмерских вјештина и знања. Нова верзија програмског језика имплементира груписање, релацију зависности и тежински фактор тестног случаја. Груписање је имплементирано новим тагом `test_suite`, који служи као контејнер одређеног броја тестних случајева (`test_case`). XMLSchema онемогућује да се унутар XML тага `test_case` налази други таг `test_case` или `test_suite`. Груписање је овако ограничено само на један ниво, али овакав начин имплементације онемогућује цикличности релације груписања.

Проширењем тестног случаја са атрибутом `dependencies` омогућена је дефиниција зависности међу тестним случајевима. Зависности се наводе као низ идентификатора тестних случајева, одвојених зарезом. У овом случају је могуће да дође до цикличности релације зависности. Тежински фактор је представљен атрибутом `importance` којем могу бити додијељени бројеви од 1 до 10. Исјечак изворног кода испод приказује дате измјене.

```
<test_suite id="1">
    <test_case id="3"
               dependencies="4,5,6"
               importance="1"
               description="check the missing port"
               name="missing port">
        ...
    </test_case>
    ...
</test_suite>
```

Инжињери су потрошили додатно вријеме да науче како се користи Окружење, али је ово вишеструко надокнађено касније у наставку развоја програмског језика. Када је у питању корисничка документација, произведена је за приближно исто вријеме и оцијењено је да обје документације имају исти ниво јасноће. Ово у ствари, само потврђује да су оба језика развијали инжињери истог нивоа вјештина и знања. Детаљнија анализа између двије имплементације, дата је у резултатима потврде концепта. Програмски језик користи све уобичајене концепте Окружења, као што су резултати, тежински фактори и функције израчунавања коначног резултата, ниједан концепт није додатно редефинисан.

9.5.2 Побољшање постојећег верификационог процеса

Није риједак случај да постоје одређени алати који служе за верификацију програмских захтјева. У поглављу о DSL сугерише се да се овакви алати што је могуће више искористе, како у развоју метамодела, тако и у развоју концепта кодирања или у самим детаљима имплементације. Такав један, постојећи процес верификације је у потпуности развијен као скуп функција у Пајтон програмском језику, груписан у модуле и пакете. Верификација није подржавала већину потребних особина, јединственост програмског захтјева, сљедивост, груписање... Свака функција је верификовала или учествовала у верификовању једног функционалног програмског захтјева. Саме функције верификације су позиване статички дефинисаним редоследом, приликом чега су им прослеђиване потребне информације – улазни подаци. Свака интервенција на програмском захтјеву укључивала је и мануелну интервенцију програмера.

Помоћу Окружења креиран је интерни VSL у програмском језику Пајтон, који заправо користи постојеће функције, али који је обогаћен свим особинама и понашањима из табеле 1.

```
from req_lib import *
id = group_id # group id
depends = [] # the ids group depends on

# requirement id=1, severity=ERROR
```

```
@requirement(1, "ERROR")
def req_1(table):
     # requirement verification body
     if verification_pass:
         return True
     return False

# requirement id=2, severity=WARNING
# depends on requirements id=3 and id=1
@requirement(2, "WARNING")
@dependency(3, 1)
def req_2(data1, data2):
     # requirement verification body
     if verification_pass:
         return True
     else:
         error("%s fails to execute %s" %(data1.c, data2.d))
     return False
# ...
```

На поједностављеном изворном програмском коду VSL-а који је приказан изнад, види се начин на који су имплементиране потребне особине. Умјесто наведеног коментара (`# requirement verification body`) позива се постојећа функција верификације а њен резултат се смјешта у локалну промјењиву `verification_pass`.

Једна функција и даље представља једну верификацију функционалног програмског захтјева док је груписање је имплементирано као Пајтон модул који садржи ову функцију. Варијабла модула названа `id` представља идентификацију групе програмских захтјева, док варијабла модула `depends` представља листу идентификатора других програмских захтјева од којих ова група програмских захтјева зависи.

Стандардна имплементација декоратора у Пајтону је искоришћена за имплементацију зависности, као и за додјелу јединственог идентификатора и тежинског фактора програмског захтјева. Декоратор `@dependency` дефинише листу идентификатора осталих програмских захтјева од којих декорисана функција зависи, док декоратор `@requirement` дефинише идентификатор (први параметар декоратора) и тежински фактор (други параметар) програмског захтјева.

Само Окружење је задужено да када установи све ове вриједности, провјерава постоји ли цикличност, као и да ли су идентификатори јединствени, зависности постоје над стварним захтјевима, и тако даље... Уколико све прође синтаксну провјеру, прво синтаксну провјеру програмског језика Пајтон, а друго Окружења, систем сам направи редослијед извршења верификација програмских задатака, без интервенције програмера (наравно, инжињер програмских захтјева мора бити укључен у промјену зависности која би изазвала промјену редослиједа). Битно је још једном напоменути да првобитне функционалности верификације нису промијењене.

9.5.3 Проширење постојећег програмског језика за спецификацију програмских захтјева

Постојећем програмском језику који се користи за спецификацију програмских захтјева додата је могућност верификације истих, проширењем помоћу Окружења. Постојећи спецификациони језик је способан да недвосмислено дефинише програмске захтјеве, као и уради њихову декомпозицију, дијелећи их на мање, прецизније захтјеве. Овај програмски језик је представљен са YAML (*Yet Another Markup Language*) језиком, а у овом случају је проширен Окружењем како би постао извршни верификациони језик.

Примјер једног од основних (*top level*) програмских захтјева, који је наведен као групни, (комплексни) програмски захтјев са идентификацијом “30-A”:

The configurable application parameters will be stored in the configuration file, named `appName.cfg`. The configuration file will be stored in user’s home subfolder named appName. If configuration file or folder (appName) does not exist, next run of the application will create one with default values. The default values of the configurable parameters are:

- ***parameter1 = value1***
- ***parameter2 = value2***
- ***parameter3 = value3***

```
id: "300-1"
```

```
  desc: "If folder userHome/appName does not exist, appName
start will create it."
  dep: "30-A"
  setup: "folder ~/appName does not exist"
  function: "appName"
  outcome: "exist ~/appName"
  output: "folder created"
    type: "file"
    name: "log/%.log"
id: "301-1"
  desc: "If userHome/appName/appName.cfg does not exist, next
appName start will create it with default values."
  dep:
    - "30-A"
    - "300-1"
  setup: "folder ~/appName/appName.cfg does not exist"
  function: "appName"
  outcome: "exist ~/appName/appName.cfg"
  output:
    - "parameter1:value1"
    - "parameter2:value2"
    - "parameter3:value3"
    type: "file"
    name: "~/appName/appName.
```

Овај је програмски захтјев на крају процеса декомпозиције је подијељен у више основних захтјева. Приказ изворног кода изнад, показује неке од ових основних, декомпонованих захтјева. Параметар `id` је искоришћен као јединствени идентификатор и кључан је за језик, док се `desc` користи за текстуални опис захтјева и није битан за сам процес верификације.

Ипак, параметар `desc` је показатељ мањкавости овог рјешења, првенствено специфицирања програмских захтјева. Наиме, овај параметар је наслијеђен из спецификационог језика и даље обезбјеђује разумљивост инжињерима програмских захтјева јер додатни код није разумљиво сви учесницима у процесу креирања и верификације програмских захтјева. Проблем који овдје настаје је у томе да се приликом ажурирања програмског захтјева може изоставити и ажурирање параметра који описује сам програмски захтјев, а који је за неке учеснике у процесу једини валидан. Овим се нарушава унутрашња конзистентност програмског захтјева.

Листа параметара дефинисана параметром `dep` су идентификатори захтјева од којих дати захтјев зависи. Битно је напоменути да се овдје могу наћи и идентификатори једноставних, али и комплексних програмских захтјева. Уколико се налази комплексни програмски захтјев, то је индикација да је дати програмски захтјев настао декомпозицијом датог комплексног захтјева, односно припадности групи. Овдје постоји конвенција додјеле идентификатора: уколико се идентификатор завршава са `-A` то значи да је у питању групни захтјев. На примјеру изворног кода изнад, види се да су програмски захтјев `301-1` зависи од програмског захтјева `300-1` и да заједно са њим се налази груписан унутар захтјева `30-A`. Синтаксна провјера, ако и имплементација провјере конвенције је већ имплементирана у постојећи програмски језик спецификације програмских захтјева.

Параметар `setup` представља процес прије верификације, одговара функцији `preVerify()`. И обрађује се као изјава ограниченог природног језика. На примјер, изјаву:

```
folder ~/appName does not exist
```

систем ће превести у команду:

```
rm ~/appName
```

Параметар `function` је *чиста* команда извршива у оперативном систему. Посљедња два параметра се користе у сврху дефинисања и израчунавања резултата верификације. Па се тако параметар `outcome`, парсиран као природни језик, преводи у скуп команди које ће у коначници резултирати логичким `TRUE` или `FALSE`, док параметар `output` чита и парсира стандардни излаз, излаз грешке или фајл (ово се додатно дефинише помоћу додатног параметра `type`). Резултат верификације је логичко И између свих извршавања парова `outcome` и `output`. Предусловљање није кориштено, односно подешено је да увијек враћа логичко `TRUE`. Што се тиче тежинског фактора, кориштен је уобичајени скуп. Исто важи и за резултате верификације и функцију израчунавања укупног резултата верификације.

9.5.4 Верификација генерисаних података

Посљедња имплементација програмског језика уз помоћ Окружења не користи се за верификацију програмских захтјева него за верификацију вјештачки генерисаних података. Вјештачки подаци могу бити синтетички или полусинтетички. Синтетички подаци су информације које су произведене вјештачки, а не генерисане из догађаја у стварном свијету [142]. Вјештачки генерисани подаци имају примјену у различитим доменима. Углавном се успостави неки модел података на основу којег се генеришу подаци. Производња вјештачких података углавном захтјева скенирање и/или израчунавање на основу одређених референтних података.

Размотримо ову ситуацију: Потребно је учитати огромну количину вјештачких генерисаних података у базу података. Ово је чест случај уколико је потребно тестирати системе, вршити визуализацију, креирати комплексне математичке моделе, подучавати вјештачку интелигенцију или једноставно, штитити осјетљиве податке. Уколико база података има интегритет над подацима, што је веома чест случај, она садржи индексна, референтна као и остала ограничења. Ова ограничења ће учинити да само учитавање података у базу одузме доста времена, јер се сваки од записа мора провјерити да ли нарушава неко од постојећих ограничења базе података. Како би се ово убрзало најчешће су ова ограничења укинута само током овог учитавања, да би се послије поново укључила.

Цијели процес ће бити успјешан ако приликом учитавања вјештачких података буду испоштована сва ограничења. Међутим, уколико то није случај, изгубљено је много времена, без икаквог резултата, уз трошкове:

1. производње вјештачких података
2. учитавања података у базу података
3. активирање ограничења базе података

На само вријеме производње вјештачких података се не може утицати, али на изгубљено вријеме поновног учитавања података и активирања ограничења је могуће утицати. уколико генерисане податке провјеримо прије учитавања. Тако нећемо губити вријеме на неуспјешно учитавање генерисаних података.

```
<group id="ScheduleTable_1" >
     <depends ref="CfgTable_1"/>
     <requirement id="3029553" severity="major" >
          <depends ref="3029872"/>
          <depends ref="3029873"/>
          <assertSizeRangeRule
              minRows="0"
              maxRows="64 * lcm(CfgTable.scheduleCount)"
              variable="ScheduleTable" />
     </requirement>
...
</group>
```

Исјечак изворног кода програмског језика који је имплементиран на основу Окружења, је дат изнад. У питању је примјер кода за верификацију вјештачки генерисаних података. Релација груписања је једноставно дефинисана тиме што програмски захтјев је XML таг који се налази унутар другом XML тага – тага групе.

Оваква репрезентација онемогућује појави цикличне релације груписања. Са друге стране, релација зависности је имплементирана помоћу `depends` тага, са атрибутом `ref` који представља идентификатор захтјева од којег дати захтјев зависи. Овај таг се може појавити и на нивоу групе и на нивоу програмског захтјева. Тежински фактор је дефинисан атрибутом `severity` програмског захтјева. Дефинисано је 27 различитих тагова провјере (`assert`) који на неки начин провјеравају одређене податке. Унутар тага програмског захтјева, могуће је имати било који број тагова провјере, итерације (`iterate`) и вишеструког услова (`case`). Тагови итерације и вишеструког услова морају имати барем један од три типа тага (`assert`, `iterate`, `case`).

У исјечку кода изнад, дат је примјер провјере која користи додатно имплементирану функцију `lcm`. Функција `lcm` проналази најмањи заједнички садржилац низа бројева. Оно што се не може видјети у примјеру јесте да сваки од основна три типа тага може бити условљен са додатним тагом предусловљавања (*prerequirement*).

10 Резултати потврде концепта

На основу имплементација из претходног поглавља, јасно је да је могуће креирати VSL који покрива све потребне аспетке IEEE стандарда и препорука и за спецификацију и за верификацију функционалних програмских захтјева. Крајњи циљ креирања Окружења – **потврда концепта је постигнута** на начин да се неки аспекти програмског језика имплементирају, уз могућност редефинисања, а да се други аспекти програмског језика остављају на обавезну имплементацију приликом креирања специфичности VSL. Осим овога, на свакој од четири имплементације постоје додатни резултати.

10.1 Програмски језик за тестирање у V-моделу животног циклуса развоја софтвера

У овом случају развијен је исти програмски језик за тестирање на два различита начина: Првобитно је развијен искључиво коришћењем програмског језика опште намјене, а затим је развијен помоћу Окружења уз додатно коришћење програмског језика опште намјене.

Захваљујући поновном развоју коришћењем Окружења од стране инжињера истог нивоа експертизе, било је могуће измјерити број линија који је био потребан да се напише исти програмски језик за тестирање, како у тестирању, тако и у продукцији. Такође, мјерено је и вријеме потрошено за развој новог програмског језика. Потребно је узети у обзир да је приликом програмирања језика помоћу Окружења било потребно додати око 100 линија које су специфичне за усклађивање са Окружењем. Табела 3 приказује резултате ових мјерења.

Табела 3 - Поређење развоја истог програмског језика са и без Окружења

Мјерење	**Традиционални приступ**	**Приступ са Окружењем**
Број линија изворног кода (LoC)	4000	1500
Однос LoC за тестирање и LoC продукције	5:3	3:2
Вријеме (инжињер дан)	150	85

Вријеме приказано у табели је измјерено помоћу система за праћење развоја пројеката (*Jira Tracking System*) који је коришћен у оба случаја развоја. Измјерени број линија изворног кода представља број линија написан у општем програмском језику како би се добио нови програмски језик за тестирање. Очигледно је да је било потребно написати мање линија кода и потрошити мање времена како би се развио нови језик, јер су неке функционалности већ развијене и тестиране унутар Окружења. Разлика у односу линија кода у тестирању и у производњи показује и да је мања потреба за тестирањем када се користи Окружење, чиме смо смањили и могућност грешке.

Комплексност је остала иста, а субјективни је утисак да је комплексност средњег нивоа. Идентификатори који су коришћени су јединствени и у случају постављања истих вриједности, интерпретер језика је враћао грешку. **Ниво озбиљности** функционише, што је могуће видјети подешавањем нивоа на 0, што је довело до тога да укупно тестирање прође иако има тестних случаја који су пали (ниво озбиљности 0 дозвољава да тест падне а да резултат цијелог процеса не буде негативан). **Груписање** је, као што је већ напоменуто, омогућено само на једном нивоу (`test_suite` садржи `test_case`) што је довољно када је у питању тестирање, али не би било довољно када је у питању верификација. **Предуслови** нису дефинисани овим рјешењем. **Зависност** између појединих тестних случајева постоји и некоректно коришћење (унос зависности од непостојећег тестног случаја или стварање **цикличности**) ће изазвати грешку. Изворни код писан у овом програмском језику, као и резултат тестирања могуће је учитати у систем за праћење развоја софтвера (*Windchill PLM Software*). Овај софтвер је у могућности да прати програмски захтјев од његове појаве у документима програмских захтјева, преко његове имполементације па све до извршавања тестова. Учитавањем тестних случајева и резултата тестирања и даље је омогућена **сљедивост**. Међутим, како ова веза између изворног кода и поменутог софтвера није двосмјерна (постоји учитавање, али не постоји креирање изворног кода од садржаја документа) није могуће пропагирати измјене које се десе у систему за праћење развоја софтвера у изворни код за тестирање.

10.2 Побољшање постојећег верификационог процеса

У овом случају, фокус није био на ефикасности процеса развоја, него на робусности самог рјешења и самих перформанси цијелог процеса. У случају да се неки функционални програмски захтјев треба промијенити, додати или избацити, у случају оригиналног верификационог процеса, потребно је додатно вријеме како би се ручно промијенио редослијед програмских захтјева у процесу верификације. Побољшано рјешење не треба додатно вријеме за сортирање, јер се сортирање обавља аутоматски унутар самог процеса.

Табела 4 приказује измјерене уштеде времена побољшаног процеса верификације у односу на оригиналну верзију. Мјерења су рађена над три различита скупа програмских захтјева, различите величине. У табели су наведене уштеде времена по различитим врстама промјена програмских захтјева. Са повећањем скупа програмских захтјева, повећава се и уштеда у времену, без обзира на врсту промјене.

Табела 4 - Уштеда времена за различите врсте промјена и различите величине скупа захтјева

Врста промјене	**Величина верификације у броју програмских захтјева**		
	50	130	290
	Просјечна уштеда времена у сатима		
Ажурирање програмског захтјева	8	20	42
Брисање програмског захтјева	7	21	43
Додавање програмског захтјева	8	17	31

Када је у питању само извршавање верификације, и ту је примјетно убрзање. Табела 5 приказује разлике у просјечном времену извршења цијелог процеса верификације над различитим скуповима програмских захтјева, код оригиналног и побољшаног рјешења. Када се пореди вријеме извршења оригиналне и повољшане верзије верификације уочено је убрзање без обзира на величину скупа програмских

захтјева. Треба напоменути, да у ово вријеме оригиналног рјешења није урачунато вријеме из табеле Табела 4, вријеме потребно за ручно сортирање програмских захтјева. Са друге стране у побољшаном рјешењу, аутоматско сортирање програмских захтјева је урачунату у вријеме извршења.

Табела 5 - Просјечно вријеме извршења верификације различите величине скупова програмских захтјева

Величина скупа програмских захтјева	Просјечно вријеме извршења [ms]	
	Оригинално рјешење	Побољшано рјешење
50	14855	8140
130	31568	22986
290	78280	53981

Разлози за оволиким побољшањем највјероватније и леже у томе што се захваљујући везама зависности, као и предусловима, неки се програмски захтјеви ни не извршавају. Сваки пут када се неки програмски захтјев не изврши, очигледно је да се уштеди вријеме, а у коначници, укупно вријеме верификације је мање, иако су програмски захтјеви аутоматски сортирани.

Као и у претходном случају, већину метрика је могуће доказати да су имплементиране, на основу тога што ће погрешно коришћење узроковати појаву грешака при извршењу: јединствени **идентификатори** (у случају нарушавања јединствености, систем јавља грешку) **зависност** (у случају непостојећих идентификатора или у случају **цикличности**, систем јавља грешку). **Ниво озбиљности** је имплементиран као декоратор, док су **предуслови** имплементирати као `if` искази унутар појединих функција. Ови искази су могући само над основним скупом глобалних података, што значи да у случају разноликости података који условљавају верификацију програмских захтјева велика количина података мора бити доступна током цијелог извршавања верификације. **Груписање** је остварено помоћу припадности функције модулу или пакету. Могуће је вишеструко груписање, али је примјећено да велики нивои груписања, заједно са већим бројем глобалних промјењивих значајно успоравају верификацију што доводи **до изразито лоших**

перформанси при верификацији. **Комплексност** је висока јер подразумјева познавање програмског језика Пајтон.

10.3 Проширење постојећег програмског језика за спецификацију програмских захтјева

За проширење постојећег програмског језика за спецификацију је потрошено 80 инжињер дана уз 1500 линија кода, од стране програмера који је већ упознат са радом Окружења. Основни језик за спецификацију програмских захтјева је развијен у року од 200 инжињер дана, за које је написано 3200 линија кода. За још 40% од већ утрошеног времена за развијање спецификационог програмског језика добијен је и верификациони језик.

Ипак, указано је на проблеме у ажурирању саме спецификације и дијела захтјева специфичног за верификацију. **Груписање** је имплементирано само дјелимично, односно постоји само један ниво груписања. **Остале особине**, осим **предуслова** су имплементиране, а **комплексност** је средњег нивоа пошто корисници осим YAML програмског језика (који је додуше прилично интуитиван) морају научити начин на који се дефинишу поједини параметри.

10.4 Верификација генерисаних података

Резултати показују да верификација генерисаних података чини 15% од укупног времена потребног за генерисање и учитавање података. То значи да верификација података за уобичајене сврхе, као што је учитавање података за бенчмаркинг можда и неће бити корисна. Ако систем који користи генерисане податке може да ради и без *лоших* података који су погрешно генерисани онда можда и нема смисла радити верификацију, али ако је случај да систем за који се генеришу подаци може да ради само над комплетним и релевантним скупом података, онда верификација има смисла.

Груписање је ограничено једним нивоом, а сљедивост није било могуће провјерити јер систем није коришћен у процесу развоја софтвера. Остале особине су имплементиране. Такође, чињеница да постоји 27 могућих начина провјере/верификовања оставља могућност да неке ситуације неће моћи да се провјере. У таквим случајевима се прибјегава имплементацији функција што потенцијално

повећава комплексност коришћења. Субјективни осјећај је да је комплексност средњег нивоа.

10.5. Резултати задовољења метрике над програмским језицима подржаним Окружењем

У табели Табела 6 дат је упоредни преглед резултата особина за програмске језике који су изграђени на основу Окружења. Коришћене су постојеће оцјене за особине и функционалности: подржава (П), дјелимично подржава (Д) и не подржава (Н), као и постојеће оцјене за комплексност: висока, средња и ниска.

Табела 6 - упоредна табела за програмске језике подржане Окружењем

DSL	јединственост	Ниво озбиљности	следивост	зависност	Провјера цикличности	груписање	предуслови	комплексност
Тестирање у V-моделу	П	П	П[13]	П	П	Д	Н	средња
Интерни Пајтон VSL[14]	П	П	П	П	П	Д	Д	висока
Проширени језик за спецификацију прогрмских захтјева[15]	П	П	П	П	П	Д	Н	средња
Верификација генерисаних података	П	П	Н	П	П	Д	П	средња

Када је комплексност у питању, очигледно је да чињеница да је програмски језик изграђен над алатом који поједностављује његово креирање не значи да ће и тај алат бити једноставнији за коришћење.

[13] Имплементација следивости је интерна и показује мане при ажурирању

[14] Груписање и предуслови могу узроковати лоше перформансе

[15] Ажурирање дијела спецификације не ажурира верификациони дио захтјева

Од осталих особина и функционалности, може се рећи да је могуће имплементирати сваку од њих, међутим имплементација **предуслова** и **груписања** на више од једног нивоа смањује перформансе при верификацији. Такође и **сљедивост** изван самог програмског језика је упитна и очекује се да постоји додатна двосмјерна интеграција са системима који користе дати програмски језик. На крају, осим двосмјерне интеграције која би обезбиједила ажурирање, потребно је и додатно интерно ажурирање, као што је случај са проширеним спецификационим програмским језиком.

11 Закључак

Процес спецификације, валидације и верификације функционалних програмских захтјева, је мултидисциплинарни процес који укључује и области као што су психологија, социологија и антропологија. Са друге стране, потребно је помирити и усагласити утицаје из ових области са софтверским инжењерингом.

Досадашња истраживања указују да се комуникација софтверског инжењеринга са експертима из других домена живота, савршено одвија помоћу програмског језика који је специфичан за дати домен. У овом случају појмови и понашања из других области постају дио програмског језика и обезбјеђују мост преко којег идеје и ставови могу да се артикулишу и у верификацију функционалног програмског захтјева.

Такође постоје и истраживања, стандарди и најбоље праксе који указују на особине и функционалности које мора посједовати спецификација али и верификацији функционалних програмских захтјева. Ово истраживање је показало да алати који постоје за верификацију функционалних програмских захтјева из разних разлога, нису у потпуности усклађени са датим стандардима и препорукама. Такође, дати алати су неприкладни за експерте изван области софтверског инжењеринга.

Како би се превазишли постојећи проблеми, као што су отежана комуникација са доменским експертима на пољу спецификације и верификације програмских захтјева, те недостатак стандардних и препоручених функционалности и особина, дисертација је предложила и истражила могућности развоја верификационог DSL (доменски специфичног програмског језика) који имплементира све потребне стандарде и препоруке. У циљу потврде концепта, развијено је Окружење које представља оквир и шаблон за развој произвољног верификационог програмског језика.

Четири примјера верификационог програмског језика која су развијена на основу датог Окружења, прате стандарде и препоруке доказују да је могуће развити програмски језик за верификацију функционалних програмских захтјева са датим особинама и функционалностима. Остварени су и додатни бенефити у области робусности, перформанси и уштеде у времену, који су и очекивани у случајевима коришћења доменски специфичних програмских језика. Ипак уочени су недостаци при

пропагирању сљедивости изван самог програмског језика, као и пад перформанси извршавања приликом комплекснијег груписања и већег скупа података потребних за предуслове. Будућа истраживања се могу фокусирати на интеграцију Окружења и програмских језика са другим системима на којим почива развој софтвера, као и на ефикаснију имплементацију груписања и предуслова.

Бенефити у перформансама и робусности при развоју програмског језика на основу Окружења су уочљиви само приликом поређења са другим програмским језицима. Истраживање се није бавило поређењем верификационог програмског језика са верификационим алатима у овом смислу, што би могло бити основа за будућа истраживања. Ово је још битније када се има на уму да неки од алата имају разумна образложења зашто неке од препоручених особина и функционалности нису имплементиране.

Такође, сва четири примјера су у основи имплементирана као формални програмски језик, што је ипак мање разумљив облик програмског језика за експерте из домена. Један од развијених програмских језика користи атрибут за додатно објашњење програмског захтјева (прилагођено), које није аутоматски ажурирано у случају ажурирања цијелог програмског захтјева. Будућа истраживања би требала да имплементирају природни програмски језик над датим Окружењем, а затим истраже могуће предности.

Истраживање је фокусирано на верификацију тестирањем, па је цијела једна грана верификације заснована на анализи модела изостављена. Овим је онемогућено кориштење датог приступа верификацији система који су компликовани и скупи за верификацију тестирањем. Будућа истраживања би могла проширити фокус и на развијање верификационог програмског језика који ће бити способан верификовати анализирањем модела.

12 Литература

[1] G. L. Steele Jr, “Growing a language,” presented at the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1998.

[2] S. Popic, N. Teslic, and M. Z. Bjelica, “Simple Framework for Efficient Development of the Functional Requirement Verification-specific Language,” *Advances in Electrical and Computer Engineering*, vol. 21, no. 3, pp. 11–20, 2021, doi: 10.4316/AECE.2021.03002.

[3] “IEEE Standard for System and Software Verification and Validation,” *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)*, pp. 1–223, 2012, doi: 10.1109/IEEESTD.2012.6204026.

[4] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, “Advancing candidate link generation for requirements tracing: the study of methods,” *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–10, Jan. 2006, doi: 10.1109/TSE.2006.3.

[5] “IEEE Draft Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition),” *IEEE P1490/D1, May 2011*, pp. 1–505, Jun. 2011, doi: 10.1109/IEEESTD.2011.5937011.

[6] B. Nuseibeh and S. Easterbrook, “Requirements engineering: a roadmap,” in *Proceedings of the conference on The future of Software engineering - ICSE ’00*, Limerick, Ireland, 2000, pp. 35–46. doi: 10.1145/336512.336523.

[7] P. Zave, “Classification of research efforts in requirements engineering,” in *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE’95)*, York, UK, 1995, pp. 214–216. doi: 10.1109/ISRE.1995.512563.

[8] V. Ilic, V. Vujanovic, Srdjan Popic, and M. Pap, “Analiza rezultata testova prilikom razvoja kompleksnih elektronskih upravljackih jedinica,” 2016, doi: 10.13140/RG.2.1.4563.4168.

[9] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*, Third edition. Amsterdam ; Boston: Elsevier, MK, Morgan Kaufmann is an imprint of Elsevier, 2015.

[10] E. M. Clarke, J. Grumberg Orna, D. Kroening, D. Peled, and H. Veith, *Model checking*, Second edition. Cambridge, Massachusetts: The MIT Press, 2018.

[11] J. M. Wing and M. Vaziri-Farahani, “A case study in model checking software systems,” *Science of Computer Programming*, vol. 28, no. 2–3, pp. 273–299, Apr. 1997, doi: 10.1016/S0167-6423(96)00020-2.

[12] A. Morkevicius and N. Jankevicius, “An approach: SysML-based automated requirements verification,” in *2015 IEEE International Symposium on Systems Engineering (ISSE)*, Rome, Italy, Sep. 2015, pp. 92–97. doi: 10.1109/SysEng.2015.7302739.

[13] J. A. Jones, M. Grechanik, and A. van der Hoek, “Enabling and enhancing collaborations between software development organizations and independent test agencies,” in *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, Vancouver, BC, Canada, 2009, pp. 56–59. doi: 10.1109/CHASE.2009.5071411.

[14] G. Kotonya and I. Sommerville, *Requirements engineering: processes and techniques*. Chichester ; New York: J. Wiley, 1998.

[15] I. Sommerville, *Software engineering*, 6th ed. Harlow, England ; New York: Addison-Wesley, 2000.

[16] R. R. Young, *The requirements engineering handbook*. Boston: Artech House, 2003.

[17] J. M. Fernandes and R. J. Machado, “Requirements Elicitation,” in *Requirements in Engineering Projects*, Cham: Springer International Publishing, 2016, pp. 85–117. doi: 10.1007/978-3-319-18597-2_5.

[18] A. Durán Toro, A. Ruiz Cortés, and M. Toro Bonilla, “An Automated Approach for Verification of Software Requirements,” *JIRA 2001: Jornadas de Ingeniería de Requisitos Aplicada (2001).*, 2001.

[19] F. Fabbrini, M. Fusani, V. Gervasi, S. Gnesi, and S. Ruggieri, “Achieving quality in natural language requirements,” 1998. Accessed: Mar. 07, 2023. [Online]. Available: https://www.semanticscholar.org/paper/Achieving-quality-in-natural-language-requirements-Fabbrini-Fusani/6e202981be5ab40dca7b02f5f11ba36b0bc589ff

[20] “ISO/IEC/IEEE International Standard - Systems and software engineering -- Life cycle processes -- Requirements engineering,” IEEE. doi: 10.1109/IEEESTD.2018.8559686.

[21] I. J. Hayes and C. B. Jones, “Specifications are not (necessarily) executable,” *Softw. Eng. J. UK*, vol. 4, no. 6, p. 330, 1989, doi: 10.1049/sej.1989.0045.

[22] Chia-Chu Chiang, “TUG: An Executable Specification Language,” in *5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse (ICIS-COMSAR’06)*, Honolulu, HI, USA, 2006, pp. 180–186. doi: 10.1109/ICIS-COMSAR.2006.85.

[23] A. U. Frank and D. Medak, “Executable axiomatic specification using functional language-case study: Base ontology for a spatio-temporal database,” *Institute for Geoinformation, TU Vienna, Vienna*, 1997.

[24] B. Meyer, “On Formalism in Specifications,” *IEEE Softw.*, vol. 2, no. 1, pp. 6–26, Jan. 1985, doi: 10.1109/MS.1985.229776.

[25] J. F. Sequeda, “Taxonomy of verification and validation of software requirement and specifications,” 2007.

[26] R. Sonbol, G. Rebdawi, and N. Ghneim, “Towards a Semantic Representation for Functional Software Requirements,” in *2020 IEEE Seventh International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, Zurich, Switzerland, Sep. 2020, pp. 1–8. doi: 10.1109/AIRE51212.2020.00007.

[27] K. Pohl, *Requirements engineering: fundamentals, principles, and techniques*. Heidelberg ; New York: Springer, 2010.

[28] N. A. Moketar, M. Kamalrudin, S. Sidek, M. Robinson, and J. Grundy, “An automated collaborative requirements engineering tool for better validation of requirements,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Singapore Singapore, Aug. 2016, pp. 864–869. doi: 10.1145/2970276.2970295.

[29] L. Zhao *et al.*, “Natural Language Processing for Requirements Engineering: A Systematic Mapping Study,” *ACM Comput. Surv.*, vol. 54, no. 3, pp. 1–41, Apr. 2022, doi: 10.1145/3444689.

[30] M. N. Velev and P. Gao, “Method for Formal Verification of Soft-Error Tolerance Mechanisms in Pipelined Microprocessors,” in *Formal Methods and Software Engineering*, vol. 6447, J. S. Dong and H. Zhu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 355–370. doi: 10.1007/978-3-642-16901-4_24.

[31] M. Völter, Ed., *DSL engineering: designing, implementing and using domain-specific languages*. S.l.: CreateSpace Independent Publishing Platform, 2013.

[32] J.-P. Tolvanen and S. Kelly, “Domain-Specific Modeling Languages for Embedded System Development,” presented at the ESWeek Workshop, 2012, p. 11.

[33] L. Lengyel *et al.*, “Quality Assured Model-Driven Requirements Engineering and Software Development,” *The Computer Journal*, vol. 58, no. 11, pp. 3171–3186, Nov. 2015, doi: 10.1093/comjnl/bxv051.

[34] D. Ghosh, *DSLs in action*. Greenwich, Conn: Manning, 2011.

[35] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[36] N. E. Fuchs, K. Kaljurand, and T. Kuhn, “Attempto Controlled English for Knowledge Representation,” in *Reasoning Web*, vol. 5224, C. Baroglio, P. A. Bonatti, J. Małuszyński, M. Marchiori, A. Polleres, and S. Schaffert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 104–124. doi: 10.1007/978-3-540-85658-0_3.

[37] P. Clark, W. R. Murray, P. Harrison, and J. Thompson, “Naturalness vs. Predictability: A Key Debate in Controlled Languages,” in *Controlled Natural Language*, vol. 5972, N. E. Fuchs, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 65–81. doi: 10.1007/978-3-642-14418-9_5.

[38] D. Amyot, A. A. Anda, M. Baslyman, L. Lessard, and J.-M. Bruel, “Towards Improved Requirements Engineering with SysML and the User Requirements Notation,” in *2016 IEEE 24th International Requirements Engineering Conference (RE)*, Beijing, China, Sep. 2016, pp. 329–334. doi: 10.1109/RE.2016.58.

[39] D. de Almeida Ferreira and A. R. da Silva, “RSLingo: An information extraction approach toward formal requirements specifications,” in *2012 Second IEEE International Workshop on Model-Driven Requirements Engineering (MoDRE)*, Chicago, IL, USA, Sep. 2012, pp. 39–48. doi: 10.1109/MoDRE.2012.6360073.

[40] I. ITU-T, “User requirements notation (URN)–language definition,” 2018.

[41] A. A. Efremov and K. I. Gaydamaka, “Incose guide for writing requirements. Translation experience, adaptation perspectives,” presented at the Proceedings of the CEUR Workshop Proceedings, Como, Italy, 2019, pp. 9–11.

[42] J. Holtmann, J. Meyer, and M. von Detten, “Automatic Validation and Correction of Formalized, Textual Requirements,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, Berlin, Germany, Mar. 2011, pp. 486–495. doi: 10.1109/ICSTW.2011.17.

[43] “ISO/IEC/IEEE International Standard - Systems and software engineering -- Life cycle processes -- Risk management,” IEEE. doi: 10.1109/IEEESTD.2021.9325968.

[44] W. D. Yu, “Verifying software requirements: a requirement tracing methodology and its software tool-RADIX,” *IEEE J. Select. Areas Commun.*, vol. 12, no. 2, pp. 234–240, Feb. 1994, doi: 10.1109/49.272872.

[45] “ISO/IEC/IEEE International Standard - Systems and software engineering--Systems and software assurance --Part 1:Concepts and vocabulary,” IEEE. doi: 10.1109/IEEESTD.2019.8657410.

[46] U. D. Ferrell, “Mindful Application of Standards for Avionics - An Intentional, Systematic, and Measurable Transformation,” in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, London, Sep. 2018, pp. 1–8. doi: 10.1109/DASC.2018.8569883.

[47] “IEEE Standard Adoption of ISO/IEC 15026-3 -- Systems and Software Engineering -- Systems and Software Assurance -- Part 3: System Integrity Levels,” IEEE. doi: 10.1109/IEEESTD.2013.6557405.

[48] “IEEE Standard for System, Software, and Hardware Verification and Validation,” IEEE. doi: 10.1109/IEEESTD.2017.8055462.

[49] O. C. Z. Gotel and C. W. Finkelstein, “An analysis of the requirements traceability problem,” in *Proceedings of IEEE International Conference on Requirements Engineering*, Colorado Springs, CO, USA, 1994, pp. 94–101. doi: 10.1109/ICRE.1994.292398.

[50] P. Rempel and P. Mader, “A quality model for the systematic assessment of requirements traceability,” in *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, Ottawa, ON, Canada, Aug. 2015, pp. 176–185. doi: 10.1109/RE.2015.7320420.

[51] C. M. Holloway, “Towards understanding the DO-178C / ED-12C assurance case,” in *7th IET International Conference on System Safety, incorporating the Cyber Security Conference 2012*, Edinburgh, UK, 2012, pp. 14–14. doi: 10.1049/cp.2012.1499.

[52] S. Kan and Z. Huang, “Detecting safety-related components in statecharts through traceability and model slicing: Detecting Safety-related Components by Traceability and Model Slicing,” *Softw Pract Exper*, vol. 48, no. 3, pp. 428–448, Mar. 2018, doi: 10.1002/spe.2526.

[53] J. Rilling, P. Charland, and R. Witte, “Traceability in software engineering–past, present and future,” 2007.

[54] R. Torkar, T. Gorschek, R. Feldt, M. Svahnberg, U. A. Raja, and K. Kamran, “REQUIREMENTS TRACEABILITY: A SYSTEMATIC REVIEW AND INDUSTRY CASE STUDY,” *Int. J. Soft. Eng. Knowl. Eng.*, vol. 22, no. 03, pp. 385–433, May 2012, doi: 10.1142/S021819401250009X.

[55] “IEEE Recommended Practice for Software Requirements Specifications,” IEEE. doi: 10.1109/IEEESTD.1998.88286.

[56] B. Ramesh and M. Edwards, “Issues in the development of a requirements traceability model,” in *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA, USA, 1992, pp. 256–259. doi: 10.1109/ISRE.1993.324849.

[57] S. Popic, M. Vulic, and I. Velikic, “Interface checks of the automotive embedded software components,” in *2017 IEEE 7th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, Berlin, Sep. 2017, pp. 163–167. doi: 10.1109/ICCE-Berlin.2017.8210618.

[58] M. Mauritz and M. Roidl, “From Requirements to Executable Rules: An Ensemble of Domain-Specific Languages for Programming Cyber-Physical Systems in Warehouse Logistics,” in *Leveraging Applications of Formal Methods, Verification and*

Validation, vol. 13036, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2021, pp. 162–177. doi: 10.1007/978-3-030-89159-6_11.

[59] J.-P. Tolvanen and S. Kelly, "How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases," in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, Paris France, Sep. 2019, pp. 155–163. doi: 10.1145/3336294.3336316.

[60] R. S. Pressman and B. R. Maxim, *Software engineering: a practitioner's approach*, Nineth edition. New York, NY: McGraw-Hill Education, 2020.

[61] V. S. Alagar and K. Periyasamy, *Specification of software systems*, 2nd ed. New York: Springer, 2011.

[62] M. Fowler, "A Pedagogical Framework for Domain-Specific Languages," *IEEE Softw.*, vol. 26, no. 4, pp. 13–14, Jul. 2009, doi: 10.1109/MS.2009.85.

[63] D. M. Weiss and C. T. R. Lai, *Software product-line engineering: a family-based software development process*. Reading, Mass: Addison-Wesley, 1999.

[64] A. Raunio, "Experiences on using DSM at EADS, DSM seminar presentation," *Proceedings of Finnish Computing Science Days, University of Jyväskylä, TU-25*, 2007.

[65] R. B. Kieburtz *et al.*, "A software engineering experiment in software component generation," in *Proceedings of IEEE 18th International Conference on Software Engineering*, Berlin, Germany, 1996, pp. 542–552. doi: 10.1109/ICSE.1996.493448.

[66] S. Ramzan, I. S. Bajwa, and B. Ramzan, "A NATURAL LANGUAGE METAMODEL FOR GENERATING CONTROLLED NATURAL LANGUAGE BASED REQUIREMENTS.," *Science International*, vol. 28, no. 3, 2016.

[67] T. Vajk, R. Kereskényi, T. Levendovszky, and Á. Lédeczi, "Raising the abstraction of domain-specific model translator development," in *2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2009, pp. 31–37.

[68] A. R. da Silva, A. C. R. Paiva, and V. E. R. da Silva, "A Test Specification Language for Information Systems Based on Data Entities, Use Cases and State Machines," in *Model-Driven Engineering and Software Development*, vol. 991, S. Hammoudi, L. F.

Pires, and B. Selic, Eds. Cham: Springer International Publishing, 2019, pp. 455–474. doi: 10.1007/978-3-030-11030-7_20.

[69] M. Ben-Ari, *Principles of the Spin model checker*. London: Springer, 2008.

[70] N. E. Fuchs and R. Schwitter, “Attempto Controlled English (ACE),” 1996, doi: 10.48550/ARXIV.CMP-LG/9603003.

[71] A. Holt, “Formal verification with natural language specifications: guidelines, experiments and lessons so far,” *South African Computer Journal*, pp. 253–257, 1999.

[72] D. Firesmith, “Generating Complete, Unambiguous, and Verifiable Requirements from Stories, Scenarios, and Use Cases.,” *J. Object Technol.*, vol. 3, no. 10, pp. 27–40, 2004.

[73] D. Amyot, L. Logrippo, and M. Weiss, “Generation of test purposes from Use Case Maps,” *Computer Networks*, vol. 49, no. 5, pp. 643–660, 2005.

[74] L. Charfi, *Formal modeling and test generation automation with Use Case Maps and LOTOS*. University of Ottawa (Canada), 2001.

[75] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Reading, Mass: Addison-Wesley, 2000.

[76] A. Pourshahid *et al.*, “Toward an Integrated User Requirements Notation Framework and Tool forBusiness Process Management,” in *2008 International MCETECH Conference on e-Technologies (mcetech 2008)*, Montreal, QC, Canada, Jan. 2008, pp. 3–15. doi: 10.1109/MCETECH.2008.30.

[77] D. Amyot and G. Mussbacher, “User Requirements Notation: The First Ten Years, The Next Ten Years (Invited Paper),” *JSW*, vol. 6, no. 5, pp. 747–768, May 2011, doi: 10.4304/jsw.6.5.747-768.

[78] N. Genon, D. Amyot, and P. Heymans, “Analysing the Cognitive Effectiveness of the UCM Visual Notation,” in *System Analysis and Modeling: About Models*, vol. 6598, F. A. Kraemer and P. Herrmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 221–240. doi: 10.1007/978-3-642-21652-7_14.

[79] D. L. Moody, P. Heymans, and R. Matulevičius, “Visual syntax does matter: improving the cognitive effectiveness of the i* visual notation,” *Requirements Eng*, vol. 15, no. 2, pp. 141–175, Jun. 2010, doi: 10.1007/s00766-010-0100-1.

[80] J. Cheng, L. Yang, Y.-J. Kuai, and D.-F. Zhang, “Non-Deterministic Feature Interaction Filtering Method Based on Scenarios with Use Case Map,” *Journal of Hunan University*, vol. 32, no. 2, pp. 104–109, Apr. 2005.

[81] R. Zhang and X.-X. Liu, “Feature Interaction Filtering Method Based on URN (基于 URN 的特征冲突过滤方法),” *Computer Engineering (计 算 机 工 程)*, vol. 35, no. 21, pp. 45–47, Nov. 2009.

[82] A. R. da Silva, “Linguistic Patterns and Linguistic Styles for Requirements Specification (I): An Application Case with the Rigorous RSL/Business-Level Language,” in *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, Irsee Germany, Jul. 2017, pp. 1–27. doi: 10.1145/3147704.3147728.

[83] L. Goncalves and A. Rodrigues da Silva, “A Catalogue of Reusable Security Concerns: Focus on Privacy Threats,” in *2018 IEEE 20th Conference on Business Informatics (CBI)*, Vienna, Jul. 2018, pp. 52–61. doi: 10.1109/CBI.2018.10046.

[84] A. Rodrigues da Silva and J. C. Fernandes, “Variability Specification and Resolution of Textual Requirements:,” in *Proceedings of the 20th International Conference on Enterprise Information Systems*, Funchal, Madeira, Portugal, 2018, pp. 157–168. doi: 10.5220/0006810801570168.

[85] D. de A. Ferreira and A. R. da Silva, “Formally Specifying Requirements with RSL-IL,” in *2012 Eighth International Conference on the Quality of Information and Communications Technology*, Lisbon, TBD, Portugal, Sep. 2012, pp. 217–220. doi: 10.1109/QUATIC.2012.30.

[86] A. R. da Silva, “Quality of requirements specifications: a preliminary overview of an automatic validation approach,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, Gyeongju Republic of Korea, Mar. 2014, pp. 1021–1022. doi: 10.1145/2554850.2555115.

[87] L. C. Hong and L. T. Ming, “To enable formal verification of semi-formal requirements by using pre-defined template and mapping rules to map to Promela specification to reduce rework,” in *2010 International Symposium on Information Technology*, Kuala Lumpur, Malaysia, Jun. 2010, pp. 1393–1397. doi: 10.1109/ITSIM.2010.5561453.

[88] G. Paun, *Membrane Computing An Introduction*, Softcover reprint of the original 1st ed. 2002. Berlin: Springer Berlin, 2013.

[89] R. Neumann, “Using Promela in a Fully Verified Executable LTL Model Checker,” in *Verified Software: Theories, Tools and Experiments*, vol. 8471, D. Giannakopoulou and D. Kroening, Eds. Cham: Springer International Publishing, 2014, pp. 105–114. doi: 10.1007/978-3-319-12154-3_7.

[90] M. del M. Gallardo, P. Merino, and E. Pimentel, “A generalized semantics of PROMELA for abstract model checking,” *Form. Asp. Comput.*, vol. 16, no. 3, pp. 166–193, Aug. 2004, doi: 10.1007/s00165-004-0040-y.

[91] J. T. Mühlberg and G. Lüttgen, “Blasting Linux Code,” in *Formal Methods: Applications and Technology*, vol. 4346, L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 211–226. doi: 10.1007/978-3-540-70952-7_14.

[92] G. J. Holzmann, “The model checker SPIN,” *IIEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, May 1997, doi: 10.1109/32.588521.

[93] F. Wang *et al.*, “An Approach to Generate the Traceability Between Restricted Natural Language Requirements and AADL Models,” *IEEE Trans. Rel.*, vol. 69, no. 1, pp. 154–173, Mar. 2020, doi: 10.1109/TR.2019.2936072.

[94] P. H. Feiler, B. A. Lewis, and S. Vestal, “The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems,” in *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, Munich, Germany, Oct. 2006, pp. 1206–1211. doi: 10.1109/CACSD-CCA-ISIC.2006.4776814.

[95] Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin, “From AADL to Timed Abstract State Machines: A verified model transformation,” *Journal of Systems and Software*, vol. 93, pp. 42–68, Jul. 2014, doi: 10.1016/j.jss.2014.02.058.

[96] G. Behrmann, A. David, and K. G. Larsen, “A Tutorial on Uppaal,” in *Formal Methods for the Design of Real-Time Systems*, vol. 3185, M. Bernardo and F. Corradini, Eds.

Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–236. doi: 10.1007/978-3-540-30080-9_7.

[97] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional Verification of Architectural Models," in *NASA Formal Methods*, vol. 7226, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 126–140. doi: 10.1007/978-3-642-28891-3_13.

[98] Z. Yang *et al.*, "Exploiting augmented intelligence in the modeling of safety-critical autonomous systems," *Form. Asp. Comput.*, vol. 33, no. 3, pp. 343–384, Jun. 2021, doi: 10.1007/s00165-021-00543-6.

[99] T. Kuhn, "Controlled English for knowledge representation," University of Zurich, 2010. doi: 10.5167/UZH-33191.

[100] S. Robertson and J. Robertson, *Mastering the requirements process: getting requirements right*, 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

[101] R. Schwitter, "Controlled natural languages for knowledge representation," presented at the Coling 2010: Posters, 2010, pp. 1113–1121.

[102] N. E. Fuchs, U. Schwertel, and S. Torge, "Controlled natural language can replace first-order logic," in *14th IEEE International Conference on Automated Software Engineering*, Cocoa Beach, FL, USA, 1999, pp. 295–298. doi: 10.1109/ASE.1999.802325.

[103] N. E. Fuchs, "First-Order Reasoning for Attempto Controlled English," in *Controlled Natural Language*, vol. 7175, M. Rosner and N. E. Fuchs, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 73–94. doi: 10.1007/978-3-642-31175-8_5.

[104] G. Sutcliffe, "The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0," vol. 59, no. 4, pp. 483–502, 2017.

[105] World Wide Web Consortium, "OWL 2 Web Ontology Language Document Overview (Second Edition)," *OWL 2 Web Ontology Language Document Overview (Second Edition)*, Nov. 11, 2012. https://www.w3.org/TR/owl2-overview/ (accessed Sep. 11, 2022).

[106] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A semantic web rule language combining OWL and RuleML," *W3C Member submission*, vol. 21, no. 79, pp. 1–31, 2004.

[107] I. Pratt-Hartmann and A. Third, “More Fragments of Language,” *Notre Dame J. Formal Logic*, vol. 47, no. 2, Apr. 2006, doi: 10.1305/ndjfl/1153858644.

[108] N. E. Fuchs and K. Kaljurand, “Attempto Controlled English meets the challenges of knowledge representation, reasoning, interoperability and user interfaces,” 2006.

[109] L. Rutledge and R. Italiaander, “Toward a Reference Architecture for Traceability in SBVR-based Systems,” presented at the Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21), 2021.

[110] R. Schwitter and N. E. Fuchs, “Attempto controlled English (ACE) a seemingly informal bridgehead in formal territory,” in *Logic Programming: Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, MIT Press, 1996, pp. 536–539.

[111] N. E. Fuchs, “Reasoning in Attempto Controlled English: Non-monotonicity,” in *Controlled Natural Language*, vol. 9767, B. Davis, G. J. Pace, and A. Wyner, Eds. Cham: Springer International Publishing, 2016, pp. 13–24. doi: 10.1007/978-3-319-41498-0_2.

[112] P. Clark, P. Harrison, T. Jenkins, J. A. Thompson, and R. H. Wojcik, “Acquiring and using world knowledge using a restricted subset of English.,” presented at the Flairs conference, 2005, pp. 506–511.

[113] T. Gao, “Controlled Natural Languages and Default Reasoning,” 2019, doi: 10.48550/ARXIV.1905.04422.

[114] H. Wan, B. Grosof, M. Kifer, P. Fodor, and S. Liang, “Logic Programming with Defaults and Argumentation Theories,” in *Logic Programming*, vol. 5649, P. M. Hill and D. S. Warren, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 432–448. doi: 10.1007/978-3-642-02846-5_35.

[115] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, “Automating test automation,” in *2012 34th International Conference on Software Engineering (ICSE)*, Zurich, Jun. 2012, pp. 881–891. doi: 10.1109/ICSE.2012.6227131.

[116] M.-C. De Marneffe, B. MacCartney, and C. D. Manning, “Generating typed dependency parses from phrase structure parses.,” presented at the Lrec, 2006, vol. 6, pp. 449–454.

[117] D. D. Sleator and D. Temperley, “Parsing English with a link grammar,” *arXiv preprint cmp-lg/9508004*, 1995.

[118] “Html unit,” *HtmlUnit*. https://htmlunit.sourceforge.io/

[119] H. Selinium, “Selenium—Web Browser Automation”.

[120] C. Davis *et al.*, *Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource*. Pearson Education, 2009.

[121] S. Thummalapenta *et al.*, “Efficient and change-resilient test automation: An industrial case study,” in *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013, pp. 1002–1011. doi: 10.1109/ICSE.2013.6606650.

[122] A. Sinha, S. M. Sutton, and A. Paradkar, “Text2Test: Automated Inspection of Natural Language Use Cases,” in *2010 Third International Conference on Software Testing, Verification and Validation*, Paris, Apr. 2010, pp. 155–164. doi: 10.1109/ICST.2010.19.

[123] A. Miller, B. Kumar, and A. Singhal, “Photon: A Domain-Specific Language for Testing Converged Applications,” in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Seattle, Washington, USA, 2009, pp. 269–274. doi: 10.1109/COMPSAC.2009.143.

[124] Gatling, “Gatling FrontLine documentation,” *Gatling Corp*. https://gatling.io/docs/gatling/reference/current/

[125] M. Bernardino, A. F. Zorzo, and E. M. Rodrigues, “Canopus: A Domain-Specific Language for Modeling Performance Testing,” in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Chicago, IL, USA, Apr. 2016, pp. 157–167. doi: 10.1109/ICST.2016.13.

[126] S. Pakin, “The Design and Implementation of a Domain-Specific Language for Network Performance Testing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, pp. 1436–1449, Oct. 2007, doi: 10.1109/TPDS.2007.1065.

[127] A. Dwarakanath, D. Era, A. Priyadarshi, N. Dubash, and S. Podder, “Accelerating Test Automation through a Domain Specific Language,” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan, Mar. 2017, pp. 460–467. doi: 10.1109/ICST.2017.52.

[128] N. Eriksson, “Interactive Execution of Non-interactive Test Framework Features: In Collaboration with Ericsson AB,” 2020.

[129] S. Rose, M. Wynne, and A. Hellesoy, “The cucumber for Java book: Behaviour-driven development for testers and developers,” *The Cucumber for Java Book*, pp. 1–338, 2015.

[130] R. Chatley, J. Ayres, and T. White, “LiFT: Driving Development Using a Business-Readable DSL for Web Testing,” in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, Paris, France, Apr. 2010, pp. 460–468. doi: 10.1109/ICSTW.2010.12.

[131] T. Deiß, A. J. Nyberg, S. Schulz, C. Willcock, and R. Teittinen, “Industrial deployment of the TTCN-3 testing technology,” *IEEE software*, vol. 23, no. 4, pp. 48–54, Jul. 2006, doi: 10.1109/MS.2006.108.

[132] J. Borcsok, W. Chaaban, M. Schwarz, H. Sheng, O. Sheleh, and B. Batchuluun, “An automated software verification tool for model-based development of embedded systems with simulink®,” in *2009 XXII International Symposium on Information, Communication and Automation Technologies*, Bosnia, Oct. 2009, pp. 1–6. doi: 10.1109/ICAT.2009.5348445.

[133] H. Soubra, A. Abran, S. Stern, and A. Ramdan-Cherif, “Design of a Functional Size Measurement Procedure for Real-Time Embedded Software Requirements Expressed using the Simulink Model,” in *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, Nara, Japan, Nov. 2011, pp. 76–85. doi: 10.1109/IWSM-MENSURA.2011.52.

[134] Z. Jiang, X. Wu, Z. Dong, and M. Mu, “Optimal Test Case Generation for Simulink Models Using Slicing,” in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Prague, Czech Republic, Jul. 2017, pp. 363–369. doi: 10.1109/QRS-C.2017.67.

[135] M. Wahler, E. Ferranti, R. Steiger, R. Jain, and K. Nagy, “CAST: Automating Software Tests for Embedded Systems,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montreal, QC, Canada, Apr. 2012, pp. 457–466. doi: 10.1109/ICST.2012.126.

[136] M. Sipser, *Introduction to the theory of computation*, Third edition. Australia ; Boston, MA: Cengage Learning, 2013.

[137] ETSI, “Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.” European Telecommunications Standardization Institute. Accessed: Dec. 06, 2022. [Online]. Available: http://www.ttcn-3.org/index.php/downloads/standards

[138] H. Dubois, M.-A. Peraldi-Frati, and F. Lakhal, “A Model for Requirements Traceability in a Heterogeneous Model-Based Design Process: Application to Automotive Embedded Systems,” in *2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, Oxford, United Kingdom, Mar. 2010, pp. 233–242. doi: 10.1109/ICECCS.2010.2.

[139] S. Popic, V. Komadina, R. Arsenovic, and M. Stepanovic, “Implementation of the simple domain-specific language for system testing in V-Model development lifecycle,” in *2020 Zooming Innovation in Consumer Technologies Conference (ZINC)*, Novi Sad, Serbia, May 2020, pp. 290–294. doi: 10.1109/ZINC50678.2020.9161781.

[140] A. Hunt and D. Thomas, *Pragmatic unit testing in Java with JUnit*. The Pragmatic Bookshelf, 2003.

[141] D. de Almeida Ferreira and A. R. da Silva, “RSL-IL: An interlingua for formally documenting requirements,” in *2013 3rd International Workshop on Model-Driven Requirements Engineering (MoDRE)*, Rio de Janeiro, Brazil, Jul. 2013, pp. 40–49. doi: 10.1109/MoDRE.2013.6597262.

[142] S. Popic, B. Pavkovic, I. Velikic, and N. Teslic, “Data generators: a short survey of techniques and use cases with focus on testing,” in *2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin)*, Berlin, Germany, Sep. 2019, pp. 189–194. doi: 10.1109/ICCE-Berlin47944.2019.8966202.

13 Публикације објављене током израде тезе

13.1 Часописи

1. S. Popic, N. Teslic, and M. Z. Bjelica, “Simple Framework for Efficient Development of the Functional Requirement Verification-specific Language,” Advances in Electrical and Computer Engineering, vol. 21, no. 3, pp. 11–20, 2021, doi: 10.4316/AECE.2021.03002.

13.2 Конференције

1. S. Popic, M. Vuleta, P. Cvjetkovic, and B. M. Todorovic, “Secure Topology Detection in Software-Defined Networking with Network Configuration Protocol and Link Layer Discovery Protocol,” in 2020 International Symposium on Industrial Electronics and Applications (INDEL), Banja Luka, Bosnia and Herzegovina, Nov. 2020, pp. 1–5. doi: 10.1109/INDEL50386.2020.9266137.
2. A. Veselinovic, S. Popic, and Z. Lukac, “Smart home system solution with the goal of minimizing water consumption,” in 2020 International Symposium on Industrial Electronics and Applications (INDEL), Banja Luka, Bosnia and Herzegovina, Nov. 2020, pp. 1–5. doi: 10.1109/INDEL50386.2020.9266238.
3. S. Popic, V. Komadina, R. Arsenovic, and M. Stepanovic, “Implementation of the simple domain-specific language for system testing in V-Model development lifecycle,” in 2020 Zooming Innovation in Consumer Technologies Conference (ZINC), Novi Sad, Serbia, May 2020, pp. 290–294. doi: 10.1109/ZINC50678.2020.9161781.
4. S. Popic, B. Majstorovic, M. Vuleta, E. Saric, and B. M. Todorovic, “Efficient Usage of Resources in SDN by Modifying YANG Modules in Linux-based Embedded Systems,” in 2019 27th Telecommunications Forum (TELFOR), Belgrade, Serbia, Nov. 2019, pp. 1–4. doi: 10.1109/TELFOR48224.2019.8971364.
5. S. Popic, T. Krnjajic, S. Doslic, and B. M. Todorovic, “Implementation of NETCONF Client in C++ Programming Language for Software Defined

Networks," in 2019 27th Telecommunications Forum (TELFOR), Belgrade, Serbia, Nov. 2019, pp. 1–4. doi: 10.1109/TELFOR48224.2019.8971071.

6. S. Popic, B. Pavkovic, I. Velikic, and N. Teslic, "Data generators: a short survey of techniques and use cases with focus on testing," in 2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin), Berlin, Germany, Sep. 2019, pp. 189–194. doi: 10.1109/ICCE-Berlin47944.2019.8966202.
7. S. Popic, G. Velikic, H. Jaroslav, Z. Spasic, and M. Vulic, "The Benefits of the Coding Standards Enforcement and it's Influence on the Developers' Coding Behaviour: A Case Study on Two Small Projects," in 2018 26th Telecommunications Forum (TELFOR), Belgrade, Nov. 2018, pp. 420–425. doi: 10.1109/TELFOR.2018.8612149.
8. S. Popic, I. Papp, and D. Đekanovic, "Processing cost in case of message parsing on the smart IoT gateway: Exploring the costs of unifying the message format to Protocol Buffer," in 2017 International Conference on Smart Systems and Technologies (SST), Osijek, Oct. 2017, pp. 169–173. doi: 10.1109/SST.2017.8188690.
9. S. Popic, M. Vulic, and I. Velikic, "Interface checks of the automotive embedded software components," in 2017 IEEE 7th International Conference on Consumer Electronics - Berlin (ICCE-Berlin), Berlin, Sep. 2017, pp. 163–167. doi: 10.1109/ICCE-Berlin.2017.8210618.
10. S. Popic, I. Velikic, and N. Teslic, "Retrieving the useful information from the binary files compiled by C compiler," in 2017 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 2017, pp. 338–339. doi: 10.1109/ICCE.2017.7889345.
11. S. Popic, D. Pezer, B. Mrazovac, and N. Teslic, "Performance evaluation of using Protocol Buffers in the Internet of Things communication," in 2016 International Conference on Smart Systems and Technologies (SST), Osijek, Croatia, Oct. 2016, pp. 261–265. doi: 10.1109/SST.2016.7765670.
12. V. Ilic, V. Vujanovic, Srdjan Popic, and M. Pap, "Analiza rezultata testova prilikom razvoja kompleksnih elektronskih upravljackih jedinica," 2016, doi: 10.13140/RG.2.1.4563.4168.

13. Srdjan Popic, V. Ilic, I. Velikic, and N. Teslic, “Analiza informacija o C kompajleru na osnovu sadržaja binarnih fajlova,” 2016, doi: 10.13140/RG.2.1.2466.2647.

14. M. Grabovica, S. Popic, D. Pezer, and V. Knezevic, “Provided security measures of enabling technologies in Internet of Things (IoT): A survey,” in 2016 Zooming Innovation in Consumer Electronics International Conference (ZINC), Novi Sad, Serbia, Jun. 2016, pp. 28–31. doi: 10.1109/ZINC.2016.7513647.

15. V. Ilic, S. Popic, and M. Kovacic, “Data flow in automated testing of the complex automotive electronic control units,” in 2016 Zooming Innovation in Consumer Electronics International Conference (ZINC), Novi Sad, Serbia, Jun. 2016, pp. 1–3. doi: 10.1109/ZINC.2016.7513639.

БИОГРАФИЈА

Срђан Попић је рођен 4. октобра 1975. године у Травнику. Основну школу је завршио у Новом Травнику, а средњу школу у Новом Саду. Факултет техничких наука у Новом Саду је уписао 1993. године, студијски програм Рачунарство и аутоматика. Основне студије је завршио 1999. године са просјечном оцјеном 8.31 (осам 31/100), одбранивши рад под темом „Једно решење симулационог окружења за тестирање аутоматске управљачке процедуре“, који му се 2015 године признао као мастер рад. Докторске студије је уписао школске 2015/2016. године на истом факултету.

Истраживачки интерес Срђана Попића је усмјерен ка верификацији рачунарских система као и самих програмских захтјева и њиховој међусобној корелацији. Године 2018. је изабран у звање Виши асистент на електротехничком факултету, Универзитета у Бањој Луци и од тада активно учествује у извођењу наставе из предмета: Оперативни системи за рад у реалном времену и Паралално програмирање у реалном времену.

Аутор је једног рада у међународним часописима, аутор или коаутор још 13 радова на међународним конференцијама и 2 рада на националним конференцијама. Од страних језика говори енглески језик.

Овај Образац чини саставни део докторске дисертације, односно докторског уметничког пројекта који се брани на Универзитету у Новом Саду. Попуњен Образац укоричити иза текста докторске дисертације, односно докторског уметничког пројекта.

План третмана података

Назив пројекта/истраживања
Прилог рјешењу ефикасне верификације функционалних захтјева помоћу програмских језика
Назив институције/институција у оквиру којих се спроводи истраживање
а) Факултет техничких наука, Универзитет у Новом Саду **б) Институт РТ-РК, Бањалука**
Назив програма у оквиру ког се реализује истраживање
Истраживање се реализује у оквиру израде докторске дисертације на студијском програму Рачунарство и аутоматика
1. Опис података
1.1 Врста студије **Докторска дисертација** 1.2 Врсте података **а) квантитативни** **б) квалитативни** 1.3. Начин прикупљања података а) анкете, упитници, тестови б) клиничке процене, медицински записи, електронски здравствени записи в) генотипови: навести врсту _____ г) административни подаци: навести врсту _____

д) узорци ткива: навести врсту_____

ђ) снимци, фотографије: навести врсту_____

е) текст, навести врсту _____

ж) мапа, навести врсту _____

з) остало: описати Рачунарске апликације_____

1.3 Формат података, употребљене скале, количина података

1.3.1 Употребљени софтвер и формат датотеке:

а) Excel фајл, датотека __

b) SPSS фајл, датотека _____

c) PDF фајл, датотека _____

d) Текст фајл, датотека _____

e) JPG фајл, датотека _____

f) Остало, датотека _____

1.3.2. Број записа (код квантитативних података)

а) број варијабли **8**_____

б) број мерења (испитаника, процена, снимака и сл.) **Велик број**_____

1.3.3. Поновљена мерења

а) да

б) не

Уколико је одговор да, одговорити на следећа питања:

а) временски размак између поновљених мера је _____

б) варијабле које се више пута мере односе се на _____

в) нове верзије фајлова који садрже поновљена мерења су именоване као _____

Напомене: _____

Да ли формати и софтвер омогућавају дељење и дугорочну валидност података?

а) ***Да***

б) Не

Ако је одговор не, образложити _____

_____

2. Прикупљање података

2.1 Методологија за прикупљање/генерисање података

2.1.1. У оквиру ког истраживачког нацрта су подаци прикупљени?

а) **експеримент**, навести тип **Рачунарске апликације**_____

б) корелационо истраживање, навести тип _____

ц) **анализа текста**, навести тип **Анализа доступне литературе**_____

д) остало, навести шта _____

2.1.2 Навести врсте мерних инструмената или стандарде података специфичних за одређену научну дисциплину (ако постоје).

_____

_____

2.2 Квалитет података и стандарди

2.2.1. Третман недостајућих података

а) Да ли матрица садржи недостајуће податке? Да **Не**

Ако је одговор да, одговорити на следећа питања:

а) Колики је број недостајућих података? _____

б) Да ли се кориснику матрице препоручује замена недостајућих података? Да Не

в) Ако је одговор да, навести сугестије за третман замене недостајућих података

_____

2.2.2. На који начин је контролисан квалитет података? Описати

Квалитет података је контролисан поређењем експерименталних и теоријских података

2.2.3. На који начин је извршена контрола уноса података у матрицу?

Контрола уноса података је изведена на основу експертског знања

3. Третман података и пратећа документација

3.1. Третман и чување података

3.1.1. Подаци ће бити депоновани у **Репозиторијуму докторских дисертација на Универзитету у Новом Саду**.

3.1.2. URL адреса _____

3.1.3. DOI _____

3.1.4. Да ли ће подаци бити у отвореном приступу?

а) ***Да***

б) *Да, али после ембарга који ће трајати до* _____

в) *Не*

Ако је одговор не, навести разлог _____

3.1.5. Подаци неће бити депоновани у репозиторијум, али ће бити чувани.

Образложење

_____

_____

3.2 Метаподаци и документација података

3.2.1. Који стандард за метаподатке ће бити примењен? **Стандард који примењује Репозиторијум докторских дисертација Универзитета у Новом Саду**

3.2.1. Навести метаподатке на основу којих су подаци депоновани у репозиторијум.

_____

_____

Ако је потребно, навести методе које се користе за преузимање података, аналитичке и процедуралне информације, њихово кодирање, детаљне описе варијабли, записа итд.

_____

_____

_____

_____

3.3 Стратегија и стандарди за чување података

3.3.1. До ког периода ће подаци бити чувани у репозиторијуму? _____

3.3.2. Да ли ће подаци бити депоновани под шифром? **Да** Не

3.3.3. Да ли ће шифра бити доступна одређеном кругу истраживача? **Да** Не

3.3.4. Да ли се подаци морају уклонити из отвореног приступа после извесног времена?

Да **Не**

Образложити

_____

_____

4. Безбедност података и заштита поверљивих информација

Овај одељак МОРА бити попуњен ако ваши подаци укључују личне податке који се односе на учеснике у истраживању. За друга истраживања треба такође размотрити заштиту и сигурност података.

4.1 Формални стандарди за сигурност информација/података

Истраживачи који спроводе испитивања с људима морају да се придржавају Закона о заштити података о личности *(https://www.paragraf.rs/propisi/zakon_o_zastiti_podataka_o_licnosti.html)* и одговарајућег институционалног кодекса о академском интегритету.

4.1.2. Да ли је истраживање одобрено од стране етичке комисије? Да **Не**

Ако је одговор Да, навести датум и назив етичке комисије која је одобрила истраживање

_____

4.1.2. Да ли подаци укључују личне податке учесника у истраживању? Да **Не**

Ако је одговор да, наведите на који начин сте осигурали поверљивост и сигурност информација везаних за испитанике:

а) Подаци нису у отвореном приступу

б) Подаци су анонимизирани

ц) Остало, навести шта

_____

_____

5. Доступност података

5.1. Подаци ће бити

*а) **јавно доступни***

б) доступни само уском кругу истраживача у одређеној научној области

ц) затворени

Ако су подаци доступни само уском кругу истраживача, навести под којим условима могу да их користе:

_____

_____

Ако су подаци доступни само уском кругу истраживача, навести на који начин могу приступити подацима:

_____

_____

5.4. Навести лиценцу под којом ће прикупљени подаци бити архивирани.

Ауторство - некомерцијално_____

6. Улоге и одговорност

6.1. Навести име и презиме и мејл адресу власника (аутора) података

Срђан Попић

srdjan.popic@gmail.com

6.2. Навести име и презиме и мејл адресу особе која одржава матрицу с подацима

Срђан Попић

srdjan.popic@gmail.com

6.3. Навести име и презиме и мејл адресу особе која омогућује приступ подацима другим истраживачима

Срђан Попић

srdjan.popic@gmail.com