Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI-23)
Sister Conferences Best Papers Track

Data-Driven Invariant Learning for Probabilistic Programs (Extended Abstract)

Jialu Bao[1], **Nitesh Trivedi**[2], **Drashti Pathak**[3], **Justin Hsu**[1] and **Subhajit Roy**[2]
[1]Cornell University, Ithaca, NY, USA
[2]Indian Institute of Technology (IIT) Kanpur, India
[3]Amazon, Bengaluru, India
jb965@cornell.edu, email@justinh.su, {nitesht,subhajit}@iitk.ac.in

Abstract

The *weakest pre-expectation* framework from Morgan and McIver for deductive verification of probabilistic programs generalizes binary state assertions to real-valued *expectations* to measure expected values of expressions over probabilistic program variables. While loop-free programs can be analyzed by mechanically transforming expectations, verifying programs with loops requires finding an *invariant expectation*.

We view invariant expectation synthesis as a *regression* problem: given an input state, predict the *average* value of the post-expectation in the output distribution. With this perspective, we develop the first *data-driven* invariant synthesis method for probabilistic programs. Unlike prior work on probabilistic invariant inference, our approach learns piecewise continuous invariants without relying on template expectations. We also develop a data-driven approach to learn *sub-invariants* from data, which can be used to upper- or lower-bound expected values. We implement our approaches and demonstrate their effectiveness on a variety of benchmarks from the probabilistic programming literature.

1 Introduction

Probabilistic programs are standard imperative programs augmented with a *sampling* command—a mechanism to draw random samples from a probability distribution. Probabilistic programs provide a formal way to describe randomized computations. While the mathematical semantics of such programs is fairly well-understood [Kozen, 1981], verification methods remain an active area of research. Existing automated techniques are either limited to specific properties (e.g., [Smith *et al.*, 2019; Albarghouthi and Hsu, 2018; Carbin *et al.*, 2013; Roy *et al.*, 2021]), or target simpler computational models [Baier *et al.*, 1997; Kwiatkowska *et al.*, 2011; Dehnert *et al.*, 2017].

Reasoning about Expectations. One of the earliest methods for reasoning about probabilistic programs is through *expectations*. Originally proposed by Kozen [Kozen, 1985], expectations generalize standard, binary assertions to quantitative, real-valued functions on program states. Morgan and McIver further developed this idea into a powerful framework for reasoning about probabilistic imperative programs, called the *weakest pre-expectation calculus* [Morgan *et al.*, 1996; McIver and Morgan, 2005]. The weakest pre-expectation calculus defines the *weakest pre-expectation* (wpe) operator that takes an expectation E and a program P to produce an expectation E' such that $E'(\sigma)$ is the expected value of E in the output distribution $[\![P]\!]_\sigma$. In this way, the wpe operator can be viewed as a generalization of Dijkstra's weakest pre-conditions calculus [Dijkstra, 1975] to probabilistic programs. The wpe operator has two key strengths: first, it enables reasoning about probabilities and expected values; second, when P is a loop-free program, it is possible to transform $\mathsf{wpe}(P, E)$ into a form that does not mention the program P via simple, mechanical manipulations, essentially analyzing the effect of the program on the expectation through syntactically transforming E. However, there is a caveat: the wpe of a loop is defined as a least fixed point, and it is generally difficult to simplify this into a more tractable form. Fortunately, the wpe operator satisfies a *loop rule* that simplifies reasoning about loops: if we can find an expectation I satisfying an *invariant* condition, then we can easily bound the wpe of a loop. Checking the invariant condition involves analyzing just the body of the loop, rather than the entire loop. Thus, finding invariants becomes the primary bottleneck towards automated reasoning about probabilistic programs.

Our Approach. Our approach to synthesizing loop invariants for probabilistic programs is inspired by data-driven invariant learning techniques [Flanagan and Leino, 2001; Ernst *et al.*, 2007] for regular programs. In these methods, the program is executed with a variety of inputs to produce a set of execution traces. This data is viewed as a training set, and a machine learning algorithm is used to find a classifier describing the invariant. Data-driven techniques reduce the reliance on templates, and can treat the program as a black box—the precise implementation of the program need not be known, as long as the learner can execute the program to gather input and output data. But to extend the data-driven method to the probabilistic setting, there are a few key challenges:

- **Quantitative invariants.** While the logic of expectations resembles the logic of standard assertions, an important difference is that expectations are *quantitative*: they map program states to real numbers, not a binary

yes/no. While invariant learning for deterministic programs is a *classification* task (i.e., predicting a binary label given a program state), our probabilistic invariant learning is closer to a *regression* task (i.e., predicting a number given a program state).

- **Stochastic data.** Invariant learning for deterministic programs assumes that the program behaves like a *function*: a given input state always leads to the same output state. In contrast, a probabilistic program takes an input state to a distribution over outputs. Since we are only able to observe a single draw from the output distribution each time we run the program, execution traces in our setting are inherently *noisy*. Accordingly, we cannot hope to learn an invariant that fits the observed data perfectly, even if the program has an invariant—our learner must be robust to noisy training data.
- **Complex learning objective.** To fit a probabilistic invariant to data, the logical constraints defining an invariant must be converted into a regression problem with a loss function suitable for standard machine learning algorithms and models. While typical regression problems relate the unknown quantity to be learned to known data, the conditions defining invariants are somehow self-referential: they describe how an unknown invariant must be related to itself. This feature makes casting invariant learning as machine learning a difficult task.

Contributions. The contributions of our work are:

- We provide a general algorithm, EXIST (EXpectation Invariant SynThesis), for learning invariants for probabilistic programs. EXIST executes the program multiple times on a set of input states, and then uses machine learning algorithms to learn models encoding possible invariants. A counterexample guided inductive synthesis (CEGIS) loop iteratively expands the dataset after encountering incorrect candidate invariants.
- We describe instantiations of EXIST tailored for handling two problems: learning *exact invariants*, and learning *sub-invariants*. Our method for exact invariants learns a *model tree* [Quinlan, 1992], a generalization of binary decision trees to regression. The constraints for sub-invariants are more difficult to encode as a regression problem, and our method learns a *neural model tree* [Yang *et al.*, 2018] with a custom loss function. While the models differ, both algorithms leverage off-the-shelf learning algorithms.
- We evaluate our implementation of EXIST on a large set of benchmarks. EXIST learns invariants for examples considered in prior work and new, more difficult versions that are beyond the reach of prior work.

This article is the extended abstract of the conference paper [Bao *et al.*, 2022a] published in CAV'22. An extended version of the article is also available [Bao *et al.*, 2022b].

2 Preliminaries

Probabilistic Programs. We will consider programs written in **pWhile**, a basic probabilistic imperative language with the following grammar, where e is a boolean or numerical expression.

$$P \coloneqq \mathbf{skip} \mid x \leftarrow e \mid x \xleftarrow{\$} d \mid P\,;P \\ \mid \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ P \mid \mathbf{while}\ e : P$$

All commands P are interpreted into maps from memories to distributions over memories [Kozen, 1981]. We write $[\![P]\!]_\sigma$ for the output distribution of program P from initial state σ. Since we will be interested in running programs on concrete inputs, *we will assume throughout that all loops are almost surely terminating*; this property can often be established by other methods (e.g., [Chatterjee *et al.*, 2016a; Chatterjee *et al.*, 2016b; McIver *et al.*, 2018]).

Weakest Pre-expectation Calculus. Morgan and McIver's *weakest pre-expectation transformer* (wpe) takes a program P and an expectation E to another expectation E', called the *pre-expectation*; wpe is defined in Fig. 1. The case for loops involves the least fixed-point (lfp) of $\Phi_E^{\mathsf{wpe}} \coloneqq \lambda X.([e] \cdot \mathsf{wpe}(P, X) + [\neg e] \cdot E)$, the *characteristic function* of the loop with respect to wpe [Kaminski *et al.*, 2016]. The characteristic function is monotone on the complete lattice $\mathcal{E}$, so the least fixed-point exists by the Kleene fixed-point theorem. The key property of the wpe transformer is that for any program P, $\mathsf{wpe}(P, E)(\sigma)$ is the expected value of E over the output distribution $[\![P]\!]_\sigma$. Intuitively, the weakest pre-expectation calculus provides a syntactic way to compute the expected value of an expression E after running a program P, except when the program is a loop. For a loop, the least fixed point definition of $\mathsf{wpe}(\mathbf{while}\ e : P, E)$ is hard to compute.

3 Problem Statement

Analogous to when analyzing the weakest pre-conditions of a loop, knowing a loop *invariant* or *sub-invariant* expectation enables one to easily bound the loop's weakest pre-expectations. However, (sub)invariant expectations can be difficult to find. We develop an algorithm to synthesize invariants and sub-invariants of probabilistic loops. More specifically, our algorithm tackles the following two problems:

1. **Finding exact invariants:** Given a loop $\mathbf{while}\ G : P$ and an expectation postE as input, we want to find an expectation I such that
$$I = \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I) \coloneqq [G] \cdot \mathsf{wpe}(P, I) + [\neg G] \cdot \mathsf{postE}. \quad (1)$$
Such an expectation I is an *exact invariant* of the loop with respect to postE. Since $\mathsf{wpe}(\mathbf{while}\ G : P, \mathsf{postE})$ is a fixed point of $\Phi^{\mathsf{wpe}}_{\mathsf{postE}}$, $\mathsf{wpe}(\mathbf{while}\ G : P, \mathsf{postE})$ has to be an exact invariant of the loop. Furthermore, when $\mathbf{while}\ G : P$ is almost surely terminating and postE is upper-bounded, the existence of an exact invariant I implies $I = \mathsf{wpe}(\mathbf{while}\ e : P, E)$.
2. **Finding sub-invariants:** Given a loop $\mathbf{while}\ G : P$ and expectations $\mathsf{preE}, \mathsf{postE}$, we aim to learn an expectation I such that
$$I \leq \Phi^{\mathsf{wpe}}_{\mathsf{postE}}(I) \coloneqq [G] \cdot \mathsf{wpe}(P, I) + [\neg G] \cdot \mathsf{postE} \quad (2)$$
$$\mathsf{preE} \leq I. \quad (3)$$

$$\mathsf{wpe}(\mathbf{skip}, E) \coloneqq E \qquad \mathsf{wpe}(x \leftarrow e, E) \coloneqq E[e/x]$$
$$\mathsf{wpe}(x \xleftarrow{\$} d, E) \coloneqq \lambda\sigma. \sum_{v \in \mathcal{V}} [\![d]\!]_\sigma(v) \cdot E[v/x] \qquad \mathsf{wpe}(P\,;Q, E) \coloneqq \mathsf{wpe}(P, \mathsf{wpe}(Q, E))$$
$$\mathsf{wpe}(\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q, E) \coloneqq [e] \cdot \mathsf{wpe}(P, E) + [\neg e] \cdot \mathsf{wpe}(Q, E)$$
$$\mathsf{wpe}(\mathbf{while}\ e : P, E) \coloneqq \mathsf{lfp}(\lambda X.\ [e] \cdot \mathsf{wpe}(P, X) + [\neg e] \cdot E)$$

Figure 1: Morgan and McIver's weakest pre-expectation transformer (wpe)

The first inequality says that I is a sub-invariant: on states that satisfy G, the value of I lower bounds the expected value of itself after running one loop iteration from initial state, and on states that violate G, the value of I lower bounds the value of postE. Any sub-invariant lower-bounds the weakest pre-expectation of the loop, i.e., $I \leq \mathsf{wpe}(\mathbf{while}\ G : P, E)$ [Kaminski, 2019]. Together with the second inequality $\mathsf{preE} \leq I$, the existence of a sub-invariant I ensures that preE lower-bounds the weakest pre-expectation.

4 Methodology

Our data-driven method runs a Counterexample Guided Inductive Synthesis (CEGIS), but differs from conventional CEGIS tools in two ways. First, candidates are synthesized by fitting a machine learning model to data consisting of program traces starting from random input states. Our target programs are also probabilistic, introducing a second source of randomness to program traces. Second, our approach seeks high-quality counterexamples—violating the target constraints as much as possible—in order to improve synthesis. For synthesizing invariants and sub-invariants, such counterexamples can be generated by using a computer algebra system to solve an optimization problem.

EXIST takes a probabilistic program $\mathcal{P}$, a post-expectation or a pair of pre/post-expectation $pexp$, to produce a loop (sub)invariant expectations. The user can provide two hyper-parameters, N_{runs} and N_{states}, to control the data-generation process. With these inputs, EXIST proceeds below:

Generate Features. EXIST starts by generating a list of features $feat$, which are numerical expressions formed by program variables used in $\mathcal{P}$.

Sample Initial States. Next, EXIST samples N_{states} number of initial $states$ by uniformly sampling values of program variables from their respective domains.

Sample Training Data. For learning exact invariants, from each initial state s_i, EXIST runs $\mathcal{P}$ *until termination* for N_{runs} times to get the list of final states $\{\sigma_1, \ldots, \sigma_{N_{runs}}\}$ and then produces the training example:

$$(s_i, v_i) \coloneqq \left(s_i, \frac{1}{N_{runs}} \sum_{j=1}^{N_{runs}} \mathsf{postE}(\sigma_j) \right).$$

Above, the value v_i is the empirical mean of postE in the output state of running $\mathcal{P}$ from initial state s_i; as N_{runs} grows large, this average value approaches the true expected value $\mathsf{wpe}(\mathcal{P}, \mathsf{postE})(s_i)$.

For learning subinvariants, from each s_i, EXIST runs *single iteration* of $\mathcal{P}$ (and then restart at s_i) for N_{runs} times to get a list of output states $\{\sigma_1, \ldots, \sigma_{N_{runs}}\}$ and then produces the training example:

$$(s_i, v_i) \coloneqq (s_i, \{\sigma_1, \ldots, \sigma_{N_{runs}}\}).$$

(Machine) Learn an Invariant. In each iteration of the CEGIS loop, first the learner $\mathsf{learnInv}$ trains models to minimize violation of the required inequalities (i.e., Eq. (1) for learning exact invariants; Eqs. (2) and (3) for learning sub-invariants) on $data$.

For learning exact invariants, we choose regression models to be model trees, which are decision trees with customizable models instead of labels on leaves. While our methods can potentially work for other leaf models, we focus on linear models or multiplicative models (which are linear models on the logarithm space of the data) because of their simplicity and expressiveness. This class of model trees suits our goal because they can be easily translated into numerical expressions, which are usually the form people use to encode expectations. With the training $data = \{(s_1, v_1), \ldots, (s_{N_{states}}, v_{N_{states}})\}$, where each v_i approximates $\mathsf{wpe}(\mathcal{P}, \mathsf{postE})(s_i)$, we train a model tree T that takes the feature vector of s_i, denoted $\mathcal{F}(s_i)$, as input and predicts v_i. We use the standard mean-square-error to measure the error between predicted values $T(\mathcal{F}(s_i))$ and the target value v_i and apply off-the-shelf tools for training.

For learning sub-invariants, we initially also want to use model trees, but the loss is more complicated and the standard model tree training mechanism based on divide-and-conquer no longer applies. A remedy is to apply gradient descent to train on that loss, but standard model trees are not differentiable. To bridge the gap, EXIST trains neural model trees [Yang *et al.*, 2018], which are neural networks for approximating model trees, to fit the data and then translate the learned neural networks back to model trees.

Extract Expectation Invariants from Models. EXIST now translates the learned models into numerical expressions and regards them as the set of $candidate(sub)invariants$.

Verification. For each generated candidate invariant inv, EXIST attempts to verify if inv satisfies the required constraints. If it cannot find any program state where inv violates the required set of constraints, the verifier returns inv as a valid invariant (or sub-invariant).

Name	postE	Learned Invariant
Bin1	n	$x + [n < M] \cdot (M \cdot p - n \cdot p)$
Fair	$count$	$(count + [c1 + c2 == 0] \cdot$ $(p1 + p2)/(p1 + p2 - p1 \cdot p2))$
Gambler	z	$z + [x > 0 \text{ and } y > x] \cdot x \cdot (y - x)$
Geo0	z	$z + [flip == 0] \cdot (1 - p_1)/p_1$
Sum0	x	$x + [n > 0] \cdot (0.5 \cdot p \cdot n^2 + 0.5 \cdot p \cdot n)$

Table 1: Exact Invariants generated by EXIST

Name	postE	preE	Learned Sub-invariant
Gambler	z	$x \cdot (y - x)$	$z + [x > 0 \text{ and } y > x] \cdot$ $x \cdot (y - x)$
Geo0	z	$[flip == 0] \cdot$ $(1 - p_1))$	$z + [flip == 0] \cdot$ $(1 - p_1)$
LinExp	z	$z + [n > 0] \cdot 2$	$[n > 0] \cdot (n + 1)$
		$z + [n > 0] \cdot 2 \cdot n$	$z + [n > 0] \cdot 2 \cdot n$
RevBin	z	$z + [x > 0] \cdot x$	$z + [x > 0] \cdot x/p$
		z	z

Table 2: Sub-invariants generated by EXIST

Data Augmentation. If inv violates any inequalities, EXIST produces a set of counterexample program states that are added to the set of initial states. To ensure that the generated counterexamples are effective data-points, EXIST looks for program states that *maximize* inv's violation of required inequalities. EXIST then adds states in counterexamples to $states$ and augments the existing $data$ with new data points generated on these counterexample states. The data augmentation process ensures that the synthesis algorithm collects more and more initial states, some randomly generated (sampleStates) and some from prior counterexamples, guiding the learner towards better candidates.

EXIST repeats the CEGIS loop by re-learning new models on the augmented dataset. Like most of the other CEGIS-based tools, our method is sound but not complete, i.e., if the algorithm returns an expectation then it is guaranteed to be an exact invariant or sub-invariant, but the algorithm might never return an answer; in practice, we set a timeout to force the procedure to terminate.

5 Evaluations

We implemented our prototype in Python, using sklearn and tensorflow to fit model trees and neural model trees, and Wolfram Alpha to verify and perform counterexample generation. We have evaluated our tool on a set of 18 benchmarks drawn from different sources in prior work [Gretz *et al.*, 2013; Chen *et al.*, 2015; Kaminski and Katoen, 2017]. We summarize our findings as follows:

- EXIST successfully synthesized and verified exact invariants for 14/18 benchmarks within a timeout of 300 seconds. Our tool was able to generate these 14 invariants in reasonable time, taking between 1 to 237 seconds. The sampling phase dominates the time in most cases. We also compare EXIST with a tool from prior literature, MORA [Bartocci *et al.*, 2020]. We found that MORA can only handle a restrictive set of programs and cannot handle many of our benchmarks. We also discuss how our work compares with a few others in (Section 6).
- To evaluate sub-invariant learning, we created multiple problem instances for each benchmark by supplying different pre-expectations. On a total of 34 such problem instances, EXIST was able to infer correct invariants in 27 cases, taking between 7 to 102 seconds.

Table 1 and Table 2 contain some of the exact invariants and sub invariants generated by EXIST.

6 Related Work

Invariant Generation for Probabilistic Programs. The PRINSYS system [Gretz *et al.*, 2013] employs a template-based approach to guide the search for probabilistic invariants. [Chen *et al.*, 2015] proposed a counterexample-guided approach to find polynomial invariants, by applying Lagrange interpolation. Unlike PRINSYS, this approach does not need templates; however, invariants involving guard expressions—common in our examples—cannot be found, since they are not polynomials. Additionally, [Chen *et al.*, 2015] uses a weaker notion of invariant, which only needs to be correct on certain initial states; our tool generates invariants that are correct on all initial states. [Feng *et al.*, 2017] improves on [Chen *et al.*, 2015] by using *Stengle's Positivstellensatz* to encode invariants constraints as a semidefinite programming problem. However, their approach cannot synthesize piecewise linear invariants.

Another line of work applies *martingales* to derive insights of probabilistic programs. [Chakarov and Sankaranarayanan, 2013] showed several applications of martingales in program analysis, and [Barthe *et al.*, 2016] gave a procedure to generate candidate martingales for a probabilistic program; however, this tool gives no control over which expected value is analyzed – the user can only guess initial expressions and the tool generates valid bounds, which may not be interesting.

Data-driven Invariant Synthesis. We are not aware of other data-driven methods for learning probabilistic invariants, but a recent work [Abate *et al.*, 2021] proves probabilistic termination by learning ranking supermartingales encoded as two-layer neural networks from trace data.

Data-driven inference of invariants for deterministic programs has been an area of interest, starting from DAIKON [Ernst *et al.*, 2007]. For inductive invariants, ICE learning [Garg *et al.*, 2016] uses a modified decision tree learning algorithm and HANOI [Miltner *et al.*, 2020] uses a CEGIS-based engine that alternates between weakening and strengthening candidates. Recent work uses neural networks to learn invariants [Si *et al.*, 2018]. Data from fuzzing has been used for *almost correct* inductive invariants [Lahiri and Roy, 2022] for programs with closed-box operations. As a data-driven learning task, invariant inference for deterministic programs can be treated as a classification problem while that for probabilistic programs becomes a regression task.

References

[Abate *et al.*, 2021] Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. Learning probabilistic termination proofs. In *CAV*, 2021.

[Albarghouthi and Hsu, 2018] Aws Albarghouthi and Justin Hsu. Synthesizing coupling proofs of differential privacy. In *POPL*, 2018.

[Baier *et al.*, 1997] Christel Baier, Edmund M. Clarke, Vasiliki Hartonas-Garmhausen, Marta Z. Kwiatkowska, and Mark Ryan. Symbolic model checking for probabilistic processes. In *ICALP*, 1997.

[Bao *et al.*, 2022a] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. Data-driven invariant learning for probabilistic programs. In *CAV*, 2022.

[Bao *et al.*, 2022b] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. Data-driven invariant learning for probabilistic programs. In *arXiv 2106.05421*, 2022.

[Barthe *et al.*, 2016] Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. Synthesizing probabilistic invariants via Doob's decomposition. In *CAV*, 2016.

[Bartocci *et al.*, 2020] Ezio Bartocci, Laura Kovács, and Miroslav Stankovič. Mora-automatic generation of moment-based invariants. In *TACAS*, 2020.

[Carbin *et al.*, 2013] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[Chakarov and Sankaranarayanan, 2013] Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In *CAV*, 2013.

[Chatterjee *et al.*, 2016a] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Termination analysis of probabilistic programs through Positivstellensatz's. In *CAV*, 2016.

[Chatterjee *et al.*, 2016b] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *POPL*, 2016.

[Chen *et al.*, 2015] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. Counterexample-guided polynomial loop invariant generation by Lagrange interpolation. In *CAV*, 2015.

[Dehnert *et al.*, 2017] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *CAV*, 2017.

[Dijkstra, 1975] Edsger W. Dijkstra. Guarded commands, nondeterminancy and a calculus for the derivation of programs. In *Language Hierarchies and Interfaces*, 1975.

[Ernst *et al.*, 2007] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 2007.

[Feng *et al.*, 2017] Yijun Feng, Lijun Zhang, David N Jansen, Naijun Zhan, and Bican Xia. Finding polynomial loop invariants for probabilistic programs. In *ATVA*, 2017.

[Flanagan and Leino, 2001] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, 2001.

[Garg *et al.*, 2016] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *POPL*, 2016.

[Gretz *et al.*, 2013] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Prinsys - on a quest for probabilistic loop invariants. In *QEST*, 2013.

[Kaminski and Katoen, 2017] Benjamin Lucien Kaminski and Joost-Pieter Katoen. A weakest pre-expectation semantics for mixed-sign expectations. In *LICS*, 2017.

[Kaminski *et al.*, 2016] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. In *ESOP*, 2016.

[Kaminski, 2019] Benjamin Lucien Kaminski. *Advanced weakest precondition calculi for probabilistic programs*. PhD thesis, RWTH Aachen University, Germany, 2019.

[Kozen, 1981] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3), 1981.

[Kozen, 1985] Dexter Kozen. A probabilistic PDL. *Journal of Computer and System Sciences*, 30(2), 1985.

[Kwiatkowska *et al.*, 2011] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, 2011.

[Lahiri and Roy, 2022] Sumit Lahiri and Subhajit Roy. Almost correct invariants: Synthesizing inductive invariants by fuzzing proofs. In *ISSTA*, 2022.

[McIver and Morgan, 2005] Annabelle McIver and Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. Springer Science & Business Media, 2005.

[McIver *et al.*, 2018] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. A new proof rule for almost-sure termination. In *POPL*, 2018.

[Miltner *et al.*, 2020] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. Data-driven inference of representation invariants. In *PLDI*, 2020.

[Morgan *et al.*, 1996] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *TOPLAS*, 1996.

[Quinlan, 1992] J. R. Quinlan. Learning with continuous classes. In *AJCAI*, volume 92, 1992.

[Roy *et al.*, 2021] Subhajit Roy, Justin Hsu, and Aws Albarghouthi. Learning differentially private mechanisms. In *SP*, 2021.

[Si *et al.*, 2018] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *NeurIPS*, 2018.

[Smith *et al.*, 2019] Calvin Smith, Justin Hsu, and Aws Albarghouthi. Trace abstraction modulo probability. In *POPL*, 2019.

[Yang *et al.*, 2018] Yongxin Yang, Irene Garcia Morillo, and Timothy M. Hospedales. Deep neural decision trees. *CoRR*, 2018.