University of Oxford

Department of Computer Science

Learning Martingales

Candidate Number:	1058369
Word Count:	15,739
Degree:	MSc Advanced Computer Science
Term:	Trinity Term 2022

Abstract

A family of novel algorithms are presented, which use machine learning to verify probabilistic programs. These methods synthesise indicating supermartingales (ISMs) and repulsing supermartingales (RepSMs), that bound the probability that a safety property will be violated in a probabilistic program.

These methods construct a martingale by using a neural network to learn a candidate martingale. This candidate martingale is verified by using satisfiability modulo theories (SMT). If a counterexample is produced, it is added to the training data, so the neural network can learn a new candidate. This continues until a valid martingale has been constructed. This process of incremental learning is called counterexample-guided inductive synthesis (CEGIS).

These methods are implemented in a new tool, and their value is demonstrated in a range of benchmarks accumulated from previous literature, as well as a selection of new benchmarks that cannot be handled by previous methods. Further, these empirical results allow for comparisons between the different methods to be made, and an evaluation of the overall approach to be given.

Contents

List of Abbreviations

AST . . . . . Almost-Sure Termination
CAS . . . . . Computer Algebra System
CEGIS . . . . Counter-Example Guided Inductive Synthesis
ISM . . . . . Indicating Supermartingale
PAST . . . . . Positive Almost-Sure Termination
PDF . . . . . Probability Density Function
PWL . . . . . Continuous Piecewise Linear
ReLU . . . . . Rectified Linear Unit
RepSM . . . . Repulsing Supermartingale
RSM . . . . . Ranking Supermartingale
SOR . . . . . Sum of ReLU
SOS . . . . . Sum of Squares
SMT . . . . . Satisfiability Modulo Theories

Chapter 1

Introduction

1.1 Motivation

Probabilistic programming extends classical imperative programming with the capability to sample from a probability distribution. It has a wide range of applications, including stochastic control systems, randomised algorithms, cryptographic protocols and Bayesian inference. This makes verification of probabilistic programs a compelling field of research.

An example of such a program is given below: the faulty marble collector. In each iteration, either a red marble or a blue marble is collected. The variables *red* and *blue* record the number of red and blue marbles respectively that are left to collect. However, in each iteration there is a small risk that an error occurs, which is modelled by *error* being set to 1. If this is the case, the marble collector will fail.

$error = 0$
while $red \geq 1$ **or** $blue \geq 1$:
    $assert(error \neq 1)$
    $p \sim \mathsf{Bernoulli}(0.999)$
    if $p == 1$:
        $q \sim \mathsf{Bernoulli}(0.5)$
        if $q == 1$:
            $red = red - 1$
        else:
            $blue = blue - 1$
    else:
        $error = 1$

A natural verification question is to establish an upper bound for the probability of failure. This can be achieved by constructing an indicating supermartingale[1] (ISM) [Takisaka et al., 2021; Chatterjee et al., 2022]. An ISM is a formal witness that a subset of states is reachable with probability at most p. Surprisingly, despite the simplicity of the faulty marble collector, existing tools are unable to synthesise

[1]Prior literature uses different terms for this structure including nonnegative repulsing supermartingales and stochastic invariant indicators.

an ISM. The difficulty is caused by the disjunction in the loop guard. Instead, if the guard was just $red \geq 1$, or alternatively just $blue \geq 1$, an ISM could be found.

The goal of synthesis is to find a martingale, such that for every program state, the required properties hold. In other words, the problem is to decide the validity of $\exists x.\forall y.P(x, y)$, by providing a witness for the martingale x. $P(x, y)$ are the constraints that must satisfied for this martingale x, across every program state y.

Previous approaches to generating martingales, use techniques such as Farkas' lemma and Positivstellensatz to translate the problem into a quantifier-free formulation, that can be handled by linear and quadratic programming [Chakarov and Sankaranarayanan, 2013; Chatterjee et al., 2016a]. Therefore, the methods are applicable for only a narrow class of programs, and exclude programs such as the faulty marble collector.

In Abate et al. [2021c] a new method to synthesise ranking supermartingales (RSMs) is introduced, in order to prove positive almost-sure termination (PAST). This method constructs a RSM by using a neural network to learn a candidate martingale. The network is trained with sampled execution traces from the program. This candidate martingale is verified by using satisfiability modulo theories (SMT). If a counterexample is produced, this is added to the training data, so the neural network can learn a new candidate. This continues until a valid martingale has been constructed. This process of incremental learning is called counterexample-guided inductive synthesis (CEGIS).

The goal of this project is to extend this idea to other types of martingales, to solve a *quantitative safety problem*. In a quantitative safety problem, the objective is to bound the probability that a program will reach an unsafe region of the state space. Concretely, a solution to such a problem is a probability p, and an accompanying certificate that demonstrates that the program will enter the unsafe region with probability at most p. In the faulty marble collector, the unsafe region of the state space is where *error* is 1.

In this project, novel algorithms will be introduced, that use neural networks to synthesise two types of martingales:

1. Indicating supermartingales (ISMs), which produce bounds with the Knaster-Tarski fixed-point theorem

2. Repulsing supermartingales (RepSMs), which produce bounds with the Azuma-Hoeffding inequality

These algorithms will be implemented in a tool for program verification. The tool includes functionality for parsing source code, performing symbolic inference, and carrying out the CEGIS procedure.

While neural networks have been applied to RSMs, applying them to ISMs and RepSMs poses multiple challenges. First, while any valid RSM is sufficient to prove PAST, a valid ISM or RepSM is not necessarily useful, unless it can be used to produce a tight bound. Further, quantitative safety problems are typically applied to rare events. However, this means that the regions of the state space that are vital to the construction of a martingale, are rarely seen in execution traces.

Therefore, there is clear motivation in applying machine learning to the quantitative safety problem. Quantitative safety is a fundamental verification problem for

probabilistic programs. However, existing tools for this problem are very restrictive in terms of the kind of programs they can handle. By contrast, neural networks are universal approximators, and can be used to construct a wide range of non-linear functions. This suggests that there is significant potential in using neural networks to construct martingales for programs that exhibit more complex behaviour.

1.2 Contributions

In this project, we make the following contributions:

- We develop the first machine learning method for the verification of the quantitative safety problem for probabilistic programs, by synthesising indicating supermartingales (ISMs) and repulsing supermartingales (RepSMs) represented by neural networks. The method handles continuous state spaces, and proves quantitative properties, both of which are out-of-scope for the most recent pre-existing work that applies learning to the verification of probabilistic systems [Bao et al., 2022]. Further, this is the first method for synthesising martingales that uses a general symbolic inference algorithm to compute post-expectations.
- We experimentally evaluate this method, demonstrating that it surpasses the state-of-the-art. Our benchmarks can be classified in a trichotomy. First, benchmarks where both methods produce comparable results. Secondly, benchmarks where this method produces significantly better results. Thirdly, benchmarks where only this method can produce results.

Chapter 2

Background

2.1 Measure Theory

This section formalises probabilistic programs, so that executions of a probabilistic program can be treated as a measurable space. The goal is to establish several technical results that lay the groundwork for subsequent sections. Of these results, the most important demonstrates that the quantitative safety properties, this project studies, are well-defined. References for the measure-theoretic treatment in this section are Rosenthal [2006]; Meyn and Tweedie [2009].

In order to study probabilistic programs, it is necessary to conceptualise them as formal mathematical objects. Two formalisms used in the literature are probabilistic transition systems (PTSs) and probabilistic control-flow graphs (pCFGs). Both systems model the probabilistic program as a graph, with program locations as nodes, and transitions as edges. The main difference is that pCFGs allow for nondeterministic behaviour, while PTSs do not.

By contrast, Abate et al. [2021c] focusses on single-loop programs. These programs consist of a loop guard G and a probabilistic update statement U. The probabilistic program runs the update statement while the loop guard holds. These programs have the following syntactic structure.

while G:
    U

This leads to a simpler formalism, that is easier to reason about. This project also concentrates on single-loop programs, except for the addition of a safety predicate P. In this model, after the loop guard G is checked, the safety predicate P is checked. If it holds, then the update statement U runs as normal. Otherwise, an assertion is considered to be violated, and the program fails.

Thus, there are two different ways for the program to end. The first is successful termination where the loop guard does not hold. The second is when an assertion is violated, and the program has failed. The syntax for these programs has the following structure.

while G:
    $assert(P)$
    U

More concretely, a program contains a finite set of variables Vars, and the program state is modelled by a function $x : \text{Vars} \to \mathbb{R}$. The collection of all such states is the state space S. Then the guard G and the safety predicate P are Borel subsets of S. It is assumed there is a fixed unique initial state $x_0 \in S$, although it is straightforward to generalise this to a non-deterministic set of initial states. Note that the state space can be expressed as the following disjoint union:

$$S = (S \setminus G) \cup (G \setminus P) \cup (G \cap P)$$

The components of this trichotomy can be described as follows.

- $S \setminus G$ consists of states, where the program has terminated successfully.
- $G \setminus P$ consists of states, where the program has failed.
- $G \cap P$ consists of states, where the program has not ended.

The final component of a probabilistic program is an update statement U, that encodes the evolution of a probabilistic program. This is a function $U : (G \cap P) \times \mathcal{B}(S) \to [0, 1]$. This function is required to be a *Markov kernel*, in other words:

- For each $x \in G \cap P$, $U(x, \cdot)$ is a probability measure.
- For each $E \in \mathcal{B}(S)$, $U(\cdot, E)$ is a measurable function.

Intuitively, $U(x, E)$ is the probability of starting in state $x \in S$, and ending in a state within the set E after a single application of the update statement U. Putting this altogether, a single-loop program can be encoded as a quintuple $(\text{Vars}, G, P, U, x_0)$.

This is an expressive formalism; single-loop programs are naturally able to encode discrete-time systems, with infinite state spaces. Such systems can mix discrete and continuous variables.

Having formalised probabilistic programs, the next step is to formalise the execution of such a program. An execution is defined as an infinite sequence of program states, i.e. an element of S^ω, each taken at the beginning of a loop iteration. In order to account for programs that terminate or fail, it is assumed that they remain in the same state, that they ended in, for future elements of the sequence.

To handle this technicality, the Markov kernel U is extended to have the domain S:

$$U^\star(x, E) = \begin{cases} U(x, E) & \text{if } x \in G \cap P \\ 1 & \text{if } x \notin G \cap P \text{ and } x \in E \\ 0 & \text{otherwise} \end{cases}$$

In other words, $U^\star(x, \cdot)$ is $U(x, \cdot)$ if $x \in G \cap P$, otherwise it is a Dirac measure, where $U^\star(x, E) = 1$ iff $x \in E$. Note that, this extension is still a Markov kernel.

The pair $(S^\omega, \mathcal{B}(S^\omega))$ forms a measurable space. It is desirable to build a probability measure on this space with the Markov kernel U, to reason about the probability of events. This is achieved with the following theorem.

Theorem 1 (Kolmogorov's extension theorem)**.** *Let $S = \mathbb{R}^k$ for some fixed k. Then suppose for each $n \in \mathbb{N}$, there is a probability measure μ_n, for the measurable space $(S^n, \mathcal{B}(S^n))$.*

Assume that this family of probability measures is consistent i.e. $\mu_{n+1}(A \times S) = \mu_n(A)$ for all $A \in \mathcal{B}(S^n)$ and $n \in \mathbb{N}$.

Then there is a unique probability measure μ for $(S^\omega, \mathcal{B}(S^\omega))$ such that for every $n \in \mathbb{N}$: $\mu(A \times S^\omega) = \mu_n(A)$.

In other words, by defining a probability measure for finite prefixes of an execution, a probability measure is obtained for executions. The family of probability measures μ_n is defined by induction.

The base case μ_0 is trivial.

$$\mu_0(\_) = 1$$

The base case μ_1 is a Dirac measure.

$$\mu_1(E_0) = \begin{cases} 1 & x_0 \in E_0 \\ 0 & \text{otherwise} \end{cases}$$

Then given a probability measure μ_n, this can be used to define a probability measure μ_{n+1}, by integrating with respect to the measure.

$$\mu_{n+1}(E_0, \ldots, E_n, E_{n+1}) = \int_{E_0 \times \ldots \times E_n} U^\star(x_n, E_{n+1})\ \mu_n(\mathrm{d}(x_0, \ldots, x_n))$$

Having constructed this family of probability measures, it can be established that a unique probability measure $\mu : \mathcal{B}(S^\omega) \to [0, 1]$ exists.

The measurable space $(S^\omega, \mathcal{B}(S^\omega))$ accompanied with the measure μ makes it possible to deliver the following technical results.

First, it is now possible to consider the probabilistic program as a stochastic process, i.e. a family of random variables $\{X_i\}$. Each random variable X_i is a projection of S^ω, and is defined as follows:

$$X_i : S^\omega \to S$$
$$X_i(x_0, x_1, \ldots) = x_i$$

In other words, each random variable X_i denotes the state at the ith iteration of the while loop.

Moreover, it is now possible to ensure certain reachability events and probabilities are well-defined.

Proposition 1. *Consider the set of executions that reach a Borel measurable set E in exactly N steps, i.e. $X_N \in E$, and for all $n < N$, $X_n \notin E$. This event is measurable.*

Corollary 1. *The following probabilities are well-defined:*

- *The probability of an execution failing.*
- *The probability of an execution terminating successfully.*

This ensures that the quantitative safety properties, this project studies, are well-defined. However, in order to reason about the soundness of RepSMs and ISMs, it is useful to consider the probability of more granular events.

Note that, the stochastic process $\{X_i\}$ satisfies the strong Markov property. This means that $\mathbb{P}[X_{i+t} \in E \mid X_i = x]$ is the same as $\mathbb{P}[X_t \in E \mid X_0 = x]$. This means that the probability of reaching E in N steps from state x is well-defined, since the past history of the execution does not matter.

Corollary 2. *The following probabilities are well-defined.*

- *The probability of an execution failing in exactly N steps from state x.*
- *The probability of an execution failing in at most N steps from state x.*
- *The probability of an execution failing from state x.*

This follows from the properties of a σ-algebra, the strong Markov property and the previous proposition.

Additionally, it is now possible to define the post-expectation operator $\mathbb{X}$. Consider the program state of the loop at the ith iteration, X_i. Assume there is a function $\eta : S \to \mathbb{R}$. Given that $X_i = x$, it is straightforward to compute $\eta(X_i)$. In the analysis of probabilistic programs, it is also useful to know $\mathbb{E}[\eta(X_{i+1})]$ given that $X_i = x$, i.e. the expectation of η at the $(i+1)$th iteration given X_i.

This is what the post-expectation operator accomplishes. The operator takes a function $\eta : S \to \mathbb{R}$ and produces a new function $\mathbb{X}\eta : S \to \mathbb{R}$. This new function takes a state x and gives the value $\mathbb{E}[\eta(X_{i+1}) \mid X_i = x]$. The operator is defined as follows:

$$\mathbb{X}\eta(x) = \int_S \eta(x')\ U^\star(x, \mathrm{d}x')$$

Naturally, it is assumed that η is a measurable function.

To conclude, this section develops a measurable space for executions and an intended probability measure. This allows the following to be established: the measurability of certain reachability events; the interpretation of probabilistic programs as stochastic processes; and the post-expectation operator. These results will be built upon by subsequent sections.

2.2 Program Analysis with Martingales

This section introduces martingales and their applications to program verification. First, RSMs are introduced, which can be used to prove PAST. Then, this section introduces ISMs and RepSMs, which this project uses to prove quantitative safety properties.

Throughout this report, the term martingale is used as a shorthand for super-martingale. Furthermore, the term martingale is 'overloaded'. Its primary definition is in terms of a sequence of scalar random variables. However, it is also used to refer to functions that map a multivariate stochastic process, representing an execution

of a program, to a scalar stochastic process. A scalar stochastic process $\{M_t\}$ is a supermartingale if for every $t \geq 0$, the following holds:

$$\mathbb{E}[M_{t+1} \mid M_t = m_t, \ldots, M_0 = m_0] \leq m_t$$

Before introducing different types of martingales, recall that the state space S can be split into the following trichotomy.

$$(S \setminus G) \cup (G \setminus P) \cup (G \cap P)$$

It is sometimes the case that not all states in G can actually be reached. It is helpful to define martingales in terms of an invariant $I \subseteq G$ which is an over-approximation of all reachable states.

One can then consider the following disjoint union:

$$(S \setminus G) \cup (I \setminus P) \cup (I \cap P)$$

Note that, $S \setminus G$ is the set of terminal states, $I \setminus P$ is the set of unsafe states, and $I \cap P$ is the set of safe states.

2.2.1 RSMs

The first type of martingale introduced for the analysis of probabilistic programs was ranking supermartingales (RSMs) [Chakarov and Sankaranarayanan, 2013]. An RSM is a map $\eta : S \to \mathbb{R}$ such that the following holds for some $\epsilon > 0$:

- Decreasing condition: $\mathbb{X}\eta(x) \leq \eta(x) - \epsilon$ for all $x \in I$
- Lower bound condition: $\eta(x) \geq K$ for all $x \in I$.

RSMs can be used to prove positive almost-sure termination. For RSMs, it will be assumed that the safety predicate $P = S$, i.e. all states are safe.

Then each execution is either terminating or non-terminating, and so can be given a termination time which is an element of $\mathbb{N} \cup \{\infty\}$. This is defined as the random variable T. The property almost-sure termination (AST) is that $\mathbb{P}[T < \infty] = 1$. The property positive almost-sure termination (PAST) is that $\mathbb{E}[T] < \infty$. Note that PAST strictly subsumes AST.

Proposition 2 (Soundness of RSMs)**.** *An RSM for a probabilistic program is a witness for PAST, i.e. if an RSM exists this means* $\mathbb{E}[T] < \infty$.

The intuition is that non-terminal states are required to satisfy $\eta(x) \geq K$, and since the RSM decreases by at least ϵ in expectation, eventually a terminal state must be reached. A proof is given in Ferrer Fioriti and Hermanns [2015].

2.2.2 ISMs

Indicating supermartingales (ISMs) have been introduced in previous literature under the names *nonnegative repulsing supermartingale* and *stochastic invariant indicators* [Takisaka et al., 2021; Chatterjee et al., 2022]. The reason for choosing the name *indicating supermartingale* is to avoid confusion with RepSMs, emphasise the connection with indicator functions, and acknowledge that ISMs are indeed supermartingales.

ISMs are used to bound the probability of leaving a safe region P of the state space. An ISM is a Borel measurable function[1] $\eta : I \to [0, \infty)$ such that the following hold:

- Non-increasing condition: $\eta(x) \geq \mathbb{X}\eta(x)$ for all $x \in I \cap P$
- Indicating condition: $\eta(x) \geq 1$, for all $x \in I \setminus P$

Let $B(I)$ denote the set of Borel measurable functions $I \to [0, \infty]$. Note that, a subset of $B(I)$ is the set of ISMs. The set $B(I)$ can be given a partial order $\sqsubseteq$. $\eta_1 \sqsubseteq \eta_2$ if $\eta_1(x) \leq \eta_2(x)$ for all $x \in I$. This partial order is, in fact, a lattice, as it can be given join ($\vee$) and meet ($\wedge$) operations.

$$(\eta_1 \vee \eta_2)(x) = \max\{\eta_1(x), \eta_2(x)\}$$
$$(\eta_1 \wedge \eta_2)(x) = \min\{\eta_1(x), \eta_2(x)\}$$

The lattice also has a bottom element ($\bot$) defined as follows:

$$\bot(x) = 0$$

Further, recall that a (higher-order) function $\Psi : B(I) \to B(I)$ is monotone if $\eta_1 \sqsubseteq \eta_2$ implies $\Psi(\eta_1) \sqsubseteq \Psi(\eta_2)$. One can define the monotone function Φ:

$$\Phi(\eta)(x) = \begin{cases} 1 & x \in I \setminus P \\ \mathbb{X}\eta(x) & \text{otherwise} \end{cases}$$

This can be made clearer by expressing it in the following form:

$$\Phi(\eta) = \chi_{I \setminus P} + \chi_{I \cap P} \mathbb{X}\eta$$

$\chi_A : B \to \{0, 1\}$ is an indicator function for a set $A \subseteq B$. Its definition is given below.

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & \text{otherwise} \end{cases}$$

Note that a pre-fixpoint of a function is x such that $f(x) \sqsubseteq x$. Then an ISM can be defined as a pre-fixpoint of the function Φ. This explains the connection between indicator functions and ISMs.

[1] Previous literature uses upper semianalytic functions, because they incorporate nondeterminism in their model of probabilistic programs.

There are two properties of ISMs that are desirable to prove: soundness and completeness. These properties relate to the function $\mathrm{Reach}(x)$, which is the probability of reaching $I \setminus P$ from the state x. In order to prove these properties, it is necessary to build up a collection of lemmas. These results relate to the function $\mathrm{Reach}^{\leq N}$, which is the probability of reaching $I \setminus P$ in at most N steps.

Lemma 1. *For every* $N \in \mathbb{N}$, $\mathrm{Reach}^{\leq N} \in B(I)$ *and* $\mathrm{Reach}^{\leq N} = \Phi^{N+1}(\bot)$.

Proof. This can be shown by induction. Consider the base case. The left-hand side is $\mathrm{Reach}^{\leq 0} = \chi_{I \setminus P}$. The right-hand side is:

$$\Phi^1(\bot) = \Phi(\bot) = \chi_{I \setminus P} + \chi_{I \cap P} \bot = \chi_{I \setminus P}$$

This follows since $\bot(x) = 0$ for all x. Now consider the inductive case. The left-hand side is:

$$\begin{aligned} \mathrm{Reach}^{\leq N+1}(x) &= \begin{cases} 1 & x \in I \setminus P \\ \int_I \mathrm{Reach}^{\leq N}(x') U^\star(x, \mathrm{d}x') & \text{otherwise} \end{cases} \\ &= \begin{cases} 1 & x \in I \setminus P \\ \mathbb{E}[\mathrm{Reach}^{\leq N}(X_{i+1}) \mid X_i = x] & \text{otherwise} \end{cases} \end{aligned}$$

So:

$$\begin{aligned} \mathrm{Reach}^{\leq N+1} &= \chi_{I \setminus P} + \chi_{I \cap P} \mathbb{X}(\mathrm{Reach}^{\leq N}) \\ &= \Phi(\mathrm{Reach}^{\leq N}) \\ &= \Phi(\Phi^{N+1}(\bot)) \\ &= \Phi^{N+2}(\bot) \end{aligned}$$

The penultimate step follows from the application of the inductive hypothesis. □

Theorem 2 (Variant of Kleene fixpoint theorem)**.** *If* $(X, \sqsubseteq)$ *is a chain-complete partially ordered set with a bottom element, and* $f : X \to X$ *is a monotone map, then* $\sqcup_{i \in \mathbb{N}} f^i(\bot)$ *is a least pre-fixpoint.*

Corollary 3. *If* $(X, \sqsubseteq)$ *is a chain-complete partially ordered set with a bottom element, and* $f : X \to X$ *is a monotone map, then* $\sqcup_{i \in \mathbb{N}} f^i(\bot)$ *is a least fixpoint.*

Proof. First note that $\sqcup_{i \in \mathbb{N}} f^i(\bot)$ is indeed a fixpoint:

$$f(\bigsqcup_{i \in \mathbb{N}} f^i(\bot)) = \bigsqcup_{i \in \mathbb{N}} f^{i+1}(\bot) = \bigsqcup_{i \in \mathbb{N}} f^i(\bot)$$

Suppose there is a different least fixpoint, then this would also be the least pre-fixpoint, which is a contradiction. □

Lemma 2. *The function* Reach *is the least pre-fixpoint of the map* Φ*. Further it is a least fixpoint, i.e.* $\mathrm{Reach} = \mu\Phi$.

Proof. Note that:

$$\text{Reach}(x) = \sup_{i\in\mathbb{N}} \text{Reach}^{\leq i}(x) = \sup_{i\in\mathbb{N}} \Phi^{i+1}(\bot) = \sup_{i\in\mathbb{N}} \Phi^{i}(\bot)$$

In other words:

$$\text{Reach} = \sqcup_{i\in\mathbb{N}} \Phi^{i}(\bot)$$

Theorem 2 establishes that this is indeed the least pre-fixpoint of Φ, and consequently by Corollary 3 it is the least fixpoint. Thus Reach $= \mu\Phi$. □

This makes it possible to prove completeness and soundness.

Corollary 4 (Completeness of ISMs)**.** Reach *is an ISM i.e.* $\Phi(\text{Reach}) \sqsubseteq \text{Reach}$.

Proof. This follows directly from Lemma 2. □

Corollary 5 (Soundness of ISMs)**.** *If* η *is an ISM, it overapproximates reachability probabilities i.e.* $\Phi(\eta) \sqsubseteq \eta$ *implies* $\text{Reach} \sqsubseteq \eta$.

Proof. Suppose η is a pre-fixpoint i.e. $\Phi(\eta) \sqsubseteq \eta$. Since Reach is a least pre-fixpoint by Lemma 2, then $\text{Reach} \sqsubseteq \eta$. □

These results shows us that an ISM can be used to upper bound the probability of reaching the unsafe region as $\eta(x_0)$. So, the goal is to find ISMs that are closer to the optimal ISM, i.e. $\mu\Phi$.

2.2.3 RepSMs

Repulsing supermartingales (RepSMs) were introduced in Chatterjee et al. [2016b]. A RepSM is a map $\eta : S \to \mathbb{R}$ such that the following holds for some $\epsilon > 0$ and $c \geq 0$:

- Decreasing condition: $\mathbb{X}\eta(x) \leq \eta(x) - \epsilon$ for all $x \in I \cap P$
- Lower bound condition: $\eta(x) \geq 0$ for all $x \in I \setminus P$
- Bounded differences condition: $|\eta(X_{t+1}) - \eta(X_t)| \leq c$ for some constant $c > 0$
- Initial state condition: $\eta(x_0) < 0$ (It follows that $x_0 \in P$)

Similarly to RSMs, the random variable T can be defined as the time until an unsafe state in $I \setminus P$ is reached, and this random variable takes a value from $\mathbb{N} \cup \{\infty\}$. The goal is to use RepSMs to upper bound $\mathbb{P}(T < \infty)$.

Theorem 3 (Azuma-Hoeffding inequality)**.** *If* $X_0, \ldots, X_n$ *is a supermartingale such that* $|X_t - X_{t-1}| \leq c_t$ *for all* $t > 1$*, then the following holds for any* $\epsilon > 0$ *and* $t \geq 1$*:*

$$\mathbb{P}(X_t - X_0 \geq \epsilon) \leq \exp\left(\frac{-\epsilon^2}{2\sum_{k=1}^{t} c_k^2}\right)$$

Note that if $c_t = c$, for all t, then, this gives the following bound:

$$\mathbb{P}(X_t - X_0 \geq \epsilon) \leq \exp\left(\frac{-\epsilon^2}{2tc^2}\right)$$

Proposition 3 (Soundness of RepSMs)**.** *The probability of reaching an unsafe state can be bounded as follows.*

$$\mathbb{P}(T < \infty) \leq \alpha \frac{\gamma^A}{1-\gamma}$$

Where:

$$\alpha = \exp\left(\frac{\epsilon m_0}{(c+\epsilon)^2}\right)$$
$$\gamma = \exp\left(-\frac{\epsilon^2}{2(c+\epsilon)^2}\right)$$
$$A = \lceil |m_0|/c \rceil$$

Proof. In order to obtain this bound from a RepSM $\{X_t\}$, an auxiliary supermartingale $\{X'_t\}$ is constructed from a run ρ, as follows:

$$X'_t(\rho) = \begin{cases} X_t(\rho) + t\epsilon & \text{if } T(\rho) \geq t \\ X'_{t-1}(\rho) & \text{otherwise} \end{cases}$$

In order to understand this supermartingale, note that $X'_0(\rho) = X_0(\rho)$, since $x_0 \notin I \setminus P$. However, at each transition the difference between X_t and X'_t grows by ϵ, until the unsafe region is entered, at which point the martingale becomes fixed.

Note that this is indeed a supermartingale. When $T(\rho) \geq t+1$, the following hold:

$$\begin{aligned} \mathbb{E}[X'_t(\rho)] &= \mathbb{E}[X_t(\rho)] + t\epsilon \\ \mathbb{E}[X'_{t+1}] &= \mathbb{E}[X_{t+1}(\rho) + (t+1)\epsilon] \\ &\leq \mathbb{E}[X_t(\rho)] - \epsilon + (t+1)\epsilon \\ &= \mathbb{E}[X_t(\rho)] + t\epsilon \\ &= \mathbb{E}[X'_t(\rho)] \end{aligned}$$

When $T(\rho) < t+1$, then trivially $\mathbb{E}[X'_{t+1}] \geq \mathbb{E}[X'_t]$. Note, in the case that the unsafe region is never entered, then this martingale is still well-defined, X'_t and X_t will simply diverge without end.

It can be similarly shown that the martingale has $(c+\epsilon)$-bounded differences. Now define the family of sets F_t as follows. An execution ρ is a member of F_t if $T(\rho) = t$. Note that if $\rho \in F_t$, this implies:

$$\begin{aligned} X'_t(\rho) &= X_t(\rho) + t\epsilon \\ &\geq t\epsilon \end{aligned}$$

This holds, since $X_t(\rho) \geq 0$, since ρ is within $I \setminus P$ at time step t.

$$\begin{aligned} \mathbb{P}(F_t) &\leq \mathbb{P}(X'_t \geq t\epsilon) \\ &= \mathbb{P}(X'_t - X'_0 \geq t\epsilon - \eta(x_0)) \end{aligned}$$

Defining $m_0 = \eta(x_0)$, and applying the Azuma-Hoeffding inequality gives us:

$$\begin{aligned}\mathbb{P}(F_t) &\leq \mathbb{P}(X'_t - X'_0 \geq t\epsilon - m_0) \\ &\leq \exp\left(-\frac{(t\epsilon - m_0)^2}{2t(c+\epsilon)^2}\right) \\ &\leq \exp\left(\frac{\epsilon m_0}{(c+\epsilon)^2}\right)\exp\left(-\frac{t\epsilon^2}{2(c+\epsilon)^2}\right)\end{aligned}$$

By summing $\mathbb{P}(F_t)$ for all $t < \infty$, the probability that the unsafe region $I \setminus P$ can be expressed as follows:

$$\mathbb{P}(T < \infty) = \sum_{t=0}^{\infty} \mathbb{P}(F_t)$$

This bound can be tightened by observing that at least A steps are need to reach $I \setminus P$, where $A = \lceil |m_0|/c \rceil$. This gives us:

$$\begin{aligned}\mathbb{P}(T < \infty) &= \sum_{t=A}^{\infty} \mathbb{P}(F_t) \\ &\leq \sum_{t=A}^{\infty} \exp\left(\frac{\epsilon m_0}{(c+\epsilon)^2}\right)\exp\left(-\frac{t\epsilon^2}{2(c+\epsilon)^2}\right) \\ &= \sum_{t=A}^{\infty} \exp\left(\frac{\epsilon m_0}{(c+\epsilon)^2}\right)\exp\left(-\frac{\epsilon^2}{2(c+\epsilon)^2}\right)^t\end{aligned}$$

Define α and γ as follows:

$$\begin{aligned}\alpha &= \exp\left(\frac{\epsilon m_0}{(c+\epsilon)^2}\right) \\ \gamma &= \exp\left(-\frac{\epsilon^2}{2(c+\epsilon)^2}\right)\end{aligned}$$

Then:

$$\mathbb{P}(T < \infty) \leq \sum_{t=A}^{\infty} \alpha\gamma^t$$

Rewriting this geometric series gives us the following bound on reaching the unsafe region $I \setminus P$:

$$\mathbb{P}(T < \infty) \leq \alpha\frac{\gamma^A}{1-\gamma}$$

□

To conclude, this section has introduced three types of martingales: RSMs, ISMs, and RepSMs. Furthermore, proofs of soundness have been given for ISMs and RepSMs are valid. This means that the bounds generated by them are sound.

2.3 Synthesis of Martingales

This section recalls previous literature on the synthesis of martingales. In particular, it presents Farkas' lemma which has been used to synthesise RSMs, ISMs, and RepSMs. Further, it presents the CEGIS-based synthesis of RSMs. This work is built upon in subsequent chapters to develop a CEGIS-based method for RepSMs and ISMs, that allows us to solve the quantitative safety problem.

A martingale η must satisfy properties of the form $\forall x.P(\eta, x)$. In other words, the overall problem has the structure $\exists\eta.\forall x.P(\eta, x)$.

One challenge of constructing martingales is the presence of universal quantification, since in general, the properties $P(\eta, x)$ need to hold for an infinite number of possible values of x. Another challenge is in choosing the structure of the martingale function η. The standard approach is to fix a parametric template for this function, shifting the problem from finding an arbitrary function to finding parameters for the template.

2.3.1 Linear RSMs

One common template is linear (technically affine) expressions over the program state. Linear RSMs can be computed with linear programming and Farkas' lemma [Chakarov and Sankaranarayanan, 2013]. Consider the program below. The program has one variable x, which is initialised to 1. In each iteration x is either incremented or decremented. The program is biased towards incrementing, which happens with probability 0.9. The loop continues until x reaches 100.

$x = 1$
while $x \leq 100$:
    $p \sim \mathsf{Bernoulli}(0.9)$
    if $p == 1$:
        $x = x + 1$
    else:
        $x = x - 1$

A linear template for this function is $wx+b$. It is necessary to compute the post-expectation $E[\eta(X_{i+1}) \mid X_i]$. This requires performing symbolic inference on the loop-free update statement, and marginalising the probabilistic choices. In general, this is a non-trivial problem. For this program, marginalising is straightforward, and the post-expectation of the template is $wx + \frac{4w}{5} + d$. The two conditions that need to be satisfied are:

$$\forall x \in I, wx + \frac{4w}{5} + d \leq wx + d - \epsilon \tag{2.1}$$

$$\forall x \in I, wx + d \geq K \tag{2.2}$$

In this case, K is chosen to be 0. Further I is the set $\{x \mid x \leq 100\}$. The constraints can be rewritten as follows:

$$\forall x \in \mathbb{R}, (x \leq 100) \implies \frac{4w}{5} + \epsilon \leq 0 \tag{2.3}$$

$$\forall x \in \mathbb{R}, (x \leq 100) \implies -wx - d \leq 0 \tag{2.4}$$

Theorem 4 (Farkas' Lemma)**.** *For $A \in \mathbb{R}^{m,n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ and $d \in \mathbb{R}$, consider the following statement.*

$$\forall x \in \mathbb{R}^n.\ Ax \leq b \implies c \cdot x \leq d$$

This statement is equivalent to the following holding:

- *There exists $x \in \mathbb{R}^n$ such that $Ax \leq b$.*
- *There exists $y \in \mathbb{R}^m$ such that $y \geq 0$ and $A^T y = c$ and $b \cdot y \leq d$.*

The constraints are now in a form where Farkas' lemma can be applied. Farkas' lemma is applied to both 2.3 and 2.4, to produce two new quantifier-free constraints:

$$(\exists x.x \leq 100) \wedge (\exists y.w + y = 0 \wedge 100y - d \leq 0 \wedge y \geq 0)$$
$$(\exists x.x \leq 100) \wedge (\exists y.y = 0 \wedge \epsilon + \frac{4w}{5} + 100y \leq 0 \wedge y \geq 0)$$

The constraints can be solved with linear programming. This produces the parameters $d = 100$ and $w = -1$, giving the RSM: $100 - x$. Consequently, it is shown that this program is PAST i.e. it will terminate almost-surely, and the expected termination time is finite.

2.3.2 Linear ISMs

The synthesis of linear ISMs has been considered with Farkas' lemma [Chatterjee et al., 2022]. Synthesising ISMs is different from RSMs. The synthesis of an RSM can be considered a decision problem, since there are two outcomes, either an RSM was found or the algorithm failed to find one.

On the other hand, synthesising a valid ISM is trivial, since one can use the function that gives 1 as a bound at every state. Instead, the goal is to find a valid ISM with a tight bound, so the problem can be seen as an optimisation problem. In contrast to traditional optimisation problems, the constraints are of the form $\forall x.P(\eta, x)$.

Linear ISMs can be constructed in a similar manner to RSMs. The approach is to use the Farkas' lemma to perform quantifier-elimination, and then perform linear programming with the value of the ISM at the initial state as the objective function. This is illustrated with the following program. The variable t is initialised to 1, and is incremented with probability $1/2$ at each iteration until $t = 10$ is reached. In each iteration, there is a very small failure probability and the goal is to find a tight ISM which bounds this failure probability.

$t = 1$
$error = 0$
while $t < 10$:
    $assert(error \neq 1)$
    $p \sim \mathsf{Bernoulli}(0.999)$
    if $p == 1$:
        $q \sim \mathsf{Bernoulli}(0.5)$
        if $q == 1$:
            $t = t + 1$
    else:
        $error = 1$

Note that t is an integer variable, and $error$ has a value of either 0 or 1. A linear template for this program is $w_{error} \cdot error + w_t \cdot t + d$. The post-expectation for this template is:

$$\frac{999 w_{error} \cdot error}{1000} + w_t \cdot t + \frac{w_{error}}{1000} + \frac{999 w_t}{2000} + d$$

To be a valid ISM, the non-increasing and indicating condition need to be satisfied:

$$\begin{aligned} &\forall error, t \in \mathbb{R}.\ (t < 10 \land error = 0) \implies w_{error} \cdot error + w_t \cdot t_0 + d \\ &\geq \frac{999 w_{error} \cdot error}{1000} + w_t \cdot t + \frac{w_{error}}{1000} + \frac{999 w_t}{2000} + d \end{aligned} \tag{2.5}$$

$$\forall error, t \in \mathbb{R}.\ (t < 10 \land error = 1) \implies w_{error} \cdot error + d + w_t \cdot t_0 \geq 1 \tag{2.6}$$

Additionally ISMs are required to have $[0, \infty]$ as the codomain, i.e. they must be nonnegative. To ensure this, the following condition is added:

$$\forall error, t \in \mathbb{R}.\ (t < 10) \implies w_{error} \cdot error + d + w_t \cdot t_0 \geq 0 \tag{2.7}$$

The initial state is $t = 1 \land error = 0$, and the ISM at this state is $w_t \cdot t + d$. Linear programming is used to minimise this expression subject to the quantifier-free formulae, obtained by applying Farkas' lemma to the constraints above. This can be used to obtain the bound ≤ 0.0181 for the probability of failure.

2.3.3 Linear RepSMs

The synthesis of linear RepSMs has also been considered with Farkas' lemma [Chatterjee et al., 2016b]. This follows a similar approach to the synthesis of linear ISMs, but with a complication. Unlike ISMs, even if a RepSM uses a linear template, the expression for the bound is not a linear expression. This means that it cannot be the objective function of a linear program.

Recall that the bound depends on ϵ, c and $\eta(x_0)$. In previous literature, the solution is to run linear programming multiple times, and to select the best bound obtained. More concretely, the method proposed is as follows:

1. Set $\epsilon = 1$, since linear RepSMs can be scaled arbitrarily.

2. Find the smallest c for which a RepSM is possible with linear programming. This is denoted as $c_{\min}$.

3. For $0 \leq j \leq N$, linear programming is used to find a RepSM that minimises $\eta(x_0)$ and has $c + j$-bounded differences.

4. Choose the value of j which gave the smallest bound.

Note that N is a fixed constant, with previous literature using $N = 1000$.

2.3.4 Neural RSMs

A major limitation of Farkas' lemma is that it can only learn linear templates. This is a significant restriction, and even if a program contains only linear inequalities and linear expressions, a linear template may be insufficient.

An extension of Farkas' lemma to polynomials is *Positivstellensatz* [Chatterjee et al., 2016a]. This allows polynomial templates to be used for RSMs, and quantifier-elimination is performed leading to constraints that can be solved by semidefinite programming. Nevertheless, polynomial templates are still restrictive, and often more general non-linear templates are needed to verify simple programs.

For instance, consider the marble collector. This is a simpler version of the faulty marble collector, presented earlier. In each iteration, either a red marble or a blue marble is collected. The variables *red* and *blue* record the number of red and blue marbles respectively that are left to collect.

while $red > 0$ **or** $blue > 0$:
    $p \sim \mathsf{Bernoulli}(0.5)$
    if $p == 1$:
        $red = red - 1$
    else:
        $blue = blue - 1$

A martingale constructed from a linear template is unable to act as a certificate of termination for this program. This means that Farkas' lemma is unable to synthesise an RSM. In Abate et al. [2021c], an alternative approach to synthesising RSMs is introduced, using neural networks. This method can handle more general probabilistic programs. For instance, this method can synthesise the following neural RSM for the program above:

$$\max(red, 0) + \max(blue, 0)$$

The method uses two kinds of neural parametric templates for probabilistic programs. The following sum of ReLU (SOR) template is used for discrete programs:

$$\eta(x) = \sum_{i=1}^{h} \mathrm{ReLU}\left(\sum_{j=1}^{|\mathrm{Vars}|} W_{i,j} x_j + b_i\right)$$

Then, the following sum of squares (SOS) template is used for continuous programs:

$$\eta(x) = \sum_{i=1}^{h} \left(\sum_{j=1}^{|\text{Vars}|} W_{i,j} x_j + b_i \right)^2$$

The difference between the two templates is the activation function. The ReLU activation function is a non-smooth function defined as follows:

$$\text{ReLU}(x) = \max(0, x)$$

Recall, that an RSM has to satisfy two constraints.

- Decreasing condition: $\mathbb{X}\eta(x) \leq \eta(x) - \epsilon$ for all $x \in I$
- Lower bound condition: $\eta(x) \geq K$ for all $x \in I$.

The lower bound condition is trivially satisfied since the templates produce non-negative functions. So, the learning process only has to focus on the decreasing constraint. This occurs in a CEGIS loop. First, the initial training data is generated from program states found by executing the probabilistic program.

Then, parameters are learnt by minimising a loss function that penalises violations of this constraint for states in the training data. Candidate martingales are produced by rounding the parameters to various precisions. The candidate martingales are then verified with an SMT solver.

If none of these are valid martingales, then counterexamples are generated and added to the training set. Then the learning process is repeated, with the enlarged training data. This process continues until a valid martingale is learnt, or a maximum number of CEGIS iterations has been reached.

The loss function is minimised with gradient descent, with backpropagation being used to compute gradients. The loss function for each program state p in the training data is as follows:

$$L(p, P') = \text{softplus}(\mathbb{E}_{p \sim P'}[\eta(p')] - \eta(p) + \epsilon)$$

Note that in this expression p' stands for the successor state of p. Rather than using a symbolic expression to compute the post-expectation $\mathbb{X}\eta(x)$, this value is instead sampled. For each program state p in the training data, there is an associated set of successor states P', which is used to estimate this value.

The loss function is designed to apply a penalty when $\mathbb{X}\eta(x) > \eta(x) - \epsilon$. In particular, the greater $\mathbb{X}\eta(x)$ is compared to $\eta(x) - \epsilon$ the higher the penalty should be. Further, if $\mathbb{X}\eta(x)$ is less than $\eta(x) - \epsilon$, then there should be no penalty. One way to do this would be the following function:

$$\text{ReLU}(\mathbb{E}_{p \sim P'}[\eta(p')] - \eta(p) + \epsilon)$$

The problem with this is that ReLU is not a smooth function, which means it is liable to the vanishing gradient problem. This is because the gradient is 0, for $x < 0$, which prevents weights from being updated. This can be problematic in learning,

so instead softplus, a smooth approximation, is used instead. These two functions are plotted in Figure 2.1.

$$\text{softplus}(x) = \ln(1 + \exp(x))$$

Note that, in the sum of ReLU case, the softplus function in the template is used as the activation function in training, for the same reason. However, when verifying the candidate martingales, the ReLU function is used.

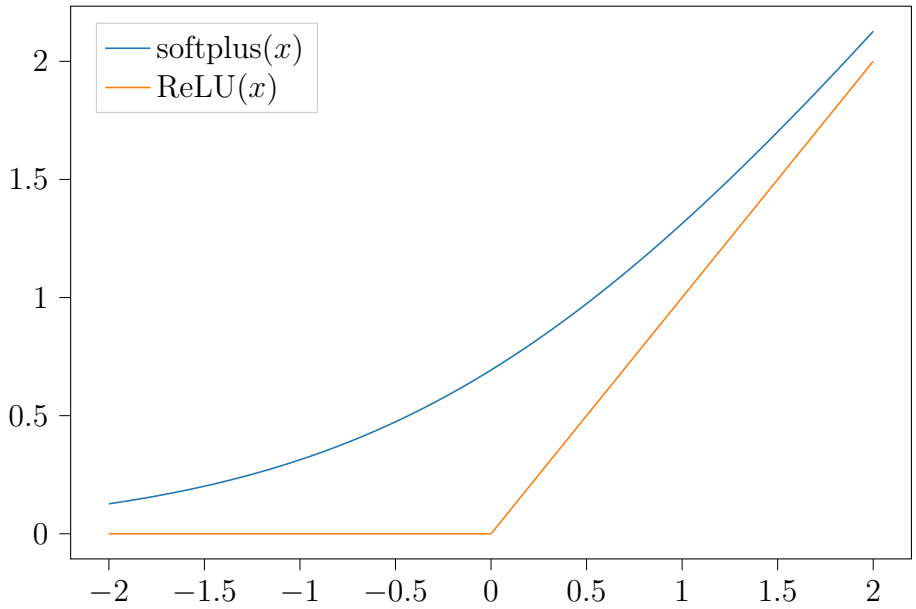

Figure 2.1: The ReLU and softplus activation functions

The loss function for each program state is used to produce the overall optimisation problem in training, given below. This objective is non-linear and non-convex, and gradient descent is used for optimisation.

$$\arg\min_{W,b} \frac{1}{|\mathcal{P}|} \sum_{(p,P')\in\mathcal{P}} L(p, P')$$

Here $\mathcal{P}$ is the training data. While the symbolic expression for the post-expectation $\mathbb{E}[\eta(X_{i+1}) \mid X_i = x]$ isn't used in training, it is used in verification. Marginalising probabilistic programs is done in multiple ways. For discrete programs, a data structure called the symbolic store tree is used, which enumerates all the possible outcomes of random variables. For continuous programs, moments are used.

To conclude, this section has illustrated Farkas' lemma, and given examples of the kinds of programs it can be applied to. It has also given an insight into the limitations of Farkas' lemma by giving an example of a program it cannot be applied to. Finally, it has shown how CEGIS can be used to synthesise RSMs, a method which is built upon in this project.

2.4 Verification of Martingales

This section introduces the process of verifying a function satisfies the conditions to be a martingale. The verifier is an important component in the CEGIS-based architecture that is introduced in this project.

To do this, some foundational concepts in first-order logic are recalled [Kroening and Strichman, 2008; Ben-Ari, 2012]. A logical formula is true with respect to a structure. A structure gives an interpretation to predicate, function and constant symbols in a first-order language, as well as free variables in a formula.

In practice, first-order languages have an intended interpretation for non-logical symbols. Therefore, first-order languages are considered with respect to a theory, a set of axioms, that fixes the interpretation of non-logical symbols. Hence, structures only need to interpret the free variables in a formula.

A formula is *satisfiable* if it is true for some structure, and a formula is *valid* if it is true for every structure. Further, a structure in which a formula holds is called a *model* of that formula.

Satisfiability modulo theories (SMT) is the problem of determining whether a formula is satisfiable with respect to a first-order theory. If the formula is satisfiable, the SMT solver will produce a certificate, in the form of a model. Typically, SMT solvers are applied to quantifier-free fragments of a first-order theory. In these fragments, all free variables are implicitly considered to be existentially quantified. Modern SMT solvers combine techniques from SAT solvers such as DPLL with theory-specific solvers.

Note that the problem of satisfiability is closely related to the problem of validity, through the following logical tautology:

$$\forall x\ P(x) \iff \neg\exists x\ \neg P(x)$$

Therefore, SMT solvers can be used to determine the validity of a statement $P(x)$ by determining whether its negation is unsatisfiable. It is worth highlighting two important theories:

- Theory of linear real arithmetic (LRA). In this theory, atoms are of the form $\sum_i c_i x_i \bowtie d$, where c_i and d are rational constants, x_i is a variable and $\bowtie \in \{=, <, >, \leq, \geq\}$. SMT solvers can handle this theory with the simplex algorithm. An example of a formula is $2x + \frac{1}{3}y \leq 19$.
- Theory of nonlinear real arithmetic (NRA). In this theory, atoms are of the form $\sum_i c_i M_i \bowtie d$, where c_i and d are rational constants, M_i is a monomial and $\bowtie \in \{=, <, >, \leq, \geq\}$. SMT solvers can handle this theory with cylindrical algebraic decomposition. An example of a formula is $xy^2 + x^3 \geq 14$.

Both theories are decidable, however since SAT can be encoded in these theories, they are as difficult as SAT from a complexity-theoretic point of view.

SMT solvers play an essential role in the CEGIS architecture. Synthesising martingales involves solving the formula $\exists\eta.\forall x.P(\eta, x)$. Note that, this is technically a second-order logical formula, since η is a function. However, since parametric templates are used, this becomes a first-order logical formula.

The learner proposes a martingale η, and the verifier, implemented with a SMT solver, must determine whether $\forall x.P(\eta, x)$ holds. In other words, it must determine the validity of $P(\eta, x)$, where η is bound and x is free. As discussed, this can be solved by determining whether $\neg P(\eta, x)$ is unsatisfiable.

If it is unsatisfiable, then the martingale is correct. If it is satisfiable, the martingale is not correct. Since the formula is satisfiable, there must be a model of $\neg P(\eta, x)$. This model is a counterexample that disproves $\forall x.P(\eta, x)$, and can be given to the learner to refine its proposal.

Chapter 3

Design

In Chapter 2, two types of martingales for solving the quantitative reachability problem were introduced. This chapter presents a family of novel methods for the synthesis of these martingales. The four methods that are developed are categorised in Figure 3.1. This classification is based on two dichotomies. The first is whether ISMs or RepSMs are being synthesised. The second is whether the algorithm is program-aware or program-agnostic. The program-aware methods use the structure of the program to learn the martingale, whereas in the program-agnostic methods, learning is entirely data-driven.

Program-Aware Synthesis of ISMs	Program-Agnostic Synthesis of ISMs
Program-Aware Synthesis of RepSMs	Program-Agnostic Synthesis of RepSMs

Figure 3.1: Table of the algorithms developed

The chapter first develops the program-aware synthesis of ISMs. In doing this, it explores the design space of parametric templates and loss functions. An important desideratum is devising a parametric template that represents a tangible improvement over previous methods, while also allowing parameters to be efficiently learnt. Furthermore, the synthesis of RSMs is a decision problem with a binary outcome, while the synthesis of ISMs and RepSMs is an optimisation problem. Therefore a key goal is to construct a loss function that balances optimising the bound with minimising constraint violation.

After this program-aware algorithm for ISMs is developed, it is adapted to give a program-agnostic algorithm. Afterwards, it is shown how these ideas can be extended from ISMs to RepSMs, and program-aware and program-agnostic algorithms for this structure are also introduced.

3.1 Parametric Template

This section begins the development of program-aware synthesis of ISMs. Like previous methods for constructing martingales, the structure of the martingale is fixed to a particular template. The problem is then to select parameters to create a martingale from that template. The choice of template is, therefore, a key attribute in the design of the algorithm, since it constrains the space of functions that can be learnt.

A limitation of previous methods is the use of a linear template. A natural starting point for this project is the sum of ReLU (SOR) template, seen earlier. These correspond to neural networks with one hidden layer.

$$\eta(x) = \sum_{i=1}^{h} \text{ReLU}\left(\sum_{j=1}^{|\text{Vars}|} W_{i,j}x_j + b_i\right)$$

One nice property of this template is that the function is nonnegative, ensuring that this condition does not need to incorporated into the loss function. Furthermore, it can be seen that this template subsumes the linear template. Any linear function, that is a valid ISM, can be trivially encoded into this template.

Furthermore, this template can encode ISMs for probabilistic programs, where a linear template is insufficient. An example of such an ISM is given below.

$$\eta(\mathit{red}, \mathit{blue}, \mathit{error}) = \max(0.0025\mathit{red}, 0) + \max(0.0025\mathit{blue}, 0) + \max(\mathit{error}, 0)$$

This martingale can be used for the faulty marble collector in Chapter 1. The reason that this template is effective in this example is that it constructs functions that are piecewise. In this martingale, branching occurs on the conditions $\mathit{red} > 0$, $\mathit{right} > 0$ and $\mathit{error} = 0$, leading to a piecewise function with eight cases.

To reason about piecewise functions more formally, we say that a function $f : \mathbb{R}^n \to \mathbb{R}$ is *continuous piecewise linear* (PWL), if there is a finite number of closed sets, whose union is $\mathbb{R}^n$, and f is affine over each set. PWL functions are straightforward to grasp in the one-dimensional case; they are functions made of a finite number of line segments. An example of this is given in Figure 3.2.

We have found that many programs that cannot be handled by linear templates can be verified with nonnegative PWL functions. This is because the piecewise structure allows conditional behaviour in the probabilistic program to be mirrored in the martingale. This is especially the case, when disjunctions and conjunctions are involved.

While all SOR functions are nonnegative PWL, there are PWL functions that cannot be encoded as SOR functions. This is a corollary of the following result.

Proposition 4. *All SOR functions are convex.*

A nonconvex PWL such as in Figure 3.3 cannot be represented as a SOR function. Therefore, we would like to find a template that is capable of representing all nonnegative CPWL functions. To do this, we define a more general ReLU neural network.

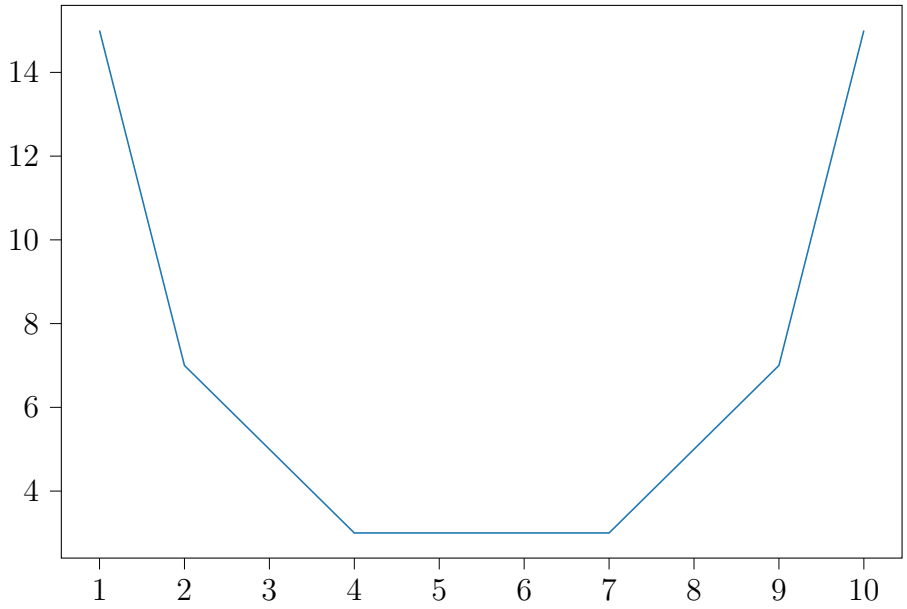

Figure 3.2: Example of a one-dimensional PWL function

A ReLU neural network with k hidden layers, layer sizes $n_1, \ldots, n_k$, and affine transformations $T_1 : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}, \ldots, T_k : \mathbb{R}^{n_k} \to \mathbb{R}$ is defined as:

$$f = T_k \circ \text{ReLU} \circ T_{k-1} \circ \ldots \circ \text{ReLU} \circ T_1$$

Theorem 5 (Arora, Basu, Mianjy, and Mukherjee [2016])**.** *A function $f : \mathbb{R}^n \to \mathbb{R}$ can be computed by a ReLU neural network iff f is PWL.*

While ReLU neural networks are capable of encoding PWLs, we are only interested in nonnegative PWLs. We therefore use the following template.

$$f = \text{Sum} \circ \text{ReLU} \circ T_k \circ \text{ReLU} \ldots \circ \text{ReLU} \circ T_1$$

Here $\text{Sum} : \mathbb{R}^n \to \mathbb{R}$ is a function that sums up the individual components.

$$\text{Sum}(x) = \sum_{i=1}^{n} x_i$$

This ensures that the network is nonnegative. Note, that this is actually a generalisation of the SOR template, allowing it to have more than one hidden layer.

One problem is that ReLU is a non-smooth function. As discussed earlier, the straightforward solution is to use softplus as an approximation in training. However, this poses a problem with ISMs. ISMs learn a function that gives a probability bound. Probabilities fall in the interval $[0, 1]$, and the softplus function is not a good approximation in this small interval. Our experiments haven shown that this makes it very difficult to learn an ISM.

The solution is to learn a function that produces scaled probabilities. Rather than learning a function that gives a probability bound p, the method learns a

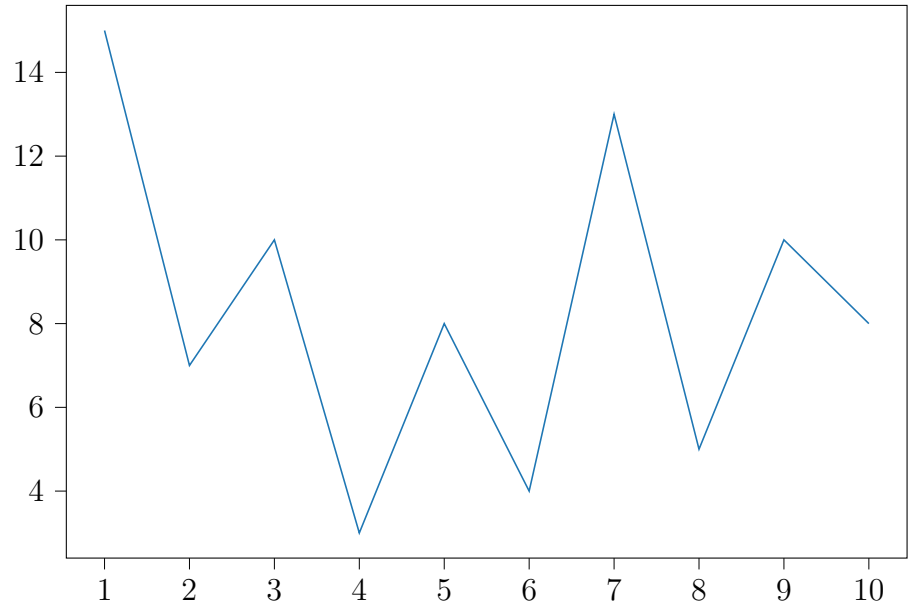

Figure 3.3: Example of a one-dimensional nonconvex PWL function

function that produces βp. The scaled probabilities fall in the interval $[0, \beta]$, and for large values of β, softplus produces better approximations, and this leads to the method performing much better. In this project, $\beta = 1000$ was used.

While this general ReLU template was used for almost all examples in this project, there was one example which used the sum of squares (SOS) template, shown below. This will be expanded upon later.

$$\eta(x) = \sum_{i=1}^{h} \left(\sum_{j=1}^{|\text{Vars}|} W_{i,j} x_j + b_i \right)^2$$

To conclude, this section has identified the class of nonnegative PWL functions to construct ISMs from. This class of functions represents a concrete improvement over the both linear templates and SOR templates. Further, these functions can be represented as neural networks, and have smooth approximations.

3.2 Loss Function

This section continues the development of the program-aware synthesis of ISMs. Having identified a neural template for these martingales, this section introduces a loss function to fit the parameters for this template. Recall that ISMs must satisfy the following two conditions.

- Non-increasing condition: $\eta(x) \geq \mathbb{X}\eta(x)$ for all $x \in I \cap P$
- Indicating condition: $\eta(x) \geq 1$, for all $x \in I \setminus P$

However, as mentioned we wish to learn a scaled ISM. So the indicating condition becomes $\eta(x) \geq \beta$. Notice that the non-increasing condition applies to the safe states $I \cap P$, and the indicating condition applies to unsafe states $I \setminus P$. So each data point in the training data is relevant to only one condition. If a data point is safe, then we want to apply a penalty if $\eta(s)$ is less than $\mathbb{X}\eta(s)$. Otherwise if a data point is unsafe, we want to apply a penalty if $\eta(s)$ is less than β.

A starting point is the following loss function. Note that $\mathcal{P}$ is the training dataset.

$$\mathcal{L}(\eta) = \frac{1}{|\mathcal{P}|} \sum_{x \in \mathcal{P}} \text{ReLU}\left(\begin{cases} \mathbb{X}\eta(x) - \eta(x) & x \in I \cap P \\ \beta - \eta(x) & \text{otherwise} \end{cases}\right)$$

The piecewise expression chooses whether $\eta(x) \geq \beta$ or $\eta(x) \geq \mathbb{X}\eta(x)$ is the relevant condition for the data point. Then the ReLU function only applies a penalty if the condition is actually violated. Furthermore, the penalty is linearly proportional to the extent of the violation.

However, the quality of the ISM depends on the bound $\eta(x_0)$. It is trivial to create an ISM $\eta(x) = \beta$, that is not useful, and this is what will happen with the ISM above. The solution is to add an additional component $\eta(x_0)$, to encourage the learning process to find martingales with lower bounds. The hyper-parameter λ_{bound} controls how much priority is given to minimising the bound, relative to the minimising constraint violation.

$$\mathcal{L}(\eta) = \frac{1}{|\mathcal{P}|} \sum_{x \in \mathcal{P}} \text{ReLU}\left(\begin{cases} \mathbb{X}\eta(x) - \eta(x) & x \in I \cap P \\ \beta - \eta(x) & \text{otherwise} \end{cases}\right) + \lambda_{\text{bound}}\eta(0)$$

Experiments have shown that the non-increasing condition is more difficult to achieve, compared to the indicating condition. Therefore, the network learns better when the loss function prioritises the non-increasing condition. This motivates the following improvement to our loss function. The function is rewritten to use two components $\mathcal{L}_{\text{safe}}$ and $\mathcal{L}_{\text{unsafe}}$. The hyperparameters λ_{safe} and λ_{unsafe} are used to prioritise the non-increasing condition.

$$\mathcal{L}(\eta) = \lambda_{\text{safe}}\mathcal{L}_{\text{safe}}(\eta) + \lambda_{\text{unsafe}}\mathcal{L}_{\text{unsafe}}(\eta) + \lambda_{\text{bound}}\eta(0)$$

Then the functions $\mathcal{L}_{\text{safe}}$ and $\mathcal{L}_{\text{unsafe}}$ handle the non-increasing condition and indicating condition respectively.

$$\mathcal{L}_{\text{safe}}(\eta) = \frac{1}{|\mathcal{P}|} \sum_{\substack{x \in \mathcal{P} \\ x \in I \cap P}} \text{ReLU}(\mathbb{X}\eta(x) - \eta(x))$$

$$\mathcal{L}_{\text{unsafe}}(\eta) = \frac{1}{|\mathcal{P}|} \sum_{\substack{x \in \mathcal{P} \\ x \in I \setminus P}} \text{ReLU}(\beta - \eta(x))$$

Note that the loss function uses $\mathbb{X}\eta$, the post-expectation of the martingale. This function is computed as a symbolic expression, and embedded into the loss function. This is why this method is an instance of program-aware synthesis, since the structure of the program is used in learning. The exact method for constructing $\mathbb{X}\eta$ is discussed in Chapter 4.

To improve learning it is helpful to make the loss function smooth, to avoid the vanishing gradient problem. In fact, our method will actually use both non-smooth and smooth versions of the loss function. Our choice of template makes it straightforward to construct a smooth version of $\eta(x)$. Despite this, it is not always possible to construct a smooth version of $\mathbb{X}\eta(x)$. In cases where it is, this is done by taking the non-smooth $\eta(x)$, computing the post-expectation $\mathbb{X}\eta(x)$, and then, from this expression, constructing the smooth version.

3.3 Synthesis of ISMs

This section finishes the development of the program-aware synthesis of ISMs. It shows how the template and loss function devised in the previous sections can be incorporated into an overall method for synthesising ISMs.

Like the neural synthesis of ISMs, a CEGIS architecture is used, involving the interaction between a learner and a verifier. The first step is generating the initial training data, by executing the probabilistic program. This can be done multiple times, and each time execution occurs up to a maximum number of iterations. The states discovered during these executions form the initial training data.

Then CEGIS iterations are performed. These consist of a learner using the training data to produce parameters, and a verifier producing counterexamples. There are two conditions that need to be satisfied, so counterexamples for both conditions will be added to the dataset. Further, unlike in the neural synthesis of RSMs, our method allows multiple counterexamples to be added.

3.3.1 Learning

These iterations can be split into three stages: a warm-up iteration; the main iterations; a final iteration. The first iteration is the warm-up iteration. The rationale for this iteration is that not many, if any, unsafe states will be present in the initial training data, if unsafe states occur rarely. Therefore, the warm-up iteration compensates for this by learning a martingale from the initial training data, and allowing unsafe states to be generated as counterexamples.

The difference between a warm-up iteration and a normal iteration, is that only a small number of iterations are needed, and the learnt parameters are discarded afterwards. After this, the main CEGIS iterations are performed. Several of these iterations may be needed. Experimentation has found that finding parameters that are perfect, and that produce no counterexamples, is difficult. So the goal of these iterations is instead to find a set of parameters that produces a low number of counterexamples, for each condition.

Then the goal is to move from parameters that is almost correct to parameters that are perfectly correct. In [Abate et al., 2021c], this is addressed through rounding. Many candidate martingales are generated by rounding the parameters to various precisions. This method was considered in this project. A more advanced method based on evolutionary algorithms was also considered. In this method, candidate martingales would be generated by adding random noise. Then the candidates with the lowest number of counterexamples would be kept. This could be

repeated for a number of rounds.

However, these methods were expensive in terms of computation, and even though they were capable of finding valid martingales, they would not necessarily produce good bounds. The method that is used in this algorithm, is instead to have a special CEGIS iteration as the final part of the algorithm. In this iteration, the parameters are saved at every 100 iterations. Then these parameters form the set of candidate martingales. Then the candidate martingales are filtered to only valid martingales. Finally, the martingale which produces the best initial bound is chosen. Note that parameters could be saved more frequently than once every 100 iterations, but verifying martingales is expensive, so this would lead to an increased runtime.

To improve the accuracy of the final CEGIS iteration, it uses a non-smooth loss function, on the basis that the parameters found by the previous iteration should be close to a valid martingale, so the vanishing gradient problem should not impede the efficacy of this iteration.

3.3.2 Verification

One part of the algorithm, that is yet to be discussed, is the implementation of the verifier. Consider the non-increasing condition $\eta(x) \geq \mathbb{X}\eta(x)$ for all $x \in I \cap P$. This can be encoded as:

$$\forall x \in S.\ (x \in I \cap P) \implies (\eta(x) \geq \mathbb{X}\eta(x))$$

This can be checked by ensuring that the following SMT formula is unsatisfiable:

$$(x \in I \cap P) \wedge (\eta(x) < \mathbb{X}\eta(x))$$

To generate multiple counterexamples, it is necessary to change the formula, so that previously found counterexamples are not found again. More precisely, the formula is modified so that future counterexamples are not 'too close' to past counterexamples. Suppose $x^{(1)}, \ldots, x^{(n)} \in \mathbb{R}^{|\text{Vars}|}$ are the past counterexamples. Then the following SMT formula is used:

$$(x \in I \cap P) \wedge (\eta(x) < \mathbb{X}\eta(x)) \wedge \bigwedge_{1 \leq i \leq n} \left(\bigvee_{1 \leq j \leq |\text{Vars}|} \left| x_j - x_j^{(i)} \right| \geq \delta \right)$$

In other words, for each one of the past counterexamples, the next counterexample must differ in at least one part of program state by δ. In this project, δ was set to 1.

One problem, that was found in a small number of probabilistic programs, was that a valid martingale had to have zero in some of its parameters. However, it is very difficult for a neural network to learn sparse models, where parameters are exactly zero. Instead, parameters very close to zero will be learnt. This will lead to the algorithm failing to learn a valid martingale. To handle this problem, a rounding strategy was applied to certain problems, where parameters very close to zero, would be rounded to zero before verification. This made it possible to successfully learn martingales for these problems.

3.4 Program-Agnostic Synthesis

Previously, the program-aware synthesis of ISMs was introduced. This section discusses certain drawbacks of this algorithm, and this motivates the development of the program-agnostic synthesis of ISMs.

In the program-aware algorithm, a key detail is that the post-expectation $\mathbb{X}\eta$ is embedded into the loss function of the program. This has two major disadvantages. The first is that the size of $\mathbb{X}\eta$ will scale with the size of the program. This affects the performance of the learning procedure. A second problem is that for learning to be efficient, a smooth version of $\mathbb{X}\eta$ needs to be used. However, even if η has a smooth version, it might be difficult to find a smooth version of $\mathbb{X}\eta$. One reason might be that $\mathbb{X}\eta$ uses indicator functions.

An alternative to embedding $\mathbb{X}\eta$ into the loss function, is computing an approximation of $\mathbb{X}\eta(x)$. This leads to a program-agnostic algorithm, which does not use the structure of the program in learning. Instead for each state in the dataset, successor states are sampled, by simulating the update statement of the loop. This can be used to produce an estimate for $\mathbb{X}\eta(x)$. Then the learning process does not require a symbolic expression for $\mathbb{X}\eta(x)$, and instead only requires data from execution traces. Hence, it is entirely data-driven. Note that, the $\mathbb{X}\eta(x)$ is still required in verification.

Let $x'^{(1)}, \ldots, x'^{(n)}$ be the sampled successor states. In the neural synthesis of RSMs, a standard Monte Carlo estimate is used to estimated $\mathbb{X}\eta(x)$:

$$\mathbb{X}\eta(x) \approx \frac{1}{n} \sum_{i=0}^{n} \eta(x'^{(i)})$$

However, there are some circumstances when this estimate is problematic. This is when there are some regions of the state space that have an extremely low probability of appearing as the successor state, but have a large impact on the value of $\mathbb{X}\eta(x)$. Consider the program below. In the update statement, the random variable p is sampled from a Bernoulli distribution with a 0.001 probability of $p = 0$.

$t = 1$
$error = 0$
while $t < 10$:
    $assert(error \neq 1)$
    $p \sim \mathsf{Bernoulli}(0.999)$
    if $p == 1$:
        $q \sim \mathsf{Bernoulli}(0.5)$
        if $q == 1$:
            $t = t + 1$
    else:
        $error = 1$

The event $p = 0$, is an example of a *rare event*. The presence of rare events mean that a conventional Monte Carlo estimate will have a high variance. This estimate will be too inaccurate to be useful, unless an unreasonably large number

of samples are used. This is especially problematic in the context of quantitative safety properties, since these properties are typically used to bound the probability of rare events.

To handle this complication, this project uses importance sampling to reduce variance. Importance sampling allows samples to be drawn from a proposal distribution $q(x)$, to compute properties of a target distribution $p(x)$. In this case, the target distribution is the original distribution that is specified in the probabilistic program, and the proposal distribution is a new distribution that increases the probability of rare events. The following derivation shows how this is achieved concretely.

$$\begin{aligned}\mathbb{E}_p[f(x)] &= \int p(x)f(x)\ \mathrm{dx} \\ &= \int q(x)\frac{p(x)}{q(x)}f(x)\ \mathrm{dx} \\ &= \mathbb{E}_q\left[\frac{p(x)}{q(x)}f(x)\right]\end{aligned}$$

To compute the expectation of $f(x)$ under $p(x)$, one can compute the expectation of $(p(x)/q(x))f(x)$ under $q(x)$. In other words, samples from the $q(x)$ are weighted by the ratio $p(x)/q(x)$. This is called the likelihood ratio.

This is implemented in our project as follows. A parameter to the program-agnostic algorithm is any random variables, which should be sampled with an alternative distribution. For instance, in the program above Bernoulli(0.999) could be replaced with Bernoulli(0.9).

Note that, this is only required in certain cases. For other cases, no random variables will need to be given an alternative distribution. Simulation will now produce samples $x'^{(1)}, \ldots, x'^{(n)}$ together with corresponding likelihood ratios $l^{(1)}, \ldots, l^{(n)}$. A weighted average is then used to approximate the post-expectation.

$$\mathbb{X}\eta(x) \approx \frac{1}{n}\sum_{i=0}^{n}\eta(x'^{(i)})l^{(i)}$$

To conclude this section, a variation of the program-aware algorithm has been introduced, that does not require embedding $\mathbb{X}\eta$ into the loss function.

3.5 Synthesis of RepSMs

Previously, the synthesis of ISMs has been developed. This section extends these ideas to develop a method for synthesising RepSMs. Recall that a repulsing supermartingale must satisfy the following conditions.

1. Decreasing condition: $\mathbb{X}\eta(x) \leq \eta(x) - \epsilon$ for all $x \in I \cap P$
2. Lower bound condition: $\eta(x) \geq 0$ for all $x \in I \setminus P$
3. Bounded differences condition: $|\eta(X_{t+1}) - \eta(X_t)| \leq c$

4. Initial state condition: $\eta(x_0) < 0$

The template used is the negation of a ReLU neural network. This function will either produce a negative value or zero. This allows the lower bound condition to be satisfied for unsafe states. It also allows safe states to have negative values, which is necessary for the initial state condition and the decreasing condition.

Notice that the decreasing condition is relevant for only safe states, and the lower bound condition is relevant for unsafe states. This is a similar situation to ISMs. Therefore, a loss function is used, with separate components for the safe and unsafe states.

$$\mathcal{L}(\eta) = \lambda_{\text{safe}}\mathcal{L}_{\text{safe}}(\eta) + \lambda_{\text{unsafe}}\mathcal{L}_{\text{unsafe}}(\eta)$$

The function $\mathcal{L}_{\text{safe}}$ focusses on ensuring the decreasing condition, by penalising safe states which do not decrease by ϵ in expectation. Like ISMs, there are program-aware and program-agnostic variations of the RepSM synthesis algorithm. In the program-aware algorithm, the exact symbolic expression for $\mathbb{X}\eta$ is used. In the program-agnostic algorithm, sampling is used to produce an estimate.

$$\mathcal{L}_{\text{safe}}(\eta) = \frac{1}{|\mathcal{P}|} \sum_{\substack{x \in \mathcal{P} \\ x \in I \cap P}} \text{ReLU}(\mathbb{X}\eta(x) - \eta(x) + \epsilon)$$

The function $\mathcal{L}_{\text{unsafe}}$ focusses on ensuring the lower bound condition, by penalising unsafe states that are negative.

$$\mathcal{L}_{\text{unsafe}}(\eta) = \frac{1}{|\mathcal{P}|} \sum_{\substack{x \in \mathcal{P} \\ x \in I \setminus P}} \text{ReLU}(-\eta(x))$$

Similar to ISMs, we do not simply want a valid RepSM but one which produces a tight bound. To do this, the loss function is modified to encourage RepSMs which give lower bounds. Recall that the bound for the RepSM is given as follows:

$$\alpha \frac{\gamma^A}{1-\gamma}$$

Where:

$$\begin{aligned}
\alpha &= \exp\left(\frac{\epsilon m_0}{(c+\epsilon)^2}\right) \\
\gamma &= \exp\left(-\frac{\epsilon^2}{2(c+\epsilon)^2}\right) \\
A &= \lceil |m_0|/c \rceil \\
m_0 &= \eta(0)
\end{aligned}$$

This was a difficulty in the linear synthesis of RepSM, since this bound was non-linear. Notice that the bound is determined by ϵ, c and m_0. RepSMs can be arbitrarily rescaled, so we set $\epsilon = 1$. Then one way to incorporate the bound into the loss function is to add c and m_0 as their own components.

$$\mathcal{L}(\eta) = \lambda_{\text{safe}}\mathcal{L}_{\text{safe}}(\eta) + \lambda_{\text{unsafe}}\mathcal{L}_{\text{unsafe}}(\eta) + \lambda_c c + \lambda_{m_0} m_0$$

It was found that this is an unsatisfactory solution, as the bound found was overly sensitive to the balance between λ_c and λ_{m_0}. This motivated a different approach, which is to effectively embed the bound directly into the loss function.

However, this is problematic owing to the definition of A, since it uses the ceil function which is not smooth, and prevents gradients from being propagated. To address this, the bound is turned into an overestimate, by removing the ceil function. In other words, A becomes $|m_0|/c$. To see that this produces an overestimate, note that $\gamma < 1$. Another concern might be that the exp function leads to issues with numerical stability. However, in practice this was not found to be an issue.

By incorporating the bound into the loss function, this also takes care of the initial state condition $\eta(x_0) < 0$. By encouraging lower bounds, this will encourage lower values of $\eta(x_0)$.

$$\mathcal{L}(\eta) = \lambda_{\text{safe}}\mathcal{L}_{\text{safe}}(\eta) + \lambda_{\text{unsafe}}\mathcal{L}_{\text{unsafe}}(\eta) + \lambda_{\text{bound}}\left(\alpha\frac{\gamma^A}{1-\gamma}\right)$$

One question that remains is how to compute c, the maximum difference in the martingale over one time step. This is needed for both learning to use in the loss function, and verification to ensure that the bounded differences condition is satisfied. The computation of c differs depending on whether the program-aware or the program-agnostic algorithm is used.

In the program-aware algorithm, a symbolic expression for c is required as an input. This can be then be embedded in learning and verification. In the program-agnostic algorithm, c is estimated. First, the largest difference between a state and a sampled successor state is found. Then a small constant is added to be safe. This value is used as c. However, to ensure that the loss function is smooth, the largest difference between a state and a sampled successor state is actually approximated with the LogSumExp function, which is a smooth approximation of the max function. It is a generalisation of softplus.

$$\text{LogSumExp}(x_1, \ldots, x_n) = \log(\exp(x_1) + \ldots + \exp(x_n))$$

In verification, there are four conditions to consider. When generating counterexamples, only the decreasing and lower bound conditions are used. These conditions are converted to SMT formulae, in the same way that the conditions for ISMs are.

The bounded differences and initial state conditions are only checked at the end. The initial state condition $\eta(x_0) < 0$ does not need to handled by an SMT solver, and can be checked by computing $\eta(x_0)$. The bounded differences condition $|\eta(X_{t+1}) - \eta(X_t)| \leq c$ states that the martingale does not change by more than c in a time step. This can be encoded as the following SMT statement.

$$\forall x, x' \in S.\ (T(x, x') \wedge (x \in I \cap P)) \implies |\eta(x) - \eta(x')| \leq c$$

The predicate $T(x, x')$ holds only if x' is a possible successor state of x. The construction of $T(x, x')$ is discussed in Chapter 4. This formula can be checked, by ensuring the following quantifier-free statement is unsatisfiable.

$$T(x, x') \wedge (x \in I \cap P) \wedge |\eta(x) - \eta(x')| > c$$

To conclude this section, a method for the synthesis for RepSMs has been developed. This has similarities with the method for ISMs such as the structure of the loss function, and differences such as using an overestimate for the bound. Additionally, a key difference between the program-aware and program-agnostic algorithms is how the value c is computed.

Chapter 4

Implementation

4.1 Introduction

4.1.1 Probabilistic Programming Language

The methods described in the previous chapter are implemented in a tool, programmed in Python. The syntax for the probabilistic programming language used by the tool is described below. Note that x stands for a variable, N stands for a real number.

$$
\begin{aligned}
\mathit{Atom} &::= x \mid N \\
\mathit{BinOp} &::= + \mid - \mid * \mid < \mid \leq \mid > \mid \geq \\
\mathit{Dist} &::= \mathsf{Bernoulli}(E) \mid \mathsf{Uniform}(E, E) \mid \mathsf{Gaussian}(E, E) \mid \ldots \\
\mathit{Expr} &::= \mathit{Atom} \mid \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr} \mid -\ \mathit{Expr} \\
\mathit{Stmt} &::= x = \mathit{Expr} \mid x \sim \mathit{Dist} \mid \textbf{if}\ (\mathit{Expr})\ \{\mathit{Block}\}\ \textbf{else}\ \{\mathit{Block}\} \mid \\
&\quad\ \ \textbf{while}\ (\mathit{Expr})\ \{\mathit{Block}\} \\
\mathit{Block} &::= \mathit{Stmt}; \mathit{Block} \mid \mathit{Empty}
\end{aligned}
$$

The implementation of the language is standard, and hence described only briefly. The abstract syntax tree described above is encoded with algebraic data types, with the *dataclasses* and *typing* packages. The *Lexer* component reads a stream of characters, and isolates them into tokens, using regular expressions. The *Parser* component converts the stream of tokens into an abstract syntax tree. The parser is a recursive descent parser. In addition, it uses precedence climbing to handle expressions. This is described in the pseudocode below.

function *parseExpression*(*currentPrecendence*)
    result = *parseAtom*()
    while *token* is *BinOp* and *precendence* > *currentPrecendence*:
        rhs = *parseExpression*(*currentPrecendence* + 1)
        result = *buildBinOpNode*(*operator*=*token*, *lhs*=*result*, *rhs*=*rhs*)
    return *result*

Several components that operate on the abstract syntax tree, do so through a combination of recursion and pattern matching. One of these is the *TypeChecker* component. This ensures that the program well-typed. In addition, it builds up a symbol table, and ensures identifiers are defined before being used. It also makes sure that program is a single-loop program.

4.1.2 Learning and Verification

The loss functions and martingales are implemented with the Jax library [Bradbury et al., 2018]. Training is implemented with the Adam optimiser from the Optax library [Kingma and Ba, 2017; Hessel et al., 2020]. The Z3 library is used for SMT solving to verify candidate martingales [De Moura and Bjørner, 2008].

An *Interpreter* component has two functions. Firstly, it is responsible for executing the probabilistic program to generate initial training data. Secondly, in the program-agnostic algorithms it is responsible for generating successor states, to estimate the post-expectation of the martingale.

An *Analysis* component is responsible for computing the post-expectation of martingales. This is needed for program-aware algorithms and for verifying candidate martingales. This component is described in further detail in its own section. This is implemented with the SymPy library [Meurer et al., 2017].

A *ReachabilityAnalysis* component is responsible for generating an SMT predicate T that is used when verifying the bounded differences condition for RepSMs. The SMT predicate $T(x, x')$ holds only when x' is reachable from x in a single time step. This is also discussed in further detail later.

There is a function *translateToJax* that converts a SymPy expression into Jax code that can be JIT compiled. Additionally, there is a function *translateToZ3* that converts a SymPy expression into a Z3 expression, which is used when verifying candidate martingales.

4.1.3 Additional Components

This tool also implements baseline methods for the synthesis of ISMs and RepSMs using linear programming, as described in Chapter 2. This reuses some of the components that have been discussed, and uses the Pulp library to solve linear programs [Mitchell et al., 2011]. This tool also includes a *Formatter* component, which is used to typeset programs into LaTeX, as used in this report.

4.2 Symbolic Inference

4.2.1 Motivation

This section introduces the *Analyser* component, which plays a key role in generating martingales. This class is responsible for computing a symbolic expression for the expectation of the martingale after the update statement of the loop has executed, i.e. $\mathbb{E}[\eta(X_{i+1}) \mid X_i = x]$, where η is a template for a martingale. This task is called symbolic inference, and the goal is to produce a probability distribution

conditioned on the program state at the beginning of the loop. This is needed for performing verification using an SMT solver, and also for training in the program-aware algorithms.

The state-of-the-art for this task is the algorithm presented in [Gehr et al., 2016]. The original implementation of the algorithm is found in the tool PSI. Rather than reusing the original implementation, a new implementation was written in Python for this project. This implementation follows the basic approach of PSI closely, but does not aim to be a perfect recreation of the PSI tool. There were multiple reasons motivating a new Python implementation, rather than simply reusing the PSI tool.

Firstly, the PSI tool was written in the D programming language. Interacting with this tool from Python, would require a clunky translation layer, that encodes the problem into a format comprehensible to PSI, and decodes the response from PSI's format.

Secondly, the reimplementation is based on the widely used SymPy computer algebra system (CAS). Building on top of a CAS meant that a lot of the functionality in PSI did not have to reimplemented. For instance, functions to perform basic algebraic simplifications or integrate continuous terms are provided by SymPy. Owing to the significant effort that has been put into SymPy's development, these parts of SymPy are well-developed. Further, SymPy is written in Python making it effortless to interface with.

It is useful to compare our method for symbolic inference, with approaches used in other tools for synthesising martingales. Most previous work has used linear templates, which makes computing the post-expectation trivial, since linearity of expectation can be applied. In Abate et al. [2021c], symbolic store trees were used to handle discrete distributions. Then, ad-hoc methods were used to handle continuous distributions, such as moments. The PSI algorithm is more general than these approaches, and can handle a wider range of probabilistic programs. It is not a complete method, since this would require solving every conceivable symbolic integral. Nonetheless, it is a consequential improvement.

4.2.2 Overview

This sections gives an overview of the symbolic inference algorithm. Recall that the algorithm will be applied to the update statement of a loop. Further note that some variables will be part of the program state, i.e. exist before the update statement begins, whereas other transient variables will be defined locally within the loop.

To elucidate the symbolic inference algorithm, the following example will be used. In this example, x is either incremented by 2 or decremented by 1, with equal probability. Note that x is part of program state, and is defined beforehand, while p is a local to the block.

$p \sim \mathsf{Bernoulli}(0.5)$
if $p == 1$:
    $x = x + 2$
else:
    $x = x - 1$

We wish to consider the post-expectation of a martingale η with respect to this update statement. The first step is to distinguish different assignments to the same variable by giving them indices. This is shown below.

$p_0 \sim \mathsf{Bernoulli}(0.5)$
if $p_0 == 1$:
    $x_1 = x_0 + 2$
else:
    $x_2 = x_0 - 1$
join $\{(x_1, x_2) \mapsto x_3\}$

Note that, x_0 is the only identifier, that refers to a value defined outside of the block. Also observe that the if statement is decorated with some additional information indicated by the join keyword. In particular, the if statement is associated with a set of triples of the form $(x, x') \mapsto x''$. This means that after the if statement x'' refers to x or x' depending on whether the if block or the else block was executed.

After the code is converted to this form, the *encode* algorithm will build a joint PDF, in a step-by-step fashion. It starts with the first line, where p_0 is sampled from a Bernoulli random variable. This produces the following PDF.

$$\frac{\delta(p_0 - 1)}{2} + \frac{\delta(p_0)}{2}$$

The Dirac delta distribution $\delta(x)$ is a way of giving PDFs to discrete distributions. $\delta(x - a)$ can be thought of as a function that is zero, unless $x = a$ where it is infinite. It has the property that:

$$\int_{-\infty}^{\infty} f(x)\delta(x - a) = f(a)$$

Returning to the PDF for p_0, the first summand can be interpreted to say that there is a 0.5 probability that $p_0 - 1 = 0$, i.e. $p_0 = 1$. Then the second summand can be interpreted to say that there is a 0.5 probability that $p_0 = 0$.

Next the assignments x_1 and x_2 will be translated PDFs.

$$\delta(x_1 - x_0 - 2)$$
$$\delta(x_2 - x_0 + 1)$$

The first PDF corresponds to the assignment $x_1 = x_0 + 2$, and the second PDF corresponds to the assignment $x_2 = x_0 - 1$.

These two PDFs need to be reduced to a single PDF for the if statement. To do this first the join mapping $(x_1, x_2) \mapsto x_3$ is incorporated. This changes the PDFs to the following form.

$$\delta(x_1 - x_0 - 2)\delta(x_3 - x_1)$$
$$\delta(x_2 - x_0 + 1)\delta(x_3 - x_2)$$

Note that, x_3 is now present in both PDFs. This means that the variables that do not exist outside the if statement, i.e. x_1 and x_2, can be marginalised out. This leads to the following PDFs.

$$\delta(x_3 - x_0 - 2)$$
$$\delta(x_3 - x_0 + 1)$$

Finally these two PDFs are combined to obtain the following piecewise function. This completes the PDF for the if statement.

$$\begin{cases} \delta(x_3 - x_0 - 2) & \text{if } p_0 = 1 \\ \delta(x_3 - x_0 + 1) & \text{otherwise} \end{cases}$$

Now the PDF for p_0 and the PDF for the if statement are multiplied together, to get a PDF for the entire block of code. This gives $\mathbb{P}[p_0, x_3 \mid x_0]$.

$$\left(\frac{\delta(p_0 - 1)}{2} + \frac{\delta(p_0)}{2}\right)\left(\begin{cases} \delta(x_3 - x_0 - 2) & \text{if } p_0 = 1 \\ \delta(x_3 - x_0 + 1) & \text{otherwise} \end{cases}\right)$$

This finishes the construction of the joint PDF. Now, suppose the martingale η, whose post-expectation we wish to find, is defined as follows.

$$\eta(x) = wx + b$$

Then the expression $\eta(x_3)$ is multiplied with the PDF. This gives the following expression for $\mathbb{P}[p_0, x_3|\ x_0]\eta(x_3)$.

$$(wx_3 + b)\left(\frac{\delta(p_0 - 1)}{2} + \frac{\delta(p_0)}{2}\right)\left(\begin{cases} \delta(x_3 - x_0 - 2) & \text{if } p_0 = 1 \\ \delta(x_3 - x_0 + 1) & \text{otherwise} \end{cases}\right)$$

The final step is to marginalise the variables p_0 and x_3 out. This gives the following value for $\mathbb{E}[\eta(x_3)|\ x_0]$, i.e. the post-expectation.

$$wx_0 - \frac{w}{2} + b$$

Thus, post-expectations can be computed through a combination of the *encode* and *marginalise* algorithm. Note that, *marginalise* is used in two ways. It is used in *encode* to handle if statements. If is also used after *encode* to compute the post-expectation from the joint PDF. The algorithms have the following type signatures, with Vars referring to the set of variable identifiers.

$$\mathit{encode} : \mathit{Block} \to \mathit{PDF}$$
$$\mathit{marginalise} : \mathcal{P}(\text{Vars}) \times \mathit{PDF} \to \mathit{PDF}$$

Subsequent sections will discuss these two algorithms in greater depth.

4.2.3 Producing a Probability Density Function

In this section, the *encode* algorithm is explained more formally. The algorithm is built from the *encodeStmt* helper function.

$$encodeStmt : Stmt \to PDF$$

Pseudocode for the assignment cases of the helper function is given below. The double square brackets indicate that its contents should be treated as an abstract syntax tree.

$$encodeStmt(x = y) = [\![\delta(y - x)]\!]$$
$$encodeStmt(x \sim \mathsf{Bernoulli}(p)) = [\![p\delta(x-1) + (1-p)\delta(x)]\!]$$
$$encodeStmt(x \sim \mathsf{Uniform}(a,b)) = \left[\!\!\left[\begin{cases} \frac{1}{b-a} & \text{if } a \le x \le b \\ 0 & \text{otherwise} \end{cases} \right]\!\!\right]$$
$$encodeStmt(x \sim \mathsf{Gaussian}(\mu,\sigma)) = \left[\!\!\left[\frac{1}{\sigma\sqrt{2\pi}} \exp\!\left(\frac{x-\mu}{\sigma}\right) \right]\!\!\right]$$

When sampling from a Bernoulli random variable, the density function for the distribution is encoded with deltas. For continuous random variables, the standard PDF can be used. For deterministic assignments, a delta is also used.

The *encodeStmt* case for if statements is more involved. Two PDFs are constructed for the if and the else blocks. The join mapping J is then applied to both PDFs by multiplying them with a delta for each element. After this the local variables assigned in the blocks can be marginalised. Finally, the two PDFs are combined in a piecewise expression. The helper function localVars(b) returns variables that are assigned in the block b.

$$encodeStmt(\textbf{if } (c)\ \{b_1\}\ \textbf{else}\ \{b_2\}\ \textbf{join}\ J) = \left[\!\!\left[\begin{cases} pdf_1 & \text{if } c \\ pdf_2 & \text{otherwise} \end{cases} \right]\!\!\right]$$
$$\textbf{where } pdf_1 = marginalise\left(localVars(b_1), encode(b_1) \cdot \left[\!\!\left[\prod_{(x,x')\mapsto x'' \in J} \delta(x - x'') \right]\!\!\right]\right)$$
$$pdf_2 = marginalise\left(localVars(b_2), encode(b_2) \cdot \left[\!\!\left[\prod_{(x,x')\mapsto x'' \in J} \delta(x' - x'') \right]\!\!\right]\right)$$

Finally the definition of *encode* is given. This goes through each statement in a block to construct a combined PDF.

$$encode(Empty) = [\![1]\!]$$
$$encode(s; b) = [\![pdf_1 \cdot pdf_2]\!]$$
$$\textbf{where } pdf_1 = encodeStmt(s)$$
$$\textbf{where } pdf_2 = encode(b)$$

This concludes the description of the *encode* algorithm. This is used to compute an expression of the form $\mathbb{P}[x_1, x_2, \ldots \mid y_1, y_2, \ldots]$, where x_i are variables introduced in the block of code, and y_i are variables that existed before the block of code. After the joint PDF is constructed, it can be combined with the martingale and marginalised to produce the post-expectation.

4.2.4 Solving Delta Integrals

This section explains the *marginalise* algorithm in more detail, explaining how it handles integrals involving delta expressions. The goal of marginalisation is to remove a variable from a joint PDF, or the product of a joint PDF and a martingale. This is done with integration, and the problem can be formulated as follows, where x_i is to be marginalised out of the expression $f(x_1, \ldots, x_n)$.

$$\int_{-\infty}^{\infty} f(x_1, \ldots, x_n) \, \mathrm{d}x_i$$

The method to handle this integral, depends on if the expression $f(x_1, \ldots, x_n)$ contains a delta. This section focusses on the case where it does, using examples from earlier in the chapter. Consider the following integral.

$$\int_{-\infty}^{\infty} \delta(x_1 - x_0 - 2)\delta(x_3 - x_1) \, \mathrm{d}x_1$$

Here the variable of integration is x_1. This can be handled by noting that a factor in this expression is a delta expression that includes the variable x_1, i.e. $\delta(x_3 - x_1)$. Then the expression $x_3 - x_1 = 0$ can be rearranged to obtained $x_1 = x_3$, which can be substituted to get the following:

$$\delta(x_3 - x_0 - 2)$$

A more complex example is given below, where the variable of integration is p_0:

$$\int_{-\infty}^{\infty} (wx_3 + b) \left(\frac{\delta(p_0 - 1)}{2} + \frac{\delta(p_0)}{2} \right) \left(\begin{cases} \delta(x_3 - x_0 - 2) & \text{if } p_0 = 1 \\ \delta(x_3 - x_0 + 1) & \text{otherwise} \end{cases} \right) \, \mathrm{d}p_0$$

This expression is also a product, but there is no factor which is a delta expression involving p_0. However, there is a factor which is a sum of delta expressions involving p_0. So, this sum can be expanded, leading to:

$$\begin{aligned} &\int_{-\infty}^{\infty} \frac{(wx_3 + b)\delta(p_0 - 1)}{2} \left(\begin{cases} \delta(x_3 - x_0 - 2) & \text{if } p_0 = 1 \\ \delta(x_3 - x_0 + 1) & \text{otherwise} \end{cases} \right) \\ &+ \frac{(wx_3 + b)\delta(p_0)}{2} \left(\begin{cases} \delta(x_3 - x_0 - 2) & \text{if } p_0 = 1 \\ \delta(x_3 - x_0 + 1) & \text{otherwise} \end{cases} \right) \, \mathrm{d}p_0 \end{aligned}$$

Then linearity of integration can be applied, and each summand can be marginalised separately leading to:

$$\frac{(wx_3 + b)\delta(x_3 - x_0 - 2)}{2} + \frac{(wx_3 + b)\delta(x_3 - x_0 + 1)}{2}$$

This section has shown how to handle integrals that contain deltas. This is sufficient for programs which use discrete random variables. However, programs which use continuous random variables will require continuous terms to be integrated, i.e. integrals without deltas.

4.2.5 Solving Continuous Integrals

This section addresses how to integrate continuous terms. To do this, the following example will be used, which mixes discrete and continuous distributions. In this piece of code, *error* is set to 1, with probability 0.001. Otherwise, t is incremented with a value sampled uniformly from $[0, 1]$.

$p \sim \mathsf{Bernoulli}(0.999)$
if $p == 1$:
    $q \sim \mathsf{Uniform}(0, 1)$
    $t = t + q$
else:
    $error = 1$

Assigning indices to the different variable assignments gives:

$p_0 \sim \mathsf{Bernoulli}(0.999)$
if $p_0 == 1$:
    $q_0 \sim \mathsf{Uniform}(0, 1)$
    $t_1 = t_0 + q_0$
else:
    $error_1 = 1$
join $\{(t_1, t_0) \mapsto t_2, (error_0, error_1) \mapsto error_2\}$

The following martingale η will be used.

$$\eta(error_0, t_0) = \max(0, b + error_0 w_{error} + t_0 w_t)$$

The first step is to compute:

$$\eta(t_2, error_2)\mathbb{P}(t_2, error_2, p_0 \mid t_0, error_0)$$

After this, the next step is marginalising out $error_2$ and p_0, which gives:

$$\eta(t_2, error_2)\mathbb{P}(t_2 \mid t_0, error_0)$$

This program leads to long expressions for the joint PDF, so the intermediate expressions are omitted. After these steps the result is:

$$\frac{999}{1000}\left(\begin{cases}1 & \text{for } t_0 - t_2 \geq -1 \wedge t_0 - t_2 \leq 0 \\ 0 & \text{otherwise}\end{cases}\right)\max\left(0, b + error_0 w_{error} + t_2 w_t\right) + \frac{1}{1000}\delta\left(-t_0 + t_2\right)\max\left(0, b + t_2 w_t + w_{error}\right)$$

There is one variable left to be marginalised out, t_2. Linearity of expectation can be applied, so we will concentrate on the left summand. Further, we will ignore constant factors. The integral is as follows.

$$\int_{-\infty}^{\infty} \left(\begin{cases} 1 & \text{for } t_0 - t_2 \geq -1 \wedge t_0 - t_2 \leq 0 \\ 0 & \text{otherwise} \end{cases} \right) \max\left(0, b + error_0 w_{error} + t_2 w_t\right) \, \mathrm{d}t_2$$

Note that this term contains does not contain any delta, so it is a continuous integral. Also note the presence of a piecewise expression and a max operator which can be difficult to handle.

The first step is introducing indicator expressions. Both the piecewise expressions and the max function are replaced with Iverson brackets. This gives the following integral.

$$\int_{-\infty}^{\infty} \left(b + error_0 w_{error} + t_2 w_t\right) \left[b + error_0 w_{error} + t_2 w_t \geq 0\right] \\ \left[t_0 - t_2 \geq -1 \wedge t_0 - t_2 \leq 0\right] \mathrm{d}t_2$$

Once this has been done, the Iverson brackets need to *linearised*, so that the only Iverson brackets (containing t_2) are of the form $t_2 \sim e$, where $\sim$ is a comparison operator, and e is an expression. For the second Iverson bracket, this is accomplished by splitting the conjunction, to obtain two new Iverson brackets.

$$\begin{aligned} \left[t_0 - t_2 \geq -1 \wedge t_0 - t_2 \leq 0\right] &\equiv \left[t_0 - t_2 \geq -1\right]\left[t_0 - t_2 \leq 0\right] \\ &\equiv \left[t_2 \leq t_0 + 1\right]\left[t_2 \geq t_0\right] \end{aligned}$$

The first Iverson bracket is more challenging.

$$\left[b + error_0 w_{error} + t_2 w_t \geq 0\right] \equiv \left[t_2 w_t \geq -b + error_0 w_{error}\right]$$

This partially improves the situation. The difficulty is that the result of dividing by w_t depends on its sign. The solution is to do case analysis based on whether w_t is positive, negative or zero. The Iverson bracket can be split into three cases depending on whether w_t is positive, negative or equal to 0.

$$\begin{aligned} \left[t_2 w_t \geq -b + error_0 w_{error}\right] \equiv \; & \left[w_t > 0\right] \left[t_2 \geq \frac{-b + error_0 w_{error}}{w_t}\right] \\ & + \left[w_t < 0\right] \left[t_2 \leq \frac{-b + error_0 w_{error}}{w_t}\right] \\ & + \left[w_t = 0\right]\left[0 \geq -b + error_0 w_{error}\right] \end{aligned}$$

Applying guard linearisation leads to the following expression.

$$\int_{-\infty}^{\infty} \left(b + error_0 w_{error} + t_2 w_t\right) \left[t_2 \geq t_0\right]\left[t_2 \leq t_0 + 1\right]\left[w_t = 0\right] \\ \left[0 \geq -b - error_0 w_{error}\right] + \left(b + error_0 w_{error} + t_2 w_t\right) \\ \left[t_2 \geq t_0\right]\left[t_2 \leq t_0 + 1\right]\left[w_t > 0\right] \left[t_2 \geq \frac{-b - error_0 w_{error}}{w_t}\right] + \\ \left(b + error_0 w_{error} + t_2 w_t\right) \left[t_2 \geq t_0\right]\left[t_2 \leq t_0 + 1\right] \\ \left[w_t < 0\right] \left[t_2 \leq \frac{-b - error_0 w_{error}}{w_t}\right] \, \mathrm{d}t_2$$

Linearity of integration can be applied, so we will consider the second summand.

$$\int_{-\infty}^{\infty} (b + error_0 w_{error} + t_2 w_t) \, [t_2 \geq t_0][t_2 \leq t_0 + 1] \\ [w_t > 0] \left[t_2 \geq \frac{-b - error_0 w_{error}}{w_t} \right] \, \mathrm{d}t_2$$

It is now possible to take advantage of the linearised Iverson brackets, to refine the integration bounds. Note that there are two lower bounds on t_2, and one upper bound. The upper bound is $t_0 + 1$. The two lower bounds are combined to produce:

$$l = \max \left(t_0, \frac{-b - error_0 w_{error}}{w_t} \right)$$

This produces the following integral:

$$[w_t > 0][l < t_0 + 1] \int_l^{t_0+1} b + error_0 w_{error} + t_2 w_t \, \mathrm{d}t_2$$

Note that the $w_t > 0$ indicator has been pulled out of the integral, and the $l < t_0 + 1$ indicator has been added, since the expression should be 0 otherwise. Note the integrand is much simpler. Once integration bounds have been refined by introducing indicators and performing linearisation, the integrals are given to SymPy's *integrate* function to finish the integration.

This concludes this section on symbolic inference. This algorithm is able to handle a significantly wider range of combinations of martingales and probabilistic programs, compared to previous methods used to synthesise martingales. For instance, it can handle programs using the uniform distribution, combined with a ReLU neural martingale.

4.3 Reachability Analysis

This section discusses the *ReachabilityAnalysis* component that is used when verifying the bounded differences condition for RepSMs. Recall that in the synthesis of RepSM the following logical formula is verified.

$$\forall x, x' \in S. \, (T(x, x') \wedge (x \in I \cap P)) \implies |\eta(x) - \eta(x')| \leq c$$

The goal is to ensure that the update statement can only change the value of the martingale by at most c. To do this, the predicate $T(x, x')$ constrains which states x' are reachable after a single iteration of the loop. To illustrate the construction of T, consider the following update statement.

$p \sim \mathsf{Bernoulli}(0.999)$
if $p == 1$:
    $q \sim \mathsf{Uniform}(0, 1)$
    $t = t + q$
else:
    $error = 1$

The first step is to add indices to the different variable assignments, so they can be distinguished.

$p_0 \sim \mathsf{Bernoulli}(0.999)$
if $p_0 == 1$:
    $q_0 \sim \mathsf{Uniform}(0, 1)$
    $t_1 = t_0 + q_0$
else:
    $error_1 = 1$
join $\{(t_1, t_0) \mapsto t_2, (error_0, error_1) \mapsto error_2\}$

Then the code is analysed in a recursive step-by-step fashion. First the variable p_0 is translated, to obtain the following formula.

$$(p_0 = 0) \vee (p_0 = 1)$$

Then if and else blocks are converted to the following formulae.

$$(q_0 \geq 0) \wedge (q_0 \leq 1) \wedge (t_1 = t_0 + q_0)$$
$$error_1 = 1$$

To combine these two formulae, first the join mapping is applied.

$$(q_0 \geq 0) \wedge (q_0 \leq 1) \wedge (t_1 = t_0 + q_0) \wedge (t_2 = t_1) \wedge (error_2 = error_0)$$
$$error_1 = 1 \wedge (t_2 = t_0) \wedge (error_2 = error_1)$$

Then these two formula are combined with a disjunction to produce a single SMT formula for the if statement.

$$((q_0 \geq 0) \wedge (q_0 \leq 1) \wedge (t_1 = t_0 + q_0) \wedge (t_2 = t_1) \wedge (error_2 = error_0)) \vee$$
$$(error_1 = 1 \wedge (t_2 = t_0) \wedge (error_2 = error_1))$$

Then the SMT formula for p_0 is combined with the SMT formula for the if statement, to produce an SMT formula for the entire block of code:

$$((p_0 = 0) \vee (p_0 = 1)) \wedge$$
$$(((q_0 \geq 0) \wedge (q_0 \leq 1) \wedge (t_1 = t_0 + q_0) \wedge (t_2 = t_1) \wedge (error_2 = error_0)) \vee$$
$$(error_1 = 1 \wedge (t_2 = t_0) \wedge (error_2 = error_1)))$$

This formula shows how the program state before the loop body $(t_0, error_0)$ relates to the program state after the loop body $(t_2, error_2)$. The procedure essentially encodes the program as an SMT formula, replacing the probabilistic semantics with nondeterminism.

Chapter 5

Evaluation

The previous chapters have described the development and implementation of new algorithms for synthesising ISMs and RepSMs. The goal of this chapter is to evaluate these algorithms, by running them on a series of benchmarks. In particular, we are interested in the following questions:

- Is our method able to verify quantitative safety for programs that are within the scope of existing tools?
- Is our method able to produce useful results on programs that are out-of-scope for existing tools?
- Does our method produce useful probability bounds in practice, and are those bounds better or worse than those produced by existing tools?
- Is the time taken by our method reasonable, and is it better or worse than the time taken by existing tools?

To compare our method against existing work, we run these benchmarks on an implementation of Farkas' lemma.

5.1 Case Studies

5.1.1 Creating Benchmarks

The benchmarks in this report have been created using two different patterns. The first is the *faulty* pattern. This begins by taking an AST single-loop program. An example of such a program is given below. There is a variable t initialised to 1, and that increments by 1 in each iteration, with probability 0.5, until $t = 10$ is reached.

$t = 1$
while $t < 10$:
    $q \sim \mathsf{Bernoulli}(0.5)$
    if $q == 1$:
        $t = t + 1$

To turn this program into a quantitative safety benchmark, a very small probability of failure is added to each iteration. This is shown in the program below. A Bernoulli random variable p is sampled, which determines if the program will fail. To indicate failure, the variable *error* is set to 1.

$t = 1$
$error = 0$
while $t < 10$:
    $assert(error \neq 1)$
    $p \sim \mathsf{Bernoulli}(0.999)$
    if $p == 1$:
        $q \sim \mathsf{Bernoulli}(0.5)$
        if $q == 1$:
            $t = t + 1$
    else:
        $error = 1$

The goal is then to bound the probability of $error = 1$ ever being reached. This transformation is applied to a range of single-loop programs, from existing literature on qualitative termination [Chakarov and Sankaranarayanan, 2013; Abate et al., 2021c]. Note that since the probability of an error occurring is so small, in these programs importance sampling is required for program-agnostic synthesis.

The second method of creating benchmarks is the *repulse* pattern. This pattern is already found in prior literature in ISMs and RepSMs [Chatterjee et al., 2016b, 2022]. In this pattern, the probabilistic program performs a random walk. An example is given below. The variable x is initialised to 10. In each iteration, it is either decremented by 2, or incremented by 1.

$x = 10$
while $x \geq 0$:
    $p \sim \mathsf{Bernoulli}(0.5)$
    if $p == 1$:
        $x = x - 2$
    else:
        $x = x + 1$

A certain region of the state space is designated as unsafe, e.g. $x \geq 100$. Then the goal is to bound the probability of reaching this unsafe region.

The difference between the two patterns, is that in the faulty pattern, it is possible to fail in every iteration. However, in the repulse pattern, the program moves closer/further to the unsafe region as the program progresses. It is not possible to move to the unsafe region in one time step from any state.

This difference affects which martingale structure is suitable to use for verification. The faulty and repulse patterns can both be verified by ISMs. However, only the repulse pattern is suitable to be verified by RepSMs. RepSMs for the faulty pattern will produce the trivial bound. The reason for this is that c, the maximum change in the value of the martingale in a single time step, plays a big role in the

bound generated by a RepSM. RepSMs for the faulty pattern will have a very large value of c since it is possible to fail in a single time step, from any state.

Having explained these two patterns, now some of the benchmarks will be discussed to explain how they benefit from our method.

5.1.2 Faulty Varying

An example of a program that can be bounded with Farkas' lemma, but that can be given a better bound using our method is the following program. In this program, t is initialised at 1. In each iteration, it is incremented by 1, with probability 0.5, until $t = 10$ is reached. In this program, the probability with which *error* is set to 1 varies, based on the value of t. If $t < 4$, then the probability is 0.99, otherwise it is 0.999.

$t = 1$
$error = 0$
while $t < 10$:
    $assert(error \neq 1)$
    if $t < 4$:
        $p \sim \mathsf{Bernoulli}(0.99)$
    else:
        $p \sim \mathsf{Bernoulli}(0.999)$
    if $p == 1$:
        $q \sim \mathsf{Bernoulli}(0.5)$
        if $q == 1$:
            $t = t + 1$
    else:
        $error = 1$

Intuitively, the probability of failure is much lower once $t = 5$ is reached. However, a linear ISM cannot account for this, and therefore Farkas' lemma produces the same bound as if the probability of failure is always 0.99. By contrast, a neural template with 2 ReLU components, can encode a piecewise branching on these two cases. This gives a tighter probability bound than can be obtained with a linear ISM. This is illustrated in Figure 5.1 which compares a linear ISM from Farkas' lemma, and our neural ISM, when $error = 0$.

5.1.3 Faulty Regions

An example of a program that cannot be bounded with Farkas' lemma, is shown below. In this program, t is initialised to 10. In each iteration, t is either incremented or decremented, while t remains in the region $[1, 19]$. In the region $[1, 9]$, t is more likely to be decremented towards 0. In the region $[10, 19]$, t is more likely to be incremented towards 20. As this is an instance of the faulty pattern, there is a small probability in each iteration, that the program fails.

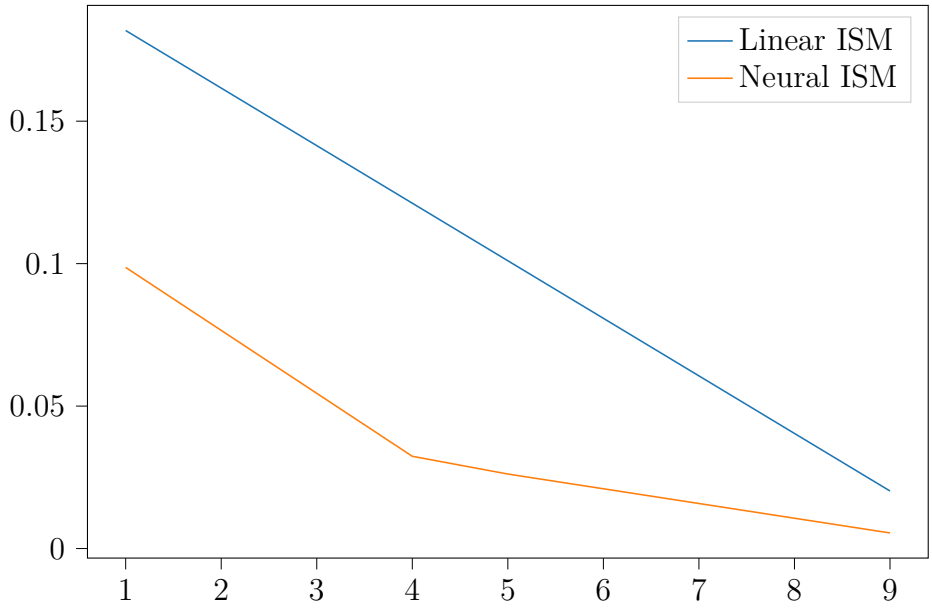

Figure 5.1: ISMs for `faulty_varying` where $error = 0$

$t = 10$
$error = 0$
while $t > 0$ **and** $t < 20$:
    $p \sim \mathsf{Bernoulli}(0.999)$
    if $p == 1$:
        $q \sim \mathsf{Bernoulli}(0.9)$
        if $t < 10$:
            if $q == 1$:
                $t = t - 1$
            else:
                $t = t + 1$
        else:
            if $q == 1$:
                $t = t + 1$
            else:
                $t = t - 1$
    else:
        $error = 1$

A linear ISM doesn't exist for this program. Intuitively, two different linear ISMs are required. One for the region $t < 10$, and another for the region $t \geq 10$. A neural ISM effectively accomplishes this, as it can encode piecewise branching on the condition $t < 10$.

The ISM when $error = 0$ is shown in Figure 5.2. Notice that this graph is concave. It would be not be possible to encode an ISM for this program using the SOR template; a two-layer ReLU network is necessary. Notably, this is first instance of multi-layer neural networks being used in the analysis of probabilistic programs.

Previous approaches have focussed on single-layer networks or regression trees.

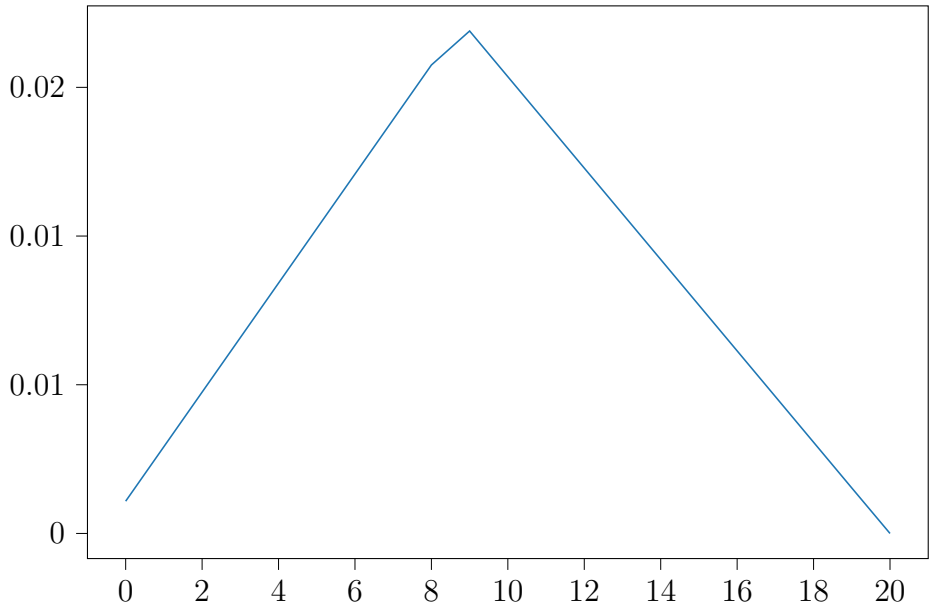

Figure 5.2: The neural ISM for `faulty_regions` where $error = 0$

5.1.4 Persist 2D

An example of a program that cannot be solved with Farkas' lemma is given below.

$x = 10$
$y = 10$
while $x > 0$ **and** $y > 0$:
    $p \sim \mathsf{Bernoulli}(0.5)$
    $q \sim \mathsf{Bernoulli}(0.5)$
    if $p == 1$:
        if $x \leq 100$:
            if $q == 1$:
                $x = x - 2$
            else:
                $x = x + 1$
        else:
            $x = x + 2$
    else:
        if $y \leq 100$:
            if $q == 1$:
                $y = y - 2$
            else:
                $y = y + 1$
        else:
            $y = y + 2$

The program state performs a 2D random walk, starting at $(10, 10)$. In each iteration the program chooses a co-ordinate to change, with equal probability. There are then two cases:

- If the co-ordinate is greater than 100, it is increased by 2.
- Otherwise, it is either decreased by 2, or increased by 1, with equal probability.

The program terminates, once one co-ordinate is below 0, and fails when both co-ordinates are above 100. It is not possible to construct a linear RepSM. This is because, there are safe states, where one co-ordinate is above 100, and another co-ordinate is below 100. The RepSM must decrease by ϵ in expectation, but one of the co-ordinates, will always increase by 2 when it is chosen. However, a neural RepSM can branch on the cases where a co-ordinate is above 100, and ignore that co-ordinate.

The neural RepSM for this program, synthesised by our algorithm, is shown in Figure 5.3. Note that increasing either the x co-ordinate or the y co-ordinate will increase the value of $\eta(x, y)$ until 100 is reached, and then the function will plateau (in that direction).

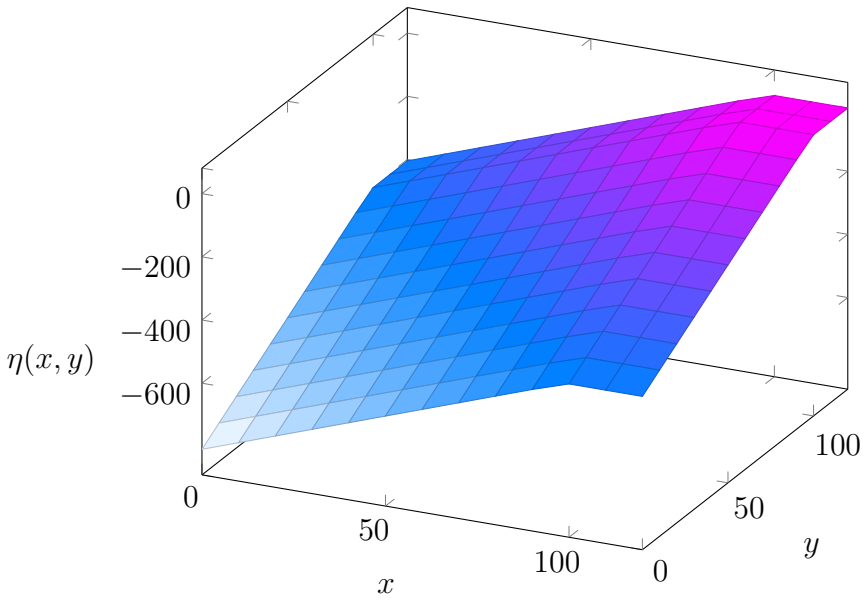

Figure 5.3: The neural ISM for `persist_2d` example

5.2 Results

This section shows the results of the benchmarks, and makes some observations based on the data. These observations will be built upon in the next section, to give a more thorough evaluation of our method.

	Neural ISM		Farkas' Lemma
	Program-Agnostic	Program-Aware	
`faulty_marbles`	≤ 0.0468	≤ 0.0373	-
`faulty_unreliable`	≤ 0.0508	≤ 0.0393	-
`faulty_probfact`	≤ 0.0402	≤ 0.0338	-
`repulse_2d`	≤ 0.1077	≤ 0.0991	-
`persist_2d`	≤ 0.1230	≤ 0.1085	-
`faulty_regions`	≤ 0.0447	≤ 0.0155	-
`faulty_varying`	≤ 0.0918	≤ 0.0869	≤ 0.1819
`faulty_concave`	≤ 0.1321	≤ 0.1290	≤ 0.1819
`fixed_loop`	≤ 0.0095	≤ 0.0094	≤ 0.0091
`faulty_loop`	≤ 0.0190	≤ 0.0185	≤ 0.0181
`faulty_rare`	≤ 0.0023	≤ 0.0021	≤ 0.0019
`faulty_uniform`	≤ 0.0203	-	≤ 0.0181
`faulty_gaussian`	≤ 0.0505	≤ 0.0440	≤ 0.0181
`faulty_easy1`	≤ 0.1067	≤ 0.0844	≤ 0.0801
`faulty_ndecr`	≤ 0.0819	≤ 0.0604	≤ 0.0561
`faulty_walk`	≤ 0.0152	-	≤ 0.0121
`repulse`	≤ 0.1080	≤ 0.1079	≤ 0.0991
`repulse_uniform`	≤ 0.1030	-	≤ 0.0991

Figure 5.4: Table comparing the bounds achieved when synthesising ISMs

Figure 5.4 shows the probability bounds obtained by synthesising ISMs for each of the benchmarks. Benchmarks are split into three sections. The first section is where both methods produce comparable results. The second section is where this method produces significantly better results. The final section is where only this method can produce results at all.

These values have been rounded up to four decimal places. There are columns for program-agnostic synthesis, program-aware synthesis and Farkas' lemma. Almost all examples used the general ReLU neural network structure. The `faulty_gaussian` example used a SOS neural template, owing to an inability to perform symbolic inference of a ReLU martingale with a Gaussian distribution.

Note that, there are some empty values in the table, indicated by dashes. In the case of program-aware synthesis, this is because the post-expectation cannot be converted into a smooth expression to be embedded in the loss function. This meant the algorithm could not be applied. In the case of Farkas' Lemma, this is because the algorithm failed to produce bounds for the program.

The first section of the benchmarks demonstrates that our method is capable of producing useful results on programs that are out-of-scope for previous methods. As expected, the program-aware algorithm produced better results, since it can compute an exact value for the post-expectation. Interestingly, the difference between the program-agnostic and program-aware methods is relatively small, around a single

percentage point. An exception is `faulty_regions`, where the difference is three percentage points. An explanation for this anomalous result it that this required a two-layer neural network.

The second section of the benchmarks demonstrates that there are examples, which Farkas' lemma solves, but where our method can yield significantly better probability bounds. For these two examples, the difference between the program-aware and program-agnostic methods are slight.

The third section of the table consists of benchmarks, where our method and Farkas' lemma produced comparable results. The worst result is `faulty_gaussian`, which is likely owing to its use of a SOS template. Apart from this example, the difference between the program-aware synthesis and Farkas' lemma is very small. As expected, program-agnostic synthesis produces worse results. The size of the difference varies in this section, from a negligible difference to about two percentage points in `faulty_ndecr` and `faulty_easy1`.

	Neural ISM				Farkas' Lemma
	Program-Agnostic		Program-Aware		
	Learn	Verify	Learn	Verify	
`faulty_marbles`	25.45	56.40	36.33	85.67	-
`faulty_unreliable`	25.81	62.29	37.27	79.37	-
`faulty_probfact`	10.09	15.82	10.68	9.67	-
`repulse_2d`	7.45	5.62	11.63	5.61	-
`persist_2d`	18.85	19.01	17.33	13.36	-
`faulty_regions`	16.08	11.66	26.01	12.77	-
`faulty_varying`	5.68	5.89	10.50	7.48	0.355
`faulty_concave`	19.73	18.36	20.84	16.42	0.393
`fixed_loop`	5.04	4.76	5.42	4.54	0.150
`faulty_loop`	6.05	4.69	5.48	4.87	0.164
`faulty_rare`	6.04	6.29	5.60	6.67	0.266
`faulty_uniform`	4.04	1.95	-	-	0.340
`faulty_gaussian`	2.34	4.57	3.09	5.38	1.365
`faulty_easy1`	34.12	14.03	7.01	16.45	0.312
`faulty_ndecr`	13.25	9.57	13.05	8.72	0.329
`faulty_walk`	3.75	4.70	-	-	0.314
`repulse`	6.46	8.22	3.79	7.46	0.186
`repulse_uniform`	4.39	4.31	-	-	0.191

Figure 5.5: Table comparing the time taken when synthesising ISMs

Figure 5.5 shows the time taken for the different algorithms to synthesise ISMs. Farkas' lemma is unsurprisingly extremely fast, while our method is significantly slower.

For the neural algorithms, the time taken is split into the time taken by the learner and the time taken by the verifier. There is a correlation between the time

taken by the learner and the verifier. This is unsurprising since each additional CEGIS iteration will increase the time spent in both learning and verification.

More complex programs tend to require more CEGIS iterations, and therefore more time, and this is shown by `faulty_marbles` and `faulty_unreliable`.

It is harder to compare the time taken by the program-agnostic and program-aware algorithms. The time taken by the program-agnostic algorithms depends on the number of samples. The time taken by the program-aware algorithms depend on the complexity of the symbolic expression for the post-expectation. In some cases, the program-agnostic algorithm is faster. In other cases, the program-aware algorithm is faster.

Most benchmarks used 200 samples. `faulty_regions` used 1,000 samples, and `faulty_easy1` and `faulty_probfact` used 10,000 samples. Time was not spent attempting to find an optimal number of samples. If program-agnostic synthesis failed, the number of samples was simply increased by an order-of-magnitude until successful results were achieved.

Figure 5.6 shows the bounds obtained for RepSMs. The results lead to conclusions that are similar to those for ISMs. First, our method can synthesise RepSMs, that are out-of-scope for previous methods. Moreover, both the program-agnostic and program-aware methods perform well, with the difference in bounds produced between the methods varying. Also, note that these benchmarks all appear in the table for ISMs as well. The results show that RepSMs allow tighter bounds to be synthesised.

	Neural RepSM		Farkas' Lemma
	Sampling	No Sampling	No Sampling
`repulse_2d`	≤ 0.0399	≤ 0.0168	-
`persist_2d`	≤ 0.0175	≤ 0.0159	-
`repulse`	≤ 0.0216	≤ 0.0179	≤ 0.0138

Figure 5.6: Table comparing the bounds achieved when synthesising RepSMs

5.3 Discussion

This section builds upon the empirical results in the previous section to give an overall discussion of the strengths and weakness of the method presented in this report.

5.3.1 Comparison with previous work

A key strength of the method is that it can handle a wider variety of programs than previous methods. This is clearly highlighted in the results from benchmarks. Conditional statements, disjunctions and conjunctions are essential to programming, and an effective method to verify probabilistic programs should be able to handle them. Our method is capable of doing this, by using ReLU neural networks, which correspond to piecewise linear functions. This is shown in `faulty_marbles` and `faulty_regions` in a concrete and compelling way. Furthermore, this method is even able to produce better bounds than Farkas' lemma, in cases where both methods can be applied, by exploiting a more general template.

A weakness of the method that has been highlighted is that our method is significantly slower than Farkas' lemma. Farkas' lemma relies on linear programming, and consequently benefits from the highly performant optimisation procedures that have been developed for it. By contrast, our method attempts to solve a non-convex and non-linear optimisation with gradient descent, and performs verification through SMT solving. Having said all of this, all benchmarks were solved within a reasonable amount of time, on standard hardware. Furthermore, the time taken is in line with other machine learning methods for probabilistic programs [Abate et al., 2021c; Bao et al., 2022].

One aspect to discuss is the quality of the bounds produced by our method. While the bounds obtained by our method are not perfect, reasonable bounds were obtained for all benchmarks. This demonstrates that neural methods are capable of proving quantitative properties.

One weakness is that our symbolic inference algorithm cannot handle a Gaussian distribution with a ReLU network. However, the fact an SOS template could be used, highlights that the method can be applied to neural templates, that use different activation functions.

Finally, there is significant potential to build upon this method to tackle the verification of more challenging probabilistic programs.

5.3.2 Comparison between program-agnostic and program-aware algorithms

The program-aware algorithm provided better bounds than program-agnostic algorithm. While all program-agnostic bounds were reasonable, in a few examples, the program-agnostic bound was multiple percentage points higher than the corresponding program-aware bound.

An advantage of the program-agnostic algorithm is that by removing the requirement to embed the post-expectation directly in the loss function, this makes

it far more realistic to apply this method to more complex examples. Indeed, this was demonstrated to an extent, through examples like `faulty_uniform` where the post-expectation could not be transformed into a smooth function to be used in the loss function.

One weakness of the program-agnostic algorithm is that rare events had to manually handled, by specifying an alternative distribution for certain random variables, so importance sampling could be used. On the other hand, this does demonstrate that such rare events are not an obstacle to program-agnostic synthesis. Indeed, this is the first machine learning approach to probabilistic programs that uses importance sampling.

A weakness of the program-aware RepSM method is that it requires a symbolic expression for the variable c to be given as input. While in these examples, it was easy to find an expression for c, in more complex examples this is an unrealistic requirement. This weakness is rectified in the program-agnostic method, which uses execution traces to estimate the variable c. This further highlights how the program-agnostic analysis is more appropriate for complex programs.

A point of interest is the difference in time taken between the program-agnostic and program-aware algorithms. There were examples where the program-agnostic algorithm was noticeably faster, such as `faulty_marbles` and `faulty_regions`. However, there were also examples such as `faulty_easy1`, where the program-agnostic examples took longer. This is closely related to the number of samples needed to synthesise a martingale. There is scope for further investigation, regarding the number of samples needed.

Chapter 6

Related Work

Chapter 2 discusses the prior literature that this project builds upon. By contrast, this chapter discusses related work more broadly.

6.1 Martingales

Previous work to synthesise RSMs, RepSMs and ISMs have been discussed in depth in Chapter 2 [Chakarov and Sankaranarayanan, 2013; Chatterjee et al., 2016a,b, 2022]. The synthesis of lexicographic RSMs, a variation of RSMs that can be used for nested loops, has been performed with linear programming [Agrawal et al., 2017]. Algebraic recurrence techniques have been used for the synthesis of RSMs [Moosbrugger et al., 2021], although this method has similar limitations to the Farkas' lemma approach.

Previous work has used martingales and similar structures to bound the expected value of a variable in program state, when a probabilistic program terminates. This variable can be interpreted as a reward or a cost [Ngo et al., 2018; Wang et al., 2019]. This work has been extended to higher moments [Wang et al., 2021]. These techniques uses linear and semidefinite programming for synthesis, unlike this work which uses learning in a CEGIS architecture for synthesis.

6.2 Probabilistic Model Checking

A well-developed approach to reasoning about probabilistic systems is probabilistic model checking [Kwiatkowska et al., 2010; Katoen, 2016]. A well established probabilistic model checking tool is PRISM [Kwiatkowska et al., 2011]. This approach encodes properties using probabilistic extensions to temporal logics such as LTL and CTL. Verification occurs through a combination of graph analysis and numerical approximation. They can be applied to systems modelled as discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markov decision processes (MDPs). Probabilistic model checking is usually applied to finite state spaces, whereas this work looks at programs with infinite state spaces, and programs that combine discrete and continuous variables.

6.3 Pre-Expectation Calculus

The weakest pre-expectation calculus is a formalism for reasoning about probabilistic programs [Kozen, 1985; Morgan et al., 1996; McIver and Morgan, 2004]. It is an extension of predicate transformer semantics for classical programs [Dijkstra, 1975]. The calculus relates expressions to their expected values after a program runs. The pre-expectation calculus is connected to martingales, and this is studied in Hark et al. [2020]. In order to reason about the weakest pre-expectations of loops, invariants are used.

Recent work concentrates on automatically finding weakest pre-expectations. Gretz et al. [2013] present a method where finding an invariant from a template is reduced to first-order logic constraints, that are given to a constraint solver to handle. Chen et al. [2015] use counterexample refinement and Lagrange interpolation to find polynomial invariants. Wang et al. [2018] introduce a method for the static analysis of probabilistic programs, based on algebraic structures. This method was applied to generate linear invariants.

Most recently, a CEGIS-like method has been used to learn invariants [Bao et al., 2022]. This work is similar to the work presented here in that it is a data-driven approach using learning, uses a computer algebra system, and uses counterexamples to explore the state space. One difference is that the work uses regression trees as models, as opposed to neural networks. A limitation of this work is that it only supports discrete distributions, whereas this work looks at probabilistic programs using continuous distributions.

6.4 Dynamical Systems

CEGIS-based methods have been used to verify dynamical systems. Dynamical systems are continuous-time systems, specified with differential equations. So far, these methods have been applied to deterministic dynamical systems. CEGIS has been used to verify stability for such systems by synthesising Lyapunov functions with polynomial and neural templates [Ahmed et al., 2020; Abate et al., 2021a,b]. This is similar to proving qualitative termination, and RSMs can be seen as stochastic analogues of Lyapunov functions. Further, safety has been verified with barrier certificates [Jagtap et al., 2019; Peruffo et al., 2020].

Chapter 7

Conclusion

This report began with the objective of integrating learning and verification to prove quantitative safety properties. This chapter draws the report to a close, putting the work achieved in the wider context. It begins with a summary of the major contributions of the project, and an explanation of what distinguishes this work from what has come before. Finally, some directions for future work are sketched out.

7.1 Contributions

We develop the first machine learning method for the synthesis of ISMs and RepSMs to solve the quantitative safety problem. This method uses ReLU neural networks as templates to synthesise martingales. These martingales are learnt by optimising a loss function using gradient descent over a training dataset of execution traces, as part of a CEGIS architecture. The loss function balances minimising constraint violation with minimising the bound.

Our method is more general than the state-of-the-art. Previous methods are restricted to linear functions, while our approach synthesises PWL functions, which are more expressive. This leads to a more general algorithm for synthesis, that can be applied to a wider class of probabilistic programs. Our claim of generality is supported by experimental results. Our method is capable of verifying probabilistic programs, that cannot be handled by previous work. In addition to this, it can even prove tighter bounds for programs, that can be handled by previous work.

In theory, our method is more scalable than previous approaches. This is because our method can be used in a program-agnostic manner. While the structure of the program is essential in previous methods to compute the post-expectation, our method can learn in an entirely data-driven process, using execution traces to estimate the post-expectation. A program-agnostic learning framework has the potential to make the verification of complex probabilistic programs more tractable.

Notably, this is the first machine learning method for the analysis of probabilistic programs to use multi-layer networks. Additionally, it uses a symbolic inference algorithm allowing it to handle a wider variety of probabilistic programs, compared to previous methods.

It is worth noting some limitations of the method. There are no theoretical

guarantees that our method will succeed in finding a bound within a given number of CEGIS iterations. Furthermore, our method is more computationally expensive than methods based on linear programming. Additionally, program-agnostic synthesis is not as strong as program-aware synthesis, in terms of the quality of the bounds produced. Nonetheless, we find that in practice our method produces useful results over a wide range of examples.

7.2 Future Work

7.2.1 Reward Properties

One direction for future work is looking at other kinds of properties about probabilistic programs. One type of property would be reward (or cost) properties, i.e. the expected value of a variable when a program terminates. As discussed, this has been considered before in the case of linear or polynomial templates. It would be unsurprising if there are simple programs, that cannot be handled by those templates, but that can be handled by PWL templates, as considered here.

7.2.2 Nondeterminism

Another area for future work is considering probabilistic programs with nondeterminism. This is where there is branching which is determined by a scheduler or a policy that is external to the program. Then a martingale might bound the maximum or the minimum probability of reaching an unsafe state.

One approach to this is to compute (or approximate) the post-expectation under every policy, and then use the policy that maximises or minimises constraint violation when computing the loss for each data point. This could be done with smooth maximum function to improve learning.

An extension to this is policy synthesis which is to try to compute a policy that minimises the probability of reaching an unsafe state. It is also natural to consider a combination of reward properties and nondeterminism. Then one could consider the policy that maximises reward.

7.2.3 Multiple Initial States

One area for future work is considering multiple initial states. There are two different semantics for this. Initial states could be determined probabilistically or nondeterministically. In the probabilistic case, one approach would be to sample initial states and then compute the average bound for the initial state. This could then be incorporated into the loss function. In the nondeterministic case, one could take the minimum or maximum bound in the loss function, assuming there are a finite number of initial states.

7.2.4 Model Expressiveness

A possible area for future work is exploring more expressive templates. This project uses templates that can construct any PWL function. One extension might be adding monomials e.g. x^2 or xy^2 as features to a neural network. Then a neural network could learn a piecewise polynomial function.

Additionally, when it was possible to use RepSMs, they were able to produce better bounds than ISMs. RepSMs use the exponential function to produce a bound, and this suggests that an ISM template which has exponential-like behaviour, could prove tighter bounds for these programs. A possible activation function might be the sigmoid function. An SMT solver, that can handle transcendental functions, such as dReal would likely have to be used in verification [Gao et al., 2013].

Bibliography

Alessandro Abate, Daniele Ahmed, Alec Edwards, Mirco Giacobbe, and Andrea Peruffo. FOSSIL: A software tool for the formal synthesis of lyapunov functions and barrier certificates using neural networks. In *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*, HSCC '21, pages 1–11, New York, NY, USA, May 2021a. Association for Computing Machinery. ISBN 978-1-4503-8339-4. doi: 10.1145/3447928.3456646.

Alessandro Abate, Daniele Ahmed, Mirco Giacobbe, and Andrea Peruffo. Formal Synthesis of Lyapunov Neural Networks. *IEEE Control Systems Letters*, 5(3): 773–778, July 2021b. ISSN 2475-1456. doi: 10.1109/LCSYS.2020.3005328.

Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. Learning Probabilistic Termination Proofs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 3–26, Cham, 2021c. Springer International Publishing. ISBN 978-3-030-81688-9. doi: 10.1007/978-3-030-81688-9_1.

Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. Lexicographic ranking supermartingales: An efficient approach to termination of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 2(POPL):34:1–34:32, December 2017. doi: 10.1145/3158122.

Daniele Ahmed, Andrea Peruffo, and Alessandro Abate. Automated and Sound Synthesis of Lyapunov Functions with SMT Solvers. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 97–114, Cham, 2020. Springer International Publishing. ISBN 978-3-030-45190-5. doi: 10.1007/978-3-030-45190-5_6.

Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding Deep Neural Networks with Rectified Linear Units. November 2016.

Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. Data-Driven Invariant Learning for Probabilistic Programs, June 2022.

Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer, London, 2012. ISBN 978-1-4471-4128-0 978-1-4471-4129-7. doi: 10.1007/978-1-4471-4129-7.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Composable transformations of Python+NumPy programs, 2018.

Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic Program Analysis with Martingales. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Natasha Sharygina, and Helmut Veith, editors, *Computer Aided Verification*, volume 8044, pages 511–526. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-39798-1 978-3-642-39799-8. doi: 10.1007/978-3-642-39799-8_34.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 3–22, Cham, 2016a. Springer International Publishing. ISBN 978-3-319-41528-4. doi: 10.1007/978-3-319-41528-4_1.

Krishnendu Chatterjee, Petr Novotný, and Đorđe Žikelić. Stochastic Invariants for Probabilistic Termination. *arXiv:1611.01063 [cs]*, November 2016b.

Krishnendu Chatterjee, Amir Goharshady, Tobias Meggendorfer, and Đor\dje Žikelić. Sound and Complete Certificates for Quantitative Termination Analysis of Probabilistic Programs. In *CAV 2022 – 34th International Conference on Computer Aided Verification*, Haifa, Israel, August 2022.

Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation, July 2015.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2.

Edsger W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. *ACM SIGPLAN Notices*, 10(6):2–2.13, April 1975. ISSN 0362-1340. doi: 10.1145/390016.808417.

Luis María Ferrer Fioriti and Holger Hermanns. Probabilistic Termination: Soundness, Completeness, and Compositionality. *ACM SIGPLAN Notices*, 50(1):489–501, January 2015. ISSN 0362-1340. doi: 10.1145/2775051.2677001.

Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT Solver for Nonlinear Theories over the Reals. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, Lecture Notes in Computer Science, pages

208–214, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-38574-2. doi: 10.1007/978-3-642-38574-2_14.

Timon Gehr, Sasa Misailovic, and Martin Vechev. PSI: Exact Symbolic Inference for Probabilistic Programs. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, volume 9779, pages 62–83. Springer International Publishing, Cham, 2016. ISBN 978-3-319-41527-7 978-3-319-41528-4. doi: 10.1007/978-3-319-41528-4_4.

Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Prinsys—On a Quest for Probabilistic Loop Invariants. In Kaustubh Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems*, Lecture Notes in Computer Science, pages 193–208, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-40196-1. doi: 10.1007/978-3-642-40196-1_17.

Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. Aiming low is harder: Induction for lower bounds in probabilistic program verification. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, January 2020. ISSN 2475-1421. doi: 10.1145/3371105.

Matteo Hessel, David Budden, Fabio Viola, Mihaela Rosca, Eren Sezener, and Tom Hennigan. Optax: Composable gradient transformation and optimisation, in JAX!, 2020.

Pushpak Jagtap, Sadegh Soudjani, and Majid Zamani. *Formal Synthesis of Stochastic Systems via Control Barrier Certificates*. May 2019.

Joost-Pieter Katoen. The Probabilistic Model Checking Landscape. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 31–45, New York NY USA, July 2016. ACM. ISBN 978-1-4503-4391-6. doi: 10.1145/2933575.2934574.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017.

Dexter Kozen. A probabilistic PDL. *Journal of Computer and System Sciences*, 30 (2):162–178, April 1985. ISSN 0022-0000. doi: 10.1016/0022-0000(85)90012-1.

Daniel Kroening and Ofer Strichman. *Decision Procedures*. Texts in Theoretical Computer Science. Springer, Berlin, Heidelberg, 2008. ISBN 978-3-540-74104-6 978-3-540-74105-3. doi: 10.1007/978-3-540-74105-3.

Marta Kwiatkowska, Gethin Norman, and David Parker. Advances and challenges of probabilistic model checking. In *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1691–1698, Monticello, IL, USA, September 2010. IEEE. ISBN 978-1-4244-8215-3. doi: 10.1109/ALLERTON.2010.5707120.

Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In Ganesh Gopalakrishnan and Shaz Qadeer,

editors, *Computer Aided Verification*, volume 6806, pages 585–591. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-22109-5 978-3-642-22110-1. doi: 10.1007/978-3-642-22110-1_47.

Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag, 2004. ISBN 978-0-387-40115-7.

Aaron Meurer, Christopher Smith, Mateusz Paprocki, Ondřej Čertík, Sergey Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian Granger, Richard Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, and Anthony Scopatz. SymPy: Symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017. doi: 10.7717/peerj-cs.103.

Sean Meyn and Richard L. Tweedie. *Markov Chains and Stochastic Stability*. Cambridge Mathematical Library. Cambridge University Press, Cambridge, second edition, 2009. ISBN 978-0-521-73182-9. doi: 10.1017/CBO9780511626630.

Stuart Mitchell, Stuart Mitchell Consulting, and Iain Dunning. PuLP: A Linear Programming Toolkit for Python, 2011.

Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. The Probabilistic Termination Tool Amber, July 2021.

Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996. ISSN 0164-0925. doi: 10.1145/229542.229547.

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 496–512, Philadelphia PA USA, June 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192394.

Andrea Peruffo, Daniele Ahmed, and Alessandro Abate. Automated and Formal Synthesis of Neural Barrier Certificates for Dynamical Models. *arXiv:2007.03251 [cs, eess]*, October 2020.

Jeffrey Seth Rosenthal. *A First Look at Rigorous Probability Theory*. World Scientific, 2006. ISBN 978-981-270-370-5.

Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. Ranking and Repulsing Supermartingales for Reachability in Randomized Programs. *ACM Transactions on Programming Languages and Systems*, 43(2):5:1–5:46, June 2021. ISSN 0164-0925. doi: 10.1145/3450967.

Di Wang, Jan Hoffmann, and Thomas Reps. PMAF: An algebraic framework for static analysis of probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages

513–528, Philadelphia PA USA, June 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192408.

Di Wang, Jan Hoffmann, and Thomas Reps. Central moment analysis for cost accumulators in probabilistic programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 559–573, Virtual Canada, June 2021. ACM. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454062.

Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. Cost analysis of nondeterministic probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 204–220, New York, NY, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314581.