Probabilistic Software Modeling: A Data-driven Paradigm for Software Analysis

Hannes Thaller, Lukas Linsbauer, Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University Linz, Austria
{hannes.thaller, lukas.linsbauer, alexander.egyed}@jku.at

Rudolf Ramler
Software Competence Center Hagenberg GmbH
Austria
rudolf.ramler@scch.at

***Abstract*—Software systems are complex, and behavioral comprehension with the increasing amount of AI components challenges traditional testing and maintenance strategies. The lack of tools and methodologies for behavioral software comprehension leaves developers to testing and debugging that work in the boundaries of known scenarios. We present Probabilistic Software Modeling (PSM), a data-driven modeling paradigm for predictive and generative methods in software engineering. PSM analyzes a program and synthesizes a network of probabilistic models that can simulate and quantify the original program's behavior. The approach extracts the type, executable, and property structure of a program and copies its topology. Each model is then optimized towards the observed runtime leading to a network that reflects the system's structure and behavior. The resulting network allows for the full spectrum of statistical inferential analysis with which rich predictive and generative applications can be built. Applications range from the visualization of states, inferential queries, test case generation, and anomaly detection up to the stochastic execution of the modeled system. In this work, we present the modeling methodologies, an empirical study of the runtime behavior of software systems, and a comprehensive study on PSM modeled systems. Results indicate that PSM is a solid foundation for structural and behavioral software comprehension applications.**

***Index Terms*—probabilistic modeling, software modeling, static code analysis, dynamic code analysis, runtime monitoring, inference, simulation, deep learning, normalizing flows**

I. Introduction

Software complexity increases with every requirement, feature, revision, module, or software 2.0 (Artificial Intelligence (AI)) component that is integrated. Complexity related challenges in traditional software engineering have many tools and methodologies that mitigate and alleviate issues (e.g., requirements engineering, version control systems, unit testing). However, tight integration of AI components in programs is still in its infancy and so are the methodologies and tools that allow combined analysis, development, testing, integration, and maintenance.

We present *Probabilistic Software Modeling (PSM)*, a data-driven modeling paradigm for predictive and generative methods in software engineering. PSM is an analysis methodology for traditional software (e.g., Java [1]) that builds a *Probabilistic Model (PM)* of a program. The PM allows developers to reason about their program's semantics on the same level of abstraction as their source code (e.g., methods, fields, or classes) without changing the development process or programming language. This enables the advantages of probabilistic modeling and causal reasoning for traditional software development that are fundamental in other domains (such as medical biology, material simulation, economics, meteorology). PSM enables applications such as test-case generation, semantic clone detection, or anomaly detection seamlessly for both, traditional software as well as AI components and their randomness. Our experiments indicate that PMs can model programs and allow for causal reasoning and consistent data generation that these applications are built on.

PSM has four main aspects: *Code (Structure), Runtime (Behavior), Modeling, and Inference*. First, PSM extracts a program's *structure* via static code analysis (*Code*). The abstraction level is properties, executables, and types (e.g., fields, methods, and classes in Java) but ignores statements, allowing PSM to scale. Second, it inspects the program's *behavior* by observing its runtime (*Runtime*). This includes property accesses and executable invocations. Then, PSM combines this static structure and dynamic behavior into a probabilistic model (*Modeling*). This step also represents the main contribution of this work. Finally, predictive or generative applications (e.g., a test-case generator or anomaly detector) leverage the models via statistical inference (*Inference*).

The prototype used for the evaluation is called *Gradient*[1] and is openly available.

First, Section II views our contribution from the perspective of existing related domains. Section III introduces an illustrative example we use throughout this paper. In Section IV we motivate our contribution by providing an outlook on possible applications and research opportunities that PSM enables. Then we briefly discuss the nomenclature and background needed to understand PSM (Section V). Section VI presents the main contribution containing the general usage pragmatism and construction methodologies for PSM models on a conceptual level. A comprehensive evaluation of whether software can be transformed into statistical models is given in Section VII and discussed in Section VIII. Section XI concludes the paper.

II. 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

[1]https://github.com/jku-isse/gradient

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 [2]) or via a library in a host language (e.g., Pyro [3], PyMC [4], Edward [5]). 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* [6] or *Object-Oriented Bayesian Networks* [7], [8] 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+ [9] or Alloy [10]). *Stochastic model checking* [11] introduces uncertainty in the rigid formalism to model, e.g., natural phenomenons. Developers specify the behavior and provide the state transition probabilities in a special-purpose language (e.g., PRISM [12], PAT [13], CADP [14]). Again, PSM analyzes a program and synthesizes a PM allowing developers to work with the programming language of their choice.

Symbolic execution [15] is an analysis method that executes a program with symbols rather than concrete values (e.g., JPF-SE [16], KLEE [17], Pex [18]). It can be used to determine which input values cause specific branching points (if-else branches) in a program. *Probabilistic symbolic execution* [19] is an extension that quantifies the execution, e.g., branching points, in terms of probabilities. This is useful for applications that quantify program changes [20] or performance [21]. 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 state explosions. Furthermore, this abstraction shifts the analysis focus to the program semantics compared to the statement semantics (e.g., what happens between methods vs. what happens at the if statement).

Probabilistic debugging [22], [23] 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. Again, in contrast to PSM, they operate on statement level. Another difference is given in the methodologies life cycle. 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 error localization, by comparing multiple revisions of the same model.

Invariant detectors [24], [25], [26], [27], [28], [29] learn assertions and add them to the source code. This helps to pinpoint erroneous regions in the source code. Invariant detectors learn rules of value boundaries of statements (i.e., pre- and post-conditions), not the actual distribution. However, this distribution allows PSM to generate new data enabling causal reasoning across multiple code elements.

III. Illustrative Example

Consider as our running example the *Nutrition Advisor* that takes a person's anthropometric measurements (height and weight) and returns a textual advice based on the *Body Mass Index (BMI)*. Figure 1a shows the class diagram of the Nutrition Advisor, consisting of three core classes and the `Servlet` class. Classes considered by PSM are annotated with *Model* (e.g., `Person`). Figure 1b 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 shown). `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 textual advice based on the BMI ("You are healthy, try a ..."). Note that, for the sake of simplicity, Figure 1a only shows a subset of the code elements from the real Nutrition Advisor (e.g., `Person.name` or `Person.age` are omitted). Given a program such as the Nutrition Advisor, PSM can be used to build a network of probabilistic models with the same structure and behavior.

IV. 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.

A. Predictive Applications

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

Visualization and Comprehension [30], [31], [32] applications help to understand software and its behavior. This includes the visualization of code elements and non-functional attributes such as performance. The PMs are the source of the visualization showing the global but also contextual behavior across code elements. For example, Figure 2b visualizes the `height`-property in which typical and less typical values can be seen in a blink. $P(Height \mid Gender = Female)$ visualizes a context-aware behavior how gender affects the height.

Semantic Clone-Detection [33], [34] applications detect syntactically different but semantically equivalent code fragments, e.g., the iterative and recursive version of an algorithm. Traditionally, clone detection compares source code fragments focusing on exact or slightly adapted clones. However, semantic equality is beyond purely static properties of source code. PSM can detect method level clones by comparing their models. The comparison can be realized, for example, via statistical tests on sampled data [35], [36], [37] (simple automated decision), via visualization techniques such as Q-Q plots [38] (comprehensive manual decision), or a combination these.

Anomaly Detection [24], [39], [40], [41] applications measure the divergence between a persisted PSM model and a newly collected observation. These applications can be deployed into a live system, in which components are monitored and checked against their models. A threshold checks for unlikely runtime observations x (i.e., $p(Weight = weight_{new}) < .1$) triggering additional actions in cause of a failure. x and its effects on other elements can then be investigated with, e.g., visualization