Towards Fault Localization
via Probabilistic Software Modeling

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

Stefan Fischer
Software Competence Center Hagenberg GmbH
Austria
stefan.fischer@scch.at

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.

Index Terms—fault localization, probabilistic modeling, multivariate testing, software modeling, static code analysis, dynamic code analysis, runtime monitoring, inference, simulation, deep learning

I. 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) [1], [2] traditionally rank statements by their likelihood of containing a fault. This leads to localization weaknesses for complex faults that span multiple lines [3] (76% of faults) or that are caused by the omission of statements (30% of faults) [4], [5].

We propose Fault Localization via Probabilistic Software Modeling (FL-PSM). PSM [6] 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.

II. RUNNING EXAMPLE

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 1 (left) shows the four classes of the Nutrition Advisor. The sequence diagram in Figure 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, ...").

III. BACKGROUND

Probabilistic Software Modeling [6] 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 PM that represents an executable. Each 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 code element 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).
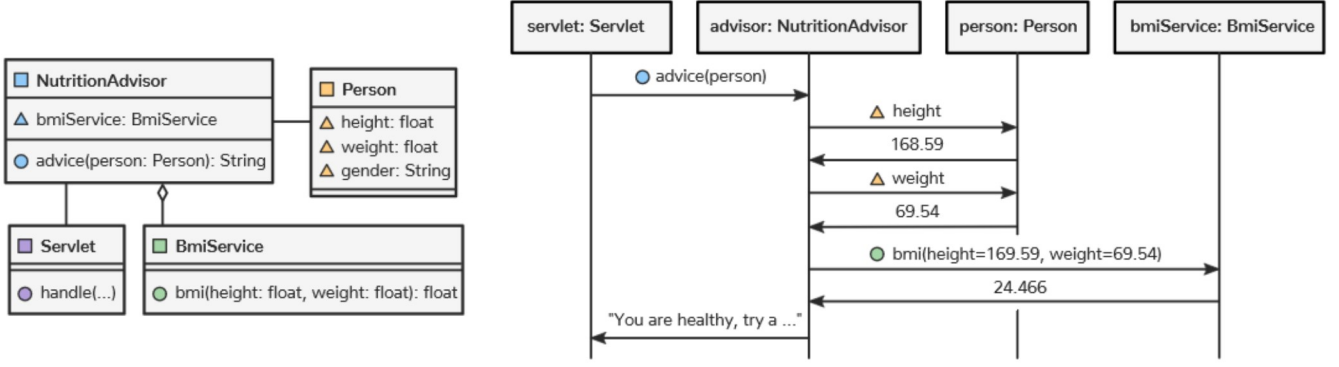
Fig. 1. Class Diagram (left) and Sequence Diagram (right) of the Nutrition Advisor [6].

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.

PMs in this work are Non-Volume Preserving Transformations (NVPs) [7], [8], which are general and expressive flow-based density estimators. Each NVP is built via neural networks 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 by our previous work [6] and Dinh [7], [8].

IV. 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,

$$LL_{D^{alt}} = \frac{1}{N} \sum_i^N p_{M^{null}} \left(D_i^{alt} \right) \qquad (1)$$

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

$$LL_{D^{alt}} - LL_{D^{null}} < c \qquad (2)$$

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.

V. 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 [9] 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 IV 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.

A. Regression Fault

Figure 2 shows the runtime behavior of a subset of code elements of the *Person.init* and *NutritionAdvisor.advice* models. Table I 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 I 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 the *Person.weight* property (rows 5 and 7). Also the invocation of the *BmiService.bmi* indicates this significant divergence (row 8).

B. Integration Fault

Figure 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
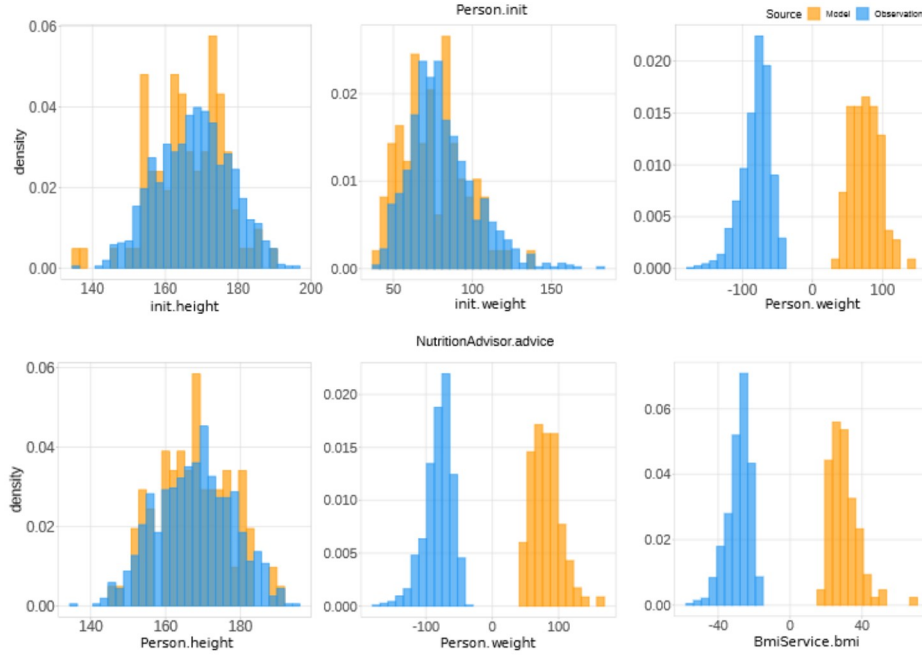
25

Fig. 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.

TABLE I. LIKELIHOOD VALUES OF A SUBSET OF ELEMENTS IN THE REGRESSION FAULT SETTING.

	Model	Element	Cardinality	LogLL	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	✔

TABLE II. 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	✔

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 II. 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 II (rows 5 and 8).

VI. 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 2 and 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.

VII. RELATED WORK

Most fault localization techniques are slice, spectrum, statistics, model, or machine learning-based [1], [3].

The most similar technique to FL-PSM is Spectrum-Based Fault Localization (SBFL) [1]. SBFL techniques observe passing and failing executions and perform statistical inference on the results. The result is a ranked list of statements, along
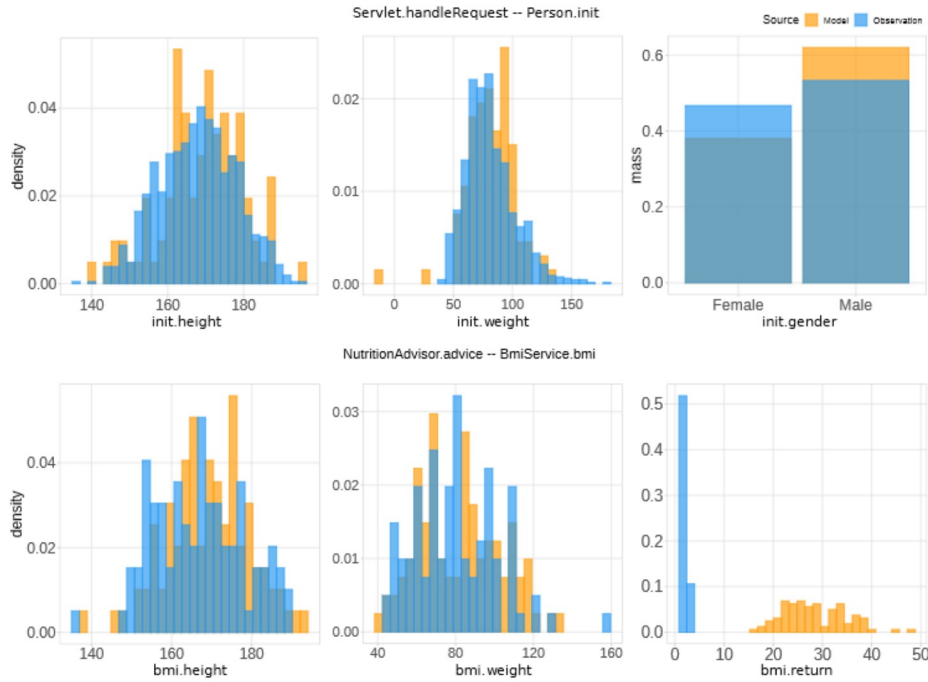
26

Fig. 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*.

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 [5] 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).

VIII. 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.

ACKNOWLEDGMENTS

REFERENCES

[1] 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, pp. 467–477.

[2] 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, pp. 290–308, Mar. 2014.

[3] 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, pp. 707–740, Aug. 2016.

[4] 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, pp. 609–620.

[5] 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.

[6] 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.

[7] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using Real NVP," *arXiv:1605.08803 [cs, stat]*, May 2016.

[8] 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.

[9] 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.