

Author
Dipl.-Ing.
Hannes Thaller

Submission
Institute for
Software Systems
Engineering

First Supervisor
Dr. Alexander Egyed

Second Supervisor
Dr. Foutse Khomh

Assistant
Thesis Supervisor
Dr. Lukas Linsbauer

October, 2021

Probabilistic Software Modeling



Doctoral Thesis
to obtain the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

Abstract

Software Engineering and the art of building computer programs is a complex endeavor. A program runs through various stages of maturity from the start of its inception to the end of its lifecycle. This means that, more than once, a program inevitably be in an actual or apparent non-deterministic state that the developers never expected. This non-determinism may arise from technical limitations such as distributed systems or parallel computing. Apparent non-determinism may arise from, for example, the complexity of the programs reaching the cognitive limits of their engineers. Modern software engineering has many methods, processes, guidelines, and strategies to limit the potential quality problems that are involved with any form of non-determinism or undefined behavior. This includes testing during development, staging, and rollout strategies during deployment and observability patterns in the live environment. An important consensus among software engineering practitioners is that the later a bug, unknown behavior, and unnecessary complexity are removed the more expensive it gets to resolve. This thesis introduces Probabilistic Software Modeling (PSM), a novel modeling paradigm for software. PSM complements existing methods via a model that can simulate and evaluate programs on a semantic level. PSM transforms a given program into a probabilistic model by extracting its structure and runtime behavior. The probabilistic model acts as a copy of the original program and allows for simulation and probabilistic quantification of program states and execution traces. This enables one to build various comprehension, predictive, and generative applications on top. This thesis presents the theoretical foundation on PSM and how a program can be transformed into a probabilistic model. Then the theory is then evaluated in the context of a feasibility study evaluating the scalability and applicability of PSM. A study on code clones provides insights into the different types of code clones acting as a primer for semantic clone detection via PSM. Then we give an outlook on how fault localization via PSM can be accomplished. Finally, a study on the PSM program representation shows the usefulness of graph databases in the context of static code analysis. The presented studies show a potent framework for future program analysis and simulation. This includes state-of-art semantic clone detection performance and precise fault localization results.

Kurzfassung

Software Engineering und die Kunst Computerprogramme zu planen und zu implementieren ist ein komplexes Unterfangen. Ein Programm durchläuft, von der initialen Idee bis hin zum Ende des Lebenszyklusses verschiedenste Qualitätstadien. Während diesem Lebenszyklusses, ist das Programm mehrmals in einem echten oder scheinbaren nichtdeterministischen Zustand. Echter Nichtdeterminismus entsteht durch technische Gegebenheit, wie zum Beispiel verteilte Systeme oder Nebenläufigkeit. Scheinbarer Nichtdeterminismus entsteht durch die steigende Komplexität von Programmen, welche die kognitiven Grenzen der Ingenieure sprengen. Die moderne Softwareentwicklung hat viele Methoden, Prozesse, Richtlinien, und Strategien um die Auswirkungen von undefiniertem Verhalten einzugrenzen. Dies inkludiert Software Testen, Staging und inkrementelle Releases, sowie Überwachungskonzepte der aktiven Laufzeitumgebung. Ein wichtiger Konsensus unter Software Ingenieure ist, je später ein Fehler, ein unerwartetes Verhalten, oder unnötige Komplexität gelöst wird, desto teuer wird die Korrektur. Diese Dissertation führt Probabilistic Software Modeling (PSM) ein. PSM ist ein neues Modellierungs Paradigma welches bestehende Methoden mittels Simulation und Evaluierung von Programmen komplementiert. Durch die Extraktion der Programmstruktur und dem Laufzeitverhalten, transformiert PSM ein gegebenes Programm in ein probabilistisches Model. Dieses Model repräsentiert eine (probabilistische) Kopie des originalen Programmes und erlaubt die Simulation, sowie die probabilistische Quantifizierung von Programzuständen und Ausführungspfaden. Dies ermöglicht unterschiedlichste Verständnis, prädiktive, und generative Applikationen. Diese Arbeit beinhaltet das theoretische Fundament zu PSM und wie Programme in probabilistische Modelle transformiert werden können. Eine Machbarkeitsstudie evaluiert dann die Skalierbarkeit und Anwendbarkeit der theoretischen Konzepte von PSM. Eine Studie zur Detektion von Code Duplikaten liefert dann wichtige Einblicke zu den verschiedensten Typen von Klonen, und bereitet das Fundament für die automatische Detektion von semantischen Klonen mittels PSM. Danach gibt eine Studie einen Ausblick auf die Fehlerfindung mittels PSM. Zum Schluss, zeigt eine Studie die Nützlichkeit von Graphdatenbanken für die statische Codeanalyse auf. Die präsentierten Studien zeigen das PSM ein potentes Framework für die Program Analyse und Simulation werden kann. Dies inkludiert state-of-art Detektoren zum Finden von semantischen Codeduplizierungen und präzise Resultate in der automatischen Fehler lokalisierung.

Statutory Declaration

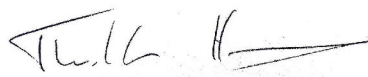
I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Parts of this thesis have been published as international conference articles (see [1], [2], [3]).

Wels, October 7, 2021

Place, Date

A handwritten signature in black ink, appearing to be 'Th. L. H.', written over a horizontal line.

Signature

Acknowledgments

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH. This research was funded in part, by the Austrian Science Fund (FWF) [P25513].

Contents

1	Introduction	6
1.1	Research Questions	7
1.2	Motivating Applications	8
1.2.1	Predictive Applications	8
1.2.2	Generative Applications	9
1.3	Outline	9
2	Running Example	11
3	Background	16
3.1	Software Engineering	16
3.1.1	Code	16
3.1.2	Runtime	17
3.2	Probabilistic Modeling	17
3.2.1	Bayesian Modeling	19
3.2.2	Factor Graph	27
3.2.3	Cluster Graph	30
3.2.4	Flow-based Models	31
4	Methodology	36
4.1	Code	37
4.1.1	Model Universe	39
4.2	Runtime	41
4.2.1	Instrumentation	42
4.2.2	Program Triggers	43
4.2.3	Behavior Graph	44
4.3	Modeling	46
4.3.1	Inference Graph — Global Structure Creation	46
4.3.2	Factor Graph to Cluster Graph	50
4.3.3	Behavior Datasets	51
4.3.4	Inference Graph — Local Structure Optimization	54
4.4	Inference	55
4.4.1	Properties of the Inference Graph	56
4.4.2	Local Marginal Inference	57
4.4.3	Local Conditional Inference	58
4.4.4	Global Marginal Inference	59
4.4.5	Global Conditional Inference	60

4.5	Related Work	61
5	Case Study: Modeling Feasibility	64
5.1	Study	64
5.1.1	Setup	65
5.1.2	Subject Systems	66
5.1.3	Controlled Variables	67
5.1.4	Response Variables	67
5.1.5	Experiment Results	67
5.2	Discussion	72
5.2.1	Code	72
5.2.2	Runtime	73
5.2.3	Modeling	73
5.2.4	Inference	73
5.3	Limitations	74
5.4	Threats to Validity	74
5.5	Conclusion and Future Work	74
6	Case Study: Clones in PLC Programs	75
6.1	Introduction	75
6.2	Background and Related Work	76
6.2.1	Clone Taxonomy	78
6.2.2	Clone Tools	78
6.2.3	Related Work	79
6.3	Industry Context	80
6.4	Code Clone Analysis	80
6.5	Code Clone Study	81
6.5.1	Study Design	82
6.5.2	Procedures	84
6.5.3	Evaluation	84
6.5.4	Interpretation	87
6.6	Threats to Validity	87
6.7	Conclusions	88
7	Case Study: Semantic Clone Detection	89
7.1	Introduction	89
7.2	Background	91
7.2.1	Clone Detection	91
7.2.2	Programs & Code Elements	92
7.2.3	Probabilistic Software Modeling	93
7.3	Semantic Clones	94
7.4	Approach	94
7.4.1	Modeling	96
7.4.2	Search Space	96

7.4.3	Static Similarity	97
7.4.4	Dynamic Similarity	98
7.4.5	Model Similarity	98
7.5	Study	100
7.5.1	Setup	100
7.5.2	Dataset	100
7.5.3	Controlled Variables	102
7.5.4	Response Variables	102
7.5.5	Comparison Approaches	103
7.5.6	Experiment Results	103
7.6	Discussion	106
7.6.1	Research Question 1 — Detection Performance	106
7.6.2	Research Question 2 — Skip Evaluation Scalability	107
7.6.3	Research Question 3 — Skip Evaluation Effects	107
7.7	Limitations	107
7.8	Threats to Validity	108
7.9	Related Work	109
7.10	Conclusions and Future Work	110
8	Case Study: Fault Localization	111
8.1	Introduction	111
8.2	Running Example	112
8.3	Background	112
8.4	Approach	113
8.5	Preliminary Study	113
8.5.1	Regression Fault	114
8.5.2	Integration Fault	115
8.6	Discussion	116
8.7	Related Work	116
8.8	Conclusion and Future Work	117
9	Case Study: Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases	118
9.1	Introduction	119
9.2	Background and Related Work	120
9.3	Research Design	121
9.4	Description of Cases	122
9.4.1	Case 1: AutoDoc for Lightweight Dependency Analysis	122
9.4.2	Case 2: SCoRe for PLC Programs	124
9.4.3	Case 3: eKNOWS Code Model Service	127
9.4.4	Case 4: Sherlock for Regression Test Case Selection	130
9.4.5	Case 5: Gradient for Probabilistic Software Modeling	133
9.5	Discussion	137
9.5.1	Advantages	137

9.5.2 Disadvantages	138
9.6 Summary and Conclusions	139
10 Discussion	142
10.1 Research Question 1	142
10.2 Research Question 2	144
10.3 Research Question 3	144
11 Conclusions	146
11.1 Thesis Contributions	146
11.2 Future Work	147

1 Introduction

Software Engineering and the art of building computer programs is a long and complex endeavor. A program runs through various stages of maturity from the start of eliciting the requirements until the end of its lifecycle. Looking at this broad spectrum of maturity of a non-trivial system, it becomes clear that medium to large scale software does not always behave as deterministic as developers wish it to be. There are two different types of non-determinism: actual and apparent non-determinism [4]. *Actual non-determinism* is the theoretical concept of undefined state transitions in a program. Actual non-determinism can arise from pure technical limitations, e.g., distributed systems, concurrency, or input-output communication. *Apparent non-determinism* is the practical concept of unknown state transition in a program caused by underspecification, cognitive limits, limits in observability, or system complexity. Both forms of non-determinism become of increasing importance in a world full of, e.g., distributed micro-services that interact with AI systems spanning the globe.

State-of-art systems use a range of development procedures (e.g., agile development), testing approaches (e.g., unit, integration, monkey testing), staging and deployment strategies, and monitoring and observability patterns. All of these methods are associated with increasing the initial software quality, i.e., the reduction of the impact radius of faults, and the observability of faults in a running system. Important in this context is that the earlier the system's actual and apparent non-determinism is tackled, the more robust and cost-effective can the system be developed and operated.

This thesis focuses on Probabilistic Software Modeling (PSM) a modeling paradigm for programs that complements all the aforementioned development procedures. PSM transforms a given program, i.e., a system or subsystem, into a probabilistic model that allows for simulation and the probabilistic quantification of program states and execution traces. The probabilistic model is a reflection of the original program including essential program elements like types, functions, and fields. PSM acts similar to a compiler that compiles source code into machine code, in that it takes a program and transforms it into its probabilistic equivalent. The resulting probabilistic model can visualize or simulate the runtime behavior of functions or variables and evaluate the likelihood of these.

Companies and developers may further improve the software quality and reduce the fault impact by leveraging the generative and evaluative capabilities of a probabilistic model. The model may simulate complex system to system integrations before fully implementing the integration. It can aid during fault localization processes that are hard to debug. It can help in comprehending legacy system and their behavior during their maintenance or redevelopment. It can bridge the gap between AI components and traditional software components by evaluating their compatibility. A final example, the model can generate tests for unlikely states in a subsystem. While many listed examples

are still future research, some use-cases can already be solved by the foundation laid in this thesis.

1.1 Research Questions

The overarching research questions in this thesis are as follows.

RQ1 Can programs be transformed into an equivalent probabilistic model?

RQ1.1 Can the probabilistic model preserve the structural properties of the modeled program?

RQ1.2 Can the probabilistic model preserve the behavioral properties of the modeled program?

RQ2 Can the transformation between programs and its probabilistic equivalent scale to a given computation horizon?

RQ3 Can the resulting probabilistic model aid in software engineering tasks?

RQ1 provides answers on whether a program can be transformed into a probabilistic model. RQ1.1 and RQ1.2 provides answers whether this transformation preserves the main properties of the program. Each of the aforementioned questions are linked to the viability of PSM. RQ2 provides answers whether the transformation can be scaled up to a specific computational horizon. The computational horizon depends on the target use case the model is build for. For example, generating integration tests between systems has a different timelines than aiding developers in their fault localization tasks. While the test generation might have a horizon of several days without losing usefulness, waiting for a model for couple of days to debug a specific fault may not be tolerable. However, a basic computational horizon is still given, in that the generation of the model may complete within hours at most days. Finally, RQ3 answers whether the transformed model is useful in the context of software engineering. While these research questions may take years after publishing this thesis to be fully answered, a clear lookout within the context of this academic research is viable.

Any solution provided to these questions is constraint by the following requirements:

1. Structural closeness between the internal structure of the program and the model;
2. Behavioral closeness between the runtime dynamics of the program and the behavior of model;
3. Generative capabilities of the resulting model;
4. Evaluation capabilities of the resulting model.

Structural closeness states that the program and the resulting model share a common structure. This requirement guarantees that any solution does not diverge too far from the original structure of the program. Extreme divergences in structure between the program

and model increase the cognitive load needed to work with the model. *Behavioral closeness* states that the program and its associated model share a common behavior. Similar to structural closeness, behavioral closeness avoids additional overload in understanding the model. *Generative capabilities* states that the model may (partially) generate the original behavior. The goal of the thesis is to construct a methodology to generate an actionable model that the developers can interact with. *Evaluation capabilities* states that the model is able to quantify aspects of the program’s behavior. This requirement guarantees the analysis capabilities of any resulting PSM solution. The requirements limit the space of the potential solution to a subset we deem useful in the problem context mentioned in the introduction. However, these requirements are limitations we impose on ourselves and do not necessarily apply to research on PSM outside the context of this thesis.

1.2 Motivating Applications

PSM is a generic framework that enables a wide range of predictive and generative applications. This section lists a selection of possible applications. Some applications are presented later in the work in greater detail while others are ideas based on the results of the evaluated applications.

1.2.1 Predictive Applications

Predictive applications seek to quantify, visualize, infer and predict the behavior and quality of a system.

Visualization and Comprehension [5, 6, 7] helps to understand programs and their behavior. This includes the visualization of code elements and non-functional attributes such as memory consumption. The PM is the source of the visualization showing the global and conditional behavior across code elements. For example, Figure 4.2 visualizes the **height**-property marginally and conditionally given that only females use the nutrition advisor ($P(\text{Height} \mid \text{Gender} = \text{Female})$).

Integrity & Compatibility Evaluation quantify the consistency between and within software components by measuring the behavioral divergence between the components. The divergence represents the likelihood that modules can successfully interact with each other. Integrity measures the divergence between two software components that depend on each other. Compatibility measures the divergence between the same component over revisions. Both measures are targeted for behavioral difference extending existing methods that are based on static descriptive measures such as the cyclomatic complexity [8]. For example, a change of units from kg to pounds for **Person.weight** yields the same shape of distribution but on a different scale. The compatibility evaluation would result in a low score between the two versions because of the shift in behavior (e.g., between revisions of **Person.weight**). The integrity evaluation would also result in a low score for methods that read the newly altered weight values (e.g., **advice-method**).

Semantic Clone-Detection [3, 9, 10] detects methods that are syntactically but exhibit similar runtime behavior. For example, the iterative and recursive implementation of the factorial computation. Both yield the same input and output behavior while the

underlying source code is different. These types of clones are beyond traditional clone detectors that focus on static properties (e.g., sequence of tokens or the abstract syntax tree). PSM can detect semantic clones by comparing the likelihood of between methods and their model representations. Hence, the likelihood acts as a distance metric while significance tests such as the Generalized Likelihood Ratio Test [11] (GLRT) allow for a final decision.

Fault Localization [12, 13] finds the most likely location of a fault in a program. Faults may be detected within or between revisions of a program. PSM uses the same principles as semantic clone detection to find these faults. This process can be guided via existing tests to decide on the type of change, i.e., whether it was an error or an expected change. PSM can also generate the error path describing all methods that diverged and the effect it has on neighboring systems.

Anomaly Detection [14, 15, 16, 17] applications measure the divergence between a persisted PSM model and observations collected from a live system. These applications can be deployed into a live system, in which components are monitored and checked against their models. Again, the GLRT can check for runtime observations that diverge too far from the model and trigger additional actions. These actions can be a notification to human operators, generate reports, or apply temporary defense actions such as subsystem suspensions.

1.2.2 Generative Applications

Generative applications leverage observations drawn from the models, e.g., method inputs or field values.

Test-Case Generation [18, 19] applications draw observations from the model for methods and fields to generate test data. The test data can be tailored regarding a specific likelihood resulting, e.g., test suites such as *typical*, *rare*, *edge*. A *typical* test-suite would be triggered on the average runtime inputs, while rare and edge cases would be triggered with observations with relatively low or nearly zero likelihoods. In addition, it is possible to provide specific conditions to test case generation, e.g., that only females use the nutrition advisor services. This allows the generated test-suites to have large *semantic* coverage instead of code coverage.

Simulations sample execution traces from the PM in a structured fashion to reproduce the running system. This probabilistically executes the original program without actually running it. Simulations can bridge boundaries between hardware and software interfaces, reducing the number of hardware dependencies during development.

1.3 Outline

The thesis is organized as follows. Chapter 1 introduces the topic of PSM, provides motivating application examples, and frames the research goals of the thesis. Section 1.2 potential applications that can be solved via PSM as additional motivation. Chapter 2 introduces the running example used throughout the thesis. Chapter 3 provides the needed background to understand the concepts in the thesis. Section 3.1 provides the

background on the software engineering aspects, while Section 3.2 provides the background on probabilistic modeling. Sections 3.2.1 to 3.2.3 introduces various probabilistic models by example and maybe skipped by readers familiar with the topic. Chapter 4 introduces the main contribution of this thesis and the methods involved with PSM. The methods are split into *code* (Section 4.1), *runtime* (Section 4.2), *modeling* (Section 4.3), and *inference* (Section 4.4). Each explaining the theoretical aspects of PSM. Chapter 5 provides a feasibility study on PSM. Chapter 6 contains a study on code clones which motivated the study on semantic clone detection in Chapter 7. Chapter 7 represents the first realized PSM use-case with important concepts on comparing programs and their associated models on a behavioral level. Chapter 8 builds on top of the concepts gained from Chapter 7 to outline the use-case of fault localization. Chapter 9 contains a case study on the benefits and strength of graph databases that were used for the structural representations of PSM. Chapter 10 provides the conclusions on PSM and the research questions provided in Section 1.1. Chapter 11 concludes the thesis and outlines potential future research directions.

2 Running Example

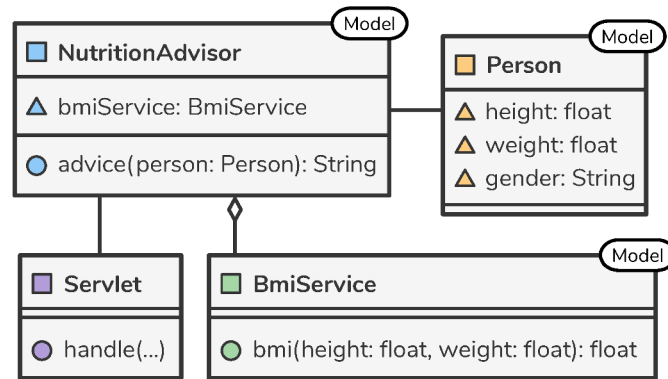


Figure 2.1: The static structure of the Nutrition Advisor, consisting of three core classes and a context class (e.g., a web interface) calling the program.

Consider as our running example the *Nutrition Advisor* that takes a person's anthropometric measurements (i.e., height and weight) and returns a piece of textual advice based on the *Body Mass Index (BMI)*. Figure 2.1 shows the class diagram of the Nutrition Advisor, consisting of three core classes and the **Servlet** class. **Person** stores data that is processed via the business logic represented by the classes **NutritionAdvisor** and **BmiService**. The servlet class acts as the entry point of the requests. Classes that should be modeled are annotated with *Model* (e.g., **Person**). Figure 2.2 depicts a sequence diagram of one program trace with concrete values. The **Servlet** receives properties (e.g., height, weight, or gender) with which it instantiates a **Person** object (not illustrated). **NutritionAdvisor.advice()** takes this **Person** object, extracts the **height** (168.59) and **weight** (69.54) and computes the person's BMI (24.466) via **BmiService.bmi()**. The result is a piece of textual advice based on the BMI ("You are healthy, try a ..."). Listings 2.1 to 2.3 show the implementation of the three main methods. The **handle()**-method (Listing 2.1) can be replaced with any other front-end implementation, e.g., a graphical user-interface callback or a command-line command. The **advice()**-method, and **bmi()**-method (Listings 2.2 and 2.3) are simple side-effect-free functions that compute their respective quantities.

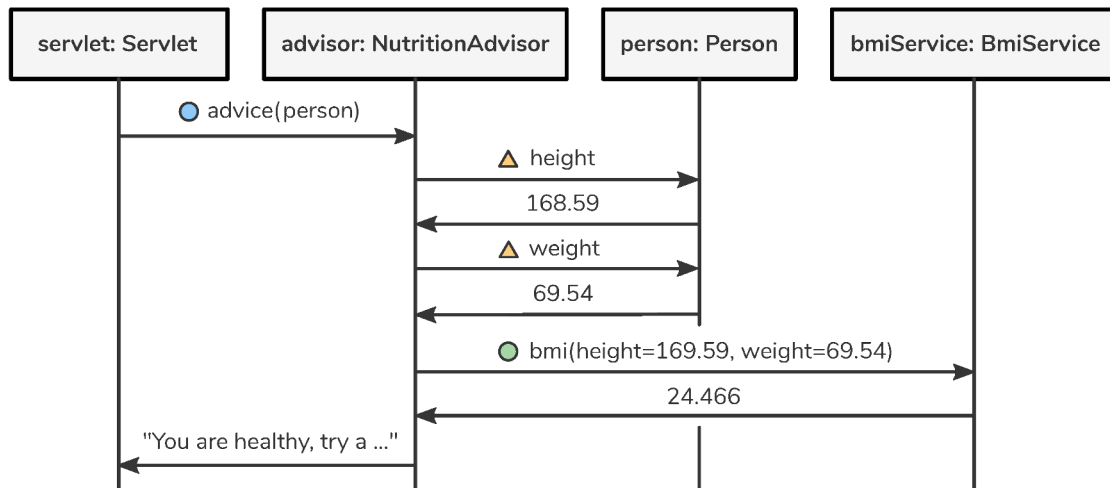


Figure 2.2: The dynamic behavior of the Nutrition Advisor, visualized by one execution trace. The `NutritionAdvisor` handles `advice` requests in which `Person` objects are received and a piece of textual advice is returned.

```

1 class Servlet{
2     NutritionAdvisor advisor = new NutritionAdvisor()
3
4     void handle(String gender, int age, float height, float weight){
5         Person person = new Person(gender, age, height, weight)
6         String advice = advisor.advice(person)
7         print(advice)
8     }
9 }
  
```

Listing 2.1: *handle* of the Nutrition Advisor.

Table 2.1: The runtime observations of the Nutrition Advisor. Each row represents the observations of one request.

	Height	Weight	Bmi	Advice	Gender	Advice encoded	Gender encoded
1	174.30	90.00	29.62	Consider skipping the meal.	Male	2	0
2	157.80	97.60	39.19	Please do not eat me!	Female	5	1
3	159.10	69.80	27.57	Consider skipping the meal.	Female	2	1
4	172.60	70.10	23.53	Your are good, eat if you want.	Male	1	0
5	172.80	63.60	21.29	Your are good, eat if you want.	Male	1	0

```

1 class NutritionAdvisor{
2     BmiService bmiService = new BmiService()
3
4     String advise(Person person){
5         float bmi = bmiService.bmi(person.height, person.weight)
6
7         String advice;
8         if (bmi <= 18.5) {
9             advice = "Do not stop eating!";
10        } else if (18.5 < bmi && bmi <= 25) {
11            advice = "Your are good, eat if you want.";
12        } else if (25 < bmi && bmi <= 30) {
13            advice = "Consider skipping the meal.";
14        } else if (30 < bmi && bmi <= 35) {
15            advice = "Hungry again?";
16        } else if (35 < bmi && bmi <= 39) {
17            advice = "Do not eat!";
18        } else {
19            advice = "Please do not eat me!";
20        }
21
22        return advice;
23    }
24 }

```

Listing 2.2: *advise* of the Nutrition Advisor.

```

1 public class BmiService {
2     float bmi(float height, float weight) {
3         float heightInMeters = height / 100;
4
5         return weight / (heightInMeters * heightInMeters);
6     }
7 }

```

Listing 2.3: *bmi* of the Nutrition Advisor

The observed data in the Nutrition Advisor is given by Table 2.1. The columns map to the key variables of the Nutrition Advisors that represent the inputs and outputs of

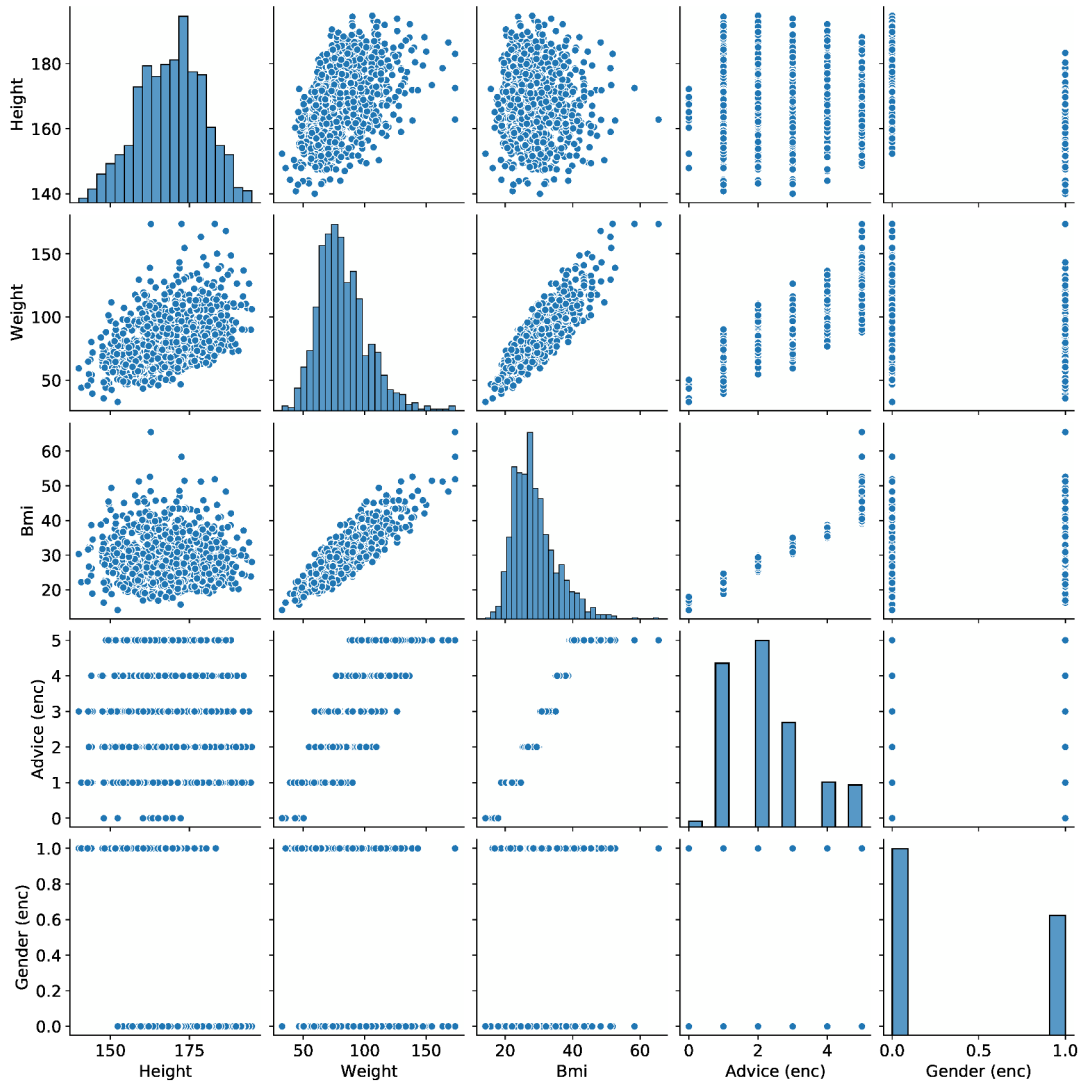


Figure 2.3: Pair-wise plot of the observed variables.

the three main methods.

The rows show 5 of the total 1000 observations each representing a request to the nutrition advisor. Each row was created by intercepting calls to the methods and tracking their values. All variables are central to the Nutrition Advisor’s behavior except for *Gender*. *Advice encoded* and *Gender encoded* represent the numerical encoding of their textual original.

Figure 2.3 shows the pair-wise plots of all variables. *Height*, *Weight*, and *Bmi* resemble a Normal distribution with strong linear relationships among them (except for *Height* and *Bmi*). *Gender* shows a slight imbalance towards males. The *Advice* has a monotonic relationship with *Bmi*.

The requests made to the Nutrition Advisor are based on the NHANES [20] data.

3 Background

Probabilistic Software Modeling (PSM) lives in the intersection of software engineering and probabilistic modeling. Understanding the basics from each domain is essential in understanding the concepts of PSM. Basic methods from *static* and *dynamic code analysis* [21] and general knowledge of programs and programming are needed for the software engineering part. Bayesian methods, statistics, and modern density estimation based on normalization flows are required for the probabilistic modeling part.

3.1 Software Engineering

Software Engineering is a broad term capturing everything related to the design, implementation, testing, and analysis of software. In the context of PSM, we will focus on the analysis of programs. More specifically, the properties of the source code, and its runtime behavior.

3.1.1 Code

PSM is interested in the structure of a program in form of a graph. This structure is extracted via static code analysis. *Static Code Analysis* describes any analysis of the source code of a program without executing it. In most static analysis applications, first, an abstract syntax tree is extracted, and then an abstract semantics graph is generated from it.

An Abstract Syntax Tree (AST) describes the syntactic structure of the textual representation of a program in form of a tree. The nodes of an AST are programming constructs. For example, *while* represents a loop construct. The edges describe a *consists-of* relation of the textual fragment. For example, *statement sequence* has a *while* as child, which in case has an *if-else branch* as child. ASTs are a low-level representation of the source code. Hence, they do not contain any typing information of variables, or values that are returned by methods. This information is constructed from the AST in a separate transformation step.

An Abstract Semantics Graph (ASG) is a concise but more complex representation of a program that operates on a higher level than an AST. An ASG is constructed from AST by adding semantic nodes and dependencies while removing purely syntactic nodes. The main difference is that an ASG is a graph that represents the semantic representation of a program. For example, call dependencies between methods or typing information of variables. This semantic view of the program allows for code refactorings and code optimizations. PSM has its own ASG with which it represents the structure of programs and that we will introduce in later chapters.

3.1.2 Runtime

PSM needs information on how the elements of a program interact at runtime. The resulting model should have the same behavior as the program in order to enable simulations and inference. This information is extracted via dynamic code analysis. Dynamic code analysis [21] is the process of observing a running program.

The program is executed by a *trigger* (parameters and environment) which is the context of the monitoring session. A running program spawns *event streams* which are sequences of *monitoring events* (e.g., Figure 2.2). These events contain information on which fields have been changed or methods were invoked. There are two types of runtime monitoring: tracing and sampling. *Tracing* tracks every possible event at runtime, e.g., every access to a predefined field. This results in a full runtime event stream tracking the full unaltered behavior of a program. However, tracing produces large amounts of data and may lead to performance issues. In contrast, *Sampling* records events according to a specific rate. The resulting event stream is not complete and only depicts parts of the program's behavior. However, the performance impact can be controlled.

Runtime monitoring is realized by changing the bytecode (e.g., the Java bytecode) of the underlying program. This may happen during the program compilation where a specialized compiler inserts monitoring instructions into the code (e.g., AspectJ compiler [22]). A specialized compiler can be used to add runtime monitoring instructions at compile time. Another approach is to defer the modification to the actual program runtime in a lazy fashion. In this case, specialized class loaders modify the program as the code is linked. Examples of tools or libraries with which runtime monitoring can be realized are AspectJ, CGLib, or ASM. *Aspectj* is a full tool-suite including a separate compiler to *weave* in monitoring aspects (small portions of programs). *CGLib* and *ASM* are bytecode manipulation libraries with which runtime monitoring can be implemented manually.

3.2 Probabilistic Modeling

A *probabilistic model* uses the theory of probability to model a complex system (e.g., Nutrition Advisor). The term *probability* can be associated with the degree of belief we have that a specific event happens. All known events span a *space* of outcomes which is denoted by Ω . For example, the event space of a `Person.gender` is $\Omega = \{male, female\}$ and of `Person.height` it might be $\Omega = \{x \mid x \sim \mathcal{N}(178, 2)\}$. " \sim " is the sample operator (different from sampling in Section 3.1.2) where the left-hand side is drawn from a distribution that is described on the right-hand side. A *probability distribution* P is a mapping from events in the system to real values. For example, $P(Person.gender) = \{female \mapsto 0.5, male \mapsto 0.5\}$ describes probability mappings of all events associated with *gender*. These values are between 0 and 1 and all values sum up to 1. Attributes over events are called *Random Variables (RVs)*, e.g., gender is an attribute of a person. This notion has parallels to fields in software engineering which are also often called attributes in that they describe a specific aspect of an object. Therefore, a *random variable* $X_i \in \mathcal{X}$ (e.g., *Weight*) captures an aspect of the system's event space (or outcomes). They allow

for simpler notations of outcomes by, e.g., $P(\text{Weight} > 80)$ captures all events where the random variable (or attribute) of a person is greater than 80. The value range of random variables is given by $Val(X_i)$ (e.g., $Val(\text{Gender}) = \{\text{female}, \text{male}\}$). The *marginal distribution* $P(X_i)$ describes the probability distribution over the events of a random variable X_i . For example, $P(\text{Gender} = \text{female}) = 0.5$; $P(\text{Gender} = \text{male}) = 0.5$ describes the marginal distribution of *gender*. The *joint distribution* $P(X_1, \dots, X_n)$ represents the probability distribution of multiple random variables in conjunction, e.g., $P(\text{Gender}, \text{Color})$. For example, $P(\text{Gender} = \text{male}, \text{Color} = \text{Green}) = 0.35$ describes the event that a male likes the color green. A *conditional distribution* $P(X | Y)$ describes the probability distribution of X given some additional information of the random variable Y is known. For example, $P(\text{Gender} = \text{male} | \text{Color} = \text{Green})$ describes the probability that *gender* is male while observing that the favorite color is *green*. The additional information that the color is green changes the degree of belief one has that the person is a male or a female. PSM captures the space of runtime events via probabilistic quantities described above to reason about the likelihood of specific events or to generate events according to a likelihood. More background information is given, e.g., by Koller and Friedman [23], Murphy [24], or Bishop [25].

The following sections present the background of the theoretical foundations of PSM by exploring classical Bayesian analysis (??), Factor Graphs, Cluster Graphs, and Normalizing Flows. Bayesian analysis provides a holistic approach to model data as a generative process. A factor graph also models a generative process. In contrast to a Bayesian network used in Bayesian analysis, it is an undirected graph. A cluster graph, similar to a factor graph, is omnidirectional but captures clusters of variables. Normalizing flows, in contrast, are multivariate density estimators that PSM uses to model clusters in the cluster graphs.

Probabilistic analysis can be organized in the *Box* analysis loop [26]. The process of the Box's loop can be summarized as follows

1. Gather data from real-world phenomena
2. Build a probabilistic model of the phenomena
3. Infer hidden quantities of the model via the gathered data
4. Criticize the model given the gathered data
5. Repeat Step 2 – 4 until the required quality is reached

For PSM, Step 1 is captured via runtime monitoring. Step 2, are transformations between the programs AST to an ASG and finally to a factor and cluster graph. Step 3 is represented by optimization of the NVPs that represent the clusters. Step 4 is measuring the convergence of the optimization process but also executing inference checks. Step 5 is not performed in the context of PSM. The underlying probabilistic structure is known from the program's graph and any change to its structure would break the PSM requirements.

3.2.1 Bayesian Modeling

A Bayesian Network [23] is a graph representation of a joint probability distribution over a set of random variables with their dependencies. More precise, it is a Directed Acyclic Graph (DAG) where nodes represent random variables. The edges represent dependencies between the variables. The acyclic and directed nature of the graph allows for efficient operations over the random variables, e.g., sampling the marginal joint distribution that the graph represents. Furthermore, the factorization into the graph structure and its variables (i.e., nodes) reduce the amount of data that needs to be gathered and stored significantly.

More formally, a Bayesian Network is a directed acyclical graph defined over

$$BN = (\mathbf{V}, \mathbf{E}).$$

The nodes \mathbf{V} are random variables $V \in \mathcal{X}$. Edges \mathbf{E} represent dependence between random variables that allow easier evaluation of conditional independence between the variables. The network itself is the global representation of how variables depend on each other. In addition, there is a local representation of the variables themselves. For example, a random variable *height* may be represented via a Gaussian distribution with mean μ and variance σ^2 with $\mathcal{N}(\mu; \sigma^2)$. \mathcal{N} would be the local model within the Bayesian network. At this point, we will refer to Koller [23] for further reading. There are many concepts, proofs, and usage patterns (exact versus approximate inference) that exceed the limits of this chapter but are worth reading.

Given now the *Nutrition Advisor*, the goal is to define a Bayesian Network representing the joint distribution of all key variables in the program.

$$P(\textit{Height}, \textit{Weight}, \textit{Bmi}, \textit{Advice}, \textit{Gender}).$$

For this we will employ approximate inference for continuous random variables in Bayesian networks. To ease the computations, we will use PyMC3 [27], a probabilistic programming framework for Python [28] that provides practitioners with a toolbox of inference methods.

In the first iteration of the modeling procedure, we focus on the `bmi()`-method. The goal is to find a variable dependency structure that represents

$$P(\textit{Height}, \textit{Weight}, \textit{Bmi}).$$

The variables are *Height*, *Weight*, and *Bmi* which are the two input parameters and the return value. Listing 2.3 shows that the BMI is computed by normalizing the weight by the squared height. Table 2.1 shows example values while Figure 2.3 visualizes the distribution of the variables. The histograms of *Height*, *Weight*, and *Bmi* show multi-modal Gaussian-like distributions. This fits the intuition of the variables. Figure 3.1 shows the Bayesian model in the Kruschke [29] notation. Variables with names and filled distributions are observed and are present in the dataset (*height*, *weight*, and *bmi*). Hollow distributions (e.g., $\mu^{\textit{height}}$ or $\sigma^{\textit{weight}}$) represent latent variables, i.e., hidden variables that we want to estimate as they allow us to generate data for observed variables.

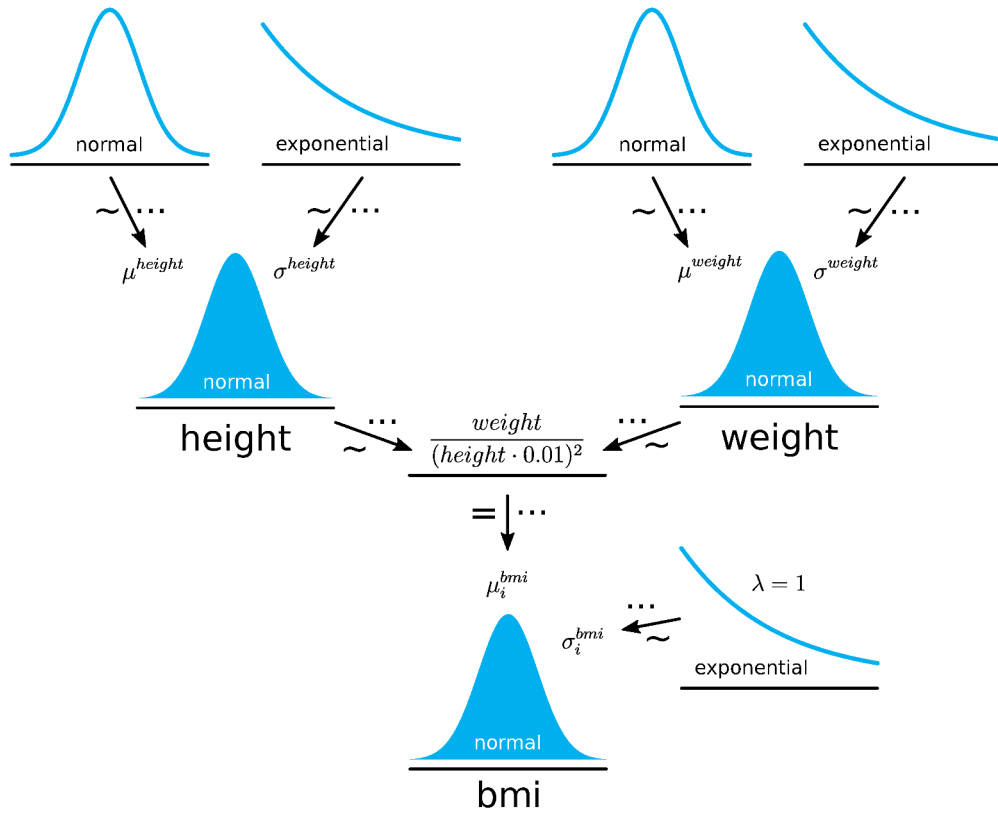


Figure 3.1: The Bayesian model for the `bmi()`-method. Height and weight are modeled independently for the sake of simplicity. A fully colored distribution represent observed variables, while the others represent the latent variables.

Both, *Height* and *Weight* are modeled via *normal* distributions. The mean μ is a latent, i.e., hidden variable modeled via a *normal* distribution. Similarly, the variance σ is modeled via an *exponential* distribution. The *bmi* is also modeled via a *normal* distribution with its mean being a deterministic function that represents the *bmi*. In this example, we decided to model *height* and *weight* fully factorized without its linear relationship that we can see in Figure 2.3. This reflects real-world situations in which not all variables and interactions are known by the modeling expert.

Modeling the bmi-method

The model in Figure 3.1 is already capable of generating data. The variables along with their distributions and dependence structure define a full generative process. Figure 3.2 shows the prior distribution, i.e., the data generated by the model without looking at the real data. *Observed* variables represent the true data similar to Figure 2.3. Each *prior predictive* represents one example distribution of the variables where the *mean prior predictive* is the mean across the distributions. This means, e.g., for the *Height* variable, that we first draw μ_0^{height} and σ_0^{height} from the latent variables. Then we sample from the observed variables $\mathcal{N}_0^{height}(\mu_0^{height}, \sigma_0^{height})$. This is called forward sampling representing the chain rule

$$P(X_1, \dots, X_n) = \prod_i^n P(X_i \mid \text{Pa}_X^G i), \quad (3.1)$$

where we sample first from the parents and then from the children to sample from the joint distribution of all variables [23].

Approximate inference [23] and the data from the observed variables can be used to optimize the parameters of the latent variables. Figure 3.3 shows the posterior predictive plots with optimized parameters. The fit converged closely to the observed data with some errors caused by the full factorization between *height* and *weight*. Figure 3.4 shows the trace plots for the inference process. The left plots show the distribution plots of the estimated hidden variables. The right plots are trace plots showing the random walk of finding the variables. Structures or lines in the trace plots hint that the inference process diverged. Figure 3.5 shows the estimated distribution with the 94% Highest Density Interval (HDI). The HDI represents the level at which most of the density is concentrated and should be interpreted as "water level" within a distribution. Furthermore, the plots contain the mean of the estimated hidden variables where the mean-variance in *height* and *weight* is 10.

In conclusion, we can state that modeling allows for some structural freedom that can be traded with model precision. Further, we estimate the hidden variables of the observed variables. The model is a generative process even without optimizing the parameters, however with suboptimal predictive power.

Modeling the advice-method

In this section, we will extend the model from above by the *advice* from the *advice* function. Similar to before, we start by modeling the underlying process. Figure 3.6

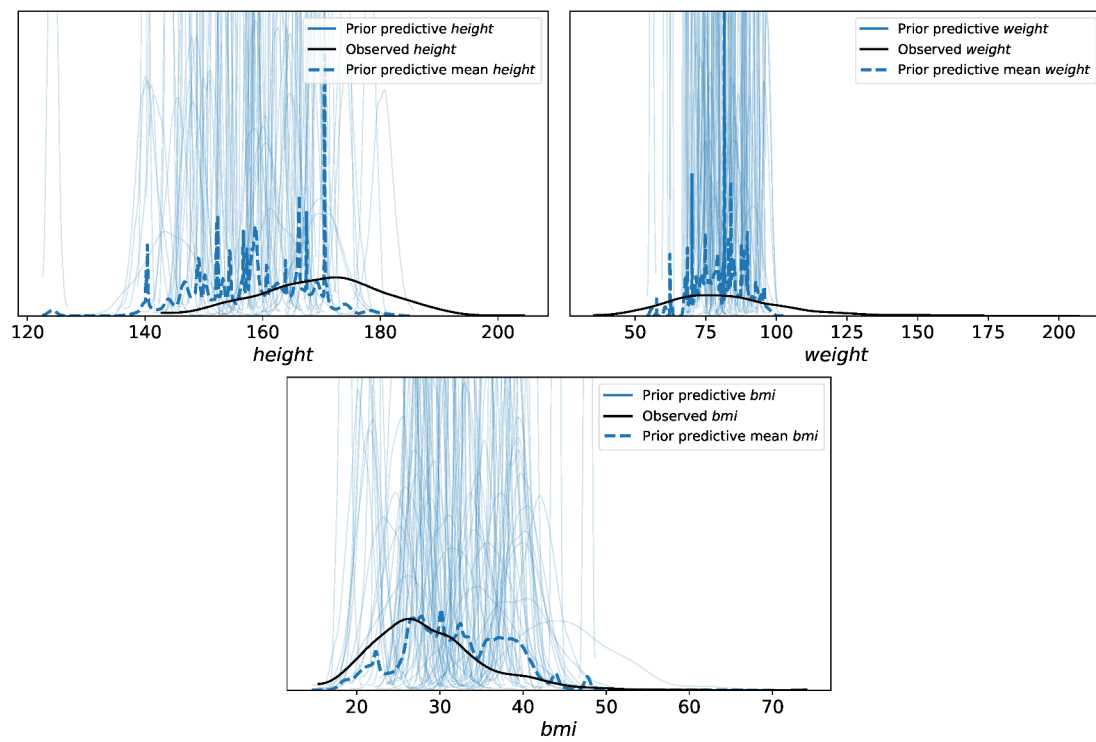


Figure 3.2: Prior predictive of the height, weight, and bmi variables.

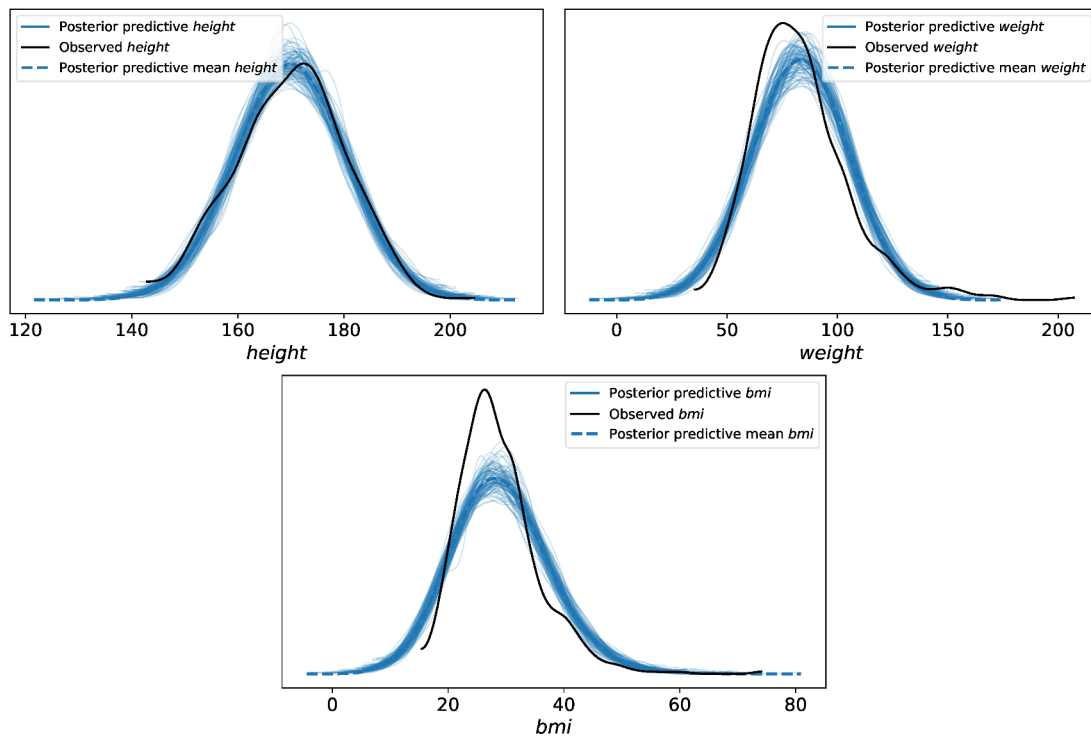


Figure 3.3: Posterior predictive of the height, weight, and bmi variables. Errors are caused by the factorized modeling of height and weight.

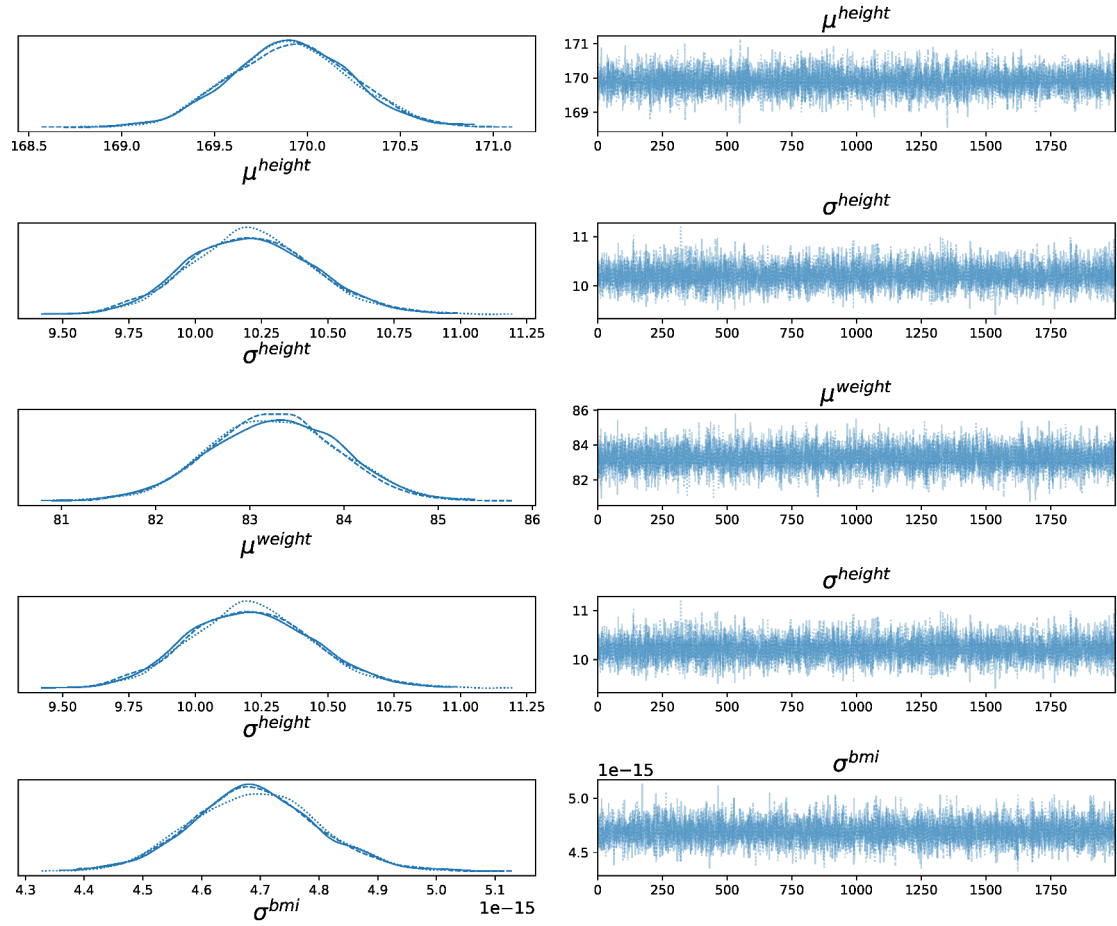


Figure 3.4: Trace plots of the latent variables. Left side shows the posterior of the latent variables. Right side shows the trace plot of the sampling process.

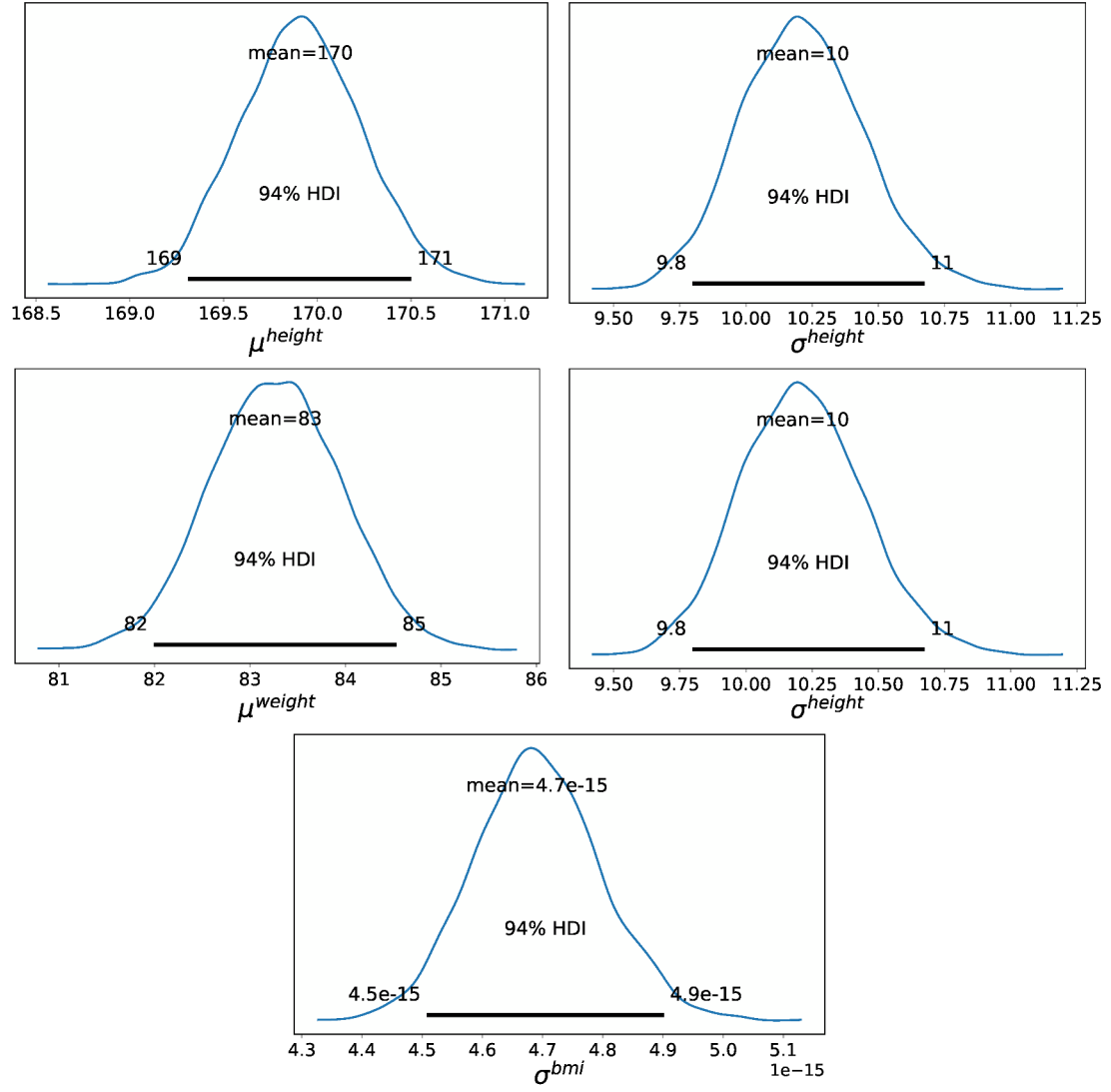
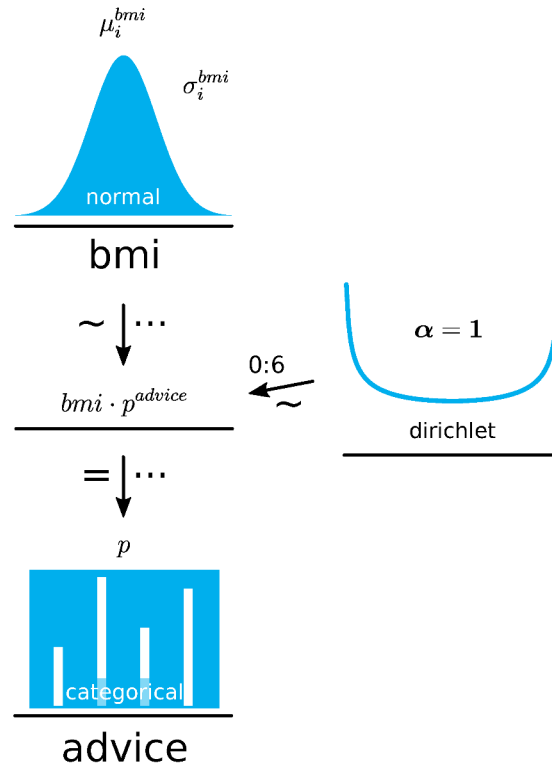


Figure 3.5: Posterior of the latent variables along with the 94 % highest density interval.

Figure 3.6: Bayesian model of the `advice()`-method.

shows the model for `advice`-method in which we estimate the category of the advice via k Dirichlet distributions. This model uses the domain knowledge that the *advice* is ordered with a monotonic increase of the *bmi*. Figure 3.7 shows prior and posterior predictive, estimated posterior, and the trace plot of the latent Dirichlet. The estimated posterior exhibits few errors and the trace plot shows a well converged model.

Modeling the `handle`-method

Finally, we model the `handle` method which contains the *gender* variable. The final model for the Nutrition Advisor is given in Figure 3.8. The gender variable is a Bernoulli variable selected one of two *height* or *weight* variables. Accordingly, we adapted the hidden variables to be bivariate too with different prior values based on our experience of the domain. Figure 3.9 again shows the prior and posterior predictive plots for the newly added *gender* variable. Furthermore, it contains the distribution of the latent θ variable and its trace plot.

Conclusions

The final model Figure 3.9 concludes the excursion in Bayesian modeling. The result is a Bayesian model of the Nutrition Advisor. While not perfect (height and weight

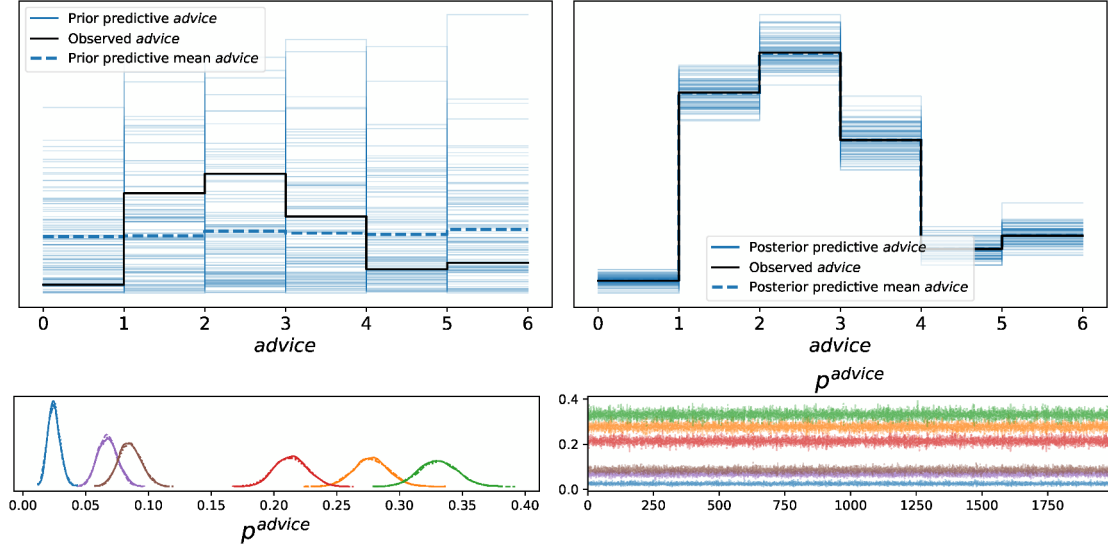


Figure 3.7: Posterior predictive and trace plots of the variables in the handle method.

relationship), it is still capable of simulating data from Table 2.1. Furthermore, one can also use the model to ask questions such as: what is the most probable gender given the 0'th advice? This is a powerful tool with which one could generate test suites, e.g., most unlikely but possible value configurations. One could visualize specific states in the model, e.g., what are height, and weight for the 5'th advice. Furthermore, one could use the model to evaluate whether a given request made to the `handle` method is an anomaly.

However, the process of modeling is not straightforward and needs a lot of experience. Approximate inference needs an iterative development approach of trial and error as convergence is a proposed model is not guaranteed. In this case, the model needs to be adapted, or the inference algorithm needs to be tweaked. This makes the whole approach intangible for large programs. PSM is an approach that automates modeling and parameter optimization. This is possible because the program itself exposes the needed structure of the model.

3.2.2 Factor Graph

A *factor graph* is a data structure of probabilistic quantities like $P(\text{Gender}, \text{Color})$. They are strongly related to Bayesian networks but allow the definition of more complex quantities because of their omnidirectionality. In fact, any Bayesian network can be transformed into a factor graph and vice versa. This must hold intuitively, as both model the joint distribution of variables. The full proof and process are given by Koller [23] which we will skip at this point.

More formally, a factor graph \mathcal{F} is an undirected bipartite graph consisting of types of

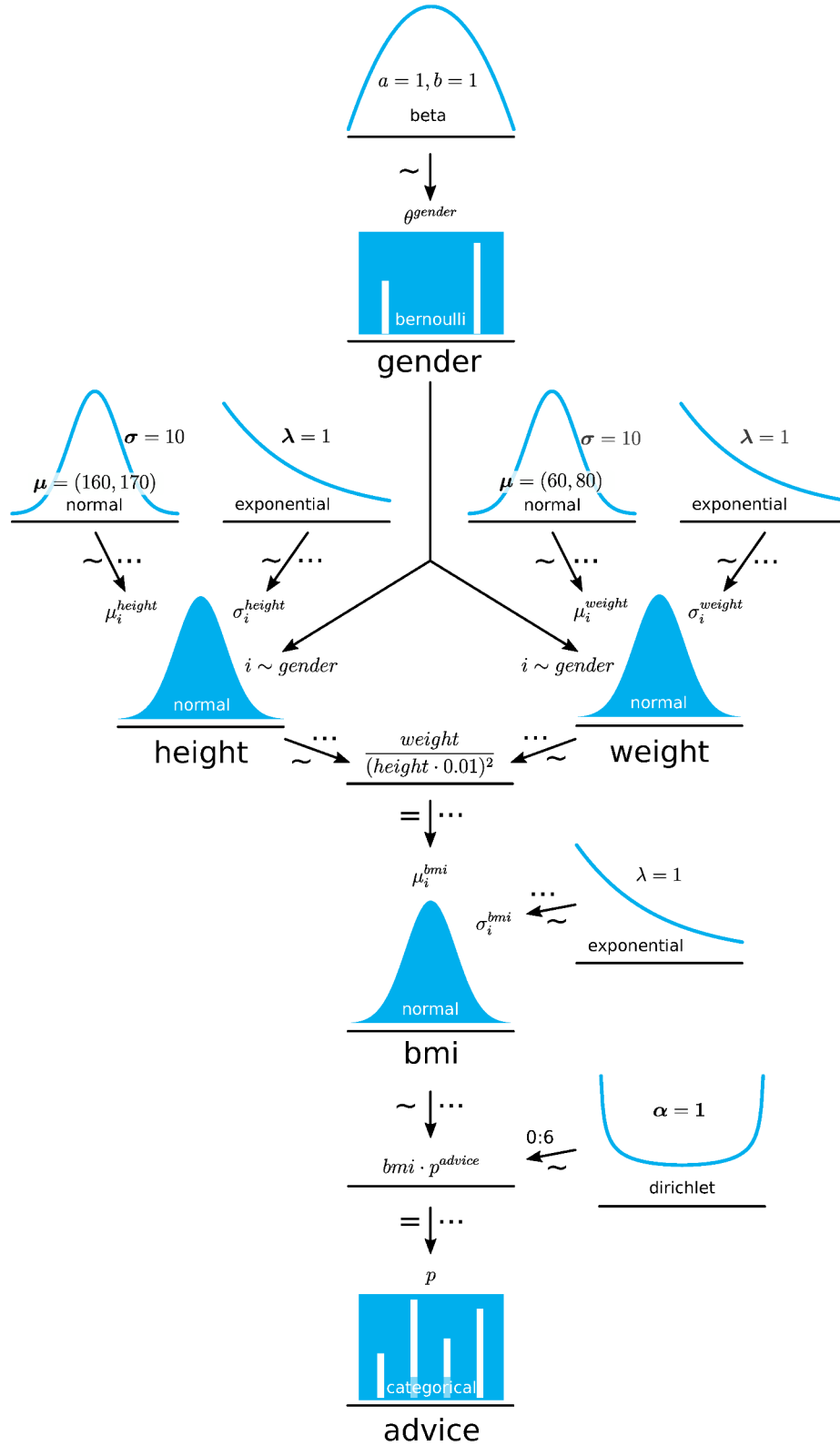


Figure 3.8: The entire Bayesian model of the Nutrition Advisor.

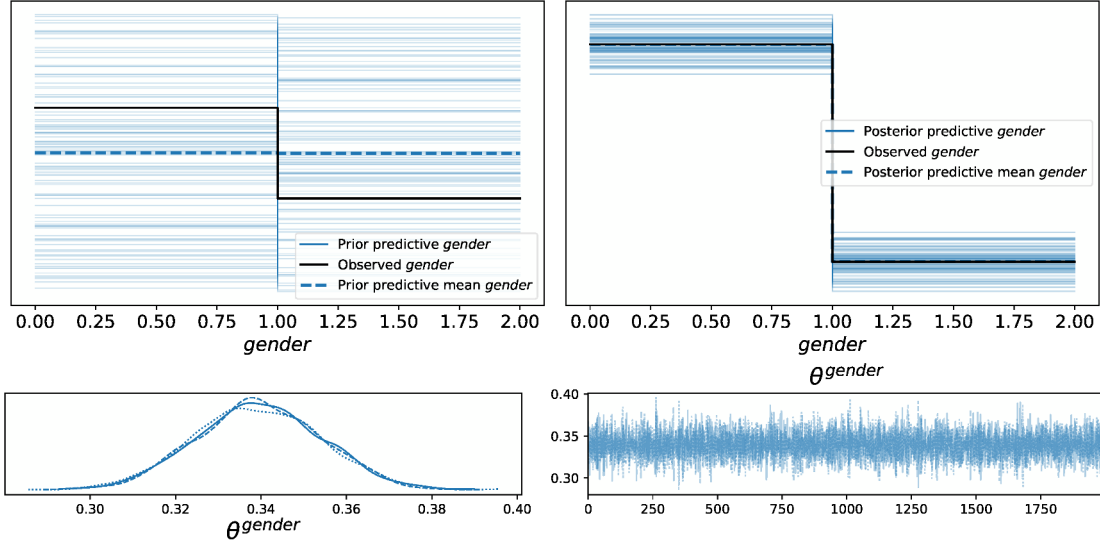


Figure 3.9: Posterior predictive and trace plots of the handle model and the gender variable.

nodes: *variables* and *factors*.

$$\mathcal{F} = (\mathbf{V}, \mathbf{F}, \mathbf{D}, \phi)$$

where

- \mathbf{V} , are variable nodes;
- \mathbf{F} , are factor nodes;
- \mathbf{D} , are dependency edges;
- $\phi : \mathbf{D} \rightarrow \{(x, y) \mid (x, y) \in \mathbf{V} \times \mathbf{F}\}$, is the incidence function mapping an unordered pair of variables and factors.

The dependence structure between factors and variables is expressed via the edges which can only connect variables with factors. A factor $f \in \mathbf{F}$ is a function $Val(V)$ to \mathbb{R}^+ that describes the *affinity* between the values from the variables $\hat{\mathbf{V}}$. For example, $f(Gender, Color) : Val(Gender, Color) \mapsto \mathbb{R}^+$ describes the factor over the values of *gender* and *color*. A concrete example would be $f(male, green) = 300$ where the actual value might be unnormalized or normalized, e.g., $f(male, green) = 0.35$. Finally, the scope of a factor $\text{Scope}[f]$ is the set of variables $\hat{\mathbf{V}}$ it factorizes. For example, $\text{Scope}[f] = \{Gender, Color\}$. Hence, a factor describes a joint distribution of multiple variables with their affinities (unnormalized) or probabilities (normalized). Multiple factors describe the factorization of multiple joint distributions $P(Gender, Color, Age) = \frac{1}{Z} f_0(Gender, Color) f_1(Color, Age)$. Each factor represents a local model that is combined into a global model and normalized via Z to describe a proper probability distribution. PSM uses factor graphs as a visual representation to describe the probabilistic quantities

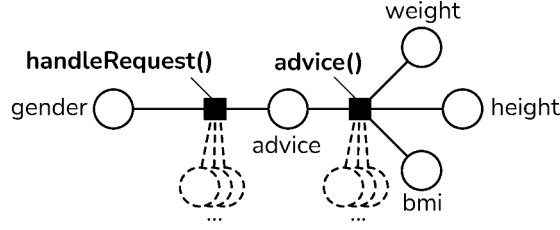


Figure 3.10: Example of the handle and advice method. Colored rectangles are factors and circles are variables.

of a program. Each executable and type will be described by a factor (a local model) over its inputs and outputs modeling. The combination of all factors describes the joint distribution of the entire program. Again, further details on factor graphs are given by Koller and Friedman [23].

PSM, itself does not directly use factor graphs but only uses it as a visual representation. Furthermore, understanding the basic notion of factor graphs helps in understanding cluster graphs.

3.2.3 Cluster Graph

A *cluster graph* [23] is an undirected graph over variables \mathbf{V} . Again, a cluster graph is tightly related to factor graphs and Bayesian networks representing the joint distribution of its variables. Nodes in a cluster graph are *clusters* representing a subset of the variables $\mathbf{v}^* \subset \mathbf{V}$. The undirected edges connect clusters with a non-empty intersection set of variables.

More formally, a cluster graph is a undirected cyclical graph defined over

$$\mathcal{C} = (\mathbf{C}, \mathbf{D}, \phi).$$

where

- \mathbf{C} , are cluster nodes aggregating multiple variables $\mathbf{v} \in C_i$;
- \mathbf{D} , are dependency edges connecting clusters and that are attributed with a non-empty sepset $\mathbf{S} \rightarrow \{f \mid fs \in C_i \cap C_j\}$;
- ϕ , is the incidence function connecting two clusters $\phi : \mathbf{D} \rightarrow \{(x, y) \mid (x, y) \in \mathbf{C} \times \mathbf{C}\}$.

Inference

Inference in cluster graphs is done via the *Sum Product Message Passing* [23] algorithm shown in Algorithm 1. First, a root cluster \mathbf{C}_r is selected containing the variable(s) of interest. Then, the algorithm passes the messages from the leaves inward to \mathbf{C}_r . Nb_i is the set of neighbors of a given cluster \mathbf{C}_i . $p_r(i)$ is the upstream neighbor of i , i.e., the

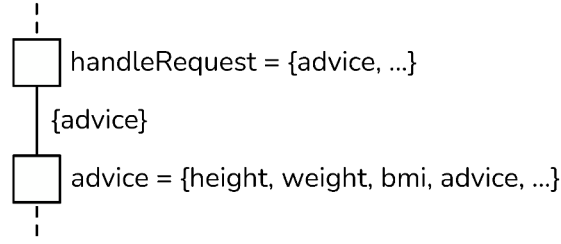


Figure 3.11: Cluster graph of the handle and advice method.

neighbor on the path to C_r . Each cluster computes a message and passes it towards the root node where a *sum-product message* is computed by

$$\delta_{i \rightarrow j} = \sum_{C_i - S_{i,j}} \psi_i \cdot \prod_{k \in (Nb_i - j)} \delta_{k \rightarrow i}. \quad (3.2)$$

All incoming messages are multiplied with the initial cluster potential C_i resulting a new factor ψ . Then it sums out all the variables that are not present in the sepset between C_i and C_j . $\beta_r(C_r)$, the *beliefs*, are the result once all messages have been passed up to the root cluster.

Algorithm 1: Sum Product Message Passing [23]

Function eliminate(ϕ, τ, α, C_r):

```

  while  $C_r$  is not ready do
    Let  $C_i$  be a ready cluster
     $\delta_{i \rightarrow pr_r(i)}(S_{i, pr_r(i)}) \leftarrow \text{sumProduct}(i, pr_r(i))$ 
     $\beta_r \leftarrow \psi_r \cdot \sum_{k \in Nb_{C_r}} \delta_{k \rightarrow r}$ 
  return  $\beta_r$ 

```

Function init():

```

   $\psi_i(C_i) \leftarrow \prod_{\phi_j: \alpha(\phi_j)=i} \phi_j$ 

```

Function sumProduct(i, j):

```

  // i ... idx sending cluster
  // j ... idx receiving cluster
   $\psi(C_i) \leftarrow \psi_i \cdot \prod_{k \in (Nb_i - j)} \delta_{k \rightarrow i}$ 
   $\tau(S_{i,j}) \leftarrow \sum_{C_i - S_{i,j}} \psi(C_i)$ 
  return  $\tau(S_{i,j})$ 

```

3.2.4 Flow-based Models

Bayesian networks, factor graphs, and cluster graphs represent a global independence structure that factorizes high-dimensional joint distributions into smaller factors (or

clusters). These factors can be represented via local probabilistic models like Conditional Probability Tables (CPD). CPDs are a tabular representation of discrete random values, e.g., gender with its values *female* and *male*, along with their probability in a given modeling setting.

PSM uses flow-based density estimators to approximate the local joint distribution of factors. For example, of the factor representing the `bmi` method with its variables `height`, `weight` and `bmi`. More concrete, it uses Non-Volume Preserving Transformations (NVPs) [30, 31] as local probabilistic models and cluster graphs as their global probabilistic data structure.

Architecture

NVPs are latent-variable density estimators that allow efficient inference, sampling, and likelihood estimation of samples. NVPs learn a deterministic bijective function $f : X \mapsto Z$ (with $g = f^{-1}$) that map the original input variable $x \in X$ to simpler latent variables $z \in Z$. This mapping is motivated via the change of variable theorem

$$p_X(x) = p_Z(f(x)) \left| \det \left(\frac{\partial f(x)}{\partial x^T} \right) \right| \quad (3.3)$$

where $\frac{\partial f(x)}{\partial x^T}$ is the Jacobian of f at x . The latent variables Z are often isotropic unit norm Gaussian $N(0, \mathbf{1})$ that are well understood in terms of sampling and likelihood evaluation. The flow learns the mapping between observed and latent variables but also how these variables depend on each other. Hence, the latent space is fully factorized alleviating the need of a covariance matrix in the latent Gaussian space. Figure 3.12 shows the probabilistic representation of the NVP of the `bmi()`-method. The observed variables *height*, *weight*, and *bmi* form a factor that is transformed by the deterministic flow into the latent factor. The latent variables are unit Gaussians, one for each observed variable.

NVPs can approximate complex high-dimensional data (e.g., images) even with small datasets [30] NVPs are build up by stacking multiple neural networks and combining them via flow modulation operators. These operators are scale and location transformations of the input data. While the transformations are simple by themselves, the combination of multiple layers of these allows the NVPs to approximate complicated multi-modal distributions. These transformation layers are called coupling layers. Figure 3.13 shows the coupling layer in its forward and inverse direction. x represents the input vector to a coupling layer with x_1 and x_2 being the left and right half of the input vector. The left half of the input, i.e., x_1 , is propagated as output y_1 without any change. The right half, i.e., x_2 , of the vector is scaled and translated with the left half and produces the new right output y_2 . The coupling layers alternate the transformation between the left and right half of the input which yielding a fully transformed x . The reverse propagation inverts the forward propagation by re-scaling and translating it back. The simplicity of the transformation makes the coupling layer invertible and differentiable. Furthermore, the Jacobian determinant can be efficiently computed by transforming only half of the input

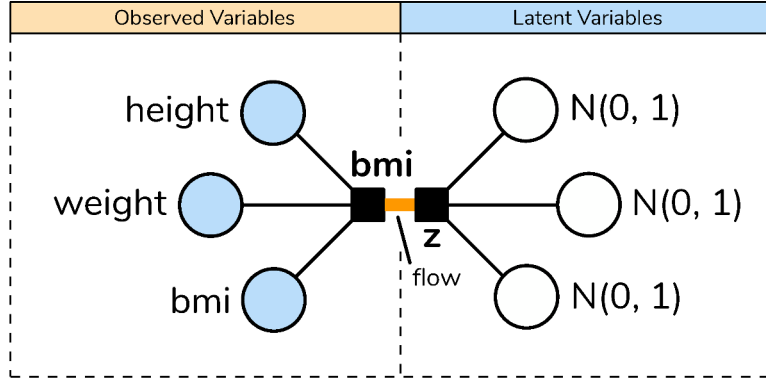


Figure 3.12: The figure shows the conceptual architecture of the **bmi**-method NVP. The observed variables represent the domain space. The latent variables represent, e.g., a Gaussian space. The flow is a bijective function mapping observed to latent variables.

per layer at a time. At this point, we will refer to Dinh et al. [30] for all mathematical proofs and details.

Parameter Optimization

NVPs are neural networks that are fitted by stochastic gradient descent [?]. A sample \mathbf{X}_0 is forwarded through the network resulting \mathbf{Z}_0 . Then we compute the loss by evaluating \mathbf{Z}_0 under the likelihood function of the latent variables while considering the change of volume. The general loss function is given by the log-likelihood of a sample computed via Equation (3.3).

$$\log(p_X(x)) = \log(p_Z(f(x))) + \log\left(\left|\det\left(\frac{\partial f(x)}{\partial x^T}\right)\right|\right), \quad (3.4)$$

where the log-likelihood of the sample (e.g., Gaussian log-likelihood) is multiplied by the Jacobian determinant representing the change in volume caused by the transformation. The resulting loss is then backpropagated, and the parameters are adapted accordingly.

Inference

PSM applications are built upon inference which consists of *sampling*, *conditioning*, and *likelihood evaluation*. This section discusses inference in the context of NVPs as the presented methods in this work solemnly use NVPs as local models. This also means that the technical realization of sampling, conditioning, and likelihood evaluation can change depending on the used local model.

Consider the running example with a model that describes the `BmiService.bmi`-method in form of $P(\text{height}, \text{weight}, \text{bmi})$ where *height* and *weight* are the parameters and *bmi* is the return value of the method.

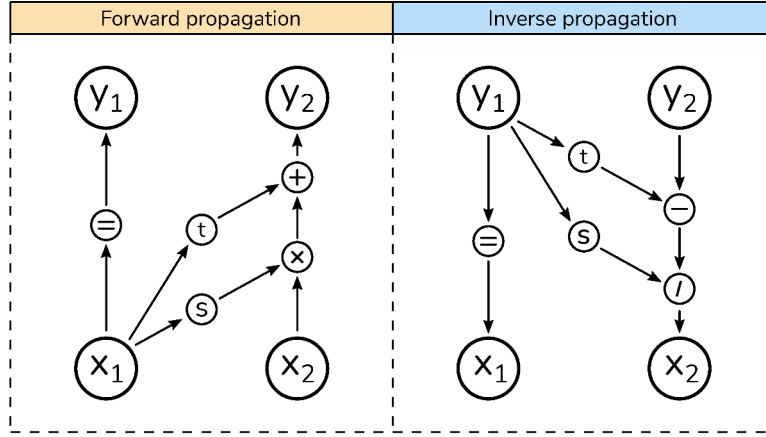


Figure 3.13: Architecture of the coupling layer. The forward path maps parts of the input directly to the output. The other parts of the input are scaled and translated.

Sampling

Sampling is the process of generating (or sample) observations from a given probabilistic model denoted by $x \sim P(X)$. For example, $(height, weight, bmi) \sim P(height, weight, bmi)$ generates triplets of observations that are distributed according to $P(height, weight, bmi)$.

In the context of NVPs, sampling is accomplished by drawing observations in the latent space and transforming it via the flow into values in the domain space. This process is illustrated via Figure 3.12 where we draw a sample from the latent Gaussian random variables and transform it via the flow into the observed variables. Hence, it is important to select latent variables that allow for easy and efficient sampling. In the case of an NVP with factorized unit Gaussians in the latent space, one only needs to sample from a univariate Gaussian.

Conditioning

Conditioning is the process of generating observations while fixing parts of the data to prior knowledge. For example, $(weight, bmi) \sim P(weight, bmi \mid height = \mathcal{N}(178, 2))$ generates *weight* and *bmi* values given an average person height of 178 cm.

Conditioning a sample can be done by optimizing the latent observations towards the observed condition via stochastic gradient descent. The process to condition is as follows, given a condition dataset \mathbf{c} containing N observations that representing the condition:

1. A random latent sample \mathbf{z} is drawn from \mathbf{Z} .
2. The observed sample is computed by flowing it through the NVP via $\mathbf{x} = g(\mathbf{z}^*)$.
3. The loss (i.e., mean squared error) is computed between the k dimensions of the condition dataset and the observed sample $\mathcal{L}(\mathbf{c}^k, \mathbf{x}^k)$.

4. Finally, the gradient regarding the condition and the generated sample is computed, and the latent observations are adapted accordingly.

Throughout the iterative process, the NVP acts as a deterministic function between \mathbf{x} and \mathbf{z} . Hence, the parameters of the NVP are immutable and not changed during the optimization. Only the N latent observations \mathbf{z} are updated during the back propagation of the loss $\mathcal{L}(\mathbf{C}^k, \mathbf{X}^k)$. Intuitively, we start with the marginal distribution of the NVP, reshaping it until it converges given our constraint \mathbf{c} . The optimization itself is very stable and reaches convergence in few iterations as only \mathbf{z} with N samples are optimized. This conditioning process can also be thought of as a data imputation process where \mathbf{c} acts as a constraint on the imputed data.

Likelihood Evaluation

The *likelihood evaluation* is the process of finding the likelihood of a sample under a given NVP. For example, $p(\text{height} = 178, \text{weight} = 80, \text{bmi} = 23)$ denotes the likelihood of the `bmi`-method being called with 178 cm, 80 kg and returning a BMI of 23. The process itself is the same procedure as one forward path during the parameter optimization.

1. Given a sample \mathbf{x} for which $\log(p_X(x))$ should be computed.
2. Transform the observed sample into its latent representation $\mathbf{z} = f(\mathbf{x})$.
3. Evaluate the log-likelihood via Equation (3.4).

4 Methodology

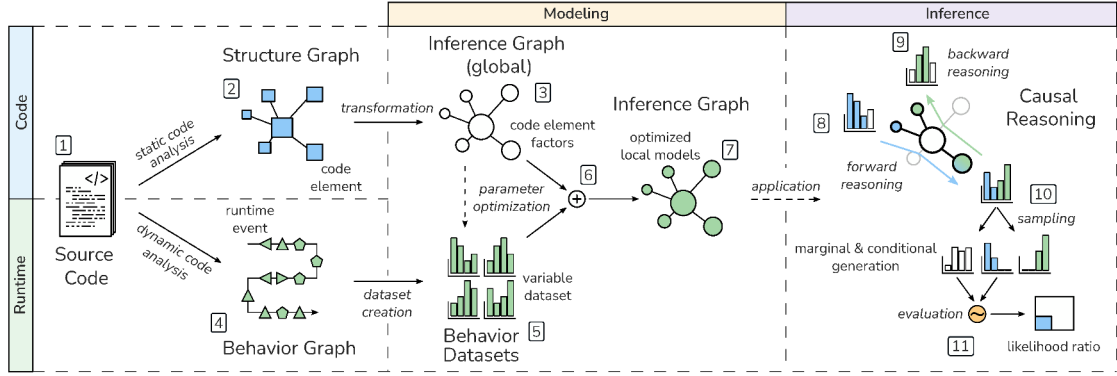


Figure 4.1: *Source Code* (1) has a *Program Structure* (2) and a *Runtime Behavior* (4) that is extracted via *Static* and *Dynamic Code Analysis*. These result in an *Inference Graph (global)* (5) and *Behavior Datasets* (4) that are combined by *Optimization* (6) into the final *Inference Graph* (7). The inference is then conducted by *forward reasoning* (8) or *backward reasoning* (9). The reasoning is based on *sampling* (10) and *likelihood evaluation* (11) that allows one to condition the IG into a specific state, or evaluate the likelihood of given states.

Probabilistic Software Modeling (PSM) is a four-fold approach illustrated in Figure 4.1.

1. **[Code]** Static code analysis extracts the program structure in form of a *Structure Graph*;
2. **[Runtime]** Dynamic code analysis observes the program behavior in form of a *Behavior Graph*;
3. **[Modeling]** Program structure and behavior are combined into a probabilistic model called *Inference Graph*;
4. **[Inference]** Applications are built by leveraging the inference graph and causal reasoning.

PSM starts by statically analyzing the *Source Code* (1) of a program. The result of this analysis is the *Program Structure* (2) in the form of a *Structure Graph* (SG). The *Structure Graph* contains information on the code elements (types, methods, and fields), their data types, and call dependencies. Then, PSM observes the running program and

extracts a *Behavior Graph (3)* (BG). The BG contains the event streams with events that represent, e.g., reading a value from a field.

The static and dynamic information is the foundation of the *Modeling* phase. The result of the *Modeling* phase is an *Inference Graph* that mirrors the structure and runtime behavior of the program. First, the structure graph is transformed into *Inference Graph (3)* capturing the global structure and dependencies between variables that map to code elements. Then, the event streams are transformed into a tabular format by tallying the stream and imputing missing data called *Behavioral Datasets (5)*. Each local model, i.e., each node in the inference graph, is optimized with the behavioral datasets to learn the original behavior of the program (6). The final result is the inference graph with the global structure reflecting the SG and the optimized local model reflecting the behavior of each code element (7).

The inference graph can then be used to create various applications (see Section 1.2). These applications are realized via *Causal Reasoning*. *Forward Reasoning (8)* starts with an upstream method and searches for the resulting effect in downstream methods. *Backward Reasoning (9)* starts with a downstream method and searches for the causing state in an upstream method. All the causal reasoning is built on top of the generative capabilities (10) and the possibility to evaluate the *likelihood (11)* of events.

4.1 Code

```

1 float bmi(float h, float w){
2     float hm = h / 100;
3     return w / (hm * hm)
4 }

```

Listing 4.1: Implementation of the BMI computation.

PSM extracts the *Structure Graph* from the *Source Code (1)* via static code analysis. First, an AST is constructed from the source code. Then, the AST is transformed into a Structure Graph (SG). An SG is an abstract semantics graph that represents high-level abstract concepts that PSM will model in later stages.

Figure 4.3 shows the meta-model of a structure graph with its *code elements* and their *relationships* (left). Types T , properties Pr , executables Ex , parameters Pa , and results Re refer to classes, fields, methods, method parameters, and method return values, e.g., in Java [32]. Types and executables are *compositional* elements representing a collection of atomic elements. Properties, parameters, and results are *atomic* code elements that have identifiable states at runtime. The structural relationship for compositional elements is that they *declare* code elements. For example, types declare executables and properties, or executables declare parameters and results. The structural relationship for atomic elements is an *instance of* relationship to a given type. The behavioral relationships center around executables and describe the interactions with other code elements. Executables *receive parameters* Pa^I , *read properties* Pr^I , and *request invocation results* Re^I , which describe its *Inputs* \mathcal{I} . Executables *return results* Re^O , *write properties* Pr^O , and *provide parameters* Pa^O , which describe its *Outputs* \mathcal{O} .

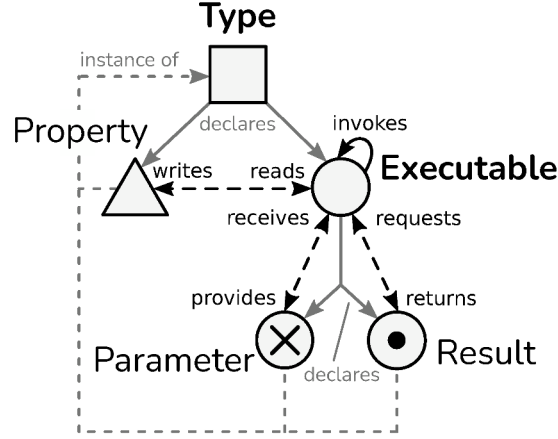


Figure 4.3: The meta-model of the structure graph. Types and executables are compositional elements that may declare atomic elements like properties, parameters, or results. Atomic elements are later the variables of the model and can be monitored at runtime.

More formally, a structure graph is a directed cyclic graph defined over the meta-model

$$\mathcal{S} = (\mathbf{C}, \mathbf{R}, \varphi)$$

where

$$\begin{aligned} \mathbf{C} &= \{T, Pr, Ex, Pa, Re\} \\ \mathbf{R} &= \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, \\ &\quad b_0, b_1, b_2, b_3, b_4, b_5, b_6\} \end{aligned}$$

and φ is defined by

$$\begin{aligned} \varphi(s_0) &= T^1 \rightarrow^* Pr & \varphi(s_1) &= T^1 \rightarrow^* Ex & \varphi(s_2) &= Ex^1 \rightarrow^* Pa \\ \varphi(s_3) &= Ex^1 \rightarrow^* Re & \varphi(s_4) &= Pr^* \rightarrow^1 T & \varphi(s_5) &= Pa^* \rightarrow^1 T \\ \varphi(s_6) &= Re^* \rightarrow^1 T \end{aligned}$$

for structural relationships and

$$\begin{aligned} \varphi(b_0) &= Ex^* \rightarrow^* Ex & \varphi(b_1) &= Pr^* \rightarrow^* Ex & \varphi(b_2) &= Ex^* \rightarrow^* Pr \\ \varphi(b_3) &= Pa^* \rightarrow^1 Ex & \varphi(b_4) &= Ex^* \rightarrow^* Pa & \varphi(b_5) &= Re^* \rightarrow^* Ex \\ \varphi(b_6) &= Ex^1 \rightarrow^* Re \end{aligned}$$

for behavioral relationships with

$s_0, \dots s_3$...declares
$s_4, \dots s_6$...instance of
b_0	...invokes
b_1, b_2	...reads, writes
b_3, b_4	...receives, provides
b_5, b_6	...requests, returns.

The nodes \mathbf{C} are called the *code elements*, the edges $s_0 \dots s_6 \in \mathbf{R}$ are called *structural relationships*, and edges $b_0 \dots b_6 \in \mathbf{R}$ are called *behavioral relationships*. φ is the incident function connecting nodes. φ for \mathcal{S} also defines the possible cardinalities similar to the Unified Modeling Language [33] for the source and target nodes. Further, we define $<$ as the *instance of* operator. For example, $person < Pr$ describes that *person* is an instance of a parameter. \mathcal{S} describes the meta-model of structure graphs $\hat{\mathcal{S}}$ that is extracted from an AST. Given the meta-model of a structure graph, the transformation from an AST to an SG is a simple exercise. Tools like Spoon [34] can simplify the SG generation substantially and do not need any further elaboration.

PSM focuses on code elements. Therefore, a concise notation is beneficial in the later chapters. Atomic elements are denoted in lowercase while compositional in boldface lowercase, e.g., h and \mathbf{bmi} in Listing 4.1. Results have the name of their executables, e.g., bmi in Listing 4.1. $\mathbf{bmi} = \{h^{Pa, \mathcal{I}}, w^{Pa, \mathcal{I}}, bmi^{Re, \mathcal{O}}\}$ denotes the code elements of Listing 4.1. We will omit the superscript classifiers if it is unambiguously possible, e.g., $\mathbf{bmi} = \{h, w, bmi\}$. $\mathbf{bmi}^{\mathcal{I}} = \{h, w\}$ and $\mathbf{bmi}^{\mathcal{O}} = \{bmi\}$ describes the subset of *inputs* and *outputs*. Finally, we define the symbol function $symbol(\cdot)$ that returns a numeric identifier for each code element, e.g., $symbol(\mathbf{bmi}) = 0$.

Figure 4.7 (top) shows the ASG of the Nutrition Advisor. It includes the types `Servlet`, `NutritionAdvisor`, `BmiService`, `Person`, and the executables `handle`, `advice`, `bmi`, `init`. `init` is the constructor of the `Person` type. Furthermore, all atomic code elements are shown with their interdependencies. Note that most of the structural relationships are omitted in Figure 4.7 for visual simplicity.

Constructing the ASG does not need any further explanation. There are many libraries that construct a general-purpose ASG from which to start (e.g., Spoon for Java [34]).

4.1.1 Model Universe

The *Model Universe (MU)* describes which code elements from the program are captured by the IG. Each code element in the MU is considered during runtime monitoring, dataset creation, and in the IG. The selection of code elements is limited by four factors: the users, technical and legal reachability, and PSM type system. *Users* might limit the MU if they are only interested in a subset of the program. The *technical reachability* limits the MU to elements that can be statically analyzed and monitored at runtime.

Table 4.1: Compatibility matrix of Java to PSM data-types. * refers to iterable types like `ArrayList` in Java. <any> refers to any other type not mentioned in the table.

	void(s)	int(s)	float(s)	text(s)	reference(s)	unknown(s)
void(*)	X					
short(*)		X				
boolean(*)		X				
byte(*)		X				
short(*)		X				
integer(*)		X				
long(*)		X				
float(*)			X			
double(*)			X			
char(*)				X		
String(*)				X		
<any>(*)					X	X

For example, intercepting calls to JDK system libraries might be prohibited by Java Virtual Machine (JVM) [35] for security reasons. The *legal reachability* limits the MU to elements for which static and dynamic analysis is legally allowed. For example, analyzing third-party libraries might be illegal given their license. Finally, the *PSM type system* limits the MU to data types that can be represented via a probabilistic model. In general, PSM models *numbers* and *text* or any coercible data type. Table 4.1 contains the most essential data type mappings implemented in the PSM prototype that runs on the JVM. Any number or boolean is mapped to integers. Any floating-point number is mapped to floats. Any string-like is mapped to text. Any other type is either a reference, i.e., a complex type, or unknown to the system.

Any code element in the meta-model (see Figure 4.3) can be added to the MU given the aforementioned constraints. Atomic code elements are directly added to the MU. Compositional elements are added by adding the element itself, and recursively all direct atomic children, e.g., adding the `Person` will add all the properties and methods declared via `Person`. Adding code elements in a PSM prototype implementation equates to annotating code elements, e.g., annotating a field in a class, with `@GradientModel`. Alternatively, all included and excluded code elements can be described by a list of regular expressions.

4.2 Runtime

PSM extracts the *Behavior Graph* (3, see Figure 4.1) via *dynamic code analysis* [21, 36]. The program is *instrumented* and executed via a *trigger* resulting in a *stream of runtime events*. The *instrumentation* tracks predefined program execution points, e.g., executable invocations or property accesses. The *trigger* is the program input and any initial state that is configured to run the program, e.g., test-suites, system tests, or program execution. The resulting *event streams* are directed acyclic graphs of events similar to the sequence diagram Figure 2.2.

The SG defines the structure of the IG. The BG defines the behavior, or the dynamic aspect, of the IG. Without the information of the BG the IG would operate solely on the prior knowledge similar to the example in Figure 3.2. Hence, SG and BG are essential in finding an usable and useful IG.

4.2.1 Instrumentation

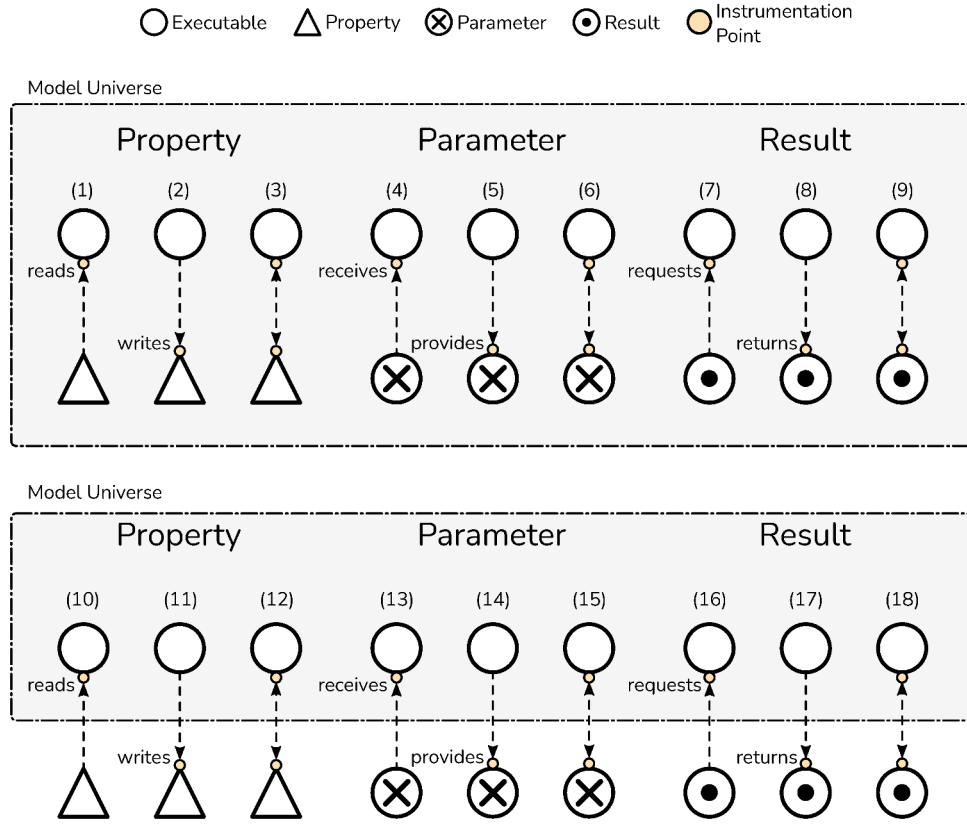


Figure 4.4: The instrumentation points of dynamic analysis are used in PSM. Any access to an atomic element in the model universe is tracked. Similarly, any access to atomic elements at the boundary of the model universe. However, outward facing relationships (i.e., writes and provides) may be optional if they are not needed in the context of the application.

PSM instruments a program by injecting instrumentation points into its binary code. These instrumentation points report a predefined set of events resulting in the BG. As discussed in Section 4.1, code elements are the lowest level of abstraction in PSM. Hence, the behavioral relationships in the meta-model of the program structure in Figure 4.3 already indicate the instrumentation points. In total, there are 18 different cases shown in Figure 4.4. Each case starts from an executable that interacts with a target code element. Cases 1 to 9 refer to instrumentation points where both the source executable and the target code element are in the model universe. Cases 10 to 18 refer to instrumentation points where only the source executable is in the model universe while the target code element is not. In general, any interaction with any code element needs to be instrumented. This is true if both source and target elements are in the MU, but also if only the source

element is in the MU.

If both source and target code elements are in the MU, then all interactions have to be instrumented to capture the entire behavior. It is still important to model the external target element in many cases even if only the source element is in the MU. For example, a function might not return the result but push it into an object that is an instance of a type that does not reside in the MU (e.g., an output stream). To enable full inference with the IG from program input to output one would still need to understand the input parameters of the output stream method. However, there are cases where the external target element is not of interest. In this case, it is useful for the PSM implementation to provide a facility to exclude these elements from the MU to reduce the resource demand and model dimensionality.

4.2.2 Program Triggers

Most programs need to be triggered to expose runtime behavior. The trigger might be provided at the start of the program, e.g., a compiler is triggered with a text file containing the source code that needs to be compiled. Other applications might be triggered continuously at runtime, e.g., the Nutrition Advisor runs on a web server receiving constantly new requests representing the triggers.

Triggers themselves can either be synthetic or natural. *Natural* triggers are triggers that were collected in the production environment of the program, e.g., invocations from real clients. *Synthetic* triggers are triggers that were generated and collected in the development environment of a program, e.g., test data in test suites, random data during fuzzing, etc. Depending on the PSM application that is realized one might prefer synthetic or natural triggers. For example, visualization and comprehension make the most sense with natural triggers to gain insights into the production environment. In contrast, for semantic clone detection synthetic triggers suffice as one is only interested in comparing the effect given input data. Usually, program triggers only represent a subset of all possible inputs. This means that also the IG will only represent a subset of the behavior. Hence, it is important to select the right program trigger relative to the PSM use case that is realized.

4.2.3 Behavior Graph

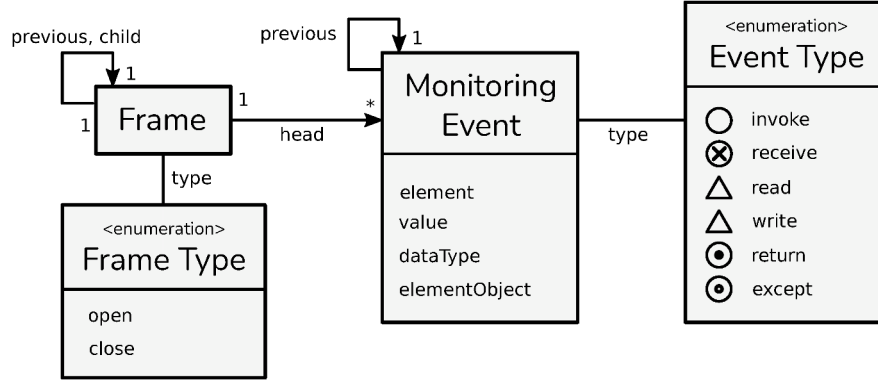


Figure 4.5: The meta-model of the behavior graph. Frame events delimit executable frames. Monitoring events track invocations, exceptions, or accesses to properties.

Runtime monitoring results in a directed acyclic graph of events called the *Behavior Graph (BG)*. The BG consists of one or multiple traces depending on the number of execution units (thread or process). Each trace itself is a Directed Acyclic Graph (DAG) with its meta-model shown in Figure 4.5. Each trace contains a backward linked list of *Monitoring Events*. Each monitoring event has a specific *Event Type* representing the observed executable interaction, e.g., an invocation or a property read. Each executable spawns a frame on the call stack. Further, there are *Frame* events capturing the execution scope of a method. Frames are linked given their parents. Furthermore, each frame points to the proportion of the event stream that happened in its scope. Frame events have a type that identifies them as opening or closing events of the frame. More formally,

$$\mathcal{B} = (\mathbf{E}, \mathbf{L}, \varphi) \quad (4.1)$$

where

$$\mathbf{E} = \{Fr, Mo\} \quad (4.2)$$

$$\mathbf{L} = \{l0, l1\} \quad (4.3)$$

and φ is defined by

$$\varphi(l0) = Fr \xrightarrow{1} Mo \quad (4.4)$$

$$\varphi(l1) = Mo \xrightarrow{1} Mo \quad (4.5)$$

$$(4.6)$$

Figure 4.6 shows an example trace of the Nutrition Advisor. The upper half visualizes the initialization phase while the lower half shows the handling of a request. The red

thread through the monitoring events is the backward linked lists through all events. Each call stack level is depicted via a color that indicates the child's relationship between different stack levels. Multiple threads would spawn multiple such BGs. In conclusion, the BG is a DAG consisting of interaction events of code elements that are in the MU.

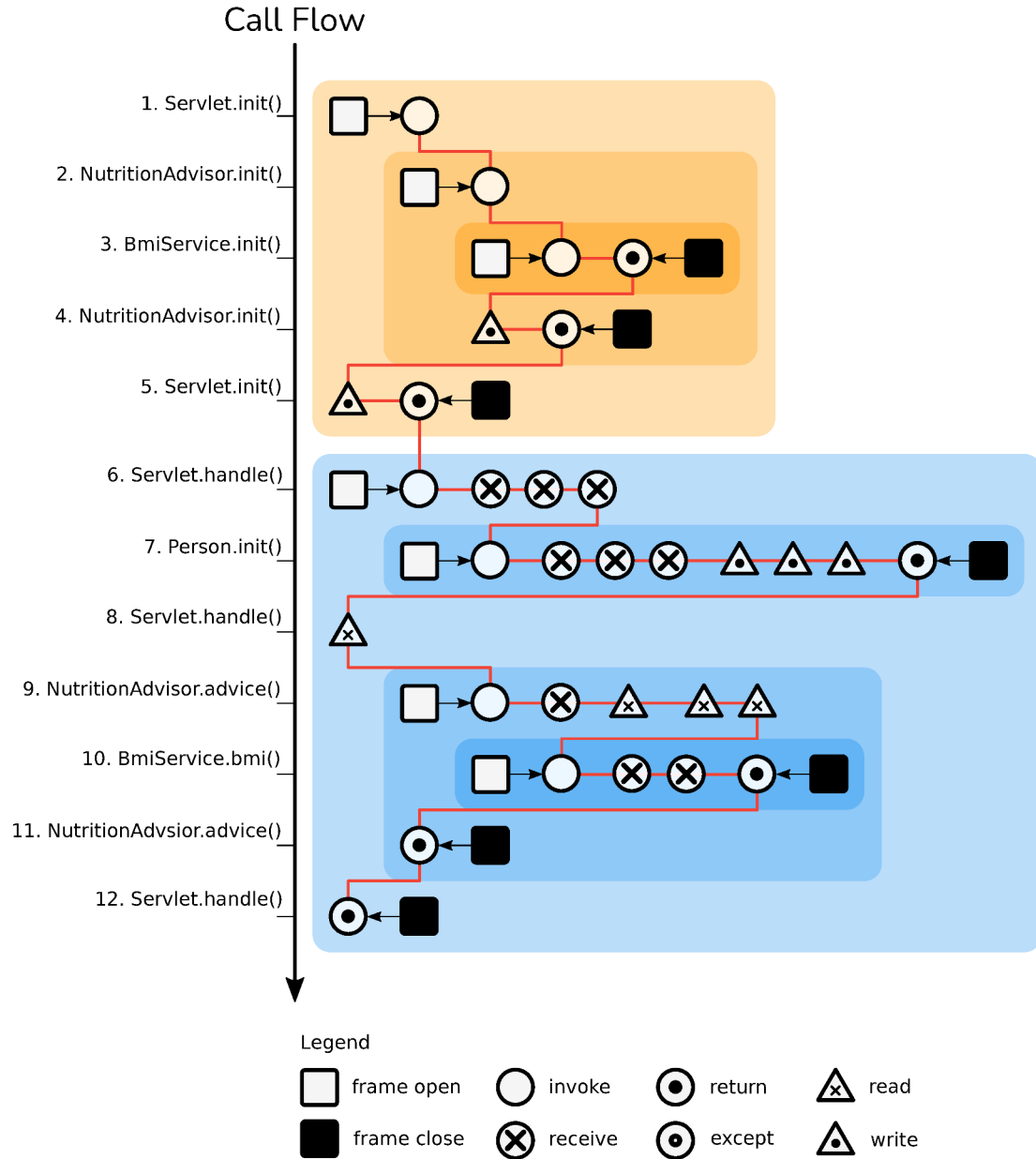


Figure 4.6: An example behavior graph of the initialization and one request of the Nutrition Advisor.

4.3 Modeling

The *Modeling* aspect in Figure 4.1 combines the Structure Graph (SG) and Behavior Graph (BG) into the Inference Graph (IG) that mirrors the original program (7). First, the SG is transformed into the prior IG. This IG has the same structure as the program, i.e., types, properties, methods, parameters, and results are represented via random variables and factors organized in the same topology (3). However, the local models (i.e., nodes/variables) solely operate on the predefined latent space without learning from the collected observed data (see Figure 3.2). Then, the BG is transformed into *Behavior Datasets* (5). These datasets are used to learn the posterior of the local models in the IG (6). The result is the final *IG (posterior)* (7) with which PSM applications can be built. Applications are realized via forward (8) and backward reasoning (9), sampling (10), and likelihood evaluation (11) in the IG.

4.3.1 Inference Graph — Global Structure Creation

The goal of the Inference Graph (IG) is the same with Bayesian networks, factor, or cluster graphs. Probabilistic modeling is concerned with finding a good dependency structure of random variables that approximates a given joint distribution $P(\mathcal{X})$. In many real-world situations, only the joint distribution is known or observed. The structure of the processes that generated the joint distribution is often unknown. In addition, often many variables cannot be observed directly (latent variables) even if parts of the structure are known. For example, modeling the process of coin flips allows us to observe the result of multiple flips, but not the composition of the coin that may lead to some bias in the flips.

PSM however, has a unique advantage in that it knows which process created the joint distribution. The variables and their dependency structure are known via the SG. All variables in the SG can be observed, and the observations are represented via the BG. Only one question is left: Which variables are explicitly modeled, and which are abstracted into data? The IG models types, properties, executables, parameters, and results explicitly as the meta-model from the SG in Figure 4.3 already hints. These represent the *observed* variables in the IG. Statements, e.g., like the control flow statements, are abstracted as unknowns and are only indirectly visible via the observed data.

This is a *design decision* that balances scalability and usability. *Scalability* because only the stack trace and property accesses need to be observed at runtime. *Usability* because it pushes the program structure and its first-class elements with their relationships into the center of the analysis, instead of algorithmic details. Adding statements would substantially increase the runtime monitoring overhead e.g., by loops that need to be traced. Also, modeling statements would increase the size of the IG up to a point where it would be unusable for inference and comprehension by humans. In conclusion, the PSM IG is a very close representation of the original generative processes that are modeled and the abstractions balance scalability and usability.

Creating the IG involves a graph transformation from an SG to an IG. The IG is a factor or cluster graph and each representation is equivalent in its dependence structure.

However, each representation shows different levels of details regarding the structure and is useful in different settings. Any element in the IG is called a model element. This applies to random variables, factors, and clusters. Any model element has an associated code element in the SG. In that, we use the term model element to refer to code elements that are modeled via the IG.

Structure Graph to Factor Graph

The first step is to transform the *Structure Graph* to a *Factor Graph*. This transformation involves the mapping of code elements to the factors and random variables as illustrated in Figure 4.7. Compositional code elements (types, executables) are mapped to factors. Atomic code elements (properties, parameters, results) are mapped to random variables. More formally, given an SG and a factor graph

$$\begin{aligned}\mathcal{S} &= (\mathbf{C}, \mathbf{R}, \varphi_{\mathcal{S}}), \\ \mathcal{F} &= (\mathbf{V}, \mathbf{F}, \mathbf{D}, \varphi_{\mathcal{F}}),\end{aligned}$$

then, the graph morphism from $\varphi_{\mathcal{SF}} : \mathcal{S} \rightarrow \mathcal{F}$ with pair mappings

$$\begin{aligned}\varphi_{V_{CF}} &: \mathbf{C}_c \rightarrow \mathbf{F}, \\ \varphi_{V_{AV}} &: \mathbf{C}_a \rightarrow \mathbf{V}, \\ \varphi_{E_{RD}} &: \mathbf{C}_b \rightarrow \mathbf{D},\end{aligned}$$

transforms \mathcal{S} to \mathcal{F} .

The mappings define the subset of code elements and how they are mapped to factor elements.

- $\varphi_{V_{CF}}$ maps non-empty compositional code elements that exist in the MU to factors with $\mathbf{C}_c = \{c \mid c < \{T, Ex\} \wedge c \in \mathbf{MU} \wedge c \neq \emptyset\}$.
- $\varphi_{V_{AV}}$ maps atomic code elements that exist in the MU to random variables with $\mathbf{C}_a = \{a \mid a < \{Pr, Pa, Re\} \wedge c \in \mathbf{MU}\}$.
- $\varphi_{V_{RD}}$ maps behavioral relationships to dependencies.

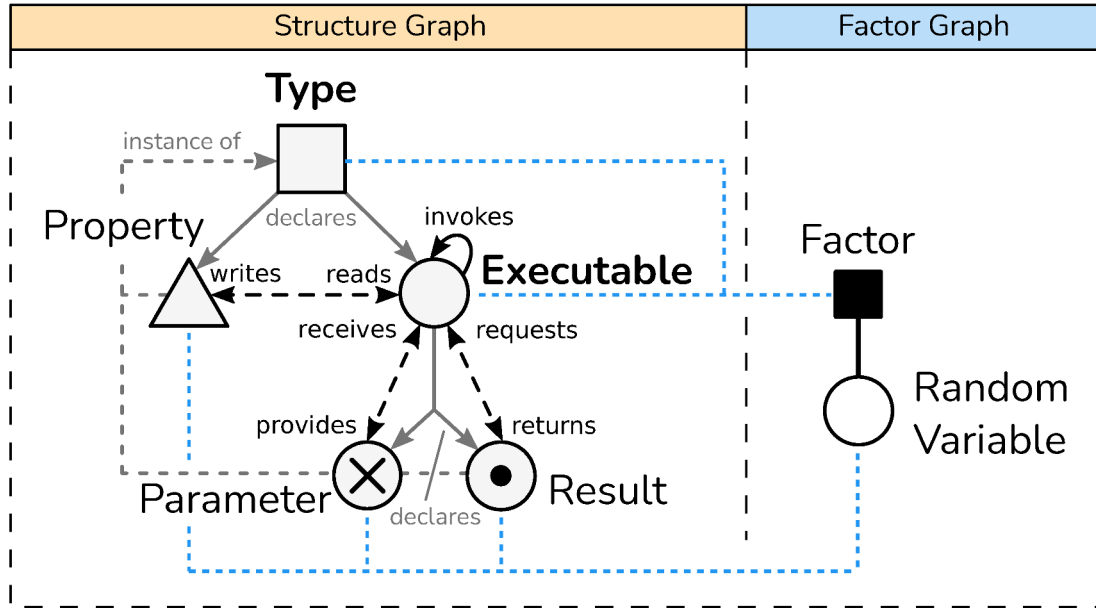


Figure 4.7: The mapping between the structure graph and factor graph metamodel. Compositional elements are mapped to factors while atomic elements are mapped to variables.

Figure 4.7 shows the mapping between the meta-models of the structure and the factor graph. The structural relationships of the structure graph are lost during the mapping. Hence, an inverse transformation from factor to structure graph is not possible without additional information.

An instance of such a transformation is given in Figure 4.8. Structural relationships and mapping lines between \mathcal{S} and \mathcal{F} are omitted for visual brevity. \mathcal{S} and \mathcal{F} are very similar as Figure 4.8 shows. The factor graph consists only of elements that exist in the MU. As described in Section 4.1.1, probabilistically modelable elements are present in the MU. Hence, the **person** parameter of the **advice** method is not present in the factor graph. However, its content, i.e., **weight** and **height**, exists in the factor where the original program dereferences the complex type. This happens in the **init**-method of **Person** but also in the **advice**-method. Furthermore, there are also no factors for compositional elements that do not have any code element in the MU, e.g., **Servlet**, **NutritionAdvisor**, **BmiService**.

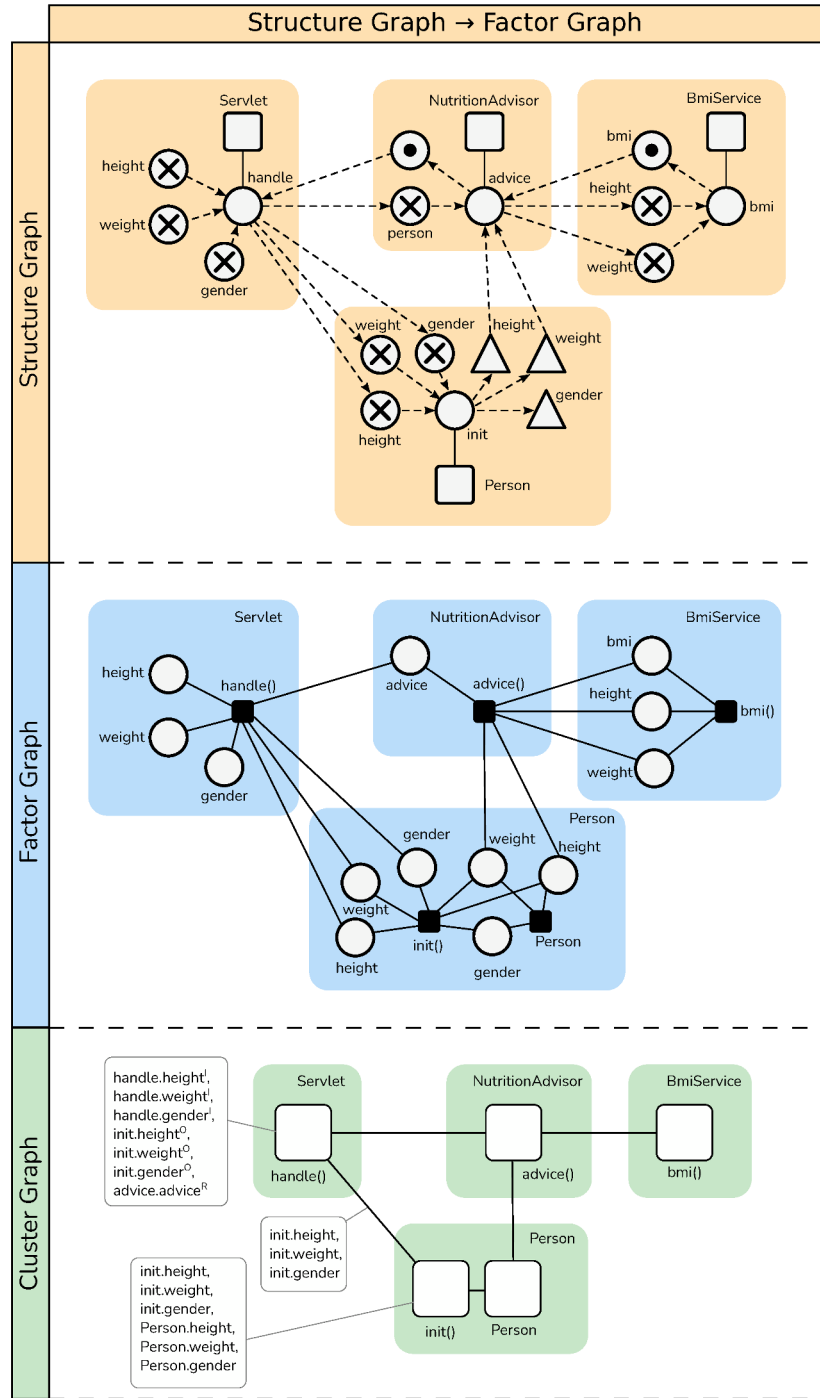


Figure 4.8: The top part shows the abstract semantic graph of the Nutrition Advisor with all its code elements. Most of the structural relationships are omitted for visual simplicity (e.g., *person instanceof Person*). The middle part shows the factor graph corresponding to the ASG. Reference elements are missing as empty compositional elements. The bottom part shows the cluster graph representation. Two clusters have their variables annotated along with one sepset.

4.3.2 Factor Graph to Cluster Graph

The *Cluster Graph* representation of the IG is structurally equivalent to the factor graph. Each factor is transformed into a cluster with the variables being part of it. More formally, given a factor graph and a cluster graph

$$\begin{aligned}\mathcal{F} &= (\mathbf{V}, \mathbf{F}, \mathbf{D}_{\mathcal{F}}, \varphi_{\mathcal{F}}), \\ \mathcal{C} &= (\mathbf{C}, \mathbf{D}_{\mathcal{C}}, \varphi_{\mathcal{C}}),\end{aligned}$$

then, the graph morphism from $\varphi_{\mathcal{FC}} : \mathcal{F} \rightarrow \mathcal{C}$ with pair mappings

$$\begin{aligned}\varphi_{V_{FC}} &: F \rightarrow C, \\ \varphi_{E_{DD}} &: D_{\mathcal{F}} \rightarrow D_{\mathcal{C}},\end{aligned}$$

transforms \mathcal{F} to \mathcal{C} .

The mappings define the factor graph elements and how they are mapped to cluster elements.

- $\varphi_{V_{FC}}$ maps factors to clusters with its variables $\text{Scope}[C_i] = \text{Scope}[F_i]$.
- $\varphi_{E_{DD}}$ maps dependencies between factors given by the connecting variables to dependencies in the cluster graph. The connecting variables form the sepset $S_{i,j} \subseteq C_i \cap C_j$.

Important in this transformation is that the resulting cluster graph is *family preserving* and adheres to the *running intersection property*.

Definition 1. A cluster graph is family-preserving when each factor $f \in \mathbf{F}$ is associated with a cluster C_i , such that $\text{Scope}[f] \subseteq C_i$ [23].

$\varphi_{V_{FC}}$ is a bijective mapping between a factor and a cluster along with its scopes $\text{Scope}[\mathbf{C}] = \text{Scope}[\mathbf{F}]$. Hence, the property is given through the construction of the cluster graph.

Definition 2. A cluster graph satisfies the running intersection property if, whenever there is a variable X such that $X \in C_i$ and $X \in C_j$, then there is a single path between C_i and C_j for which $X \in S_e$ for all edges e in the path [23].

Figure 4.7 shows an example of the factor and cluster graph transformation. Again, for visual brevity, not all sepsets and cluster variables are shown but only a representative subset. Each factor is a cluster with the same scope as the original factor. The sepsets are the elements that intersect between two clusters. The topology of elements between structure, factor, and cluster graph is essentially equal as all represent the same underlying program structure. The biggest difference is in the level of abstraction.

In conclusion, the IG is either a factor or cluster graph constructed from the structure graph closely representing the original program. What is left is now to instantiate the local probabilistic models for each cluster and optimize them towards the observed runtime data.

Truncated Inference Graph

The IG can either be a factor or cluster graph containing types, executables, properties, parameters, and is extracted from the structure graph. In Figure 4.1 we can see that the IG is the basis for all PSM applications that are implemented. The presented IG in the previous section represents the full definition in which full causal reason is possible. However, there are many examples of applications in which causal inference is not needed. For example, semantic clone detection, which is presented in Chapter 7 does not need global clausal reasoning but only the likelihood evaluation on local models. For these applications, it would be wasteful to fit the entire model. Any IG that only models parts of the program is called a *Truncated Inference Graph*. One truncated version of the IG is to skip the type models. For example, in the context of semantic clone detection, only the behavioral differences between methods are important. Neither type models nor global causal reasoning is used.

In conclusion, a *truncated inference graph* is an IG that is missing specific code elements. This is useful for applications in which global inference (in the cluster graph) is not needed but only local inference (within the clusters).

4.3.3 Behavior Datasets

The *Behavior Graph* (4) contains all the observed data collected at the runtime of the program. The *Dataset Creation* tallies the event stream into *Behavior Datasets* (5) for each model element. This process involves temporal alignment of events but also the imputation of missing values.

Section 4.2 described the BG and how it represents the runtime behavior. The BG contains the runtime information in form of *invoke*, *receive*, *read*, *write*, *return*, and *except* events. Each event is associated with a code element from the SG. In addition, *receive*, *read*, *write*, *return*, and *except* events also contain the observed value and its PSM data type (see Table 4.1). Recall that clusters and factors are compositional elements, i.e., types or executables. Thus, there are two different dataset creation processes.

In the following section, we will define the dataset creation in a formal way as close as possible to the implementation of the prototype. However, there are many ways to impute missing data and stratify the observations of the code elements. This means the presented stratification and imputation is *a* solution but not necessarily the only or best solution for every use-case. We define the following functions for the dataset creation

- $\phi(e) = e^{open}$ returns the *frame open event* to which e belongs;
- $val(e)$ extracts the value from an event, i.e., $e.value$;
- $type(e)$ returns the event type, i.e., $e.type$
- $element(e)$ returns the element of a given event, i.e., $f.element$;
- $head(f)$ returns the first monitoring event of a frame event, i.e., $f.head$;
- $child(f)$ returns the child frame event of a given frame event, i.e., $f.child$;

- A modified value function that either returns the value of an event or a given default.

$$\widehat{Val}(e, c, x) = \begin{cases} \text{Element}(e) = c, & \text{val}(e) \\ \text{Element}(e) \neq c, & x \end{cases} \quad (4.7)$$

- `object(f)` that returns the object on which the event happened, i.e., `f.elementObject`;

Type Datasets

Types are compositional elements that consist of their properties. Hence, type datasets will contain columns each representing a property. The data points in the type datasets come from the state changes of properties throughout the lifetime of an object. Hence, the context of a property is the lifetime of an object. More formally, there exists a BG and IG in form of a factor graph

$$\begin{aligned} \mathcal{B} &= (\mathbf{E}, \mathbf{L}, \varphi_{\mathcal{B}}), \\ \mathcal{F} &= (\mathbf{V}, \mathbf{F}, \mathbf{D}\varphi_{\mathcal{F}}), \end{aligned}$$

and the functions defined in Section 4.3.3. The dataset for a given type T with its properties \mathbf{Pr}^T is then defined over the object datasets \mathbf{o}_i with

$$\mathcal{D}^T = \bigcup_i \mathbf{o}_i$$

An object dataset is then a tuple of data points defined by

$$\mathbf{o} = (d_0, \dots, d_{i-1}, \delta(e_j, d_{i-1}), \dots, d_n)$$

where $e_j \in \mathbf{E} \wedge \text{Element}(e) \in \mathbf{Pr} \wedge \text{Object}(e) = o$. A data point d_i is created by

$$\delta(e, d_{i-1}) = (\widehat{Val}(e, \hat{p}, x) \mid (\hat{p}_j, x_j) \in \mathbf{Pr} \times d_{i-1} \mid j \in \mathbf{I})$$

where \mathbf{I} is the index set of size $|\mathbf{Pr}|$. d_{i-1} represents the previous data point, i.e., the previous state of the object.

Intuitively, we create a data point with one entry for each property in a type. Each state change of the object will generate a new data point with the new entry at the associated tuple position. The scope of a state change may be defined such that each *write* indicates a state change. Another possibility is to delineate a state change by an executable frame, e.g., the execution of the `init`-method. A third possibility is to delineate a state change by an executable scope and a repeated change of a property. Each possibility has advantages and disadvantages. Delineating a state change by any write may lead to noise in the data, e.g. if a Person object is reused. For example, the data may contain persons with male height and female weight if gender is the first property that is changed. Delineating a state change by frames may ignore assignments that happened

Table 4.2: addme

Person	height	weight	gender
0	174.30	90.00	Male
1	157.80	97.60	Female
2	159.10	69.80	Male

within a frame if an object is reused multiple times. Delineating a state change by frames and repeated writes partially solves this problem but might lead to more complex logic. Finally, a PSM implementation may offer developers to annotate properties if they diverge from a given default state change mode. Our prototype delineates by frames per default. An example dataset for **Person** with properties **height**, **weight**, and **gender**, is given in Table 4.2. Each state change in **Person** given by the `init()`-method results in one data point in the dataset.

In conclusion, the type datasets are constructed by collecting the values from the event stream grouped by code elements and objects.

Executable Datasets

Executables are compositional elements that consist of parameters, results, and properties. Again, each column in an executable dataset will represent one of these elements. The data points in the type datasets come from the initial executable invocation. Furthermore, any interaction with an external relationship may trigger additional data points. The external relationships may be accesses to properties or invocations of dependent executables. The context of executable datasets is therefore the executable frame, i.e., the lifetime of the executable on the call stack. More formally, there exists a BG and IG in the form of a factor graph

$$\begin{aligned}\mathcal{B} &= (\mathbf{E}, \mathbf{L}, \varphi_B), \\ \mathcal{F} &= (\mathbf{V}, \mathbf{F}, D\varphi_{\mathcal{F}}).\end{aligned}$$

The dataset for a given executable \mathbf{Ex} with its relationships $c \in \mathbf{Ex}$ is then defined over the union of all frame datasets \mathbf{f}_i with

$$\mathcal{D}^{\mathbf{Ex}} = \bigcup_i \mathbf{f}_i.$$

A frame dataset \mathbf{f} is then a tuple of data points defined over a fixed but arbitrary frame open event e_0^{open} where $\text{element}(\text{head}(e_0^{open})) = \mathbf{Ex}$

$$\mathbf{f} = (d_0, \dots, d_{i-1}, \delta(e_j, d_{i-1}), d_{i+1}, \dots, d_n)$$

where $e_j \in \mathbf{E} \wedge \text{element}(e_j) \in \mathbf{Ex} \wedge \phi(e_j) = e_0^{open}$. An arbitrary but fixed data point d_i for the frame is then created by

$$\delta(e, c, d) = (\widehat{Val}(e, c, x_j) \mid (c_j, x_j) \in \mathbf{Ex} \times d \mid j \in \mathbf{I})$$

Table 4.3: addme

advice	height	weight	bmi	advice
0	174.30	90.00	29.62	Consider skipping the meal.
1	157.80	97.60	39.19	Please do not eat me!
2	159.10	69.80	27.57	Consider skipping the meal.

where \mathbf{I} is the index set of size $|\mathbf{Ex}|$. d_{i-1} is the previous data point in the frame dataset and indicates a forward filling of missing values. Important is that forward or backward filling depends on the order of code elements in \mathbf{Ex} . For simplicity, we omitted the backward filling as it works analogues.

Intuitively, we create multiple data points per frame depending on the amount of interaction with code elements that are not declared in \mathbf{Ex} . The logic of creating multiple data points per frame is similar to creating multiple data points per object state in Section 4.3.3. Table 4.3 shows an example of the dataset of the **advice**-method.

In conclusion, the executable datasets are constructed by collecting the values from the event stream grouped by code elements and frames.

4.3.4 Inference Graph — Local Structure Optimization

The last step in the *modeling* aspect of PSM is to take the global *Inference Graph* and optimize its local probabilistic model with the *Behavior Datasets* (6). The result is the final *Inference Graph* (7) with a global structure reflecting the program structure and local models representing types or executables.

Each cluster in the (cluster graph) IG will be a flow-based model i.e., a Non-Volume Preserving Transformation (NVP) [30]. An NVPs is a probabilistic model based on a neural network architecture. The NVPs are the local models for each cluster and are density estimators from which one may sample observation \mathbf{x} but also evaluate the likelihood $p_{\mathbf{x}}(\mathbf{x}; \boldsymbol{\theta})$. Each local model will learn the joint distribution of the compositional code elements, i.e., types or executables. For example, the local model for the **bmi**()-method in the Nutrition Advisor will model $p_{bmi}(\mathbf{bmi}; \boldsymbol{\theta}_{bmi})$. The variables of the local model are the observed variables \mathbf{x} and latent variables $f(\mathbf{x})$ where f is the flow. These variables \mathbf{x} are also present in the associated behavior dataset as the construction is based on the IG. The NVPs are then optimized iteratively via stochastic gradient descent and the procedures discussed in Section 3.2.4 and [30]. This optimization reduces the divergence between the target distribution $p_{\mathbf{x}}^*(\mathbf{x})$ and the flow-based model $p_{\mathbf{x}}(\mathbf{x}; \boldsymbol{\theta})$. The result of this process is an optimized NVP. The NVP can then be used to evaluate the likelihood of observations or to generate a new set of samples. Each cluster in the IG is then an NVP agreeing on the underlying data as it is sampled from the original program which naturally factorizes the behavior in clusters.

Optimization of Flow-based models

Each local model is a probabilistic model represented via a flow-based model. This specific instance of the flow-based model is in the context of PSM an NVP. However, any other model with similar properties can be used. Given a flow-based model $p_x(\mathbf{x}; \boldsymbol{\theta})$ that is fitted to a target distribution $p_x^*(\mathbf{x})$ one needs to minimize the divergence between both. The parameters of the flow $\boldsymbol{\theta} = \{\phi, \psi\}$ with ϕ being the parameters of the flow f and ψ of the latent distribution $p_z(\mathbf{Z})$. The flow is then optimized via the forward Kullback-Leibler (KL) divergence

$$\begin{aligned}\mathcal{L} &= D_{KL}[p_x^*(\mathbf{x}) \parallel p_x(\mathbf{x}; \boldsymbol{\theta})] \\ &= -\mathbb{E}_{p_x^*(\mathbf{x})} [\log p_x(\mathbf{x}; \boldsymbol{\theta})] + \text{const} \\ &= -\mathbb{E}_{p_x^*(\mathbf{x})} [\log p_z(f(\mathbf{x}; \phi); \psi) + \log |\det J_{f^{-1}}(\mathbf{x}; \phi)|] + \text{const}\end{aligned}$$

In the case of NVPs with a Gaussian latent space, update function of the parameters are then given by the closed-form likelihood of a Gaussian plus the Jacobian determinant

$$\begin{aligned}\nabla_{\phi} \mathcal{L}(\boldsymbol{\theta}) &= -\frac{1}{N} \sum_n \nabla_{\phi} \log p_z(f(\mathbf{x}_n; \phi); \psi) + \nabla_{\phi} \log |\det J_f(\mathbf{x}_n; \phi)| \\ \nabla_{\psi} \mathcal{L}(\boldsymbol{\theta}) &= -\frac{1}{N} \sum_n \nabla_{\psi} \log p_z(f(\mathbf{x}_n; \phi); \psi)\end{aligned}$$

∇_{ϕ} is the update of the parameters of the flow, i.e., the NVP. ∇_{ψ} is the update of the parameters of the latent distribution. However, in practice, the latent space is selected to be a unit Gaussian $\mathcal{N}(0, \mathbf{I})$ rendering this update unnecessary. The optimization stops once no improvement can be observed or when the predefined time budget is used. e.g., 100 epochs.

Intuitively, for each batch in the iterative process, one transforms the samples into the latent space \mathbf{z} and evaluates its likelihood. This likelihood, along with the Jacobian of the transformation, is then used to update the parameters of the neural network architecture. The Jacobian, in the case of NVPs, can be computed efficiently given its architecture [30]. A thorough discussion on flow-based models with all details is given by Papamakarios et al. [31].

In conclusion, the optimization of the local models, i.e., the NVPs, is an iterative stochastic gradient descent algorithm that uses the behavior datasets to optimize the neural network architecture. This optimization is done for each cluster in the IG. The result is the final IG with its global structure and local probabilistic models fitted to the program structure and behavior.

4.4 Inference

Inference is the fundament of all applications motivated in Section 1.2 and illustrated in Figure 4.1. The three tightly connected main aspects of inference are *sampling* (generation), *conditioning* (information propagation), *likelihood evaluation* (criticism). *Sampling*

draws observations from one (local) or multiple (global) nodes (NVPs) in the PSM network. This enables the probabilistic execution of e.g., an executable or a subsystem. *Conditioning* sets the models into a specific state. For example, Figure ?? illustrates the height property in its unconditioned and conditioned state. Local conditioning sets one node into a state. Global conditioning propagates a state across multiple clusters. *Likelihood Evaluation* quantifies samples in terms of their likelihood under a given cluster.

Figure 4.1 illustrates the combination of these aspects and combines them into causal forward (8) and backward (9) reasoning. Forward reasoning (8) (e.g., `Person.height` to `BmiService.bmi`) samples a conditional distribution and propagates it through the network to set downstream nodes into a conditioned state. Backward reasoning (9) starts at a conditioned downstream node and searches for the most likely cause. At every step, it is possible to draw conditional or unconditional samples. The directional aspect (forward and backward) is based on the program's call dependencies. However, the IG itself is an omnidirectional cluster graph.

Marginal inference finds the marginal probabilities of one or multiple variables in the absence of a condition. Conditional inference finds the probabilities of one or multiple variables in the presence of a condition imposed by other variables. This inference can happen locally and globally, i.e., within or across clusters.

4.4.1 Properties of the Inference Graph

The IG has some advantageous properties for inference: the running intersection property and marginal completeness.

Running Intersection Property

Definition 3. A cluster graph \mathcal{C} satisfies the running intersection property if, whenever there is a variable X such that $X \in C_i$ and $X \in C_j$, then there is a single path between C_i and C_j for which $X \in S_e$ for all edges e in the path. [23].

The property states that every cluster containing variable X spans a unique tree. Hence, there is only a single path in the cluster graph containing *direct* information about X . This property holds for *parameters* and *results* but only partially for *properties*. Types and executables are excluded from the definition as they are compositional code elements in the SG and clusters in the IG. Properties are global variables shared across multiple clusters. Any cluster may access the information stored in them. The implication of the *running intersection property* is important when doing global inference in the IG as it guarantees that information only flows along one path. In addition, properties might store parts of the information while traveling along the path.

The proof that the running intersection property is satisfied is best done by looking at the atomic variables one at a time. A *parameter* can only exist in the cluster C_i that represents the method that defines the parameter *and* a calling method C_j . Hence, there exists only a single path containing exactly one edge e connecting them satisfying the running intersection property. A similar proof can be constructed for a method *result*. The proof for *properties* only holds in the case where X consists of a single property.

A *property* can only exist in the cluster C_0 that represents the type that defines the property *and* any method C accessing it. Any set between C_0 and C_i will contain the property variable connecting both. However, there exists no edge D_C connecting any of C as a method cannot propagate a property but only parameters. Hence, the only valid path between $C_i \in C$ and $C_k \in C$ is over C_0 . From a program's viewpoint, a property can only be dereferenced within a method. Providing the dereferenced values to a calling method would result in a parameter.

Marginal Completeness

Definition 4. *An inference graph is marginal complete if, there exists exactly one cluster C for each variable $X \in \mathcal{X}$ that defines its marginal behavior $p(X)$.*

This property defines that the IG contains exactly one cluster that explicitly captures the marginal behavior of a given variable. This property holds for *types*, *properties*, *executables*, *parameters*, and *results*.

The proof that the IG is marginal complete is again best done by inspecting each type of variable. The IG is a reflection of the SG and adheres to its structural constraints. Any code element within a program may only be defined once with a given canonical name (or namespace). This means that the SG does not contain two code elements representing the same program definition. This means that the IG has exactly one defining cluster (or the cluster itself) that represents the code element. The defining cluster contains the entire marginal behavior of the code element and its associated variable.

Intuitively, a program (Java semantics) may not contain a code element, e.g., a property, with the same canonical name. A property may only be defined once in a program. This holds for all major programming languages to the best of our knowledge. This means that the property captures the entire behavior of itself, while consuming code elements may only capture parts of the behavior. The same holds for the IG as it is a reflection of the SG. This implies that we may retrieve the marginal probability of any variable by consulting its defining cluster, e.g., the type cluster associated with the type declaring the property. This also holds for joint probabilities defined in the same cluster e.g., the parameter list of an executable. An exception is given for code elements at the MU boundary that are included for the sake of compatibility. For example, parameters that are provided to an executable that resides outside the MU will have no defining cluster in the IG. Hence, only the conditional probabilities relative to the calling executable will be available.

4.4.2 Local Marginal Inference

Local marginal inference finds the marginal probabilities of one or multiple variables within a cluster in the IG. For example, evaluating the marginal distribution of the `bmi()` method with its cluster C_{bmi} and the joint probability $p(\text{height}, \text{weight}, \text{bmi})$.

Marginal inference within a cluster means marginal inference in the NVP. This is done by sampling from the latent space and transforming it via the flow into observed variables.

More formally, let \mathbf{z} be a latent vector with the probability distribution $p_z(\mathbf{z})$ of a cluster C represented via a flow f . Then, the marginal local inference of $p_x(\mathbf{x})$ is given by

$$p_x(\mathbf{x}) = p_z(f^{-1}(\mathbf{x})) \left| \det J_{f^{-1}}(\mathbf{x}) \right|.$$

Intuitively, we transform the observed sample \mathbf{x} into its latent representation \mathbf{z} and evaluate its probability by considering the change in volume via the Jacobian determinant. Inversely, to sample a marginal sample from the flow one can

$$\mathbf{x} = f(\mathbf{z})$$

where $\mathbf{z} \sim p_z(\mathbf{z})$. First, we sample from the Gaussian latent space and then transform it back into the observed space. Local marginal sampling with NVPs will always produce the full joint distribution of all variables in the scope of the cluster.

In conclusion, local marginal inference with NVPs transforms the latent space joint probability distribution into the observed space by applying the flow.

4.4.3 Local Conditional Inference

Conditional inference finds the probabilities of one or multiple variables in the presence of a condition (or a constraint) imposed on other variables. Conditional inference effectively allows one to ask questions to a given cluster. For example, $p_{bmi}(\text{height}, \text{weight} \mid bmi = 23)$ is the conditional joint probability of the `height` and `weight` given that we fix the result of `bmi` to 23.

Conditional inference within a cluster uses the capabilities of NVPs to condition the clusters. Given is a set of variables $\boldsymbol{\eta} \in C$, a condition given in form of an observation \mathbf{x} , and a *trained* flow f (i.e., a NVP) $q(\boldsymbol{\eta}; \boldsymbol{\phi})$. We can then use variational inference to approximate the posterior via

$$p(\boldsymbol{\eta} \mid \mathbf{x}) \approx q(\boldsymbol{\eta}; \boldsymbol{\phi}) = q_z(\mathbf{z}) \left| \det J_f(\mathbf{z}; \boldsymbol{\phi}) \right|^{-1},$$

where $q_z(\mathbf{z})$ is the latent distribution (or base distribution) and $f(\cdot; \boldsymbol{\phi})$ is the flow with its parameters $\boldsymbol{\phi}$. The posterior is then found by maximizing the *evidence lower bound* (ELBO)

$$\begin{aligned} \log p(\mathbf{x}) &\geq \mathbb{E}_{q(\boldsymbol{\eta}; \boldsymbol{\phi})} [\log p(\mathbf{x}, \boldsymbol{\eta})] - \mathbb{E}_{q(\boldsymbol{\eta}; \boldsymbol{\phi})} [\log q(\boldsymbol{\eta}; \boldsymbol{\phi})] \\ &= \mathbb{E}_{q_z(\mathbf{z})} [\log p(\mathbf{x}, f(\mathbf{z}; \boldsymbol{\phi}))] - \mathbb{E}_{q_z(\mathbf{z})} [\log q_z(\mathbf{z})] + \mathbb{E}_{q_z(\mathbf{z})} [\log |\det J_f(\mathbf{z}; \boldsymbol{\phi})|] \\ &= \mathbb{E}_{q_z(\mathbf{z})} [\log p(\mathbf{x}, f(\mathbf{z}; \boldsymbol{\phi}))] + \mathbb{H}_{q_z(\mathbf{z})} [\log q_z(\mathbf{z})] + \mathbb{E}_{q_z(\mathbf{z})} [\log |\det J_f(\mathbf{z}; \boldsymbol{\phi})|], \end{aligned}$$

where $\mathbb{H}[q_z(\mathbf{z})]$ is the differential entropy of the base distribution, which is constant relative to $\boldsymbol{\phi}$ [31]. The expectations can be estimated by Monte Carlo using multiple samples from the base distribution and averaging the results

$$\begin{aligned} \mathbb{E}_{q_z(\mathbf{z})} [\log p(\mathbf{x}, f(\mathbf{z}; \boldsymbol{\phi}))] &\approx \frac{1}{N} \sum_{n=1}^N \log p(\mathbf{x}, f(\hat{\mathbf{z}}_n; \boldsymbol{\phi})) \\ \mathbb{E}_{q_z(\mathbf{z})} [\log |\det J_f(\mathbf{z}; \boldsymbol{\phi})|] &\approx \frac{1}{N} \sum_{n=1}^N \mathbb{E}_{q_z(\mathbf{z})} [\log |\det J_f(\hat{\mathbf{z}}_n; \boldsymbol{\phi})|]. \end{aligned}$$

However, variational inference is only one possibility to compute conditional probabilities with NVPs. Papamakarios et al. [31] present alternative methods and additional detail into the discussion.

Intuitively, we fix the parameters of the flow and optimize the parameters of the random variables in the latent space via stochastic gradient descent given a set of observations. Once converged, we have a set of latent space parameters (or configurations), that approximate, once transformed via f^{-1} , the original observations best. Important is that the condition can contain only parts of \mathbf{x} as the remaining parts may be inferred during the process (imputed).

In conclusion, local conditional inference finds the best latent parameters that approximate its observed representation via variational inference. This is an iterative approach and may impute missing dimensions during the process.

4.4.4 Global Marginal Inference

Global marginal inference finds the marginal probabilities of one or multiple variables in the IG. For example, $p_{bmi}(\text{height}, \text{weight})$ represents the joint probability of the **height** and **weight** parameter in the **bmi**-method. The IG is *marginal complete* as described in Section 4.4.1. This means finding the marginal probabilities of one variable reduces to evaluating the probability of its defining cluster. The same holds for the joint marginal probabilities of multiple variables defined in one cluster.

To find the joint marginal probabilities for variables defined in different clusters one needs to calibrate the probabilities between the clusters containing the variables. Calibrating a cluster graph is done via the *sum-product belief propagation* algorithm. Calibration describes the process of finding a belief (a normalized or unnormalized probability) between two clusters such that both agree on its extent. A cluster graph is then calibrated if for each edge $(i - j)$ connecting cluster C_i and C_j we have

$$\sum_{C_i - S_{i,j}} \beta_i = \sum_{C_j - S_{i,j}} \beta_j$$

where β is the belief. That is, any two connected clusters agree on the marginal of variables in their sepset. Algorithm 2 [23] is an adaptation of Algorithm 1 for graphs in which we propagate beliefs in the graph until convergence. The algorithm will result in a calibrated graph that provides the joint marginal probabilities of variables. Important is that properties act as a global store of information, just like in the underlying programs. They do not adhere to the running intersection property and are received and updated in any cluster that uses them during the algorithm.

While calibrating the IG is possible, most program-related use-cases may not be as useful as someone thinks. An example use-case might be a program `foo1(a, b)` and `foo2(c, d)` calling `bar(e)`. Evaluating the joint marginal of $p(a, d)$ may be possible, but unlikely to be of great importance for programmers.

Algorithm 2: Calibration using sum-product belief propagation in a cluster graph [23]

Function CGraph-SP-Calibrate(Φ, \mathcal{C}):

```

  Initialize-CGraph
  while graph is not calibrated do
    Select  $(i - j) \in \mathbf{D}$ 
     $\delta_{i \rightarrow j}(\mathbf{S}_{i,j}) \leftarrow \text{SP-Message}(i, j)$ 
  for each cluster  $i$  do
     $\beta_i \leftarrow \psi_i \cdot \prod_{k \in Nb_i} \delta_{k \rightarrow i}$ 
  return  $\{\beta_i\}$ 

```

Function Initialize-CGraph():

```

   $\mathcal{C}$ 

```

Function SP-Message(i, j):

```

  // i ... idx sending cluster
  // j ... idx receiving cluster
   $\psi(\mathbf{C}_i) \leftarrow \psi_i \cdot \prod_{k \in (Nb_i - j)} \delta_{k \rightarrow i}$ 
   $\tau(\mathbf{S}_{i,j}) \leftarrow \sum_{\mathbf{C}_i - \mathbf{S}_{i,j}} \psi(\mathbf{C}_i)$ 
  return  $\tau(\mathbf{S}_{i,j})$ 

```

In conclusion, the IG is a marginal complete cluster graph in which each variable has a defining cluster containing its marginal behavior. Finding the marginal probabilities for a given variable reduces to finding the cluster that represents the code element that defines the variable in the original program. To find the marginal probabilities for variables with different defining clusters we may apply the sum-product belief propagation algorithm.

4.4.5 Global Conditional Inference

Global conditional inference finds the conditional probabilities of one or multiple variables in the context of the IG. Global conditional inference allows us to answer questions across multiple code elements in a program. For example, evaluating $p_{bmi}(\mathbf{x})$ given that the `handle`-method only receives requests from females $p_{handle}(\mathbf{x} \mid \text{gender} = \text{female})$. Inference in traditional cluster graphs is done via the *loopy belief propagation* algorithm. However, many use cases in the IG will condition and propagate along a path down or up the call graph of the program. In these cases, it suffices to use the simpler *sum-product* algorithm in Algorithm 1 if the inference path is a tree.

In the general case, let there be a condition \mathbf{c} on cluster C_i and target variables $\boldsymbol{\eta} \in C_j$ where $C_i \neq C_j$. Furthermore, there exists a path $(i - j)$ connecting cluster C_i and C_j . To compute the global conditional probability we apply Algorithm 2 while initializing and fixing the conditioned cluster to the constraint \mathbf{c} . In the case where the path forms a tree, we may apply Algorithm 1 while initializing and fixing the conditioned cluster to the constraint \mathbf{c} . The local clusters are conditioned according to Section 4.4.3.

Intuitively, global conditional inference is the same as marginal inference while fixing the conditioned variables to their constraint. The constraint itself acts like an unchangeable fact that we *observed*. Any probabilistic relationship to the observed variable will adapt accordingly to the observation and propagate the information through the network.

In conclusion, global conditional inference is global marginal inference while fixing variables to a specific *observed* fact or constraint.

4.5 Related Work

To position PSM, it is useful to distinguish between *programming paradigms* and *software analysis methods*. A programming paradigm is a collection of programming languages that share common traits (e.g., object-oriented, logical, or functional programming). Analysis methods extract information from programs (e.g., design pattern detection, clone detection). PSM is an *analysis* method that analyzes a program given in an object-oriented programming language and *synthesizes* a probabilistic model from it.

Probabilistic programming is a programming paradigm in which probabilistic models are specified. Developers describe probabilistic programs in a domain-specific language (e.g., BUGS [37]) or via a library in a host language (e.g., Pyro [?], PyMC [27], Edward [38]). In contrast, PSM analyzes a program written in a traditional programming language and translates it into a probabilistic program. This difference also holds for modeling concepts like *Bayesian Networks* [23] or *Object-Oriented Bayesian Networks* [39, 40] that can be implemented via a probabilistic programming language.

Formal methods are a programming paradigm that leverages logic as a programming language (e.g., TLA+ [41] or Alloy [42]). *Stochastic model checking* [43] introduces uncertainty in the rigid formalism to model, e.g., natural phenomena. Developers specify the behavior and provide the state transition probabilities in a special-purpose language (e.g., PRISM [44], PAT [45], CADP [46]). Furthermore, extracts the state transition probabilities from runtime traces. This may be more imprecise than an expert that provides the probabilities. However, it opens up PSM to a substantially larger set of programs and problems that can be applied since expert knowledge might not always be available. Furthermore, PSM analyzes a program and synthesizes a PM from it. This means that there is no additional education of the developers needed in order to apply the methodology.

Symbolic execution [47] is an analysis method that executes a program with symbols rather than concrete values (e.g., JPF-SE [48], KLEE [49], Pex [50]). It can be used to determine which input values cause specific branching points (if-else branches) in a program. *Probabilistic symbolic execution* [51] is an extension that quantifies the execution, e.g., branching points, in terms of probabilities. This is useful for applications that quantify program changes [52] or performance [53]. Probabilistic symbolic execution operates on the statement level while PSM abstracts statements capturing, e.g., inputs and outputs of methods. This abstraction makes PSM computationally scalable while symbolic execution suffers from potential path explosions. Furthermore, this abstraction shifts the analysis focus to the program semantics compared to the statement semantics.

An analysis on statement level seeks answers to program properties that occur at, e.g., a specific branching point. In contrast, an analysis on the program level seeks answers to program properties that occur, e.g., between two methods. Both approaches are complementary to each other and extend the possible range of analyzes that can be conducted.

Probabilistic debugging [54, 55] is an analysis method that supports developers in debugging sessions. The debugger assigns probabilities to each statement and updates them according to the most likely erroneous statement. In contrast to PSM, most probabilistic debuggers operate on a statement level. Another difference is given in the life-cycle of the methodologies. Debugging has an operational life-cycle only valid until the bug is found. PSM and the resulting models are intended to be persisted along with the matching source code revision. This allows, e.g., method-level fault localization, by comparing multiple revisions of the same model.

Invariant detectors [14, 56, 57, 58, 59, 60] learn assertions and add them to the source code. This helps to pinpoint erroneous regions in the source code. These assertions are found by observing the runtime of a program and learning rules that bound the value region of statements and variables. For example, adding a precondition assertion to a method by asserting the minimal observable value of an integer parameter. PSM in contrast learns the full joint distribution of inputs and outputs of executables instead of *thresholds* that partition the data into regions. This allows PSM to generate complex assertions that include multiple variables and conditions on them.

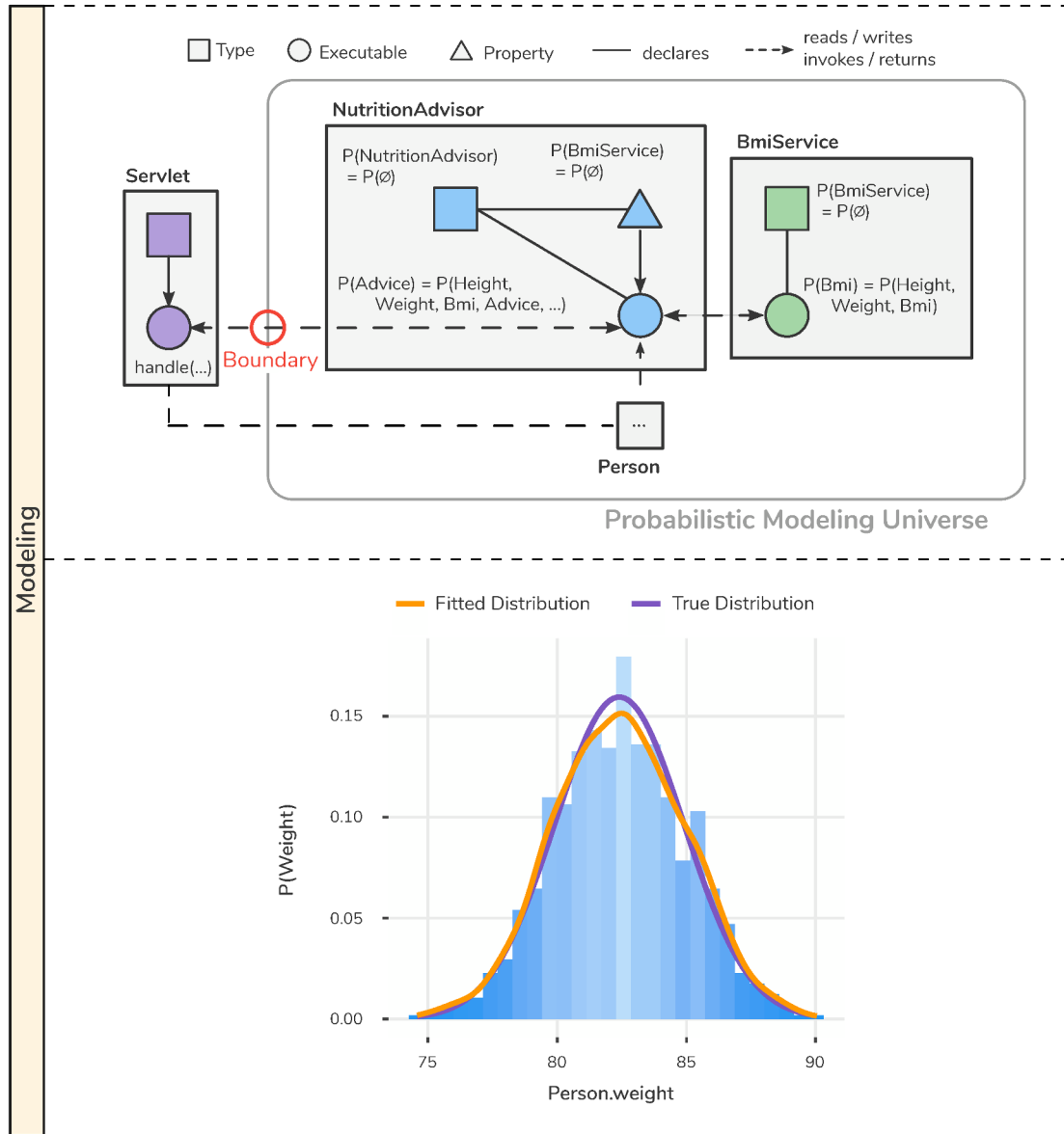


Figure 4.2: The top shows an abstract representation of the IG of the Nutrition Advisor. The bottom shows one dimension of the behavior dataset (histogram), the estimated density, and the true distribution.

5 Case Study: Modeling Feasibility

In the previous Chapter 4, we introduced PSM and its theoretical fundament. This chapter will evaluate the method’s basic feasibility by applying the methods to a range of different projects. More concretely, a PSM prototype will extract structure graph, behavior graph, and inference graph from several projects. Then we will evaluate each graph on its own for its essential properties. The evaluation of PSM applications as described in Section 1.2 are beyond the scope of this chapter. However, Chapters 7 and 8 present the implementation of two PSM applications in great detail and naturally extend this chapter.

5.1 Study

The core hypothesis of PSM is that programs can be transformed into a probabilistic model, i.e., the inference graph. This study (i.e., the prototype, research questions, analyses, and discussions) focuses on evaluating the core PSM methodologies presented in Chapter 4. Specifically, the study answers the following questions, providing evidence for the core hypothesis:

- RQ1 [**Code**] Are projects exposing enough code elements that are eligible for PSM?
- RQ2 [**Runtime**] Are code elements creating enough runtime data with which the model parameters can be optimized?
- RQ3 [**Modeling**] Are probabilistic models capable of capturing the runtime data of eligible code elements?
- RQ4 [**Inference**] Is information propagation consistent and stable in the inference graph?

RQ1 addresses the precondition of whether projects expose enough data (i.e., number or text) code elements that can be modeled. RQ2 addresses the precondition of whether these (data) code elements create a sufficient amount of runtime data that can be modeled. RQ3 addresses the central question of whether the behavior of a program in the form of its runtime data can be approximated via models. Finally, RQ4 evaluates the usefulness of the approach and whether PSM is a sound basis for the applications presented in Section 1.2. The four questions are scoped by structured programs that can be executed and support runtime monitoring. The empirical evidence in this chapter is essential for any future endeavor regarding PSM.

Table 5.1: Hyper-parameters used in the experiments.

#	Stage	Name	Values
1	Data	Size	20 to 10 000
2	Data	Test Split	10%
3	Preprocessing	Number	Standardization
4	Preprocessing	Discretization Threshold	16
5	Preprocessing	Discretization Encoding	Base 10
6	Preprocessing	Text Encoding	Base 10
7	Optimizer	Algorithm	Adam [61]
8	Optimizer	Learning Rate	5×10^{-4}
9	Optimizer	Weight Decay [62]	5×10^{-2}
10	Optimizer	Batch Size	full dataset
11	Optimizer	Max Epoch	1000
12	Optimizer	Early Stopping Patience	20 epochs
13	NVP [30]	Coupling Count	6
14	Coupling Layer [30]	Linear Layer Count	2
15	Coupling Layer [30]	Hidden Units Count	32 (low) 128 (high)
16	Coupling Layer [30]	Latent-Space	$\mathcal{N}(0, 1)$
17	Coupling Layer [30]	Translation Activations	Gelu [63]
18	Coupling Layer [30]	Scale Activations	Gelu [63], Tanh

5.1.1 Setup

We implemented a prototype called Gradient that reflects the process and workflow presented in Figure 4.1.

1. The input *Source Code* were open-source subject systems written in Java (see next Section 5.1.2).
2. The *Structure Graph* was extracted using Spoon [34] and via methods presenting in Section 4.1.
3. AspectJ 1.9.1 was used to weave monitoring aspects (tracing) into the subject systems to capture their *Runtime Behavior* in the modeling universe.
4. The *Inference Graph* was created as discussed in Section 4.3 for each code element. The generated IG was truncated, i.e., no type models were constructed. The shape and size of the NVPs is given in Table 5.1.
5. The *Behavior Datasets* were created as presented in Section 4.2, by tallying the event stream. This includes splitting the dataset into training and evaluation partitions and preprocessing them. Preprocessing consisted of encoding text features by enumerating (starting from 0) and encoding them in a base 10 vector space.

Table 5.2: Overview of the projects used in the study. LoC are the lines of code in a project.

Project	Version	#Files	#LoC	Type	Property			
					Data	Ref	Unk	Total
Nutrition Advisor	0.1.0	5	154	5	11	3	1	15
Structurizr	1.0.0	115	9941	123	229	85	24	338
jLatexmath	1.0.7	156	21 369	191	490	121	81	692
PMD	6.5.0	799	89 349	981	1858	503	481	2842
		1075	120 813	1300	2588	712	587	3887

Data = {Number, Text}, Ref = Reference, Unk = Unknown

Table 5.3: Overview of the projects used in the study. LoC are the lines of code in a project.

Project	Parameter				Executable				
	Data	Ref	Unk	Total	Data	Ref	Void	Unk	Total
Nutrition Advisor	19	1	0	20	10	0	19	1	30
Structurizr	725	342	26	1093	320	302	508	20	1150
jLatexmath	1115	556	153	1824	269	416	511	59	1255
PMD	2933	2910	1943	7786	3222	719	3445	2073	9459
	4792	3809	2122	10 723	3821	1437	4483	2153	11 894

Data = {Number, Text}, Ref = Reference, Unk = Unknown

Number dimensions were considered discrete if less or equal than 16 values were found and underwent the same base 10 encoding procedure. Finally, all non-encoded dimensions were standardized to have a mean of zero and a standard deviation of 1.

6. Model parameters were optimized with their datasets, and the best parameter setting was retained (w.r.t. evaluation performance).

7. Finally, the persisted models were used in the analysis scenarios (see Section 5.1.5).

Hyper-parameters of the experiments are given in Table 5.1. The chosen values are based on additional non-reported experiments evaluated on a synthetic dataset. All experiments were executed on a single machine (Intel i7, Nvidia GTX 970).

5.1.2 Subject Systems

The study uses four subject systems listed in Tables 5.2 and 5.3. *Nutrition Advisor* is the running example introduced in Chapter 2. *Structurizr* [64] is a developer-focused

software architecture visualization tool. *jLatexmath* [65] is a library for rendering LaTeX formulas. *PMD* [66] is a static code analysis tool for Java applications.

All code elements of the projects were included in the modeling universe (excluding inherited third-party elements). *Nutrition Advisor* received 1000 advice requests as a trigger with data based on the NHANES r[20] dataset. *jLatexmath* and *Structurizr* were executed with examples provided in their documentation. *PMD* received the *Nutrition Advisor* as input program.

5.1.3 Controlled Variables

The study controls for one variable: *Capacity*.

- **Capacity:** The capacity describes the number (*low* = 32, *high* = 128) of units in the linear layers of the NVPs.

5.1.4 Response Variables

The response is split into quantitative and qualitative parts. The quantitative part evaluates the *Events per Code Element (ECE)*, *Distinct Values per Code Element (DCE)*, and *Negative Log-Likelihood (NLL)*. The qualitative part assesses the visual fidelity of the samples generated by the model compared to the original dataset. Furthermore, it evaluates the usefulness of the PSM network via a scenario-based evaluation given in Section 5.1.5.

- **Events per Code Element (ECE):** ECE measures the number of events emitted by code elements. This provides insights into the runtime activity of elements and how many models need to be fitted. We report ECE1 and ECE10 to distinguish between dependencies/constants and real behavior carrying code elements. ECE1 includes all code elements with at least one event (all active code elements at runtime). ECE10 includes only code elements that emitted at least 10 events at runtime.
- **Distinct Values per Code Element (DCE):** DCE measures the number of distinct values emitted by code elements. This provides insights into the capacity requirements of the models. We report DCE1 and DCE10 where DCE10 includes code elements with at least 10 distinct values.
- **Average Negative Log-Likelihood (NLL):** Measures the average Negative Log-Likelihood (Equation 3.4) of data points under the model in natural units of information (nats; lower is better).

5.1.5 Experiment Results

The study results are split into four groups: *Code*, *Runtime*, *Modeling*, and *Inference*.

Table 5.4: Events are the number of events observed at runtime. ACT10 are the number of events observed at runtime on code elements with at least 10 events. DCT10 are the number of distinct values on code elements with at least 10 distinct values.

Project	Data Type	Events				ACT10				DCT10			
		Mdn	Q1	Q3	Total	Mdn	Q1	Q3	Total	Mdn	Q1	Q3	Total
Nutrition Advisor	Data	1000	1000	1000	21 000	1000	1000	1000	21 000	524	363	824	8040
	Others	1000	252	1001	9008	1001	1000	1501	9002	1000	1000	1000	1000
Structurizr	Data	6	2	17	35 852	25	16	67	35 041	21	13	46	2514
	Others	12	3	36	58 489	34	17	104	57 607	29	16	59	3331
jLatexmath	Data	130	15	526	6 415 336	274	61	1297	6 414 919	39	18	81	24 495
	Others	66	6	530	1 377 280	257	56	1064	1 376 553	107	30	408	42 592
PMD	Data	35	5	154	15 069 591	117	37	267	15 068 209	39	18	91	24 511
	Others	18	5	117	1 882 176	64	20	185	1 879 058	30	16	123	69 569
		21	5	138	24 868 732	83	25	306	24 861 389	39	17	102	176 052

Mdn = Median, Q1/3 = Quartile

Data = {Number, Text}, Others = {Reference, Unknown}

Code

The projects contained a total of 27 804 property, parameter, and executable code elements. PMD is the largest project containing 76 % of the total code elements. Nutrition Advisor is the smallest project containing 0.25 %. Most elements were executables (43 %) or parameters (39 %). 42 % of the elements were *data* elements, i.e., had either a number or text type that is eligible for PSM modeling. 22 % were references within the modeling universe and the remaining 36 % were elements of unknown type that were not within the modeling universe. Tables 5.2 and 5.3 shows detailed results per subject system, element type, and data type.

Runtime

Monitoring sessions lasted for a median duration of 136.55 s (IQR = 3.27 to 369.35) and were concurrently executed with the modeling sessions of other projects. The median processing speed was 25 101 events per second (IQR = 24 727 to 26 283).

During the monitoring session, a total of 24 868 732 events were emitted from 6002 code elements (22 % of total code elements). 36 % of the 6002 code elements emitted *data* (text or number) events. 68 % were generated by the PMD project, while the least events were generated by the Nutrition Advisor 0.12 %. 87 % of the events were *data* (text or number) events while the remaining 13 % were either *reference* or *unknown* events.

The *event analysis* shows that most of the events (24 861 389) occurred on 3868 (14 % of total) code elements. This excludes elements that emitted less than 10 events (ECE10). 36 % of the 3868 code elements generated *data* (text or number) events. Percentages for the largest and smallest, as for the data types match those of the events. Differences are given in Table 5.4 in terms of the central tendencies.

The *distinct value analysis* shows that a total of 176 052 distinct values were generated by 914 code elements (3.29 %). This excludes elements that emitted less than 10 events

Table 5.5: Model analysis results split across projects, and capacity. Lower is better for NLL results.

Capacity	Project	Models	Data Points				Dimensions			
			Mdn	Q1	Q3	Total	Mdn	Q1	Q3	Total
Low	Nutrition Advisor	4	1000	1000	1000	4000	6	5	8	27
	Structurizr	50	67	31	137	14 715	3	3	4	179
	jLatexmath	146	393	82	1248	206 820	4	3	7	763
	PMD	574	133	56	337	454 545	4	3	5	2511
Low		774	151	56	472	680 080	4	3	5	3480
High										

Mdn = Median, Q1/3 = Quartile

Table 5.6: Model analysis results split across projects, and capacity. Lower is better for NLL results.

Capacity	Project	Training NLL				Test NLL			
		Mdn	Q1	Q3	Total	Mdn	Q1	Q3	Total
Low	Nutrition Advisor	-1.37	-4.40	1.92	-4.44	-1.61	-4.51	1.69	-4.80
	Structurizr	-0.83	-2.77	1.75	-48.86	-0.93	-2.95	2.08	-39.27
	jLatexmath	-3.10	-7.64	1.06	-617.12	-3.10	-7.91	1.29	-598.81
	PMD	-3.96	-6.84	-3.15	-3080.96	-3.96	-6.69	-2.94	-3034.99
Low		-3.95	-6.67	-1.96	-3751.38	-3.95	-6.58	-1.96	-3677.88
High		-3.95	-7.22	-2.03	-3985.55	-3.99	-7.30	-1.99	-3946.18

Mdn = Median, Q1/3 = Quartile

(DCE10). 44 % of the 914 code elements generated *data* events. Most of the distinct values come from the PMD project that makes up 53 %. The least distinct values were generated by the Structurizr with 3.32 %. Distinct values related to *Data* were encountered 34 % while others were encountered 66 % of the time.

Modeling

Tables 5.5 and 5.6 contains the detailed results of the low capacity setting and the margins for the high capacity setting. The total wall time to optimize the parameters of all models was 195 min (111 min for high capacity). The median time needed to optimize one model in the low capacity setting was Mdn = 72.42, IQR = 55.21 to 93.16 (Mdn = 38.60, IQR = 29.11 to 50.16 for high capacity).

A total of 774 models were fitted. PMD accounted for 74 % of the models. In sum, 680 080 data points were used in the process where Nutrition Advisor had the most data points available per model (1000). A total of 3480 dimensions exist across all models where PMD accounts for 72 % of all dimensions. However, the Nutrition Advisor models

had the highest amount of dimensions per model. 62 % of the dimensions were related to continuous features and the remainder to discrete features. A total of 12 787 800 parameters were used (Mdn = 15 780, IQR = 15 000 to 16 560) in the low capacity setting for the models. The high capacity setting had a total of 165 172 056 parameters (Mdn = 210 468, IQR = 207 384 to 213 552). Finally, all projects yielded a total test NLL of -3677.88 (low capacity). On average, the models found in the PMD project had the best NLL with -3.96 and the worst in the Structurizr -0.93 . No significant divergence between training and test NLL can be seen.

The qualitative inspection of the models revealed a good approximation with two caveats. First, an imprecise density estimate is given for categorical dimensions that include high mass levels. The high mass levels cause an increase of mass in the surrounding levels compared to the original data. Proximity in categorical data is introduced by the 10-ary encoding and the continuous nature of NVPs. Second, an imprecise density estimate is given in continuous dimensions with disconnected high-density modes being connected. This issue occurs more frequently in the low capacity setting than in the high capacity setting indicating under-fitted models.

Inference

The qualitative assessment of the inference capabilities of PSM are split into two scenarios presented in Figure 5.1 and Figure 5.2. These scenarios extend the running example by adding the `Servlet` to the Modeling Universe.

The first scenario in Figure 5.1 shows a simulation in which the Nutrition Advisor is conditioned on women requests. The circles at the top illustrate the original call hierarchy and parts of the PSM network from Figure 5.1. Each node was fitted on the original data without any restrictions or conditions. The contour plots below show the *height* and *weight* variables in each model conditioned by gender (see Figure 5.2 for unconditional version). The density plots at the bottom present the *bmi* variable of the same respective model. In the background is the original unconditioned distribution (i.e., including males). **Only** the *handle*-model has direct access to the *gender* property. The original conditional information (i.e., *Person.gender = Female*) flows through the network by iteratively sampling n observations, propagating, and conditioning. This equals n (probabilistic) executions of the program. Finally, the right part in Figure 5.1 shows the degree of information degradation in a forward and backward inference setting with 10 round-trips (40 information hops). The centers and shapes are mostly preserved, but a minimal reduction of the variance can be seen. The density of the *bmi* variable was preserved over the 40 hops without substantial loss of information.

The second scenario in Figure 5.2 assumes that *Servlet* and *NutritionAdvisor* are developed by Company A while *BmiService* is developed by Company B specialized in ML. Company A uses the simple height/weight formula to stub the *BmiService* until Company B delivers its service based on a supervised machine learning model. Company A has a PSM model M_{null} of the system. Company A builds a second revision M_{alt} of its PSM model, including the new component they received from Company B (*BmiService*). The automated compatibility checks during continuous integration failed for code elements

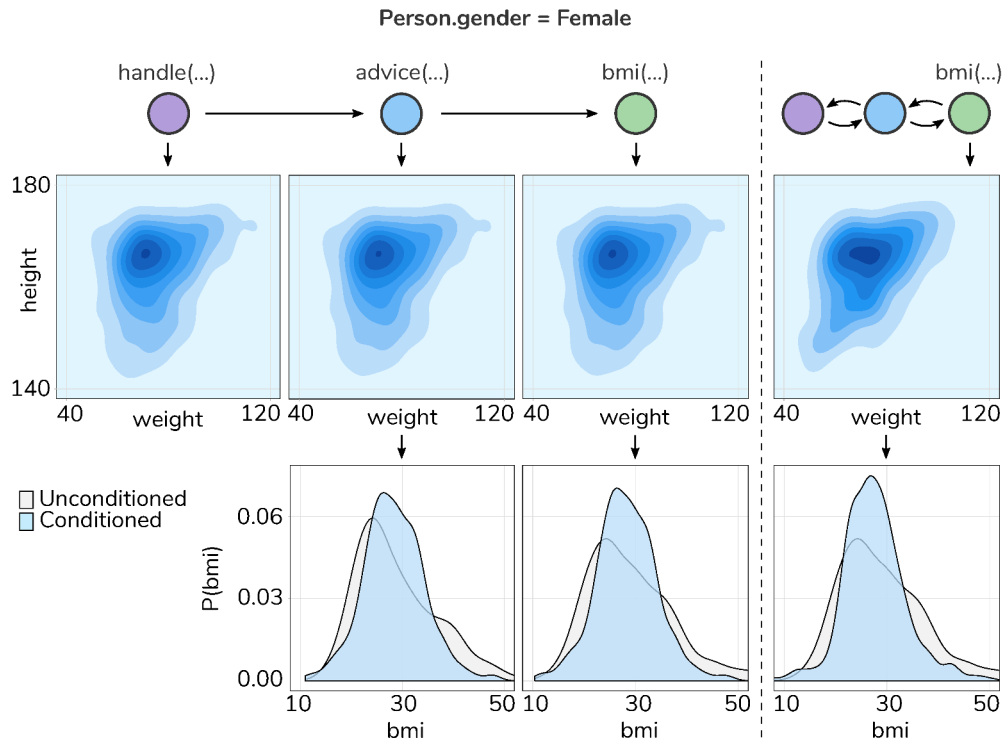


Figure 5.1: Shows an inference example with a condition caused by a latent variable starting at the handle-method. Gender, only accessible in the handle-method is conditioned to females. Height and weight are propagated while *bmi* jointly adapts to the condition. The last column shows a roundtrip of 10 (40 propagation hops) and its effect compared to the original distribution.

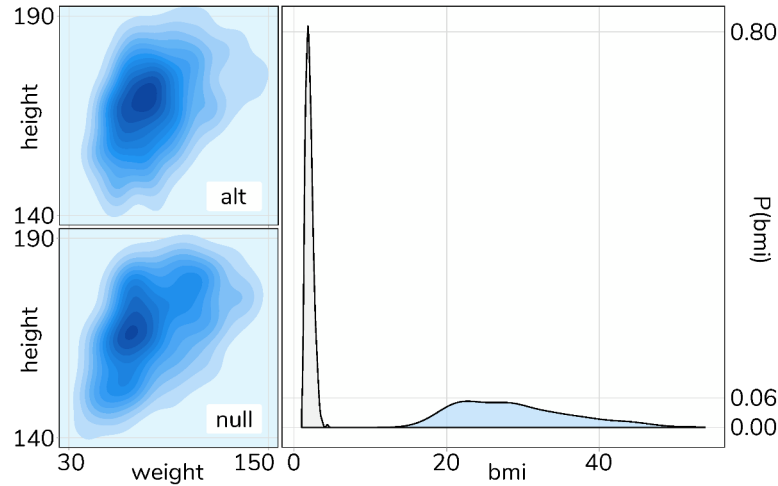


Figure 5.2: Shows an example for semantic testing and criticism where the *null*-model and *alt*-model come from different teams. The clear difference between the return values was detected automatically and works for traditional software and AI components.

in *bmi(...)* and *advice(...)*). However, the checks for the other elements were successful. Revisiting the call graph in reverse order reveals a semantic error in the new component illustrated in Figure 5.2. The inputs match (contour plots on the left) but the outputs diverge drastically (density plot on the right). The issue was that Company A uses the metric measurement system while Company B uses the imperial system.

The scenario is based on real data. However, the regression model was substituted by the simple imperial BMI formula. Compatibility checks were done via Kolmogorov-Smirnov Tests [67].

The remarkable aspect of this scenario is the ignorance of PSM regarding the true underlying implementation (code vs AI model). Traditional tests would need to draw the correct assertion values from the model. Not only are these tests flawed, but every update of the model's parameters would trigger cascading changes in the tests. In contrast, PSM tests the behavior of the functionality, not the code (semantic tests).

5.2 Discussion

The results presented in Section 5.1.5 provide direct or indirect evidence for the research questions in Section 5.1.

5.2.1 Code

The code analysis (see Section 5.1.5) shows that the total project size is secondary for PSM. Nearly half (43%) of the code elements in a project are eligible for modeling. The remaining elements are either referencing eligible code elements or are external

dependencies. This large proportion justifies the use of PSM for projects independent of their size (RQ1).

In conclusion, projects, independent of their size, expose enough code elements eligible for PSM.

5.2.2 Runtime

The results of the runtime analysis (see Section 5.1.5) show that most events are related to actual data (87%), providing evidence for RQ2 and support for PSM. These data events are emitted by a rather small portion of the active code elements (14%, ACT10). Regarding RQ3, this means that few models will capture most of a program's behavior. Most of the variability is generated by few code elements 3.29%. Nearly half of the variability is related to data (44%) while the other half are mainly object references. This means that the average capacity can be low (RQ3).

In conclusion, active code elements are creating enough data that can be used for PSM.

5.2.3 Modeling

The results of the modeling analysis (see Section 5.1.5) show that most models have few dimensions. This supports the use of low capacity models. The selected capacity does not hint at overfitting to specific portions of the data given that training and test NLL are not significantly different. The qualitative inspections revealed high-quality models with good approximations with two caveats (mass leakage and mode connectivity). The two issues are mainly related to the capacity of the model which is too high for discrete, but too low for continuous variables. One way to alleviate these issues would be an adaptive model class and parameter selection.

In conclusion, the qualitative and quantitative assessments suggest that probabilistic models can approximate the behavior of a program.

5.2.4 Inference

The inference analysis (see Section 5.1.5) evaluated the inference capabilities of PSM models by two illustrative scenarios. The first scenario (Figure 5.1) illustrated multi-dimensional information propagation with latent factors across multiple models. The second scenario (Figure 5.2) focused on model/data evaluation in a software development context in which software and AI components are integrated. The scenarios distill the foundations on which any PSM application (see Section 1.2) is built on: sampling (generation), conditioning (information propagation), and likelihood evaluation (criticism).

In conclusion, results show that local (within model) and global (between models) generation is sensitive and consistent to inference conditions.

5.3 Limitations

There are several limitations to the approach or the current prototype. The approach needs a structured program, and it must be observable at runtime. Large methods that handle multiple tasks will reduce the usefulness of PSM.

The current prototype is focused on data. References are handles to objects that might contain data or more references. PSM naturally dereferences these handles since models only contain, e.g., properties, that are accessed. This means that PSM is not useful for libraries whose sole purpose is reference management, e.g., a collection library.

The current prototype explodes lists into separate value assignments, i.e., a list of two elements acts as two assignments to a non-list variable. Hence, no order relationship is captured by the model. Extending PSM via sequential models may alleviate this limitation. However, the usefulness is subject to the actual application that is realized.

5.4 Threats to Validity

An external threat to validity is given by the number of projects used in the study. Rigorous internal evaluation and projects of various sizes and types minimize the threat. Different project sizes control for the expectation that large projects have better or worse models because of their perceived complexity. Different project types (e.g., PMD as system or jLatexmath as application software) control for the element type distribution and their runtime behavior (user vs. synthetic data). Finally, the evaluation models *all* eligible code elements and measures the variance across the projects. The NLL across projects in Tables 5.5 and 5.6 does not hint at a by-chance selection of projects that are simple to model.

5.5 Conclusion and Future Work

In this work, we presented Probabilistic Software Modeling (PSM), a data-driven approach for predictive and generative methods in software engineering.

We have discussed applications, pragmatics, construction details, and technical considerations of PSM. We evaluated the viability and usability of PSM on multiple projects and discussed scenarios that provide insights on how PSM is used. The results have shown that PSM is not only viable but naturally integrates with software 2.0, i.e., AI components.

Our future work will focus on the realization and evaluation of applications and their comparison to the current state of the art.

In conclusion, PSM analyses a program and synthesizes a probabilistic model that is capable of simulating and quantifying it. The resulting models are repeatable, persistable, shareable, and quantifiable representations of the program and act as a foundation from which solutions can be derived.

6 Case Study: Clones in PLC Programs

Abstract:

The reuse of code fragments by copying and pasting is widely practiced in software development and results in code clones. Cloning is considered an anti-pattern as it negatively affects program correctness and increases maintenance efforts. Programmable Logic Controller (PLC) software is no exception in the code clone discussion as reuse in development and maintenance is frequently achieved through copy, paste, and modification. Even though the presence of code clones may not necessarily be a problem per se, it is important to detect, track and manage clones as the software system evolves. Unfortunately, tool support for clone detection and management is not commonly available for PLC software systems or limited to generic tools with a reduced set of features. In this chapter, we investigate code clones in a real-world PLC software system based on IEC 61131-3 Structured Text and C/C++. We extended a widely used tool for clone detection with normalization support. Furthermore, we evaluated the different types and natures of code clones in the studied system and their relevance for refactoring. Results shed light on the applicability and usefulness of clone detection in the context of industrial automation systems, and it demonstrates the benefit of adapting detection and management tools for IEC 61131-3 languages. Furthermore, the results provide a preliminary context for semantic clone detection discussed in Chapter 7.

Authors:

H. Thaller, R. Ramler, J. Pichler, and A. Egyed

Parts of this Chapter was published at the 22nd IEEE International Conference on Emerging Technologies and Factory Automation (2017) [1].

6.1 Introduction

The increasing share of software for Programmable Logic Controllers (PLCs) and its practical importance have recently been acknowledged by the authors of the 2016 ranking of programming languages in IEEE Spectrum [68]. They found that languages for PLCs are on the rise, although in contrast to general-purpose languages such as C, C++ or Java, they specialize in a niche. Yet their relative popularity indicates just how big that niche really is [68]. With the growing importance of PLC software, an increasing demand for software engineering best practices and tool support is essential. The focus of this chapter is on detecting and analyzing code clones in PLC programs.

Code clones are source code fragments that have been duplicated for reuse, e.g., by copying and pasting [69]. Code clones have the reputation to negatively affect program correctness [70] and to increase maintenance efforts [69]. This form of reuse is widely considered an anti-pattern in software development [71] and clones are treated as a bad smell in code [72]. Recent studies have shown that there are various reasons why code clones are introduced and that the presence of clones is not per se a problem. However, the ability to detect, track and manage clones as the software system evolves is of the essence in successful software development [73].

PLC systems are no exception in the discussion of clones as reuse in the industrial automation domain is often achieved through cloning and modifying of existing software systems or sub-systems [74]. This is caused by technological restrictions introduced by programming languages such as the lack of inheritance or polymorphism, organizational limits like time constraints, or simply by the system's complexity. Furthermore, cloning is often used as a lightweight software product line strategy to cope with various hardware options and application environments.

A wide range of tools and techniques for clone detection and management is available for programming languages such as C, C++ or Java [75]. Similar support for IEC 61131-3 languages is mostly limited to general purpose tools, which lack analysis features that require the interpretation of the syntactical structure of the analyzed language. The contributions of this chapter are as follows:

- Quantitative results from a code clone analysis in a real-world PLC software system consisting of IEC 61131-3 Structured Text (ST) and C/C++.
- The extension of the widely used clone detector Simian [76] with language support for Structured Text.
- A comprehensive study on the relevance and natures of the found clones.
- An assessment of whether language support in clone detectors is of importance or not.

6.2 Background and Related Work

Duplicating code fragments during software development activities has a long history. Definitions, taxonomies, tools for detecting, analyzing, visualizing and managing code clones exist for several languages and technologies. The interest in code clones is also reflected in the wealth of existing research and the widespread use of tools and techniques in quality management and continuous integration.

Clone pairs and *clone classes* [69] are basic terms used in the context of clone detection. A clone pair describes two code blocks, also called fragments, that are equal according to a similarity operator. A clone class is the set of all blocks that are equal according to a similarity operator, effectively forming an equivalence class. Figure 6.1 shows an example for a clone pair formed by Block 1 and Block 2, where the similarity operator

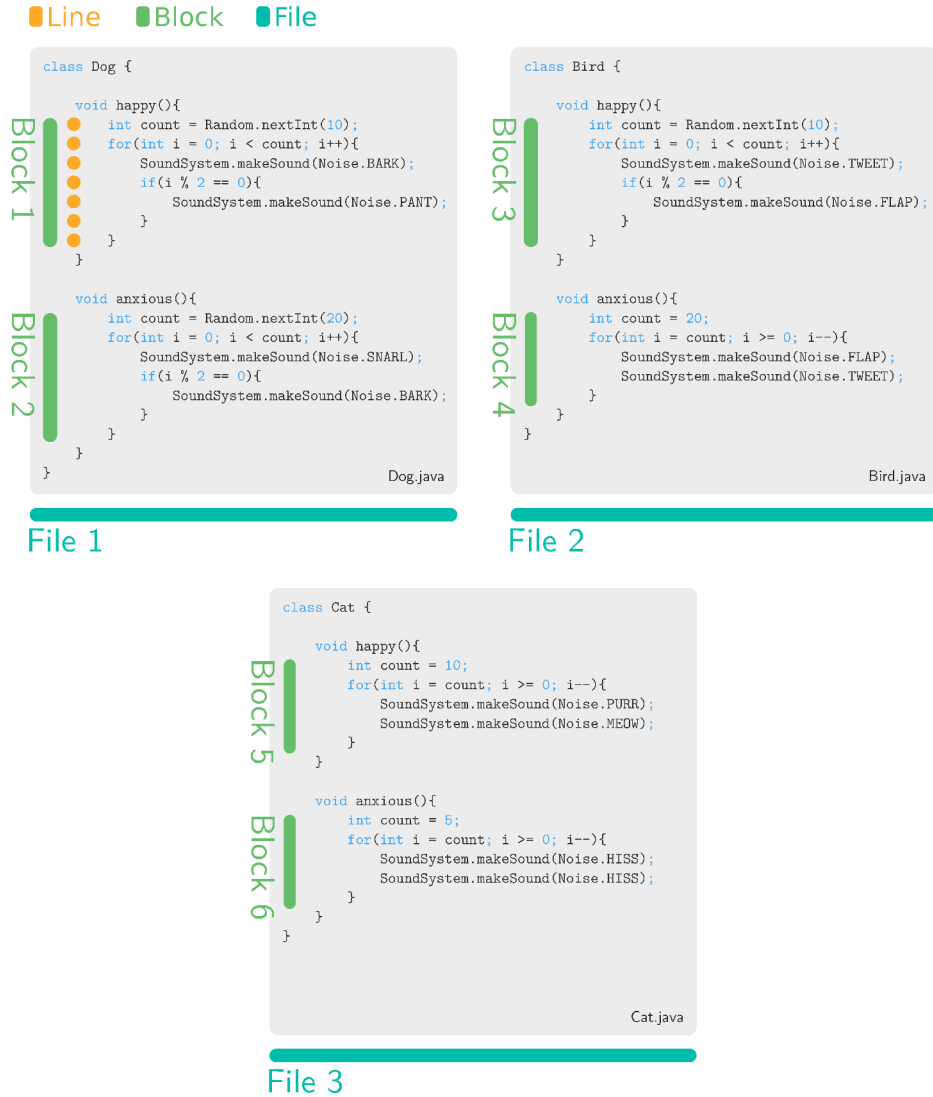


Figure 6.1: Clones may be within a file but also between files and all similar blocks form a clone class (e.g., Block 1-3).

ignores literals and constants. Examples for clones classes are Block 1-3 and Block 4-6, as they all contain equivalent blocks.

Despite these basic notions, no single holistic definition exists for code clones. This is due to the different tools and their associated publications that redefine code clones according to the capabilities of the respective tool. The issue is that code clones are intuitively well-understood, but hard to formalize such that a clear and consistent definition, that covers all applications, has not been found yet. This study relies on the definition by Baxter et al. [77] as it abstracts detection method specifics without being too vague: *Code clones are segments of code that are similar according to some definition*

of *similarity*.

6.2.1 Clone Taxonomy

A taxonomy based approach helps to align the understanding of code clones, complementary to existing definitions. The typical and most frequently used categorization of code clones [75, 78, 69] is:

Type 1 (Exact Clones): Program fragments that are identical except for variations in whitespace and comments.

Type 2 (Parameterized Clones): Program fragments that are structurally/syntactically similar except for changes in identifiers, literals, types, layout and comments.

Type 3 (Near-Miss Clones): Program fragments that include insertions or deletions in addition to changes in identifiers, literals, types and layouts.

Type 4 (Semantic Clones): Program fragments that are functionally similar (i.e. perform the same computation) without textual similarity.

These types yield basic insights into the vague term of *similarity* in the code clone definition. Code fragments can be similar based on their textual representation (Type 1-3) or can have similar functionality without textual similarities (Type 4). Type 1 and 2 clones are the focus in this chapter. Code clone types also characterize the accepted difference between code fragments participating in a clone, and they further define capability levels of detection tools.

6.2.2 Clone Tools

Clone detection tools can be categorized into detection, analysis and management tools, which are often integrated into quality management platforms. Detection tools find code clones; the results are then filtered, visualized and categorized by means of analysis tools. Management tools track existing clones and their evolution to make them an integral part of the quality management process.

Detection tools can be basically categorized into text, token, tree, graph, metrics and model-based tools or hybrid approaches [78, 75]. Text-based detectors use string-matching algorithms to find similar source code parts. Token-based methods leverage lexical analysis to extract token sequences fed into a suffix-tree/array to discover clones. Tree-based tools expose the abstract syntax tree to apply tree similarity algorithms. Each approach has advantages and disadvantages that often limit their capabilities in detecting certain clone types. Text-based tools can only detect Type 1 clones and by using language dependent normalizations their capabilities improve up to Type 3 clones. Tree-based tools use computational intensive algorithms but can detect clones up to Type 3. To summarize, detection tools are the basis of clone detection and differ in their algorithmic interpretation of source code, which ultimately affects their capabilities.

A detailed overview of detection tools is given by Bellon et al. [75], Koschke [79], and Rattan et al. [78].

Clone analysis is concerned with filtration, visualization, and categorization of clones and is often tightly coupled with clone management. Common visualizations are tree maps and scatter plots [80, 81, 82, 83, 84], but also parallel plots [85] are used. A tree map uses interactive tiles that reflect the directory hierarchy colored according to their duplication intensity. Scatter plots enumerate files along the x and y-axis where each data point reflects a duplication relationship. The combination of both provides insight into the clone relationship between but also the clone intensity within files. Filtering and ranking of clones help to organize the typically large result sets of detectors. This is done manually in conjunction with visualizations or automatically based on metrics and predefined criteria. For instance, Gemini [83] or CLICS [86, 87] are tools that make use of metrics and filtering criteria to provide the most relevant subset of clones. Categorization sorts clones into views such that the inspection can be focused on a specific task. These views may be related to the location (*Same File, Same Directory*, etc.), the region (*Function to Function Clones, Macro Clones*, etc.), or the block classification (*Initialization Clones, Loop Clones*, etc.) [86] of the code fragments.

Management tools help to track the clones such that they can be actively incorporated into quality assessments and architectural decision processes, but also to evaluate their evolution. This is especially important with the increased demand of modularization in the machine and plant industry [88]. Clones are often deliberately introduced as lightweight variability mechanism, hence they exceed typically one product life-cycle. One way to manage these clones is, for example, CloneTracker [89]. It builds a model of the tracked clones and provides notifications if cloned code is changed or edited simultaneously. Another example is ECCO, Extraction and Composition for Clone-and-Own [90]. It uses fork clones in conjunction with a feature model to build a proper Software Product Line (SPL). This allows active reuse of fork clones as they are transformed into a well-defined corpus of reusable and combinable modules.

6.2.3 Related Work

Code clones are well investigated by the research community resulting in a good understanding why source code is copied. Roy and Cordy [69] presented a comprehensive overview of reasons for cloning extracted from various publications. For instance, Kim et al. [91] conducted an ethnographic study on the code clone behavior of software developers by recording the file changes. Not only language limitations forced the developers to copy code, but developers actively used the copy and paste history to determine the abstractions within their system. Another example, given by Kapser and Godfrey [92], describes several forking patterns in which large proportions of the system are copied in order to enable software ports, specific hardware implementations or (experimental) variants.

These reasons indicate – in contrast to the incentive earlier publications give [93, 77, 80, 94, 95, 96, 97, 70] — that code clones are not universally bad or result of unskilled programmers. In fact, follow-up publications found quite the contrary [98, 91, 92, 99, 100,

101, 102, 103, 104], as Rattan [78] reported, especially with respect to the stability and faults caused by code clones. Possible advantages of clones during development activities are risk avoidance [69], architectural improvements [92], performance improvements (e.g., loop unfolding, reduced call overhead) and improved code stability [101, 103, 99]. Interestingly many found advantages also show up as disadvantage indicating that measuring the impact of clones is a non-trivial task. Concluding, it is clear that it depends on more than just whether code is duplicated to make statements about the quality of a system. Nevertheless, awareness and suitable methods to track and process clones are recommended, so that benefits of cloning can be leveraged and drawbacks can be mitigated.

6.3 Industry Context

The work described in this chapter was conducted with our industry partner, a large high-tech company in the domain of machinery for metal processing. Together we analyzed a pre-release version of a machine control software system. It consisted of modules implemented in the IEC 61131-3 language Structured Text and modules written in the C/C++ programming language. The total size of the software system was 191 kLOC (Lines Of Code, LOC) with 157 kLOC in ST and 34 kLOC in C/C++, at the time the study was conducted. These numbers include only the code authored by our industry partner.

The software system was part of a large industry project and had already been evolved over several iterations with an overall development history of more than two years. In each iteration, major functional extensions were integrated, tested and stabilized. Furthermore, every iteration also included extensions that added support for different machine types and hardware variants. It was expected that this evolution led to code clones, as existing software routines were reused for similar hardware options by following a simple forking approach. Hence, code fragments up to entire subsystems were copied from the existing implementation to support the requirements of the new machine types or hardware variants.

6.4 Code Clone Analysis

We analyzed the PLC software system with Simian [76], a proprietary text-based clone detector, and evaluated a subset of the found clones. Simian can detect Type 1 clones in all text sources but incorporates additional normalization features for several common programming languages. These normalization features were re-implemented for Structured Text such that all languages used in the studied system (ST and C/C++) could be analyzed on the same capability level, i.e., Type 2 clones.

The analyzed source code has in total 99 538 C/C++ and 160 132 ST significant (non-whitespace) lines distributed over 372 C/C++ and 770 ST files. This includes C/C++ and ST libraries as header files. The source base contains multiple variants of the system for the different machine types, consequently, large portions of the code are

Table 6.1: Results of the Simian clone detection on the entirety of the code base including libraries and definition files.

Language	Option	Files with Clones	Duplicated Lines	Duplicated Blocks	Total Files	Total Sig. Lines
C/C++	Default	257	58,741	1,510	372	99,538
	Identifier	334	99,620	4,005		
	Literal	274	62,051	1,828		
	Identifier/Literal	340	117,080	4,776		
ST	Default	552	43,697	4,591	770	160,132
	Identifier	633	105,787	10,930		
	Literal	558	57,557	5,179		
	Identifier/Literal	650	133,488	12,291		

Clone overlap is allowed

Minimum number of lines = 5

very similar, leading to many clones. Many of these clones are deliberately introduced and manually managed to simplify the product line aspect of the development process. Table 6.1 contains the number of duplicated lines, blocks, and files found by the detector. The clone analysis did allow for clone overlaps in order to find partial copies of variant files while simultaneously allowing full copies. The number of duplicated lines is strongly dependent on the minimum number of lines a clone is allowed to have, which was 5 lines throughout the study. This setting is already fairly low but was chosen with the subsequent study in mind.

6.5 Code Clone Study

A group of experts inspected clones found during the clone analysis (Section 6.4) within the system of our industry partner (Section 6.3). These inspections evaluated the nature (type) of the clones as well as their relevance for refactoring in order to help to answer the following questions:

1. What natures of clones exist within the system?,
2. Is there a difference between the natures between C/C++ and ST?,
3. How does the tool support influence the relevance of clones?, and
4. How does the clone selection approach influence the relevance of clones?

Ultimately, these questions provide the first incentive for developers of PLC software to adapt existing clone detection tools for IEC 61131-3 languages such as Structured Text.

6.5.1 Study Design

A subset of the clone detector results was selected and presented to a group of experts. The experts inspected the clones and provided an evaluation of each clone with respect to their nature and relevance. The responses were then analyzed and used to explore the relationships and occurrences of clones, tool support, selection approach, and languages.

Controlled Variables

The usage of clone detection tools raises some typical questions related to the tool configuration (normalizations, minimal line length, etc.), as well to the selection approach used when analyzing the discovered clones. Each answer potentially changes the type of clones and their perceived relevance. Therefore, it is important to understand the impact of these variables when managing clones in development and maintenance projects. This study controls for the variables programming language, tool option, and clone selection approach.

- **Language:** Refers to the used programming languages and captures the heterogeneity of the system's code base, which reflects a typical setup in which multiple technologies are used in concert to solve a complex problem. The used languages are *C/C++* and *Structured Text* (ST). C/C++ are widespread general purpose programming languages for "system-near" applications. ST is a high-level block structured language designed for PLCs defined by the IEC 61131-3 standard. Both languages are procedural and imperative. They exhibit basic similarities but nonetheless they differ in syntax and expressiveness.
- **Option:** Simian offers basic analysis capabilities that can detect Type 1 clones in any text source. In addition, it provides normalization features for several popular programming languages to support the detection of Type 2 clones. In this study the following combinations of options were used: *Text* (source code is interpreted as normal text), *Identifier* (identifiers are normalized to a common symbol), *Literal* (literals are normalized to a common symbol), and *Identifier + Literal* (identifiers and literals are normalized to a common symbol).
- **Selection:** Ranking and filtering of clones is used to cope with the usually large result sets. The (confounding) selection variable reflects this behavior with the following common clone selection approaches: *Random* (clones are selected randomly), *Lines* (clones are selected in ascending order according to the number of lines they span), *Blocks* (clones are selected in ascending order according to the number of blocks they include).

Response Variables

Each clone was evaluated,

1. whether it is relevant for refactoring and,

Table 6.2: Intraclass Correlation Coefficient (ICC) of the expert responses measured by a two-way model with a fixed set of k raters.

Response	Type	ICC	95% Confidence Interval		F Test			
			Lower Bound	Upper Bound	Value	df1	df2	Sig
Aspect	ICC3k	0.819	0.789	0.845	5.528	479	958	0
Logical	ICC3k	0.902	0.886	0.916	10.193	479	958	0
Structural	ICC3k	0.962	0.955	0.967	26.080	479	958	0
Syntactical	ICC3k	0.505	0.423	0.577	2.019	479	958	0
Relevance	ICC3k	0.800	0.767	0.829	4.999	479	958	0

Number of subjects = 480

Number of raters = 3

Two-way consistency averaged-measures ICC

2. to which degree it associates to the four natures: Aspect, Structural, Syntactical or Logical.

The resulting five response variables are given by a 5-point symmetric Likert scale ranging from *Strongly disagree* to *Strongly agree* with the neutral mid-point *Neither agree nor disagree*. This evaluation scheme is based on the findings of Walenstein et al. [105] that human raters do not agree on whether a clone should be refactored or not as different developers have different emphasizes. The Likert scale mitigates this issue by avoiding a binary decision and providing different levels of association and disassociation. Each response is evenly mapped onto a scale between -1 and 1 and averaged through all raters. This results in interval scale data with respect to the raters but also to the number of clones inspected within each group enabling the usage of standard statistical methods [106].

- **Aspect:** Clones of this nature contain statements related to cross-cutting concerns, e.g., debugging, logging, permission and authentication, data monitoring, etc. These clones are often unavoidable and cannot be removed with common clone refactoring strategies. Aspect Oriented Programming (AOP) frameworks are a solution to these clones and a general review on AOP methods is given by Kurdi [107], while Bengtsson [108] describes an approach specialized for IEC61131-3.
- **Logical:** Code fragments of logical nature describe an algorithmic unit fulfilling a specific task. They contain a dense occurrence of computations and operations on data structures nested within control flow constructs.
- **Structural:** Code fragments are of structural nature if they exhibit many definitions and initializations. They build up the structure of a software system. Typical

examples are class, struct or variable definitions or initializations in header files or global constant definition files.

- **Syntactical:** Clones of syntactical nature are the result of text-based detectors that do not interpret whitespace or syntactical symbols (braces, brackets, etc.). For example, series of closing braces belonging to deeply nested control flow constructs may be detected as a clone by a text-based detector.
- **Relevance:** Relevance captures the likelihood that an expert would issue a refactoring of a particular clone in a general maintenance scenario. It reflects the typical true and false positive classification but avoids the forced binary decision.

6.5.2 Procedures

A subset of the found clones from Section 6.4 was selected and presented to three experts. Each expert was briefed in the meaning of the response variables. Each rater was free to evaluate the clones on his own pace and the inspection sessions were done self-managed. Each of the experts had a very strong background in software engineering. The average experience of the experts was 11.33 years (SD = 4.04 years).

Evaluation procedures computed the Inter-Rater Reliability (IRR) to quantify consistency and agreement among experts. Further, a linear model was fitted to expose relationships between the relevance of clones and the other variables. Finally, a set of hypothesis tests were conducted to give a further incentive on whether tool support specific for IEC 61131-3 languages are justified.

6.5.3 Evaluation

Overall 480 clones distributed over 32 groups (2 languages \times 4 options \times 4 selection approaches) with each containing 15 clones were inspected. This results in a total of 1440 inspections (480 clones \times 3 raters) conducted by the experts. A two-way, consistency, averaged, Intraclass Correlation Coefficient (ICC) measure [109] was used to assess the reliability of the 1440 inspections with respect to the nature and relevance. Results within Table 6.2 show that there was a high degree of agreement among the expert ratings over the 480 clones. The consistency was excellent (Cicchetti interpretation guidelines [110]), except for the Syntactical nature only being fair ($ICC3k_{Syntactical} = 0.505$). Given the high ICC, a minimal amount of measurement error was introduced by the experts affecting the power of subsequent analysis. Ratings on the syntactical nature were deemed too erroneous therefore excluded from further analysis.

Figure 6.2 shows the expert averaged inspection result distributions for the natures *Aspect*, *Logical* and *Structural* as well as for *Relevance*. The left facet captures inspections of clones detected only via the *Text* mode of the detector. On the right facet are inspections of clones detected with additional normalizations (*Support*) in place, i.e., normalization of identifiers, literals or both. The distributions indicate that the experts had a clear idea whether a clone is positively associated with a nature or not. This can be seen by the slim bellies in the neutral region of the response scale.

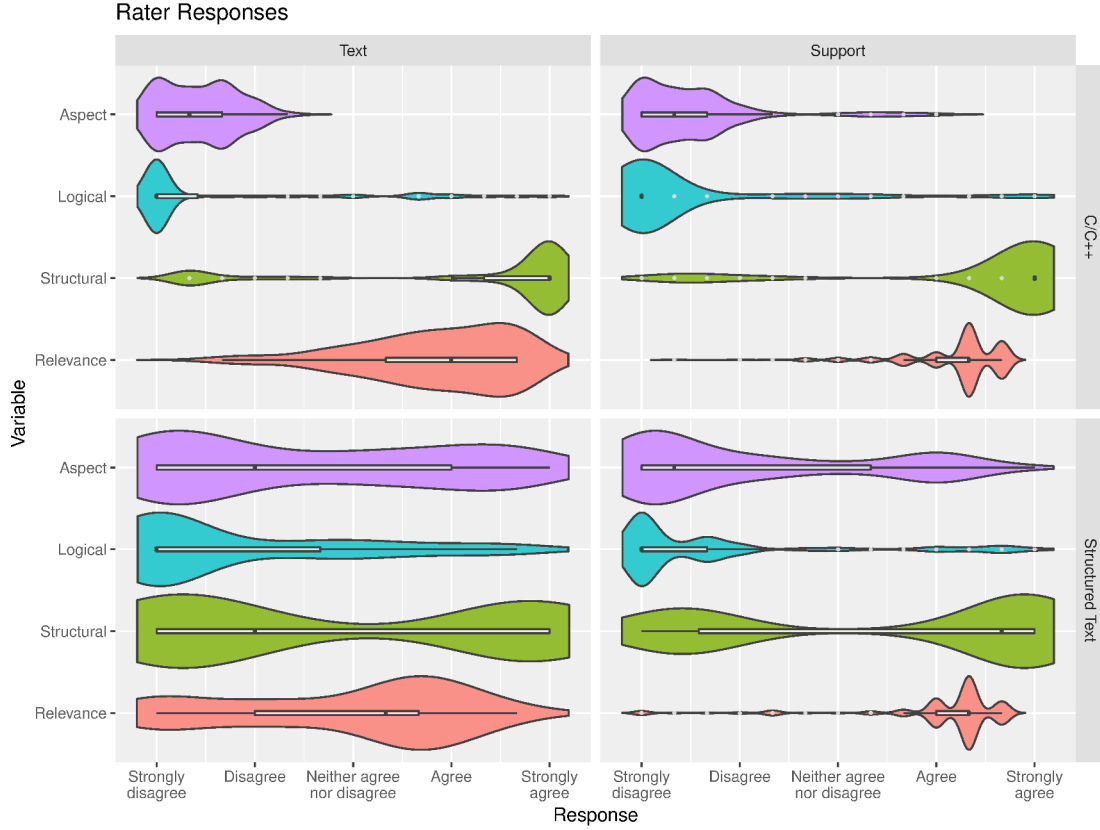


Figure 6.2: Responses with respect to the languages and natures. For easier interpretation the plot uses the Likert labels on the response axis (x-axis) although being values between -1 and 1. The Support facet includes responses from *Identifier*, *Literal* and *Identifier/Literal* option.

Clones are not associated with the *Aspect* nature for C/C++ in the text mode. This slightly changes with the usage of normalization support indicated by the longer tail of the violin shape. ST clones are stronger associated with the *Aspect* nature indicated by the third quartile reaching beyond the neutral region of the response scale. However, the central tendency is still in the disagree range for both detector capability modes. *Logical* clones are scarce and the central tendency for C/C++ and ST are both in the strongly disagree area. The outliers indicate the few clones that contain algorithmic content. Most *Logical* clones were found in the text mode for ST, nevertheless, the tendency is still towards a disassociation. Many clones found in C/C++ are of *Structural* nature indicated by the median located at strongly agree. The normalization support increases the number of *Structural* clones even more. For C/C++ the first quartile moves towards the strongly agree region, for ST a shift from dissociation to an association in the central tendency of the responses is measured.

Relevance is slightly worse for ST compared to C/C++ as the median shows, never-

Table 6.3: Significance tests of specific contrasts with respect to the depended variable Relevance.

Contrast				95% Confidence Interval						
	Language	Option	Selection	Mean Diff.	Std. Error	Lower Bound	Upper Bound		Sig.	
1	Language	Text - Support	Selection	-0.225	0.034	-0.316	-0.135	$4.82 \cdot 10^{-10}$	***	
2	C/C++	Text - Support	Selection	-0.116	0.035	-0.209	-0.023	0.005	**	
3	ST	Text - Support	Selection	-0.335	0.058	-0.488	-0.182	$6.78 \cdot 10^{-8}$	***	
4	Language	Option	Random - (Blocks, Lines)	0.063	0.027	-0.011	0.136	0.074	.	
5	Language	Option	Lines - Blocks	0.007	0.051	-0.128	0.143	0.985		
6	C/C++ - ST	Option	Selection	0.052	0.031	-0.030	0.136	0.202		
7	C/C++ - ST	Text	Selection	0.217	0.067	0.040	0.394	0.005	**	
8	C/C++ - ST	Support	Selection	-0.002	0.030	-0.083	0.079	0.986		

Sig. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Adjusted p values – free method

theless using the additional support removes this offset and places both into the strong positive range.

Linear Model

A multivariate linear regression was calculated to predict *Relevance* based on *Language*, *Option*, *Selection* but also through the natures *Logical* and *Structural*. (Note: *Syntactical* has been excluded because of an insufficient reliability of the ratings and *Aspect* did not reach significance.) The maximum positive response of *Relevance* is 1 for a perfect association, 0 for neutral and -1 for a perfect disassociation. A significant regression equation was found ($F(28, 451) = 40.04, p < 2.2 \cdot 10^{-16}$), with an R^2 of .713. Interactions between *Language* and *Option*, *Option* and *Selection*, and between *Selection* and the included natures were significant. Regression residuals show acceptable departures from normality and parallel lines as a pattern. Patterns were expected because of the Likert scale being averaged by only three raters. An unacceptable variation in the variances was detected, therefore heteroscedasticity corrected hypothesis tests were conducted.

The linear model shows a strong positive logarithmic relationship to the number of lines a clone spans, increasing its relevance by 0.1 for each magnitude in lines. Clones from ST have a lower base relevance compared to C/C++ clones (-0.22) but strong positive significant interactions ($0.18 - 0.26$) with the different options. Similar, interactions that constitute blocks and normalizations in which identifiers are normalized (Identifier, Identifier+Literal), are positive significant with estimates between $0.29 - 0.37$. All these coefficients indicate combinations of options, selection methods and languages that greatly increase the relevance of clones. In terms of nature, there were strong significant coefficients that represent the interaction between *Structural* or *Logical* with the (between file) blocks selection method.

Statistical Tests

The planned tests investigated whether there is a statistically significant effect in *Relevance* between different groups of clones. Contrasts measure effects within and between languages given the text mode and the average of all normalizations but also the effects of selection methods. Table 6.3 contains contrasts and their respective hypothesis tests with *free* [111] adjusted p values that account for multiple comparisons. Test 2 and 3 compare the differences between text mode and additional support within the two languages where both reach significance, although ST with a larger effect. No marginal significant effect between random selection, and block or line oriented selection approaches could be found (Test 4). However, the low p -value and the existence of interactions indicate that significant effects between specific levels of the variables are present. Tests 6, 7 and 8 represent between language tests with no marginal effect (Test 6). Test 7 shows a significant difference of clones between the two languages given that they were detected with the text mode. Test 8 shows that this significance is not found if clones are detected with additional normalizations.

6.5.4 Interpretation

The results show that the tool support has a positive effect on the relevance of clones. This can be seen in Figure 6.2 where the median and first quartile moves into the strong positive region, but also in the hypothesis tests. This positive effect is given in the within language tests (Test 2 & 3) but also in between languages tests where the initial significant difference is removed by the additional normalization features. The between language effect is most likely caused by the header files (.h-files) of C/C++ that introduce more structural duplicates, which in hindsight are often relevant (linear model coefficient).

The selection approach does not influence the relevance of clones on average. However, there were positive effects associated with selection methods based on the number of blocks that are shared between files.

Most clones are of *Structural* nature and the usage of normalizations increases their total proportions making them more likely to be encountered. *Logical* clones are inversely proportional to structural clones and therefore less often encountered with normalizations. Clones of *Aspect* nature are mostly found with a low minimum line count of clones but remain strongly dependent on the application context. The nature of clones between the languages is fairly similar with mostly structural clones and some logical clones. ST code contained more aspect clones nevertheless these are less prevalent if normalizations are used.

6.6 Threats to Validity

The study faces threats to validity that might reduce the power of the analysis. First, the generalization of results is limited because only one software system was analyzed. However, the system is a real-world example and the applied development approach can be considered representative for many other evolving software systems for industrial

automation [74]. Furthermore, the choice of the analysis tool and its implementation, as the experts and their specific background in software development may also have introduced a bias. Finally, the confounding configuration problem discussed by Wang [112] might be an issue. We chose the minimal line count for a clone fairly low such that aspect clones are easier spotted.

6.7 Conclusions

In this chapter, we presented the results from the analysis of code clones in a real-world PLC software system, which has been evolved over several development iterations as part of a large industry project. The software system contained code written in C/C++ and in IEC 61131-3 Structured Text.

We found that clones do exist in PLC software systems regardless of the applied programming language. Awareness for clones is an important aspect of professional software development, independent whether they are viewed positive or negative. Industry projects require support for detecting, tracking and managing clones as software systems evolve. Similarly to previous studies [74], we can also conclude that the existing tool support for PLC languages with respect to clone detection is insufficient. Furthermore, we found that language adaptations for detectors, that enable the use of normalizations, improve the relevance of clones significantly. This is especially true for maintenance scenarios focusing on structural deficiencies. Concluding, companies that develop PLC systems can justify investments in clone detector adaptations. These investments widen the range of clone detection, analysis and management tools and strengthen professional software development within the industrial automation industry.

Possible future extensions to the work include methodologies for efficient filtering of clones based on their nature through complexity metrics and the repetition of the study on other PLC software systems including systems from different industry partners.

7 Case Study: Semantic Clone Detection

Abstract:

Semantic clone detection is the process of finding program elements with similar or equal runtime behavior. For example, detecting the semantic equality between the recursive and iterative implementation of the factorial computation. Semantic clone detection is the de facto technical boundary of clone detectors. This boundary was tested over the last years with interesting new approaches. This chapter contributes a semantic clone detection approach that detects clones with 0 % syntactic similarity. We present Semantic Clone Detection via Probabilistic Software Modeling (SCD-PSM) as a stable and precise solution to semantic clone detection. PSM builds a probabilistic model of a program that is capable of evaluating and generating runtime data. SCD-PSM leverages this model and its model elements for finding behaviorally equal model elements. This behavioral equality is then generalized to semantic equality of the original program elements. It uses the likelihood between model elements as a distance metric. Then, it employs the likelihood ratio significance test to decide whether this distance is significant, given a pre-specified and controllable false-positive rate. The output of SCD-PSM are pairs of program elements (i.e., methods), their distance, and a decision whether they are clones or not. SCD-PSM yields excellent results with a Matthews Correlation Coefficient greater 0.9. These results are obtained on classical semantic clone detection problems such as detecting recursive and iterative versions of an algorithm, but also on complex problems used in coding competitions. Semantic clone detection is the process of finding program elements with similar or equal runtime behavior.

Authors:

H. Thaller, L. Linsbauer, and A.Egyed

Parts of this chapter was published at the 14th International Workshop on Software Clones (IWSC), London, ON, Canada (2020) [3].

7.1 Introduction

Copying and pasting source code fragments leads to code clones that are considered an anti-pattern. Code clones increase maintenance costs [96, 98], promote bad software design [113, 72, 114], and introduce or propagate bugs [115, 116, 117]. However, duplicating code fragments allows faster adaption to requirements, the reuse of stable and well-tested solutions [99, 100], and helps to overcome language limitations [92, 69] lowering development costs. The impact of code clones and the contradicting evidence various

studies provide are the topics of an ongoing discussion in the community. However, developers will continue duplicating source code to leverage their benefits despite their drawbacks. The key is the awareness and management of clones to maximize efficiency while balancing quality.

Traditionally, the clone taxonomy distinguishes between four types of clones [69, 75, 78]. Type 1-3 describe code clones caused by copying and pasting the source code with or without changes. Type 4 clones describe code clones that do not have any syntactic similarity but implement the same functionality (semantic equivalence). For example, the recursive and iterative implementation of an algorithm (e.g., Fibonacci computation) have no syntactic similarity while implementing the same functionality. Existing tools have limited or no capabilities to detect Type 4 clones [118]. Most current studies exclude Type 4 clones because of the lack of tool support [119, 69, 75, 120, 121]. Nevertheless, Type 4 clones exist, and recent research efforts try to deepen the understanding of them [118, 122, 123]. This chapter provides a significant contribution to semantic clone detection in the form of novel concepts and a prototype implementing them.

We present *Semantic Clone Detection via Probabilistic Software Modeling (SCD-PSM)*. SCD-PSM extends on the Probabilistic Software Modeling (PSM) framework presented in Chapter 4 via a semantic clone detection pipeline. PSM builds probabilistic models (PMs) from programs. It analyzes the static structure and dynamic runtime behavior and replicates the program in the form of a generative probabilistic model. These models allow developers to reason about a program’s semantics via causal reasoning. SCD-PSM extends this work by leveraging the PMs and causal reasoning to find semantically (i.e., behaviorally) equivalent code elements (e.g., methods or properties with similar runtime behavior). SCD-PSM allows full quantification of the behavioral distance of code elements via likelihoods (i.e., probabilities). Furthermore, the likelihood evaluation via PMs allows for statistical significance tests (i.e., likelihood ratio test) to decide whether a pair of code elements are clones or not. SCD-PSM detects semantic clones with no textual similarity, such as the iterative and recursive version of an algorithm mentioned above. The average performance of the approach reaches a Matthews Correlation Coefficient of 0.965 on the dataset indicating a robust method for semantic clone detection.

Section 7.2 provides the background needed to understand SCD-PSM including the basics of PSM. Section 7.3 clarifies what semantic clones are in the context of this chapter. Section 7.4 presents the approach in which representation, search space, and the various similarity stages are described. The study in Section 7.5 evaluates the approach and Section 7.6 discusses its results. Limitations of the approach and possible threats to validity of the study are given in Section 7.7 and Section 7.8. Section 7.9 sets the results in context of the current state-of-art and Section 7.10 concludes this chapter.

```

1 int fa(int n){
2     product = 1
3     for(i = 1; i <= n i++)
4         product *= i
5     return product
6 }

```

Listing 7.1: *for-loop* implementation of factorial

```

1 int fb(int n){
2     product = 1
3     i = 1
4     while(i <= n)
5         product *= i
6         i++
7     return product
8 }

```

Listing 7.2: *while-loop* implementation of factorial

```

1 int fc(int n){
2     if(n <= 1) return 1
3     return fc(n - 1) * n
4 }

```

Listing 7.3: *Recursive* implementation of factorial

```

1 int fd(int n, String guard){
2     if(n < 1 && guard == "val") return -1
3     if(n < 1 && guard == "throw") throw Exception()
4     return fc(n)
5 }

```

Listing 7.4: *Delegate* implementation of factorial

7.2 Background

The clone detection research community has a long history and defines many concepts, algorithms, and tools. In contrast, Probabilistic Software Modeling (PSM) is relatively new and combines software engineering and probabilistic modeling. Some terms need clarification; others require an introduction if they diverge from their traditional domain names.

7.2.1 Clone Detection

Clone detection is the process of finding two similar program fragments. Listings 7.1 to 7.4 are four different implementations of the factorial function ($n!$). Listing 7.1 is a *for-loop* implementation, Listing 7.2 uses a *while-loop*, and Listing 7.3 is recursively defined. Finally, Listing 7.4 delegates its implementation to `fc()` from Listing 7.3 but may also return -1 in case of invalid inputs (including $n = 0$).

Clone detection revolves around the representation, pairing, and the similarity evaluation. *Representations* can be, e.g., text, graphs (e.g., AST), or probabilistic models like in this work. *Pairing* describes the selection of two fragments that are potentially clones (e.g., `fa()` and `fb()`). Each pair is called a *candidate clone pair* (or candidate pair). The *similarity evaluation* measures the similarity between the fragments of a candidate pair (e.g., counting the number of different characters). Finally, the *clone decision* labels the candidate pair as a clone or not given a criterion on the similarity (e.g., less than ten different characters).

The properties of the similarity metric split clones into two groups [69]. Type 1-3 clones capture textual similarity while Type 4 clones capture semantic similarity [75, 119, 124, 69, 78, 1].

Type 1: (Exact Clones) Program fragments that are identical except for variations in white-space and comments.

Type 2: (Parameterized Clones) Program fragments that are structurally/syntactically similar except for changes in identifiers, literals, types, and comments.

Type 3: (Near-Miss Clones) Program fragments that include insertions or deletions in addition to changes in identifiers, literals, types, and layouts.

Type 4: (Semantic Clones) Program fragments that are functionally/semantically similar (i.e., perform the same computation) without textual similarities.

These types are increasingly challenging to detect, with Type 4 being the most complex one. Note, that the definition of *Semantic Clones* is often relaxed where up-to 50% syntactic similarity of the code fragments is allowed (e.g., BigCloneBench [120]). However, we consider these clones as complex Type 3 clones (additions, deletions, reordering) and *not* as semantic clones. This means that semantic clones in the context of this work are clones with no syntactic similarity except for per-chance similarities (e.g., similar parameter names).

We will use $a \simeq b$ to denote that a is a clone of b . Furthermore, $a \not\simeq b$ denotes that a is not a clone of b .

7.2.2 Programs & Code Elements

PSM generalizes object-oriented terms to describe *code elements* in a program. Code elements are *types* T , *properties* Pr , and *executables* Ex that refer to, e.g., classes, fields, and methods in Java [32], or classes, properties, and functions in Python [28].

Additional code elements are *parameters* Pr and *results* Re of executables that refer to parameters (arguments) and return values of a method. Properties, parameters, and results are *atomic* code elements that have identifiable states at runtime. Types and executables are *compositional* elements that act as a collection of atomic elements.

Types *declare* properties and executables, capturing structural relationships. Executables have behavioral relationships that are categorized into *Inputs* (I) and *Outputs* (O). *Inputs* are *received parameters* Pa^I , *read properties* Pr^I , and *requested invocation results*

$Re^{\mathcal{I}}$. Outputs are returned executable results $Re^{\mathcal{O}}$, written properties $Pr^{\mathcal{O}}$, and provided parameters $Pa^{\mathcal{O}}$.

We will denote atomic elements in lowercase, and compositional elements in bold-face lowercase, e.g., n and \mathbf{fa} in Listing 7.1. Executable results are named after their executables, e.g., fa in Listing 7.1. $\mathbf{fc} = \{n^{Pa, \mathcal{I}}, fc^{Re, \mathcal{I}}, fc^{Re, \mathcal{O}}\}$ denotes the code elements of Listing 7.3. For readability's sake, we will omit the superscript classifiers if it is unambiguously possible e.g., $\mathbf{fa} = \{n, fa\}$. The subset of *inputs* is denoted by $\mathbf{fc}^{\mathcal{I}} = \{n^{Pa, \mathcal{I}}, fc^{Re, \mathcal{I}}\}$ and *outputs* by $\mathbf{fc}^{\mathcal{O}} = \{fc^{Re, \mathcal{O}}\}$. Finally, the set of all input and output combinations is given by

$$\mathbf{ex}^{\mathcal{IO}} = \{(i, o) \in \mathbf{ex}^{\mathcal{I}} \times \mathbf{ex}^{\mathcal{O}}\}. \quad (7.1)$$

For example, $\mathbf{fd}^{\mathcal{IO}} = \{(n, fd), (guard, fd)\}$ describes the IO pairs of $\mathbf{fd}()$.

7.2.3 Probabilistic Software Modeling

Probabilistic Software Modeling (PSM) [125] is a data-driven modeling paradigm that transforms a program into a probabilistic model. PSM extracts the structure and behavior of a program. The *structure* are the code elements of the program as described in Section 7.2.2. This includes the actual elements, but also their call and structural hierarchy. The *behavior* is the runtime of the program, i.e., the actual data that is transformed by the program. The resulting PM and its model elements (i.e., random variables) is a copy of the original program with its code elements.

Model elements are the code elements in the context of a PM, and there is a one-to-one relationship between them. We will reuse the notation of code elements for model elements and augment it via probabilities. $P(x)$ denotes the probability distribution of variable x , e.g., $P_{\mathbf{fa}}(n)$ denotes the probability distribution of input parameter n of the \mathbf{fa} -method. $p(x)$ denotes the probability of a specific event of a variable, e.g., $p_{\mathbf{fa}}(n = 2)$. The notation of code elements and model elements is very similar. However, model elements reasons about the behavior of code elements instead of their structural properties.

Model elements can generate observations (i.e., runtime events), but also evaluate their likelihood. Each model element is a flow-based latent variable model [30] that learns an invertible mapping between the original observations and an isotropic unit norm Gaussian $\mathcal{N}(0, 1)$ with $f : X \mapsto Z$. An example for $x \in X$ may be $n \in \mathbf{fa}$ with $n^z \in \mathbf{fa}^z$ being its latent Gaussian representation. The Gaussian latent space enables the model elements to generate new samples and evaluate the likelihood of samples.

Generation (or Sampling) draws, either marginally or conditionally, observations from a model element simulating the execution of the corresponding code element. For example, drawing 100 observations from $\mathbf{fa} \sim P_{\mathbf{fa}}(n, fa)$, i.e., values for $n^{\mathcal{I}}$ and $fa^{\mathcal{O}}$, simulates 100 program executions of this method. An example for *conditional generation* would be $\mathbf{fa}_{|n < 10} \sim P_{\mathbf{fa}}(fa | n < 10)$ that only draws observations where $n < 10$. The process involves sampling from the latent Gaussian variables, and inverting the Gaussian samples to the original domain via the flow $f^{-1}(z) = x$. *Evaluation* takes observations and evaluates their likelihood under a model element. For example, $P_{\mathbf{fa}}(n = 4, fa = 24)$

evaluates the likelihood of input 4 and output 24 under the fa model element. Intuitively, it asks the model how likely is the method fa called with 4 while returning 24. The process of evaluation involves mapping a given sample into the latent space and evaluating it under the Gaussians $p_{\mathcal{N}(0,1)}(f(\mathbf{x}))$. Generation and evaluation are the core of any PSM applications and of SCD-PSM. How generation and evaluation work in detail and from a technical perspective is given by Thaller et al. [126].

7.3 Semantic Clones

A clear understanding of what SCD-PSM defines as *semantic clone* is essential in understanding the approach and its design choices.

Definition 5. *A semantic clone is a pair of executables whose (partial) input, and output relationships exhibit significant (conditional) similarities.*

Definition 5 defines semantic clones over the similarity between IO relationships of executables. This holds if the IO relationships are only partially similar, i.e., not all combinations of IO pairs between executables have to be similar. For example, fd in Listing 7.4 has two IO pairs ($\mathbf{fa}^{IO} = \{(n, fd), (guard, fd)\}$) while fa in Listing 7.1 has one IO pair ($\mathbf{fa}^{IO} = \{(n, fa)\}$). According to the definition, at least one IO pair comparison needs to be similar such that both executables are declared as a semantic clone (e.g., $(n, fd) \simeq (n, fa)$).

Furthermore, the similarities between IO pairs may only be conditional, i.e., the similarity of matching IO pairs might be depending on the state of any other code element in the comparison context. For example, the IO pair $(n, fd) \simeq (n, fa)$ is only a perfect clone in case that `fd.guard != "val"`. If `fd.guard == "val"` the IO behavior would differ in case of $n = 1$ (`fd(1) ↦ -1` while `fa(1) ↦ 1`). According to the definition, at least parts of the behavior need to be similar, capturing complex multidimensional behavioral patterns in IO relationships.

The rationale behind the comparison of IO relationships is one of cause and effect. If a pair of executables exhibit similar effects given similar causes, then their computational behavior is identical. Extending this rationale by multiple inputs and outputs results in multiple IO relationships leading to *partial conditional similarity*.

7.4 Approach

Figure 7.1 illustrates SCD-PSM. It is a five-fold approach in consisting of the following steps:

- A. **[Modeling]** PSM builds a probabilistic model that reflects the original program;
- B. **[Search Space]** A search space of candidate pairs is constructed by pairing executable model elements;
- C. **[Static Similarity]** The static similarity accepts candidate pairs with matching data types;

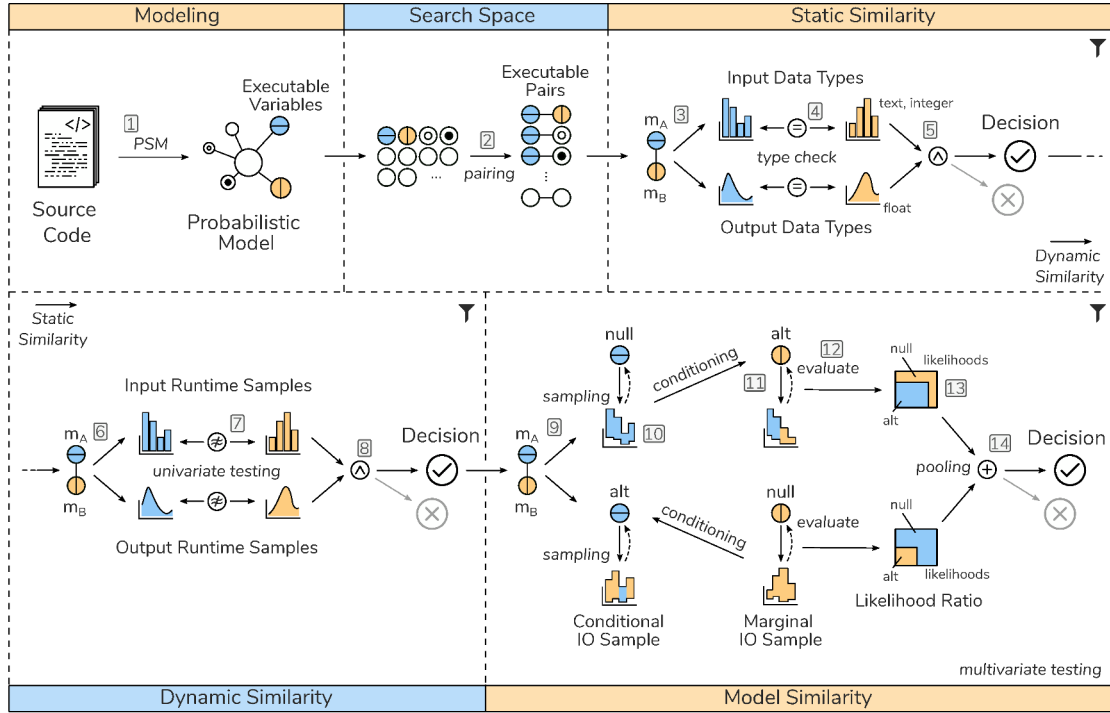


Figure 7.1: The modeling phase transforms the program into a PM. The search space phase then pairs the PM model elements into candidate pairs. Finally, Static-, Dynamic- and Model Similarity evaluates the behavioral equality of the candidates.

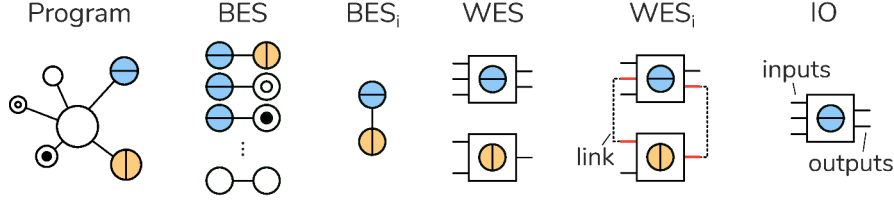


Figure 7.2: SCD-PSM operates on four levels of abstraction: program, between executable, within executable, and the IO level.

- D. **[Dynamic Similarity]** The dynamic similarity accepts candidate pairs with similar runtime data;
- E. **[Model Similarity]** The model similarity accepts candidate pairs with similar model behavior.

The approach represents a rejecting filter pipeline that candidate pairs must traverse to be declared a clone. Static-, Dynamic-, and Model Similarity represent filter stages of increasing complexity.

The main contribution of this chapter is the implementation of a semantic clone detection pipeline on top of PSM. Further, we provide a simple yet effective process of traversing the search space of potential clones while respecting the pipeline’s performance properties. Finally, we show that the behavioral equivalence of model elements generalizes to the semantic equivalence of code elements.

7.4.1 Modeling

Starting from the *Source Code* in Figure 7.1, PSM builds a *Probabilistic Model* (PM) [125] of the program (1). The PM is also called Inference Graph (IG) and represents a cluster graph with NVPs [30] as local models as nodes. SCD-PSM uses this PM as a representation for the clone detection similar to text-based clone detectors that use text fragments. While being the first step of SCD-PSM, we consider the PM as given since we only use PSM to build a model.

Executable model elements in the PM act as a surrogate to the executables in the original program (e.g., `fa()`). SCD-PSM uses these elements and computes their similarity finding behaviorally equivalent model element pairs and extrapolates this knowledge to semantically equivalent code element pairs. In conclusion, the SCD-PSM allows for method-level semantic clone detection based on PMs representing the original executables in the program.

7.4.2 Search Space

SCD-PSM conducts method-level semantic clone detection, which operates on multiple abstraction levels. Figure 7.2 illustrates these levels starting with the program and ending with the inputs and outputs of an executable.

The second step in Figure 7.1 builds a *within- and between-executable space* that SCD-PSM searches for clones. The *Between-Executable Space (BES)* is the set of executable combinations

$$\mathbf{BES} = \{\{\mathbf{a}, \mathbf{b}\} \in \mathbf{Ex} \times \mathbf{Ex} \mid \mathbf{a} \neq \mathbf{b}\}, \quad (7.2)$$

where $\mathbf{exa}, \mathbf{exb}$ is a *candidate pair* (or executable pair), and \mathbf{Ex} is the set of all executables in the current analysis (illustrated in Figure 7.2). The theoretical size of the between-executable space are all 2-length combinations without replacement given by

$$|\mathbf{BES}| = \frac{|\mathbf{Ex}|!}{2 \cdot (|\mathbf{Ex}| - 2)!}, \quad (7.3)$$

where $|\cdot|$ describes the size of the underlying set. Note that the size of the BES is smaller than the Cartesian product since $\{a, b\} = \{b, a\}$. Figure 7.1 shows this paring process in the Search Space aspect (2). The *Within-Executable Space (WES)* is the product of IO pairs

$$\mathbf{WES}^{ab} = \{(i, j) \in \mathbf{a}^{\mathcal{IO}} \times \mathbf{b}^{\mathcal{IO}}\}. \quad (7.4)$$

Figure 7.2 illustrates the WES and one IO pair from the WES that we also call *link*. The theoretical size of the within-executable space is

$$|\mathbf{WES}^{ab}| = |\mathbf{a}^{\mathcal{IO}}| \cdot |\mathbf{b}^{\mathcal{IO}}| \quad (7.5)$$

For the sake of visualization IO pairs are not shown in Figure 7.1 but are abstracted in their executable elements. The maximum theoretical search space is

$$S = \sum_i |\mathbf{wes}(\mathbf{BES}_i)|, \quad (7.6)$$

given that \mathbf{wes} describes a construction function according to Equation (7.4), and \mathbf{BES}_i is the i 'th candidate pair.

In practice, SCD-PSM evaluates only a fraction of possible combinations because of the skip evaluation. The *skip evaluation* consists of two search space limiting factors: greedy evaluation and transitive similarity. *Greedy evaluation* stops the search through the WES once a similar pair is found. The initial detection process is only interested in whether there is a similarity or not. Post-analysis can then find all possible similarities for the program comprehension and developer guidance. *Transitive similarity* skips evaluations in the BES because if $a \simeq b \simeq c$ then also $a \simeq c$ holds. In sum, *skip evaluation* allows for efficient processing by stopping the search through the WES once a clone pair is found, and by skipping pairs in the BES if the transitive similarity is given.

In conclusion, SCD-PSM compares IO pairs of executable model elements and uses greedy evaluation and transitive similarity to traverse the search space efficiently.

7.4.3 Static Similarity

The static similarity stage is a filter that accepts candidate pairs based on their data type, as shown in Figure 7.1. Data types in a PSM model are integers, floats, and text.

Input (3) of the stage are the IO pairs $WES^{ab} = wes(\{a, b\})$ of a candidate. The filter *criteria* (4) accepts a candidate pair if at least *one* link (i.e., IO pair) has a matching data type, i.e., the input but also the output have a matching data type. *Output* (5) is a boolean decision whether the candidate pair is a clone or not from a static viewpoint. If positive, then the candidate pair is moved to the next pipeline stage, i.e., the *Dynamic Similarity* evaluation (see Figure 7.1). If negative, then the candidate pair is marked as being *not* a clone $a \not\simeq b$ and no further processing is conducted. For example, the IO pairs $(n, fa) \simeq (n, fb)$ would be statically accepted as clones as both inputs and outputs have the same data type (integer). A counterexample is given by $(n, fa) \simeq (guard, fd)$ where the input data types are integers and text.

The static similarity indicates that the analyzed program is given in a programming language that allow for static analysis. Programs written in programming language without static typing can not make use of this filter stage. In conclusion, the static similarity stage filters candidates based on their data type.

7.4.4 Dynamic Similarity

The dynamic similarity stage is a filter that accepts candidate pairs based on the runtime data, as shown in Figure 7.1.

Input (6) of the stage are again the IO pairs of a candidate as in the static similarity Section 7.4.3. The filter *criteria* (7) accepts a candidate pair if at least *one* IO pair has an *insignificant* diverging runtime distribution. The divergence of inputs and outputs can be computed via univariate statistical tests. This work uses the Kolmogorov-Smirnov test [67], which is a general non-parametric test. *Output* (8) is a boolean decision whether the candidate pair is a clone or not from a dynamic viewpoint. Again, the candidate is removed from the pipeline or forwarded to the next pipeline stage, depending on the decision. For example, the IO pair $(n, fa) \simeq (n, fd)$ with `guard == true` fails the filter given that runtime events with $n = 0$ reach a critical mass. In comparison, $(n, fa) \simeq (n, fb)$ would be accepted by the stage.

A requirement is that the candidates use a synthetic trigger. Otherwise, the comparison of the data distributions may fail because of the different *modus operandi* of the program. For example, running `fa` and `fb` where $n_{fa} = \mathcal{U}(0, 4)$ and $n_{fb} = \mathcal{U}(5, 10)$ would cause the dynamic stage to fail even if the implementations are equivalent. Property-based testing [127] can be used for this purpose to generate diverse synthetic inputs.

In conclusion, the dynamic similarity pre-filters candidates based on univariate tests on the input and output events.

7.4.5 Model Similarity

The model similarity stage is a filter that accepts candidate pairs based on the model data, as shown in Figure 7.1. The model data is sampled from the models of the executables in the candidate pairs. This stage conducts a multivariate test as opposed to the univariate test the dynamic similarity computes. The multivariate testing considers the conditional similarities that are not part of the currently inspected IO behavior but

maybe conditionally influencing the IO pair. For example, $(n, fd) \simeq (n, fa)$ holds but is conditionally dependent on *guard*. The dynamic stage can only look at the marginal (i.e., average) behavior of the IO pair, while the model similarity can factor *guard* into its decision.

Input (9) are the IO pairs of a candidate $\mathbf{WES}^{ab} = \text{wes}(\{\mathbf{a}, \mathbf{b}\})$. Then the cross-wise log-likelihood ratio of the models is computed by (*conditional*) *generation* and *evaluation*. *Output* is a boolean decision whether the candidate pair is a clone or not from a model viewpoint. Figure 7.1 illustrates the entire process of the model similarity.

- (A) A reference model $M^{null} = \mathbf{a}$ and an alternative model $M^{alt} = \mathbf{b}$ (candidate pair executables).
- (B) A IO-pair $p = \mathbf{WES}_i^{ab}$ is selected that is target of the comparison (link).
- (C) A reference sample D^{null} is drawn from M^{null} (10).
- (D) An alternative sample $D^{alt|null}$ is drawn from M^{alt} by optimizing towards the p dimensions in the D^{null} effectively conditioning the drawn samples (11).
- (E) D^{null} is evaluated under M^{null} resulting the reference log-likelihood LL^{null}
- (F) $D^{alt|null}$ is evaluated under M^{alt} (12) yielding the alternative log-likelihood LL^{null} .
- (G) Finally, likelihood ratio of the link is computed

$$\lambda = LL^{alt} - LL^{null} \quad (7.7)$$

This procedure is repeated for the reverse assignment of null and alt roles. Finally, both log-likelihood ratios are combined by a pooling operator into the final decision (14).

The role-swap is needed to avoid sub-model relationships. For example, if $M^{null} = \mathcal{N}(0, 3)$ and $M^{alt} = \mathcal{N}(0, 1)$ then LL^{alt} will be very high because M^{alt} is a sub-model from M^{null} . However, reversing the roles highlights the differences in the models.

The final decision is based on the Generalized Likelihood Ratio Test (GLRT) [11]. It measures whether the log-likelihoods are significantly different from 0 where λ (see Equation (7.7)) is the test statistic. The null hypothesis is that the models are significantly different and rejects for small ratios $\lambda \leq c$ where c is set to an appropriate Type 1 error (false-positive rate). For example, $\lambda < \log(0.01)$ allows 1 out of 100 candidates to be a false-positive, i.e., wrongly rejecting semantic equivalence. The pooling operator combines the link results either via hard or soft pooling. *Hard pooling* conducts for both links a GLRT yielding a positive decision if *both* links are positive. *Soft pooling* averages the link log-likelihoods ratios and then computes the GLRT yielding a positive decision if the joint GLRT is positive. Hard pooling does not allow any sub-model relationships, while soft pooling relaxed this constraint.

In conclusion, the model similarity conducts a multivariate significance test between two models, including possible conditional dependencies.

7.5 Study

This study answers the following research questions.

- RQ1 Does behavioral equality between model elements generalize to semantic equality of code elements?
- RQ2 Does the skip evaluation significantly reduce the computational demand of SCD-PSM?
- RQ3 Does the skip evaluation negatively impact the detection performance (i.e., precision, recall, and MCC)?

RQ1 answers the essential question if semantic clones can be detected via SCD-PSM. RQ2 answers whether the search space can be efficiently processed using skip evaluation. RQ3 answers the question of how the skip evaluation influences the performance of the detection process. This is important because candidate pairs might be skipped based on false-positives or false-negatives.

7.5.1 Setup

We implemented a prototype for SCD-PSM on top of Gradient [125], a prototype for PSM. The elements and data flow of the detection process are shown in Figures 7.1 and 7.2.

1. The input *Source Code* were 13 different clone classes with a total of 108 implementation variants (see Table 7.1). This includes classical algorithms implemented recursively and iteratively (e.g., bubble sort), but also hard problems from the programming competition Google Code Jam¹.
2. The *Probabilistic Model* (i.e., the inference graph) was computed via Gradient, a PSM prototype. We used the same hyper-parameters as reported in Table 5.1. The inference graph is a truncated inference graph not modeling types as they are not needed in a comparison between executables.
3. The *Search Space*, i.e., the *BES* and *WES*, was created according to Section 7.4.2.
4. Each valid candidate pair was then submitted to the *Static*-, *Dynamic*-, - and *Model-Similarity* and filtered according to Sections 7.4.3 to 7.4.5. Candidates that passed the entire filter pipeline were marked as clones.

7.5.2 Dataset

The study uses three well-known algorithms and 10 Google Code Jam 2017 (GCJ)¹ problems as listed in Table 7.1. The total dataset contains 108 implementation variants across 13 clone classes described by *Instance*.

Each clone class was differentially tested to verify the behavior across instances. Factorial, Fibonacci, and Sort do not need any further explanation. The GCJ problems

Table 7.1: Semantic clone detection dataset consisting of 108 instances across 13 clone classes.

Problem	Type	Property	Source Code			Clone	
			Executable	Parameter	Loc	Class	Instance
Factorial	2	0	4	2	21	A	2
Fibonacci	2	0	4	2	10	B	2
Sort	4	0	12	18	110	C	4
R0AA	11	4	39	54	384	D	10
R0AB	10	0	33	27	374	E	10
R0AC	12	4	32	45	433	F	10
R1AA	10	0	26	41	543	G	10
R1AB	13	8	28	55	595	H	10
R1BA	12	4	31	53	269	I	10
R1CA	17	18	35	62	463	J	10
R1CB	15	18	34	82	772	K	10
R1CC	10	0	21	41	338	L	10
R2AA	10	0	23	18	471	M	10
128		56	322	522	4783	13	108

are well specified complex optimization problems packaged in an everyday theme. R0AA in Table 7.1 is the round 0 (qualification), category A, and task A problem. It asks developers to compute the minimum amount of flips of pancakes that are needed for all pancakes to be on the upside. The constraint is that always k pancakes are flipped at the same time.

The dataset contains in total 5778 (see Equation (7.3)) candidate pairs of which 458 are semantic clones and 5320 are not. This yields a positive to negative ratio of 1/11.6, indicating a highly imbalanced distribution. A similar or even more pronounced imbalance is to be expected in real-world applications.

Each instance was triggered with input data to allow PSM to model the different implementations. Factorial, Fibonacci, and Sort were triggered by sampling from a uniform distribution $\mathcal{U}(0, 20)$. GCJ problems were triggered by the input data provided by the competition. Each instance received the same trigger.

GCJ problems read from and write to the standard stream, which is impractical in terms of reproducibility. Our dataset is constructed such that each implementation has a `run`-method representing the cloned executable. The study results are limited to the `run`-method even if the solutions use helper methods. Helper methods, e.g., may be methods that compute parts of the final solution, or reorganize the data. This guarantees a proper problem scope, well-defined recall and precision values, and a clearly defined

¹<https://codingcompetitions.withgoogle.com/codejam/archive>

benchmark for future reproducibility.

7.5.3 Controlled Variables

The study controls for the search space *Evaluation* strategy, *Dynamic False-Positive Rate (D-FPR)*, *Model False-Positive Rate (M-FPR)*, and *Pooling*.

Evaluation describes how the search space is processed: *exhaustive*, or *skip*. The exhaustive evaluation compares each executable candidate with each other. The skip evaluation uses the transitive similarity (see Section 7.4.2) and may skip evaluation if possible.

Dynamic False-Positive Rate (D-FPR) defines the critical value α of the Kolmogorov-Smirnov test with 0.001 and 0.01 at which similarity is rejected.

Model False-Positive Rate (M-FPR) defines the critical value c of the Generalized Likelihood Ratio test with 0.001 and 0.01 at which similarity is rejected.

Pooling defines how the likelihood ratios from the two link directions are combined (see Figure 7.1, (8)) with values: *hard*, or *soft*. *Hard* pooling evaluates whether each link reaches the critical value c and accepts the clone if both links evaluate positive.

$$\lambda_{Link_A} \leq \frac{\log c}{2} \text{ and } \lambda_{Link_B} \leq \frac{\log c}{2} \quad (7.8)$$

Soft pooling evaluates the average log-likelihood ratios (geometric mean of likelihoods)

$$\frac{\lambda_{Link_A} + \lambda_{Link_B}}{2} \leq \log c, \quad (7.9)$$

and compares it against the critical value c .

An additional fixed parameter is the *number of particles*. *Number of particles* defines the samples size that is generated during the model similarity $|D| = 50$.

7.5.4 Response Variables

The response measures of the study are the number of *Skip Evaluations*, processing *Duration*, *TP*, *FP*, *TN*, *FN*, *Precision*, *Recall*, *F1*, and *Matthews Correlation Coefficient*.

Skip Evaluations measures the number of evaluations that were skipped because of the skip evaluation strategy.

Duration measures the elapsed time to compute one candidate pair.

TP, FP, TN, FN measures the True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) detection results compared to the ground truth.

Precision measures the fraction of detected clones that are truly clones

$$\frac{TP}{TP + FP} \quad (7.10)$$

Recall measures the fraction of semantic clones that have been found

$$\frac{TP}{TP + FN} \quad (7.11)$$

F1 measures the accuracy of a binary classification as the harmonic mean of recall and precision

$$2 \cdot \frac{precision \cdot recall}{precision + recall} \quad (7.12)$$

Matthews Correlation Coefficient (MCC) measures the quality of the clone detection in form of a correlation ranging from -1 to 1 with 0 being a random selection. The MCC will be the reference performance metric as it is the most robust metric in an imbalance binary classification setting [128]. It is a correlation coefficient which may be interpreted by the guidelines proposed by Evans [129].

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (7.13)$$

7.5.5 Comparison Approaches

In total, eight alternative approaches are used to provide some context to the performance of SCD-PSM. The selected tools have a large variety in terms of internal representation and clone detection capabilities as listed in Table 7.5. ASTNN (8) and ASTNN Leaky (9) is the same approach but have different evaluation methods. ASTNN Leaky (9) uses a random split of the dataset as reported by the authors [130]. ASTNN Leaky overestimates the performance of the approach as code fragments belonging to a clone class are present in the training and test proportion of the dataset. For example, $fa \simeq fb$ and $fa \simeq fc$ might be in the train split while $fb \simeq fc$ might be in the test split. ASTNN (8) uses a group-wise Cross Validation (CV) where clone classes are entirely isolated either into the training or test proportion of the dataset. This represents a real-world situation where first the detector is fitted and then applied to a new system with unknown code fragments.

Detectors that report lines instead of methods may produce more results (TP, FP, TN, FN) than present in the dataset. Similarly, the ASTNN Leaky uses CV which also produces different quantities as the approach is evaluated repeatedly on different proportions of the dataset.

7.5.6 Experiment Results

Creating the PSM model with Gradient took 2134.38 s, resulting in an average modeling time of 19.75 s for the 195 executables. This includes 87 helper methods.

Table 7.2: Results of the top-5 and bottom-1 experiment along with the average performance of the top-5.

Nr	Controlled Variables				Response Variables									
	Evaluation	D-FPR	M-FPR	Pooling	Duration	TP	FP	TN	FN	Skip	Precision	Recall	F1	MCC
1	skip	0.100	0.001	soft	1560	437	0	5320	21	345	1.000	0.954	0.977	0.975
2	skip	0.010	0.001	soft	1620	437	0	5320	21	345	1.000	0.954	0.977	0.975
3	exhaustive	0.010	0.001	soft	1680	425	0	5320	33	0	1.000	0.928	0.963	0.960
4	skip	0.010	0.010	soft	1920	423	0	5320	35	332	1.000	0.924	0.960	0.958
5	exhaustive	0.100	0.001	soft	2040	421	0	5320	37	0	1.000	0.919	0.958	0.955
16	exhaustive	0.100	0.010	hard	2820	293	0	5320	165	0	1.000	0.639	0.780	0.787
1-5	skip	0.010	0.001	soft	1740	428	0	5320	29	340	1.000	0.936	0.967	0.965

Duration in seconds

Table 7.3: Performance breakdown of the best performing experiment listed as Nr. 1 in Table 7.2.

Stage	Duration	TP	FP	TN	FN	Precision	Recall	F1	MCC
initial	–	458	5320	0	0	0.079	1.000	0.147	–
static	0.0001	458	1504	3816	0	0.233	1.000	0.379	0.409
dynamic	0.208	451	50	5270	7	0.900	0.985	0.941	0.936
model	1.749	437	0	5320	21	1.000	0.954	0.977	0.975
	0.344	437	0	5320	21	0.996	0.954	0.977	0.975

Duration in seconds

Table 7.2 contains the aggregate results of the top-5 experiments along with the results of the worst experiment. The bottom line in Table 7.2 is the average performance of the top-5 experiments. The generally expected performance of the approach *very strong* with an MCC of 0.965. High confidence for negative examples is given with no false-positives reflecting the pipelines FPR rates ($D\text{-FPR} \times M\text{-FPR}$).

A multiple linear regression was calculated to predict MCC based on the control variables: evaluation, D-FPR, M-FPR, and pooling. A significant regression equation was found shown in Table 7.4 ($F(4, 11) = 39.8, p < 0.01$), with an R of 0.911. MCC increased by 0.056 for the *skip* evaluation and by 0.080 for *soft* pooling. Larger D-FPR and M-FPR decrease the MCC by -0.042 and -5.455 . All predictors were significant ($p < 0.01$) except for D-FPR.

The best experiment is given with a *skip evaluation*, *0.100* D-FPR and *0.001* M-FPR rates, and *soft pooling* (Nr. 1) with an MCC of 0.975. A total of 345 candidates were skipped while reaching a recall of 0.933. Table 7.3 lists the cumulative performance of the best model, starting with an initial prediction that all candidates are semantic clones (rejecting pipeline). The *static* stage finds 71.729 % (3816) of the FPs improving the MCC by 0.409. The *dynamic* stage additionally removes another 27.330 % (1454) of FPs but introduces 1.528 % (7) of the possible FNs. An improvement of the MCC by 0.527 is

Table 7.4: Predicting MCC given the controled variables.

	<i>Dependent variable</i>	
	MCC	Sig.
Evaluation Strategy - greedy	0.040 (0.011)	***
Dynamic FPR	−0.094 (0.123)	
Model FPR	−7.152 (1.232)	***
Model Pooling - soft	0.104 (0.011)	***
Constant	0.858 (0.014)	***
Observations	16	
R ²	0.925	
Adjusted R ²	0.898	
Residual Std. Error	0.022 (df = 11)	
F Statistic	34.002*** (df = 4; 11)	

*p<0.1; **p<0.05; ***p<0.01

achieved via the dynamic stage. Finally, the *model* stage removes the remaining 0.939 % (50) FPs but introduces additional 3.056 % (14) FNs. The model stage improves the MCC by 0.039.

The worst experiment is given with *exhaustive evaluation*, D-FPR of 0.100, M-FPR of 0.010, and *hard pooling* (Nr.16) with a *strong* MCC of 0.787.

On average 5.884 % (340) of the total 5778 evaluations could be skipped. This equals 74.235 % of the total 458 TPs. On average 37.359 % (50 354) of the total 134 782 IO pair evaluations could be saved via greedy evaluation. The average duration of the exhaustive experiments was 2394s leading to 414ms per candidate. Skip experiments lasted on average for 1988s with 344ms per candidate. The static stage lasted on average for <0.001 %, the dynamic stage for 0.106 %, and the model stage for 0.893 % of the time per candidate (see Table 7.3).

Table 7.5 lists the detection results of eight alternative clone detectors.

Simian, NiCad, and CCAaligner found no clones in the dataset. PMD, SourcererCC, OreO, and iClones found some clones (< 20) but still have low recall (4 %). Nevertheless, each mentioned detector has a *very weak* performance below an MCC of 0.20. ASTNN 3-Group CV has a *strong* performance with an MCC of 0.711. ASTNN with the leaky evaluation has a *very strong* performance with an MCC of 0.976. A 3-fold grouped cross validation of ASTNN resulted *strong* performance of an MCC of 0.711 ± 0.15 . The longest computational duration is given by ASTNN with 1034min.

Table 7.5: Detection results of other clone detectors on the dataset.

Nr	Tool	Note	Repr.	Type	Duration	TP	FP	TN	FN	Precision	Recall	F1	MCC
1	Simian [76]		Text	1	0.138	0	0	5320	458	–	0.000	–	–
2	NiCad [131]		Text	3	1.291	0	0	5320	458	–	0.000	–	–
3	CCAligner [132]		Token	3	1.109	0	4	5316	458	0.000	0.000	–	-0.007
4	PMD [66]		Token	2	1.389	8	12	5308	450	0.400	0.017	0.033	0.069
5	SourcererCC [133]		Token	3/4	36.86	10	0	5320	448	1.000	0.021	0.042	0.142
6	Oreo [134]		Model	3/4	79.00	17	5	5315	441	0.772	0.037	0.070	0.158
7	iClones [135]		Token	3/4	0.980	13	0	5320	445	1.000	0.028	0.055	0.161
8	ASTNN [130]	3-Group CV	Model	4	1034	296	29	1415	162	0.911	0.646	0.756	0.711
9	ASTNN (Leaky)	Random Split	Model	4	2028	442	4	5316	16	0.991	0.965	0.978	0.976
10	SCD-PSM	Top 1-5	Model	4	1740	428	0	5320	29	1.000	0.936	0.967	0.965

Duration in seconds

7.6 Discussion

The goal of the study was to provide evidence of whether behavioral equality of model elements generalizes to semantic equality of code elements (RQ1). Furthermore, we were interested in the skip evaluation and its performance implications (RQ2 and RQ3).

7.6.1 Research Question 1 — Detection Performance

Table 7.2 and Table 7.3 present strong results in favor for RQ1. The MCC for the top-5 experiments was *very strong* with all MCCs being above 0.9. Even the worst experiment still yielded a *moderate* performance of 0.749.

Table 7.5 provides additional context to the results by presenting the detection results of alternative clones detectors. As expected, tools relying heavy on the textual representation of clones have very low recall (Simian, NiCad, CCAligner, PMD). Most clones found by the alternative tools are related to implementations that span only couple of lines. iClones however, also finds large clones that include many array accesses and manipulations. ASTNN is the best comparison tool and finds many clones with good precision. The approach is sensitive to hyper-parameters and to the training and test split. This can in the worst case lead to a test performance close to an MCC of 0. The low recall for Type 1-3 detectors indicate the high quality of the dataset. The moderate recall for Type 3/4 detectors indicate the high quality of SCD-PSM. Given this evidence, we conclude that RQ1 holds.

RQ1 — Behavioral equality between model elements generalizes to semantic equality of code elements allowing for semantic clone detection via probabilistic software modeling.

7.6.2 Research Question 2 — Skip Evaluation Scalability

The goal of the static and dynamic stage is to reduce the number of evaluations that the model stage must conduct. Each stage incurs an increasing cost of evaluation per candidate, with the model stage taking the largest share of 89 % of the evaluation time. Every TP has to pass the model stage to be declared a clone (rejecting pipeline). The skip evaluation avoided, on average, the re-computation of 74 % (340) of the TP candidate pairs. The greedy evaluation avoided, on average, the evaluation of 37 % of IO pairs. This offloads most of the evaluation time to the earlier stages that are computationally inexpensive while shortcutting the model stage. In comparison to the alternative detectors SCD-PSM needs substantial more time to compute (1.32 min vs. 29 min). An exception is ASTNN which has a similar runtime as SCD-PSM. Most of the runtime of SCD-PSM is caused by the operational overhead, e.g., loading the model from the database. Optimizing this overhead, as a theoretical maximum, could reduce the overall runtime of SCD-PSM on the dataset to 6.49 min given the average durations for each stage in Table 7.3. In conclusion, the skip evaluation reduces the number of model evaluations that are responsible for most of the evaluation time to a quarter.

RQ2 — Skip evaluation reduces the number of evaluations for the most expensive stage (model) in the SCD-PSM pipeline significantly.

7.6.3 Research Question 3 — Skip Evaluation Effects

Skip evaluation can cause cascading errors given an FP. Once an FP is introduced, every semantic clone related to the FP has a chance to become an FP in the same (wrong) clone class itself. These cascading FPs are potential sources of significant performance degradation. Skip evaluation experiments are ranked higher and are significantly better than experiments that conducted an exhaustive search. However, the absolute performance gain is merely a MCC of 0.056 hinting a per-chance significance introduced by the small sample size (16 experiments). Nevertheless, given the evidence in Table 7.2 and Section 7.5.6, we can conclude that skip evaluation does not affect the performance of the detector.

RQ3 — The skip evaluation has no negative impact on the performance of the detector given low false-positive rates.

In conclusion, SCD-PSM can detect semantic clones with high precision and recall.

7.7 Limitations

SCD-PSM inherits the limitations of PSM.

PSM only models data in an application and not references. References are changing addresses with no relation to the running program. Hence, they have no meaningful underlying distribution that can be modeled. However, once references are dereferenced, e.g., by accessing a field, their accessed data will be part of the model. Once part of the model, it can be used in the context of SCD-PSM. Nevertheless, algorithms with the sole purpose of manipulating and comparing references are unfit for SCD-PSM.

PSM explodes lists into singular values since distributions do not contain any order information. This means, executables that change the order of sequences are matched based on the values, not their order. As a consequence, an ascending and descending sorting algorithm would be detected as a false positive. Extending PSM to distributions of sequences alleviates the issue but is not a trivial task.

Another limiting factor is that the approach needs to run the program to build the model. This means that SCD-PSM can only be applied to runnable programs.

Finally, SCD-PSM cannot detect Type 2-3 clones since textual similarities represent a different problem set. A constructive proof is given by applying a textual and a semantic clone detector on Listing 7.1 *fa* and a copy *fx* where Line 4 has a division instead of a multiplication symbol. Most textual detectors will ignore this little change reporting $fa \simeq fx$. Semantic clone detectors would report $fa \not\simeq fx$ since the behavior changed (quotions of quotations instead of products of products). Inversely, we can add any random number of syntactically correct statements to a copy of Listing 7.1 that do not change the input and output behavior, e.g., adding `(1 * 2 - int("2") * math.ceil(0.56879))` at Line 5. Such behaviorally neutral statements may be added in any number to any statement. Textual clone detectors will at some point report $fa \not\simeq fx$ because the text fragment diverges too far from its original. However, semantic clone detectors will report $fa \simeq fx$ as long as the added statements do not influence the original behavior.

7.8 Threats to Validity

A threat to validity is given by the code competition dataset (Google Code Jam) that contains hard optimization problems that are not found in this density in large systems. However, this evaluation strategy grows in popularity in recent years [136, 137, 138]. This growth is partially motivated by the raising evidence and criticism of opportunistic evaluations, i.e., detecting clones in random open-source projects and letting human oracles (often the authors themselves) decide on the correctness of clones [105, 139, 140, 141, 142, 1, 143]. Only the source code developers can decide whether a clone is a clone. Any external reviewer can only hypothesize about the decision and process that created the clone. Many latent factors can contribute to this process. For example, test-, debug-, prototype-, or dead-code, deliberately introduced clones, or code that is related to structural aspects of a system may not be considered a clone or do not warrant any additional resources. To worsen things, the true recall is unknown for a given system even if it would be possible to inquire the developers of the code for their reasons of cloning. Given all these uncertainties, we decided to evaluate the performance in a controlled laboratory environment. Finally, we decided against the use of BigCloneBench [120] as

the benchmark does not contain compilable sources (only text fragments) nor do the semantic clones follow the 0% syntactic similarity target of this work.

7.9 Related Work

We started this chapter by defining what *semantic clones* means in the context of our approach (Section 7.3). While our definition is motivated in the capabilities of our approach, we can see strong similarities to the definition of Juergens [118]. Both definitions define behavioral similarity via IO relationships. Also, Juergens already discussed a notion of partial and conditional similarity. This understanding of Type 4 clones can be seen in multiple more recent studies [144, 145, 146]. In that, we see the progress of the community in terms of Type 4 clones as the definition becomes more specific.

Many studies evaluated textual clones. However, only a few studies are reporting results on semantic clones without relaxing the definition of Type 4. Rattan [78] et al. provided a review of clone detection studies. The review also investigated approaches that tackle Type 4 clones. They conclude that some approaches solve approximations (i.e., complex Type 3 clones) of Type 4 clones.

Test-based methods randomly trigger the execution of candidates and measure whether equal inputs cause similar outputs. Jiang and Su [147] were able to find semantically equivalent methods without any syntactical similarities. A similar approach was presented by Deissenboeck et al. [145]. One issue with test-based clone detection is that candidates need a similar signature. Differences in data types or the number of parameters and conditional dependencies can not be effectively handled by the test-case generators or the similarity measurement. SCD-PSM works similarly to test-based methods in that it observes the runtime and compares the resulting behavior. However, SCD-PSM builds generative models from the observed behavior capable of generating and evaluating data. Missing dimensions are imputed by conditioning and sampling via the models. This conditioned sample is then evaluated, allowing SCD-PSM to overcome the issue of signature mismatches. Furthermore, PSM abstracts the data types into text, integer, and floats mitigating data type mismatches.

Zhao and Huang [148] developed DeepSim, which learn control and data flow encodings that are then used in a binary classification task. DeepSim uses neural networks to learn a latent representation of the control and data flow. PSM also uses neural networks (each model element) but learns an underlying representation of the runtime. Hence, the most significant difference is in *what* is modeled (static vs. runtime information). DeepSim also evaluated on a Google Code Jam dataset however used the problems from 2016. DeepSim reached an F1 score of 0.76 on the GCJ 2016 competition, while SCD-PSM reached 0.967 on the GCJ 2017. While not entirely comparable, we expect these results to be a good approximation since the type of problems match.

7.10 Conclusions and Future Work

In this chapter, we presented Semantic Clone Detection via Probabilistic Software Modeling (SCD-PSM). PSM builds a Probabilistic Model (PM) from a program that can be used to generate or evaluate runtime events. We used these PMs to detect semantic clones in programs that have 0 % syntactic similarity.

We discussed the representation, search space, static-, dynamic-, and model-similarity that form the main aspects of SCD-PSM. Our study evaluated SCD-PSM in great detail, with an average MCC greater than 0.9. Also, the study showed the capability to control the false-positive rate, which is a significant point for industry adoption. Finally, we concluded that behavioral equality of model elements generalizes to semantic equality of code elements.

Our future endeavors will focus on constructing a comprehensive semantic clone detection benchmark. Furthermore, we see the future of semantic clone detection in enabling fault localization applications [2].

8 Case Study: Fault Localization

Abstract:

Software testing helps developers to identify bugs. However, awareness of bugs is only the first step. Finding and correcting the faulty program components is equally hard and essential for high-quality software. Fault localization automatically pinpoints the location of an existing bug in a program. It is a hard problem, and existing methods are not yet precise enough for widespread industrial adoption. We propose fault localization via Probabilistic Software Modeling (PSM). PSM analyzes the structure and behavior of a program and synthesizes a network of Probabilistic Models (PMs). Each PM models a method with its inputs and outputs and is capable of evaluating the likelihood of runtime data. We use this likelihood evaluation to find fault locations and their impact on dependent code elements. Results indicate that PSM is a robust framework for accurate fault localization.

Authors:

H. Thaller, L. Linsbauer, A.Egyed, and S. Fischer

Parts of this chapter was published at the 3rd International Workshop on Validation, Analysis, and Evolution of Software Tests (VST), London, ON, Canada (2020) [3].

8.1 Introduction

Modern software development aims to design and control the quality of software. Testing techniques, such as unit, integration, or system testing, and their automation via continuous integration, provide a feasible and generally applicable approach for software quality assurance. Software testing aims to find faults in a program. However, tests can not localize the faults within a program's source code. This is no issue for unit testing since the tests are small enough (typically methods). However, fault localization for integration and system tests can become a time-consuming task.

Fault Localization (FL) is the task of automatically finding faults in a program such that a developer or an automated process can repair them. Finding a fault, i.e., the real cause of an error, is a hard problem. Not only is it difficult to distinguish a symptom (cascading error) from a cause (actual fault), but also multiple faults can work in conjunction, complicating the localization process. The state-of-the-art FL techniques like Spectrum-based Fault Localization (SBFL) [149, 150] traditionally rank statements by their likelihood of containing a fault. This leads to localization weaknesses for complex faults that span multiple lines [13] (76% of faults) or that are caused by the omission of statements (30% of faults) [12, 151].

We propose Fault Localization via Probabilistic Software Modeling (FL-PSM). PSM builds a network of Probabilistic Models (PMs) of the executables (e.g., methods in Java) in a program. We use the PMs built by PSM to locate the most likely fault location. FL-PSM is a dynamic approach using either the test-suite (such as test-based FL techniques) or the actual execution of a program to fit each PM. PSM uses runtime data to construct behavioral datasets with which it fits the PMs. Then, runtime data from another program version, or failing tests, are used to find the most likely fault location.

8.2 Running Example

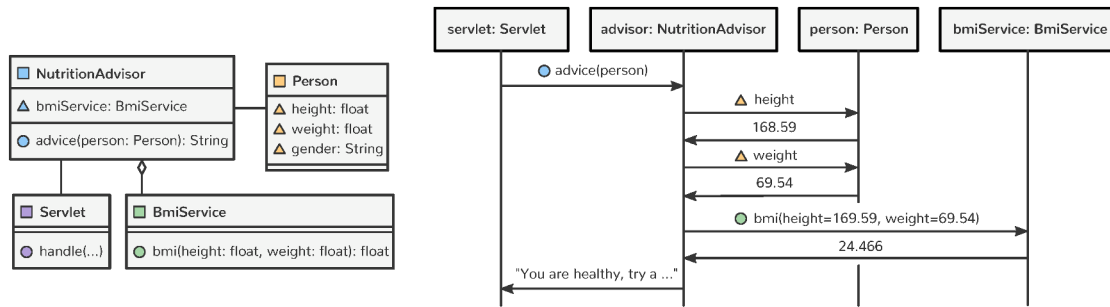


Figure 8.1: Class Diagram (left) and Sequence Diagram (right) of the Nutrition Advisor.

We use as an illustrative example the *Nutrition Advisor* that takes a person's anthropometric measurements (e.g., height and weight) and returns a piece of textual advice based on the Body Mass Index (BMI). The class diagram in Figure 8.1 (left) shows the four classes of the Nutrition Advisor. The sequence diagram in Figure 8.1 (right) shows a possible runtime trace of a request handled by the program. The `Servlet` handles requests (`handle()`) and initializes a `Person` object. This `Person` object is received by the `NutritionAdvisor.advice`-method that extract the person's height (168.59) and weight (69.54). Both values are the parameters for the `BmiService.bmi` call that returns the BMI (24.466) with which a textual advice is returned ("You are healthy, ...").

8.3 Background

Probabilistic Software Modeling describes a methodology for transforming a program into a network of probabilistic models. It extracts a program's structure represented by properties, executables, and types (fields, methods, and classes in Java) along with their call dependencies to build a network of probabilistic models. Every node in the network is a local probabilistic model (i.e., a NVP [30]) that represents an executable. Each local PM in the network is optimized towards a program execution. These execution traces are extracted from the system in its production environment or triggered via tests.

Each local PM represents an executable (e.g., a Java method) in the program. Inputs are parameters, property reads, invocation return values, while outputs are the method return value, property writes, and invocation parameters. The distinction between inputs and outputs exists only on a logical level for the program. However, the models themselves are multivariate density estimators (unsupervised models) with no notion of input and output (joint model of all variables). Each model can generate new observations that are similar to the initially trained data, e.g., to generate likely or rare (but plausible) test data. Furthermore, each model can evaluate the likelihood of a given observation (e.g., to evaluate the adequacy of given test data). This evaluation is relative to the runtime trace that was used to fit the model, e.g., a model based on production runtime will evaluate observations differently than a model based on tests.

Local PMs in this work are Non-Volume Preserving Transformations (NVPs) [30, 152], which are general and expressive flow-based density estimators. Each NVP is built via a neural network that learn a function that maps latent random variables (e.g., Gaussian variables) to the data (runtime events). Evaluating the likelihood with NVPs is done by transforming the runtime events into the known Gaussian latent-space and computing the Gaussian likelihood of the transformed events. More details on PSM and NVPs are given in the previous Chapter 4 and Dinh [30, 152].

8.4 Approach

FL-PSM is built upon PSM. The fault localization is based on the likelihood evaluation of these models. Given is a null-model M^{null} of an executable and either an alt-dataset D^{alt} of runtime events or an alt-model M^{alt} with which a dataset is generated. FL-PSM localizes faults by computing the mean log-likelihood of D^{alt} on M^{null} and comparing it to a critical value. More specifically,

$$LL_{D^{alt}} = \frac{1}{N} \sum_i^N p_{M^{null}}(D_i^{alt}) \quad (8.1)$$

computes the average log-likelihood where N is the number of data points in D . Finally,

$$LL_{D^{alt}} - LL_{D^{null}} < c \quad (8.2)$$

evaluates whether there exists a significant difference between model and data. $LL_{D^{null}}$ is the log-likelihood of M^{null} to itself and captures the inherent bias. The critical value c controls for Type-1 errors (false-positives) similar to other significance tests, e.g., $\log(0.001)$ indicates that 1 out of 1000 events is falsely considered to be significantly different from the model.

8.5 Preliminary Study

This preliminary study shows how FL-PSM finds possible fault locations. Given is the Nutrition Advisor to which 3000 requests are made based on data from the NHANES [153]

dataset. The resulting model is the null-model M_{null} . Then we seeded two errors in the Nutrition Advisor and collected the alt-datasets $D1_{alt}$ and $D2_{alt}$. The first error simulates a regression (between versions) caused by a typo in the `Person` constructor that assigns `-weight` instead of `weight` to the field. The second error simulates an integration fault (within version) caused by the miscommunication between teams using different measures. Team A that also built the *null* Nutrition Advisor, computes the BMI in meters while Team B that revises the implementation computes the BMI in inches.

We used the computation from Section 8.4 with a critical value (i.e., false-positive rate) of $c = \log(0.001) = -3$. This means log-likelihoods below -3 are significantly diverging from the model.

8.5.1 Regression Fault

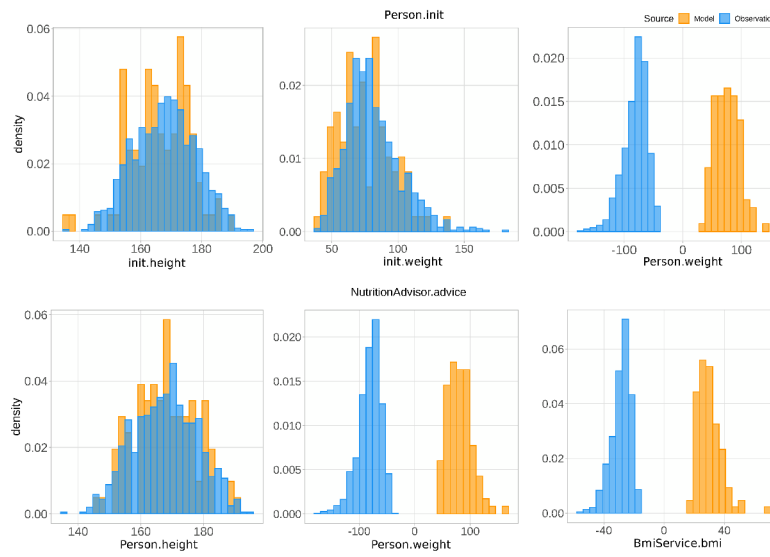


Figure 8.2: A subset of elements in the regression fault setting. For example, the first row shows *Person.init* parameters and property writes between the original version and a regressed version of the same component.

Figure 8.2 shows the runtime behavior of a subset of code elements of the *Person.init* and *NutritionAdvisor.advice* models. Table 8.1 lists the likelihood and significance of these elements along with the multivariate model likelihood that considers all elements at once. The visualization of the code elements allows developers to see that there is a significant difference between the model and the observations. The constructor parameter *init.weight* is aligned with the model while the property writes to *Person.weight* are clearly different. This difference is also significant as Table 8.1 shows (rows 1 and 4). Other elements in the same model are insignificantly different as both the visualization and the table show.

The difference propagates to the depending *NutritionAdvisor.advice* method that reads

Table 8.1: Likelihood values of a subset of elements in the regression fault setting.

	Model	Element	Cardinality	LL	Sig
1	Person.init	init	multivariate	-6787	✓
2	Person.init	init.height	univariate	-1.74	✗
3	Person.init	init.weight	univariate	-2.17	✗
4	Person.init	Person.height	univariate	-49.49	✓
5	NutritionAdvisor.advice	advice	multivariate	-4281	✓
6	NutritionAdvisor.advice	Person.height	univariate	-0.96	✗
7	NutritionAdvisor.advice	Person.weight	univariate	-97.23	✓
8	NutritionAdvisor.advice	BmiService.bmi	univariate	-82.67	✓

the *Person.weight* property (rows 5 and 7). Also, the invocation of the *BmiService.bmi* indicates this significant divergence (row 8).

8.5.2 Integration Fault

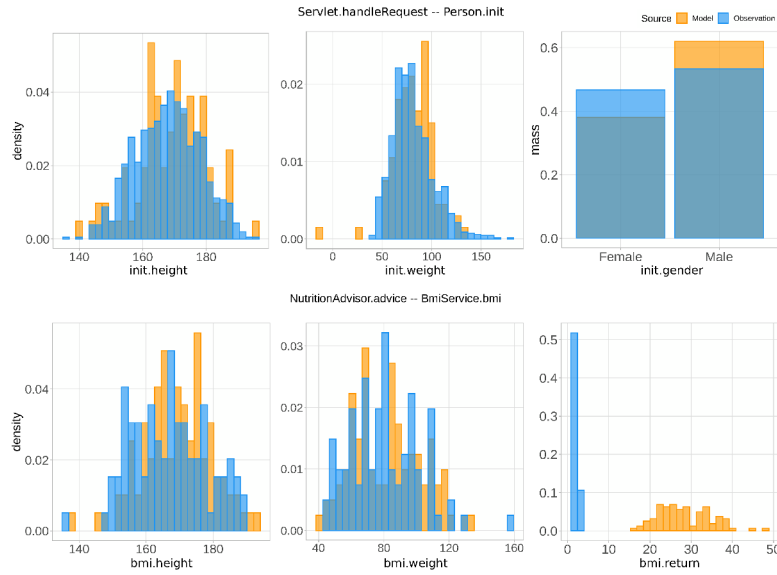


Figure 8.3: A subset of elements in the integration fault setting. For example, the first row shows *Person.init* parameter values caused by the invocation from *Servlet.handleRequest*.

Figure 8.3 shows the runtime behavior of a subset of code elements of the *Servlet.handleRequest* calling *Person.init*. In this case, *Servlet.handleRequest* model evaluates parameters or return values of *Person.init*. The visualization shows no significant difference in the integration between the *Servlet* and *Person*. This insignificance is also given in Table 8.2.

Table 8.2: Likelihood values of a subset of element in the integration fault setting.

	Model	Element	Cardinality	LL	Sig.
1	Servlet.handle	Person.init	multivariate	0	✗
2	Servlet.handle	init.height	univariate	-1.46	✗
3	Servlet.handle	init.weight	univariate	-2.33	✗
4	Servlet.handle	init.gender	univariate	-1.95	✗
5	NutritionAdvisor.advice	BmiService.bmi	multivariate	-6373	✓
6	NutritionAdvisor.advice	bmi.height	univariate	-0.95	✗
7	NutritionAdvisor.advice	bmi.weight	univariate	-0.50	✗
8	NutritionAdvisor.advice	bmi.return	univariate	-13.22	✓

The integration between *NutritionAdvisor.advice* and *BmiService.bmi*, with the first being the model, shows a difference in the return value of *BmiService.bmi*. Again, this difference is also reflected in Table 8.2 (rows 5 and 8).

8.6 Discussion

The preliminary study showed how FL-PSM localizes faults. This localization is automated via likelihood-based significance tests that allow for statistical control of the false-positive rate. The other important aspect is the visualization of the faults (Figures 8.2 and 8.3) and its impact on dependent elements. This allows for precise analysis of the error chain and its influence across the program.

FL-PSM can only be applied if there is at least a version of the program. This is not an issue from an industrial point of view since FL-PSM can be used after a few development sprints. Another consideration is that FL-PSM localizes behavioral changes, including intended changes. These intended changes can be filtered by incorporating source code change information in the localization process. In addition, the visualization capabilities of FL-PSM allow for quick manual inspections in cases of doubt.

In summary, the results and usability of FL-PSM are promising. Nevertheless, there are still open questions concerning multiple fault sources and their clear separation.

8.7 Related Work

Most fault localization techniques are slice, spectrum, statistics, model, or machine learning-based [149, 13].

The most similar technique to FL-PSM is Spectrum-Based Fault Localization (SBFL) [149]. SBFL techniques observe passing and failing executions and perform statistical inference on the results. The result is a ranked list of statements, along with their likelihood of being the fault location. While similar, FL-PSM works slightly differently in terms of the abstraction level and source model. PSM abstracts statements and only considers

properties, executables, and types along with their call dependencies. In contrast, SBFL techniques predominately work on the statement level. This might seem like a drawback at first. However, Parnin and Orso [151] identified that the detail of the results in combination with high false-positive rates are one of the main issues of the low industrial adoption of SBFL. PSM improves on these issues by providing control of the false-positive rate and its level of abstraction (executables).

8.8 Conclusion and Future Work

We presented Fault Localization via Probabilistic Software Modeling (FL-PSM). FL-PSM builds upon PSM and uses statistical inference to find possible fault locations in a program. The localization is based on evaluating the likelihood of runtime events under the model. We have shown how FL-PSM localizes and visualizes faults. In addition, we discussed the difference between FL-PSM and its close relative SBFL.

Future work will focus on a full evaluation of the approach with multiple complex subsystems. Furthermore, we want to conduct a user study for its practicality and applicability.

In conclusion, FL-PSM is a promising new FL approach built upon PSM that provides a general framework for probabilistic analysis of software programs.

9 Case Study: Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases

Abstract:

Source code and related artifacts of software systems encode valuable expert knowledge accumulated over many person-years of development. Analyzing software systems and extracting this knowledge requires processing the source code and reconstructing structure and dependency information. In analysis projects over the last years, we have created tools and services using graph databases for representing and analyzing source code and other software engineering artifacts as well as their dependencies. Graph databases such as Neo4j are optimized for storing, traversing, and manipulating data in the form of nodes and relationships. They are scalable, extendable, and can quickly be adapted for different application scenarios. In this chapter, we share our insights and experience from five different cases where graph databases have been used as a common solution concept for analyzing source code and related artifacts. They cover a broad spectrum of use cases from industry and research, ranging from lightweight dependency analysis to analyzing the architecture of a large-scale software system with 44 million lines of code. We discuss the benefits and drawbacks of using graph databases in the reported cases. The benefits are related to representing dependencies between source code elements and other artifacts, the support for rapid prototyping of analysis solutions, and the power and flexibility of the graph query language. The drawbacks concern the generic frontends of graph databases and the lack of support for time series data. A summary of application scenarios for using graph databases concludes the chapter.

Authors:

R. Ramler, G. Buchgeher, C. Klammer, M. Pfeiffer, C. Salomon, H. Thaller, and L. Linsbauer

Parts of this chapter was published at the International Conference on Software Quality (2019) [154]. Section 9.4.5 discuss an early technical realization of PSM called Gradient and represents our main contribution in this chapter. Additionally, we contributed to the findings presented in Section 9.5.

9.1 Introduction

The size and complexity of real-world software systems are continuously increasing. Today, many companies develop and maintain software systems containing hundreds or thousands of source code files encompassing up to several million lines of code. They often consist of a mix of various technologies and, additionally, a wide range of related software engineering artifacts such as tests, documentation, change requests, bug reports, and execution logs. Source code and artifacts encode valuable expert knowledge accumulated over decades of development. They represent complex structures and related information about various parts of the software system.

Nevertheless, the source code and a large part of the artifacts are stored in text files organized in conventional directory structures. Analyzing software systems requires parsing these files and reconstructing the structure and relationship information. This first step is typically the precondition for a further, more advanced analysis aiming at software understanding, interactive exploration, fault detection, visualization, and documentation. Advanced analysis applications benefit from infrastructure for processing and representing the structure of software systems in a scalable and extensible way.

In several projects developing tools and services for software analysis, we have created different implementations of such infrastructures using graph databases for representing source code, software engineering artifacts and their relationships. Graph databases [155] are NoSQL databases that support graph data models, i.e., data represented in the form of nodes connected via edges with each other. Graph databases are particularly useful if relationships between nodes are a central characteristic of the stored data. They are optimized for storing, querying, and manipulating vast amounts of highly connected data by native support for relationships and enhanced traversal capabilities. Hence, they are frequently used in a wide range of applications such as recommendation engines, social networks, collaboration platforms, and medical research systems.

The objective of this chapter is to collect and share our experiences with graph databases in representing and analyzing source code and software engineering artifacts. We describe five different cases related to different application scenarios and project contexts. Across all cases, the use of graph databases has emerged as a common element in the implemented solutions – yet with variations in how the data is modeled, stored and accessed. By comparing and discussing the different approaches, data models, and underlying design decisions, we provide insights into the advantages and disadvantages of graph databases for building analysis tools and services.

The remainder of the chapter is structured as follows. Section 2 provides an introduction to graph databases and outlines related work. The research design of our experience report is described in Section 3. Details about the five presented cases are described in Section 4. The discussion of identified advantages and disadvantages follows in Section 5. Finally, Section 6 concludes the chapter by summarizing the key findings and suggestions for future work.

9.2 Background and Related Work

A graph is composed of nodes and relationships. A node represents an entity (e.g., `class`, `method` or `variable`) and a relationship represents how two nodes are associated (e.g., a class `contains` a method, a method `calls` a method or `reads` a variable). Nodes and relationships can have properties that are usually specified in form of key-value-pairs (e.g., `name='foo'`). This general-purpose concept enables creating arbitrary connected structures that closely match the modeled characteristics of the problem domain [156].

A graph database is an online database management system with `create`, `read`, `update` and `delete` operations optimized for graph data models [155]. Graph databases are specifically designed to support fast and scalable management, storage, and traversal of nodes and relationships. This support allows to specify all relationships representing connections between entities at the time the data is created, and storing them persistently in the database. When the database is queried, these relations can be quickly traversed without the need to compute them dynamically via foreign keys and costly join operations.

The widespread adoption of NoSQL databases for many problem domains also led to the development of several databases specialized on graph data. Examples include AllegroGraph (Franz Inc.), InfiniteGraph (Objectivity Inc.), Neo4j (Neo4j Technology Inc.), and OrientDB (Callidus Software Inc.). A review and comparison of contemporary graph databases can be found in [157].

The cases we describe in this chapter use the graph database *Neo4j*¹. It is one of the most popular graph databases, also offered open source. Neo4j is based on a native graph storage and processing engine. It comes with the declarative graph query language *Cypher* that supports the definition, manipulation, and querying of graphs. Cypher queries can be issued using programming language specific drivers or the Web-based user interface *Neo4j browser*. In addition, Neo4j supports a programming language independent REST API and a low-level Java driver that can directly access database search facilities.

Many approaches and tools exist for analyzing software systems [158] and for performing queries on source code [159]. Most of these approaches and tools rely on database technologies to store structure and dependency information. Although the use of relational databases is still prevalent, NoSQL databases are receiving more and more attention. Zhang et al. [160] implemented a framework for querying heterogeneous code repositories using the document-oriented database MongoDB.

Graph databases have been applied in a few instances, by Yamaguchi et al. [161] for analyzing code to discover vulnerabilities, by Urma and Mycroft [162] for querying source code, by Goonetilleke et al. [163] to implement the tool Frappe for code comprehension, and in the open source tool jQAssistant² for ensuring code quality of Java programs. These applications are related to the cases described in our study. However, an analysis of large-scale software systems similar to what we present in our case study has only been described for Frappe [164].

¹<https://neo4j.com>

²<https://jqassistant.org>

9.3 Research Design

This chapter reports our insights and experiences gained from using graph databases in the form of a collection of individual cases. Each case provides a first-hand account by the authors, who have been personally involved in the reported cases and the development of the associated tools and services.

Research goals and questions: Besides describing *how graph databases are used*, the chapter explores *what are the advantages* and *what are the disadvantage* of using graph databases for representing and analyzing source code and related artifacts.

We conducted the following steps to provide answers to these questions.

1. **Case selection:** The key criteria for selecting a case to be included in our report were the use of a graph database, its application for supporting software analysis tasks, and the development of related tools or services.
2. **Case description:** We used a template with a uniform structure related to a set of open questions to describe the cases. The descriptions were prepared by the authors involved in the cases.
3. **Review of descriptions:** The case descriptions were reviewed by co-authors not involved in the case to assure that the descriptions are complete and consistent. Variations and extensions to the structure of the descriptions were introduced to capture individual aspects of the reported cases.
4. **Compile overview:** A table showing the essential characteristics of all reported cases was prepared for comparing the cases and for identifying their commonalities and individualities.
5. **Exploration of individual cases:** The cases were discussed and explored further w.r.t. design decisions, encountered challenges, open issues, and feedback from users. The findings were the basis for deriving a list of advantages, disadvantages, and lessons learned for each case.
6. **Synthesis of findings:** The findings from the individual cases were aggregated to high-level advantages and disadvantages as well as general lessons learned for presentation in the chapter.

Several measures were taken to mitigate threats to validity. We decided to select multiple cases with different characteristics to support generalization. The information about the cases was provided by authors who were personally involved. To reduce the resulting bias, we used a shared template for preparing case descriptions, which were then reviewed and discussed with authors not directly involved in the cases.

Nevertheless, analyzing cases does not always allow to accurately identify the boundary between the observed phenomenon and the context [165]. The advantages, disadvantages and lessons learned we identified in our work may therefore still depend on influence factors rooted in the specific project or application context. To counter this threat, we

decided to put the primary focus of this chapter on the presentation of the individual cases to support the reader in transferring the insights and experiences to his/her own context.

9.4 Description of Cases

The five presented cases (*Case 1-5*) cover a variety of different applications scenarios related to the analysis of source code and related artifacts. Each of them is based on a unique goal and motivation, derived from its application in industry projects³ or research. The common theme shared by all cases is the implementation of tool support and services for which graph databases have been applied. Table ?? provides an overview of the key characteristics of the five cases, summarizing the spectrum of different applications realized with graph databases.

In the subsections below, the descriptions of the cases are based on the following structure.

- **Project context:** In what application scenarios is the graph database used?
- **Data model:** What data model is used to represent the structure and relationships of the analyzed source code or software engineering artifacts?
- **Data sources:** How is the data created and imported into the database?
- **Access and usage:** How is the data in the database accessed and used?
- **Status and ongoing work:** What is the current state of the work and what are the next steps?

9.4.1 Case 1: AutoDoc for Lightweight Dependency Analysis

AutoDoc is a lightweight and flexible dependency analysis tool based on static code analysis for various programming languages.

Project context:

The tool has been developed in a project with industry partners from the domain of embedded systems. These systems have to fulfill high quality demands. The project supported the industry partners in improving software quality by providing unit testing and source code analysis technology. *AutoDoc* statically analyzes the source code of the software system and generates information about components and their interrelations. Furthermore, a set of code-related metrics is calculated including Halstead, McCabe complexity, and the Microsoft maintainability index. The results are used by developers in various ways, for example, to gain an overview of interdependencies between the units of a software system, to determine areas for refactoring and testability improvements, or

³If not already revealed in previous publications, details about involved industry partners have been omitted due to confidentiality obligations.

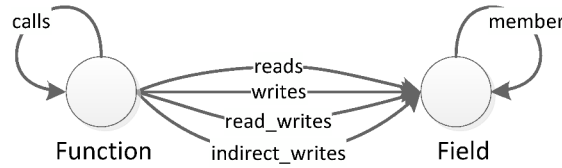


Figure 9.1: Data model of AutoDoc.

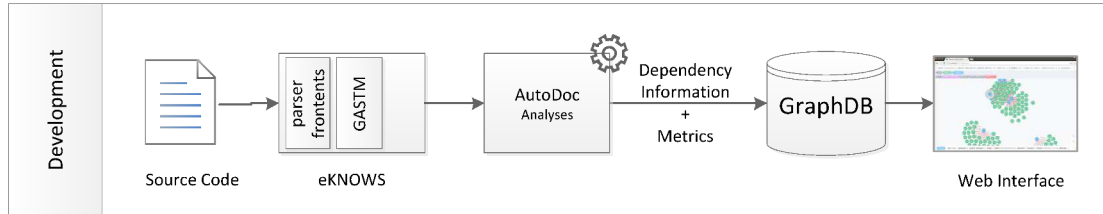


Figure 9.2: System overview AutoDoc.

to estimate the impact of architectural changes. By using a language agnostic parsing frontend, the tool is applicable for systems implemented in various different programming languages; currently, it is used to analyze systems implemented in C.

Data model:

AutoDoc stores dependency analysis results as graph into the Neo4j database. The data model is depicted in Figure 9.1. It models dependencies due to function calls and field access. The nodes are of type **function** or **field**. Relations of type **calls** exist between functions; **reads**, **writes**, **read_writes**, or **indirect_writes** relations between functions and fields. The more abstract **indirect_writes** relations show that a member of a field is written by a function. Metrics values, e.g., complexity measures, are stored as attributes for function nodes.

Data sources:

The foundation of the tool is the general-purpose code analysis platform part of SCCH's eKNOWS tool family [166], which is used in different projects in the field of static code analysis, domain knowledge extraction, and re-documentation. The platform provides parser frontends for numerous programming languages to transform the code into the Generic Abstract Syntax Tree Model (GASTM). The generic representation allows to make downstream transformations and additional analyses available for any of the supported programming languages. AutoDoc utilizes the C code parser frontend of eKNOWS to transform the code into its GASTM representation (Figure 9.2). Call graph and dependencies are calculated from this model. The resulting graph is stored in the Neo4j graph database. Analysis runs are triggered either manually on demand or periodically by integrating the tool into the build process.

Access and usage:

The data stored in the graph database is accessed via the standard Neo4j Web interface, which supports users in exploring the analysis results by writing queries or executing previously saved queries. We provide predefined queries to answer common questions with respect to system components, dependencies, and accessed variables, for example, “*How often are global variables accessed?*”, “*What global variables are written by different functions?*”, “*What functions access many different global variables?*”, “*What are the most complex functions?*”, or “*Which functions have a high maintainability index?*”. The typical target audience of AutoDoc are developers performing a detailed analysis of the system under development. We found that these users are comfortable with formulating ad-hoc queries using the easy to understand Cypher query language.

Status and ongoing work:

AutoDoc is currently used by our industry partners in two main ways, first, for exploring and refactoring of legacy code and, second, for continuous code quality assurance. While the Neo4j browser seems sufficient for the first use case, the generation of an analysis report is considered for the later. The currently analyzed software systems contain up to 72,000 lines of embedded C code. Nevertheless, it is planned to expand the application of AutoDoc to analyzing a system of systems with more than a million lines of code.

9.4.2 Case 2: SCoRe for PLC Programs

SCoRe is a static analysis tool developed explicitly for analyzing PLC programs written in IEC 61131-3 programming languages for industrial automation and production systems.

Project context:

The programming languages defined by the IEC 61131-3 standard [167] are used in industry to implement the control software of real-time systems. The software runs on dedicated hardware, i.e., programmable logic controllers (PLCs). Due to the focus on a relatively small niche, these software systems have received little attention in the past. Only a few software engineering tools are available, mainly proprietary programming environments tied to PLCs of specific vendors. However, the size and complexity of today’s industrial control systems increased the demand for additional tools supporting quality assurance, testing, software architecture, and design tasks.

Together with our industry partners, we developed a tool for automated static code analysis of large-scale PLC programs (c.f. [168], [169], [170]). The tool *SCoRe* (for *Source Code Review*) supports detecting a range of problematic code constructs, violations of programming conventions, and potential defects. In addition, we implemented support for analyzing the software design and architecture of control systems by exporting the structure and dependency information extracted in static code analysis available to the graph database Neo4j. This solution allows exploring and examining the various program elements and their dependencies via custom queries and by browsing the graph

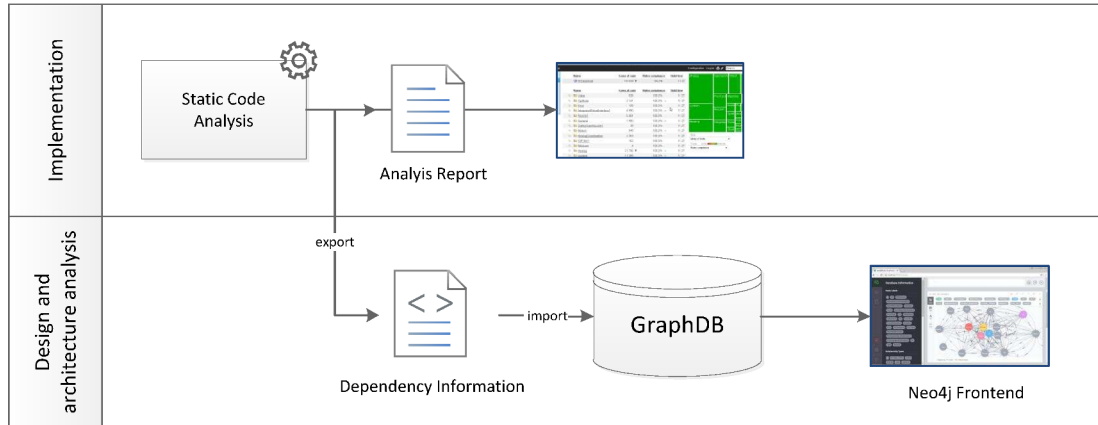


Figure 9.3: Overview and Usage of SCoRe.

visualization in the Neo4j Web interface. Figure 9.3 provides an overview of the tool chain. It distinguishes the tool’s application in *implementation* for automated static code analysis based on a set of predefined rules from *design and architecture analysis*, which requires support for interactively exploring and reviewing the software system.

Data model:

The Neo4j database contains 24 distinct node types that represent the various building blocks commonly used in PLC programs [167]. They are structured into **functional units** that contain **program units** (e.g., function blocks or functions) and define **system variables**. Program units can **call** other program units, **read** and **write** system variables, and to **send events** or **listen for events**. Instances of program units are assigned to process **tasks** for execution in cyclic intervals. Figure 9.4 provides an overview of the most important building blocks (nodes) and their dependencies (relations). Nodes can contain additional attributes such as the path to the source code files or optional metric values (e.g., number of imports, or fan-in and fan-out).

Data sources:

The SCoRe tool parses the PLC program, constructs the abstract syntax tree, the control flow graph, and the data flow graph. The predefined rules are used to detect issues in the source code, which are then listed as report and in SonarQube’s quality dashboard. As an add-on, the tool also exports the information about program elements and dependencies to CSV files, which can be imported into the Neo4j graph database. The imported files represent the snapshot of the system at the analysis time. The database is wiped before an import, as updates or incremental changes are currently not supported.

However, the data in the database can be enhanced after the import by computing additional attributes and relations not present in the initial dataset. Metric values are a typical example. They are computed with queries on imported data, and the results are

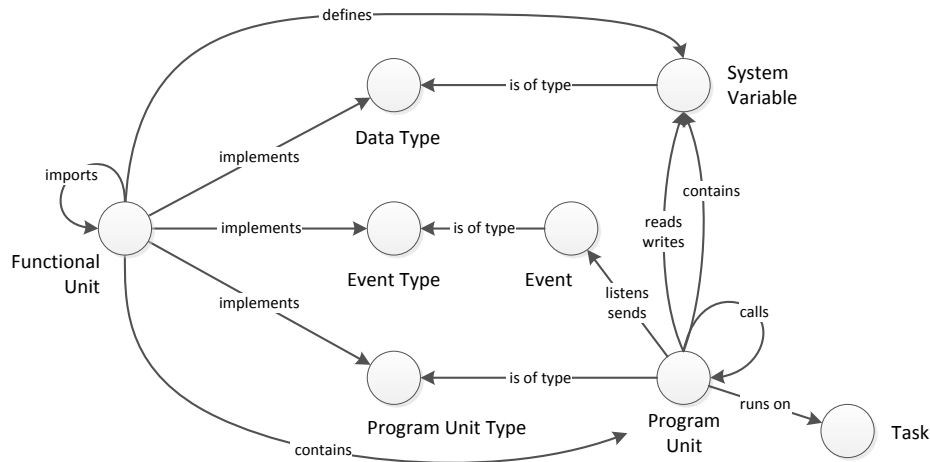


Figure 9.4: Data Model of SCoRe.

inserted as additional attributes to existing nodes. In that way, additional abstraction layers can be created, which can again be used in querying and exploring the system.

Access and usage:

The generic Neo4j Web frontend is used to for submitting custom Cypher queries and interactive exploration of result graphs. It can be easily accessed by all members of the development team as only a Web browser is required. Together with our industry partners we identified the following application scenarios for SCoRe: Evaluating the compliance of the implementation with design decisions and guidelines (e.g., use of global variables), computing metrics (e.g., coupling and cohesion), support for refactoring (e.g., identifying large program units), and analyzing the potential impact of changes (e.g., dependencies on changed elements).

Status and ongoing work:

The tool SCoRe is currently applied by two industry partners developing industrial automation systems. It has been used to analyze systems up to 742 KLOC implemented in the IEC 61131-3 programming languages, which resulted in about 450,000 nodes and 2,500,000 relationships in the graph database. The import into Neo4j required 21.4 seconds on a standard desktop computer. We are currently working on extending the tool to support the simultaneous analysis of IEC 61131-3 and C/C++ as our industry partners are also using a combination of both technologies in PLC programs.

9.4.3 Case 3: eKNOWS Code Model Service

The *eKNOWS Code Model Service* (eKNOWS CMS) is a service that provides reusable static code analysis functionality for Java programs via a dedicated REST API.

Project context:

eKNOWS CMS has been developed as the foundation of a microservice-based system for extracting architectural information from large-scale service-oriented software systems via static code analysis [171]. The system has been developed in close cooperation with Raiffeisen Software GmbH (RSG), a provider of IT solutions for the finance domain in Austria.

System overview:

Figure 9.5 depicts an overview of the eKNOWS CMS. As shown in the figure, eKNOWS CMS is implemented as a microservice that provides static code analysis functionality to a set of other microservices and tools. These analyses are provided via a dedicated Representational State Transfer (REST) API. Static code analysis in the eKNOWS CMS differs from many other code analysis approaches where analysis is performed via abstract syntax tree (AST) visitors for deriving information from the system implementation. Instead, we have implemented static code analysis by means of Cypher queries. We provide the following kinds of analyses:

- Search for type and interface declarations of a specified module.
- Search for type, field, and method declarations with specified metadata.
- Search for extended types and implemented interfaces of a specified type declaration, search for all type declarations derived from a specified type, and search for all type declarations implementing a specified interface.
- Search for import relationships of a specified module, and search for modules importing a specified module.
- Search for method declarations of types and interfaces.
- Calculation of call graphs and caller graphs for specified method declarations
- Search for XML documents and elements and attributes of XML documents
- Search for MANIFEST files and their attributes
- Calculation of type dependency relationships.

On top of the eKNOWS CMS we have developed a set of services that use the provided code analysis functionality. These services then provide information to different tools used at RSG. A detailed description of the developed services and used tools can be found in [171].

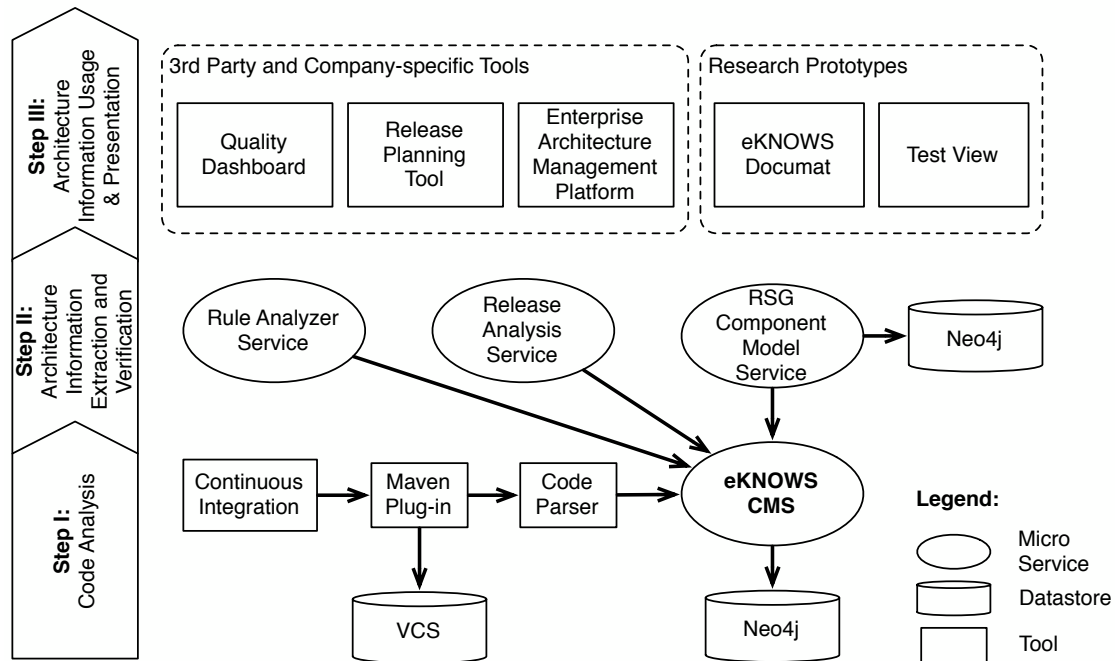


Figure 9.5: eKNOWS Code Model Service - System Overview

Data model:

Figure 9.6 depicts an excerpt of the data model of the eKNOWS CMS. The data model consists of 83 different node and 88 relationship types that are used for storing complete **implementation artifacts** (i.e., source code, XML, and Manifest files) in Neo4j by converting these artifacts into graph structures. All implementation artifacts are assigned to a dedicated **module**, which is the unit of versioning and deployment. Modules define dependencies to other modules via **import relationships**. The eKNOWS CMS can store multiple **versions** of the system implementation in Neo4j, i.e., we store all released versions of a module along with the version of the current development iteration that is overwritten whenever a cyclic build process is triggered. Modules can be aggregated to **applications** to describe modular systems. Finally, we also store the results of resource-intensive analyses, i.e., call graphs and dependencies between types in Neo4j to avoid redundant analyses.

Data Sources:

The eKNOWS CMS operates on implementation artifacts that are written to Neo4j as part of cyclic build processes. A dedicated Maven Plug-in fetches the system implementation from version control systems (VCS) and stores this data in Neo4j (see Fig. 9.5).

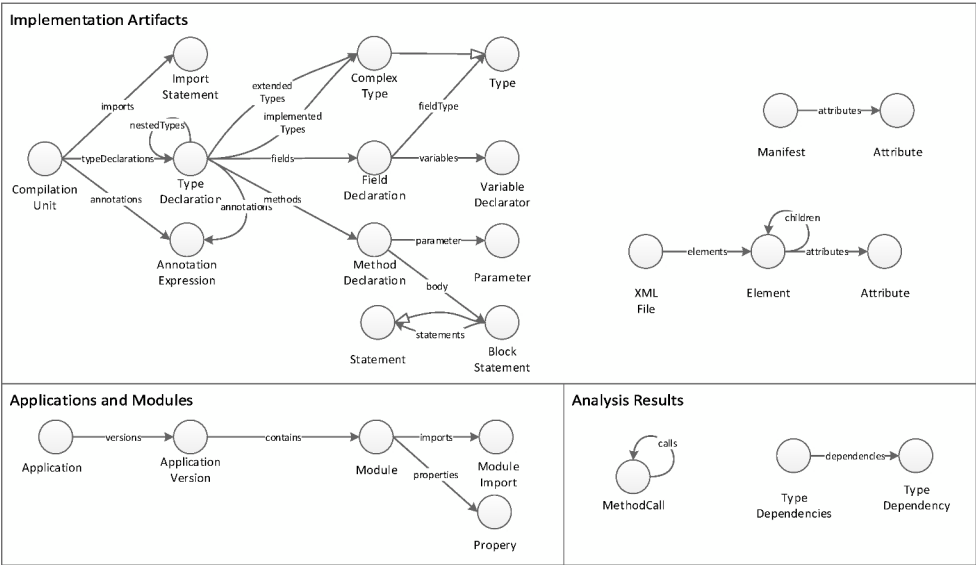


Figure 9.6: eKNOWS CMS Data Model (Excerpt).

Status and ongoing work:

eKNOWS CMS has been successfully evaluated in an industrial case study at RSG in which we have analyzed the entire codebase of RSG's latest online banking solution (see [171]). We have analyzed over 44 million lines of code, which were stored as 138,595,573 nodes and 138,141,947 relationships in Neo4j. The correlation between the number of nodes and relationships results from the fact that we store implementation structures 1:1 in the database without relationships between implementation artifacts. Such relationships are either calculated on demand (e.g., inheritance and implemented interfaces relationships), or they are stored as dedicated data structures (e.g., call graphs and type dependencies) where references are defined via node IDs.

Currently, we are working on supporting additional use cases of stakeholders at RSG by providing corresponding architecture information using the analysis functionality of eKNOWS CMS. We will further extend the eKNOWS CMS with additional kinds of analyses to improve our support for automated generation of viewpoint-based software architecture documentation.

9.4.4 Case 4: Sherlock for Regression Test Case Selection

Sherlock is a tool that supports regression test case selection in manual system testing based on test coverage and code changes.

Project context:

Regression testing [172] is performed after making changes to an existing software system to ensure that these changes do not have unexpected adverse side effects on the behavior of existing, unchanged parts of the software system. The straightforward approach to regression testing is to re-execute all existing test cases to make sure they still pass. However, many software projects have a large number of test cases, and it is often impossible to re-execute all of them every time a change has been made. Regression test case selection aims at selecting a reasonably small subset of the existing test cases, which still has a high chance of detecting any issues introduced by changes.

We developed the tool *Sherlock* for selecting regression test cases based on a list of locations in the source code where changes have been made and the information which test cases cover these source code locations [173]. *Sherlock* specifically supports interactively selecting test cases for manual regression testing in the context of a large-scale software product by OMICRON electronics GmbH. This software product encompasses more than 30 modules (about 2.5 MLOC in total, mostly implemented in C++) that interact with each other and share a common framework as well as various base libraries and hardware drivers. The system has grown to its current size over a time span of more than two decades. Engineers in different roles (i.e., developers, architects) have contributed over time, creating a large and sophisticated software system with complex dependencies between application modules, framework components, custom interfaces, and various third-party libraries. Thus, today, one of the foremost challenges of effective and efficient

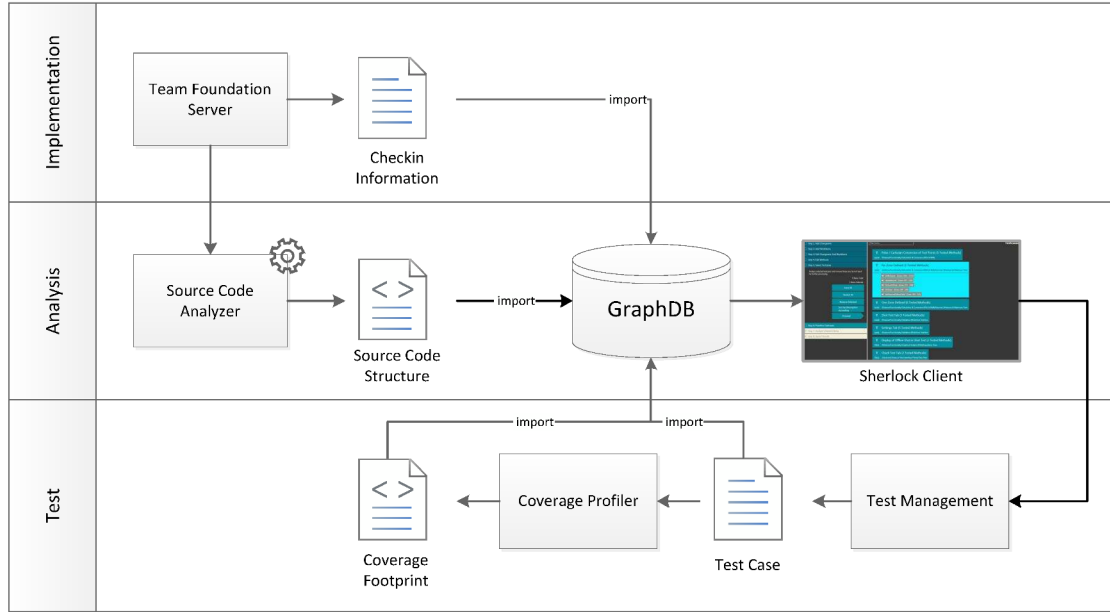


Figure 9.7: Sherlock System Overview.

regression testing lies in acquiring and managing the knowledge about the huge amount of dependencies in the software system.

Data sources:

As regression testing is a time-consuming activity [174], Sherlock helps to reduce required efforts and costs by concentrating on those tests, which exercise the parts of the system that are affected by changes. For identifying and selecting the relevant test cases, Sherlock incorporates information from three data sources (as shown in Figure 9.7):

1. Information about source code and changes (e.g., check-ins) is extracted from the *version control system* of Microsoft's Team Foundation Server (TFS).
2. The list of available test cases and their properties are retrieved from the *test management system* SilkCentral Test Manager.
3. The relationship between test cases and code changes is determined from coverage analysis results produced by the *profiler* SmartBear AQtime Pro.

Custom implemented adapters based on Neo4jClient binding for .NET are used to extract, transform and load (via a bulk Cypher import) this information into a *Neo4j graph database* that acts as Sherlock's central data store.

Data model:

Sherlock stores the data in a tool-agnostic graph format (Figure 9.8). Central information in the Sherlock data model are nodes holding information about all `methods` in the system

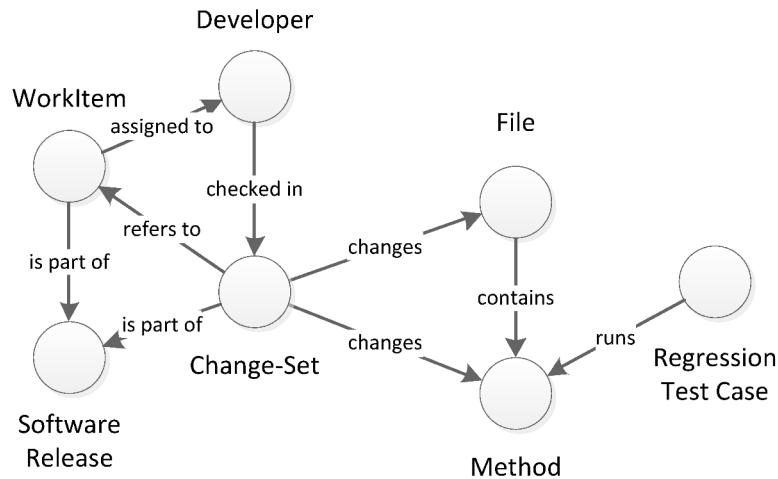


Figure 9.8: Sherlock Data Model.

under test, including details such as name or line numbers stored as attributes. Methods are grouped by **files** and are part of one or more **change-sets** (check-ins performed by a **developer**). A change-set in TFS may be connected to a **work item** that contains either a task, bug, or feature description, and which is assigned to a **software release**. A method is furthermore connected to a **regression test case** if it is part of the test's coverage footprint, i.e., if the method is called during the execution of this test case.

Access and usage:

Sherlock supports the testers in selecting the minimal set of regression tests to target the changes made by the developers. Therefore the testers access the aggregated information stored in the graph database using a rich client application.

For compiling a regression test suite, the client application implements several Cypher queries so that testers can identify test cases related to an individual change (e.g., a bug fix), for all changes within a specified time range (e.g., all fixes and enhancements combined in a maintenance release), or all changes made on a branch before it is merged back into the trunk (e.g., all changes made while implementing a new feature). The resulting set of test cases is the basis for subsequent regression test runs scheduled from within the test management system. Sherlock provides an export interface to update the test plans in the test management system accordingly. In addition, the tool also supports the analysis of the available test cases in general. For example, it indicates coverage gaps when changed source code is not covered by the set of selected test cases or any test at all.

Status and ongoing work:

Initially, the project started with test case selection based on static code analysis results [173], but experiments showed that in this case, static analysis approaches are not able to

reveal enough dependencies relevant for test case selection, e.g., because of multi-language interoperability or reflection mechanisms. For this reason, code coverage information was collected to expose dynamic dependencies as well. Today, check-in information and source code structure are kept up-to-date by a Windows service that runs on a daily basis. This service uses custom adapters to import 21,000 check-ins and more than 200,000 methods that are grouped in 20,000 files. Coverage foot-prints for currently 400 tests are imported and updated manually after a test case was successfully profiled.

Sherlock is a valuable aid for providing guidance in selecting appropriate regression test cases for testers who lack detailed knowledge of the structure and dependencies of the system under test. In an evaluation, we found that a junior tester using Sherlock was able to produce test suites with less or equal effort and at the same level of accuracy as highly experienced testers who accomplished the same tasks manually [175].

9.4.5 Case 5: Gradient for Probabilistic Software Modeling

Gradient is a Probabilistic Software Modeling (PSM) [176] system prototype that uses static and dynamic analysis to model the structure and behavior of a program.

Project context:

PSM systems allow engineers to inspect a program's structure (Types, Properties, Executables) *and* behavior (runtime objects) using statistical models. These statistical models can be used in applications such as visualization of runtime behavior (e.g., possible values of property `age` form a `Person` class), finding the most likely value combination of the parameters of an executable or test-case generation. The program structure is extracted via static code analysis while the behavior observations are extracted via dynamic code analysis. The static and dynamic information is then used to build a network of probabilistic models with similar behavior as the original program. The objective of Gradient is to empower software engineers with the possibility of behavioral analysis of programs without switching the level of abstraction (Types, Properties, Executables) or to content themselves with a single execution trace (e.g., debugging).

System overview:

Gradient leverages static and dynamic code analysis and builds a network of models that mirror the system under inspection. Naturally, it needs multiple stages, components, and technologies to work.

Figure 9.9 shows an overview of the Gradient system that is split into two parts, the client- and server-side, operating on three levels: Development, Runtime, and Modeling. First, ① the program structure is extracted from the *Source Code* and stored directly into the *Graph Database* (Neo4j) ②. Then the source code is compiled and patched with monitoring aspects that execute the monitoring logic. The *Patched Byte code* ④, containing additional monitoring logic, directly stores *Runtime Events* into the *Document Database* (MongoDB). This entire process is handled by the Gradient client which in addition reports (not shown in Figure 9.9) the analysis progress to the Gradient server.

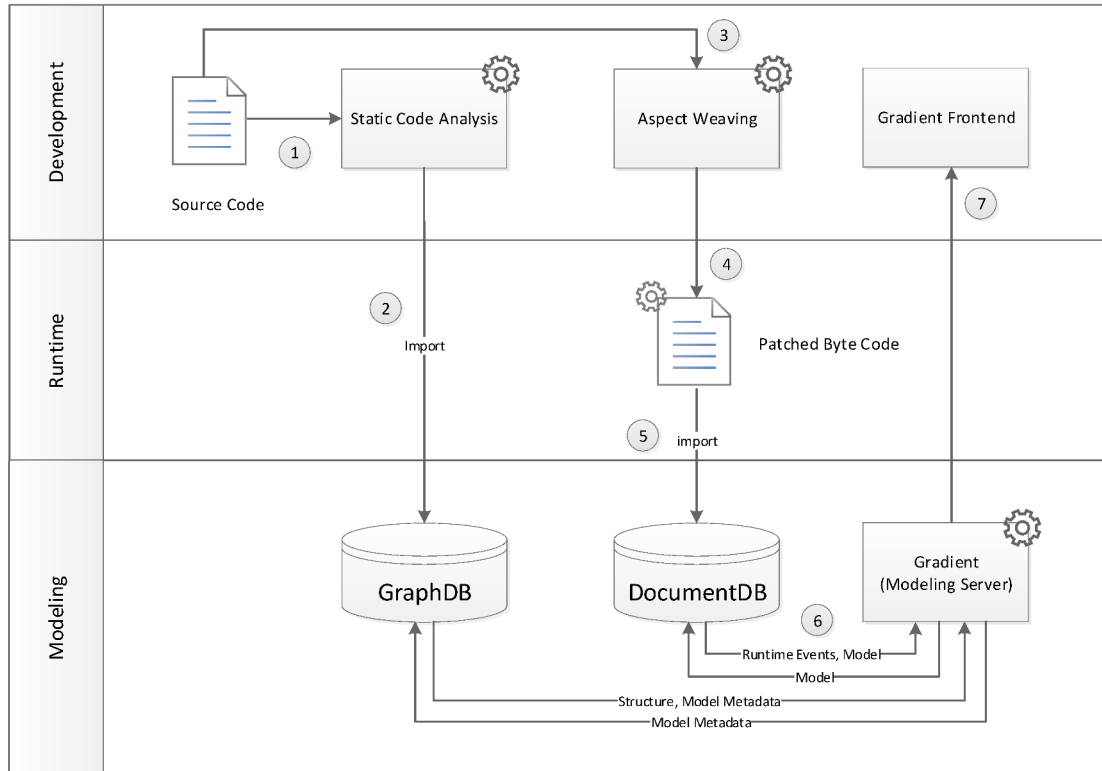


Figure 9.9: Source code is statically analyzed and stored in the graph database. Furthermore, it is compiled, and the resulting byte code is patched with monitoring code that reports runtime events to the document database. Both, structure and runtime events are combined into a statistical model that is written back to the database for later analysis.

This client is provided to users in the form of Gradle [177] (build tool) plugin that handles the entire tool-chain in a non-intrusive and transparent fashion. The Gradient Server then retrieves the structural and behavioral data from the databases and combines them into statistical models ⑥. The *Structure* of the program is retrieved from Neo4j along with the respective *Runtime Events* from MongoDB. The resulting models are stored back in the databases for later use where Neo4j stores the *Model Metadata* and MongoDB the raw model data. At last, the *Gradient Frontend*, hosted on the Gradient Server as a web application, can be used to access the statistical models to inspect the behavior of the analyzed program.

Data model:

Gradient uses a generic high-level data model that can be split into roughly three categories 1. *Code Elements*, 2. *Project Elements* and, 3. *Model Elements*, where the digest of it is shown in Figure 9.10. Code Elements are related to the source code

data model. Project Elements give code elements a project context and enable model versioning. Model Elements capture additional concepts related to the statistical models.

The *Code Elements* section in Figure 9.10 shows that the Gradient data model has a higher level of abstraction than the traditional Abstract Syntax Tree (AST) as it only considers *Types*, *Properties*, and *Executables*. In compensation, the data model introduces *Invocation* and *Access* nodes as explicit relationship concepts that are only implicitly captured on a statement level in an AST. Another addition is the *ElementType* that allows direct access to typing information of typed elements, which cannot be straightforwardly retrieved from an AST.

Project Elements section contains *Project* nodes and *Version* nodes used to manage different projects registered on the same Gradient server. Type, Property, Executable are also *Versionables* containing a version hash that, along with their qualified name, uniquely identifies them within a project and its versions.

This also enables the database to reuse *Versionables*, along with their associated statistical models, with the same qualified names and version hash across different versions. *Modeling Elements* attach model specific information to code elements that are modeled. For example, *Model* contains the id of the statistical model stored in the MongoDB.

Data sources:

Gradient has two sources of data, 1. static code analysis on the source code, 2. dynamic code analysis executed by the patched byte code. The static analysis parses the source code via Spoon [34] and transforms it into the Gradient model. This graph is then written into Neo4j providing the structure. The dynamic analysis is executed by the patched byte code that contains instructions to write monitoring events into the MongoDB. Neo4j can also be thought of an index database where each interaction starts by retrieving specific nodes that point to raw data in MongoDB. This raw data can be millions of runtime events, each being a JSON document or binary data of the statistical models.

Access and usage:

Gradient allows engineers to interact with their source code by inspecting the behavior of types, properties or executables. Neo4j in this setting is used as a persistent data structure of the source code that reflects the parts that are exposed to the user, and as index database for binary data stored in MongoDB. Engineers that use Gradient interact mainly via a graph in list or visual form with the statistical models, or via pre-configured queries and tasks that fully abstract the structure.

Status and ongoing work:

Gradient is an ongoing research prototype for Java to demonstrate the feasibility of Probabilistic Software Modeling. It currently implements the static and dynamic analysis as most parts of the statistical modeling and simple views to view structure, models, and the raw data. High priority features for the future are tools for test-case generation, anomaly detection, and a frontend that allows simple interaction with the models for

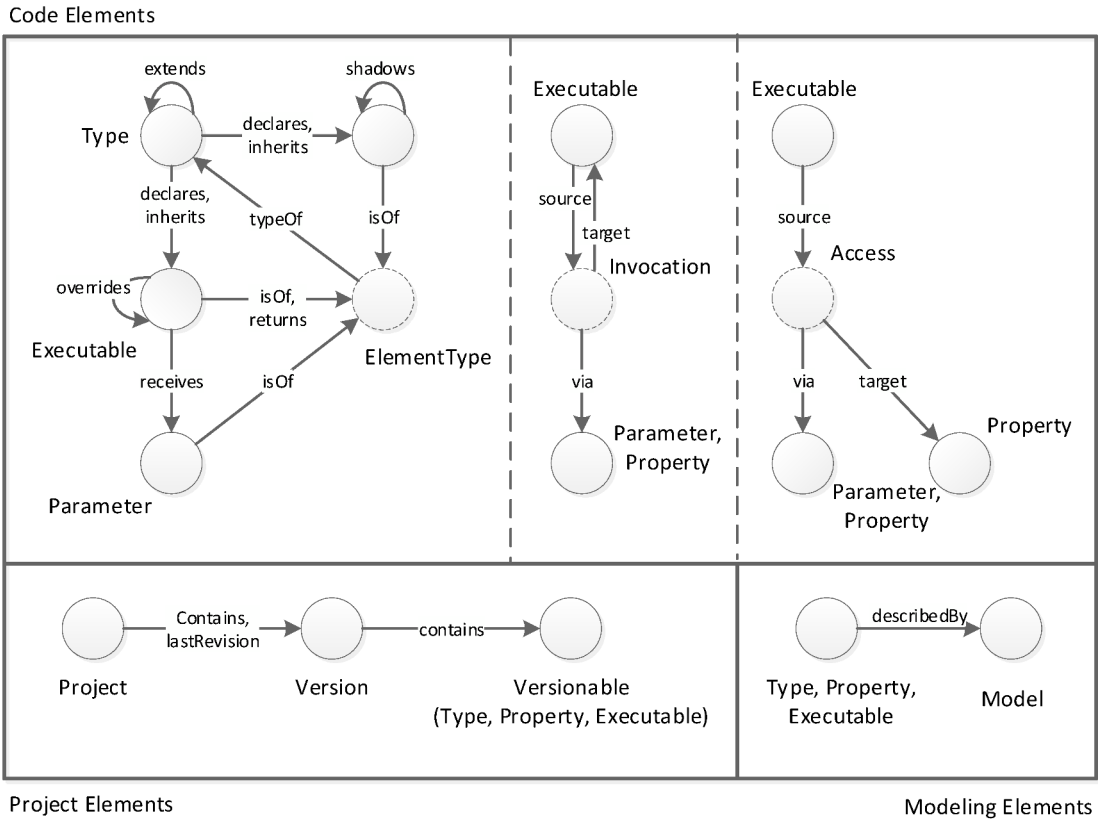


Figure 9.10: The graph data model is split into 3 categories: Code Elements, Project Elements, Model Elements. Code Elements model source code concepts, Project Elements model project and versioning concepts, and Modeling Elements model concepts related to the statistical models built by Gradient.

software developers unfamiliar with statistical modeling. Also, interactions between statistical models of different versions of the source systems are part of the future work.

9.5 Discussion

This section summarizes the insights and lessons learned from the five reported cases via a discussion of advantages and disadvantages of using graph databases. In each case, slightly different aspects of often the same advantage or disadvantage were observed. We therefore aggregated the individual findings collected from the different cases into high-level statements. For each of these statements, examples describing the experienced benefits and drawbacks are given, including references to the cases where they have been found.

9.5.1 Advantages

Graph databases (e.g., Neo4j in our case) are a suitable choice for *storing and querying the data extracted from source code and related artifacts*.

- + Graphs are a natural way to represent the manifold dependencies that are omnipresent in software systems. Working with dependencies has been an essential motivation and was often the central aspect of the tools we developed (*Case 1-5*).
- + Graph databases can handle data from large-scale software systems up to several millions of lines of code as demonstrated by *Case 3*. The limit of Graph databases rather lies in the type of data that has to be processed. For log-like execution data as in *Case 5* a document-centered database (e.g., MongoDB) is preferable; in this case, both databases were used in combination.

Graph databases provide excellent *support for rapid prototyping and exploring different options* for working with artifacts from software engineering. This advantage derives from NoSQL databases being schema-less and highly extensible.

- + Building tools on top of graph databases allow to start using them early, while still under development, and to advance the tools and the underlying data model whenever new requirements or usage scenarios are encountered (*Case 1-5*).
- + In *Case 4* the Neoclipse plugin for Eclipse has been used in a first prototype to demonstrate the integrating of the tool Sherlock into the development environment.
- + In all five cases presented above the implemented tools emerged out of research projects, where the initial versions of the tools were repeatedly revised and successively extended, e.g., to match the diverse needs of our industry partners (*Case 1-4*).

Graph query languages (e.g., Cypher in our case) provide a powerful and simple way to understand, retrieve, and manipulate graphs representing source code or related artifacts.

- + Structured processing of code and other artifacts was found to be easier when using the expressiveness of graph queries than with conventional, programmatic approaches (*Case 2* and *3*). For instance, searching the AST for all method call expressions of a particular method declaration can be achieved by a simple query instead of implementing an AST visitor that requires visiting all method call expressions of a compilation unit and determining for each expression if it belongs to the specified method declaration by checking all parent elements until a method declaration expression is found.
- + In *Case 1* and *Case 2*, the users of our tools were developers. For them, it was straightforward to write queries after a short introduction to the Cypher query language. Dozens of queries have been created so far, supporting a wide range of common analysis tasks. The queries are stored as scripts that can be easily shared and adapted to new analysis tasks.
- + The standard Neo4j Web frontend was used for querying the graph structure and to retrieve the required information about the analyzed software system. Therefore it was not necessary to develop a dedicated client or user interface in *Case 1* and *Case 2*.

9.5.2 Disadvantages

The *generic frontends* available for graph databases (e.g., Web-based Neo4j browser) are often not adequate for supporting end users in performing the specific tasks involved in the studied cases. Custom user interfaces had to be implemented for several of our tools.

- The standard Web interface of Neo4j provides a convenient way to submit queries to the database and to review the results using a visualization of the graph. Nevertheless, the dynamic visualization makes it difficult to maintain the overview when working with large result sets containing dependency data (*Case 1* and *Case 2*). Alternative clients (e.g., yFiles Neo4j Explorer) offer improved layouts and comfortable filtering, but the inherent weaknesses of a generic solution remain.
- Support for specific graphical representations cannot always be provided. In *Case 2*, for example, company partners suggested to display all program elements implemented in the same unit grouped using visual containers (e.g., boxes) representing these units. In contrast, a generic visualization will show the “implemented in” relationship as lines connecting each of the program elements with nodes representing the units.
- The generic user interface does not provide any guidance for users to perform tasks step by step, e.g., in selecting test cases based on a previous selection of a set of code changes (*Case 4*).
- Available frontends are usually restricted to explore data of only one graph database. It is not possible to connect data from two or more databases running in parallel

or from an external data source, e.g., when combining the static structure of a software system with code changes (*Case 4*) or its dynamic behavior (*Case 5*).

Graph databases show a *lack of support for time series data*. This deficiency can be attributed to their specialization on graph data and the philosophy followed by many NoSQL databases, which is “do one thing and do it well”. Neo4j, as we used it in our cases, does not offer any features specific for storing or querying time series data.

- In modeling software engineering artifacts, however, time-related dependencies play a major role due to the evolutionary and incremental approach in which software is developed. Thus, we had to develop graph models that can represent a specific combination of code, artifacts, and dependencies at a particular point in time, e.g., by relating them to dedicated nodes representing software releases or versions (*Case 3-5*).
- In *Case 4*, some of the nodes (e.g., work items) also contain timestamps as attributes, which were required to formulate queries with an additional **where** clause to retrieve all elements in a specific time span.
- In *Case 5*, log-like time series data from execution is stored in a separate database. The characteristic property of such data is the sequential ordering of the entries, which are recorded over time.
- Time points are also relevant in *Case 1* and *Case 2*. However, in these cases, the pragmatic solution was to store only a snapshot of the software system at a particular point in time in the graph database. For analyzing another snapshot, e.g., a new build or version, the entire data set has to be replaced. Managing the dependencies to builds, releases, versions, etc. is left to the users applying the tools.

9.6 Summary and Conclusions

In this chapter, we described our experiences and lessons learned from building software analysis tools and services based on graph databases. We presented five different cases related to different application scenarios and project contexts. Each of the five cases (*Case 1-5*) is an example showing that graph databases can be effectively used for representing and analyzing source code and software engineering artifacts. The diversity as well as the size and complexity of the reported cases underpin this finding.

A *broad range of program elements* is stored in the form nodes and relationships in the graph databases. They range from dependencies such as function calls and read/write access to variables (*Case 1*) to the entire AST of large-scale software systems (*Case 3*). Software written in *various programming languages* has been represented as graphs, e.g., C, C++, C#, Java, and IEC 61131-3 languages (*Case 1-5*). The resulting *size and complexity of the graph structures* range from only two distinct node types and 6 distinct relationship types (*Case 1*) to 83 node types and 88 different relationship types (*Case 3*).

The graph databases showed a high level of *scalability* when used for analyzing up to 44 million lines of code at the level of individual syntax elements. The resulting graph

contained more than 138 million nodes and about the same amount of relationships, which were stored in the database (*Case 3*).

The graph models were used for capturing the data extracted from various *different artifacts*: Source code files (*Case 1-5*); system configurations in form of XML files (*Case 3*); work items, check-ins, software tests, and coverage information (*Case 4*); project and version information (*Case 3-5*).

The databases are populated in a single initial import replacing existing data (*Case 1-2*) or they are updated in increments partially extending the data in the database (*Case 3-5*). The various approaches for accessing the data include Neo4j's standard Web interface (*Case 1-2*), custom built client applications (*Case 3-5*), a REST API (*Case 3*), and an export interface (*Case 4*).

The insights and lessons learned we collected from using graph databases have been compiled into a list of advantages and disadvantages to support decisions in related and future applications.

The key *advantages*, relevant for choosing graph databases as storage option in software analysis, are related to the *versatility of the graph data model*. It was found suitable to represent all kind of structures and relations usually encountered in software systems. It can be used to represent dependencies between individual program elements as well as for links across technology boundaries. In addition, the flexibility and scalability of graph databases provide an ideal basis for *prototyping and evolving analysis solutions*. Finally, *specialized graph query languages* are a powerful yet easy to use means for traversing the huge amounts of nodes and relations required to represent large and complex software systems.

The identified *disadvantages* concern, first, the *limited usefulness of standard database frontends* for end users. The issue with highly generic clients such as the Neo4j browser and similar tools is not a limitation in their functionality. On the contrary, they provide too many options. Custom interfaces built for end users offer only a fraction of their functionality, but they are meaningful in context of a specific usage scenario. Second, graph databases provide *no support for time series data*. Although this is natural consequence of the specialization of graph databases, there is nevertheless the need to represent time-related aspects in all kind of data produced in software development processes.

In future we expect to see a rising number of projects using graph databases for source code analysis and related software engineering tasks. As contribution we plan to investigate ways to combine storage approaches specialized for graph and time series data for building a new tool and service infrastructure.

Acknowledgements.

The research reported in this chapter was supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry for Digital and Economic Affairs, and the Province of Upper Austria in the frame of the COMET center SCCH.

Table 9.1: Overview of the Reported Cases.

	AutoDoc	SCoRe	eKNOWS CMS	Sherlock	Gradient
Goal and motivation	Lightweight, flexible, customizable code analysis	Analyzing industrial PLC software	Service for reusable static code analysis	Dependency analysis for regression test selection	Probabilistic software modeling
Application context	Industry projects	Industry projects	Industry projects	Industry projects	Research prototype
Represented source code or artifacts	Call, read, and write dependencies in various languages (currently C)	PLC programs written in IEC 61131-3 languages	Java systems (from high-level config to source code statements)	Source code structure (C++, C#), code changes, system tests, test coverage	Static code structure and models of behavior (Java VM languages)
Users	Developers	Developers, software architects	Developers of software engineering tools and services	Software testers, quality and release managers	Researchers, developers
Usage scenarios	Analysis of dependencies enriched with metrics	Design and architecture review, support for refactoring	Analysis of large-scale systems, evolution analysis	Selecting regression tests for source code changes	Program comprehension and behavioral analysis
Data access and user interfaces	Neo4j Web interface	Neo4j Web interface	Different (3rd party) tools, REST API	Custom client, export to test tool	Custom Web interface
Data sources	Source code files	Source code files	Version control systems (VCS)	VCS, task management, coverage analysis, test management	Source code, execution traces from run-time
Schema size (distinct elements)	2 node types, 6 relationship types, 9 metric values	24 node types, 15 relationship types	83 node types, 88 relationship types	9 node types, 15 relationship types	11 node types, 20 relationship types
Size of analyzed system	72 000 LOC C code	742 000 LOC IEC61131-3	44 million LOC Java	2.5 million LOC C++ and C#	120 000 LOC Java
Data import/update strategy	Import, existing data replaced	Bulk import, existing data replaced	Cyclic builds, data partially updated	Nightly and manual updates, data partially replaced	Triggered builds with incremental updates

10 Discussion

The concluding discussion should evaluate the research questions posed in 1.1.

10.1 Research Question 1

RQ1 answers whether programs can be transformed into probabilistic models. In Section 3.2 we introduced probabilistic modeling. From this section, it should be clear that most processes can be modeled into a probabilistic model with some appropriate structure of variables. However, many processes are too complex to be effectively modeled. In Chapter 4 we presented how a program can be systematically and automatically transformed into a model. The procedure is a hybrid approach of static and dynamic analysis implying a complex process. This complexity also implies that PSM is hard to reproduce when implemented from scratch. However, PSM builds upon well-known concepts such as static analysis, dynamic analysis, and inference with probabilistic models. Also, the generated artifacts, i.e., structure, behavior, and inference graph, existed already in one or the other form. The structure graph is an abstract semantics graph used in any program analysis task such as compiler optimization, or design pattern detection. While the details might differ, the conceptual essence is the same. Similarly, the behavior graph exists in many tracing applications and is ultimately only an extended stack trace of the program. The inference graph is a cluster graph whose local models are NVPs. Both are well-known and used throughout the machine learning and probabilistic modeling community. The feasibility study in Chapter 5 evaluated an actual implementation of the presented methods. The studies showed that the process of generating the models is stable and yields good results in terms of their estimated densities. This evidence was confirmed in Chapter 7 in which the models could successfully and with high reliability detect semantic clones. However, both chapters also showed weaknesses in the transformation. PSM works best for business logic. Libraries that only manage references but never dereference the underlying data are not a good fit for PSM. Also, the implemented prototype does not model the distribution of sequences. This may cause incorrect inference results given certain PSM applications (e.g., semantic clone detection). Given the theoretical and practical evidence, we can conclude that:

RQ1 — Programs can be transformed into probabilistic models via an automated and systematic process.

RQ1.1 answers whether the structure of a program can be preserved in the resulting model. In Section 3.2 we saw that probabilistic models represent the joint distribution of

variables. In general, there are many dependency structures that can model the same joint distribution. However, they vary in their accuracy. PSM uses the structural information that is present in the program itself. Not only is this the true structure of the process that generates the behavior, but it also leads to a model that reflects the program itself. This is in line with the structural closeness requirement. The mapping between program and IG needs little mental overview especially in its cluster graph representation. In comparison, the factor graph representation provides more details and is useful if one wants to zoom into a subset of the program. However, there are differences between the IG and SG. These differences are caused by reference elements that are not directly present in the cluster graph. Also, the factor graph representation does not highlight the call order of factors. Any of these elements may be added as visual guides but are not included per se. Important here is that statements themselves are not structural elements but rather algorithmic details that are not the target of PSM. Given that the code and model elements have a one-to-one relationship, and that elements that are not modeled can be added as visual guides we conclude that:

RQ1.1 — The probabilistic model preserves the structural properties of the modeled program.

RQ1.2 answers whether the behavior of a program can be preserved in the resulting model. A clear distinction between use-case and behavior preservation is needed to answer this question. First, it is important to recognize that each model can only contain the behavior of specific executions of a program. These executions can be representative of the behavior or are only representative of a small subset of it. For example, if the (apparently) nondeterministic behavior of a program is of interest, then the model will only capture this behavior. This leads to an important fact, PSM only builds a model of the program, not a perfect copy. Think of a simulated car which can only go back and forth but can not take turns. It is of little use in a navigation simulation through a city center, however, it is valid if one wants to evaluate the aerodynamics of the car. Similarly, PSM models have a certain use-case, e.g., semantic clone detection. In this case the actual inputs are secondary but the input and output *relationship* in executables are of interest. *Hence, the scope of behavior that is modelled is part of the use-case and the responsibility of the PSM practitioner.* The responsibility of the model is to capture the behavior of the use case as precisely as possible. Chapter 5 included a visual inspection of the data showing that the density estimates are of high fidelity. A sample of this is given Section 5.1.5. Chapter 7 showed that PSM can effectively detect semantic clones. These programs are part of a programming competition with problems of varying complexity. The precision with which the clones were found, especially the false-positive rate, indicates that the models are capable of detecting semantic differences and similarities. Chapter 8 modeled a simple but typical error in programs, which was successfully localized along with an error path. These are only a handful of examples and there is further research needed to fully evaluate this research question. However, given the current evidence, we can conclude that:

RQ1.2 — The probabilistic model preserves the behavioral properties of the modeled program.

10.2 Research Question 2

RQ2 answers if PSM scales up to a certain computational horizon. The computational horizon depends on the application and needs a case-by-case evaluation. However, there are some computational limits associated with most use-cases, e.g., that the transformation from program to the IG does not take several days. Chapter 5 showed that the extraction of the SG is a negligible overhead. Furthermore, we have seen that the extraction of the BG takes only minutes. However, this depends on the use-case and trigger that is applied. For example, extracting the BG for semantic clone detection is fast as the triggers can be limited (less than 20sec for SG, BG, and IG). Ultimately, it is a trade-off between accuracy and speed where the accuracy saturates quickly. A more involved use case is the probabilistic modeling of server environments, e.g., to detect anomalies. In these cases, the IG needs to be updated continuously.

The time required for creating the IG and optimizing the local models is in the range of seconds. This is an acceptable time frame for most applications. However, there are use-cases that may not scale, such as debugging. Debugging has a use-case horizon ranging from seconds to a few minutes. In such a case, running, monitoring, and constructing the entire IG may not scale. Local construction of target subsystems or incremental IG construction may alleviate potential scaling issues for such use-cases. However, these approaches still need investigation. For the presented use-cases, i.e., program modeling, semantic clone detection, and fault localization, we have seen acceptable computational costs in the range of seconds for executables. Given the evidence on the presented use-cases, we can conclude that:

RQ1.2 — PSM scales for program modeling, semantic clone detection, and fault localization. However, more research on various use-cases are needed to fully confirm RQ2.

10.3 Research Question 3

RQ3 answers whether PSM is useful for software engineering tasks. Evaluating the usefulness of a framework takes years. However, a first glimpse of the potential of PSM is already given in this thesis. In Chapter 7 we present a semantic clone detector that is capable of detecting behavioral similarities between executables. The use-case of finding semantic clones is not only state-of-art but enables many related applications. For example, fault localization (Chapter 8) builds upon the same concepts as semantic clone detection. The main difference is that semantic differences are detected between versions

of the same executable. Anomaly detection is a use-case that builds upon the same concepts of semantic similarity but operates in production environments of programs. Each of these use-cases is important in the context of software engineering. However, it is still too early to conclude general usability of PSM as envisioned in this thesis. Given the present evidence, we can conclude that:

RQ3 — PSM is useful for detecting semantic similarities in programs and between program revisions. However, more research is needed to conclude the general usefulness of PSM.

11 Conclusions

In this thesis, we presented Probabilistic Software Modeling (PSM), a modeling paradigm for programs. The presented methods comprehensively describe how programs given by their source code can be transformed into a probabilistic model. These probabilistic models can be used in a wide range of analysis and generative applications. Furthermore, we provided preliminary chapters that lead to PSM or extended it.

11.1 Thesis Contributions

The main contributions of this thesis were organized as follows.

Vision and Applications We presented our vision for probabilistically modeling programs. This complements the existing body of work on software modeling and creates new research directions and applications. We outlined potential PSM applications and their use-cases for program comprehension, software analysis, and generative analysis of programs. These applications were later evaluated. Some, such as semantic clone detection or fault localization, were evaluated directly in the form of a study. Others, were evaluated indirectly as part of the presented work. Finally, the vision and applications point to potential future work in the context of software modeling and PSM.

PSM Method The methods involved with PSM are the main contribution of this thesis. This includes 1. the general workflow on how a program can be transformed into a probabilistic model, 2. the extraction of the program structure in form of a structure graph, 3. the extraction of the program behavior in form of a behavior graph, 4. the combination of the structure and behavior graphs into an inference graph and the operations that can be applied to it. Finally, we presented the theoretical foundation of PSM by unifying existing research on probabilistic models with the concepts used in PSM.

Feasibility The feasibility study provided basic but essential insights into the modeling of programs via PSM. These insights showed us that programs expose many code elements that are eligible for PSM modeling. Furthermore, we saw that only a small proportion of the code elements cause most of the program behavior. We then presented the extent of the inference graph of real-world programs. Finally, we evaluated the capabilities of preserving information in long inference chains.

Text-based Clone Detection As a preliminary study into semantic clone detection, we presented a comprehensive study of textual clones in PLC programs. The study

showed that textual clones are present in many types of programs. Furthermore, the study showed that existing text based clone detectors are good at finding structural similarities in programs. However, logical clones were hardly found and confirmed by human raters. This result lead to the hypothesis that semantic clones might only reliably be found by inspecting the runtime behavior of programs.

Semantic Clone Detection A natural follow-up from the text-based clone detection study was the study on semantic clones. In this study, we presented a state-of-art clone detector that could reliably detect semantic clones by comparing the input and output behavior of methods. It extended PSM with a detection pipeline that compares methods on a static, dynamic, and model level. The main contribution was the use of the inference graph and the likelihood ratio test to find semantically similar model elements. This approach is also the foundation for other comparative PSM applications such as fault localization. The semantic clone detector showed a strong performance with an MCC of more than 0.9 on detecting semantic clones in coding competitions and classical semantic clone problems.

Fault Localization The preliminary study on fault localization showed how PSM can be used to find faults within programs. It used the same mechanism as semantic clone detection in which the behaviors of two different versions of a method were compared. The study focused on the analysis of the Nutrition Advisor. Behavioral changes in the program including the path of changes in the structure graph were successfully detected and quantified. The work is an important step for a more elaborate study on fault localization via PSM.

Source Code Analysis via Graph Databases Finally, we used the structure graph of PSM to conduct a study on static code analysis via graph databases. The study evaluated various different tools including Gradient, a PSM prototype, and their fit for static code analysis. We showed that graph databases have various advantages for static code analysis. Especially, the graph query language is particularly advantageous for the various graphs used in PSM. However, the lack of scalability for other complex data formats, such as time-series, pose potential problems for a fully-fledged PSM implementation.

11.2 Future Work

Probabilistic Software Modeling is a young methodology compared to existing methods in software engineering. Many applications from Section 1.2 are potential future directions of research. For example, extending the fault localization in Chapter 8 by evaluating it on existing fault localization benchmarks. Another future direction which is of high relevance, and which fits well with the generative capabilities of PSM, is test-case generation. A more elaborate future work would combine fault localization with test-case generation to automatically generate tests for detected faults. Another future direction is to integrate program performance metrics such as execution time and memory consumption of

executables. Extending the existing inference graph with these performance metrics could allow the inference of potential performance bottlenecks in programs.

A challenge is given by lists and sets in programs that are exploded into singular values in the current version of PSM. An extension of PSM might consider modeling distribution of sequences for these code elements, in order to reduce the modeling error that propagates to applications like semantic clone detection.

Improving the model fidelity in regard to discrete variables would improve the precision of the inference and their applications. One could replace the NVPs with more recent versions of flow-based models that are more expressive. Another possibility is to optimize the model class of local models by their model elements.

Another avenue of future research is incremental IG construction. Evolving the model with the application use-case allows one to effectively reduce the number of computations. For example, instead of re-creating the entire model, one re-creates only parts of the model if a change is pushed by a developer. This could reduce the creation time of the fully updated model to several seconds, making it usable in live programming sessions, such as debugging.

In conclusion, there are many practical and theoretical challenges in the context of PSM. This thesis presented the foundation of PSM, and paves the way to a challenging new landscape of software analysis.

Bibliography

- [1] H. Thaller, R. Ramler, J. Pichler, and A. Egyed, “Exploring code clones in programmable logic controller software,” in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Limassol: IEEE, Sep. 2017, p. 1–8.
- [2] H. Thaller, L. Linsbauer, A. Egyed, and S. Fischer, “Towards Fault Localization via Probabilistic Software Modeling,” in *2020 IEEE 3rd International Workshop on Validation, Analysis, and Evolution of Software Tests (VST)*. IEEE, 2020, p. 24–27.
- [3] H. Thaller, L. Linsbauer, and A. Egyed, “Towards Semantic Clone Detection via Probabilistic Software Modeling,” in *2020 IEEE 14th International Workshop on Software Clones (IWSC)*. IEEE, 2020, p. 64–69.
- [4] D. Firesmith, “The challenges of testing in a non-deterministic world,” Carnegie Mellon University’s Software Engineering Institute Blog, Jan. 9, 2017. [Online]. [Online]. Available: <http://insights.sei.cmu.edu/blog/the-challenges-of-testing-in-a-non-deterministic-world/>
- [5] S. Jayaraman, B. Jayaraman, and D. Lessa, “Compact Visualization of Java Program Execution,” *Software: Practice and Experience*, vol. 47, no. 2, p. 163–191, 2017.
- [6] M. H. Brown and R. Sedgewick, “Techniques for Algorithm Animation,” *IEEE Software*, vol. 2, no. 1, p. 28–39, Jan. 1985.
- [7] S. Mukherjea and J. T. Stasko, “Toward Visual Debugging: Integrating Algorithm Animation Capabilities Within a Source-Level Debugger,” *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 3, p. 215–244, Sep. 1994.
- [8] T. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, p. 308–320, 1976.
- [9] M. Gabel, L. Jiang, and Z. Su, “Scalable Detection of Semantic Clones,” in *Proceedings of the 13th International Conference on Software Engineering - ICSE ’08*. ACM Press, 2008, p. 321.
- [10] H. Kim, Y. Jung, S. Kim, and K. Yi, “Mecc,” *Proceeding of the 33rd international conference on Software engineering - ICSE ’11*, p. 301, 2011.

- [11] J. Fan, C. Zhang, and J. Zhang, “Generalized Likelihood Ratio Statistics and Wilks Phenomenon,” *The Annals of Statistics*, vol. 29, no. 1, p. 153–193, 2001.
- [12] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, “Evaluating and Improving Fault Localization,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, p. 609–620.
- [13] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A Survey on Software Fault Localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, p. 707–740, Aug. 2016.
- [14] S. Hangal and M. S. Lam, “Tracking down Software Bugs Using Automatic Anomaly Detection,” *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, p. 291–301, 2002.
- [15] L. Aniello, C. Ciccotelli, M. Cinque, F. Frattini, L. Querzoni, and S. Russo, “Automatic Invariant Selection for Online Anomaly Detection,” in *Computer Safety, Reliability, and Security*, A. Skavhaug, J. Guiochet, and F. Bitsch, Eds. Cham: Springer International Publishing, 2016, vol. 9922, p. 172–183.
- [16] V. Kotu and B. Deshpande, “Anomaly Detection,” in *Data Science*. Elsevier, 2019, p. 447–465.
- [17] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly Detection: A Survey,” *ACM Computing Surveys*, vol. 41, no. 3, p. 1–58, Jul. 2009.
- [18] L. Cseppentő and Z. Micskei, “Evaluating Code-Based Test Input Generator Tools: Evaluating Code-Based Test Input Generator Tools,” *Software Testing, Verification and Reliability*, vol. 27, no. 6, p. e1627, Sep. 2017.
- [19] G. Fraser and A. Zeller, “Mutation-Driven Generation of Unit Tests and Oracles,” *IEEE Transactions on Software Engineering*, vol. 38, no. 2, p. 278–292, Mar. 2012.
- [20] C. f. D. C. a. P. nhanes2013, “Centers for Disease Control and Prevention, National Health and Nutrition Examination Survey NHANES 2013 - 2014,” <https://wwwn.cdc.gov/nchs/nhanes/continuousnhanes/default.aspx?BeginYear=2013>.
- [21] T. Ball, “The Concept of Dynamic Analysis,” in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. London, UK, UK: Springer-Verlag, 1999, p. 216–234.
- [22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An Overview of AspectJ,” in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP ’01. Springer-Verlag, 2001, p. 327–353.
- [23] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*, ser. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press, 2009.

-
- [24] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*, ser. Adaptive Computation and Machine Learning Series. Cambridge, MA: MIT Press, 2012.
 - [25] C. M. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. New York: Springer, 2006.
 - [26] D. M. Blei, “Build, compute, critique, repeat: Data analysis with latent variable models,” *Annual Review of Statistics and Its Application*, vol. 1, no. 1, p. 203–232, jan 2014.
 - [27] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, “Probabilistic programming in Python using PyMC3,” *PeerJ Computer Science*, vol. 2, p. e55, Apr. 2016.
 - [28] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.
 - [29] J. Kruschke, “Doing bayesian data analysis: A tutorial with r, jags, and stan,” 2014.
 - [30] L. Dinh, J. Sohl-Dickstein, and S. Bengio, “Density estimation using Real NVP,” *arXiv:1605.08803 [cs, stat]*, May 2016.
 - [31] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan, “Normalizing Flows for Probabilistic Modeling and Inference,” *arXiv:1912.02762 [cs, stat]*, Dec. 2019.
 - [32] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.
 - [33] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.
 - [34] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, “SPOON: A Library for Implementing Analyses and Transformations of Java Source Code,” *Software: Practice and Experience*, p. n/a–n/a, 2015.
 - [35] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.
 - [36] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” 1999.
 - [37] D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best, “The BUGS project: Evolution, critique and future directions,” *Statistics in Medicine*, vol. 28, no. 25, p. 3049–3067, 2009.
 - [38] D. Tran, R. A. Saurous, M. D. Hoffman, E. Brevdo, K. Murphy, and D. M. Blei, “Deep Probabilistic Programming,” p. 1–17, 2017.

-
- [39] D. Koller and A. Pfeffer, “Object-oriented Bayesian Networks,” in *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, ser. UAI’97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, p. 302–313.
 - [40] F. Musella and P. Vicard, “Object-oriented Bayesian networks for complex quality management problems,” *Quality & Quantity*, vol. 49, no. 1, p. 115–133, Jan. 2015.
 - [41] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu, “Specifying and Verifying Systems with TLA+,” in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, ser. EW 10. New York, NY, USA: ACM, 2002, p. 45–48.
 - [42] D. Jackson, “Alloy: A Lightweight Object Modelling Notation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, p. 256–290, Apr. 2002.
 - [43] M. Kwiatkowska, G. Norman, and D. Parker, “Stochastic Model Checking,” in *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*, ser. Lecture Notes in Computer Science, M. Bernardo and J. Hillston, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, p. 220–270.
 - [44] —, “PRISM 4.0: Verification of Probabilistic Real-Time Systems,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Springer Berlin Heidelberg, 2011, p. 585–591.
 - [45] Y. Liu, J. Sun, and J. S. Dong, “PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers,” in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. Hiroshima, Japan: IEEE, Nov. 2011, p. 190–199.
 - [46] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, “CADP 2011: A toolbox for the construction and analysis of distributed processes,” *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 2, p. 89–107, Apr. 2013.
 - [47] J. C. King, “Symbolic Execution and Program Testing,” *Communications of the ACM*, vol. 19, no. 7, p. 385–394, Jul. 1976.
 - [48] S. Anand, C. S. Păsăreanu, and W. Visser, “JPF-SE: A Symbolic Execution Extension to Java PathFinder,” in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS’07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 134–138.
 - [49] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, p. 209–224.
 - [50] N. Tillmann and J. de Halleux, “Pex–White Box Test Generation for .NET,” in *Tests and Proofs*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds. Springer Berlin Heidelberg, 2008, p. 134–153.

-
- [51] J. Geldenhuys, M. B. Dwyer, and W. Visser, “Probabilistic Symbolic Execution,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. ACM, 2012, p. 166–176.
 - [52] A. Filieri, C. S. Pasareanu, and G. Yang, “Quantification of Software Changes through Probabilistic Symbolic Execution (N),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Lincoln, NE, USA: IEEE, Nov. 2015, p. 703–708.
 - [53] B. Chen, Y. Liu, and W. Le, “Generating Performance Distributions via Probabilistic Symbolic Execution,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: ACM, 2016, p. 49–60.
 - [54] Z. Xu, S. Ma, X. Zhang, S. Zhu, and B. Xu, “Debugging with Intelligence via Probabilistic Inference,” p. 11, 2018.
 - [55] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu, “Statistical Debugging Using Latent Topic Models,” in *Machine Learning: ECML 2007*, J. N. Kok, J. Koronacki, R. L. de Mantaras, S. Matwin, D. Mladenič, and A. Skowron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4701, p. 6–17.
 - [56] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically Discovering Likely Program Invariants to Support Program Evolution,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, p. 99–123, Feb. 2001.
 - [57] T.-D. B. Le and D. Lo, “Deep Specification Mining,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*. ACM Press, 2018, p. 106–117.
 - [58] Z. Zuo, S.-C. Khoo, and C. Sun, “Efficient Predicated Bug Signature Mining via Hierarchical Instrumentation,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, p. 215–224.
 - [59] R. Gore, P. F. Reynolds, and D. Kamensky, “Statistical debugging with elastic predicates,” in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Nov. 2011, p. 492–495.
 - [60] D. Lo and S. Maoz, “Scenario-based and value-based specification mining: Better together,” *Automated Software Engineering*, vol. 19, no. 4, p. 423–458, Dec. 2012.
 - [61] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Dec. 2014.
 - [62] A. Krogh and J. A. Hertz, “A simple weight decay can improve generalization,” in *Proceedings of the 4th International Conference on Neural Information Processing Systems*, ser. NIPS’91. Denver, Colorado: Morgan Kaufmann Publishers Inc., Dec. 1991, p. 950–957.

-
- [63] D. Hendrycks and K. Gimpel, “Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units,” *CoRR*, vol. abs/1606.08415, 2016.
 - [64] Structurizr, “Structurizr,” Structurizr, 2019.
 - [65] opencollab, “Jlatexmath,” opencollab, 2019.
 - [66] PMD, “Pmd,” PMD, 2019.
 - [67] F. J. Massey, “The Kolmogorov-Smirnov Test for Goodness of Fit,” *Journal of the American Statistical Association*, vol. 46, no. 253, p. 68–78, Mar. 1951.
 - [68] S. Cass, “The 2016 top programming languages,” 2016. [Online]. Available: <http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages>
 - [69] C. K. Roy and J. R. Cordy, “A Survey on Software Clone Detection Research,” *Queen’s School of Computing TR*, vol. 115, p. 115, 2007.
 - [70] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, “Do Code Clones Matter?” in *Proceedings - International Conference on Software Engineering*, 2009, p. 485–495.
 - [71] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1998.
 - [72] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, ser. The Addison-Wesley Object Technology Series. Reading, MA: Addison-Wesley, 1999.
 - [73] R. Koschke, “Frontiers of Software Clone Management,” in *Frontiers of Software Maintenance, 2008. FoSM 2008*. IEEE, 2008, p. 119–128.
 - [74] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, “Evolution of Software in Automated Production Systems: Challenges and Research Directions,” *Journal of Systems and Software*, vol. 110, p. 54–84, 2015.
 - [75] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and Evaluation of Clone Detection Tools,” *IEEE Transactions on Software Engineering*, vol. 33, no. 9, p. 577–591, 2007.
 - [76] S. Harris, *Simian - Similarity Analyser*, 2003.
 - [77] I. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone Detection Using Abstract Syntax Trees,” *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, p. 368–377, 1998.
 - [78] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” *Information and Software Technology*, vol. 55, no. 7, p. 1165–1199, Jul. 2013.

-
- [79] R. Koschke, I. Baxter, M. Conradt, and J. Cordy, “Software Clone Management Towards Industrial Application (Dagstuhl Seminar 12071),” *Dagstuhl Reports*, vol. 2, no. 2, p. 21–57, 2012.
 - [80] S. Ducasse, M. Rieger, and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM’99). ‘Software Maintenance for Business Change’ (Cat. No.99CB36360)*, no. c, p. 109–118, 1999.
 - [81] M. Rieger, S. Ducasse, and M. Lanza, “Insights into System-Wide Code Duplication,” in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2004, p. 100–109.
 - [82] M. Asaduzzaman and C. Roy, “VisCad: Flexible Code Clone Analysis Support for NiCad,” *Proceeding of the 5th*, p. 77–78, 2011.
 - [83] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “Gemini: Maintenance Support Environment Based on Code Clone Analysis,” *Proceedings - International Software Metrics Symposium*, vol. 2002-Janua, p. 67–76, 2002.
 - [84] R. Tairas, J. Gray, and I. Baxter, “Visualization of Clone Detection Results,” *Proc. ETX at OOPSLA*, p. 50–54, 2006.
 - [85] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “ARIES : Refactoring Support Tool for Code Clone,” *3-WoSQ Proceedings of the third workshop on Software quality*, p. 1–4, 2005.
 - [86] C. Kapser and M. W. Godfrey, “Aiding Comprehension of Cloning Through Categorization,” *Proceedings of the Principles of Software Evolution, 7th International Workshop*, p. 85–94, 2004.
 - [87] —, “Improved Tool Support for the Investigation of Duplication in Software,” *IEEE International Conference on Software Maintenance, ICSM*, vol. 2005, p. 305–314, 2005.
 - [88] F. Li, G. Bayrak, K. Kernschmidt, and B. Vogel-Heuser, “Specification of the Requirements to Support Information Technology-Cycles in the Machine and Plant Manufacturing Industry,” in *IFAC Proceedings Volumes (IFAC-PapersOnline)*, vol. 14, 2012, p. 1077–1082.
 - [89] E. Duala-Ekoko and M. P. Robillard, “CloneTracker: Tool Support for Code Clone Management.”
 - [90] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, “Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants,” *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, p. 391–400, 2014.

-
- [91] M. Kim, L. Bergman, T. Lau, and D. Notkin, “An Ethnographic Study of Copy and Paste Programming Practices in OOPL,” *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on*, p. 83–92, 2004.
 - [92] C. J. Kapser and M. W. Godfrey, ““Cloning considered harmful” considered harmful: Patterns of cloning in software,” *Empirical Software Engineering*, vol. 13, no. 6, p. 645–692, Dec. 2008.
 - [93] B. Baker, “On Finding Duplication and Near-Duplication in Large Software Systems,” *Proceedings of 2nd Working Conference on Reverse Engineering*, p. 86–95, 1995.
 - [94] T. Kamiya, “Agec: An Execution-Semantic Clone Detection Tool,” in *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, p. 227–229.
 - [95] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, “Pattern Matching for Clone and Concept Detection,” *Automated Software Engineering*, vol. 3, no. 1-2, p. 77–108, 1996.
 - [96] Mayrand, Leblanc, and Merlo, “Experiment on the automatic detection of function clones in a software system using metrics,” in *Proceedings of International Conference on Software Maintenance ICSM-96*. Monterey, CA, USA: IEEE, 1996, p. 244–253.
 - [97] A. Lozano, M. Wermelinger, and B. Nuseibeh, “Evaluating the Harmfulness of Cloning: A Change Based Experiment,” *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007*, p. 0–3, 2007.
 - [98] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, “Software quality analysis by code clones in industrial legacy software,” in *Proceedings Eighth IEEE Symposium on Software Metrics*. Ottawa, Ont., Canada: IEEE Comput. Soc, 2002, p. 87–94.
 - [99] J. Krinke, “Is Cloned Code More Stable than Non-Cloned Code?” *Proceedings - 8th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2008*, p. 57–66, 2008.
 - [100] —, “Is Cloned Code Older than Non-Cloned Code?” 2011.
 - [101] F. Rahman, C. Bird, and P. Devanbu, “Clones: What Is That Smell?” *Empirical Software Engineering*, vol. 17, no. 4-5, p. 503–530, 2012.
 - [102] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, “Studying the Impact of Clones on Software Defects,” *Proceedings - Working Conference on Reverse Engineering, WCRE*, p. 13–21, 2010.

-
- [103] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An Empirical Study on Inconsistent Changes to Code Clones at the Release Level," *Science of Computer Programming*, vol. 77, no. 6, p. 760–776, 2012.
 - [104] N. Göde and J. Harder, "Clone Stability," *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, p. 65–74, 2011.
 - [105] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems Creating Task-Relevant Clone Detection Reference Data," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, vol. 2003-Janua, 2003, p. 285–294.
 - [106] G. Norman, "Likert Scales, Levels of Measurement and the "Laws" of Statistics," *Advances in Health Sciences Education*, vol. 15, no. 5, p. 625–632, 2010.
 - [107] H. A. Kurdi, "Review on Aspect Oriented Programming," vol. 4, no. 9, p. 22–27, 2013.
 - [108] K. Bengtsson, B. Lennartson, O. Ljungkrantz, and C. Yuan, "Developing control logic using aspect-oriented programming and sequence planning," *Control Engineering Practice*, vol. 21, no. 1, p. 12–22, 2013.
 - [109] K. O. McGraw and S. P. Wong, "Forming Inferences about Some Intraclass Correlations Coefficients," *Psychological Methods*, vol. 1, no. 1, p. 30–46, 1996.
 - [110] D. V. Cicchetti, "Guidelines, Criteria, and Rules of Thumb for Evaluating Normed and Standardized Assessment Instruments in Psychology," *Psychological Assessment*, vol. 6, no. 4, p. 284–290, 1994.
 - [111] P. H. Westfall, R. D. Tobias, and R. D. Wolfinger, *Multiple Comparisons and Multiple Tests Using SAS*. SAS Institute, 2011.
 - [112] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for Better Configurations: A Rigorous Approach to Clone Evaluation," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, p. 455, 2013.
 - [113] R. C. Martin, Ed., *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2009.
 - [114] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Reading, Mass: Addison-Wesley, 2000.
 - [115] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, p. 73, Dec. 2001.
 - [116] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, p. 176–192, 2006.

-
- [117] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger, “Relation of Code Clones and Change Couplings,” in *Fundamental Approaches to Software Engineering*, L. Baresi and R. Heckel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 3922, p. 411–425.
 - [118] E. Juergens, F. Deissenboeck, and B. Hummel, “Code Similarities Beyond Copy & Paste,” in *2010 14th European Conference on Software Maintenance and Reengineering*. Madrid: IEEE, Mar. 2010, p. 78–87.
 - [119] R. Koschke, “Survey of research on software clones,” in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
 - [120] J. Svajlenko and C. K. Roy, “Evaluating clone detection tools with BigCloneBench,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Bremen, Germany: IEEE, Sep. 2015, p. 131–140.
 - [121] F. Farmahinifarahani, V. Saini, D. Yang, H. Sajnani, and C. V. Lopes, “On Precision of Code Clone Detection Tools,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2019, p. 84–94.
 - [122] H.-H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI’17. Melbourne, Australia: AAAI Press, Aug. 2017, p. 3034–3040.
 - [123] V. Kafer, S. Wagner, and R. Koschke, “Are there functionally similar code clones in practice?” in *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. Campobasso: IEEE, Mar. 2018, p. 2–8.
 - [124] J. Krinke, “Identifying Similar Code with Program Dependence Graphs,” *Proceedings Eighth Working Conference on Reverse Engineering*, p. 301–309, 2001.
 - [125] H. Thaller, L. Linsbauer, and A. Egyed, “Feature Maps: A Comprehensible Software Representation for Design Pattern Detection,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Hangzhou, China: IEEE, Feb. 2019, p. 207–217.
 - [126] H. Thaller, L. Linsbauer, R. Ramler, and A. Egyed, “Probabilistic Software Modeling: A Data-driven Paradigm for Software Analysis,” *arXiv:1912.07936 [cs]*, Dec. 2019.
 - [127] G. Fink and M. Bishop, “Property-based testing: A new approach to testing for assurance,” *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 4, p. 74–80, Jul. 1997.

-
- [128] S. Boughorbel, F. Jarray, and M. El-Anbari, “Optimal classifier for imbalanced data using matthews correlation coefficient metric,” *PloS one*, vol. 12, no. 6, p. e0177678, 2017.
 - [129] J. D. Evans, *Straightforward Statistics for the Behavioral Sciences*. Pacific Grove: Brooks/Cole Pub. Co, 1996.
 - [130] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” may 2019.
 - [131] J. R. Cordy and C. K. Roy, “The NiCad Clone Detector,” in *2011 IEEE 19th International Conference on Program Comprehension*, Jun. 2011, p. 219–220.
 - [132] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, “Ccaligner: a token based large-gap clone detector,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, p. 1066–1077.
 - [133] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “Sourcerercc: Scaling code clone detection to big-code,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, p. 1157–1168.
 - [134] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, “Oreo: Detection of clones in the twilight zone,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. Lake Buena Vista, FL, USA: ACM Press, 2018, p. 354–365.
 - [135] N. Göde and R. Koschke, “Incremental Clone Detection,” in *2009 13th European Conference on Software Maintenance and Reengineering*, Mar. 2009, p. 219–228.
 - [136] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, “Code relatives: detecting similarly behaving software,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. ACM Press, 2016.
 - [137] S. Wagner, A. Abdulkhaleq, I. Bogicevic, J.-P. Ostberg, and J. Ramadani, “How are functionally similar code clones syntactically different? An empirical study and a benchmark,” *PeerJ Computer Science*, vol. 2, p. e49, Mar. 2016.
 - [138] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, “Detecting code clones with graph neural network and flow-augmented abstract syntax tree,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, feb 2020.
 - [139] C. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, and P. Weißgerber, “Subjectivity in clone judgment: Can we ever agree?” in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no.

06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. [Online]. Available: <http://drops.dagstuhl.de/opus/volltexte/2007/970>
- [140] Y. Dang, S. Ge, R. Huang, and D. Zhang, “Code clone detection experience at microsoft,” in *Proceeding of the 5th international workshop on Software clones - IWSC11*. ACM Press, 2011.
- [141] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère, “An empirical assessment of bellon’s clone benchmark,” in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering - EASE ’15*. ACM Press, 2015.
- [142] A. Charpentier, J.-R. Falleri, F. Morandat, E. B. H. Yahia, and L. Réveillère, “Raters’ reliability in clone benchmarks construction,” *Empirical Software Engineering*, vol. 22, no. 1, p. 235–258, feb 2016.
- [143] C. K. Roy and J. R. Cordy, “Benchmarks for software clone detection: A ten-year retrospective,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, mar 2018.
- [144] R. Elva and G. T. Leavens, “JSCTracker : A Semantic Clone Detection Tool for Java Code,” 2012.
- [145] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner, “Challenges of the Dynamic Detection of Functionally Similar Code Fragments,” in *2012 16th European Conference on Software Maintenance and Reengineering*, Mar. 2012, p. 299–308.
- [146] G. Li, H. Liu, Y. Jiang, and J. Jin, “Test-Based Clone Detection: An Initial Try on Semantically Equivalent Methods,” *IEEE Access*, vol. 6, p. 77643–77655, 2018.
- [147] L. Jiang and Z. Su, “Automatic Mining of Functionally Equivalent Code Fragments via Random Testing,” in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA ’09. New York, NY, USA: ACM, 2009, p. 81–92.
- [148] G. Zhao and J. Huang, “DeepSim: Deep learning code functional similarity,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, Oct. 2018, p. 141–151.
- [149] J. Jones, M. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, May 2002, p. 467–477.
- [150] W. E. Wong, V. Debroy, R. Gao, and Y. Li, “The DStar Method for Effective Software Fault Localization,” *IEEE Transactions on Reliability*, vol. 63, no. 1, p. 290–308, Mar. 2014.

-
- [151] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*. Toronto, Ontario, Canada: ACM Press, 2011, p. 199.
 - [152] G. Papamakarios, D. Sterratt, and I. Murray, “Sequential Neural Likelihood: Fast Likelihood-free Inference with Autoregressive Flows,” in *The 22nd International Conference on Artificial Intelligence and Statistics*, Apr. 2019, p. 837–848.
 - [153] CDC, “National Health and Nutrition Examination Survey Data,” *Centers for Disease Control and Prevention (CDC). National Center for Health Statistics (NCHS). Hyattsville, MD: U.S. Department of Health and Human Services, Centers for Disease Control and Prevention*, 2013.
 - [154] R. Ramler, G. Buchgeher, C. Klammer, M. Pfeiffer, C. Salomon, H. Thaller, and L. Linsbauer, “Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases,” in *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud*, D. Winkler, S. Biffl, and J. Bergsmann, Eds. Cham: Springer International Publishing, 2019, vol. 338, p. 125–148.
 - [155] I. Robinson, J. Webber, and E. Eifrem, *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.", 2015.
 - [156] R. Angles and C. Gutierrez, “Survey of graph database models,” *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 1, 2008.
 - [157] R. Angles, “A comparison of current graph database models,” in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*. IEEE, 2012, p. 171–177.
 - [158] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca, “Static architecture-conformance checking: An illustrative overview,” *IEEE software*, vol. 27, no. 5, p. 82–89, 2010.
 - [159] T. L. Alves, J. Hage, and P. Rademaker, “A comparative study of code query technologies,” in *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*. IEEE, 2011, p. 145–154.
 - [160] T. Zhang, M. Pan, J. Zhao, Y. Yu, and X. Li, “An open framework for semantic code queries on heterogeneous repositories,” in *Theoretical Aspects of Software Engineering (TASE), 2015 International Symposium on*. IEEE, 2015, p. 39–46.
 - [161] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, p. 590–604.

-
- [162] R.-G. Urma and A. Mycroft, “Source-code queries with graph databases—with application to programming language usage and evolution,” *Science of Computer Programming*, vol. 97, p. 127–134, 2015.
 - [163] O. Goonetilleke, D. Meibusch, and B. Barham, “Graph data management of evolving dependency graphs for multi-versioned codebases,” in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, p. 574–583.
 - [164] N. Hawes, B. Barham, and C. Cifuentes, “Frappé: Querying the linux kernel dependency graph,” in *Proceedings of the GRADES’15*. ACM, 2015, p. 4.
 - [165] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.
 - [166] G. Fleck, W. Kirchmayr, M. Moser, L. Nocke, J. Pichler, R. Tober, and M. Witschil, “Experience report on building astm based tools for multi-language reverse engineering,” in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, p. 683–687.
 - [167] K. H. John and M. Tiegelkamp, *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science & Business Media, 2010.
 - [168] H. Prähofer, F. Angerer, R. Ramler, and F. Grillenberger, “Static code analysis of iec 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application,” *IEEE Transactions on Industrial Informatics*, vol. 13, no. 1, p. 37–47, 2017.
 - [169] F. Angerer, H. Prähofer, R. Ramler, and F. Grillenberger, “Points-to analysis of iec 61131-3 programs: Implementation and application,” in *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE, 2013, p. 1–8.
 - [170] H. Prahofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger, “Opportunities and Challenges of Static Code Analysis of IEC 61131-3 Programs,” in *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2012.
 - [171] G. Buchgeher, R. Weinreich, and H. Huber, “A platform for the automated provisioning of architecture information for large-scale service-oriented software systems,” in *Software Architecture*, Springer. Springer International Publishing, 2018, p. 203–218.
 - [172] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” *Software Testing, Verification and Reliability*, vol. 22, no. 2, p. 67–120, 2012.

- [173] G. Buchgeher, C. Ernstbrunner, R. Ramler, and M. Lusser, “Towards tool-support for test case selection in manual regression testing,” in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, p. 74–79.
- [174] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlogel, and A. Wubbeke, “Regression test selection of manual system tests in practice,” in *2011 15th European Conference on Software Maintenance and Reengineering*, March 2011, p. 309–312.
- [175] R. Ramler, C. Salomon, G. Buchgeher, and M. Lusser, “Tool support for change-based regression testing: An industry experience report,” in *International Conference on Software Quality*. Springer, 2017, p. 133–152.
- [176] H. Thaller, “Probabilistic Software Modeling,” Jun. 2018.
- [177] H. K. Ikkink, *Gradle Dependency Management*. Packt Publishing, 2015.