УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Аутоматизација развоја API-базираних генератора рударењем идиома из изворног кода

Докторска дисертација

Automating the Development of API-Based Generators Using Source Code Idioms Mining

Doctoral dissertation

Ментор:
проф. др Гордана Милосављевић

Кандидат:
Ненад Тодоровић

Нови Сад, 2026

УНИВЕРЗИТЕТ У НОВОМ САДУ **ОБРАЗАЦ – 5а**
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА[1]

Врста рада:	Докторска дисертација
Име и презиме аутора:	Ненад Тодоровић
Ментор (титула, име, презиме, звање, институција):	др Гордана Милосављевић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду
Наслов рада:	Аутоматизација развоја API-базираних генератора рударењем идиома из изворног кода
Језик и писмо рада:	Енглески језик, латиница
Физички опис рада:	Страница: 104 Поглавља: 9 Референци: 120 Табела: 3 Слика: 35 Графикона: 0 Прилога: 0
Научна област:	Електротехничко и рачунарско инжењерство
Ужа научна област (научна дисциплина):	Примењене рачунарске науке и информатика
Кључне речи / предметна одредница:	Софтверско инжењерство вођено моделима, трансформација модела у текст, рударење идиома из кода, API-базирани генератори

[1] Аутор докторске дисертације потписао је и приложио следеће обрасце:
5б – Изјава о ауторству;
5в – Изјава о истоветности штампане и електронске верзије докторског рада и дозвола за објављивање личних података;
5г – Изјава о коришћењу.
Ове Изјаве се чувају у институцији у штампаном и електронском облику и не кориче се са радом.

Резиме:	API-базирани генератори (енгл. *API-Based Generator – ABG*) омогућавају развојном тиму да користи предности софтверског инжењерства вођеног моделима (*Model-Driven Software Engineering – MDSE*) без увођења значајних измена у тренутни развојни процес или архитектуру решења. Међутим, коришћење ABG-ова са API-јем дефинисаним у терминима синтаксе циљног програмског језика може бити захтевно и непрактично. Ручни развој ABG-а тако да подржи концепте вишег нивоа апстракције који се могу наћи у постојећем решењу захтева пуно времена и енергије. Ова дисертација представља нови приступ за полуаутоматски развој API-базираних генератора по имену SAGED (енгл. *Semi-automatic API-based Generators Development*), чији је циљ да убрза креирање ABG-ова прилагођених увођењу MDSE приступа у постојеће софтверске пројекте. SAGED приступом се ово постиже кроз i) пружање увида у идиоме који би се могли сматрати добрим кандидатима за генерисање кода и ii) аутоматизацију креирања API-ја ABG-а за генерисање кода, дефинисаног терминима идентификованих идиома. SAGED приступ се ослања на рударење идиома из постојећег, нелабелираног изворног кода коришћењем технике машинског учења по имену непараметарске Бајесове пробабилистичке граматике супституције стабала (енгл. *Probabilistic Tree Substitution Grammar – PTSG*). Главни доприноси ове дисертације укључују спецификацију SAGED приступа, примену и оптимизацију *Type-Based Markov Chain Monte Carlo* (MCMC) метода за апроксимацију непараметарске Бајесове PTSG методе, као и развој језгра за закључивање идиома које се може проширити на различите програмске језике. Такође, дисертација садржи и имплементацију SAGED приступа за програмски језик C#, заједно са студијама случаја које демонстрирају њену ефикасност. Језгро за закључивање идиома и имплементација SAGED приступа за C# су доступни као техничка решења отвореног кода.

Датум прихватања теме од стране надлежног већа:	04.06.2026
Датум одбране (Попуњава накнадно институција):	
Чланови комисије:	**Председник:** др Игор Дејановић, редовни професор, Факултет техничких наука, Универзитет у Новом Саду **Члан:** др Слађан Бабарогић, редовни професор, Факултет организационих наука, Универзитет у Београду **Члан:** др Немања Милошевић, доцент, Природно-математички факултет, Универзитет у Новом Саду **Члан:** др Жељко Вуковић, доцент, Факултет техничких наука, Универзитет у Новом Саду **Ментор:** др Гордана Милосављевић, 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:	Nenad Todorović
Supervisor (title, first name, last name, position, institution):	Gordana Milosavljević, PhD, Full professor, Faculty of Technical Sciences, University of Novi Sad
Thesis title in English:	Automating the Development of API-Based Generators Using Source Code Idioms Mining
Language and script:	English language, Latin script
Physical description:	Pages: 104 Chapters: 9 References: 120 Tables: 3 Figures: 35 Graphs: 0 Appendices: 0
Scientific field:	Electrical and Computer Engineering
Scientific subfield (scientific discipline):	Applied Computer Science and Informatics
Subject, Key words:	Model-Driven Software Engineering, Model-to-Text Transformations, Code Idioms Mining, API-Based Generators

[2] The author of the doctoral dissertation has signed the following Statements:
5б – Statement on the authorship,
5в – Statement that the printed and e-version of the doctoral dissertation are identical and authorization to use personal data,
5г – Copyright statement.
The paper and e-versions of Statements are held at the institution and are not included into the printed thesis.

Abstract:	API-Based Generators (ABGs) allow practitioners to achieve the advantages of Model-Driven Software Engineering (MDSE) without making significant changes to their current workflow or the architecture of their solution. However, using ABGs that are defined in terms of the syntax rules of a target language can be cumbersome and impractical. Additionally, manually developing an ABG based on higher-level concepts already used in an existing solution is labor-intensive and time-consuming. This thesis introduces a new approach called Semi-automatic API-based Generators Development (SAGED) to expedite the creation of ABGs customized for MDSE development of existing solutions. SAGED accomplishes this by (i) providing insight into code idioms that could be considered good candidates for code generation, as they frequently appear in the codebase, and (ii) automating the creation of a code-generation API for an ABG, defined in terms of the identified code idioms. The SAGED approach relies on mining code idioms from existing, unlabeled source code based on a machine learning technique called the non-parametric Bayesian Probabilistic Tree Substitution Grammar (PTSG). The main contributions of this thesis include the introduction of the SAGED approach, an explanation and adaptation of the Type-Based MCMC as a method for approximating the non-parametric Bayesian PTSG, and the development of an open-source inference core for implementing the inference method in different programming languages. Furthermore, the thesis presents a solution for implementing SAGED in the C# programming language, along with case studies that demonstrate its effectiveness. The inference core and the C# implementation of SAGED are publicly available as open-source solutions.

Date of endorsement by the scientific board:	04.06.2026.
Date of defence (Filled in by the institution):	
Thesis defence board: (title, first name, last name, position, institution):	**President:** Igor Dejanović, PhD, Full professor, Faculty of Technical Sciences, University of Novi Sad **Member:** Slađan Babarogić, PhD, Full professor, Faculty of Organizational Sciences, University of Belgrade **Member:** Nemanja Milošević, PhD, Assistant professor, Faculty of Sciences, University of Novi Sad **Member:** Željko Vuković, PhD, Assistant professor, Faculty of Technical Sciences, University of Novi Sad **Mentor:** Gordana Milosavljević, PhD, Full professor, Faculty of Technical Sciences, University of Novi Sad
Note:	

Мојим најмилијима,
јер су увек уз мене.

Резиме

Увод и мотивација

Дисертација се бави аутоматизацијом развоја API-базираних генератора кода (енгл. *API-Based Generators* – ABG) ради лакшег увођења софтверског инжењерства вођеног моделима у животни циклус постојећих софтверских пројеката.

Развој и одржавање сложених софтверских решења која су годинама имплементирана могу постати изузетно захтевни. Френс и Румпе [1] као значајан узрок тешкоћа у развоју сложеног софтвера наводе широк концептуални јаз између домена проблема и домена имплементације (енгл. *problem-implementation gap*). Програмер решење имплементира апстракцијама нижег нивоа од оних којима је проблем изражен, чиме се у развој уноси сложеност која није својствена самом развоју софтвера [2]. Софтверско инжењерство вођено моделима (енгл. *Model-Driven Software Engineering* – MDSE) овај јаз сужава тако што моделе третира као примарне артефакте развоја, а извршиву апликацију изводи из модела применом трансформација [3]. Трансформације чији је излаз текст називају се трансформације модела у текст (енгл. *Model-to-Text* – M2T), а њихова примена ради добијања програмског кода назива се генерисање кода [3]. Пошто су модели на вишем нивоу апстракције и ближи домену проблема од програмских језика опште намене, погоднији су за спецификацију, развој и одржавање сложених софтверских решења [4].

Најефикаснији начин преласка на MDSE је измена постојећег развојног процеса тако да укључи препоручене MDSE технике, што обично повлачи за собом и промену архитектуре. Такав прелазак носи потенцијалне трошкове и ризике, а заинтересоване стране често сумњају у корист коју промене могу донети и опиру се значајним изменама својих развојних процеса [5]. Отуда

потреба за *постепеним* преласком, при чему је увођење MDSE приступа ограничено постојећим окружењем, процесима и архитектуром, а ручне измене кода морају бити подржане барем док прелазак не буде завршен.

Кључно је питање: како очувати ручне измене између два циклуса генерисања кода? Већина аутора предлаже да се ручно писани и генерисани код држе одвојено и накнадно интегришу механизмима као што су генерацијски јаз, парцијалне класе или заштићене зоне [6]. Ови механизми, међутим, намећу структуру пројекту, коју постојећа архитектура или коришћени програмски језици не морају нужно подржати. Такође, наручилац може поставити захтев да ручно мења генерисани код [7].

Да би се омогућило генерисање кода уз очување ручних измена без промене архитектуре постојећег пројекта, могу се користити API-базирани генератори – библиотеке и радни оквири који обезбеђују API за прецизне измене програмског кода уз очување његове синтаксне исправности, чиме се спречавају грешке које би могле ометати процес развоја и нарушити поверење наручиоца [8]. Примери су JavaParser API [9] за језик Java и Syntax Tree API платформе Roslyn [10] за језик C#. У нашим претходним истраживањима, ABG-ови су омогућили неометану (*seamless*) интеграцију генерисаног и ручно писаног кода при аутоматизацији развоја линије софтверских производа у индустријском окружењу [11, 12, 13].

Употреба ABG-ова, међутим, има своје препреке. API се формулише у терминима граматике програмског језика за који је ABG намењен, а ефикасност коришћења ABG-а као основе за развој генератора кода директно зависи од нивоа апстракције његовог API-ја [8]. Програмски језици опште намене (енгл. *General-Purpose Programming Language* – GPL) имају сложене граматике, па API заснован на њиховим концептима захтева навођење свих граматичких детаља и детаљно познавање синтаксних правила. Наша претходна истраживања показују да је развој генератора кода знатно спорији када се уместо обрађивача шаблона користе ABG-ови [12]. Ручно проширење скупа подржаних концепата може приближити ABG домену проблема, али захтева познавање постојећег кода, идентификовање понављајућих концепата вредних укључивања у API и имплементацију одговарајућих API метода.

Пошто је у фокусу ове дисертације постепена интеграција MDSE у развој постојећих система, може се искористити чињеница да су елементи пословног и техничког домена већ присутни у постојећем коду. Обрасци погодни за аутоматизацију могу се пронаћи рударењем идиома: концептуално заокружених образаца који се често понављају у програмском коду софтверских решења и имају једну семантичку улогу [14, 15]. На основу

тако стеченог знања може се аутоматизовати специјализација ABG-а, чиме његов API постаје концизнији и примењивији за дати задатак.

Предмет, циљ и хипотезе истраживања

Потреба за истраживањем произлази из следећег:

- Интеграција генерисаног и ручно писаног кода путем специјализованог MDSE алата може се постићи помоћу ABG-ова јер пружају више предности, укључујући могућност прецизних захвата над програмским кодом уз очување синтаксне исправности. Међутим, ABG-ови опште намене са API-јима дефинисаним концептима GPL-а су тешки за коришћење, а писање програмске логике може узети доста времена [8, 12].

- Ручно креирање ABG-ова специфичних за одређени домен захтева од аутора MDSE алата познавање и проблемског домена и програмског кода, као и техника за проширивање ABG-ова новим API методама.

- Постојећа истраживања сугеришу да је проналажење идиома у коду могуће коришћењем метода за рударење идиома [14, 16]. Међутим, у литератури се не могу наћи сви детаљи математичког модела који би омогућио паралелну обраду више чворова синтаксних стабала одједном, нити опис скалабилне имплементације.

- Према нашем сазнању, истраживање како успешно аутоматизовати развој ABG-ова и приближити их проблемском домену до сада није обрађивано у научној и стручној литератури.

Примарни циљ дисертације је дефинисање приступа и пројектовање имплементације за:

i. полуаутоматско откривање идиома који се понављају у коду постојећих пројеката, који су добри кандидати за генерисање кода и корисни за будући развој, и

ii. полуаутоматско креирање API-ја за генерисање кода, дефинисаног у терминима идентификованих идиома, који улази у састав ABG-а.

Крајњи циљ је ABG чији је API дефинисан истовремено у терминима синтаксе програмског језика и релевантних концепата вишег нивоа, при

чему свака измена начињена таквим ABG-ом мора произвести синтаксно исправан код. Приступ мора бити независан од конкретног програмског језика и ABG-а, а његова имплементација проширива за подршку произвољном програмском језику.

У складу са наведеним, постављена је општа хипотеза:

H0: Могуће је креирати решење за аутоматизацију развоја API-базираних генератора кода прилагођених техничком домену који је специфициран програмским кодом датог софтверског решења.

Из ње су изведене следеће хипотезе:

H1: Могуће је полуаутоматски идентификовати релевантне идиоме у датом програмском коду.

H2: Идентификација идиома може се извести на скалабилан начин.

H3: Могуће је аутоматизовати развој ABG-ова на основу идентификованих идиома.

Поред могућности креирања таквог решења, потребно је истражити и ефикасност добијених ABG-ова, кроз хипотезу:

H4: API методе ABG-а, аутоматски изведене из релевантних идиома кода, ефикасније су по питању времена развоја и потребне количине кода у односу на основне API методе базиране на концептима граматике језика.

Теоријске основе и преглед литературе

У другом поглављу изложене су теоријске основе дисертације: концепти MDSE парадигме и основе рударења идиома кода. MDSE је позиционирано у односу на софтверско инжењерство засновано на моделима (енгл. *Model-Based Software Engineering* – MBSE), развој софтвера вођен моделима (енгл. *Model-Driven Software Development* – MDSD) и архитектуру вођену моделима (енгл. *Model-Driven Architecture* – MDA) [3, 17]. Објашњени су његови кључни концепти на које се дисертација ослања: модел, метамодел, четворослојна архитектура мета-нивоа [2, 18], апстрактна и конкретна синтакса, као и трансформације модела [3, 19, 20].

Код генерисања кода анализирани су следећи механизми: (1) генерисање засновано на шаблонима и (2) генерисање засновано на API-ју [8, 21]. Први је једноставан и брз за имплементацију, шаблон се пише тако да обезбеди очекивани излаз, али се синтаксна исправност ретко гарантује, а интеграција са постојећим програмским кодом је ван домена разматрања. Други генерише и прецизно мења програмски код методама ABG-а чији је дизајн наметнут правилима граматике.

Поред наведеног, разматрани су и различити односи модела и кода [22], као и механизми за интеграцију ручно писаног и генерисаног кода [6].

Идиоми су дефинисани као мали фрагменти који се често јављају у програмском коду датог софтверског решења и имају јасно одређену семантичку улогу. Одговарају често понављаним подстаблима синтаксних стабала [14, 23]. Могу садржати метапроменљиве – места која се разликују од њихове појаве до појаве у коду. Од интереса су за проширење ABG-а јер документују како се језик, развојни оквир или библиотека користе у постојећем систему, а метапроменљиве се природно пресликавају у параметре API метода.

Алтернативне технике за откривање понављања у коду су такође анализиране, иако су мање погодне за нашу примену. Детекција клонова [24] налази највећи, а не најкориснији фрагмент [14]. Рударење API-ја [25] занемарује синтаксну структуру, а алгоритми рударења честих стабала [26] враћају мале и генеричке фрагменте [27].

У трећем поглављу је дат преглед сродних истраживања. Дискутована су наша ранија решења која разматрају могућност увођења MDSE без потребе за изменом постојећег развојног процеса [11, 12, 13] коришћењем ABG-а, са нагласком на могућим побољшањима. Показано је да лакоћа коришћења ABG-а директно зависи од нивоа апстракције његовог API-ја [8]. На пример, Roslyn Syntax Tree API омогућава интервенцију на било ком елементу синтаксног стабла јер API подржава све концепте врло комплексне граматике C# језика. Дакле, API је на врло ниском нивоу апстракције и тежак је за коришћење. Концизнији ABG-ови попут JavaPoet [28] су ограничени на фиксан подскуп концепата језика. Хибридна решења, као што су Testura.Code [29] и наша библиотека RoseLib [11, 30], нуде API на вишем нивоу апстракције уз могућност коришћења основног Roslyn Syntax Tree API-ја за случајеве које не подржавају.

Пошто је циљ ове дисертације да се из постојећег кода извуку концепти релевантни за даљи развој, који би се потом уврстили у API ABG-а, у наставку поглавља се анализира приступ Аламаниса и Сатона [14] за рударење идиома из кода по имену непараметарска Бајесова пробабилистичка граматика супституције стабала (енгл. *Probabilistic Tree Substitution Grammar* – PTSG). То је најсавременији приступ за рударење идиома из синтаксних стабала који припада пробабилистичким моделима за ненадгледано откривање образаца у нелабелираном коду [31]. За практичну имплементацију Аламанис и Сатон предлажу методу Type-Based MCMC (енгл. *Markov Chain Monte Carlo*) [32], али у литератури се не може наћи опис како је применити на домен рударења идиома из кода.

На крају поглавља су анализирани радови који се баве аутоматизацијом развоја MDSE алата: детекција клонова за извођење QVT (енгл. *Query/View/Transformation*) шаблона [33] и језика специфичних за домен (енгл. *Domain-Specific Language* – DSL) [34], учење правила превођења по примеру [35, 36], LSTM (енгл. *Long Short-Term Memory*) неуронске мреже за закључивање трансформација [20] и велики језички модели (енгл. *Large Language Models* – LLM) [37, 38]. Наведени радови не разматрају аутоматизацију развоја ABG-ова. За нашу примену неуронске мреже нису погодне јер су „црне кутије“ чија се научена правила тешко мењају, док су граматике супституције стабала разумљиве, а добијени идиоми лако измењиви.

SAGED приступ

Централни допринос дисертације је SAGED (енгл. *Semi-automatic API-based Generators Development*), нови приступ за полуаутоматски развој API-базираних генератора који је презентован у поглављу 4. Приступ се састоји од две фазе и четири корака (слика 4.1):

- Фаза 1: Откривање концепата:
  - корак S1: закључивање идиома из програмског кода,
  - корак S2: курирање.
- Фаза 2: Проширење ABG-а:
  - корак S3: генерисање API метода,
  - корак S4: интеграција API метода у ABG.

За закључивање идиома изабрана је апроксимација непараметарске Бајесове PTSG методе [39, 16], реализована прилагођеном варијантом Type-Based MCMC методе [32]. Избор је мотивисан њеним својствима дискутованим у претходним поглављима: у питању је ненадгледано машинско учење, које не захтева лабелирање кода. Метода је заснована на статистичком моделу, па закључује идиоме који су значајнији за нашу примену од метода које се ослањају само на учесталост појаве сличних фрагмената, а закључени идиоми нужно поштују граматичка правила језика. Успех закључивања зависи од улазног скупа: програмски код мора показивати известан степен репетитивности, што није значајно ограничење јер је изворни код неретко

репетитиван [40]. Типични примери су пројекти који следе устаљене развојне праксе или користе радне оквире који прописују начин програмирања, као и скупови сродних варијанти истог производа попут SPL-а.

Међутим, метода гарантује синтаксну исправност идиома, али не и његову потпуност (на пример, идиом може почети `else` граном без `if` гране, слика 5.2). Због тога наш приступ и алат обезбеђују додатне, језички специфичне информације о томе која врста чвора стабла сигурно није почетак фрагмента, чиме се закључивање усмерава ка креирању потпуних идиома. Пошто су семантика и квалитет идиома субјективни и зависе од контекста [31], а ова метода често налази плитке идиоме, неопходан је корак курирања у коме се врши преглед резултата, бирање корисних идиома, те спајање и прилагођавање случајевима коришћења. По потреби, закључивање се може поновити са другим параметрима или на подскупу датотека са програмским кодом. Курирани идиоми, са контекстуалним подацима потребним за интеграцију, улаз су у другу фазу.

У другој фази се полази од постојеће ABG основе (нпр. Roslyn Syntax Tree API) која подржава основне операције креирања, читања, ажурирања и брисања (енгл. *Create, Read, Update, Delete* – CRUD) над синтаксним елементима, управљање стањем програма приликом уланчавања операција, као и откривање синтаксних грешака. ABG основа би требало да има флексибилну структуру у коју се интегришу и генерисане и ручно додате API методе. У кораку S3 се на основу одабраних идиома генеришу API методе за постојећу ABG основу које користе њену подршку и поштују њену структуру. У кораку S4 оне се спајају са основом.

Резултујући ABG треба да задовољи следеће захтеве:

1. прецизну селекцију постојећих елемената кода,
2. инкременталне измене, које омогућавају креирање конструкција већих од оних које ствара појединачна метода,
3. очување синтаксне исправности, уз јављање упозорења када се она не може постићи,
4. механизме раздвајања ручно писаних и генерисаних API метода,
5. свеобухватну подршку за измене, тј. покривање већине очекиваних сценарија измене синтаксног стабла, уз давање алтернативних начина за неподржане.

Закључивање идиома кода

Пето поглавље даје математичке основе методе закључивања и дискутује допринос дисертације у овој области. Полазећи од контекстно слободне граматике (енгл. *Context-Free Grammar* – CFG) и њене пробабилистичке варијанте (енгл. *Probabilistic CFG* – PCFG), уводи се граматика супституције стабала (енгл. *Tree Substitution Grammar* – TSG) и њена пробабилистичка варијанта PTSG. TSG је карактеристична по томе што нетерминал може бити преписан целим фрагментом стабла, а не само непосредним потомцима (пример је дат на слици 5.2). Фрагменти PTSG кодирају нелокални контекст, па су погодни за представљање идиома, а нетерминални листови природно представљају метапроменљиве. Проблем проналажења идиома своди се на учење PTSG која проширује познату граматику G_0 и добро објашњава скуп за обучавање [14, 16].

Закључивање се изводи Бајесовим приступом, при чему се као априорна расподела над фрагментима користи Дирихлеов процес. Он има бесконачну подршку, тако да сваком фрагменту додељује позитивну априорну вероватноћу и испољава феномен „богати постају богатији“, по коме се за фрагмент који је већ уочен у скупу података предвиђа да ће се у будућности појављивати чешће [39]. Базна расподела фрагмента рачуна се као производ геометријске расподеле над величином фрагмента и вероватноћа PCFG продукција од којих је фрагмент састављен. Пошто тачну апостериорну расподелу није могуће израчунати, она се апроксимира Гибсовим узорковањем [41] које итеративно генерише узорке из апостериорне расподеле сваке променљиве, условљене вредностима свих осталих.

Сваком чвору стабла придружена је бинарна променљива b_x: вредност 1 означава да је чвор корен новог фрагмента, а вредност 0 да је чвор унутрашњи чвор текућег фрагмента. Вероватноћа да два суседна фрагмента f_s и f_t буду спојена у један фрагмент f_{join} рачуна се из апостериорних вероватноћа сва три фрагмента, тако да фрагменти који се често јављају заједно имају већу вероватноћу да остану заједно. Метода се тако своди на пет корака: (1) парсирање датотека познатом граматиком G_0, (2) трансформацију добијених стабала (нпр. бинаризацију), (3) додељивање почетних вредности променљивама b_x ради формирања почетних фрагмената, (4) итеративно ажурирање вредности b_x узорковањем из наведене расподеле и (5) узорковање фрагмената након сваке итерације, по истеку периода сагоревања (енгл. *burn-in*), на основу броја њихових појављивања.

Гибсово узорковање није скалабилно јер обрађује чвор по чвор. Из тог

разлога преузимамо побољшање Лианга и сарадника [32] по имену Type-Based MCMC, на основу којег се чворови истог типа групишу у блок и ажурирају заједно у једном кораку алгоритма. Према нашем сазнању, у овој дисертацији је метода први пут детаљно протумачена и разрађена за домен PTSG, слично прилагођавању које су Пенг и Гилдеа [42] извели за синхроне контекстно слободне граматике.

Тип чвора према општој дефиницији одређен је утицајем који додела вредности променљивој b_x има на вектор бројача фрагмената. Допринос дисертације је поједностављење: у домену PTSG додела нуле оставља фрагмент f_{join} целовитим, а додела јединице га дели на фрагменте f_s и f_t. На овај начин се тип дефинише тројком (f_{join}, f_t, f_s), чиме се уместо целог вектора промена чува само делимична информација о окружењу чвора, што значајно смањује потребу за вођењем евиденције и утрошак меморије. Чворови истог типа групишу се само ако нису у конфликту, тј. ако се њихови фрагменти f_{join} не преклапају. За блок B неконфликтних чворова истог типа расподела броја јединица m које се додељују блоку пропорционална је $\binom{|B|}{m} g(m)$ [32], где је $g(m)$ функција бројача фрагмената ван блока, параметра концентрације и базне расподеле. У дисертацији се изводи упрошћена варијанта ове функције, прилагођена домену PTSG: једини чиниоци производа различити од јединице јесу они који се односе на три фрагмента која дефинишу тип – f_{join}, f_t и f_s – и на нетерминале у њиховим коренским чворовима (енгл. *root node*). Функција стога има два облика, зависно од тога да ли је коренски чвор фрагмента f_s исте врсте као заједнички коренски чвор фрагмената f_{join} и f_t. Корак узорковања се завршава узорковањем m и доделом m јединица чворовима блока.

Поглавље се завршава презентацијом имплементационих детаља везаних за предобраду парсираних стабала. Прво, стабла се бинаризују, чиме граматика постаје мање ретка, а идиоми у секвенцама наредби боље се обухватају. Друго, делови конструкција који не могу самостално да постоје придружују се целини којој припадају (нпр. `catch` блок уз `try` блок), како би закључени идиоми били потпуни. Треће, поједине врсте чворова унапред се фиксирају као коренски, односно некоренски чворови фрагмената, чиме се закључивање усмерава ка идиомима чији су коренски чворови структурни елементи програмског језика које ABG основа може да прихвати.

RoseLib као основа ABG-а

Шесто поглавље описује RoseLib [11, 43, 30], ABG отвореног кода писан у језику C#, развијен у нашим раније истраживањима као слој изнад Roslyn Syntax Tree API-ја, који се у дисертацији користи као ABG основа. RoseLib садржи методе везане за концепте језика C# познате програмерима, механизме за навигацију кроз програмски код, проверу ручних измена кода између циклуса генерисања и синтаксно исправне измене Roslyn C# синтаксних стабала (енгл. *Roslyn C# Syntax Tree* – RCST). Обилазак реализују *навигатори*, класе везане за концепте језика чије се методе селекције уланчавају тако да употреба следи правила граматике, уз праћење посећених чворова у класи `StatefulVisitor`. Метода `StartComposing` обезбеђује типски безбедан прелазак на измену и враћа одговарајући *композер* – члан хијерархије класа (слика 6.2) са основним операцијама креирања, ажурирања и брисања које се комбинују у сложеније трансформације. Стањем приликом употребе се управља помоћу *пивот* чвора, централног за операцију, и *референтног* чвора, потомка који даје додатни контекст.

RoseLib поседује и додатне функционалности које омогућавају лакшу употребу у MDSE домену: повезнике за праћење (енгл. *traceability links*), варијанту механизма који користи језик MOFM2T [44], и проверу интегритета (енгл. *integrity checks*). Ради имплементирања повезника за праћење, развијен је језик CSPath, који је инспирисан језиком XPath [45] и имплементиран метајезиком textX [46]. Његов интерпретер је заснован на обрасцу ланца одговорности [47]. CSPath омогућава писање текстуалних израза који се могу чувати ван програмског кода и који омогућавају да се одређени елемент поново пронађе у коду у наредним циклусима генерисања. Провере интегритета израчунавају хеш вредност фрагмента кода који одговара подстаблу са кореном у изабраном чвору, након уклањања коментара и нормализације, тако да секундарна нотација не утиче на резултат. Поређењем са раније сачуваном вредношћу генератор утврђује да ли је програмски код ручно мењан између два циклуса.

На крају поглавља је дат пример коришћења генератора.

Имплементација SAGED приступа за језик C#

Седмо поглавље представља референтну архитектуру (слика 7.1) и специјализацију SAGED методе за програмски језик C#. Корак S1 реализују компоненте *Предобрађивач* и *Инференцер*. Предобрађивач чисти улазне да-

тотеке од секундарне нотације и елемената који немају значаја за закључивање (директиве, `using` наредбе) и може да групише датотеке на основу стабла наслеђивања, које се у C# пројектима често мапира на слојеве апликације. Овим се побољшавају расподеле при закључивању и убрзава обрада.

У поглављу је описано *Језгро закључивања* (енгл. *Inference core*), језички независна компонента отвореног кода која имплементира непараметарску Бајесову PTSG методу апроксимацијом Type-Based MCMC, како је описано у петом поглављу. У оквиру истраживања је језгро закључивања проширено како би се имплементирао инференцер за C#. Уместо синтаксних стабала конкретног парсера, језгро ради над интерним стаблима (класе `Tree` и `Node`, слика 7.2), а апстрактне класе `TreeCreator`, `NodeCreator` и `Writer` су тачке проширења: прве две за превођење синтаксних стабала у интерну репрезентацију, а трећа за исписивање закључених фрагмената као идиома кода. Класа `TypeBasedSampler` изводи закључивање, а `BookKeeper` води евиденцију о чворовима, типовима и бројачима фрагмената и коренских чворова. Инференцер проширује језгро ослањајући се на Roslyn парсер.

Корак S2 подржавају *Визуализатор*, који генерише HTML приказе улазних датотека (слика 7.4) са обојеним идиомима, њиховом учесталошћу и везама ка другим појавама истог идиома, и *Руковалац*, који омогућава спајање идиома уз очување синтаксне исправности и груписање датотека са заједничким идиомом ради одвојене обраде. На крају курирања потребно је задати име API методе, имена параметара пресликаних на метапроменљиве, и RoseLib композер у који се метода интегрише.

Фазу 2 реализује *Трансформатор*, уз *базу знања* која садржи податке о томе које врсте чворова поједини композер може да обради, које шаблоне треба користити и где се налазе RoseLib и генерисани делови композера.

Трансформатор спроводи следећи поступак: изабрани фрагмент претвара у C# интерполирани стринг у коме су метапроменљиве замењене параметрима, креира декларацију методе, генерише тело методе шаблоном везаним за изабрани композер и на крају додаје методу у парцијалну класу композера. Механизам парцијалних класа обезбеђује неопходно раздвајање ручно писаних и генерисаних метода.

Свака тако генерисана метода (пример дат на слици 7.6) се при позиву понаша на следећи начин: најпре проверава да ли је тренутно стање селекције погодно за намеравану измену, затим интерполира и парсира фрагмент, па одбија операцију ако добијено подстабло није синтаксно исправно. У супротном, уграђује подстабло у RCST и селектује његов коренски чвор, тако да се наредне операције могу надовезати.

Евалуација

Осмо поглавље евалуира приступ кроз три истраживачка питања:

- **RQ1**: Да ли фаза откривања концепата производи релевантне идиоме кода?
- **RQ2**: Да ли је та фаза скалабилна?
- **RQ3**: Да ли су API методе изведене SAGED приступом ефикасније за употребу од постојећих API метода ABG-а?

Одговори на RQ1 и RQ2 потврђују хипотезе H1 и H2, хипотеза H3 потврђена је успешном имплементацијом решења из седмог поглавља, а позитиван одговор на RQ3 потврђује хипотезу H4.

Квантитативна евалуација је мерила прецизност (удео закључених идиома пронађених у тест корпусу), покривеност (проценат синтаксних чворова тест корпуса покривених идиомима закљученим из корпуса за обучавање), просечну дужину идиома и скалабилност (однос броја линија кода и времена након којег се почиње са узорковањем). Прве две метрике наликују прецизности и одзиву (енгл. *recall*) у претраживању информација, прилагођеним домену рударења идиома [14].

Скуп података се састојао од три ручно прегледана пројекта израђена у сличној технологији:

- колекције студентских пројеката [48] (шеснаест Rent-A-Car веб апликација развијених од стране двочланих тимова на основу исте почетне спецификације, што одговара скупу производа у SPL-у),
- School Bus апликације [49] (поседује јасну структуру са добро дефинисаним слојевима и високу репетитивност програмског кода) и
- Licensing апликације [50] (изабрана је јер је сложена, са недовољно добро дефинисаним слојевима).

Уз одабране пројекте, за евалуацију је искоришћено и једанаест C# пројеката отвореног кода из скупа Аламаниса и сарадника [51]. Дакле, анализирано је укупно четрнаест пројеката (табела 8.1) величине од око 4,5 хиљаде до преко 600 хиљада линија кода, обједињених у јавно доступан Mendeley Data скуп ради проверљивости експеримената.

Сваки пројекат подељен је на корпус за обучавање (70% датотека) и тест корпус (30%). Почетна вероватноћа да чвор буде изабран за коренски била

је 0,85, параметар геометријске расподеле 0,025, параметар концентрације α 5, период сагоревања 75 итерација, а праг појављивања идиома 2 уз најмање 8 чворова. У просеку је 40,1% (±18,8) идиома закључених из корпуса за обучавање пронађено у тест корпусу. Идиоми су покрили 39,1% (±17,4) чворова тест корпуса, а просечна дужина идиома била је 14 чворова.

Метода је исправно указала на мању репетитивност Licensing апликације (прецизност 25,9%, покривеност 29,0%) у односу на School Bus (71,1%, 65,0%) и студентске пројекте (75,7%, 69,5%). И код осталих пројеката, они репетитивнији су давали више вредности. Време обучавања расло је линеарно са бројем линија кода: за највећи пројекат, са преко 600.000 линија након чишћења, 75 итерација трајало је 1,5 сати на рачунару са процесором Apple M1 Pro и 32 GB меморије.

Квалитативна евалуација је примењена на мерење броја идиома које је током курирања требало ручно мењати да би били употребљиви за генерисање API метода за ABG. Закључивање је поновљено на прва три пројекта по слојевима, издвојеним помоћу Предобрађивача и ручном инспекцијом, уз фиксиране коренске чворове на структурним елементима (простори имена, класе, методе) ради усклађености са RoseLib композерима. Чланови нашег тима са искуством у ASP.NET развоју, који нису били раније упознати са разматраним пројектима, курирали су идиоме за потребе креирања MDSE алата који аутоматизује одржавање једног пројекта кроз све слојеве, а спољни програмери, са више од пет година искуства у датом технолошком стеку, оценили су њихову вредност.

За студентске пројекте идентификовано је једанаест група сродних артефаката. Курирање је трајало два сата и дало преко 50 изабраних идиома (примери дати на слици 8.2). Поред тога, 12% идентификованих идиома мало је измењено Руковаоцем да би се добиле погодније API методе. Спољни програмери препознали су семантику идиома иако им није дат никакав контекст.

Licensing апликација је показала два ограничења SAGED методе. Прво, метода је осетљива на промену редоследа елемената кода: атрибути над класама контролера, на пример, нису увек наведени истим редоследом, па закључени идиом на том месту добија метапроменљиву уместо конкретног израза. Друго, могућности закључивања ограничене су код нерепетитивног императивног кода. У овој апликацији пословна логика није издвојена у засебан слој, већ се прожима и кроз слојеве који су иначе репетитивни, попут слоја контролера, што негативно утиче на расподеле при закључивању. Оба проблема резултују плитким и генеричким обрасцима који уместо конкретних исказа садрже метапроменљиве.

Евалуација ефикасности поредила је време развоја и број линија кода при имплементацији истог MDSE алата коришћењем прво основне а затим и проширене верзије библиотеке RoseLib у коју су додате API методе генерисане на основу курираних идиома (примери њихове употребе су дати на слици 8.3). Задатак MDSE алата је био да аутоматизује одржавање једне студентске Rent-A-Car веб апликације кроз седам слојева. Са проширеним ABG-ом написано је 24% мање линија кода, а развој алата трајао је око 20% краће. Допринос је био тим већи што је изабрани идиом био сложенији.

Ови резултати су упоређени и са нашим ранијим истраживањем у индустријском окружењу, у оквиру компаније Schneider Electric у Новом Саду [12], где је за комплексни MDSE алат током више од две године израђено неколико стотина генератора, од којих око стотину за C# артефакте помоћу Roslyn-а и прве верзије RoseLib-а. Тамо је развој генератора за једну врсту артефакта у просеку захтевао 20 човек-часова, док је са SAGED приступом сличан генератор развијен за просечно 15,5 човек-часова, што за пројекте ове величине представља значајну уштеду.

Евалуација је показала да су одговори на сва три истраживачка питања потврдни, чиме су потврђене хипотезе H1–H4 и, посредно, општа хипотеза H0. Уочени су и изазови:

- Курирање би требало унапредити рангирањем и претрагом идиома, јер алат не помаже при избору најкорисније верзије међу сличним идиомима.
- Метода је осетљива на редослед елемената кода, што је делимично ублажено сортирањем у Предобрађивачу, али императивни код захтева даља побољшања.
- Приступ је најделотворнији за апликације са устаљеним конвенцијама и репетитивним кодом, док је код велике варијабилности, где конвенције нису испоштоване, неопходно веће ангажовање развојног тима.

Претње валидности односе се на:

- исправност имплементације Бајесове PTSG методе, што је адресирано прегледом кода од стране више чланова тима и јединичним тестовима (директно поређење са оригиналним радом [14] није било могуће јер су његови скупови писани на Java програмском језику).
- подобност скупа података, што је адресирано коришћењем пројеката различитог порекла, величине и квалитета.

- објективност мерења ефикасности, што је адресирано поређењем са комплексним индустријским алатом имплементираним истим технологијама, у сличном окружењу и са истим развојним тимом.

Закључак и правци даљег истраживања

У дисертацији је показано да API-базирани генератори омогућавају да се MDSE искористи за аутоматизацију даљег развоја постојећих пројеката, уз очување њихове архитектуре и развојних процеса. Употребљивост API-базираних генератора се битно повећава када се API дефинише у концептима вишег нивоа апстракције, изведеним из идиома програмског кода датог пројекта.

Поред самог SAGED приступа, који закључује идиоме непараметарском Бајесовом PTSG методом из нелабелираног кода и аутоматизује креирање API метода за куриране идиоме, доприноси дисертације су и:

- систематизација литературе о закључивању идиома кода, са објашњењем свих корака одабране методе и имплементацијом отвореног кода,
- детаљна разрада Type-Based MCMC методе за апроксимацију непараметарске Бајесове PTSG и њено прилагођавање домену PTSG ради поједностављења и оптимизације,
- језички независно језгро закључивања, објављено као решење отвореног кода и прошириво на различите програмске језике,
- проширење језгра за језик C#, заједно са архитектуром ABG основе и начином њеног проширивања новим методама.

Приступ је евалуиран квантитативно на четрнаест пројеката, квалитативно на скупу сродних студентских пројеката и два упоредива пројекта отвореног кода уз спољну валидацију, и кроз изградњу MDSE алата.

У закључку је сумирана и потврда хипотеза: у табели 9.1 је приказано како је свака од хипотеза H0–H4 потврђена и где се у раду налазе одговарајући докази. Размотрене су и импликације резултата, а објављени скуп података и измерене вредности чине основу за поређење будућих унапређења методе закључивања.

Будући рад усмерен је на смањење потребе за ручним курирањем путем унапређења алата за преглед и манипулацију идиомима, као и истраживањем побољшања методе која би, поред синтаксе, требало да узме у обзир и семантику кода. Планирано је проширење имплементације на друге

програмске језике, што језгро већ омогућава уз одговарајући парсер, као и квалитативна евалуација корисничког искуства по узору на евалуацију језика специфичних за домен. Значајан број сличних идиома који су откривени у студентским пројектима и School Bus апликацији отвара питање поновне употребе специјализованих генератора у сродним техничким доменима, у смислу креирања дељене библиотеке API метода. Најзад, све већа распрострањеност генерисања кода коришћењем великих језичких модела (LLM) отвара интересантан правац истраживања — њихову интеграцију са формалнијим MDSE методама, при чему би трансформације које изводе језички модели могле бити сачуване као поновно употребљиви и предвидиви ABG изрази.

Методологија истраживања

Истраживање је спроведено по методологији истраживања у науци о дизајну (енгл. *Design Science Research Methodology* – DSRM) [52, 53] у варијанти Пеферса и сарадника [54], чији су кораци: (1) идентификација проблема, (2) дефинисање циљева решења, (3) пројектовање и развој, (4) демонстрација решења, (5) евалуација и (6) комуникација резултата.

1. Идентификација проблема и мотивација је дискутована у уводу у секцијама 1.1 и 1.3, теоријским основама у поглављу 2 и кроз анализу сродних истраживања у поглављу 3.

2. Дефинисање циљева решења, хипотеза, захтева који се постављају пред решење и очекиваних резултата је дато у секцијама 1.2, 1.4 и 4.2.

3. Пројектовање и развој је обрађено кроз спецификацију SAGED приступа у поглављу 4, извођење математичког модела закључивања идиома са прилагођавањем Type-Based MCMC методе домену PTSG у поглављу 5, опис RoseLib библиотеке као основе за развој ABG-а у поглављу 6 и, на крају, кроз референтну архитектуру са имплементацијом свих компоненти и алата покривених јединичним тестовима у поглављу 7.

4. Демонстрација решења је покривена кроз примену фазе откривања концепата на три пројекта и изградњу MDSE алата помоћу изведених API метода (секције 8.2 и 8.3).

5. Евалуација је извршена у поглављу 8 кроз квантитативну евалуацију на четрнаест пројеката (секција 8.1), квалитативну евалуацију са спољном валидацијом (секција 8.2), поређење ефикасности са основним ABG-ом и студијом из индустрије [12] (секција 8.3), дискусију и анализу претњи валидности (секције 8.4 и 8.5).

6. Комуникација резултата је обављена кроз ову дисертацију, објављене радове [43, 11, 12, 13], решења отвореног кода [30, 55, 56, 57] и јавно доступан скуп података [48].

Abstract

API-Based Generators (ABGs) allow practitioners to achieve the advantages of Model-Driven Software Engineering (MDSE) without making significant changes to their current workflow or the architecture of their solution. However, using ABGs that are defined in terms of the syntax rules of a target language can be cumbersome and impractical. Additionally, manually developing an ABG based on higher-level concepts already used in an existing solution is labor-intensive and time-consuming. This thesis introduces a new approach called Semi-automatic API-based Generators Development (SAGED) to expedite the creation of ABGs customized for MDSE development of existing solutions. SAGED accomplishes this by (i) providing insight into code idioms that could be considered good candidates for code generation, as they frequently appear in the codebase, and (ii) automating the creation of a code-generation API for an ABG, defined in terms of the identified code idioms. The SAGED approach relies on mining code idioms from existing, unlabeled source code based on a machine learning technique called the non-parametric Bayesian Probabilistic Tree Substitution Grammar (PTSG). The main contributions of this thesis include the introduction of the SAGED approach, an explanation and adaptation of the Type-Based MCMC as a method for approximating the non-parametric Bayesian PTSG, and the development of an open-source inference core for implementing the inference method in different programming languages. Furthermore, the thesis presents a solution for implementing SAGED in the C# programming language, along with case studies that demonstrate its effectiveness. The inference core and the C# implementation of SAGED are publicly available as open-source solutions.

Contents

List of Algorithms

List of Tables

List of Figures

List of Acronyms

ABG	API-Based Generator
ANN	Artificial Neural Network
API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Context-Free Grammar
CRUD	Create, Read, Update, Delete
DP	Dirichlet Process
DSL	Domain-Specific Language
DSRM	Design Science Research Methodology
GCS	Graphical Concrete Syntax
GPL	General-purpose Programming Language
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
LLM	Large Language Model
LoC	Lines of Code
LSP	Language Server Protocol
LSTM	Long Short-Term Memory

M2M	Model-to-Model (transformation)
M2T	Model-to-Text (transformation)
MBSE	Model-Based Software Engineering
MCMC	Markov Chain Monte Carlo
MDA	Model-Driven Architecture
MDSD	Model-Driven Software Development
MDSE	Model-Driven Software Engineering
ML	Maximum Likelihood
MOF	Meta Object Facility
MOFM2T	MOF Model to Text Transformation Language
OMG	Object Management Group
PCFG	Probabilistic Context-Free Grammar
PDM	Platform Description Model
PTSG	Probabilistic Tree Substitution Grammar
QVT	Query/View/Transformation
RCST	Roslyn C# Syntax Tree
REST	Representational State Transfer
SAGED	Semi-automatic API-based Generators Development
SE	Software Engineering
SPL	Software Product Line
T2M	Text-to-Model (transformation)
T2T	Text-to-Text (transformation)
TBCG	Template-Based Code Generation
TCS	Textual Concrete Syntax

TSG	Tree Substitution Grammar
UI	User Interface
UML	Unified Modeling Language
VCS	Version Control System

1 Introduction

1.1 API-Based Generators in Model-Driven Software Engineering

Further manual development and maintenance of a software solution with a large codebase, which has rapidly grown over the years, can become challenging. To change or significantly extend such a solution without unwanted consequences, one must have detailed knowledge of its internal structure and the interconnections between its components.

France and Rumpe [1] argue that a significant factor behind the difficulty of developing complex software is the wide conceptual gap between the problem and the implementation domains of discourse. The manifestation of such a problem-implementation gap is that a developer implements software solutions using abstractions that are at a lower level than those used to express the problem. So, in the case of complex problems, bridging the gap using methods that rely almost exclusively on human effort will introduce significant accidental complexities that are not inherent to the development of software [2]. In effect, current manual code-centric development approaches are not always adequate for mitigating and managing the complexities of such large-scale solutions.

Model-Driven Software Engineering (MDSE) is a software development approach that utilizes models as primary software development artifacts, with the key premise that programs are automatically derived from such models [3]. Models are key to success in modern software engineering, in domains such as railway systems, automotive, business process engineering, and embedded systems [58]. Reports claim various benefits of using MDSE, such as shorter development time with fewer bugs, easier maintenance, and a better understanding of the problem and design [59, 60, 61].

The advantage of MDSE is that models are at a higher level of abstraction, much closer to the problem domain relative to most programming languages, which makes models more suitable to specify, produce, and maintain complex software solutions [4]. An executable solution is usually derived from models using transformations. Transformations aimed at producing textual output are called Model-to-Text (M2T) transformations, and the operation of applying M2T transformations to create code is called code generation [3]. The automatic derivation of an executable solution through code generation helps bridge the problem-implementation gap without introducing unnecessary accidental complexities.

From the perspective of an MDSE practitioner, the most effective way to transition from manual development to the MDSE approach is to modify the current system's development workflow to incorporate MDSE practices and update or rebuild the existing architecture to support it. However, this transition comes with costs and risks. Current research indicates that stakeholders may doubt the benefits of this change and resist making significant modifications to their development processes [5].

1.1.1 Gradual transition to MDSE facilitated by ABGs

When aiming for a gradual transition, the implementation of MDSE is constrained by the existing environment, processes, and architecture. Manual changes must be supported at least until the transition is complete. Therefore, it is important to consider how manual changes are preserved between code generation iterations. Most authors suggest that hand-written and generated code should be kept and maintained separately to prevent overwriting manual changes by a code generator. Their integration can be achieved using mechanisms such as generation gap, partial classes, protected regions, and others [6]. However, there may be cases where the existing architecture or code artifacts do not allow for the use of these mechanisms. Additionally, stakeholders may request to retain flexibility and directly change the generated code [7].

Application Programming Interfaces (APIs) are the user interfaces of libraries or functionalities used by programmers [62]. They support programmers by promoting code reuse, providing abstractions that simplify programming tasks, and unifying their programming experience [63]. Libraries and frameworks that offer an API for making precise modifications to programming code, while maintaining its syntax correctness, are known as API-Based Generators (ABGs), and they implement the fundamental code generation pattern of the same name [8]. Examples of ABGs include JavaParser API [9], for generat-

ing Java source files, and the Syntax Tree API of Roslyn [10] (code name for the .NET Compiler Platform). Comprehensive ABGs allow for precise changes to specific parts of the code while maintaining syntax correctness, which helps prevent mistakes that could disrupt the development process and damage stakeholders' trust.

1.1.2 Problems with ABGs and a possible solution

The suitability of an ABG depends on the abstraction level of its API. General-purpose Programming Languages (GPLs) have complex grammars, and an API based on these grammars must be expressive enough to support various use cases. Using ABGs as a foundation for developing MDSE tools for complex projects can be time-consuming and labor-intensive due to the need to specify every grammar detail [8]. Additionally, it requires a deep understanding of the complex syntax rules of a GPL, which may discourage some developers from using ABGs for code generation. Our previous research indicated that the development of code generators is much slower when using ABGs in comparison with templates [12]. Although these two approaches to developing code generators cannot be directly compared, as the use cases are not entirely overlapping, it would be helpful to make development with ABGs less cumbersome.

To make ABGs more user-friendly, we need to define the API in terms of higher-level concepts that extend beyond syntax-related elements. This would enable coarse-grained code manipulation at the level of code fragments associated with these concepts. By reducing the need to specify detailed grammar rules, ABGs would become more concise and easier to use. Consequently, using ABGs to automate development tasks would become more accessible. However, obtaining such a customized ABG for a specific use case requires identifying the appropriate set of higher-level concepts.

As we focus on the gradual transition from the manual development of an existing system to development based on the MDSE approach, we can leverage the circumstances. We can learn from the parts of the business and technical domains that have already been explored and identify patterns in the code that seem useful for automation. Learning can be done through code idioms mining, where code idioms are useful concept-embodying patterns contained within the codebase of a software solution [14, 15]. Based on the obtained knowledge, we can automate an ABG's specialization to make its API more concise and practical for the given task.

1.2 Defining and describing the subject of the research

This research aims to utilize inference of useful code idioms from the codebase of a software solution and automatically derive a specialized ABG that can enable easier automation of the software solution's future development.

With the explained goal of the research, we first specify the general hypothesis **H0**.

- **H0**: It is possible to create a solution for automating the development of API-based code generators tailored to the technical domain specified by the given codebase.

From the hypothesis H0, we can formulate the following hypotheses that we aim to validate. These correspond to the two parts of the specified goal of the research.

- **H1**: It is possible to semi-automatically identify relevant code idioms in the given codebase.
- **H2**: The identification of code idioms can be performed in a scalable manner.
- **H3**: It is possible to automate the development of ABGs based on the identified code idioms.

The combination of hypotheses **H1** to **H3**, together with the supporting implementation of a system that facilitates reaching the set goal, can be used to support assumptions related to the solution from **H0**.

After addressing the possibility of crafting a solution, we also need to focus on the effectiveness of ABGs produced using the said solution.

- **H4**: API methods of an ABG, automatically derived from relevant code idioms, are more efficient in terms of development time and the amount of code required compared to existing basic language-related ABG API methods.

1.3 Explaining the need for research

To summarize, the need for research stems from the following:

- Creating the logic for integrating generated and manually written code through a specialized MDSE tool can be achieved with ABGs, as they provide a few benefits. They facilitate precise integration and a syntactically valid output. However, comprehensive ABGs with APIs defined in terms of a GPL are cumbersome to use, and writing the logic can take a lot of time [8, 12].

- Bringing ABGs closer to the domain problem being solved manually requires MDSE technicians not only to become familiar with both the problem domain and the code, but also to know how to extend an ABG of interest with a more suitable API.

- Existing research suggests that finding repetitive code idioms in a codebase is possible using code idioms mining [14, 16]. However, the literature does not specify all the needed steps for a scalable implementation, so a systematization of the literature is needed.

- To the best of our knowledge, the research on how to successfully automate the development of ABGs and bring them closer to the domain problem is missing from the literature.

1.4 Research objective with emphasis on expected results

The primary research objective is to define an approach and design an implementation for:

- The semi-automatic identification of recurring code idioms in the existing codebase that are good candidates for code generation and beneficial for future development needs.

- The semi-automatic creation of a code-generation API for an ABG defined in terms of the identified code idioms.

The end goal of the defined approach is: a scenario where the API of an ABG, meeting the requirements, is defined in terms of both the syntax of a programming language and relevant higher-level concepts. The API of an ABG is defined in terms of the grammar of a target programming language, as well as more complex constructs corresponding to code idioms inferred from the existing codebase. Naturally, code alterations performed using such an ABG

must produce syntactically correct code. The expectation is that by expanding the range of supported concepts, implementing code generation becomes more manageable as the ABG aligns more closely with the problem domain.

The designed approach has to be general-purpose, independent of any specific programming language or ABG. Its implementation must be extensible to support any arbitrary programming language.

Given the primary objectives, the end goal, and the accompanying explanation, the more concrete expected results are:

- A novel approach for the semi-automatic API-based generators development called SAGED, which defines all the necessary steps to achieve the set goals.
- A Type-Based MCMC applied to the domain of code idiom mining, and utilized to approximate the non-parametric Bayesian PTSG method, with all necessary algorithms formally defined and explained in detail.
- A language-independent inference core, suitable for creating extensions for various programming languages.
- An extension of the inference core specialized for C#.
- A visual tool for curating inferred code idioms, allowing for the review and manipulation of results.
- An ABG that supports the manipulation of C# code.
- A code generator that extends the ABG with new API-based methods for selected code idioms.

1.5 Methods to be applied

To conduct research in a systematic, structured way, Design Science Research Methodology (DSRM) can be used to streamline efforts to derive concrete, generalizable results [52, 53]. Peffers et al. [54] defined what a DSRM research process should look like, as shown in Figure 1.1. The literature contains variations of the process flow, but the presented one is commonly used. It defines six nominal steps in the process. Although these steps are sequential, there is also support for iterative refinement. Furthermore, DSRM does allow research to have four different entry points (depicted in the lower half of the figure), from

which the research can proceed. In the next few paragraphs, we explain the steps introduced by the authors and how they relate to this thesis.

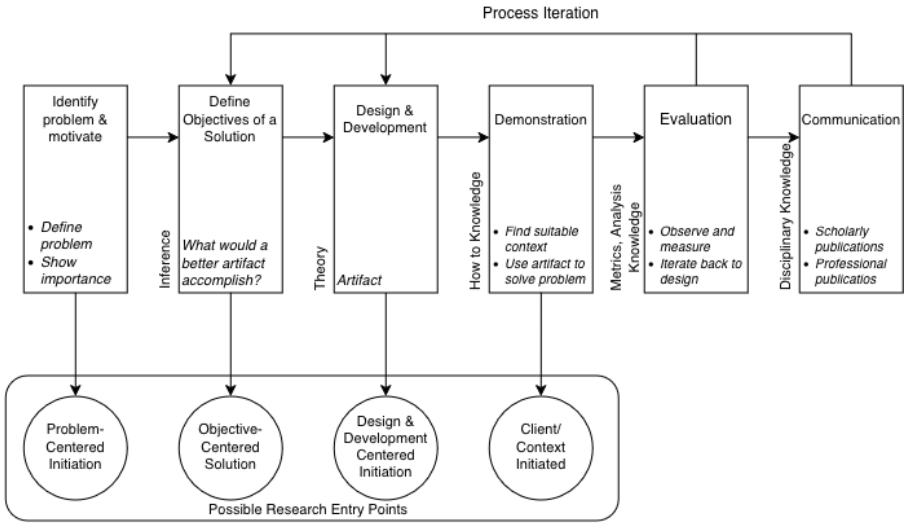

Figure 1.1: The DSRM process, adapted from [54] and [52]

The first step of the process is *Problem Identification and Motivation*. It necessitates the definition of the specific research problem and justification of the value of a solution. The second step is *Define Objectives of a Solution*, where the objectives of a solution are inferred from the problem definition. Section **1.5.1 Theoretical examinations** discusses how both of these steps are covered within this thesis.

The third step is *Design and Development*, which includes determining the artifact's desired functionality and its architecture, and then creating the actual artifact. Subsections **1.5.2 Development of a theoretical-conceptual model** and **1.5.3 Development of a technical solution** cover this step jointly, explaining the theoretical model used to create the solution, the solution's architecture, and its components.

Next, DSRM includes two steps to define the usefulness of the solution: *Demonstration*, where the efficacy of the artifact to solve the problem is demonstrated, and *Evaluation*, which involves comparing the objectives of a solution to actual observed results from the use of the artifact in the *Demonstration* step.

The way these two steps are covered by the thesis is explained in Subsection **1.5.4 Verification and evaluation**.

The last step of DSRM is *Communication*, which obliges researchers to communicate the problem and its importance, the artifact, its utility and novelty, the rigor of its design, and its effectiveness to researchers and other relevant audiences. Of course, the thesis itself serves this purpose, together with the related published papers and artifacts in the form of open-source solutions provided to the broader audience. Thus, this step is omitted from further discussion.

1.5.1 Theoretical examinations

The first part of the theoretical examinations begins by further exploring existing approaches, code-generation patterns, and tools for integrating generated and manually written code. Then, the research focuses on the approaches applicable to MDSE and M2T transformations, with an emphasis on those that support preserving manual changes through generation iterations.

After building a comprehensive theoretical background on the state of research on M2T transformations, a set of requirements that an ABG must conform to is defined to ensure the efficient use of ABGs for transitioning to MDSE. Then, existing ABGs are compared against the requirements to assess their suitability.

Similarly, the literature on the automatic discovery of repetitive, useful concept-embodying patterns in code is further explored. Requirements are formalized and used to compare relevant techniques.

1.5.2 Development of a theoretical-conceptual model

The development phase of a theoretical-conceptual model includes the design of an approach for semi-automatic development of ABGs. All steps of the approach are explained in detail. The steps include the code-mining process, where the application of Type-Based MCMC to this domain, as an approximation for non-parametric Bayesian PTSG, is explained, along with the included algorithms.

Besides the mining step, other steps include curation and the automated development of an ABG. The resulting ABG should conform to the requirements enumerated as part of the theoretical examination.

1.5.3 Development of a technical solution

The development of the ABG should reflect the requirements defined as part of the theoretical examinations. At this point, it seems sensible to use and adapt some of the existing ABGs with the API at a higher level of abstraction, making them a working basis. This basis includes extension points that can be used to automatically add new API methods.

The needed components reflecting the design of the approach, which help in further semi-automatic development of the ABG, are:

- Inference core, a component that finds existing code idioms in the codebase. The goal is to have this component be made in a (programming) language-independent way, and to allow future specializations for any language;
- Inferencer, a component which extends the inference core with the possibility to infer code idioms from codebase written in C# programming language;
- Visualizer, which allows practitioners to see found code idioms and navigate through the code, providing insight into repetitiveness and the context in which the found code idiom occurs;
- Handler, a component that allows practitioners to manipulate code idioms in a controlled manner. For example, a practitioner might want to join two adjacent found code idioms, and the tool should allow them to, but it should also take care that the resulting code idiom conforms to the syntax rules of the language;
- An ABG basis with API methods purely related to language concepts, with an architecture suitable for automated extension.
- A code generator that generates appropriate API methods and integrates them with the ABG basis.

Unit tests must cover all components. Moreover, as finding code idioms involves complex algorithms, special care is taken to ensure they are implemented correctly.

1.5.4 Verification and evaluation

All hypotheses stemming from the general hypothesis **H0** must be validated. Therefore, we need to evaluate whether the developed theoretical-conceptual

model and the technical solution contribute to affirmatively answering the general hypothesis.

To address the hypothesis **H1**, we have to verify that the steps involved in the semi-automatic identification of code idioms in a codebase indeed find code idioms relevant for future development. The evaluation of the steps should be both quantitative and qualitative. To mitigate bias, we must externally validate that the identified and curated code idioms are indeed relevant for future development. We can rely on external developers who possess general knowledge of the technical domain and can reason about their usefulness.

The scalability of the method used to find idioms needs to be assessed to address the hypothesis **H2**. As the identification process also includes manual curation, it is also required to evaluate its labor intensity.

The steps to perform this evaluation are as follows:

- Quantitative evaluation of the relevance of automatically inferred code idioms, both on datasets from the literature [51] and on datasets with known properties. The metrics to be measured are (i) **precision** — the percentage of inferred idioms found in the test corpora, (ii) **coverage** — the percentage of the test corpora covered by the inferred idioms, and (iii) **average length of the inferred idioms**, as explained in [14].
- Qualitative evaluation of the relevance of found code idioms, by measuring the number of idioms that require modification to be suitable for future development, along with external validation of their relevance.
- Measuring scalability of both the inference method and the curation process. To achieve this, the inference should be conducted on projects of varying sizes and properties.

Based on the relevant code idioms identified in the previous steps, we can now address the hypotheses **H3** and **H4** as follows:

- The identified code idioms are used to automatically derive API methods that extend the ABG basis.
- To compare the efficiency of using the extended ABG against the baseline (an ABG with the API defined purely in available language terms), we can build the same MDSE solution with both and compare the required time and the number of written lines of code.

Through these steps, by validating hypotheses **H1** to **H4**, we can deduce the answer to the general hypothesis **H0**.

1.6 The possibility of applying the expected results

Industry reports highlight various benefits of transitioning to MDSE [64]. However, many stakeholders remain unconvinced that the transition would help alleviate their problems to justify the costs [5]. Having the possibility of a more gradual transition is crucial in addressing stakeholders' concerns. Seamless integration of MDSE tools with the existing manual development workflows is essential for having a gradual transition. Central to this problem is the integration of manually written and generated code.

ABGs help solve this problem without introducing significant changes to the current architecture and setup. They allow practitioners to create MDSE tools customized to specific projects [13]. However, developing MDSE tools using ABGs can be laborious. The industry case study shows that developing code generators using ABGs takes significantly longer than building them using templates [12].

Provided that the outcomes of this research are successful, the possible applicability of the results is as follows:

- The resulting code-mining technique and related algorithms, implemented within the open-source component named Inference Core, can be extended to identify code idioms in codebases written in any programming language.
- A set of open-source tools specialized for the C# programming language can be utilized to find and visualize code idioms, providing insights into codebase repetitiveness. Additionally, tools for measuring the performance of the inference method can quantify this repetitiveness.
- The automated development of an ABG for C# can relieve practitioners from needing to understand ABG implementation details, thereby reducing the effort required to develop them.
- The SAGED approach, in general, can facilitate the adoption of the MDSE paradigm across various scenarios, including the automation of mature software projects that traditional MDSE approaches have not previously covered.

1.7 Thesis organization

The remainder of this thesis is organized as follows:

Chapter 2 lays out the theoretical foundations, including the core concepts of MDSE with an emphasis on code generation and the integration of hand-written and generated code, and the notion of code idioms along with techniques for mining them.

Chapter 3 reviews related work, discussing the benefits and challenges of ABGs, the state of the art in code idiom mining, and existing attempts to automate the development of MDSE tools.

Chapter 4 introduces the SAGED approach, its two phases and four steps, and the requirements that the resulting ABG must fulfill.

Chapter 5 presents the mathematical background of code idiom inference based on non-parametric Bayesian PTSGs and explains how Type-Based MCMC is adapted to this domain to process multiple syntax tree nodes simultaneously.

Chapter 6 describes RoseLib, the C# ABG developed in our previous research, which serves as the ABG basis.

Chapter 7 presents the reference architecture and the implementation of SAGED for C#, covering the language-independent inference core, its C# extension, the extension of RoseLib with generated API methods, and the practitioner tooling.

Chapter 8 evaluates the approach quantitatively and qualitatively, measures the efficiency of using the derived API, and discusses the results and threats to validity.

Finally, Chapter 9 summarizes the contributions and outlines directions for future work.

2 Theoretical Foundations

It is difficult to overstate how transformative software has been and still is for humanity as a whole. It is so ubiquitous and so impactful across all aspects of our lives that it is hard to find even one aspect left unaffected. Pretty remarkable considering that the first stored program was run just eight decades ago [65]. A lot has been achieved since then, and software entities are more complex for their size than perhaps any other human construct [66].

Of course, the process of creating software has been gradually improved over time. As the need for more predictable development grew, software engineering (SE) was recognized as a distinct field of engineering. And there was a reason for it, as it was early recognized that writing useful and efficient computer programs within the required time was difficult, leading the community to coin the term ***software crisis*** [67]. Furthermore, the community came to understand that just adding additional personnel to a project does not ease the problem [68].

SE is an engineering discipline concerned with all aspects of software production, from the early stages of system specification through to maintenance after the system has gone into use. It focuses on selecting the most appropriate method for a set of constraints and circumstances in which the development takes place, to achieve the required quality within schedule and budget [69].

And even though decades have passed since the SE discipline became established, and much work has been done to systematize and organize the development process, there are still many reports of software projects going wrong and of ***software failures***. Software engineering is criticized as inadequate for modern software development [69]. The sheer and increasing complexity of software systems has often been singled out as one of the main reasons for this [66, 70, 71, 69].

In his seminal paper called "No Silver Bullet - Essence and Accidents of Software Engineering" [66], Frederick Brooks Jr. split the complexities of software

development into two types: i) essential complexities that are part of the very nature of software, and ii) accidental complexities, those related to the tools we use. Then, he argued that no single technological advancement can lead to improvements in the development process at an order-of-magnitude scale, as long as accidental complexities are not ten times as difficult to manage as the essential ones.

Even though it was published 40 years ago, this paper sparked a debate that remains heated today. Perhaps it is better to say ***especially today***, with the advent of large language models (LLMs) and LLM-based tools that are being developed to help write and understand code [72, 73]. It remains to be seen how these advances will influence the future of SE, as it is unclear how stochastic LLM-based development approaches can be combined with more formal approaches to ensure predictability, as evidenced by mixed results in early studies [37].

This thesis focuses on an SE paradigm called Model-Driven Software Engineering (MDSE). The core idea behind MDSE is that there is a great difference between a problem domain and the domain of the software implementation - a problem-implementation gap. MDSE tries to mitigate and manage software system complexities by narrowing the problem-implementation gap. The narrowing is achieved by raising the level of abstraction and automatically deriving the implementation [1]. In the next section, we will explain the main principles of MDSE, the foundations for the thesis's contribution. As this thesis leverages advancements in the field of code idiom mining to advance MDSE, the second section of this chapter introduces the necessary theoretical foundations of this field.

2.1 Model-Driven Software Engineering

MDSE provides a comprehensive vision for software development that treats models as primary artifacts used for all aspects of software production. In other words, MDSE aims to manage software development complexity through models, abstracting away concepts that are not important to the problem at hand. The running software is then derived from models using automation.

MDSE has made incredible contributions to leverage abstraction and automation in almost every area of software and systems development and analysis. In many domains, including railway systems, automotive, and business process engineering, models are key to success in modern software engineering processes [58]. The success of MDSE is not limited to business domains but

extends to technical domains, such as reverse engineering [74], testing [75], and developer operations [64].

The success has been affirmed by numerous claims and empirical evidence of the benefits of using modeling and MDSE. Torchiano et al. [76] conducted a survey of the Italian industry to investigate the relevance, benefits, and problems of using modeling. Their findings indicate that adopting simple modeling yields common benefits (better design support, improved documentation, better maintenance, and higher software quality). At the same time, model-driven techniques make it easier to achieve improved standardization, higher productivity, and platform independence.

Burden et al. [77] state that a key benefit of MDSE is that it allows domain experts to be directly involved in the development process because they understand the model. According to them, direct involvement leads to a higher-quality product. This is confirmed by a recent study in which Alfraihi and Lano [78] conducted a survey. Participants agreed that MDSE offers clear benefits, especially improved software quality and communication. However, these studies, as well as others such as [5], report problems predominantly associated with tool immaturity and the complexities of integrating them into the development workflow.

2.1.1 Positioning MDSE

Modeling is considered to be one of the most essential processes of the human mind [79]. Thus, inherently, it has always been a part of efforts to build software. However, not all fields of SE necessarily rely on formal models. Different fields of software engineering can be distinguished and interrelated based on their use of models [3]. Figure 2.1 depicts the four main modeling areas of SE. Note that this thesis includes the word *software* and the letter S in these acronyms to make them more specific, but they are often omitted.

The broadest field is Model-Based Software Engineering (MBSE), which refers to the use of models in the software engineering endeavor. Here, models are not necessarily a key part of software development, although they are important [17]. For example, under MBSE, a class diagram may be created solely as a specification - a blueprint for the development team [80]. MDSE is stricter in this regard, as it emphasizes the role of models to the point where they are considered key engineering artifacts, and it requires them to be formal so that working software can be derived [17].

While MDSE covers different parts of the software lifecycle such as requirements engineering [81], development, and deployment [64], Model-Driven Soft-

ware Development (MDSD) focuses on the software development process, and is considered a subset of MDSE [17, 3]. Lastly, MDA, or Model-Driven Architecture, is considered a specific style of MDSD [22, 82]. MDA is an approach to software design, development, and implementation spearheaded by the Object Management Group (OMG). MDA provides guidelines for structuring software specifications that are expressed as models, with the main goal of separating business and application logic from underlying platform technology [83].

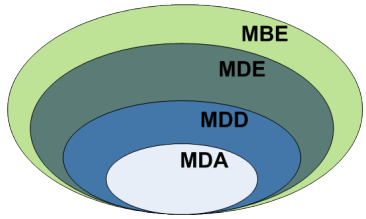

Figure 2.1: Different MD* acronyms and their relation, taken from [3]

The contributions of this thesis lie in MDSE. Thus, the following subsections explain the fundamental concepts of MDSE. Subsection 2.1.2 first briefly explains the core concepts of the paradigm. Then, it focuses on the modeling part of MDSE. Afterward, the automation part is analyzed in more depth, as it establishes the foundation for the primary contributions of this thesis. As a continuation, code generation techniques are explained in Subsection 2.1.3. Lastly, as we cannot expect the automation to cover 100% of development needs, Subsection 2.1.4 discusses how manually written code gets integrated with the automatically derived artifacts.

2.1.2 The Core Concepts

Figure 2.2, adapted from Brambilla et al. [3], illustrates the primary concepts of MDSE across two fundamental dimensions of software development. Conceptualization refers to the act or process of forming an idea or principle in mind [84], the output of which is a model: an abstract representation of a system or process that enables predictions and inferences to be made [18]. Implementation, the second dimension, concerns mapping models to existing or future running systems [3]. In essence, it concerns how executable software is derived from models.

Focusing on the first column of the figure, models serve as the primary arti-

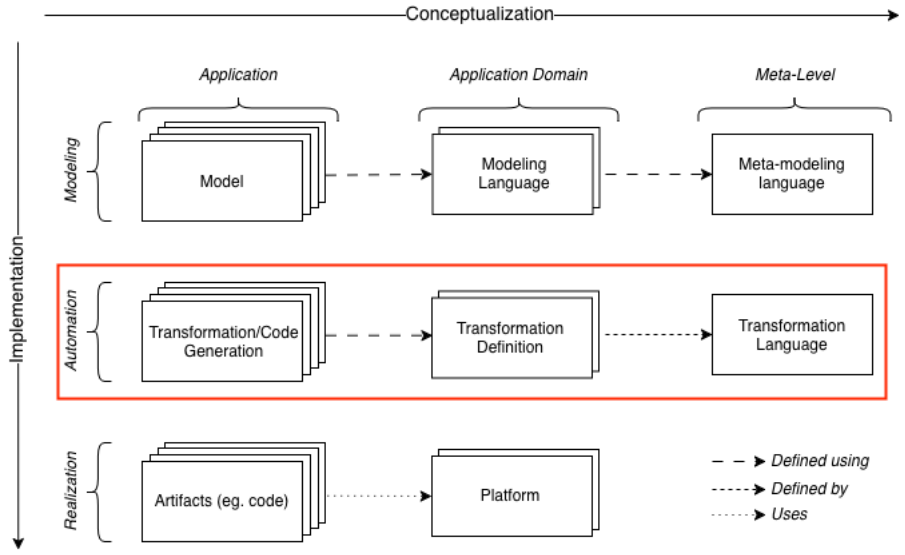

Figure 2.2: Conceptualization and Implementation, adapted from [3]. The automation row is marked red to emphasize that contributions lie in this area of MDSE.

facts to describe the target application. If the problem domain is complex, the narrowing of the problem-implementation gap is addressed through a variety of models that describe complex systems at multiple levels of abstraction and from multiple perspectives [1]. These models can be transformed into one another, and MDSE aims to develop, maintain and evolve software by performing model transformations [19]. When the transformation finally generates code artifacts, we call that process *code generation*, and that type of transformation *model-to-text transformations* [20].

Let us now analyze how items in this first column map onto other concepts following the conceptualization axis. The model itself needs to be defined using a formal language – a modeling language. The modeling language contains concepts of the application domain, which can be both technical and business in nature. Furthermore, transformations also need to be defined in a way that allows their execution. Later in the text, we will discuss both the modeling languages and transformations in more detail. Concerning transformations, most of the attention will be given to M2T transformations and code generation, as

this is where the contributions of this thesis lie.

Although it nicely presents the core concepts, this figure represents a simplified, most straightforward scenario for deriving the implementation in MDSE. The reality is that, for example, in some scenarios, no code needs to be produced because an engine can interpret the model directly. Furthermore, when there is a need to produce code, it is not necessarily a simple process. Sometimes, the produced code needs to follow certain rules for integration with an existing codebase, or it needs to be woven with manually written code. As further development of existing software systems and integration into existing code-first development processes [13] are the focus of this thesis, we will need to explain these intricacies in detail later in the text.

Modeling and Meta-Modeling

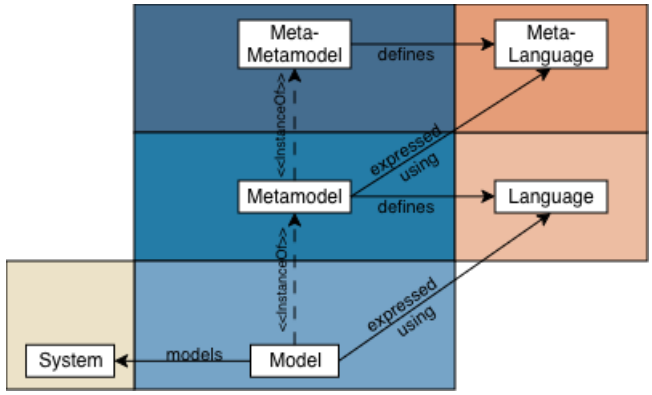

Figure 2.3: Meta-modeling, adapted from [18]

The first row of Figure 2.2 maps nicely to Figure 2.3 from [18]. To understand it, we first need to clarify what we mean by a model. According to Stachowiak [85], a model must possess three features: i) a mapping feature, meaning that it is based on an original (the system); ii) a reduction feature, meaning that it reflects only a relevant selection of the system's properties; and iii) a pragmatic feature, meaning that it can be used in place of the system with respect to some purpose. When modeling a system, we can employ different types of models, each describing the system from a different angle – a viewpoint [86].

A model, however, cannot be expressed in a vacuum. In the very act of describing a system, we commit to a set of concepts and rules for combining them – that is, to a modeling language. This language is itself captured by a model: a *metamodel*, defined simply as a model of models [2]. The metamodel makes explicit which constructs are available and how they may be combined, so that any model conforming to it is a valid sentence in the language it defines. In other words, the metamodel provides the language in which we write models of the system.

This relationship is recursive, because a metamodel is, after all, still a model, and therefore needs its own language to be expressed. That language is provided by a *meta-metamodel*, which stands one level higher on the conceptualization axis and supplies the concepts used to write metamodels. The recursion does not continue indefinitely: it terminates at a level expressive enough to describe itself, so that the meta-metamodel conforms to itself and no further layer is required [18]. Reading the top row of Figure 2.2 along the conceptualization axis thus reveals a ladder of abstraction in which each level defines the language used to express the level below it – the meta-metamodel yields the language for writing metamodels, and a metamodel in turn yields the language for building models of a concrete system.

This ladder corresponds to the four-layer metamodeling architecture standardized by the OMG [2], which names these layers **M3**–**M0**. The meta-metamodel sits at the topmost layer, **M3**, occupied by the Meta Object Facility (MOF), which is self-describing and therefore closes the recursion. A metamodel resides at **M2**, the level of modeling languages such as UML. A model of a concrete system lives at **M1**, conforming to its M2 metamodel. Finally, **M0** holds the actual runtime system that the M1 model represents.

What the metamodel defines is the *abstract syntax* of a modeling language – its concepts and the rules for combining them, independent of any notation. To actually read and write models, this abstract syntax has to be paired with a *concrete syntax*, the notation through which the language is presented to its users [3]. A concrete syntax can be graphical (a Graphical Concrete Syntax, GCS), where models are drawn with diagrammatic symbols such as boxes and arrows, as in UML class diagrams, or textual (a Textual Concrete Syntax, TCS), where models are written as text. The same abstract syntax may be equipped with several concrete syntaxes, allowing the same underlying model to be rendered graphically, textually, or both [3].

Model Transformations

In software development, data, structured as models, are manipulated on a daily basis through systematic model manipulation operations. Automating such operations, typically in the form of model transformations, can reduce the time-to-market of project development and improve its quality [20]. Let us now introduce more concrete definitions of model transformation.

Kleppe et al. [87] provided the following definition: A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. Finally, a transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language. Mens and Van Gorp [19] suggest that a model transformation should also be applicable to multiple source models and/or multiple target models.

Building upon these definitions, we can categorize model transformations based on source and target artifacts. Given its role in MDSE, a model can be considered a default source, that is, an input to a model transformation. There can be multiple intermediate steps along the implementation axis, which refine the model, and are often automated as model-to-model (M2M) transformations. The last step consists of a model-to-text (M2T) transformation that takes these low-level models and generates a textual output, i.e., the final code, as a result [20]. To emphasize once more: we call this M2T process *code generation*.

Furthermore, more complex MDSE workflows can also include text-to-model (T2M) transformations, which are employed in the context of reverse engineering, and text-to-text (T2T) transformations, utilized for program translation [20, 36]. While some authors prefer to reserve the term *model transformation* strictly for M2M processes (e.g., [8]), Mens and Van Gorp [19] discuss that a model can range from abstract analysis representations of the system, over more concrete design models, to very concrete models of source code. Hence, from this broader perspective, model transformations include all these combinations of transformations.

Another classification of model transformations is based on the metamodel: transformations with source and target domains conforming to a single metamodel are referred to as endogenous or rephrasings, whereas transformations with different source and target metamodels are referred to as exogenous or translations [88]. Examples of endogenous model transformations include model refactoring, optimization, normalization, and examples of exogenous include

synthesis (e.g., code generation) and reverse engineering [19, 89].

Two things have to be considered when designing a model transformation: how data is extracted from a source model, and how the target gets assembled. The contribution of this thesis is tied to the domain of M2T transformations; more specifically, how the target code gets generated. There are different patterns for extracting data from the source model in this scenario, such as visitor-based and metamodel-based [88, 90], all compatible with how code is generated in our approach.

2.1.3 Code Generation

In more general terms, code generation has been used since the 1950s [91], when the first compilers were built. In fact, most standard techniques used in compiler construction can also be applied directly to model-based automatic code generation [4]. However, in MDSE, we mostly think of code generation from the perspective of automatic programming. The underlying principle of automatic programming is that a user defines what they expect from the program, and the program should be generated automatically by software without any assistance from the user [21].

There are many different ways to generate code. Traditionally, it is considered that there are two fundamentally different approaches to generating code: template-based code generation and API-based generation, and that all other code generation mechanisms and patterns stem from these two [8]. Recently, LLMs have also been considered for this specific use case, with promising results [92, 93], but it is evident that more research is needed to achieve the precision of deterministic approaches. Thus, in the following text, we only analyze and compare the two fundamental approaches.

Template-based Code Generation

Other than being fundamental, template-based code generation is one of the most commonly used approaches for implementing M2T transformations. It fits nicely into the straightforward implementation flow of the MDSE (presented in Figure 2.2). Per the definition from [21], a synthesis technique is a template-based code generation (TBCG) if it consists of a set of templates specified in a formalism, where the specification of a template is based on a design-time input such that, when executed, it reads a runtime input to produce a textual output.

Figure 2.4, adapted from [21], shows the general components along with their inputs and outputs. The main component is the template. Templates allow

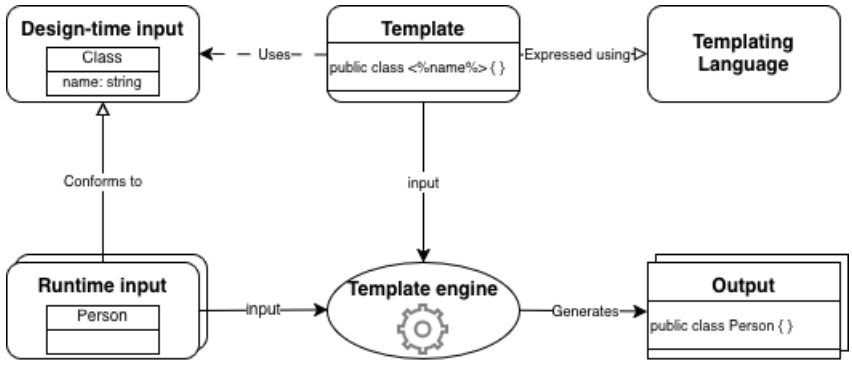

Figure 2.4: Template-based code generation, adapted from [21]

users to specify exactly what they expect the output to be. They are written in a templating language, formally specified so that they can be interpreted by a template engine. It has a static part, text fragments that appear in the output "as is". It also has a dynamic part embedded with markings, splices of meta-code that encode the generation logic. One reason for the popularity of template-based code generation is that the template itself resembles the output to be created.

The dynamic part — markings — uses design-time input. In MDSE, this input is the metamodel. A model, that is, data that conforms to the metamodel, is the runtime input to the template engine. So, based on the data and the template, the template engine can produce the output.

The most prominent examples of TBCG tools are: Eclipse XPand [94], Acceleo [95], which implements the MOFM2T transformation language [44], FreeMarker [96], Microsoft's T4 [97], Jinja [98], and others. When thinking about the limitations of these tools and TBCG in general, note that the output's syntax correctness is rarely guaranteed, as most tools are general-purpose and agnostic to the syntax of any programming language. Additionally, how the generated code is integrated into any codebase is out of scope for these tools.

API-based Code Generation

As stated in the Introduction, libraries and frameworks that offer an API for making precise modifications to programming code, while maintaining its syntax correctness, are known as API-Based Generators (ABGs), and they implement

the fundamental code generation pattern of the same name [8]. Examples include JavaPoet [28], a Java API for generating Java source files, and the Syntax Tree API of Roslyn [10]. There are also ABGs that can rewrite bytecode, such as ASM [99] and ByteBuddy [100]. The contributions of this thesis are related to source code, so in the following text we are only concerned with ABGs that generate source code.

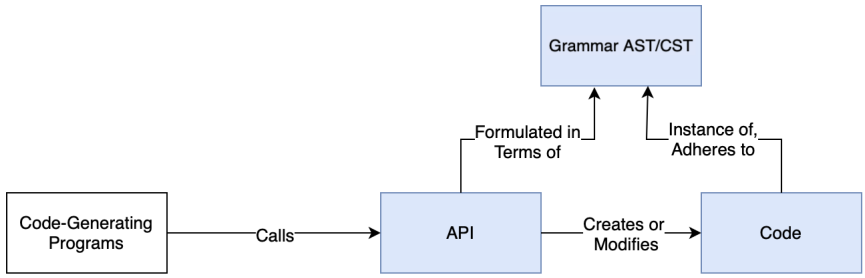

Figure 2.5: API-based code generation, adapted from [8]

Figure 2.5 shows the general concepts related to an ABG. When creating M2T transformations, practitioners write programs that use the API of an ABG. This API is formulated in terms of the grammar of the programming language it serves to create or modify. Generated code, of course, needs to adhere to the grammar. In practice, not all ABGs can really modify the code. Some of them, such as JavaPoet [28], can only generate new code. Others, such as JavaParser API [9] and Roslyn's SyntaxTree API [10], can perform full modifications of an existing codebase, allowing for precise changes to specific parts of the code while maintaining syntax correctness.

In Figure 2.6, under a), you can see a method named *sayIt*, which we would like to add under the *HelloWorld* class. Under b), you can see how the JavaParser API allows us to try to find this class and, if it is present, build the *sayIt* method part by part and add it. This is a significant difference from using templates: the built code follows the grammar rules, as the API design enforces them. Furthermore, the API allows us to integrate the created code.

2.1.4 Integrating Hand-Written and Generated Code

So far we have only discussed MDSE concepts from a simplified perspective, where deriving the implementation is straightforward and unidirectional. The

```
package com.example.helloworld;

public class HelloWorld {
  public void sayIt(String[] names) {
  }
}
```

a)

```
cu.getClassByName("HelloWorld").ifPresent(helloWorldClass -> {
  Parameter param = new Parameter(
    new ArrayType(
      StaticJavaParser.parseClassOrInterfaceType("String")
    ),
    ”names");
  BlockStmt body = new BlockStmt();

  MethodDeclaration mainMethod = new MethodDeclaration()
        .setName(”sayIt")
        .setModifiers(Modifier.Keyword.PUBLIC)
        .setType("void")
        .addParameter(param)
        .setBody(body);
  helloWorldClass.addMember(mainMethod);
});
```

b)

Figure 2.6: An example of using JavaParser API; the code with a green background under a) gets created and integrated by the code under b).

reality can be much more complicated. Brown et al. [22] discuss different uses of models in assisting software practitioners to create running applications, and the relationship between the models and the code. Note that not every case presented in Figure 2.7 would be classified as MDSE. A term that has not yet been introduced is Roundtrip Engineering, where changes can be made to either the code or the model, and the two are then reconciled [101]. On the surface, Roundtrip Engineering provides the most flexibility of all the cases. However, it is not always possible or sensible to model every specific part of a system or its behavior, nor to map every code variation back to a model. Furthermore, it is also relatively difficult to implement compared to the other cases.

Thus, most MDSE solutions aim for the ”model-centric” or the ”model-only” cases. We will now focus on the ”model-centric” case, where the model gets translated to code. To support having case-specific code manually written and added to the codebase, MDSE solutions have to provide mechanisms for such partial code generation [3]. In these partial code generation scenarios, the code generation process needs to be crafted so that it does not overwrite manually written changes in future code generation cycles. Various mechanisms have been proposed to resolve this problem [6], such as protected regions and the generation gap, the main premise being that hand-written and generated code are kept separate and then integrated afterward.

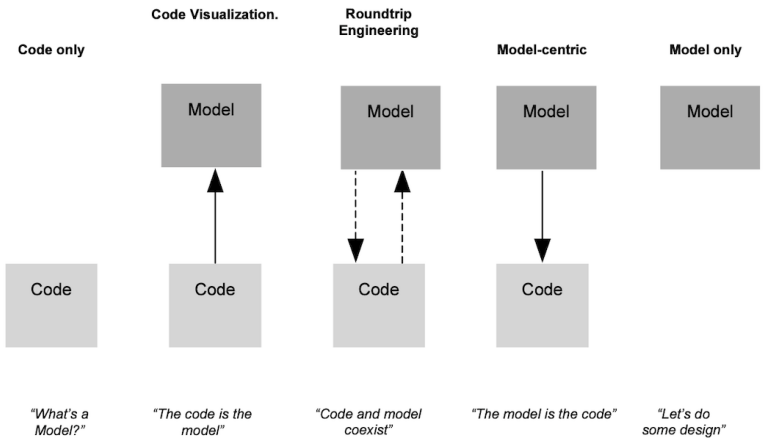

Figure 2.7: Modeling spectrum from [22]

Imposing such a structure on a project might not be received well by stakeholders [7], especially if the project started in the "code-only" case and there is an initiative to move towards the "model-centric" case [12]. It is also possible to rely on Version Control Systems to integrate hand-written and generated code, but then developers have to perform the integration manually whenever there is a merge conflict. As part of our broader research [12, 13, 11], our group has worked on a workflow that revolves around semi-automatic integration, built upon API-based generators.

2.2 Code Idioms Mining

2.2.1 Code Idioms

Code idioms represent small code fragments that occur frequently in the codebase and have a single semantic role [14]. In other words, code idioms are pieces of code, patterns of various complexities, that software engineers repeatedly use on different occasions with specific intent. They can contain metavariables, which denote places in the code that can vary per occurrence. Another way of describing code idioms would be to say that they represent frequently recurring subtrees of syntax trees [23], where ending nonterminals map to metavariables.

Examples of code idioms are shown in Figure 2.8: a) an idiom used for reading from a file and b) an idiom for opening a connection to a database. Code idioms are particularly interesting for automating the extension of ABGs because they capture how a programming language, framework, or library is used in an existing system. Additionally, the metavariables naturally map to the parameters of the API methods.

```
if (!File.Exists(path))
{
    using (FileStream fs = File.Create(path))
        $Body
}
```

a)

```
using (SqlConnection connection =
       new SqlConnection($Name))
{
    try
    {
        connection.Open();
        $Body
    }
    catch (Exception ex)
    {
        logger.Error(ex.Message);
        throw ex;
    }
}
```

b)

Figure 2.8: Code idioms that contain metavariables denoted with a $ sign.

The process of automatically identifying a set of code idioms given only a corpus of previously written idiomatic code is called code idiom mining [14]. Mining code idioms has been used in different settings, from helping developers with better suggestions while programming [14], detecting correct or incorrect code idiom usage [15], or finding them to optimize performance [102]. However, no papers trying to use them to automatically develop API-based generators have been uncovered during our literature review.

2.2.2 Mining Techniques

Given that code idioms are patterns of code used repeatedly in a codebase, we have to distinguish among different, perhaps simpler, techniques that could be used in this scenario. The most prominent technique for uncovering patterns in a codebase is code clone detection [24]. However, code clone detection methods have an inherent problem: they tend to find the largest fragment rather than the most useful one [14]. Furthermore, they try to uncover contiguous blocks of code that are not necessarily syntactically correct.

Another class of methods that could possibly be used to find patterns in code is API mining, where techniques aim to automatically extract a set of API patterns, which are lists of API methods that are usually used together, and that together characterize how an API is used [25]. API mining is interesting because techniques try to uncover how an API is used when documentation is lacking. However, as stated by Allamanis and Sutton [14], idiom mining differs markedly from API mining because idioms have syntactic structure.

The techniques we aim to utilize are those that address the syntactic structure of code idioms while striking a balance between repetitiveness and size; that is, techniques that dissect syntax trees to uncover recurring subtrees. The two main classes are Frequent Tree Mining [26] and probabilistic methods [14, 103]. We discuss both in Chapter 3. Furthermore, because this thesis also makes contributions in this area, Chapter 5 discusses in depth a method called Type-Based MCMC.

3 Background and Related Work

In this chapter, we will discuss the benefits of ABGs and the challenges associated with using them as a foundation for the development of MDSE tools. Next, we will explore the process of inferring useful code idioms from the codebase. Finally, we will present currently available research on automating the development of MDSE tools using various artificial intelligence techniques.

3.1 Benefits of ABGs

From the perspective of an MDSE practitioner, the most effective way to transition from manual development to the MDSE approach is to modify the current workflow to incorporate MDSE practices and update or rebuild the existing architecture to support it. However, this transition comes with costs and risks. While there are success stories [60, 61], current research indicates that stakeholders may doubt the benefits of this change and resist making significant modifications to their development processes [5]. The successful adoption of MDSE depends on the organization's willingness to adapt [61]. In our experience [12], it is beneficial to build stakeholder trust by demonstrating the effectiveness of MDSE without immediately imposing significant changes. Therefore, when aiming for a gradual transition, the implementation of MDSE is constrained by the existing environment, processes, and architecture.

In a gradual transition, manual changes must be supported until the transition is complete. Therefore, it is important to consider how manual changes are preserved between generation cycles. Most authors suggest that hand-written and generated code should be kept and maintained separately to prevent over-

writing manual changes by a code generator. Their integration can be achieved using mechanisms such as generation gap, partial classes, protected regions, and others [6]. However, there may be cases where the existing architecture or code artifacts do not allow for the use of these mechanisms. Additionally, stakeholders may request the ability to retain flexibility and directly change the generated code.

One solution to prevent overwriting of hand-written code during the code generation process is to rely on Version Control Systems (VCS). Bernaschina et al. [7] present a model and code co-evolution workflow that preserves the code changes (delta) between two consecutive generations by treating the code generator as a Virtual Developer. Newly generated code is then combined with the existing codebase by relying on the merge functionality of Git. However, relying on VCS mechanisms shifts the responsibility to developers performing the merge, as VCSs are not content-aware.

If we want the MDSE tool to handle the load, its code generation process could be based on ABGs. ABGs allow for precise changes to specific parts of the code while maintaining syntax correctness, which helps prevent mistakes that could disrupt the development process and damage stakeholders' trust. In one of our papers, we demonstrated how ABGs can enable the seamless integration of generated and hand-written code, automating the development of a large-scale Software Product Line (SPL) in industrial settings [12]. By combining precise code changes with automated conflict detection and handling, integration was automated to a greater extent than by relying solely on VCSs.

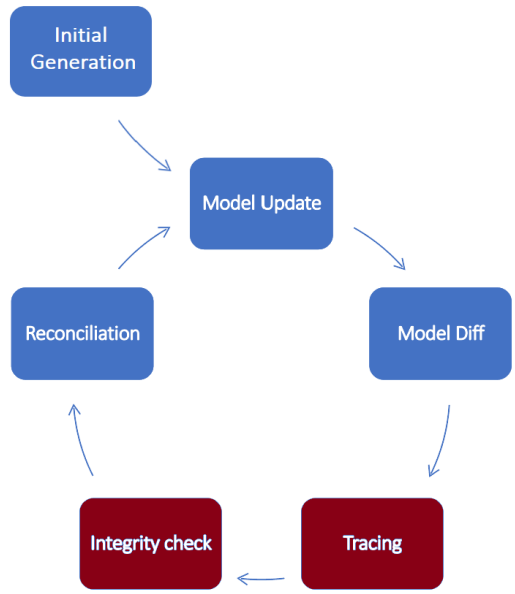

Figure 3.1: Seamless MDSD Workflow, adapted from [11]

Figure 3.1 shows the Seamless Workflow we devised as a part of our broader research, enabled by using ABGs [11, 12, 13]. The workflow starts

by generating initial code parts based on a model. If there are any changes made to the model that would require new code to be generated, we: i) compare the old model version with the new one to find the differences at the Model Diff step; ii) try to trace the code elements generated in the previous generation cycle; iii) if they are found, check whether they were altered manually at the Integrity Check step; iv) if possible, reconcile the model and code automatically. If automatic reconciliation is not possible, manual assistance from the user is needed. In this workflow, ABGs facilitate the Tracing, Integrity Check, and Reconciliation steps. However, using ABGs comes with its own set of hurdles.

3.2 ABG challenges

When tool developers want to implement MDSE tools to handle the integration of generated and manually written code, they face a new set of challenges. The effectiveness of ABGs in different scenarios and their suitability for complex use cases depend on the abstraction level of their API [8]. Existing ABGs typically offer abstractions based on the syntax of a target language, resulting in a relatively low abstraction level. General-purpose programming languages often have complex syntax, making the APIs provided by these ABGs difficult to use.

An example of an ABG with a lower level of abstraction is the Roslyn Syntax Tree API. This API enables manipulation of syntax trees, which is useful for code generation. Figure 3.2 demonstrates the complexity of the grammar in C#, showing how the API is used to add nodes for the `Console.WriteLine("Hello World");` statement to the syntax tree of a program. The resulting syntax subtree containing the inserted statement nodes is shown in Figure 3.3. The nodes colored blue are instances of concepts defined by the C# grammar. Because the API is based on these concepts, it can be challeng-

```
ExpressionStatement(
  InvocationExpression(
      MemberAccessExpression(
        SyntaxKind.SimpleMemberAccessExpression,
        IdentifierName("Console"),
        IdentifierName("WriteLine")))
  .WithArgumentList(
     ArgumentList(
       SingletonSeparatedList<ArgumentSyntax>(
         Argument(
           LiteralExpression(
             SyntaxKind.StringLiteralExpression,
             Literal("Hello World")))))));
```

Figure 3.2: An example of using the Roslyn ABG.

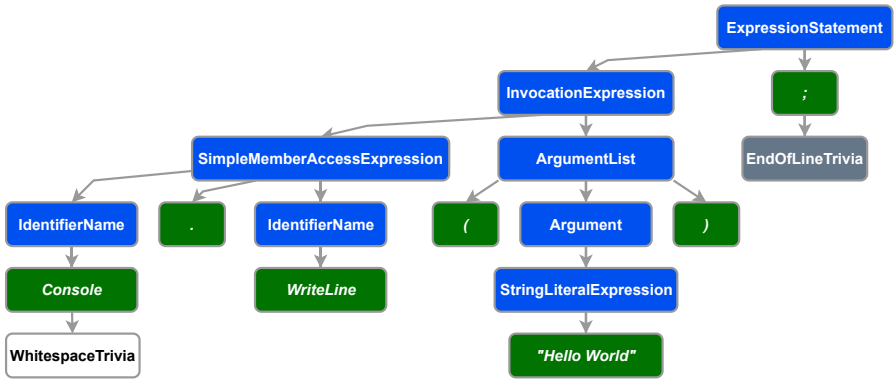

Figure 3.3: An example of the complexity of GPL syntax: a syntax subtree representation of the *Console.WriteLine("Hello World");* method invocation

ing to use for code generation since every grammatical detail must be specified. Additionally, the large number of syntax-related concepts to learn can hinder its adoption. However, this low-level API is necessary for comprehensive support in altering syntax trees.

Projects such as JavaPoet and KotlinPoet offer more concise and easier-to-use ABGs. These projects focus on specific concepts related to the syntax of the programming language they are designed for. They provide an API that facilitates code navigation and the creation of complex constructs. An example of using the JavaPoet ABG to define a *HelloWorld* class, with a *main* method and a `System.out.println("Hello World");` statement, is displayed in Figure 3.4. Instead of defining code constructs step-by-step using syntax tree nodes, whole language-related constructs are defined using the supported API methods. However, these kinds of ABGs often do not provide comprehensive support for altering code.

There are hybrid solutions such as Testura.Code [29], which builds upon Roslyn's Syntax API. It serves as a wrapper, trying to mitigate the complexities of working with the Syntax API. Whenever a user needs to manipulate code in a way not supported by Testura.Code, they can always continue to use the plain and comprehensive Syntax API. This thesis builds upon a similar hybrid ABG named RoseLib, which we developed in our previous research and describe in detail in Chapter 6.

Although these concise ABGs are more user-friendly, the range of concepts they support is defined and fixed by their authors. Expanding this range can make an ABG more relevant to a given problem domain, enabling the manipulation of code fragments related to additional concepts. Expanding the range of concepts that RoseLib supports would make it even more relevant to a given problem domain. Doing so manually, however, requires becoming familiar with the existing code, identifying recurring concepts worth including in the API, and designing appropriate API methods. The following sections explore existing literature, which suggests that this process could be semi-automated using code idiom mining.

```
MethodSpec main = MethodSpec.methodBuilder("main")
    .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
    .returns(void.class)
    .addParameter(String[].class, "args")
    .addStatement("$T.out.println($S)",
                  System.class, "Hello World")
    .build();

TypeSpec helloWorld = TypeSpec
    .classBuilder("HelloWorld")
    .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
    .addMethod(main)
    .build();
```

Figure 3.4: An example of using the JavaPoet ABG for defining a class, a method, and a statement.

3.3 Code Idioms Mining

Techniques from data mining, frequent tree mining, and probabilistic grammars can be applied to mine idioms from source code repositories [104]. Frequent tree mining algorithms [105, 26, 106, 107] have been extensively studied in the literature. They focus on finding subtrees that often occur in a database of trees. These algorithms tend to return small and generic code fragments [27]. There is a straightforward explanation for why this is the case. Given a fragment T, it can always be made more frequent by removing one of the leaves, even if it is strongly correlated with the rest of the tree. Pham et al. [27] describe a method that tries to extract larger and more useful fragments by applying several constraints on the FREQT [108] frequent tree mining algorithm.

A method that inherently tries to infer larger idioms can be used to avoid constraining algorithms in such ways. The method presented by Allamanis and Sutton [14] uses statistics to address the problem of inferring larger and more significant code idioms. The presented method belongs to a class of pattern-mining probabilistic models of code. It aims to discover a finite set of human-interpretable patterns from source code without annotation or supervision [31]

by inferring a Probabilistic Tree Substitution Grammar (PTSG) that is good at explaining the codebase. The method is state-of-the-art in code idiom mining from syntax trees, but relying on the syntax to retrieve concepts has limitations. The resulting code idioms tend to be shallow [51]. There have been attempts to improve the method further for different use cases.

Allamanis et al. [51] experimented with enriching ASTs with variable mutability and function purity information to mine code idioms, which better explain the loops found in code. The information is added either manually or retrieved through dynamic analysis using tests. Likewise, Sivaraman et al. [103] rely on augmenting ASTs using dataflow information to infer better idioms found in imperative code.

However, within our research, we are not solely interested in code idioms related to imperative code or loops. We will rely purely on mining code idioms from syntax trees using the method explained by Allamanis and Sutton [14]. Enhancements similar to those that utilize the enrichment of syntax trees with additional information will remain an opportunity to be considered for future work. To create a practical implementation of the method, the authors suggest using Type-Based MCMC [32]. However, to the best of our knowledge, there is currently no explanation in the existing literature on how to practically apply the Type-Based MCMC method to the domain of mining code idioms. In this thesis, we have provided a detailed explanation, the necessary steps to perform it, and its open-source implementation.

3.4 Automating the Development of MDSE tools

To the best of our knowledge, there is currently no research in the literature on automating the development of ABGs. Although there are papers that discuss transitioning to a model-driven approach, they mainly focus on a complete transition and do not consider the specific requirements we outlined for automating the development of ABGs, as explained in Section 4.2. Nevertheless, papers that contribute to the area of automating code generation in MDSE have to be investigated, as the techniques they employ may have potential applicability to our use case.

Note that in the last few years, much attention has been given to the automation of M2M transformation creation. Different machine learning techniques have been applied to automate this process [109, 110, 111]. Although

much inspiration can be drawn from it, code has its specifics, and these M2M approaches are therefore excluded from the following analysis.

Chénard et al. [33] explore techniques that facilitate the migration of systems to a model-driven environment using Model-Driven Architecture [112]. They address the challenge of discovering a Platform Description Model (PDM) from a system's implementation. One aspect of this challenge involves identifying a set of QVT (Query/View/Transformation) transformation templates, which are parameterized models that capture the source code of the system's implementation platform. Their approach takes the source code of a legacy system, its platform-independent model, and the UML model of the implementation platform as input. They utilize code clone detection to identify repetitive sections in the existing code and create the templates.

Similarly, Rost [34] suggests leveraging code clone detection to extract a DSL and associated generator templates from multiple reference applications. The goal is to develop a tool that reduces the initial effort required for creating an MDSE infrastructure. We also considered using code clone detection methods to identify repetitive code sections. However, we opted to use the inference of PTSGs instead, as we believe it is a better fit for our use case.

Lano and Xue [35, 36] present a solution that utilizes symbolic artificial intelligence techniques to streamline the development of code generators. Their focus is on inferring rules specified in a textual concrete syntax transformation notation called *CSTL* [113]. These rules define how parsed subtrees from one textual language can be translated into subtrees of another textual language, given specific conditions. Their approach aims to automatically derive *CSTL* code generation rules based on a dataset of corresponding text from source and target languages, a process they refer to as *code generation by-example*. However, their work primarily focuses on automating program translation rather than automating the further development of existing programs. This means that the source and target in their approach are different, whereas in our approach, they are the same. Furthermore, their work requires labeled examples, which are not necessary when utilizing unsupervised machine learning.

Artificial Neural Networks (ANNs) have also been used for the inference of transformations. Burgueño et al. [20] present a method that utilizes Long Short-Term Memory (LSTM) ANNs to infer different types of model transformations. The focus of the authors is to train their networks to take the UML class model as input and generate the structural part of the corresponding Java code. These neural networks learn rules based on the shared concepts of object-oriented programming in both languages, as well as project-specific rules such as primitive data type conversions.

Advances made with another type of ANN — Large Language Models (LLMs) — have spurred discussion about the possibilities for their usage across the scientific community. It is becoming evident that LLMs may change how we develop software in many ways. However, it is also clear that the traditional computer science formal and precise analytical methods often do not apply to using LLMs [38]. Initial reports regarding automating the development of software systems using LLMs confirm this statement. Acher and Martinez [37] experimented with synthesizing variants into SPLs using GPT-4 [114]. They tried using GPT-4 on five different cases with mixed results. For example, they successfully generated an annotated SPL out of Java variants. However, the created code templates contained syntax errors, preventing the whole SPL from being compiled.

One drawback of using ANNs is their black-box nature, where learned rules cannot be easily modified. In contrast, TSGs are relatively easy to comprehend, and resulting code idioms can be easily altered.

4 Overview of SAGED

This chapter introduces SAGED, a novel approach for simplifying the process of creating code-generating programs tailored to specific solutions. It consists of two phases: *Phase 1 (P1) — Concept Discovery* aimed at identifying code idioms that are good candidates for code generation, and *Phase 2 (P2) — ABG Extension* for automating the creation of API methods based on the identified idioms. A discussion of the approach is given in the following sections (see Figure 4.1).

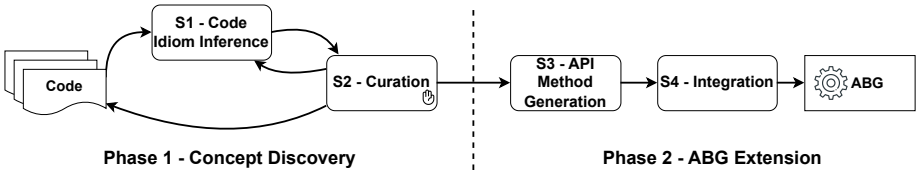

Figure 4.1: Overview of the SAGED approach

4.1 Phase 1 — Concept Discovery

The first step of the SAGED approach, belonging to the *P1 — Concept Discovery* phase, is *Step 1 (S1) — Code Idiom Inference*. The method we use to infer code idioms from an existing codebase is an approximation of a technique known as non-parametric Bayesian PTSG [39, 16]. To support this approximation, we have implemented and adapted a variant of the MCMC method called Type-Based MCMC [32]. We chose this method because it is suitable for our use case for several reasons: (i) it is a form of unsupervised machine learning,

allowing us to find patterns in source code without annotation and supervision; (ii) the method is based on a statistical model, which can infer more meaningful code idioms compared to methods that rely solely on fragment frequency; and (iii) the inferred code fragments adhere to the grammar rules of the chosen programming language. In Chapter 5, we explain non-parametric Bayesian PTSG in detail and provide a practical explanation of Type-Based MCMC. We also define the needed input and outline all the necessary steps and optimizations.

The success of the inference relies heavily on the properties of the provided dataset. The code within the dataset should exhibit some level of repetition in order for the method to produce satisfactory results. It is worth noting that this property is not overly restrictive, as previous research has shown that source code tends to be repetitive and non-unique [40]. Examples of source code that can be used to infer idioms include projects that adhere to established best practices or utilize frameworks with well-defined structures. Additionally, a single project may consist of multiple slightly different variants, such as a set of products in an SPL.

Certain application-specific details about the inference process must also be considered. Firstly, the chosen method ensures that an inferred code idiom conforms to syntax rules, but it does not guarantee that it will be complete enough to be immediately usable. For example, an inferred idiom may start with an else clause without a preceding if clause. Since the goal is to generate an ABG that produces both syntactically valid and usable code, it is necessary to ensure that the method achieves this. Fortunately, there is a way to avoid incomplete fragments by providing additional language-specific information. Another detail to consider is that different types of code idioms may be of interest depending on the specific use case. For instance, someone might be interested in code idioms related to the structural components of code. To guide the method, it is possible to specify where code fragments should start.

A list of inferred code idioms provides valuable insights into the repetitive nature of the code. The repetition further supports the assumption that these idioms provide a practical opportunity for automation. However, the semantics and quality of the inferred code idioms are subjective and context-dependent [31]. Furthermore, as explained in Chapter 3, code idioms inferred using the inference method tend to be shallow. This is why the next step, identified as *Step 2 (S2) — Curation*, is necessary. Useful code idioms can be manually curated to generate more applicable ABG methods. This approach also allows for re-running the inference if the method's parameter values need adjustment or if the input codebase needs to be changed, such as by only performing inference on a subset of available files. Finally, the selected and curated

code idioms, combined with additional contextual data required for their future integration into an ABG, serve as input for the second phase of the approach.

4.2 Phase 2 — ABG Extension

The SAGED approach enables the expansion of supported concepts in response to future development needs. By broadening the range of supported concepts, it becomes easier to implement code generation as the ABG is more closely aligned with the problem domain.

Figure 4.2 builds upon Figure 2.5, and illustrates the end goal of the approach, a situation where the API of an ABG is defined in terms of both the syntax of a language and useful higher-level concepts. The API of an ABG is shown at the center of the figure. The API is defined in terms of the grammar of a target programming language and in more complex terms corresponding to code idioms inferred from the existing codebase. Of course, code alterations performed using such an ABG must produce syntactically correct code. The resulting ABG should comply with the following requirements:

- **Requirement 1 — Precise Selection**: To enable precise changes to existing artifacts, the resulting ABG should allow selection of existing code elements.
- **Requirement 2 — Incremental Changes**: An ABG should enable programmers to make incremental changes to target code artifacts, facilitating the creation of code constructs larger than those that individual methods of the ABG can create.

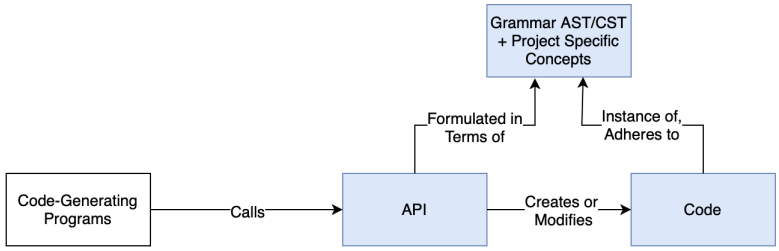

Figure 4.2: A figure from [8] adapted to our scenario — ABGs defined in terms of both grammar and domain concepts

- **Requirement 3 — Preserving Syntax Correctness**: To ensure that code remains syntactically correct after modifications, the ABG's API methods that allow for manipulation should be aware of syntax and provide warnings if syntax correctness is not met.
- **Requirement 4 — Separation Mechanisms**: There may be cases where useful code idioms are not found through inference, or it may be easier to manually define an API method for working with them. Therefore, the architecture of the ABG needs to provide a separation mechanism to accommodate both manually created and generated methods.
- **Requirement 5 — Comprehensive Support**: Modern GPLs are complex, making it difficult to have a concise ABG while also providing comprehensive support for changing artifacts. The ABG should support most anticipated change scenarios, but it should also provide an alternative for unsupported ones.

To create a complete and usable ABG that meets these requirements, it is important to start with an existing ABG basis that can handle the generated methods. This ABG basis should support basic CRUD operations on syntax elements, state management, and error detection, upon which the generated methods will rely. Additionally, the ABG basis should have a flexible structure that allows for integration of both the generated methods and any manually added API methods. During the second phase of the SAGED approach, in *Step 3 (S3) — API Method Generation*, methods should be generated that utilize the support provided by the ABG basis and adhere to its structure. Finally, in the last step of the second phase, known as *Step 4 (S4) — Integration*, the generated methods should be merged with the ABG basis.

5 Code Idiom Inference

Given that code idioms are essentially fragments of syntax trees, the problem of mining source code idioms can be considered a problem of inferring commonly occurring syntax tree fragments from the dataset. In their paper, Allamanis and Sutton [14] describe a language-independent method for mining code idioms named non-parametric Bayesian PTSG, which we find suitable for generating the output ABG. The advances presented by the authors, combined with the theoretical background in the field of NLP presented by Cohn et al. [39, 16], form the basis for the *S1 — Code Idiom Inference* step of the SAGED approach. However, the presented approximation for the method called Gibbs sampler [41] has the limitation of processing one node at a time. Considering that both codebases and syntax trees can be relatively large, it is essential to address the scalability of the implementation. The authors mention that they perform optimizations without going into detail. To optimize the method, we rely on the enhancement described in a paper by Liang et al. [32], which allows us to group multiple nodes into blocks that are then processed jointly, speeding up the inference process. Furthermore, as this is a general-purpose enhancement, we adjust it to the domain so that we can create a memory-efficient implementation as well. This adjustment is similar to the one described in a paper by Peng and Gildea [42], where the authors optimize the approach in the domain of Synchronous Context-Free Grammars. In this chapter, we provide a detailed background on the inference method so that we can describe our implementation in Section 7.1.1. To explain how the inference works, we first introduce probabilistic models of source code.

5.1 Syntactic Probabilistic Models of Source Code

The inference method we consider relies on PTSGs for representing code idioms. To define a PTSG, we can start with the definition of a *Context-Free Grammar* (CFG). A CFG is formally defined as a tuple G = (Σ, N, S, P), where:

- Σ is a set of terminal symbols;
- N is a set of nonterminal symbols;
- S ∈ N is the distinguished root nonterminal; and
- P is a set of productions (rules) r in the form of $x \to y$, where $x \in$ N and $y \in (\text{N} \cup \Sigma)^*$.

A *Probabilistic Context-Free Grammar* (PCFG) assigns a probability to each grammar production. The probability of a production is denoted as *P(r | x)*, where $x \in$ N is a nonterminal, and $r \in$ P is a production with x on its left-hand side. Given the assigned probability, a PCFG is formally defined as a tuple G = (Σ, N, S, P, Π), where Π is a set of distributions *P(r | x)*.

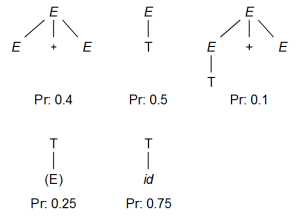

Figure 5.1: An example of a PTSG

Similar to a CFG, a *Tree Substitution Grammar* (TSG) is also defined as a tuple G = (Σ, N, S, P). However, a TSG allows nonterminals to rewrite as entire tree fragments, instead of just immediate children, as in a CFG. That is, productions P in a TSG have the form of $x \to f$, where $x \in$ N is a nonterminal, and f is a tree fragment of height ≥ 1. Each leaf of a tree fragment can be labeled as a terminal or a nonterminal. Analogous to a PCFG, a PTSG also assigns a probability to each grammar rule (Figure 5.1). Fragments of a PTSG can be used to encode non-local context, making them suitable for representing code idioms. Furthermore, nonterminal fragment leaves are convenient for representing idiom metavariables.

We can say that a PTSG G1 extends a CFG G0 if every tree with a positive probability under G1 is grammatically valid according to G0. The problem of finding code idioms then comes down to learning a PTSG that extends a known grammar G0 and is good at explaining the training set.

Essentially, the inference method we use to extract knowledge from the dataset assumes that the CFG is known and tries to identify productions from that grammar that commonly co-occur in the same sequence. Figure 5.2 depicts a code snippet written in the C# programming language under a, and a corresponding syntax subtree under b, obtained using the .NET Compiler Platform. Parsing of a dataset is the first step of the inference process. The compiler used contains knowledge of the C# CFG. The same figure also shows what could be one interesting rule of a TSG, where an *IfStatement* produces a subtree circled with a red line. We chose this rule because it corresponds to a code idiom written under c, which represents a common way of validating input parameters of a method and reacting by throwing exceptions if the prerequisites are unmet. The Bayesian PTSG method aims to find such subtrees. The inference process is described in the following sections.

5.2 Inferring a PTSG

Now that we have defined syntactic probabilistic models of code, we can present a mathematical model that is used to infer a PTSG from the dataset. The goal is to identify the posterior distribution over fragments $\mathbf{f}$, given the training corpus $\mathbf{t}$. Bayes' Rule gives a model with the form provided in Equation 5.1. As $P(\mathbf{t} \mid \mathbf{f})$ equals 1 when $\mathbf{t}$ and $\mathbf{f}$ are consistent, or 0 otherwise, the prior distribution over the fragments is the part of the model that does the work [39], so we now focus on it.

$$P(\mathbf{f} \mid \mathbf{t}) \propto P(\mathbf{t} \mid \mathbf{f})P(\mathbf{f}) \tag{5.1}$$

The Dirichlet Process (DP) is used as the prior. It has infinite support, meaning it can assign a positive probability in the prior to all possible fragments. Additionally, the DP exhibits a 'rich-get-richer' phenomenon, which means that if a fragment has been observed in a dataset, it predicts that it will occur more frequently in the future. These properties make the DP useful for this use case. Applying this to PTSGs, we can specify the form of a distribution from a set of distributions over TSG productions denoted as Π_x as follows:

$$\begin{aligned} \Pi_x \mid \alpha_x, \mu_{xf} &\sim DP(\alpha_x, \mu(\cdot \mid x)) \\ f \mid x &\sim \Pi_x \end{aligned}$$

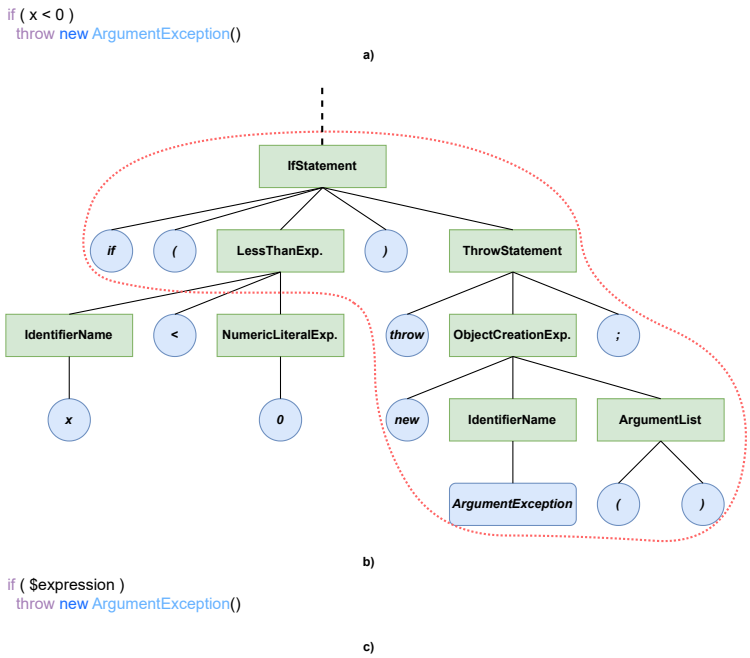

Figure 5.2: An example of a) a code snippet written in C# where a method's prerequisites are checked; b) a subtree of a syntax tree retrieved using Roslyn, corresponding to the code snippet above, where one possible TSG rule stemming from an *IfStatement* node is circled in red. Green rectangles denote nonterminal nodes, while terminals are colored blue; c) a code idiom corresponding to such a rule.

α_x is a concentration parameter that can be used to control the impact of the 'rich-get-richer' phenomenon. While it is possible to automatically specify values for α_x to some extent, we set it manually and refine it over several runs, since our goal is exploratory knowledge extraction rather than fully automatic model fitting. μ_{xf} represents the base distribution for a fragment given the fragment root nonterminal x. At this stage, it helps to understand PTSG inference as a CFG extension problem. Since the CFG is already known, the PCFG can be estimated using Maximum Likelihood (ML) from the training corpus. To calculate μ_{xf}, we draw from the probabilities for the expansion of nonterminals,

as shown in Equation 5.2, where $|f|$ represents the size of fragment *f*, P_{geom} is a geometric distribution with parameter $p_\$$, and r ranges over the multiset of productions from the CFG that are used to create *f*.

$$\mu_{xf} = P_{geom}(|f|, p_\$) \prod_{r \in f} P_{ML}(r) \tag{5.2}$$

As it is not possible to calculate the exact posterior distribution Π_x, we can approximate it using a Markov Chain Monte Carlo (MCMC) algorithm called Gibbs sampler [41]. The Gibbs sampler iteratively generates samples from the posterior distribution of each variable in the model, conditioned on the values of all other variables. A number of initial samples are discarded until a stationary distribution is reached, which is called a burn-in period. It is worth mentioning that the order in which variables are considered does not affect the probability, as DP is exchangeable. Equation 5.3 represents the integration over all possible values of Π_x, which we use for sampling from the posterior. As the equation denotes, besides α_x and μ_{xf}, the posterior also depends on the total number of times the fragment has occurred (*count(f)*) and the total number of times the nonterminal x has been rewritten (the function *h(f)* returns all the fragments rooted at x).

$$P_{post}(f) = \frac{count(f) + \alpha_x \mu_{xf}}{count(h(f)) + \alpha_x} \tag{5.3}$$

Lastly, we explain how this approach for sampling from the posterior can be used to learn a PTSG that can better explain the dataset. Given that CFG G_0 is known, parsing a file using a PTSG can be done in two steps. The first step is to parse a file from the corpus using CFG G_0, which results in a CFG tree T. The second step is to separate the nodes composing T into fragments, according to which production in the PTSG was used to generate them. In other words, we parse the files at the input, as the dataset provides hints about which fragments must have been in the grammar.

A binary variable b_x is associated with each node in the tree to represent the separation into fragments. The assignment $b_x = 1$ indicates that a cut of a preceding fragment has occurred and that the associated node is both a leaf node of the preceding fragment and a root node of a new, adjacent fragment. Conversely, the assignment $b_x = 0$ indicates that the associated node is not a root node of a new fragment but remains an internal node of the current fragment. Nonterminal nodes of Figure 5.3 for which b_x is equal to one, and that are considered leaf nodes of a preceding fragment and root nodes of adjacent

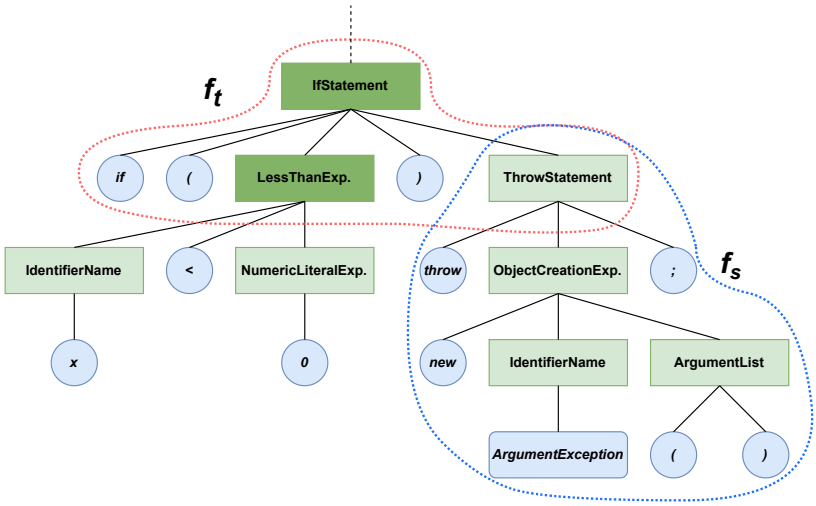

Figure 5.3: An example showing potential separation of f_{join} into f_t and f_s at node *ThrowStatement*. Fragment root nodes are colored dark green, and the inner nodes of a fragment are colored light green.

fragments, are colored dark green. Currently, the value of b_x associated with the *ThrowStatement* node is zero. So, this node is an inner node of a fragment rooted at *IfStatement*. However, if this value changed, the fragment would split into two: a fragment circled in red, which contains the *ThrowStatement* as a leaf node (f_t), and a fragment circled in blue (f_s), of which it is a root node.

We can use Equation 5.4 to calculate the probability that a node will be declared an internal node and that two adjacent fragments f_s and f_t will be joined into one. The equation implies that the more two fragments f_s and f_t occur jointly, the higher the chances they should not be split. Calculating the probability that b_x=0 for the *ThrowStatement* node of the example provided with Figure 5.3 includes calculating the posterior for the whole fragment f_{join}, fragment f_t, and fragment f_s using Equation 5.3.

Taking everything explained so far into account, the method for inferring idioms from the source code could be performed in the following steps:

1. Parse the files from the dataset into trees using the known grammar G_0
2. Transform trees to better fit the method being used, e.g., binarization

3. To create initial fragments, assign values of 0 or 1 to b_x variables associated with nodes of a tree

4. Iteratively update the value of b_x associated with each node by sampling from the distribution given in Equation 5.4.

5. After the burn-in period, sample fragments after each iteration based on the number of times they occur in the output.

$$P(b_x = 0) = \frac{P_{post}(f_{join})}{P_{post}(f_{join}) + P_{post}(f_s)P_{post}(f_t)} \tag{5.4}$$

5.3 Processing Multiple Nodes at Once

So far, we have provided an explanation of the inference procedure and the steps to perform it. However, the Gibbs sampler used for inference is not scalable, as it moves one node at a time and can take a relatively long time to reach a stationary distribution for large datasets. To address this, we rely on the enhancement presented by Liang et al. [32], who introduced a method to update a block of highly coupled variables together. The variables are grouped into blocks based on their **type**, hence the name — Type-Based sampler. A move of the Type-Based sampler is essentially a blocked Gibbs move, as it samples from the posterior distribution in the same way. Here, we explain the formalism and equations introduced by the authors, interpret them from the perspective of PTSGs, and simplify the implementation where possible.

To define a type, the authors introduced the structure of choices $\mathbf{z}$, which, in the context of PTSGs, contains elements in the form of a tuple *(x, f)*. A node's type is defined based on the impact that assigning either value to the b_x can have on $\mathbf{z}$. This impact can be observed through the total count of fragment occurrences in $\mathbf{z}$. Let $n_{xf}(\mathbf{z}) = |\{z \in \mathbf{z} : z.x = x, z.f = f\}|$ be the total number of times a nonterminal x has been rewritten using fragment *f*. Next, let $\mathbf{n(z)} = \{n_{xf}(z)\}$ denote the vector comprised of counts for all the fragments found in the dataset. The additions to $\mathbf{n(z)}$ based on whether 0 or 1 has been assigned to b_x, denoted with $\Delta\mathbf{n(z)}^{x:0}$ and $\Delta\mathbf{n(z)}^{x:1}$ respectively, form the basis for the definition of a type:

$$t(\mathbf{z}, x) \stackrel{\text{def}}{=} (\Delta\mathbf{n}^{x:0}, \Delta\mathbf{n}^{x:1}) \tag{5.5}$$

To minimize the need for bookkeeping, we modified the previously explained general-purpose definition of a type based on changes to the domain of PTSGs. When we assign the value of 0 to the variable b_x, we are left with f_{join}. On the other hand, assigning the value of 1 leaves us with fragments f_s and f_t. Only increments corresponding to these fragments can be added to $\mathbf{n(z)}$. Due to this, we can simplify the definition of a type to Equation 5.6. Simplification allows us to store only partial information about the node's surrounding fragments, rather than the entire vector of changes to $\mathbf{n(z)}$. This significantly reduces the need for bookkeeping and, consequently, the impact on memory.

$$t(\mathbf{z}, x) \stackrel{\text{def}}{=} (f_{join}, f_t, f_s) \tag{5.6}$$

There is still one prerequisite that must be considered when grouping nodes of the same type into a block. It is said that two nodes x and x' have the same type if their possible additions to the $\mathbf{n(z)}$ are the same, meaning that $\Delta\mathbf{n(z)}^{x:b} = \Delta\mathbf{n(z)}^{x':b}$, or, in other words, $(f_{join}, f_t, f_s) = (f'_{join}, f'_t, f'_s)$. In order to group such nodes, we also need to take into account an edge case where changing a value of b_x would change the type of x' or vice versa. Translated to the domain of PTSGs, if there is an intersection between f_{join} and f'_{join}, we cannot update x and x' jointly, and we consider these two nodes to be **conflicting**. Only non-conflicting nodes from the dataset with the same type can be grouped into a block B that can be updated in a single move.

Figure 5.4 provides an example of what defines a type. The type of the *ThrowStatement* node under a) is determined only by fragments f_t, f_s, and the combined f_{join}. Anything outside of this scope does not matter. The current value of the b_x variable associated with the *ThrowStatement* also does not matter. The *ThrowStatement* node under example b) is almost of the same type as the *ThrowStatement* under a). The only thing that would need to happen for them to be grouped as part of a type block is that the dark green colored *IdentifierName* under b) changes to a non-root. There is no possibility of having conflicting nodes for these two *ThrowStatement* nodes because conflicts are caused by repetition in a part of a syntax tree.

The following mathematical proposition is used as a basis for performing the update:

Proposition 1. *For any set B of non-conflicting nodes with the same type,*

$$p(\boldsymbol{b}_B \mid \boldsymbol{b} \setminus \boldsymbol{b}_B) \propto g(m) \tag{5.7}$$

$$p(m \mid \boldsymbol{b} \setminus \boldsymbol{b}_B) \propto \binom{|\boldsymbol{B}|}{m} g(m) \tag{5.8}$$

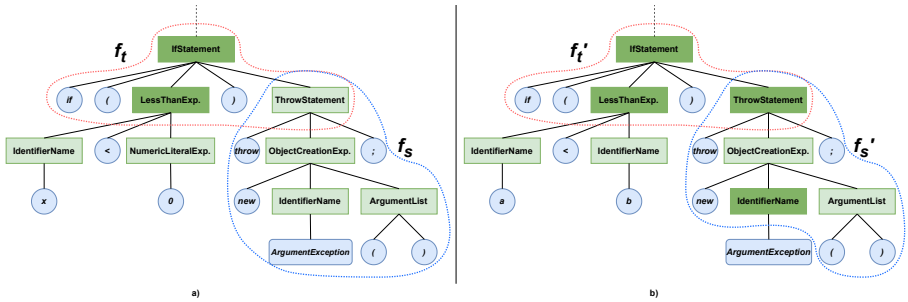

Figure 5.4: Fragment root nodes are colored dark green, and the inner nodes of a fragment are colored light green. An example shows two nodes under a) and b) of the *ThrowStatement* kind, which are not the same type. For these two nodes to become the same type, the only change that would need to happen is that the dark green colored *IdentifierName* node becomes a non-root.

where $\boldsymbol{b} = \{b_x\}$*, a vector of all binary variables associated with nodes;* $\boldsymbol{b}_B = \{b_x \mid x \in B\}$*, a vector of binary variables associated with nodes grouped into a type block;* $m = \sum_{x \in B} b_x$*, a sum over binary variables associated with nodes grouped into a type block; and* g*, a function of* m *[32].*

This proposition provides a way to sample values for variables associated with a block's members, denoted with $\mathbf{b}_B$. Equation 5.7 indicates that the sampling of a vector $\mathbf{b}_B$ conditioned on the binary variables associated with the rest of the nodes is proportional to g, a function of m. Variable m represents the number of ones that could be assigned to these $\mathbf{b}_B$ variables, and its value ranges from 0 to the size of a block $\mid B \mid$. It is not relevant how values are distributed inside a block because variables have the exchangeability property. Thus, there are $\binom{|\mathbf{B}|}{m}$ possible ways to distribute the values, each with the same probability $g(m)$, resulting in Equation 5.8.

The exchangeability property also implies that the likelihood of a new assignment of values to $\mathbf{b}_B$ depends on the changes to the number of fragments affected by such an assignment. By definition provided with Equation 5.9, the difference of changes made to the vector of counts of all fragments found in the dataset equals the sum of changes made by assigning m ones to arbitrary nodes of a type block and those made by assigning $(|\boldsymbol{B}| - m)$ zeroes to the rest of the nodes of the type block. A single value in this vector of count changes is denoted with $\Delta n^{B:m}$.

$$\Delta\mathbf{n}^{\mathbf{B}:m} \stackrel{\text{def}}{=} m\Delta\mathbf{n}^{\mathbf{x}:1} + (|\mathbf{B}| - m)\Delta\mathbf{n}^{\mathbf{x}:0} \tag{5.9}$$

As derived by Liang et al. [32], the function *g(m)* is calculated according to Equation 5.10. Recall that α_x is a concentration parameter and that μ_{xf} is a base distribution for a fragment given the fragment root nonterminal x provided by Equation 5.2. $n_{xf}(\mathbf{z}^{-\mathbf{B}})$ denotes the number of times a fragment f has rewritten a nonterminal x when we exclude the type block from the structure of choices $\mathbf{z}$. Similarly, the $n_{x\cdot}(\mathbf{z}^{-\mathbf{B}})$ denotes the number of times any fragment f has rewritten a nonterminal x when we exclude the type block from the structure of choices $\mathbf{z}$. The superscript operation found in Equation 5.10 is the ascending factorial.

Per the definition of the ascending factorial $a^{(k)}$, if k equals zero, the value of $a^{(k)}$ is one. As previously discussed, the only changes that can be made to the vector of counts of all the fragments found in the dataset are of those fragments defining a type. That means that in the sequences of all the nonterminal nodes and the fragments that rewrite them in Equation 5.10, the only non-one factors obtained with the equation are related to the type-defining fragments f_{join}, f_t, f_s and their roots. Thus, a simpler version of Equation 5.10 adjusted to the domain of PTSG can be derived.

$$g(m) = \prod_{x \in N} \frac{\prod_f (\alpha_{xf}\mu_{xf} + n_{xf}(\mathbf{z}^{-\mathbf{B}}))^{(\Delta n_{xf}^{B:m})}}{(\alpha_x + n_{x\cdot}(\mathbf{z}^{-\mathbf{B}}))^{(\Delta n_{x\cdot}^{\mathbf{B}:m})}} \tag{5.10}$$

For the sake of brevity of the final equation, let us single out a part of the numerator related to a single fragment and its root (Equation 5.11). Let us also single out a part of the denominator related to a single root nonterminal (Equation 5.12). Then, we can write a variant of Equation 5.10 adjusted to the domain of PTSG as presented in Equation 5.13. As presented, there are two cases of the equation. Fragments f_{join} and f_t always have the same root node. However, the root node of an f_s fragment might be of the same kind as the root node of the other two fragments of a type-defining triplet. For those two scenarios, different non-one factors of the sequences of Equation 5.10 are obtained, and thus, the two cases. We will denote the root nonterminal of f_{join} and f_t as x_1, and root nonterminal of f_s as x_2.

$$gnp_{xf}(m) = (\alpha_{xf}\mu_{xf} + n_{xf}(\mathbf{z}^{-\mathbf{B}}))^{(\Delta n_{xf}^{B:m})} \tag{5.11}$$

$$gdp_x(m) = (\alpha_x + n_{x\cdot}(\mathbf{z}^{-\mathbf{B}}))^{(\Delta n_{x\cdot}^{\mathbf{B}:m})} \tag{5.12}$$

$$g(m) = \begin{cases} \frac{gnp_{x_1 f_{join}}(m) gnp_{x_1 f_t}(m) gnp_{x_1 f_s}(m)}{gdp_{x_1}(m)} & \textit{if } x_1 \textit{ and } x_2 \textit{ are of the same kind} \\ \frac{gnp_{x_1 f_{join}}(m) gnp_{x_1 f_t}(m) gnp_{x_2 f_s}(m)}{gdp_{x_1}(m) gdp_{x_2}(m)} & \textit{otherwise} \end{cases} \tag{5.13}$$

After calculating *g(m)*, values produced using Equation 5.8 that approximate the posterior distribution need to be normalized, as the distribution is only proportional to the expression on the right-hand side. Sampling from the distribution given by Equation 5.8 and assigning m ones to the type block nodes completes a block move.

5.4 Preprocessing of Parsed Syntax Trees

There are a few things that can be done to improve the inference process by altering syntax trees to better suit the inference method. The first issue to address is the sparseness of the found CFG rules. To address this problem, syntax trees are binarized to make the CFG less sparse and better capture code idioms in sequential statements.

Another consideration is what we aim to achieve through the inference process. Inferred idioms should possess three properties: syntactic correctness, completeness, and conformance to the desired ABG. As mentioned, GPL syntax trees can be complex, and inference is a stochastic process. Given that a subtree of a PTSG is derived by following CFG rules, each inferred fragment is guaranteed to conform to syntax rules. However, a fragment may not be complete enough to be usable. For example, consider a try-catch statement in C#. Neither part of this statement can exist independently, and it would not make sense to have them as stand-alone idioms. Therefore, it is appropriate to link the catch part to the try part of the statement.

Also, we need to consider what the ABG basis we integrate into can handle. RoseLib, used as an ABG basis and described in detail in Chapter 6, primarily focuses on the structural elements of the C# language, such as classes and methods. Thus, it is suitable for discovering code idioms rooted in these elements. By declaring these nodes as roots, we guide the inference process to find potentially useful code idioms that can be integrated into the ABG basis. To bind a node to the preceding fragment, we set the value of the associated binary b_x to zero. To declare it as a root, we set the value to one.

6 RoseLib

In our research, as part of efforts to mitigate the difficulties of working with Roslyn's Syntax Tree API, we developed an open-source ABG called RoseLib [11, 43, 30], written in C#. RoseLib provides a layer above the Syntax Tree API, enabling the easier creation of code-generating programs that make precise modifications to C# files without compromising their syntactic correctness.

It contains methods related to C# language concepts that are familiar to programmers, and includes mechanisms for code navigation, checking for changes in existing code between code generation cycles, and making syntactically valid alterations. The provided methods can be separated into two distinct groups — tree traversal and tree modification. Furthermore, to allow for easier use through different code generation cycles, it provides traceability links for tracking code elements. This is a variation of an established mechanism for tracing model elements to corresponding generated text parts, similar to the one in the MOFM2T language [44]. Lastly, it provides a new mechanism called integrity checks, which facilitates checking whether previously generated code has possibly been altered between code generation cycles. The following text explains this in detail, ending with a proof-of-concept example for clarity.

6.1 Traversal and Composition

In this section, we discuss how RoseLib can handle basic operations for traversing and modifying Roslyn C# Syntax Trees (RCSTs). A simplified class diagram of the tree traversal part of the API is shown in Figure 6.1. Classes central to traversing trees are *navigators*. Concrete navigators are related to C# language concepts, hence their names. The starting point for the traversal using a concrete navigator is a selected RCST node of the corresponding syntax kind.

Traversal is implemented through selection methods organized in different *selector* interfaces. Selecting descending nodes of arbitrary depth is enabled, and one navigator class can implement multiple selectors. Selection methods are designed to allow chaining so that usage follows the language rules (stemming from the grammar). Navigator methods' return value types reflect this but are omitted from the diagram for clarity. Each navigator inherits the *StatefulVisitor* class to track the nodes of a tree that were visited. Lastly, each navigator has a *StartComposing* method to facilitate a type-safe beginning of tree modification, which returns the requested *composer* class if the current selection allows it.

The *StartComposing* method returns an instance of a requested *composer* class, a member of a hierarchy of classes that can alter RCSTs shown in Figure 6.2. Concrete composers are also related to C# language elements and provide associated basic create, update, and delete operations. These operations can be combined to create more complex transformations. As shown in the diagram, every composer is associated with a stateful visitor to keep track of the state through changes. The state contains a stack of syntax nodes and hides the fact that these are part of an immutable RCST.

Two concepts are introduced to help manage the state: pivot and reference nodes. Pivot nodes are of a syntax kind corresponding to the concrete composer and are the central point of operations. Reference nodes are one of the descending nodes and provide additional context for operations. An example would be having a class for a pivot node and one of its fields as a reference node. The operation that adds a new field would then add it to the pivot class under the reference field. Pivot and reference nodes are chosen when transitioning from traversal to the change stage. The *StartComposing* method of navigators invokes the static *CanProcessSelection* method of a concrete composer to check if this composer can handle a given pivot and a possible reference node. If it can, a new instance of this composer is instantiated, and changes can take place.

6.2 The CSPath Grammar

CSPath is the language used to implement traceability links, where a textual expression saved outside of the codebase can be used to locate code elements within it. The CSPath grammar is inspired by the XPath language [45] because it serves a similar purpose, and expressions written using it have a similar form. CSPath was implemented using the textX meta-language [46]. A class diagram corresponding to the grammar is shown in Figure 6.3, together with representative expressions. As shown, a path is composed of multiple parts. Each part

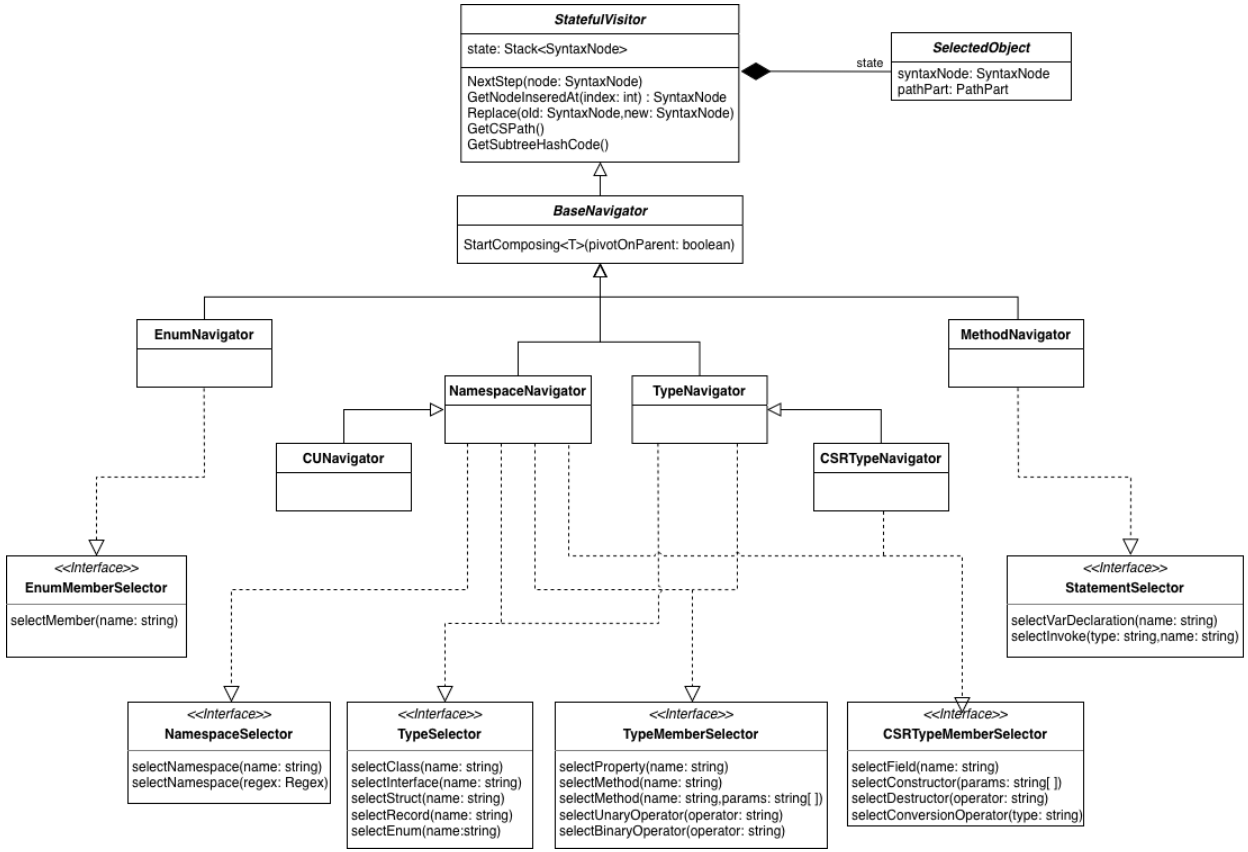

Figure 6.1: A simplified class diagram of the traversal API

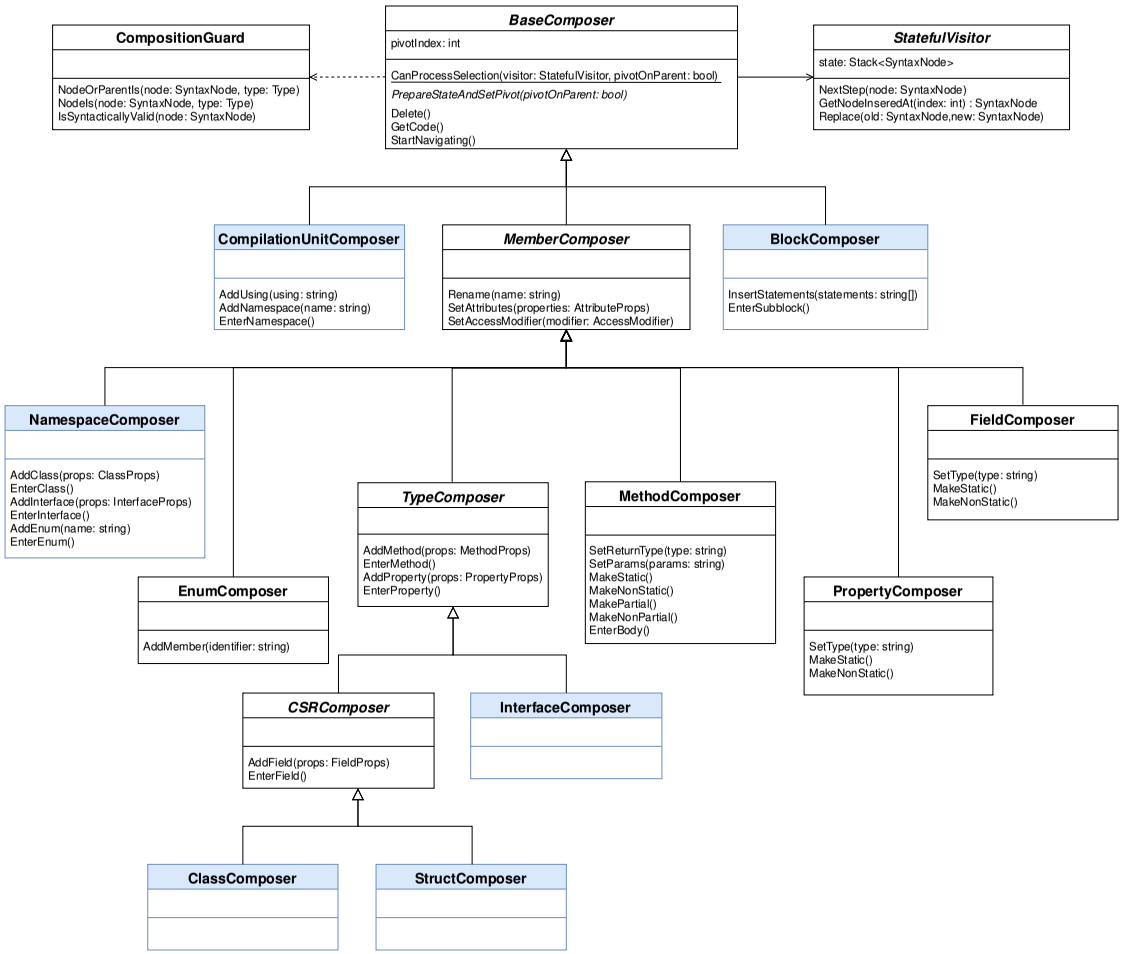

Figure 6.2: A simplified class diagram of the composition API

denotes a descent towards an instance of some concept. Descent is always relative in CSPath, meaning that the descendant syntax node of interest does not need to be a direct descendant of the current one found in the state. CSPath concepts are mapped to concepts of the C# language. The most significant structural C# concepts are supported by RoseLib, such as namespaces, classes, and their members. Every descent starts at the syntax node corresponding to the *Compilation unit* concept of C#. This concept is not explicitly defined as a CSPath concept but is automatically implied and found in the state. CSPath expressions can also contain predicates associated with concepts, which serve to find the instance of a concept (syntax node) with the appropriate attribute value. Possible attributes are dependent on the concrete concept the predicate is applied to and are not defined by the grammar but by the RoseLib library instead. These attributes correspond to the parameters of RoseLib's API methods for navigating syntax trees. Note that RoseLib can create CSPath expressions automatically based on the current selection.

The example expressions in Figure 6.3 illustrate the following:

- A descent towards a first namespace with a name matching the regular expression;
- A descent towards a first method with *TestMethod* as a name. From the expression itself, it is not known what the parent concept of such a method is; it could be a class, an interface, or some other C# concept.
- A descent towards a method named *TestOut* with a class as an ancestor.

6.3 CSPath Engine

Now that we have explained the structure of CSPath expressions, we can explain how the engine interprets them. A diagram of the engine's design is shown in Figure 6.4. The engine relies on textX to create an instance of the CSPath model based on a given expression. The processing of path parts is done by the logic implemented using the Chain of Responsibility pattern [47]. The chain comprises handlers. Each handler inherits *BaseHandler*, implements the *Handle* method, and holds a reference to its successor (except the last one).

For interpretation to occur, the engine instantiates a context, combining a *CUNavigator* instance with a path part first in line for processing. This initial context is then passed through the chain. If a handler can handle a specific

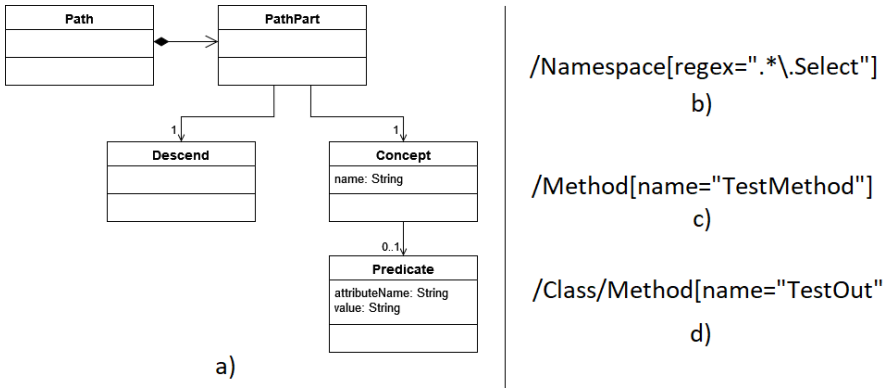

Figure 6.3: Showing a) the CSPath model and b-d) example expressions

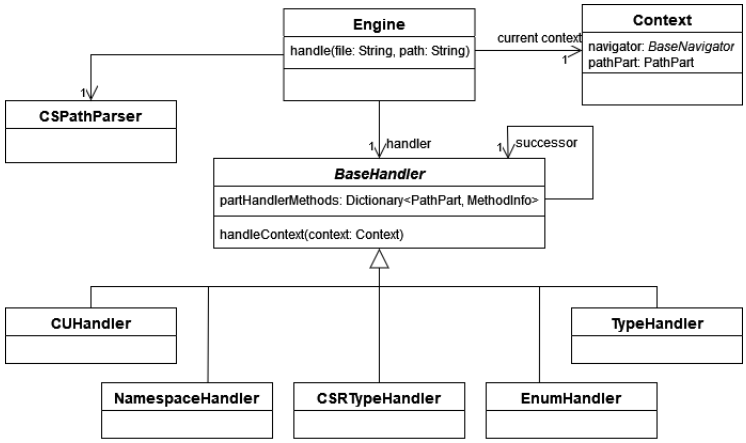

Figure 6.4: The CSPath interpreter engine

navigator and path part pair, the descent is performed by applying the appropriate selection method. The selection method returns a navigator appropriate for the selected syntax tree node, and the processing of the current context is finished. If none of the handlers can handle the context, the path cannot be processed. Path parts are iteratively combined with succeeding navigators to form new context instances, which get passed through the chain. The handling

procedure is repeated until all the parts have been processed and the descent is over. The last navigator returned by the handle method contains an appropriate state and can be used for further descent or to start syntax tree modification.

6.4 Integrity checks

Once a node has been selected, it is possible to calculate a hash value for the code snippet corresponding to the syntax tree fragment rooted at this node. The way this node was selected is not significant, and it could be selected either directly using selection methods or indirectly using methods for manipulating syntax trees. Both navigator and tree manipulation classes have the *GetSubtreeHashCode* method, which performs the calculation.

Before hashing occurs, we need to ensure that secondary notation does not influence the value, which is important because insignificant changes to the code snippet would impact the resulting value, making the integrity checks less useful. For example, changes in indentation or added comments would cause the integrity check to fail, even though the code's semantics did not change. To prevent this, we temporarily remove all comments from the code snippet and normalize indentation. Then, we can calculate the hash value of the code snippet. To perform an integrity check later in time, code generators must invoke the calculation on the same tree fragment once again and compare the newly derived value with the one obtained originally. If these are not identical, a change has occurred between the generation cycles.

6.5 Proof-of-Concept Example

To showcase how the Seamless workflow from Section 3.1 can be implemented, we present an example based on the metamodel shown in Figure 6.5. In this example, an instance of a document is mapped to a C# class with the same name, and document fields are mapped to the corresponding fields of the class. Based on this metamodel, we define a conforming model: a *Work order* document with Order ID (Integer) and Order date (DateTime) fields. Lastly, suppose that such a class has been generated in the initial generation cycle.

With these prerequisites, we now describe how an example code generator can handle a model change in which the *Order ID* field is modified to the String type. Figure 6.6 shows sample code for a generator that can handle this change. To change the field type in the code, the generator first needs to ensure that

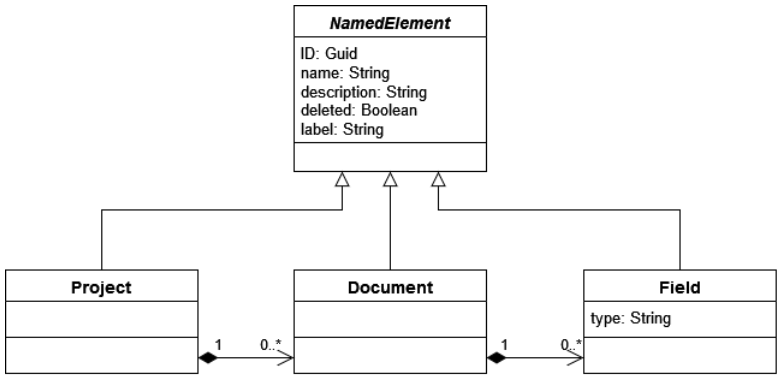

Figure 6.5: The metamodel for the example

the field is in the navigator state. As shown, the generator does not need to implement any case-specific logic for descending through the syntax tree, as it can rely on *CSPathEngine* to interpret the CSPath expression and perform the descent, returning the resulting navigator. To check that the field has not been changed between the cycles, the generator can calculate its current hash value and compare it to the value saved in the previous generation cycle. If the check fails, the code generation cycle stops. The integrity check logic is also not case-specific. The example ends by changing the field type and producing the resulting code.

```
CPathEngine cSPathEngine = new CPathEngine();
var navigator = cSPathEngine.Evaluate(
    reader,
    "/Class[name='WorkOrder]/Field[name='OrderId']'"
    );

var hashValue = navigator.GetSubtreeHashCode();
if (!hashValue.Equals(previousHashValue))
{
    throw Exception("Code was changed!")
}

var output = navigator.StartComposing<FieldComposer>()
.SetType("String")
.GetCode();
```

Figure 6.6: A code generator that can update the type of a field

7 Application of SAGED

In this chapter, we present our solution for implementing the SAGED approach, designed to handle code written in C#. However, the main component responsible for the inference method, referred to as the inference core, is language-independent and can be extended for other programming languages as well. We chose C# due to our prior experience with ABGs in industrial projects using this language. Additionally, we can utilize our previously published results [12] as a baseline to evaluate the SAGED approach's effectiveness.

Figure 7.1 displays an overview of the reference architecture. The following sections discuss the solution's components and the two phases of the approach.

7.1 Concept Discovery

Two components enable the implementation of the *S1 — Code Idiom Inference* step: the *Preprocessor* and the *Inferencer*. The Preprocessor is responsible for cleansing and preparing the input files by removing most secondary notation elements from the code, such as comments and documentation, and eliminating primary notation elements that are not useful for inference, such as directives and using statements.

Additionally, the Preprocessor can group input files based on the inheritance tree, a common mechanism for defining layers in C# projects. Code within the same layer typically addresses similar tasks, so we expect useful repetitions to be found there. This grouping has two key benefits: first, it improves inference results by positively impacting inference distributions; second, it mitigates scalability issues and enables faster inference by performing inference on a smaller dataset.

The prepared files serve as input for the Inferencer, which is built by extend-

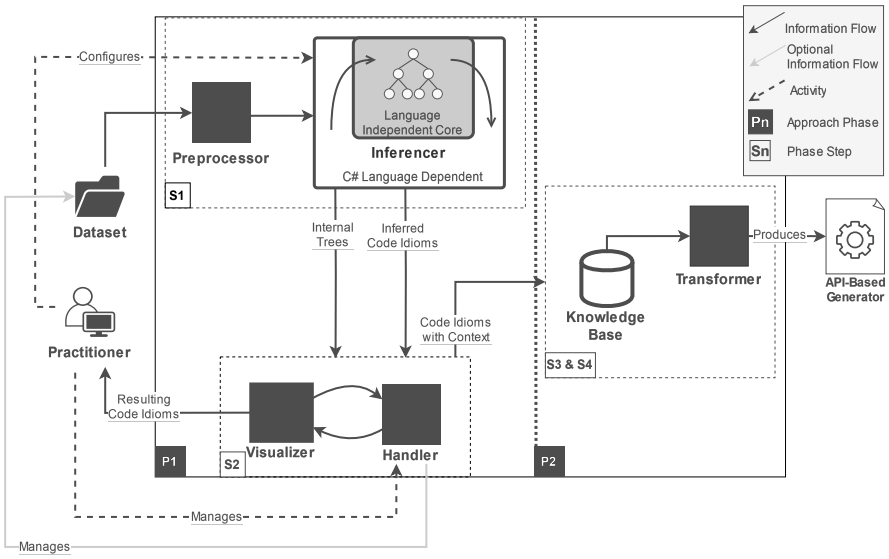

Figure 7.1: Overview of the SAGED reference architecture

ing the inference core to perform inference on C# files.

7.1.1 The Language-Independent Inference Core

The inference core implements the non-parametric Bayesian PTSG inference method explained in Chapter 5. This implementation is available as an open-source project [55].

The data structure presented in Figure 7.2 is designed to make the inference core language-independent. Essentially, the inference core does not handle syntax trees created by language-specific parsers; instead, it operates on a matching set of internal trees. Parsed syntax trees must be translated into internal trees, where an internal tree is an instance of the *Tree* class, comprised of nodes that are instances of the *Node* class ordered into a parent-child hierarchy. Each node has a label and fields that store the information used for the inference process.

Of course, some details about the language, such as a CFG and syntax tree structure, must be provided to execute the method. Therefore, the core includes abstract classes focused on input and output operations that need to be concretized. The classes *TreeCreator* and *NodeCreator* are abstract and must

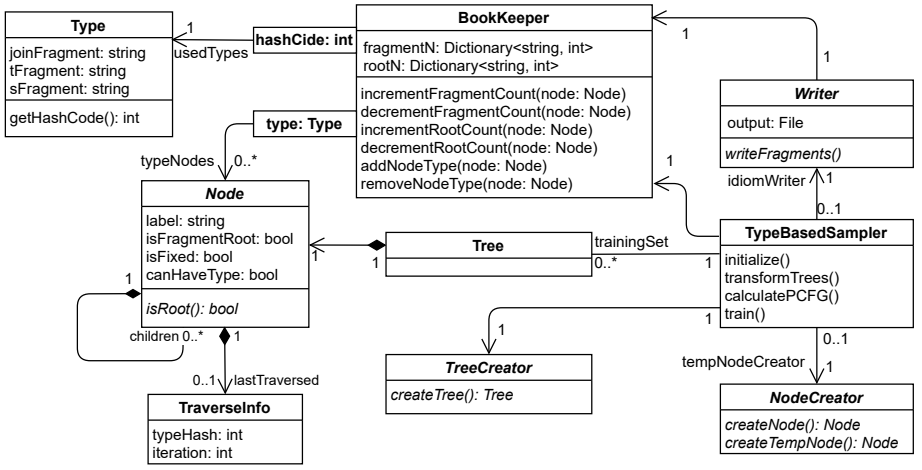

Figure 7.2: Class Diagram of the Inference Core

be implemented when extending the core to build internal trees. The *Writer* is also an abstract class that must define how a fragment of an internal tree is transformed into a code idiom.

The *TypeBasedSampler* class performs the inference, while the *BookKeeper* class supports the process. The *BookKeeper* keeps track of nodes, their corresponding types, and the counts of fragments and roots found in a parsed dataset. The specific processes and algorithms of the inference methods implemented with *TypeBasedSampler*, which complement the theoretical background described in Chapter 5, are explained below.

The inference method begins by creating appropriate internal trees from the dataset. An overview of the process for creating these internal trees is presented in Figure 7.3. The process consists of three steps. In the first step, *syntax trees are obtained* from a language-dependent parser. In the second step, the obtained *syntax trees are translated into internal trees* composed of instances of the *Node* class, which the inference method can handle. The parser provides knowledge

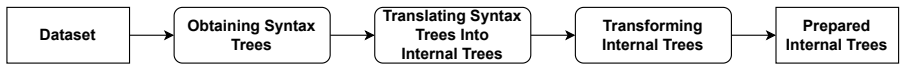

Figure 7.3: Overview of the internal tree creation process

about the grammar, which we capture during translation by setting the label of an internal tree node to the type provided by the parser. These labels will be used later to derive the CFG and calculate the PCFG. In the third and final step of the process, the *internal trees are transformed* to be more suitable for the algorithm, as explained in Section 5.4.

To bind a node to the preceding fragment, we set the property *isFragment-Root* of the *Node* class to *false*. To declare it as a root, we set it to *true*. The property *isFixed* of the same class prevents these values from being changed. During the binarization process, the inserted binarization nodes receive labels related to the parent node whose children are being binarized. The two mentioned abstract classes, *TreeCreator* and *NodeCreator*, facilitate the creation of the internal representation. Implementing the corresponding *create* methods for each case should follow the steps explained above, along with any other steps deemed useful for achieving better results.

Once the internal trees are created, we can proceed with the next steps of Algorithm 1. The subsequent step involves deriving the CFG by examining the labels assigned to different nodes and identifying the existing productions in the grammar. Based on this derivation, the PCFG is constructed by dividing the count of each production by the total count of productions with the same left-hand side. The derived productions and their corresponding calculated probabilities are then used as input for calculating the base distribution using Equation 5.2.

Now that we have an internal representation to support inference and a PCFG to extend, we randomly divide the internal trees into initial fragments. It is important to maintain a high probability of dividing the trees, as the method takes larger steps when the fragments are initially smaller. We then count the roots and their fragments, sorting the nodes according to their type. The *BookKeeper* class is used to keep track of this information as the fragments change. The different types of nodes are represented by the *Type* class. To conserve memory, we reuse instances of the *Type* class for all nodes of the same type. This is necessary because having millions of instances could significantly impact memory consumption. We continuously update the information stored in the BookKeeper as the method progresses. The order in which types and nodes are stored in the BookKeeper is determined randomly. We shuffle the types and nodes with each iteration of the algorithm to prevent any unintentional bias from affecting the inference process through ordering.

Algorithm 1: Training of Type-Based MCMC

```
CreateTrees()
TransformTrees()
CalculatePCFG()

CreateInitialFragments()
SetNodeTypes()
currentIteration = 0
while currentIteration <  totalIterations do
    ShuffleTypesAndNodes(types)
    foreach type in types do
        typeBlock = CreateTypeBlock(type)

        valuesForM = [ ]
        for m = 0; m ≤ typeBlock.size(); m++ do
            valueForM = CalculateEq5.8ForM(type, m)
            valuesForM.Add(valueForM)

        probabilitiesForM = Normalize(valuesForM)
        drawnM = SampleM(probabilitiesForM)
        ones = SampleOnes(typeBlock.size(), drawnM)
        UpdateSitesAndTypes(typeBlock, ones)
    if currentIteration > burnIn then
        WriteFragments()
    currentIteration++;
```

The remainder of the method follows the steps of Type-Based MCMC inference discussed earlier. The only step that requires further explanation is how we create type blocks. As mentioned, only non-conflicting sites (nodes) of the same type can be grouped into a type block. To determine if two sites of the same type are conflicting, we need to check whether changing the value of the b_x variable for one site would alter the type triplet (f_{join}, f_t, f_s) of another site. Each potential site to be added to the type block is considered individually to check for conflicts with the sites already in the block. Once a site is determined to be non-conflicting, the nodes in the associated f_{join} fragment are marked. The assigned mark is unique to the considered type and iteration, helping to identify any potential intersections with subsequent type sites. For each subsequent site of the same type, we search for these marks within the associated fragment f_{join}. If no marks are found, we conclude that there are no intersections and that the site does not conflict with any previously added sites. The pseudocode in Algorithm 2 describes the process that each site must undergo before being included in the type block. The *CanAddToTypeBlock* method checks for possible

conflicts, and the *SetLastTraversed* method is used for marking purposes.

Algorithm 2: Checking site for Conflict

```
Function CanAddToTypeBlock(siteNode,iteration):
    joinRoot = FindJoinRoot(siteNode)
    if !IsConflicting(siteNode, iteration, joinRoot) then
        SetLastTraversed(siteNode, iteration, joinRoot)
        return true
    return false
Function IsConflicting(siteNode, iteration, node):
    if IsNodeSeen(node, siteNode.Type, iteration) then
        return true
    foreach child in node.Children do
        if IsBorderNode(child, siteNode) then
            if IsNodeSeen(child, siteNode.Type, iteration) then
                return true
        else
            if IsConflicting(siteNode, iteration, child) then
                return true
    return false
Function SetLastTraversed(siteNode, iteration, node):
    node.SetTraverseInfo(siteNode.Type, iteration)
    foreach child in node.Children do
        if IsBorderNode(child, siteNode) then
            child.SetTraverseInfo(siteNode.Type, iteration)
        else
            SetLastTraversed(siteNode, iteration, child)
```

7.1.2 Supporting Concept Discovery from C# Files

To extend the inference core so that it can infer code idioms from C# files and create the Inferencer, we rely on the Roslyn platform. Roslyn provides the parser containing the knowledge about the C# grammar needed to obtain RCSTs. The Inferencer implements the *TreeCreator* and *NodeCreator* abstract classes, depicted in Figure 7.2, allowing the creation of internal trees from RCSTs. During creation, we also adjust the structure imposed by Roslyn to better suit the inference method. For example, we introduce additional nodes for identifiers

where necessary, ensuring that the results can be more generic.

We also allow practitioners to specify which RCST node kinds should be fixed as roots or non-roots, enabling them to guide the inference process. Upon completion of the inference, the inferred fragments represented by the internal trees must be translated into C# code idioms with metavariables. Consequently, the Inferencer also implements the *Writer* abstract class provided by the core.

The output of the Inferencer is a list of code idioms enriched with contextual data needed for future integration into the API. This output includes: (i) a fragment with included metavariables, which should be replaced with exact code fragments when generating code using the resulting API; (ii) the RCST kind of a concrete fragment root node. Additionally, internal trees enriched with information about the PTSG as a result of the inference process are preserved for future use.

Practitioners are responsible for the *S2 — Curation* step. Curation is a subjective process, so practitioners may review the results of the inference, manipulate the code idioms, and select the ones they find useful. Two components

Figure 7.4: An input file enriched with code idiom information using the Visualizer. Idioms are assigned a unique color for distinction. The tooltip provides insight into the frequency of a code idiom.

facilitate this process: the Visualizer and the Handler. The Visualizer provides insights into the inference results by allowing practitioners to browse input C# files enriched with information about inferred code idioms, as shown in Figure 7.4. The displayed information includes aggregate data — such as the number of times a code idiom occurred and the number of subtrees corresponding to each code idiom present in the input files. Additionally, the Visualizer enables users to navigate through the code by providing links to other locations in the code where the same code idiom is found.

The Visualizer assigns a unique mark to each code idiom. This mark can then be used for curation with the Handler. The Handler facilitates the joining of inferred code idioms while ensuring syntax correctness. It also allows for the grouping of files that share a common code idiom. These grouped files can be processed separately. This grouping extends the inheritance-based grouping carried out by the Preprocessor, enhancing inference distributions.

At the end of the curation process, practitioners must provide the information needed to generate ABG API methods. This includes the API method name, parameter names mapped to metavariables, and the ABG structure elements where the generated methods should be integrated. To assist practitioners, we have created a page (shown in Figure 7.5) that displays all the required items. The page also offers guidance by listing the possible ABG elements, referred to as *Composers*, where a code idiom can be integrated based on the root node kind. The following section discusses the integration process and these *Composers*.

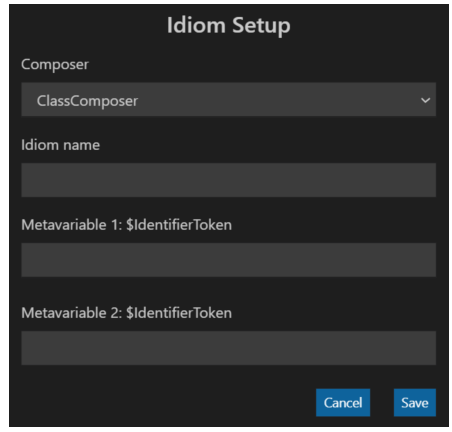

Figure 7.5: The page for providing data needed for code idiom integration

7.2 RoseLib Extension

The implementation of SAGED is based on RoseLib. The goal is to enhance RoseLib's API by adding methods that handle the addition of subtrees to RCSTs based on inferred idioms. This transformation turns RoseLib into an ABG that combines the syntax of a

language with higher-level concepts. These methods tackle the most challenging aspect of using ABGs: creating larger transformations. In this section, we refer to RoseLib's structural elements and the mechanisms used for the extension.

7.2.1 Addressing the Requirements

To explain how having RoseLib as a basis assists us in obtaining an ABG with the desired key properties, we analyze the requirements specified in Section 4.2:

- **Requirement 1 — Precise Selection**: RoseLib utilizes Roslyn's parser to parse C# files. To enable a gradual descent through an RCST in search of a desired node, RoseLib provides mechanisms to keep track of the selection state contained within a class *StatefulVisitor*. Keeping track of selected nodes enables chaining methods that perform the descent relative to the selected node. If the desired node of an RCST is present, it will be the final element added to the state; if not, an error is raised.

- **Requirement 2 — Incremental Changes**: The selection state within *StatefulVisitor*, together with surrounding mechanisms and node selection methods, provides a basis for methods performing RCST manipulation to allow incremental changes to target RCSTs. Each change made with RoseLib's API updates the state so that another change can follow.

- **Requirement 3 — Preserving Syntax Correctness**: RoseLib provides multiple functionalities to help maintain the code's syntactic correctness. It facilitates a controlled transition from the selection of nodes to the modification stage. A component that modifies RCSTs declares that it can modify a specific node kind. Additionally, it provides guards that can be called before a modification takes place to ensure that the current selection state is consistent with the intended modification. Lastly, some validity check methods can be called to ensure the RCST is syntactically correct after modifications. New API methods added with the SAGED approach can utilize these mechanisms.

- **Requirement 4 — Separation Mechanisms**: RoseLib contains a set of manually written selection and modification methods related to a subset of C# concepts. It also has an underlying structure that can be expanded when creating new methods with the SAGED approach. The *Partial Classes* mechanism [6] is used to enable the extension of the existing RoseLib structure.

- **Requirement 5 — Comprehensive Support**: As explained in Section 3.2, RCSTs are complex and contain many concepts unfamiliar to most software developers. RoseLib was designed to conceal these complexities and provide support limited to the most familiar language concepts. However, RoseLib's selection state, which comprises RCST nodes, can be accessed whenever needed, and Roslyn's Syntax Tree API can be used to cover cases unsupported by RoseLib.

7.2.2 Performing Extension

We extend RoseLib's API with methods for adding subtrees to RCSTs based on inferred code idioms. The extension relies on the *composers* introduced in Chapter 6 — the hierarchy of RoseLib's classes that can alter RCSTs, primarily associated with the structural elements of the C# language such as classes, methods, and blocks. The extension is relevant for composers that add complex subtrees to a selected node.

The Transformer, depicted in Figure 7.1, is responsible for steps *(S3) — generation of API methods* based on selected idioms and *(S4) — integration* of generated methods with the RoseLib composers, covering the whole *(P2) — ABG Extension* phase of the SAGED approach. Template-based code generation is used for generating API methods, and the *Partial Classes* mechanism available in the C# language is used to integrate generated API methods with the existing ABG basis. The following text explains this process in detail.

A portion of the data required for successful API method generation is provided during the *P1 — Concept Discovery* phase. This includes idioms with metavariables and contextual data, as well as data added by the practitioner during curation — names for a method and its parameters, and a chosen composer to which the generated method should be added. The Knowledge base component provides the other portion of the required data. It provides the Transformer with information about (i) which node kinds an individual composer can handle, (ii) which templates should be used by the Transformer to generate a concrete API method, and (iii) the location of the RoseLib code and the folder where generated parts of composers are stored.

A simplified descriptive pseudo-code corresponding to the algorithm used by the Transformer to generate methods is provided with Algorithm 3. Data resulting from the *P1 — Concept Discovery* phase, together with data contained in the Knowledge base, serve as input for this algorithm. The steps of the algorithm are as follows. First, an inferred and selected fragment with metavariables is transformed into a C# string literal. In this step, metavariables are substi-

```
public ClassComposer AddGetOne(string name, string returnType)
{
    CompositionGuard.NodeOrParentIs(Visitor.CurrentNode, typeof(ClassDeclarationSyntax));

    var fragment = $"public {returnType} {name}(int id) {{ return _context.getOne(id); }}";
    var member = SyntaxFactory.ParseMemberDeclaration(fragment);
    if (member!.ContainsDiagnostics)
    {
        throw new Exception("Parsed code not syntactically valid.");
    }

    var referenceNode = TryGetReferenceAndPopToPivot();
    var newEnclosingNode = AddMemberToCurrentNode(member!, referenceNode);
    Visitor.ReplaceNodeAndAdjustState(Visitor.CurrentNode!, newEnclosingNode);

    var navigator = BaseNavigator.CreateTempNavigator<CSRTypeNavigator>(Visitor);
    navigator.SelectMethodDeclaration(name);

    return this;
}
```

Figure 7.6: The resulting method that can insert a code idiom into an RCST

tuted with parameters whose values will be interpolated at runtime. Then, a method declaration is created based on the provided method and parameter names. A body for this method is generated using a template related to the chosen composer. Finally, the generated method is added to a partial class of the chosen composer.

Algorithm 3: Generating an API method

```
var transformedFragment = TransformFragment(fragment, parameters)

var method = CreateMethodDefinition(name, parameters)
var template = RetrieveTemplate(composer)
var body = template.TransformText(transformedFragment,
  rootNodeType)
method.WithBody(body)

var path = RetrieveComposerLocation(composer)
if(!Path.Exists(path)) CreateComposerBase(composer)
AddMethodToComposer(method, composer)
```

An advantage of automating ABG API method development is that users do not need to be familiar with ABG's internal details. However, to explain how the generated code fulfills the set requirements, we provide an example of a generated API method. An example method added to the *ClassComposer* can be seen in Figure 7.6.

The code generated by the Transformer depends on the provided input, but the example shows how such methods generally work. As shown, first, a method of *CompositionGuard* is invoked to check whether the current state is appropriate for the intended modifications. Then, a code fragment in the form of a string literal is interpolated and parsed. If the parsing of the fragment creates a syntactically correct subtree, the subtree can be included in the existing RCST. Otherwise, the operation fails. Finally, the root node of the created subtree is selected and added to the state so that possible subsequent operations can take place.

7.3 Practitioner Tooling

So far, we have introduced all the solution components to explain how the SAGED approach is implemented. In this section, we will explain these components from the perspective of an MDSE practitioner. Some overlap with previous sections is intentional, as the aim is to add implementation details relevant to practitioners.

7.3.1 Visual Studio Code Plugin

From the practitioner's perspective, the main tool is the Visual Studio Code [115] plugin [56], which integrates the approach seamlessly into the existing developer toolkit. It provides UI components that interact with the introduced components of the architecture and related artifacts: it lets users cleanse the code, specify parameters for inferring code idioms from the currently active project, understand the algorithm's behavior, visualize the inferred code idioms, and generate RoseLib methods.

Visual Studio Code is built with Electron, a JavaScript framework that allows creating desktop applications that are rendered by the Chromium rendering engine [116]. This means that standard web development technologies such as JavaScript, HTML, and CSS are used when building the UI of a plugin. However, to keep the editor fast and responsive, Microsoft created the Language Server Protocol (LSP) [117]. LSP separates the often heavy work of code analysis from the user interface. The two communicate using JSON RPC. So, we built our language server to perform the business logic, and UI components communicate with it.

The plugin has different pages that correspond to different functionalities, related to steps and phases of the SAGED approach. For example, one page

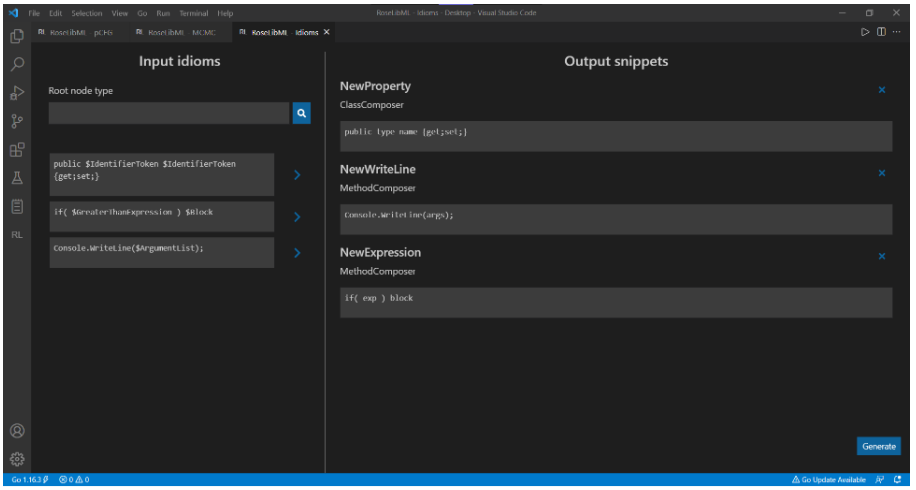

Figure 7.7: Visual Studio Code plugin, where a page for selecting code idioms and creating corresponding RoseLib methods is shown.

configures the inference parameters, another lets you review the inferred idioms, and another lets you select code idioms to generate a RoseLib method. Figure 7.7 shows the page for generating RoseLib methods, where it is specified how a method will be called, in which composer it will be included, and how the code snippet looks after configuration.

7.3.2 The Visualizer

As stated, the Visualizer provides insights into the inference results by allowing practitioners to browse input. While the basic Visual Studio Code plugin only lets users list and search for found code idioms, the Visualizer creates whole files that color and mark the found code idioms, giving users additional information in the familiar context of their codebase.

The algorithm of the Visualizer goes as follows: it iterates through the files, assigning unique hash values to each of the found idioms. If there are two instances of the same code idiom, they will receive the same hash value. While iterating, it creates a map, keeping track of how many such code idioms there are and their location.

Then, for each processed file, it creates an HTML file using the T4 templating

engine [97]. The end result is HTML specifically because it can easily be shown with the Visual Studio Code plugin. While creating the file, the map is used to retrieve the information about which hash code, color, and a set of links to other instances should be included with each separate code idiom.

7.3.3 Measurement Instrumentation

Understanding the codebase's repetitiveness and choosing the right set of parameters are not straightforward. It often requires some experimentation. The tool that we developed, named StatEval, can be used specifically for this purpose. It is an auxiliary tool that measures (i) precision — the percentage of inferred idioms found in the test corpora, (ii) coverage — the percentage of the test corpora covered by the inferred code idioms, and (iii) the average length of the inferred idioms.

StatEval is used for the evaluation of the SAGED approach presented in Chapter 8. To ensure trust in its results, we covered its functionalities with unit tests, and the code is available as open source [57].

8 Evaluation

In Section 1.2, we formulated the hypotheses that we aim to validate. With them in mind, we defined the following research questions that the evaluation aimed to answer:

- RQ1: Can the *Concept Discovery* phase of the SAGED approach produce relevant code idioms?
- RQ2: Is the *Concept Discovery* phase scalable?
- RQ3: Are API methods derived using the SAGED approach more efficient to use than existing ABG API methods?

In Section 8.1, we quantitatively measure the performance of the inference method on a large dataset to provide insights into the algorithm's behavior. Then, in Section 8.2, we perform a qualitative analysis to evaluate the usefulness of the inference output and the need for curation. Together, these answer **RQ1** and **RQ2**, validating hypotheses **H1** and **H2**.

To address RQ3, in Section 8.3, we compare the efficiency of using an ABG extended with API methods generated by the SAGED approach against the efficiency of using a basic, unmodified ABG for developing a custom MDSE tool. We also compare the obtained results with the results of our previous research conducted in industrial settings in collaboration with Schneider Electric's development center in Novi Sad [12]. Hypothesis **H3** is validated constructively through the successful implementation of the system described in Chapter 7, which is actively utilized to generate the customized ABGs tested in RQ3. A positive resolution to **RQ3** directly validates hypothesis **H4**.

This corresponds to the Demonstration and Evaluation steps of the DSRM. In DSRM terms, the artifact under evaluation is the SAGED approach together with its reference implementation, and the objectives of the solution against

which it is assessed are the hypotheses H1–H4, operationalized as research questions RQ1–RQ3.

8.1 Computing the Metrics for the Inference Performance

During the quantitative evaluation, we measured (i) **precision** — the percentage of inferred idioms found in the test corpora, (ii) **coverage** — the percentage of the test corpora covered by the inferred idioms, (iii) **average length of the inferred idioms**, and (iv) **scalability** — the relationship between the number of Lines of Code (LoC) in a project and the time needed to achieve burn-in. Metrics (i) and (ii) resemble precision and recall in information retrieval but are tailored for the code idiom mining domain [14]. The data were obtained using StatEval, the auxiliary tool we developed for this purpose, introduced in Section 7.3.3.

Methodology — To ensure a thorough analysis of the inference method's behavior, we performed a quantitative evaluation across a range of projects, as presented in Table 8.1. Initially, our dataset was based on three items, listed in the upper section of the table: the *Student projects dataset*, the *School Bus application*, and the *Licensing application*, all utilizing a similar technology stack. We manually inspected these projects to evaluate their repetitiveness for accurate result interpretation. Then, we added eleven open-source projects written in C#, introduced by Allamanis et al. [51]. These projects, listed in the lower portion of the table, enhance the generalizability of our dataset. The wide range of projects [1] enabled us to deduce reference values for metrics (i) and (ii). In the following paragraphs, we introduce the initial three dataset items we have chosen and then describe the setup for running the inference process.

The *Student projects dataset* [48] consists of sixteen projects developed by two-member student teams in a Web Development undergraduate course. These projects were not created specifically for this study. All teams received the same initial project structure and requirements for a Rent-A-Car web application. This collection can be viewed as a set of products in an SPL. The rationale behind choosing this dataset item is that automation is most cost-effective when the code has similarities and repetitions. The development process can be summarized as follows:

[1] To enhance the reproducibility, we combined all the projects into a Mendeley Data dataset at https://data.mendeley.com/datasets/6jmbv4gjz8/1

- The students were asked to implement various common functionalities for a web project. Tasks included data management and persistence, file handling, handling HTTP requests, WebSockets-based communication, data validation, and addressing security-related requirements.
- All projects were built using C# and the ASP.NET framework. The libraries used included Entity Framework as an object-relational mapper, Unity for dependency injection, and SignalR for real-time communication using web sockets.
- Students were encouraged to follow the same practices when faced with different problems but were also given the freedom to independently search for answers.

Both the *School Bus* and the *Licensing* applications are open-source projects representing web servers with REST API endpoints, developed under the supervision of the Government of British Columbia, Canada. The web server of the *School Bus application* [49] was included due to its well-defined layered structure and high level of repetitiveness within the layers. In contrast, the *Licensing application* [50] was added for its complexity and lack of clearly defined layers, which results in most of the business logic being placed within the controller layer.

To perform the inference, we split each project into training and testing corpora, using 70% of files for inference and the remaining 30% for testing the results. The files were cleansed, and imports were removed. We set the probability of designating a node as a fragment root during initial fragment creation to 0.85. The parameter for the geometric distribution in Equation 5.2 was set to 0.025. Based on experimental results, we assigned the value of alpha in Equation 5.10 to 5. The burn-in period for the run was set to 75. We established the threshold for the number of occurrences of a single code idiom at 2, expecting such an idiom to contain at least 8 nodes.

Results — The summary results across all dataset items are presented in the last row of Table 8.2. The calculated metric values indicate that, on average, **40%** of idioms inferred from the training corpora are found in the test corpora. Additionally, on average, **39%** of the test corpora AST nodes can be explained by the found idioms. Lastly, the average length of an idiom is **14 nodes**. The table also provides standard deviation values for the calculated metrics, giving a baseline for interpreting results for individual dataset items.

Table 8.2 also shows metric results for the three dataset items we manually inspected to verify their repetitiveness. Based on the results and our under-

Project	Git SHA	# LoC
Licensing App	`603b285`	40 115
School Bus App	`a29a056`	71 705
Student projects	`fbee3fb`	69 623
castleproject/Core	`28b8715`	87 285
JoshClose/CsvHelper	`852bd46`	36 154
dotliquid/dotliquid	`0398754`	17 246
libgit2/libgit2sharp	`50d6978`	46 406
apache/logging-log4net	`00cc486`	24 546
apache/lucenenet	`7401968`	607 098
mathnet/mathnet-numerics	`f196418`	221 184
etishor/metrics.net	`dee08fd`	14 599
mongodb/mongo-csharp-driver	`836aa60`	387 662
jdiamond/Nustache	`9d7d84e`	7723
Sandra/Sandra.Snow	`72b097d`	4532

Table 8.1: Projects on which we tested the inference process. Projects in the lower part of the table are introduced by Allamanis et al. [51]. The number of LoC of each item was calculated after it was cleansed for inference.

Project	**Precision** (%)	**Coverage** (%)	**Avg Length** (# of Nodes)
Licensing App	25.9	29.0	14.42
School Bus App	71.1	65.0	15.55
Student projects	75.7	69.5	16.75
castleproject/Core	57.4	43.3	13.95
JoshClose/CsvHelper	42.6	44.5	14.40
dotliquid/dotliquid	24.1	27.8	13.18
libgit2/libgit2sharp	25.6	35.3	12.57
apache/logging-log4net	26.9	22.4	12.93
apache/lucenenet	47.5	48.7	13.14
mathnet/mathnet-numerics	54.4	53.8	14.98
etishor/metrics.net	23.3	19.4	13.13
mongodb/mongo-csharp-driver	43.4	51.9	13.37
jdiamond/Nustache	21.3	24.8	13.64
Sandra/Sandra.Snow	22.6	12.1	14.01
Across All	**40.1 ± 18.8**	**39.1 ± 17.4**	**14.00 ± 1.15**

Table 8.2: Samples of measurement for all the dataset items. The averages and the standard deviations across all the dataset items are shown in the last row.

standing of the repetitiveness in these items, we conclude that the method correctly indicated that the *Licensing application* is less repetitive than the *School Bus application* and the *Student projects dataset*. Regarding the items first presented as a dataset in [51], a brief manual inspection led us to conclude that the results align with our expectations; the more repetitive projects yielded higher values.

Lastly, we present Figure 8.1 to illustrate scalability. The inference was performed on a computer with an Apple M1 Pro processor and 32 GB of RAM. The horizontal axis represents the number of LoC in a whole cleansed corpus, while the vertical axis shows the time needed to reach the burn-in set at 75 iterations. The results indicate that the time required for training increases linearly with the number of LoC. As shown in the graph, for the largest project with over 600,000 LoC after cleansing, it took 1.5 hours to complete 75 iterations of training.

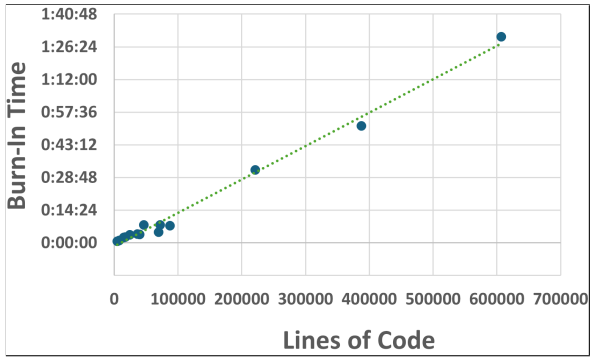

Figure 8.1: Time required to complete the training with a burn-in period set to 75 iterations compared to the number of LoC of the whole cleansed corpus

8.2 Qualitatively Evaluating the Inference Output

The quality of the inference output is measured by the number of inferred idioms of interest that need to be changed manually during curation to be considered useful for an ABG.

Methodology — We re-ran the inference process on the first three dataset items to identify idioms that may occur across the training dataset and those specific to certain layers. By examining the inheritance trees using the Preprocessor, we isolated groups of related files corresponding to the layers of an application. We also conducted a manual inspection to find groups not defined by inheritance. For each identified layer, we used a setup similar to that in Section 8.1. To obtain output that aligns with the RoseLib composers, we applied the configuration related to permanent cut nodes for the dataset, focusing on C# structural components, such as namespaces, classes, and methods.

Our team members, experienced in developing ASP.NET web applications but unfamiliar with the dataset projects, curated the inferred code idioms and measured the changes needed to make them useful. The curation focused on idioms for deriving APIs for custom MDSE tool development. The purpose of the MDSE tool was to automate the maintenance of a selected project from the dataset, covering all application layers. The team members singled out idioms deemed useful for automation. External developers with over five years

of experience in the technology stack assessed the value of the resulting idioms.

We analyzed the results for each dataset project. We will primarily focus on the *Student projects dataset* in the following text. This is because there is a large overlap between the technologies used for the *Student projects dataset* and those used for both the *School Bus application* and *Licensing application*, resulting in the most commonly used code idioms being the same.

Results — For the *Student projects dataset*, we identified eleven relevant groups of related artifacts. The curation process took two hours and resulted in over 50 selected code idioms addressing the intended use case. Additionally, **12%** of the identified idioms were slightly modified using the Handler to avoid overly generic idioms. Some idioms were combined to derive more suitable API methods. A team of external developers successfully recognized the semantics of the resulting idioms without any context provided.

Figure 8.2 presents several idioms selected from this dataset to demonstrate their properties. A new domain model class structure is shown in (a). To ensure that Entity Framework recognizes a newly added model class, the project's database context class must include its declaration. Many such declarations were inferred and are shown in (b). However, numerous inferred declarations were specific to the domains of the projects, reflecting the existence of multiple similar projects.

A new migration class file must be created once the model is added to the database context, as shown in (c). This allows for specifying changes to be made to the database schema. A function for paginated data retrieval is provided in (d). To facilitate data manipulation through REST API endpoints, the ASP.NET framework allows the addition of controller classes. These controllers are mostly defined in a similar way and contain various methods for HTTP operations. An example of a method that handles HTTP PUT requests is provided in (e). When updating information through an API, a database concurrency exception can occur. The idiom shown in (f) demonstrates how this exception is handled.

For the *School Bus application*, using a combination of the Preprocessor and manual inspection, we isolated eight application layers. Within them, various code idioms were discovered, similar to those found in the *Student projects dataset*. Again, some idioms were too generic and required improvement through curation. We conducted a qualitative analysis of the inferred idioms and concluded that they are relevant for extending the ABG with suitable API methods.

In the *Licensing application*, the controller layer highlighted the limitations of the inference method, which is highly dependent on the order of code elements. For instance, attributes placed over controller classes may not always be

Figure 8.2: Inferred and curated code idioms from the student dataset

in the same order, leading the inference method to create a generic code pattern that matches most cases, using a metavariable instead of a specific value. Another issue arose from the absence of a service layer, resulting in diverse and non-repetitive code in the controller layer, negatively impacting inference distributions.

8.3 Evaluating the Efficiency of Using the Derived API

To evaluate the efficiency of using the derived API methods, we compare their use to the challenges of developing custom MDSE tools using basic ABGs. The metrics we use to evaluate the efficiency are development time and the number of LoC.

Methodology — To calculate the metric values, we developed a custom MDSE tool that automated the maintenance of a single project from the *Student projects dataset* using both the basic and extended versions of RoseLib. Both versions of the tool generated code for seven different application layers. Both variants were implemented by the same developers from our team, who had

```
CompilationUnitComposer composer = new
CompilationUnitComposer();
var classComposer = composer
    .AddModelClass("vehicle_properties",
                   "VehicleProperties")
    .StartNavigating()
    .SelectClassDeclaration()
    .StartComposing<ClassComposer>()
foreach(var property in properties){
  classComposer.AddProperty(property);
}
var output = classComposer.GetCode();
```

a)

```
CompilationUnitNavigator navigator =
    new CompilationUnitNavigator(reader);
var updatedDbContextCode = navigator
                        .SelectClassDeclaration(
                            "IdentityDbContext"
                        )
                        .StartComposing<ClassComposer>()
                        .AddDbSet("VehicleProperty",
```

b)

```
CompilationUnitComposer composer =
    new CompilationUnitComposer();
composer.AddControllerClass(
            "VehiclePropertiesController"
        )
       .EnterClass()
       .AddPutMethod("api/vehicle-properties/{id}",
                     "PutVehicleProperty",
                     "VehicleProperty property")
                    .GetCode();
```

c)

```
CompilationUnitNavigator navigator = new
CompilationUnitNavigator(reader);
navigator
    .SelectMethodDeclaration("PutVehicleProperty")
    .EnterBody()
    .InsertConcurrencyExceptionHandling(
        "unitOfWork.VehicleProperties.Update(property)"
    )
    .GetCode();
```

d)

Figure 8.3: Examples of using API methods derived from curated idioms

more than two years of prior experience with Roslyn and the basic version of RoseLib from the industrial project described below. The team developed the basic variant first, then the extended variant. Development time reflects the total person-hours spent implementing the generators for all seven layers, and the LoC count covers the generator code written against the ABG API.

Examples of derived API methods based on the idioms from Section 8.2 used in the MDSE tool are shown in Figure 8.3. In a), a newly added API method called *AddModelClass* generates a model class named *VehicleProperties*. In b), the existing RoseLib API method *SelectClassDeclaration* selects a class named *IdentityDbContext*, and the newly added API method *AddDbSet* generates and inserts code into the *IdentityDbContext*, registering the generated class *VehicleProperties* with Entity Framework. In c), we demonstrate how RoseLib's methods can be combined to create larger code constructs. Two newly added API methods are combined to generate the new *VehiclePropertiesController* REST controller class and insert the *PutVehicleProperty* method to handle HTTP PUT requests. Lastly, in d), we illustrate how the *PutVehicleProperty* method is extended to handle concurrency exceptions using the newly added *InsertConcurrencyExceptionHandling* API method.

As a second part of evaluating the efficiency, we estimate the time savings of using an extended ABG by comparing the person-hours needed to develop a code generator for a specific file artifact based on our previous research [12].

In collaboration with Schneider Electric's development center in Novi Sad, we built a complex, custom MDSE tool to automate the development of their large-scale SPL [12]. This tool allowed users to model their domain and automate the development of SPL instances. The industrial context served as a valuable example of a case where established MDSE practices had to be adapted for a successful adoption, and where we used ABGs to facilitate a seamless workflow. The development process took over two years, during which we created several hundred code generators within the MDSE tool, each handling one type of artifact. Approximately one hundred of these generators managed C# artifacts using Roslyn and the initial version of RoseLib. The architecture of the SPL applications was well-defined, with code generation focusing on files with repetitive structural elements [12].

Results — For the *Student projects dataset*, the difference in written LoC when using the basic ABG and the ABG extended with the derived API methods depends heavily on the size and structure of the code idioms used. Generally, the more complex the selected code idiom, the greater the benefits. When using the extended version of RoseLib to build the MDSE tool, we wrote **24%** fewer LoC compared to the basic version [2]. Similarly, we measured the time savings for developing the tools, and it took around **20%** less time to develop the tool using the extended ABG.

We also assessed time savings with respect to our results of the research conducted in collaboration with Schneider Electric. On average, in that setting, it took 20 person-hours to develop a code generator for a specific file artifact using ABGs with APIs on lower abstraction levels [12]. The code generator typically had to generate or modify several code segments of the given artifact type using multiple ABG API methods. With the SAGED approach, developing a similar code generator took an average of **15.5 person-hours**. Notably, here we also observed fewer LoC per code generator. However, we do not report an exact figure for this setting. The industrial generators combined ABG calls with template-based code generation, which was introduced precisely to mitigate the difficulties of working with Roslyn's Syntax Tree API. As a result, their LoC are not directly comparable to those of generators written purely against the extended ABG, and the LoC comparison remains qualitative.

[2] The examples of both the basic and the extended ABG being used to generate the code for the layers can be found via the following link: https://github.com/nenadTod/RoseLib/tree/master/Tests/Examples

8.4 Discussion

Based on the findings in Sections 8.1 and 8.2, we conclude that the answer to *RQ1: Can the Concept Discovery phase of the SAGED approach produce relevant code idioms?* is affirmative. The inference method successfully indicated the repetitiveness of the three selected projects in the dataset. We inferred, identified, and curated code idioms, which external developers evaluated and confirmed as significant. Additionally, we found that the implementation facilitating the *Concept Discovery* phase is scalable, as the training time grew linearly with the number of LoC, answering *RQ2*.

Regarding *RQ3: Are API methods derived using the SAGED approach more efficient to use than existing ABG API methods?*, we also reached an affirmative conclusion. We demonstrated efficiency through development time and the number of LoC written. The extended ABG API supported more efficient development of code generators with fewer LoC compared to both custom-made MDSE tools: the demo tool for automating maintenance of the student project for Rent-A-Car and the complex tool designed for industrial settings.

Through the affirmative resolution of these research questions, we have systematically validated our hypotheses. By confirming the derived hypotheses **H1** through **H3**, we successfully validated the assumptions underlying the general hypothesis, **H0**. Furthermore, a positive resolution to **RQ3** directly validates hypothesis **H4**, providing concrete measurements that demonstrate the efficiency and practical value of the SAGED approach.

The evaluation identified several challenges with the SAGED approach:

- The curation process could be improved. The tooling provides helpful information for determining the value of an idiom but does not assist in selecting the most useful version among similar idioms. Introducing idiom ranking and search functionality would help mitigate this issue.
- The inference method is sensitive to the order of code elements. We added sorting functionality to the Preprocessor to help in cases where the order of code elements is not important, which is consistent with the iterative nature of the DSRM process (the identified shortcomings should inform subsequent artifact design). However, further enhancements are needed for imperative code; improvements listed in Section 3.3 aim to address this issue.
- The SAGED approach is most effective for applications with established coding conventions and repetitive codebases. This was not the case with

the *Licensing application*, leading to some idioms not being identified. Manual curation can help identify idioms in applications without established coding conventions and with high variability, but it requires greater involvement from practitioners.

Given the promising results from the evaluation of the SAGED approach, we will strive to further enhance its effectiveness by addressing these challenges. The threats to validity discussed in the following section address the rigor of the evaluation, which DSRM requires to be communicated alongside the results.

8.5 Threats to Validity

We have identified several threats to the validity of our study. The first threat pertains to the accuracy of the implementation of the Bayesian PTSG described in this thesis. To address this concern, multiple team members thoroughly reviewed the source code and wrote unit tests that cover the algorithms and various mechanisms. Unfortunately, we cannot directly compare the obtained inference results to those reported in the original paper [14], as their datasets are different and in Java, while our implementation currently only supports C#.

The next concern is the suitability of the projects in the dataset for quantitative and qualitative evaluation. To ensure validity, in addition to student projects from our department, we utilized two external open-source projects from British Columbia and eleven projects provided by Allamanis et al. [51]. All of these were used for quantitative evaluation. For qualitative evaluation, we selected the *Student projects dataset*, *School Bus application*, and *Licensing application*, and performed the whole *Concept Discovery phase*. External developers assessed the value of the resulting idioms. By using projects of different origins, sizes, and qualities, we were able to draw conclusions and highlight the shortcomings of the SAGED approach.

The final threat concerns our objectivity in measuring the efficiency of using API methods automatically derived from curated idioms. To tackle this, we created a demo MDSE tool that automates the maintenance of a selected student project by utilizing API methods derived from curated code idioms. We measured the average development time using SAGED and the number of LoC produced per code generator for one type of artifact. We then compared these results with those obtained from a complex MDSE tool developed in collaboration with Schneider Electric's development center in Novi Sad, with the LoC comparison limited to a qualitative one, for the reason given in Section 8.3.

Despite significant differences in complexity between the two tools, we believe this comparison is valid due to the use of the same technologies, similar project settings, and the involvement of the same developers.

Since the basic variant of the demo tool was built before the extended one, a learning effect could, in principle, inflate the measured time savings. We consider this effect limited to familiarity with the target project, because the developers' proficiency with Roslyn and the basic RoseLib API had been established over two years of industrial use.

9 Conclusions

API-based generators offer a way to introduce MDSE to automate the development of existing software projects while keeping their architecture and workflow intact. They allow developers to make changes to specific parts of the code while maintaining syntax correctness. However, ABGs have their challenges. Creating them to support larger and more complex systems is time-consuming and labor-intensive. Furthermore, an understanding of the complex syntax rules of a GPL is required. To enhance the usability of ABGs, it is important to define the API in terms of higher-level concepts that go beyond syntax-related elements, enabling easier and more concise code manipulation.

One approach to defining these higher-level concepts is by identifying and analyzing code idioms of an existing project. Code idioms represent recurring code fragments with a single semantic role found in existing codebases. Leveraging idioms allows for tailoring ABGs to specific use cases, making automation more accessible and practical.

The thesis presents the SAGED approach for developing ABGs. This approach simplifies the development process by i) utilizing the non-parametric Bayesian probabilistic tree substitution grammar for inferring code idioms from unlabeled source code, ii) enabling manual curation of suitable code idioms through a provided graphical tool, and iii) automating the creation of ABG API methods for curated idioms.

9.1 Contributions of the Thesis

The main contributions of this thesis include the introduction of the SAGED approach, an explanation and adaptation of the Type-Based MCMC as a method for approximating the non-parametric Bayesian PTSG, and the development of

an open-source inference core for implementing the inference method in different programming languages. They can be grouped into three categories: methodological, technical, and empirical.

Methodological contributions. SAGED is a novel approach for simplifying the process of creating code-generating programs tailored to specific solutions (Chapter 4). It consists of two phases: Phase 1 — Concept Discovery, aimed at identifying code idioms that are good candidates for code generation, and Phase 2 — ABG Extension, for automating the creation of API methods based on the identified idioms.

To the best of our knowledge, the research on how to successfully automate the development of ABGs and bring them closer to the domain problem was missing from the literature, and no papers trying to use code idioms to automatically develop API-based generators have been uncovered during our literature review.

The approach also specifies the requirements that the resulting ABG should comply with (Section 4.2).

Within the inference step, the thesis systematizes the available literature on code idiom inference methods. As far as we know, there was no explanation in the existing literature on how to practically apply the Type-Based MCMC method to the domain of mining code idioms. In this thesis, we have provided a detailed explanation, the necessary steps to perform it, and its open-source implementation (Chapter 5).

In particular, we modified the general-purpose definition of a type to the domain of PTSGs, defining it through the fragment triplet (f_{join}, f_t, f_s). This simplification allows us to store only partial information about the node's surrounding fragments, rather than the entire vector of changes, which significantly reduces the need for bookkeeping and, consequently, the impact on memory.

Technical contributions. The needed components reflecting the design of the approach were implemented for the C# programming language and released as open source (Chapter 7):

- the language-independent inference core [55], which can be extended to different programming languages;
- the Inferencer, which extends the inference core with the possibility to infer code idioms from a codebase written in C#;

- the Preprocessor, which cleanses the input files and groups them based on the inheritance tree;
- the Visualizer, which allows practitioners to see found code idioms and navigate through the code;
- the Handler, which allows practitioners to manipulate code idioms in a controlled manner; and
- the Transformer with its Knowledge base, which generates appropriate API methods and integrates them with the ABG basis.

From the practitioner's perspective, the main tool is the Visual Studio Code plugin [56], which integrates the approach seamlessly into the existing developer toolkit. We also developed StatEval [57], an auxiliary tool that measures precision, coverage, and the average length of the inferred idioms.

The solution is built upon RoseLib [30], our ABG that provides a layer above Roslyn's Syntax Tree API (Chapter 6). It provides traceability links, implemented through the CSPath language, where a textual expression saved outside of the codebase can be used to locate code elements within it, and integrity checks, a mechanism that facilitates checking whether previously generated code has possibly been altered between code generation cycles.

Empirical contributions. The inference process was tested on fourteen projects. On average, 40% of idioms inferred from the training corpora were found in the test corpora, 39% of the test corpora AST nodes could be explained by the found idioms, and the average length of an idiom was 14 nodes. The time required for training increased linearly with the number of LoC. It took 1.5 hours to complete 75 iterations of training for the largest project with over 600,000 LoC.

To enhance reproducibility, we combined all the projects into a publicly available Mendeley Data dataset.

In the qualitative evaluation, the curation process took two hours and resulted in over 50 selected code idioms; only 12% of the identified idioms were slightly modified to avoid overly generic idioms, and a team of external developers successfully recognized the semantics of the resulting idioms without any context provided.

Finally, when using the extended version of RoseLib to build the MDSE tool, we wrote 24% fewer LoC, and it took around 20% less time compared to the basic version, while developing a code generator for a specific file artifact

took an average of 15.5 person-hours with the SAGED approach, compared to 20 person-hours in our earlier industrial research [12].

9.2 Validation of the Hypotheses

We evaluated the suitability of the SAGED approach. To examine the effectiveness of the inference method, we first statistically evaluated the capabilities of the inference method on a range of different projects. Then, we assessed the quality of the inferred idioms on (i) a set of student projects [48] to cover a scenario when there is a group of related projects, such as in the case of an SPL, and (ii) two comparable open-source projects developed under the supervision of the Government of British Columbia. We inspected and curated inferred idioms and invited external developers to validate their quality. Finally, we investigated scenarios where curated code idioms are utilized to obtain new ABG API methods. The obtained ABG methods were used to create an MDSE tool that automates further development of one of the student projects. This allowed us to draw conclusions about the efficacy of development using extended ABGs.

Table 9.1 summarizes how each hypothesis stated in Section 1.2 was validated and where the supporting evidence is presented.

Viewed through the lens of the DSRM that guided this research:

- The problem and the objectives of a solution were established through the theoretical examinations in Chapters 2 and 3.
- The artifacts (the SAGED approach, the language-independent inference core, its C# extension, and the extended RoseLib ABG) were designed and developed in Chapters 4, 5, 6, and 7.
- Their utility was demonstrated and evaluated in Chapter 8; the resulting validation of the hypotheses is summarized in Table 9.1.
- This thesis, its publications, and the open-source repositories fulfill the remaining DSRM step (communication), by making the results available to academia and industry. This also enables the results to be reproduced.

9.3 Implications for Research and Practice

The automated development of an ABG for C# can relieve practitioners from needing to understand ABG implementation details, thereby reducing the effort

Hyp.	Validation	Evidence
H1	It is possible to semi-automatically identify relevant code idioms in the given codebase (RQ1).	Quantitative evaluation on fourteen projects (Section 8.1); qualitative evaluation with external validation on three projects (Section 8.2).
H2	The identification of code idioms can be performed in a scalable manner (RQ2).	Training time grows linearly with the number of LoC, 1.5 h for over 600,000 LoC (Section 8.1); two-hour curation for the Student projects dataset (Section 8.2).
H3	It is possible to automate the development of ABGs based on the identified code idioms.	Validated constructively: the identified code idioms are used to automatically derive API methods that extend the ABG basis (Chapter 7, Section 8.3).
H4	API methods automatically derived from relevant code idioms are more efficient in terms of development time and the amount of code required (RQ3).	24% fewer LoC and around 20% less development time; 15.5 vs. 20 person-hours per code generator compared with the industrial baseline (Section 8.3).
H0	It is possible to create a solution for automating the development of API-based code generators tailored to the technical domain specified by the given codebase.	Follows from H1–H4 and the successful implementation of the system described in Chapter 7.

Table 9.1: Summary of hypothesis validation

required to develop them.

The set of open-source tools specialized for the C# programming language can be utilized to find and visualize code idioms, providing insights into codebase repetitiveness, while the tools for measuring the performance of the inference method can quantify this repetitiveness.

The SAGED approach, in general, can facilitate the adoption of the MDSE paradigm across various scenarios, including the automation of mature software projects that traditional MDSE approaches have not previously covered.

For researchers, in contrast to approaches based on artificial neural networks, whose black-box nature means that learned rules cannot be easily modified, tree substitution grammars are relatively easy to comprehend, and the resulting code idioms can be easily altered. Together with the published dataset and the

measured metric values, this provides a baseline for future enhancements of the inference method.

9.4 Future Work

The main goal of our future work is to overcome the observed constraints and reduce the need for manual curation by further automating the process of code idiom inference. Two actions can be taken in that regard. First, the existing tooling can be improved to provide a better overview of inferred code idioms and facilitate easier manipulation of code idioms. Second, as mentioned in Section 3.3, the inference method that relies on syntax to retrieve concepts has limitations, as the resulting code idioms tend to be shallow. To address this, we plan to explore enhancements to the inference method and investigate other techniques that consider code semantics.

We also intend to extend our implementation with support for different programming languages. The inference method is language-agnostic, and so is our implementation of the inference core. As a part of our research, we extended it to support inference from C# files as described in Section 7.1.2, but given a suitable parser, it can be extended to support any language.

There are also opportunities to further evaluate the SAGED approach and its results. So far, the approach has been evaluated in terms of the LoC that needs to be written and the time required to write the code. However, given the expectations developers might have of a resulting ABG, it would be beneficial to conduct a qualitative evaluation similar to how Domain-Specific Languages (DSLs) are evaluated [118]. In essence, performing such an evaluation would provide insight into user experience in relation to usability, reliability, and productivity. Additionally, using such an ABG might improve the maintainability of code-generating tools built with it.

Lastly, in our opinion, the presented research can provide the basis for interesting research paths. Given the significant effort required to analyze a specific software solution and produce a specialized ABG for it, it is highly beneficial to consider the reusability of these generators across other solutions within similar technical domains. As demonstrated during the qualitative evaluation of the inference output in Section 8.2, a substantial overlap in identified code idioms was observed among the student projects, as well as between those projects and the *School Bus* application. Consequently, there is a clear opportunity to leverage the development effort invested in one codebase to reduce the overhead required for another.

Another, more distant research path is related to LLMs. The rapid success of LLM-based programming tools such as Claude Code [119] and OpenAI Codex [120] across the broader programming community has demonstrated strong interest and motivation to automate programming tasks. As already discussed in this thesis, it remains unclear how the unpredictability of stochastic LLMs could be mitigated. It could be beneficial to integrate LLM-based tools, such as Claude Code, with more formal development methods, such as those provided by MDSE, where transformations performed by LLMs could be serialized and represented as reusable and predictable ABG expressions.

References

[1] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[2] Object Management Group. MDA guide version 1.0.1. Technical Report omg/2003-06-01, Object Management Group, 2003.

[3] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[4] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[5] Andreas Vogelsang, Tiago Amorim, Florian Pudlitz, Peter Gersing, and Jan Philipps. Should I stay or should I go? on forces that drive and prevent MBSE adoption in the embedded systems industry. In *International Conference on Product-Focused Software Process Improvement*, pages 182–198. Springer, 2017.

[6] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, et al. A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 74–85. IEEE, 2015.

[7] Carlo Bernaschina, Emanuele Falzone, Piero Fraternali, and Sergio Luis Herrera Gonzalez. The virtual developer: Integrating code genera-

tion and manual development with conflict resolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–38, 2019.

[8] Markus Völter. A catalog of patterns for program generation. In *EuroPLoP*, pages 285–320, 2003.

[9] Sreenivasa Viswanadha, Júlio Vilmar Gesser, and Danny van Bruggen. JavaParser. `https://javaparser.org`. [Accessed 6-Jul-2026].

[10] .NET Foundation. Roslyn — the .NET Compiler Platform. `https://dotnetfoundation.org/projects/project-detail/dotnet-compiler-platform`. [Accessed 6-Jul-2026].

[11] Nenad Todorović, Bojana Dragaš, and Gordana Milosavljević. Supporting integrative code generation with traceability links and code fragment integrity checks. In Miroslav Trajanović, Nenad Filipović, and Milan Zdravković, editors, *Disruptive Information Technologies for a Smart Society: Proceedings of the 13th International Conference on Information Society and Technology (ICIST 2023)*, volume 872 of *Lecture Notes in Networks and Systems*, pages 490–501, Cham, 2024. Springer.

[12] Bojana Dragaš, Nenad Todorović, Tijana Rajačić, and Gordana Milosavljević. SeamlessMDD: Framework for seamless integration of generated and hand-written code. In *International Conference on Web Engineering*, pages 163–177. Springer, 2024.

[13] Bojana Dragaš, Nenad Todorović, Tijana Rajačić, Gordana Milosavljević, and Željko Vuković. A novel approach to integration of manual changes in generated code: SeamlessMDD. *Journal of Web Engineering*, 24(4):499–528, 2025.

[14] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483, 2014.

[15] Zejun Zhang, Zhenchang Xing, Xiaoxue Ren, Qinghua Lu, and Xiwei Xu. Refactoring to Pythonic idioms: A hybrid knowledge-driven approach leveraging large language models. *Proceedings of the ACM on Software Engineering*, 1(FSE):1107–1128, 2024.

[16] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. Inducing tree-substitution grammars. *The Journal of Machine Learning Research*, 11:3053–3096, 2010.

[17] Jordy Cabot. Clarifying concepts: MBE vs MDE vs MDD vs MDA. https://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/. [Accessed 10-Jun-2026].

[18] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[19] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic notes in theoretical computer science*, 152:125–142, 2006.

[20] Loli Burgueño, Jordi Cabot, Shuai Li, and Sébastien Gérard. A generic LSTM neural network architecture to infer heterogeneous model transformations. *Software and Systems Modeling*, 21(1):139–156, 2022.

[21] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52:43–62, 2018.

[22] Alan W Brown, Jim Conallen, and Dave Tropeano. Introduction: Models, modeling, and model-driven architecture (MDA). In *Model-Driven Software Development*, pages 1–16. Springer, 2005.

[23] Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. Learning programmatic idioms for scalable semantic parsing. *arXiv preprint arXiv:1904.09086*, 2019.

[24] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98, 2016.

[25] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic API mining across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 254–265, 2016.

[26] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.

[27] Hoang Son Pham, Siegfried Nijssen, Kim Mens, Dario Di Nucci, Tim Molderez, Coen De Roover, Johan Fabry, and Vadim Zaytsev. Mining patterns in source code using tree mining algorithms. In *International Conference on Discovery Science*, pages 471–480. Springer, 2019.

[28] Square. JavaPoet. `https://github.com/square/javapoet`. [Accessed 6-Jul-2026].

[29] Testura. Testura.Code. `https://github.com/Testura/Testura.Code`. [Accessed 6-Jul-2026].

[30] Aleksandar Lukić and Nenad Todorović. RoseLib. `https://github.com/nenadTod/RoseLib`. [Accessed 10-Jun-2026].

[31] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[32] Percy Liang, Michael I Jordan, and Dan Klein. Type-based MCMC. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 573–581, 2010.

[33] Gino Chénard, Ismaïl Khriss, and Aziz Salah. Towards the automatic discovery of platform transformation templates of legacy object-oriented systems. In *Proceedings of the 6th International Workshop on Models and Evolution*, pages 51–56, 2012.

[34] Wolf Rost. Mining of DSLs and generator templates from reference applications. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–7, 2020.

[35] Kevin Lano and Qiaomu Xue. Code generation by example. In *MODELSWARD*, pages 84–92, 2022.

[36] Kevin Lano and Qiaomu Xue. Code generation by example using symbolic machine learning. *SN Computer Science*, 4(2):170, 2023.

[37] Mathieu Acher and Jabier Martinez. Generative AI for reengineering variants into software product lines: An experience report. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume B*, pages 57–66, 2023.

[38] Benoit Combemale, Jeff Gray, and Bernhard Rumpe. Large language models as an "operating" system for software and systems modeling. *Software and Systems Modeling*, 22(5):1391–1392, 2023.

[39] Trevor Cohn, Sharon Goldwater, and Phil Blunsom. Inducing compact but accurate tree-substitution grammars. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 548–556, 2009.

[40] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156, 2010.

[41] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, (6):721–741, 1984.

[42] Xiaochang Peng and Daniel Gildea. Type-based MCMC for sampling tree fragments from forests. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1735–1745, 2014.

[43] Nenad Todorović, Aleksandar Lukić, Bojana Zoranović, Renata Vaderna, Željko Vuković, and Sebastijan Stoja. RoseLib: A library for simplifying .NET Compiler Platform usage. In *Konjović, Z., Zdravković, M., Trajanović, M. (Eds.) ICIST 2018 Proceedings Vol.1, pp.216-221, 2018*, pages 216–221, 2018.

[44] Object Management Group. MOF model to text transformation language, v1.0. https://www.omg.org/spec/MOFM2T/1.0/PDF. [Accessed 10-Jun-2026].

[45] James Clark, Steve DeRose, et al. XML path language (XPath), 1999.

[46] Igor Dejanović, Renata Vaderna, Gordana Milosavljević, and Željko Vuković. TextX: a Python tool for domain-specific languages implementation. *Knowledge-based systems*, 115:1–4, 2017.

[47] Steve Vinoski. Chain of responsibility. *IEEE Internet Computing*, 6(6):80–83, 2002.

[48] Nenad Todorović. Student projects. `https://github.com/nenadTod/student-projects.git`, 2024. [Accessed 10-Jun-2026].

[49] Government of British Columbia. School bus application. https://github.com/bcgov/schoolbus/tree/master/Server/SchoolBusAPI. [Accessed 6-Jul-2026].

[50] Government of British Columbia. Licensing application. https://github.com/bcgov/jag-lcrb-carla-public/tree/develop/cllc-public-app. [Accessed 6-Jul-2026].

[51] Miltiadis Allamanis, Earl T Barr, Christian Bird, Premkumar Devanbu, Mark Marron, and Charles Sutton. Mining semantic loop idioms. *IEEE Transactions on Software Engineering*, 44(7):651–668, 2018.

[52] Jan Vom Brocke, Alan Hevner, and Alexander Maedche. Introduction to design science research. In *Design science research. Cases*, pages 1–13. Springer, 2020.

[53] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, Berlin, Heidelberg, 2014.

[54] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[55] Nenad Todorović, Aleksandar Lukić, and Marijana Kološnjaji. RoseLibML. https://github.com/lukic-aleksandar/RoseLibML. [Accessed 10-Jun-2026].

[56] Marijana Kološnjaji. Grafičko okruženje za pomoć pri otkrivanju idioma u programskom kodu [a graphical environment that facilitates code idiom mining]. Master Thesis, Faculty of Technical Sciences, University of Novi Sad, 2021.

[57] Nenad Todorović. StatEval. https://github.com/lukic-aleksandar/RoseLibML/tree/master/StatEval. [Accessed 10-Jun-2026].

[58] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13, 2020.

[59] Luis Jiménez-Navajas, Ricardo Pérez-Castillo, and Mario Piattini. Code generation for classical-quantum software systems modeled in UML. *Software and Systems Modeling*, pages 1–27, 2025.

[60] Miroslaw Staron. Adopting model driven software development in industry–a case study at two companies. In *International Conference on Model Driven Engineering Languages and Systems*, pages 57–72. Springer, 2006.

[61] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[62] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A. Myers. Improving API documentation using API usage information. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 119–26. IEEE, 2009.

[63] Martin P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Software*, 26(6), 2009.

[64] Eban Escott. Jidoka: automation with a human touch. *Software and Systems Modeling*, pages 1–20, 2024.

[65] Roland N Ibbett. The University of Manchester MU5 project. *IEEE Annals of the History of Computing*, 21(1):24–33, 2002.

[66] FP BROOKS. No silver bullet-essence and accidents of software engineering. In *Proc. IFIP 10th World Computer Congress*, pages 1069–1076. North-Holland, Amsterdam, 1986.

[67] Brian Randell. The 1968/69 NATO software engineering reports. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html, 1996. [Accessed 30-Jun-2026].

[68] Frederick P Brooks Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[69] Ian Sommerville. *Software Engineering*. Pearson, Boston, 10th edition, 2016.

[70] Steven D Fraser, Frederick P Brooks Jr, Martin Fowler, Ricardo Lopez, Aki Namioka, Linda Northrop, David Lorge Parnas, and David Thomas.

No silver bullet reloaded: Retrospective on essence and accidents of software engineering. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 1026–1030, 2007.

[71] Ben Moseley and Peter Marks. Out of the tar pit. *Software Practice Advancement (SPA)*, 2006, 2006.

[72] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an LLM to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[73] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli calling international conference on computing education research*, pages 1–12, 2023.

[74] MohammadHadi Dehghani, Shekoufeh Kolahdouz-Rahimi, Massimo Tisi, and Dalila Tamzalit. Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning. *Software and Systems Modeling*, 21(3):1115–1133, 2022.

[75] Fabiano C Ferrari, Vinicius HS Durelli, Sten F Andler, Jeff Offutt, Mehrdad Saadatmand, and Nils Müllner. On transforming model-based tests into code: A systematic literature review. *Software Testing, Verification and Reliability*, 33(8):e1860, 2023.

[76] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Relevance, benefits, and problems of software modelling and model driven techniques—a survey in the Italian industry. *Journal of Systems and Software*, 86(8):2110–2126, 2013.

[77] Håkan Burden, Rogardt Heldal, and Jon Whittle. Comparing and contrasting model-driven engineering at three large companies. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2014.

[78] Hessa Alfraihi and Kevin Lano. Trends and insights into the use of model-driven engineering: a survey. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 286–295. IEEE, 2023.

[79] Jeff Rothenberg. The nature of modeling. In Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen, editors, *Artificial Intelligence, Simulation, and Modeling*, pages 75–92. John Wiley & Sons, New York, 1989.

[80] Martin Fowler. UML modes. https://martinfowler.com/bliki/UmlMode.html. [Accessed 10-Jun-2026].

[81] Saïd Assar. Model driven requirements engineering: Mapping the field and beyond. In *2014 IEEE 4th International Model-Driven Requirements Engineering Workshop (MoDRE)*, pages 1–6. IEEE, 2014.

[82] Andreas Metzger. A systematic look at model transformations. In *Model-driven Software Development*, pages 19–33. Springer, 2005.

[83] Object Management Group. Model-driven architecture. https://www.omg.org/mda/. [Accessed 10-Jun-2026].

[84] Cambridge University Press. Meaning of conceptualization in English. https://dictionary.cambridge.org/dictionary/english/conceptualization, 2026. [Accessed 09-Apr-2026].

[85] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien, New York, 1973.

[86] Erik Johannes Burger. Flexible views for view-based model-driven development. In *Proceedings of the 18th international doctoral symposium on Components and architecture*, pages 25–30, 2013.

[87] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[88] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM systems journal*, 45(3):621–645, 2006.

[89] Kevin Lano, Shekoufeh Kolahdouz-Rahimi, Sobhan Yassipour-Tehrani, and Mohammadreza Sharbaf. A survey of model transformation design patterns in practice. *Journal of Systems and Software*, 140:48–73, 2018.

[90] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Model-transformation design patterns. *IEEE Transactions on Software Engineering*, 40(12):1224–1259, 2014.

[91] Charles Rich and Richard C. Waters. Automatic programming: Myths and prospects. *Computer*, 21(8):40–51, 1988.

[92] Qiaomu Xue and Kevin Lano. Comparing LLM-based and MDE-based code generation for agile MDE. *AgileMDE 2025, STAF 2025*, 2025.

[93] Lola Burgueño, Davide Di Ruscio, Houari Sahraoui, and Manuel Wimmer. Automation in model-driven engineering: A look back, and ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–25, 2025.

[94] Eclipse Foundation. Eclipse Xpand. `https://projects.eclipse.org/projects/modeling.xpand`. [Accessed 06-Jul-2026].

[95] Eclipse Foundation. Acceleo. `https://eclipse.dev/acceleo/`. [Accessed 06-Jul-2026].

[96] Apache. Apache FreeMarker. `https://freemarker.apache.org`. [Accessed 06-Jul-2026].

[97] Microsoft. T4 templating engine. `https://learn.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates`. [Accessed 10-Jun-2026].

[98] Pallets. Jinja. `https://jinja.palletsprojects.com/en/stable/`. [Accessed 06-Jul-2026].

[99] OW2. ASM OW2. https://asm.ow2.io, 2026. [Accessed 01-Jul-2026].

[100] Rafael Winterhalter. Byte Buddy. https://bytebuddy.net/#/, 2026. [Accessed 01-Jul-2026].

[101] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, 18:2361–2397, 2019.

[102] Joao PL De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. KernelFaRer: replacing native-code idioms with high-performance library calls. *ACM Transactions On Architecture And Code Optimization (TACO)*, 18(3):1–22, 2021.

[103] Aishwarya Sivaraman, Rui Abreu, Andrew Scott, Tobi Akomolede, and Satish Chandra. Mining idioms in the wild. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 187–196, 2022.

[104] Niranjan Hasabnis and Justin Gottschlich. ControlFlag: a self-supervised idiosyncratic pattern detection system for software control structures. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, pages 32–42, 2021.

[105] Kim Mens, Siegfried Nijssen, and Hoang-Son Pham. The good, the bad, and the ugly: mining for patterns in student source code. In *Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*, pages 1–8, 2021.

[106] Yun Chi, Yirong Yang, Yi Xia, and Richard R Muntz. CMTreeMiner: Mining both closed and maximal frequent subtrees. In *Pacific-asia conference on knowledge discovery and data mining*, pages 63–73. Springer, 2004.

[107] Dmitry Orlov. Finding idioms in source code using subtree counting techniques. In *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part II 9*, pages 44–54. Springer, 2020.

[108] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroshi Sakamoto, Hiroki Arimura, and Setsuo Arikawa. Efficient substructure discovery from large semi-structured data. *IEICE TRANSACTIONS on Information and Systems*, 87(12):2754–2763, 2004.

[109] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. An LSTM-based neural network architecture for model transformations. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 294–299. IEEE, 2019.

[110] Martin Eisenberg, Hans-Peter Pichler, Antonio Garmendia, and Manuel Wimmer. Towards reinforcement learning for in-place model transformations. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 82–88. IEEE, 2021.

[111] Quentin Brilhault, Esma Yahia, and Lionel Roucoules. Reinforcement learning for model transformations to support model-driven plug-and-play interoperability. *Journal of Industrial Information Integration*, 52:101150, 2026.

[112] ADM Task Force. Architecture-driven modernization roadmap. Technical report, Technical report, OMG, 2006. Draft, 2006.

[113] Kevin Lano, Qiaomu Xue, and Shekoufeh Kolahdouz-Rahimi. Agile specification of code generators for model-driven engineering. *Proceedings of ICSEA 2020*, pages 9–15, 2020.

[114] OpenAI. GPT-4. `https://openai.com/gpt-4`. [Accessed 06-Jul-2026].

[115] Microsoft. Visual Studio Code. `https://code.visualstudio.com`. [Accessed 10-Jun-2026].

[116] OpenJS Foundation. Electron. `https://www.electronjs.org`. [Accessed 10-Jun-2026].

[117] Microsoft. Language Server Protocol. `https://langserver.org`. [Accessed 10-Jun-2026].

[118] Gökhan Kahraman and Semih Bilgen. A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, 14(4):1505–1526, 2015.

[119] Anthropic. Claude Code. https://claude.com/product/claude-code. [Accessed 30-Jun-2026].

[120] OpenAI. Codex. https://openai.com/codex/. [Accessed 30-Jun-2026].

Biography

Nenad Todorović was born in 1993 in Subotica, where he graduated from the Svetozar Marković Gymnasium in 2012. He then enrolled at the Faculty of Technical Sciences, University of Novi Sad, majoring in Computer Science and Automation, and graduated in 2016 as a recipient of the Dositej Obradović scholarship of the Fund for Young Talents. He continued with master's studies at the same faculty and in 2017 defended his master's thesis, "The Architecture of a Code Generator for Software Product Lines." He enrolled in doctoral studies the same year.

In 2018, he became a Teaching Assistant at the Faculty of Technical Sciences, combining teaching and research. He also began working part-time at Schneider Electric in Novi Sad, where he collaborated with his mentor and close colleagues on research problems in Model-Driven Software Engineering (MDSE), which eventually led to the topic of this thesis. In parallel, he contributed to the research of the DAIS group at the faculty on introducing MDSE into collaborative manufacturing. To date, he has co-authored eleven research papers. Two of these were published in journals indexed in the Science Citation Index Expanded, and a third has been accepted for publication.

In 2022, he co-founded Code2, a company through which he has broadened his experience with industrial problems and the application of MDSE in different settings. His research interests include model-driven software engineering and code generation and, more broadly, the software development lifecycle.

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
Аутоматизација развоја API-базираних генератора рударењем идиома из изворног кода Automating the Development of API-Based Code Generators Using Code Idioms Mining
Назив институције/институција у оквиру којих се спроводи истраживање
а) Факултет техничких наука, Универзитет у Новом Саду б) в)
Назив програма у оквиру ког се реализује истраживање
Докторске академске студије, студијски програм Рачунарство и аутоматика
1. Опис података
1.1 Врста студије *Укратко описати тип студије у оквиру које се подаци прикупљају* **Емпиријско истраживање у софтверском инжењерству које комбинује аутоматизовани рачунарски експеримент, вишеструку студију случаја и анализу изворног кода.** 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 _Актуелна литература у области истраживања_

ж) мапа, навести врсту _____

з) остало: **Ради евалуације коришћени су постојећи, јавно доступни подаци: изворни код тринаест пројеката отвореног кода писаних у програмском језику C#, као и шеснаест студентских пројеката које су студенти сами јавно објавили на GitHub-у у оквиру предмета Веб програмирање.**

Додатно, за део евалуације коришћени су и резултати истраживања спроведеног у сарадњи са Schneider Electric развојним центром у Новом Саду: време развоја генератора кода и квалитативни подаци о обиму кода.

Истраживање не укључује испитанике у смислу прикупљања њихових података; квалитативну процену издвојених идиома извршили су чланови тима и екстерни програмери, без прикупљања њихових података.

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 _шест (6) мерених варијабли_____

б) број мерења (испитаника, процена, снимака и сл.) _13 пројеката отвореног кода, уз 16 студентских пројеката_

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

Ради евалуације коришћени су постојећи, јавно доступни подаци: изворни код тринаест пројеката отвореног кода писаних у програмском језику C# (два одабрана пројекта Владе Британске Колумбије и једанаест пројеката из скупа који су предложили Allamanis и сарадници), као и шеснаест студентских пројеката које су студенти сами јавно објавили на GitHub-у у оквиру предмета, а који нису настали за потребе овог истраживања. Из студентских пројеката издвојен је искључиво изворни код, без историје верзија и без података о ауторима, и тако обједињен у један репозиторијум, тако да скуп не садржи везу са студентским налозима нити друге личне податке. Сви пројекти преузети су са GitHub-а уз наведене SHA-1 идентификаторе комита, и обједињени у јавно доступан скуп података на платформи Mendeley Data.
Додатно, за део евалуације коришћени су и резултати истраживања спроведеног у сарадњи са Schneider Electric развојним центром у Новом Саду: време развоја генератора кода и квалитативни подаци о обиму кода.

Истраживање не укључује испитанике у смислу прикупљања личних података; квалитативну процену издвојених идиома извршили су чланови тима и екстерни програмери, без прикупљања њихових личних података.

2.1.1. У оквиру ког истраживачког нацрта су подаци прикупљени?

а) **експеримент: полу-аутоматизовани експеримент над фиксираним верзијама софтвера**

б) корелационо истраживање, навести тип _____

ц) анализа текста, навести тип _Анализа доступне литературе_

д) остало: **студије случајева, као и анализа изворног кода**_

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

_Нису коришћени стандардизовани мерни инструменти специфични за дисциплину. _

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

Ради контроле квалитета података, коришћени су репозиторијуми отвореног кода препоручени у литератури. Поред препоручених, додата су два ручно прегледана продукциона софтверска решења. Додатно, коришћени студентски пројекти нису настали ради дисертације, већ су резултат независне израде на предмету који се тиче веб програмирања.

Верзије изворног кода су фиксиране, а сви депоновани пројекти су приложени уз SHA-1 идентификаторе комита. Mendeley Data репозиторијум аутоматски генерише SHA-256 контролну суму.

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

Подаци су преузети из јавно доступних извора и ручно прегледани.

3. Третман података и пратећа документација

3.1. Третман и чување података

3.1.1. ***Подаци ће бити депоновани у Mendeley Data репозиторијум.***

3.1.2. URL адреса *https://data.mendeley.com/datasets/6jmbv4gjz8/1*

3.1.3. DOI 10.17632/6jmbv4gjz8.1

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

__Није примењиво. Подаци ће бити депоновани у Mendeley Data репозиторијум__

3.2 Метаподаци и документација података

3.2.1. Који стандард за метаподатке ће бити примењен? **Mendeley Data шема за метаподатке, која одговара schema.org и Dublin Core стандардима. Пратећа документација обухвата упутство са описом скупа, пореклом података, и упутством за репродукцију.**

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

Скуп је депонован на платформи Mendeley уз следеће метаподатке: наслов („C# projects for code idioms mining"), аутор и институција, датум објаве (1. мај 2025), верзија (1), опис садржаја и намене скупа, категорија (Data Mining), лиценца (CC BY 4.0), везе ка повезаним публикацијама и ка репозиторијуму софтвера RoseLibML, као и податке о датотеци (назив, величина и SHA-256 контролна сума коју платформа аутоматски генерише). Ревизије појединачних пројеката документоване су hash вредностима Git комита у дисертацији.

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? **_Током животног века**

Mendeley Data репозиторијума, у складу са његовом политиком задржавања.

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 **(свима)**

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

Да **Не**

Образложити

Не планира се уклањање података из отвореног приступа.

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.

Није примењиво; истраживање не укључује људе као испитанике у смислу узимања њихових података.

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.

_____

6. Улоге и одговорност

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

___ Ненад Тодоровић, nenadtod@live.com_____

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

___ Ненад Тодоровић, nenadtod@live.com_____

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

____ Ненад Тодоровић, nenadtod@live.com_____