

Towards Semantic Clone Detection 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

Abstract—Semantic clones are program components with similar behavior, but different textual representation. Semantic similarity is hard to detect, and semantic clone detection is still an open issue. We present semantic clone detection via Probabilistic Software Modeling (PSM) as a robust method for detecting semantically equivalent methods. PSM inspects the structure and runtime behavior of a program and synthesizes a network of Probabilistic Models (PMs). Each PM in the network represents a method in the program and is capable of generating and evaluating runtime events. We leverage these capabilities to accurately find semantic clones. Results show that the approach can detect semantic clones in the complete absence of syntactic similarity with high precision and low error rates.

Index Terms—clone detection, semantic clone detection, probabilistic modeling, multivariate testing, software modeling, static code analysis, dynamic code analysis, runtime monitoring, inference, simulation, deep learning

I. INTRODUCTION

Copying and pasting source code fragments leads to code clones. Code clones are considered an anti-pattern as they increase maintenance costs, promote bad software design, and propagate bugs [1], [2], [3], [4], [5], [6], [7], [8]. Code clones are traditionally split into four categories. Type 1-3 [9], [10], [11] code clones are textual copies of a program fragment with possible changes. Type 4 code clones are behavioral copies of a program fragment that do not have any syntactic similarity but implement the same functionality (semantic equivalence). For example, the iterative and recursive implementations of the Fibonacci algorithm have no syntactic similarity while implementing the same functionality.

Juergens et al. [12] have shown that existing tools only have limited capabilities for detecting Type 4 clones. This limitation can also be seen in various clone detection tool comparisons [13], [14], [9], [10], [15] through the absence or explicit exclusion of Type 4 clones. Nevertheless, Type 4 clones exist and tools for detecting them are needed [12], [16].

We present *Semantic Clone Detection via Probabilistic Software Modeling (SCD-PSM)*. SCD-PSM detects semantic clones with no textual and structural similarity. First, a network of Probabilistic Models (PMs) is built via Probabilistic Software Modeling (PSM) [17]. Each PM models an executable (e.g., a method in Java) in the program under analysis. SCD-PSM leverages these PMs and their inferential capabilities to detect semantically equivalent executables. Probabilistic inference enables a similarity measure based on probabilities. These

probabilities are used to conduct statistical tests (Generalized Likelihood Ratio Test) that produce the final clone decision.

II. BACKGROUND

A basic understanding of clone detection and probabilistic software modeling is needed to understand the approach. We will use a monospace font to refer to program elements (e.g., `factorial_a`) and *italics* to refer to the corresponding model elements (e.g., *factorial_a*).

A. Clone Detection

Clone detection is the process of finding pairs of similar program fragments as illustrated in Figure 1. Figure 1 shows three different implementations of the factorial computation. Figure 1a uses a for loop, while Figure 1b uses a while loop implementation. Finally, Figure 1c uses recursion to compute the factorial of n . The clone detection process includes the representation (e.g., text fragments), pairing (e.g., of text fragments of similar size), the similarity evaluation (e.g., counting the differences in the text fragments), and the clone decision (e.g., less than 10 differences).

Representations can be, for example, text (e.g., source code), graphs (e.g., AST), or probabilistic models (like in this work). *Pairing* is the process of selecting two code fragments that are potentially a clone. Each pair is called a *candidate clone pair* (or candidate pair). The *similarity evaluation* measures the similarity between the fragments of a candidate pair. The *clone decision* labels the candidate pair as a clone given that the similarity fulfills some criteria.

The properties of the similarity metric splits clones into two groups [9]. Type 1-3 clones capture textual similarity while Type 4 clones capture semantic similarity [10], [14], [18], [9], [11], [19]. These types are increasingly challenging to detect, with Type 4 being the most complex one. Figure 1a and Figure 1b are an instance of a Type 3 clone while Figure 1a (or Figure 1b) and 1c are an instance of a Type 4 clone. Note, that the definition of a *semantic clone* is often relaxed where up-to 50% syntactic similarity of the code fragments is allowed [13], [20]. 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., equal parameter names).

```

factorial_a(n){
  product = 1
  for(i = 1; i <= n; i++){
    product *= i
  }
  return product
}

factorial_b(n){
  product = 1
  i = 1
  while(i <= n){
    product *= i
    i++
  }
  return product
}

factorial_c(n){
  if(n <= 1){
    return 1
  }
  return factorial_c(n - 1) * n
}

```

(a) A *for* implementation of factorial. (b) A *while* implementation of factorial. (c) A *recursive* implementation of factorial.

Figure 1: The *for* and *while* implementations are complex Type 3 clones in which new lines were added and some changed. The *recursive* implementation is a Type 4 clone of the *for* and *while* implementations without any syntactic resemblance.

B. Probabilistic Software Modeling

Probabilistic Software Modeling (PSM) [17] is a data-driven modeling paradigm that transforms a program into a network of Probabilistic Models (PMs). PSM extracts a program’s structure given by types, properties, and executables (e.g., classes, fields, and methods respectively in Java). This structure includes the call dependencies between the different code elements which defines the topology of the PM network. Each PM is optimized towards a program execution. The program execution can either be synthetic (e.g., random testing), from tests (e.g., developer tests), or from the program in its production environment. In the context of clone detection, synthetic program executions suffice as the results are based on differential comparisons of two elements.

Each PM represents an executable (e.g., a method in Java) in the program. Inputs are parameters, property reads, and invocation return values. Outputs are the method return value, property writes, and invocation parameters. The distinction between inputs and outputs is only a logical view from a software engineering perspective. The actual PMs are multivariate density estimators without such distinction (joint model of all variables). PMs can generate observations that are similar to the initial training data. More importantly, each model can evaluate the likelihood of data. The likelihood is used to detect behavioral equivalence between models, which is then generalized to the semantic equivalence between executables in the program.

The PMs in the network are real Non-Volume Preserving transformations (NVPs) [21], a generative likelihood-based latent-variable model for density estimation. NVPs learn a function that maps data to a known latent-space, e.g., input parameter values n and return values `product` of `factorial_a`, to a bivariate normal distributions. More formally, each NVP is a neural network that learns a bijective function $f : \mathbf{X} \mapsto \mathbf{Z}$ (with $g = f^{-1}$) between the original data $\mathbf{x} \in \mathbf{X}$ and predefined latent-variables $\mathbf{z} \in \mathbf{Z}$. The latent-variables are selected, such that sampling, conditioning, and likelihood evaluation is efficient and straightforward, e.g., via an isotropic unit norm Gaussian $\mathcal{N}(0, 1)$.

Sampling generates data $\mathbf{x} \in \mathbf{X}$ by drawing observations from the latent-variables $\mathbf{z} \sim \mathbf{Z}$ and inverting them via the

NVP to the original data-space $\mathbf{x} = g(\mathbf{z}) \sim \mathbf{X}$.

Conditioning finds a latent-space configuration (i.e., a latent-code) $\hat{\mathbf{z}}$ such that the associated data $g(\hat{\mathbf{z}}) = \hat{\mathbf{x}}$ satisfies a given condition. First a proposal code is drawn from the latent-space $\hat{\mathbf{z}}$ which is then inverted to its data form $\hat{\mathbf{x}} = g(\hat{\mathbf{z}})$. Then the error is measured on the conditioned dimensions via, e.g., Mean Squared Error (MSE). The error is used to update the latent code $\hat{\mathbf{z}}$ and the procedure is repeated until convergence. For example, one can condition the return value from `factorial_a` on to the return value of the `fibonacci` method. First, samples are drawn from the `factorial_a` model retaining only the dimension associated with the return value. Then, samples are drawn from the `fibonacci` model and the error between the return value dimensions is computed. This error is back-propagated to the latent-code which is updated according to the errors. After convergence of the optimization the `fibonacci` sample contains the same return values as imposed by the `factorial_a` sample. Furthermore, the remaining dimension n is resampled (imputed) in such a way that it adheres to the joint relationship of all the variables in `fibonacci`. Finally, `fibonacci` can be used to evaluate the likelihood of the conditioned sample.

III. APPROACH

SCD-PSM uses the models built by PSM and compares them for behavioral equivalence. The behavioral equivalence is then generalized to semantic equivalence of executables (i.e., methods).

A. Similarity Evaluation

The similarity evaluation computes the cross-wise likelihood of the models by *sampling* and *conditioning*. Given is a pair of candidate PMs, each representing an executable. The similarity evaluation starts by selecting a reference model (null-model) M_k^{null} and an alternative model (alt-model) M_k^{alt} . Then, null-dimensions M_k^{null} and alt-dimensions M_k^{alt} are selected from the models, e.g., parameter n from `factorial_a` is compared to parameter n of `factorial_b`. Then, a reference sample D_k^{null} is generated by M_k^{null} as illustrated in Figure 2 (1) representing the behavior of M_k^{null} . This reference sample is used to generate a conditioned alternative sample $D_k^{alt|null}$ (2) representing the behavior of M_k^{alt} given that dimensions \mathbf{k}

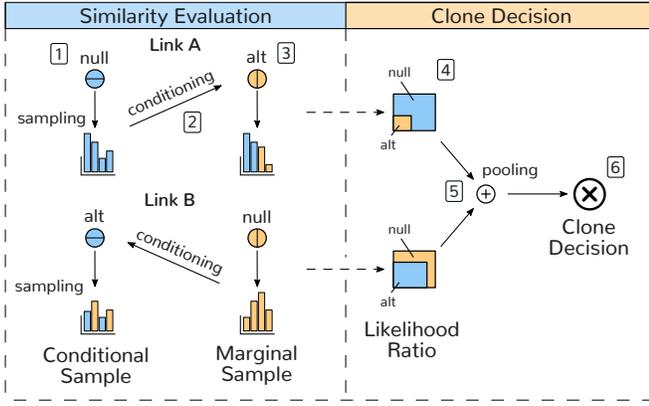


Figure 2: SCD-PSM evaluates the similarity of a pair of models via their data likelihood. The likelihoods are combined into the final clone decision.

are fixed to the behavior of M_k^{null} (3). Finally, the likelihood of D^{null} is evaluated under M_k^{null} , resulting in the base likelihood of the reference sample under the null-model LL^{null} , and $D^{alt|null}$ is evaluated under M^{alt} , resulting in the likelihood of the conditioned alternative sample under the alt-model $LL^{alt|null}$. Then, the *null* and *alt* roles are swapped and the procedure is repeated (see Figure 2 Link B).

The swapping of roles is necessary because of sub-model relationships. For instance, one model returns data distributed according to $N(0, 1)$ and the other according to $N(0, 5)$. One link will lead to a high likelihood (sub-model is null) while the other link will result in low likelihood (super-model is null).

In conclusion, the similarity evaluation tests the likelihood of the models in the context of each other. The final clone decision is based on these likelihood values.

B. Clone Decision

The final step is to combine the likelihood values from the similarity evaluation to a final decision as shown in Figure 2). The two *likelihood ratios* (4) are combined by a *pooling* operator (5) and compared against a critical value yielding the final *clone decision* (6).

More formally, the procedure makes use of the Generalized Likelihood Ratio Test (GLRT) [22]. The log-GLRT measures whether the log-likelihoods are significantly different from 0 with

$$\lambda = LL^{alt} - LL^{null}, \quad (1)$$

where LL is the log-likelihood. The null hypothesis is that the models are equal. It is rejected for small ratios $\lambda \leq c$ where c is set to an appropriate Type 1 error, i.e., 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 *Clone Decision* (6) is computed by *pooling* (5) the link results. *Hard pooling* accepts the candidate pair as a clone if the null hypothesis for *both* links could not be rejected. *Soft pooling* accepts the candidate pair as a clone if the average log-likelihood ratio of both links cannot be rejected. Hard

Table I: The 8 subject examples used in the evaluation.

Subject	Style	Clone Class	Parameter	Executable	
Factorial	iterative	A	1	1	
Factorial	recursive	A	1	1	
Fibonacci	iterative	B	1	1	
Fibonacci	recursive	B	1	1	
BubbleSort	iterative	C	1	1	
BubbleSort	recursive	C	3	2	
MergeSort	iterative	C	6	2	
MergeSort	recursive	C	8	3	
				22	12

pooling does not allow any sub-model relationship, while the soft pooling relaxes this condition slightly.

The final requirement is that a candidate pair is only accepted as a clone if the selected dimensions k of both, M_k^{null} and M^{alt} contain at least one input and output dimension. That is, methods are semantically equivalent if at least parts of their input and output relationship is equivalent.

In conclusion, the clone decision combines the link results and controls the results for a predefined false positive rate.

IV. STUDY

We implemented a prototype for SCD on top of PSM and applied the similarity evaluation given in Section III.

- 1) The study uses 8 well-known algorithms listed in Table I distributed in 3 clone classes. Each clone class is a well-understood example of semantic clones with 0% syntactic similarity. Each subject was triggered with positive uniform distributed random values.
- 2) The *Probabilistic Model Network* was computed via Gradient, a PSM prototype [17]. The same hyper-parameters were selected as in our previous reported experiments.
- 3) The *Candidate Clone Pairs* were all combinations of dimensions of the PMs. The candidate pairs were formed from all 8 subject systems.
- 4) Each valid candidate pair was tested for behavioral equality by cross-wise likelihood evaluation described in Section III-A.
- 5) The clone decision was computed via the GLRT and the results were pooled as described in Section III-B.

A. Controlled Variables

The study controls for *pooling*, the *Type 1 error*, and the *number of particles* used in the similarity evaluation (Section III-A).

Pooling describes how likelihoods are combined to the final clone decision {soft, hard} (see Section III-B).

Type 1 error, or the false-positive rate, defines the critical value c at which clones are considered significantly different {0.001, 0.01} (Section III-B). The critical value is the total Type 1 for both links.

Number of Particles are the number of samples that are sampled during the similarity evaluation for the reference sample D^{null} and the alternative sample D^{alt} . A low number of particles is faster to compute but has a higher variance in the results.

B. Response Variables

The performance of the clone detection is measured via *precision*, *recall*, and the *balanced accuracy*. These metrics are computed by the True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) proportion of detected clone instances, e.g., correctly identifying a clone pair counts towards TP.

Precision measures the performance to detect only relevant instances given by

$$\frac{TP}{TP + FP} \quad (2)$$

Recall measures the performance of detecting all relevant instances given by

$$\frac{TP}{TP + FN} \quad (3)$$

Balanced Accuracy measures the performance of detecting relevant and irrelevant instances but considers a possible imbalance between the number of relevant and irrelevant instances. It is given by

$$\frac{TP}{TP + FN} + \frac{TN}{TN + FP} \quad (4)$$

C. Experiment Results

The study results are given in Table II. Average precision was 0.991, recall was 0.797, and the balanced accuracy was 0.870. The precision across experiments was excellent, indicating that models can reliably detect behavioral equality. This is reflected in the low number of FPs. However, the FNs indicate that some positive examples are missed. Reducing the Type 1 error, i.e., falsely rejecting semantic equality, improves on the FNs. The recall was good for most evaluates. However, hard pooling caused a slight drop in the recall. The balanced accuracy is good to excellent for most experiment configurations. Perfect scores are given for experiment 1 and 12.

No effect of Pooling, Type 1, and Particles on the accuracy can be seen.

V. DISCUSSION

The results from Section IV-C are encouraging. The general performance was good to excellent. No significant difference between the different levels of pooling, Type 1, and Particles in Table II can be seen. However, a larger sample size is needed to precisely attribute effects on the performance. In the 10-particle setting a higher variance of performance can be seen caused by per-chance errors. The number of FPs is in all experiments low which is expected given that the Type 1 error was set to 0.001 and 0.01. In contrast, the number of FNs is acceptable. This is reflected in the Recall that ranges from 0.64 to 1. The

balanced accuracy shows high detection rates of the approach in most experiments settings.

VI. LIMITATIONS

SCD-PSM inherits the limitations of PSM. PSM models data. Object references are handles to containers (objects) that store data. Thereby, SCD-PSM cannot detect semantic clones of executables that solely manage object references, e.g., a collection library. However, this limitation does only hold if the program never accesses the underlying data. Furthermore, 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, invoking a wrongly implemented, e.g., sorting algorithm, would result in a false positive. Extending PSM to model distributions of sequences will alleviate this issue.

A limitation of the detection process is that it is built on runtime observations. This means that the approach can only be applied to runnable source code.

The final limitation is that the approach cannot detect Type 2-3 clones. Slight changes, e.g., flipping a plus sign to a minus, have large implications on the resulting runtime behavior. These changes will impact the semantic detection process such that the candidate clone pair will not be accepted. For example, common clone detectors will report Listing 1 and Listing 2 as clones since they differ only by one character (ignoring names and reducing minimum size). However, this does not hold for Type 4 detectors because the input and output relationship is different. In contrast, many clone detectors will not detect Listing 1 and 3 as clones because of the many additions. Type 4 detectors will report this pair as clones since the behavior of adding one to the input is identical. This hints that Type 2-3 and Type 4 clones represent detached concepts that share less common ground than expected. More importantly, this raises the question whether existing detectors that report Type 3-4 detection capabilities generalize as expected.

```
inc(a: Int): Int{
    return a + 1
}
```

Listing 1: Increment method

```
dec(a: Int): Int{
    return a - 1
}
```

Listing 2: Decrement method

```
inc(a: Int): Int{
    b = 1 * 3.12
    c = b / 2
    d = c + -0.5
    return (Int) a + d
}
```

Listing 3: Complicated increment method

Table II: Results of the clone detection experiments.

	Controlled Variables			Response Variables						
	Pooling	Type I	Particles	TP	FP	TN	FN	Precision	Recall	Balanced Accuracy
1	hard	0.001	10	22	0	14	0	1.00	1.00	1.00
2	hard	0.001	50	18	0	10	8	1.00	0.69	0.78
3	hard	0.001	100	20	0	12	4	1.00	0.83	0.89
4	soft	0.001	10	14	0	18	4	1.00	0.78	0.89
5	soft	0.001	50	22	0	10	4	1.00	0.85	0.89
6	soft	0.001	100	22	0	10	4	1.00	0.85	0.89
7	hard	0.010	10	8	0	26	2	1.00	0.80	0.94
8	hard	0.010	50	14	0	14	8	1.00	0.64	0.78
9	hard	0.010	100	14	0	14	8	1.00	0.64	0.78
10	soft	0.010	10	16	2	10	8	0.89	0.67	0.72
11	soft	0.010	50	20	0	12	4	1.00	0.83	0.89
12	soft	0.010	100	22	0	14	0	1.00	1.00	1.00

VII. RELATED WORK

Many studies have evaluated textual clones. However, there are only a few studies reporting reliable results on semantic clones without relaxing the definition of Type 4.

Rattan [11] 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.

Horwitz [23] detected textual and semantic differences in programs via a Program Representation Graph, which is similar to a Program Dependency Graph (PDG). PDG-based approaches [18], [24], [25] use (static) data and control dependencies to find similar sub-graphs between the candidates. They can detect complex Type 3 clones, e.g., Figure 1a and Figure 1b. However, the compared PDG sub-graphs are a representation of the source code; thereby, the approaches still rely on syntactic similarity [26].

Another category of semantic clone detectors are test-based methods. Test-based methods randomly trigger the execution of two candidates and measure whether equal inputs cause similar outputs. Jiang and Su [27] were able to detect semantic clones without syntactical similarities. A similar approach was presented by Deissenboeck et al. [28]. One issue with test-based clone detection is that candidates need a similar signature. Differences in data types or the number of parameters can not be effectively handled by the test-case generators or the similarity measurement. SCD-PSM works similar 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. This allows SCD-PSM to overcome the issue of signature mismatches. Furthermore, PSM abstracts the data types into text, integer, and floats mitigating data type mismatches.

Finally, the clone detector Oreo [20] has also reported Type 3 to Type 4 detection capabilities. Oreo uses a combination of representations and detection stages to find clones. Most important is the semantic similarity comparison based on

actions a method takes, e.g., accessing an array, writing a property, or invoking a method. These actions correspond, to some extent, to the dimensions of PSM models, i.e., represent entry-points of information (e.g., field accesses, invocations, etc.) Oreo counts these entry-points and compares then between the fragments in a candidate pair. No analysis of the runtime assignments is conducted, nor is the relationship between the actions analyzed like SCD-PSM does. Oreo reports many complex Type 3 and Type 4 clones up to 50% syntactic similarity based on this semantic similarity (and the additional pipeline steps). However, more research is needed to identify the weaknesses and strengths of both approaches. This highlights the need for a hard but well understood baseline dataset of Type 4 clones similar to the examples in our study but extended with a larger variety of semantic clones.

VIII. CONCLUSION AND FUTURE WORK

In this work, we presented a viable approach for semantic clone detection - Semantic Clone Detection via Probabilistic Software Modeling (SCD-PSM). SCD-PSM leverages the PMs of PSM to detect method level semantic clones with 0% syntactic similarity.

We have discussed the similarity evaluation and the clone decision that represent the central aspect of a clone detector. We evaluated the concepts on a set of well-known semantic clones that provide a hard baseline for Type 4 detectors.

Our future work is to evaluate the scalability of the approach with large programs. Furthermore, we want to compare SCD-PSM with existing Type 3 clone detectors.

In conclusion, SCD-PSM is capable of detecting semantic clones with 0% syntactic similarity.

ACKNOWLEDGMENTS

The research reported in this paper has been supported by the Austrian ministries BMVIT and BMDW, and the Province of Upper Austria in terms of the COMET - Competence Centers for Excellent Technologies Programme managed by FFG.

REFERENCES

- [1] 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, pp. 244–253.
- [2] 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, pp. 87–94.
- [3] R. C. Martin, Ed., *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2009.
- [4] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, ser. The Addison-Wesley Object Technology Series. Reading, MA: Addison-Wesley, 1999.
- [5] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Reading, Mass: Addison-Wesley, 2000.
- [6] 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.
- [7] 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, pp. 176–192, 2006.
- [8] 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, pp. 411–425.
- [9] 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.
- [10] 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, pp. 577–591, 2007.
- [11] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, Jul. 2013.
- [12] 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, pp. 78–87.
- [13] 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, pp. 131–140.
- [14] 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.
- [15] 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, pp. 84–94.
- [16] 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, pp. 2–8.
- [17] 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.
- [18] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 301–309, 2001.
- [19] 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, pp. 1–8.
- [20] 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, pp. 354–365.
- [21] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using Real NVP," *arXiv:1605.08803 [cs, stat]*, May 2016.
- [22] J. Fan, C. Zhang, and J. Zhang, "Generalized Likelihood Ratio Statistics and Wilks Phenomenon," *The Annals of Statistics*, vol. 29, no. 1, pp. 153–193, 2001.
- [23] S. Horwitz, "Identifying the Semantic and Textual Differences Between Two Versions of a Program," in *PLDI*, 1990.
- [24] 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.
- [25] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS '01. London, UK, UK: Springer-Verlag, 2001, pp. 40–56.
- [26] 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.
- [27] 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, pp. 81–92.
- [28] 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, pp. 299–308.