Expressing and Verifying Probabilistic Assertions

Author: Dustin Hütter
Advisor: Prof. Dr. Ir. Joost-Pieter Katoen

Winter term 2014/2015
RWTH Aachen (Chair for Software Modeling and Verification)
Seminar work for seminar *Probabilistic Programs*

Abstract

While there exist advanced tools for verifying traditional assertions, tools for the verification of probabilistic assertions are still not as sophisticated as desired. Since problems occurring in machine learning, approximate computing and data obfuscation, deal with probabilistic calculations, there is a necessity to develop such tools enabling programmers to check whether certain assumptions in their code hold. The following report presents a verification template for the verification of probabilistic assertions and its implementation in a tool called *MAYHAP*. *MAYHAP* models slices of probabilistic programs as Bayesian networks and verifies probabilistic assertions on the latter either by direct verification of the optimized Bayesian network or by sampling. As we will see, *MAYHAP* outperforms competing tools by several orders of magnitude.

Chapter 1

Introduction

This seminar report presents a verification scheme for probabilistic assertions. The scheme is presented in [SPM+14] and its soundness proof is given in [SPM+]. Some of the illustrations and formalisms occurring in this report can be found in these sources. Before we introduce the verification scheme, we will have a look at required terminology and the theoretical foundations that the scheme bases on. We will then proceed with optimizations for the scheme and judge the applicability by considering its benchmark results.

1.1 (Probabilistic) Assertions

Assertions are statements evaluating to a Boolean and are a powerful concept of formal verification to express conditions that are desired to hold in a code snippet. Model checking these enables us to either verify that the assertion is always satisfied or not in which case a possible bug is revealed. We take the following lines of code into account in which we want to ensure that a value that we divide by is unequal to zero:

```
1 float a = 0, b = 0;
2 a = some_calculation_1();
3 b = some_calculation_2();
4 assert b != 0:
5 return a/b;
```

Figure 1.1: Sample code

The assertion in Figure 1.1 states that b has to be different from zero on every execution of this program. While applying these assertions is useful for a lot of problem classes especially deterministic ones, it is not for programs exhibiting probabilistic behavior. These occur for example in approximate computing, machine learning and data obfuscation. Instances

of domains involving probabilistic computation inherently do not provide the exact same output on every execution. Therefore, applying classical assertions and the corresponding verification mechanisms to these would be inappropriate. Hence, we will extend classical assertions to probabilistic assertions of the form *passert e p c*. The latter provides a notion to verify whether a Boolean condition e over the variables of the program holds with probability p and confidence c. The following example illustrates that:

```
1  float[] noisy_database_entry(float[] to_be_obscured)
2  {
3    // Create noise
4    obsc_0 = random.Gauss(0,1);
5    obsc_1 = random.Gauss(0,1);
6    float[] res = to_be_obscured;
7    // Add noise
8    res[0] = to_be_obsc[0]+obsc_0;
9    res[1] = to_be_obsc[1]+obsc_1;
10   passert eucl_dist(res,to_be_obsc) ≤ 100 0.85 0.9
11   return res;
12 }
```

Figure 1.2: Sample code

Figure 1.2 shows a procedure that takes a database entry and obfuscates the first two parameters for privacy reasons, e.g. because they represent the salary and the height of a person. Before we return *res* we probabilistically assert that the Euclidean distance between the original and the obfuscated value is less or equal than 100 ensuring that the latter is still useful for further proceeding, e.g. for statistical purposes. In contrast to classical assertions we do not require the satisfaction of the assertion for every execution but for 85% of the cases with a confidence of 0.9. The confidence is a measure for the reliability of the verification result. In the chapter about the sampling approach we will formalize the notion of confidence. The classical assertion

$$\text{assert eucl\_dist(res,to\_be\_obsc)} \leq 100$$

would not be satisfied in every execution since the noise could be too big. But satisfaction in every execution is actually not required because our data is not distorted too much if the probabilistic assertion is satisfied sufficiently often. Hence, probabilistic assertions are appropriate for these kind of queries.

1.2 Probabilistic Programs

In order to have a formal fundament for the notion of a probabilistic program, we introduce the grammar shown in Figure 1.3 for a simple prob-

abilistic language. This language, called *PROBCORE*, is used to verify the correctness of the presented method and provides an intuitive way of modeling probabilistic behavior. Another way of modeling certain classes of probabilistic programs is given by *Markov chains*. The interested reader may consult [SPH84] to get familiar with this.

$$
\begin{aligned}
P &\equiv\ S \text{ ;; passert } C \\
C &\equiv\ E < E \ \mid\ E = E \ \mid\ C \wedge C \ \mid\ C \vee C \ \mid\ \neg C \\
E &\equiv\ E + E \ \mid\ E * E \ \mid\ E \div E \mid\ R \ \mid\ V \\
S &\equiv\ V := E \mid\ V \leftarrow D \mid\ S; S \mid\ \text{skip} \mid\ \text{if } C\ S\ S \mid\ \text{while } C\ S \\
R &\in \mathbb{R}, V \in \text{Variables}, D \in \text{Distributions}
\end{aligned}
$$

Figure 1.3: PROBCORE

PROBCORE is a plain imperative language exhibiting conditionals, loops, assignments and other known features of non-probabilistic languages extended by the feature to assign variables with random draws from a probability distribution denoted $V \leftarrow D$. Furthermore, one can specify assertions. Using this grammar, a program P can intuitively be constructed. The parameters probability p and the confidence c of the *passert* are not given in this grammar because *PROBCORE* is used to deduce the satisfaction of the *passert* for single program runs. Either by direct verification or a sampling approach the user-defined p and c are involved in the verification.

Figure 1.4 shows an example *PROBCORE* program with input values a and b in which v_1 and v_2 are assigned with the results of deterministic procedures and are added. The value of v_3 is from the uniform distribution which ranges from 0 to 2. After further operations v_5 is assigned -1 if the value of v_3 is smaller than 1 respectively assigned 1 otherwise. Then we probabilistically assert that v_5 should be at least 0.

```
v1 := detProc(a);
v2 := detProc(b);
v2 := v2 + v1;
v3 ← Unif(0,2);
v4 := v3 − v2;
v5 := 0;
if v3 < 1   v5 := −1   v5 := 1;;
passert v5 ≥ 0
```

Figure 1.4: An example PROBCORE program

While this example seems artificial, it is appropriate to illustrate the verification scheme that we will see later.

1.3 Concrete Semantics

After having established the syntax of a *PROBCORE* program, we now have a look at a (concrete) semantics for this language taking random draws from distributions when probabilistic calculation occurs. The use of the term *concrete* gets evident when we introduce the main verification scheme which, in contrast, is symbolic. Firstly, we introduce the notions of *big-step* [Big] and *small-step semantics* [Sma] which are especially used for our purposes to formalize the concrete and the symbolic semantics.

1.3.1 Big-Step Semantics

Big-step semantics of programming languages provide the opportunity to interpret the syntactic constructs of a programming language in a suitable domain. Such a semantics is given by a system of inference rules. Big-step semantics contain steps of the form $E \Downarrow R$ meaning that the evaluation of E provides R, e.g. the evaluation of an arithmetic operation E yields the real number R.

1.3.2 Small-Step Semantics

While big-step semantics evaluate constructs of programming languages, small-step semantics model computation steps as transitions from one configuration say C_1 to the configuration say C_2. Configurations are typically tuples containing variable valuations and the program fragment that is to be evaluated. We denote such a transition by $C_1 \rightarrow C_2$. Besides the instructions that are processed, a configuration of the small-step parts of our concrete semantics includes a sequence of draws Σ for the generation of random samples and a heap H keeping track of the variable valuations. Σ and H are introduced below more precisely. Small-step semantics are also defined by a system of inference rules. We use $C_1 \rightarrow^* C_n$ for the reflexive and transitive closure of the transition relation, meaning that the configuration C_n can be reached from C_1 in a finite and non-negative number of steps.

In order to model variable valuations, we use a heap H for which $H(v)$ is the value of variable v. Furthermore, to model the generation of probabilistic values, we use a draw sequence Σ. The draws can be imagined as the seed for generators of pseudo-random numbers. When a statement of the form $v \leftarrow D$ occurs, the top element of Σ is taken to compute a random value according to the distribution D. We will have a look at the inference rules for all important syntactical elements of a *PROBCORE* program and

see how they are used to evaluate a given program concretely. The whole set of inference rules can be found in [SPM$^+$].

The first rule that we consider models arithmetic operations with $\circ \in \{+, *, \div\}$:

$$\frac{(H, e_1) \Downarrow_c v_1 \quad (H, e_2) \Downarrow_c v_2}{(H, e_1 \circ e_2) \Downarrow_c v_1 \circ v_2}$$

$\Downarrow_c$ is the symbol for our concrete big-step semantics. The inference rule is stating that when two expressions e_1 respectively e_2 evaluate with a heap configuration H to values v_1 respectively v_2, $e_1 \circ e_2$ evaluates to the value $v_1 \circ v_2$. The subscript c in $\Downarrow_c$ stands for 'concrete'.

Similarly, we define for conditions with $\circ' \in \{\wedge, \vee\}$ that when condition c_1 evaluates to a Boolean b_1 and a condition c_2 to a Boolean b_2, $c_1 \circ' c_2$ evaluates to $b_1 \circ' b_2$:

$$\frac{(H, c_1) \Downarrow_c b_1 \quad (H, c_2) \Downarrow_c b_2}{(H, c_1 \circ' c_2) \Downarrow_c b_1 \circ' b_2}$$

We now consider the transition relation. The assignment of an expression e, evaluating to a value x, to a variable v is defined. The heap assignment for v is updated and all other variables stay untouched.

$$\frac{(H, e) \Downarrow_c x}{(\Sigma, H, v := e) \rightarrow_c (\Sigma, (v \mapsto x) : H, \text{skip})}$$

The concrete semantics follows one branch of an if statement depending on whether the condition of the if statement is satisfied or not. For the case that the condition holds (the complementary case is completely analogously), we obtain:

$$\frac{(H, c) \Downarrow_c \text{true}}{(\Sigma, H, \text{if c } s_1 s_2) \Downarrow_c (\Sigma, H, s_1)}$$

The execution of a loop is intuitively modeled in the way that the loop is executed as long as the loop condition is satisfied:

$$\frac{}{(\Sigma, H, \text{while c } s) \rightarrow_c (\Sigma, H, \text{if c } (s; \text{ while c } s) \text{ skip})}$$

Sampling a value of a certain distribution corresponds to taking the first value σ of Σ and calculating the sample value $d(\sigma)$ with it:

$$\frac{\Sigma = \sigma : \Sigma'}{(\Sigma, H, v \leftarrow d) \rightarrow_c (\Sigma', (v \mapsto d(\sigma)) : H, \text{skip})}$$

Also consider:

$$\frac{}{(\Sigma, H, \text{skip}; s_2) \rightarrow_c (\Sigma, H, s_2)}$$

The whole set of rules enables us to define whether a probabilistic assertion is satisfied by stating that it is when its condition is satisfied after the execution of the preceding program:

$$\frac{(\Sigma, H_0, s) \rightarrow_c^* (\Sigma', H', \text{skip}) \quad (H', c) \Downarrow_c b}{(\Sigma, H_0, s \ ; \ ; \text{ passert } c) \Downarrow_c b}$$

Figure 1.5 shows the evaluation of a *PROBCORE* program according to the inference rule for probabilistic assertions.

PROBCORE program:

$x \leftarrow \text{Gauss}(0, 1)$;
if $x > 0.1$ $x := 1$ $x := -1$; ;
passert $x = 1$

Premise 1:
$(\Sigma = \sigma_1 : \Sigma', \emptyset, x \leftarrow \text{Gauss}(0, 1)) \rightarrow_c$
$(\Sigma', \{x \mapsto d(\sigma_1) = 0.2\}, \text{skip}) \rightarrow_c$
$(\Sigma', \{x \mapsto 0.2\}, \text{if } x > 0.1 \ x := 1 \ x := -1) \rightarrow_c$
$(\Sigma', \{x \mapsto 0.2\}, \text{if } 0.2 > 0.1 \ x := 1 \ x := -1) \rightarrow_c$
$(\Sigma', \{x \mapsto 0.2\}, \text{if true } x := 1 \ x := -1) \rightarrow_c$
$(\Sigma', \{x \mapsto 0.2\}, x := 1) \rightarrow_c$
$(\Sigma', \{x \mapsto 1\}, \text{skip})$

Premise 2:
$(\{x \mapsto 1\}, x = 1) \Downarrow_c \text{true}$

Figure 1.5: Sample *PROBCORE* program and the evaluation of the probabilistic assertion with the concrete semantics

Chapter 2

Algorithm for Verifying Probabilistic Assertions

After we have introduced the concrete semantics of *PROBCORE*, this chapter focuses on the actual procedure of verifying probabilistic assertions. It overcomes the naive procedure of executing a probabilistic program several times and comparing the relative share where the corresponding probabilistic assertion, say *passert e p*, is met, with p. The following verification scheme is more sophisticated since it removes deterministic computations having the same output on every execution and uses statistical knowledge to reduce the model that is generated from the part involving probabilistic computations. In addition to that, we can specify a certain confidence level c.

2.1 Verification Scheme

The verification scheme is depicted in Figure 2.1. The input, a probabilistic program with a (probabilistic or concrete) input value, is first transformed into a *Bayesian network* representation encoding the probabilistic behavior of the input that is relevant for the probabilistic assertion of the program. A Bayesian network is a *directed acyclic graph* (DAG) where the nodes are random variables. The edges of a Bayesian network model conditional dependencies between the random variables. After the obtained Bayesian network representation has been optimized, the probabilistic assertion is either directly verified or sampling is applied.

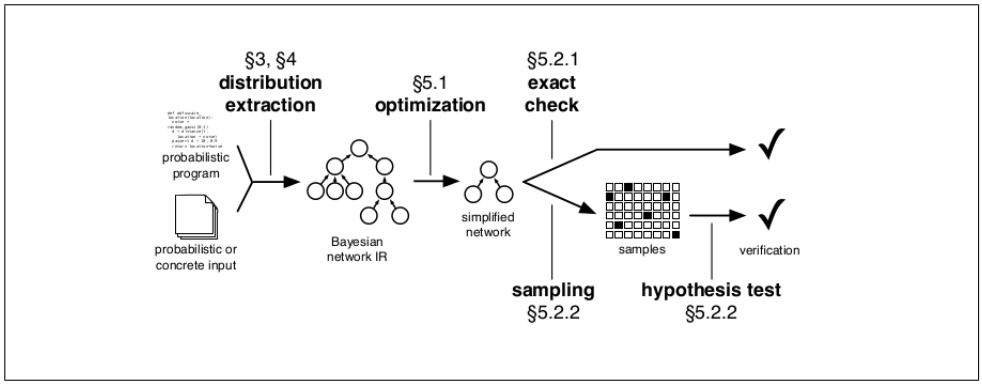

Figure 2.1: Verification scheme [SPM+14]

2.2 Distribution Extraction

Distribution extraction yields a Bayesian network by passing over the input program until the *passert* is reached. We aim to only include those parts in the Bayesian network contributing to the *passert*. We call such a part *probabilistic slice*. This process provides the advantage of removing deterministic computations having the same output on every execution and enables us to apply stochastic laws such as the *Central Limit Theorem* in order to minimize the resulting Bayesian network. Therefore, we concretely evaluate deterministic parts and create symbolic nodes for probabilistic values. In particular, we do not take draws from distributions when variables are assigned to a probabilistic value but represent these variables symbolically by their distribution. After having passed over the input we obtain an *expression DAG* which represents a Bayesian network.

We illustrate this by a small example. The expression DAG corresponding to the probabilistic program from Figure 1.4 is presented in the left half of Figure 2.2. We first observe that the addition of the results of the calls of $detProc(a)$ and $detProc(b)$ is merged into one node labeled with c due to the fact that it is a purely deterministic computation. Therefore, deterministic computations are not repeated over and over again. Secondly, the red nodes do not have to be considered since they are not reachable from the node for the *passert*. Hence, this formalization removes redundant deterministic computations and only considers those parts of the input program that are relevant for the probabilistic assertion to be considered. Reverting the direction of the edges of such an *expression DAG* yields a Bayesian network in which the nodes model random variables and the edges model the dependencies that the program creates between the latter. Constants are simply point-mass distributions. The representation as a Bayesian network which is a well-known formalism in statistics provides the advantage of a rich variety of optimizations on the latter.

The resulting Bayesian network of the example program in Figure 1.4 is visualized in the right half of Figure 2.2. The double circled node models a random variable for the probabilistic assertion. Note that in symbolic verification both branches of every if statement need to be considered, since we consider taking samples from a distribution symbolically and do not take random draws. This is done by considering both branches and merging conflicting updates of variable contents. The chapter about the symbolic semantics shows how this is technically done.

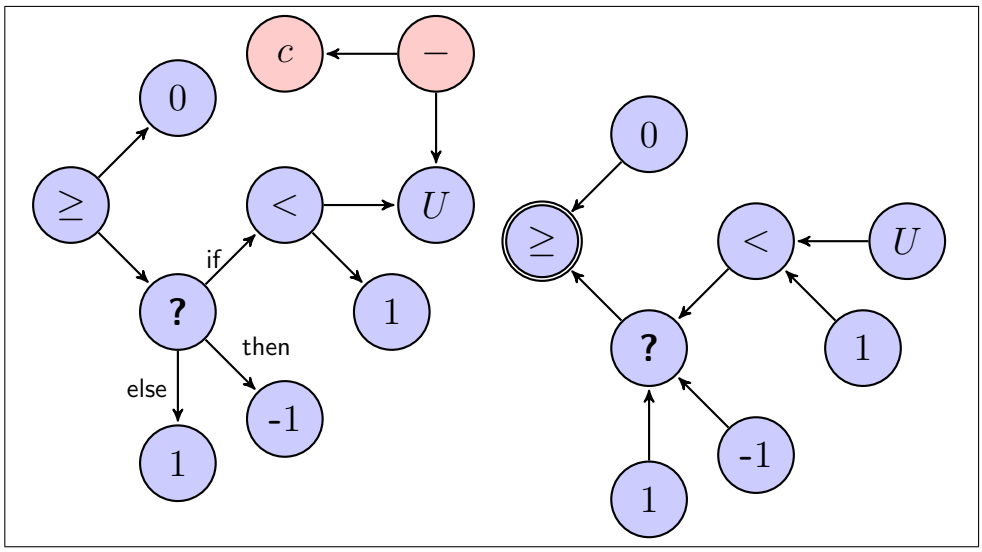

Figure 2.2: Expression DAG (left) and Bayesian network (right) of the example program in Figure 1.4

2.3 Handling Loops

One problem of integrating loops in symbolic execution is that these can have an unbounded number of iterations meaning that a Bayesian network which models a slice of a probabilistic program containing such a loop would contain cycles. This would violate the DAG property of Bayesian networks. Consider for example a repeated coin flip situation:

```
v1 ← Bernoulli(0.5) ;
while v1 = HEAD v1 ← Bernoulli(0.5);
```

Figure 2.3: Repeated coin flip

Such loops are modeled as *summary nodes* where variables that are read in the loop body lead to edges into these nodes and variables that are written

in the loop body lead to edges out of these nodes. These summary nodes, or black boxes, are used to take samples from them in order to prune unlikely paths. To be precise, *MAYHAP* uses path pruning to iteratively exclude paths that are unlikely to occur. In particular, *MAYHAP* tries to show that the probability of conditions that would result in following a certain path goes below some threshold and excludes these paths from further considerations. Programs that have a high probability of non-termination can still cause non-termination of the analysis. However, this rather suggests a bug in the program. As the authors of [SPM$^+$14] point out, this approach is far from being complete and leave its extension to future work.

When the loop condition is deterministically bounded, e.g. when the number of iterations only depends on constants, *MAYHAP* is able to derive the distributions modeling such loops.

2.4 Soundness

In this section we sketch how the soundness of the presented verification scheme is shown. The complete soundness proof can be found in [SPM$^+$]. The general idea for the soundness proof is to use the concrete semantics that we have presented before and a symbolic semantics for the distribution extraction. Proceeding by a structural induction over the program, one obtains that both semantics evaluate a given *PROBCORE* program equally.

2.4.1 Symbolic Semantics

The symbolic semantics formalizes the distribution extraction process. Values in the symbolic semantics are expression trees representing Bayesian networks. The outcome of a symbolic execution is the expression tree of the passert condition. We will only consider those inference rules that distinguish the symbolic semantics from the concrete one. While we have used a sequence of draws Σ for the concrete semantics, the symbolic semantics does not require that because the distributions are not evaluated. Instead it requires a stream offset, a natural number keeping track of how many samples have already been taken, for every sample of a distribution. We will see how it is integrated into the semantics and why its use is important. As we have already mentioned, statements evaluate to expression trees representing Bayesian network. This is exemplified in the rules for the arithmetic operations where e.g. $\{x_1 + x_2\}$ represents an expression in the expression tree. The curly braces indicate delayed evaluation. Let $\circ \in \{+, *, \div\}$.

$$\frac{(H, e_1) \Downarrow_s \{x_1\} \quad (H, e_2) \Downarrow_s \{x_2\}}{(H, e_1 \circ e_2) \Downarrow_s \{x_1 \circ x_2\}}$$

A lot of the other rules that we considered in the concrete semantics are defined analogously. The interested reader finds them in [SPM$^+$].

The inference rule for sample statements shows that we increase the stream offset by one and save the increased stream offset and the corresponding distribution for the variable that is assigned with the sample:

$$\frac{}{(n, H, v \leftarrow d) \rightarrow_s (n+1, (v \mapsto \{(d, n)\}) : H, \text{skip})}$$

This rule points out the necessity of the stream offset. Rather than keeping track of the concrete value of v, it is recorded that it is the n-th sample. If we would not apply that, different programs would have identical symbolic semantics as for example $x_1 \leftarrow Uniform(0,1); x_3 = x_1 * x_1$ and $x_1 \leftarrow Uniform(0,1); x_2 \leftarrow Uniform(0,1); x_3 = x_1 * x_2$.

Since the distribution extraction does not evaluate distributions, if-branches whose condition depends on probabilistic values are both executed. For conflicting heap updates of the branches we merge the resulting heaps depending on the if-condition. See [SPM$^+$] for the straight-forward definition of the merge operation. For two heaps obtained by the execution of different branches of an if-statement, it symbolically sets conflicting variable valuations to the value that is assigned in the branch that has to be executed depending on the if-condition. Accordingly, we also obtain a symbolic stream offset depending on the if-condition. The inference rule is given by:

$$\frac{(H, c) \Downarrow_s \{x\} \quad (n, H, b_t) \rightarrow_s^* (m_t, H_t, \text{skip}) \quad (n, H, b_f) \rightarrow_s^* (m_f, H_f, \text{skip})}{(n, H, \text{if } c\ b_t\ b_f) \rightarrow_s (\{\text{if } x\ m_t\ m_f\}, merge(H_t, H_f, \{x\}), \text{skip})}$$

Since the rule for while loops that we used for the concrete semantics would create infinite Bayesian networks we can not simply adopt it. The presented symbolic semantics only handles terminating while loops whose conditions do not depend on a probabilistic value. Further formalization of while loops is left to future work. Still, when a loop condition is proven to be false, we can skip it which is covered by the following inference rule ($\Downarrow_o$ is introduced after the passert rule):

$$\frac{(H, c) \Downarrow_s \{x\} \quad \forall \Sigma(\Sigma, \{x\}) \Downarrow_o \text{false}}{(n, H, \text{while } c\ s) \rightarrow (n, H, \text{skip})}$$

The symbolic evaluation of programs is defined as follows:

$$\frac{(0, H_0, s) \rightarrow_s^* (n, H', \text{skip}) \quad (H', c) \Downarrow_s \{x\}}{(H_0, s\ ;\ ;\ \text{passert } c) \Downarrow_s \{x\}}$$

The symbolic semantics enabled us to create an expression tree representing a Bayesian network. To evaluate the expression tree, we introduce $\Downarrow_o$ which evaluates an expression tree $\{x\}$ with a given draw sequence Σ denoted by $(\Sigma, \{x\}) \Downarrow_o v$. Thereby we concretely evaluate the symbolic parts. Consider for example the inference rule for the evaluation of arithmetic operations with $\circ \in \{+, *, \div\}$ and correspondingly the one for samples where σ_k denotes the k-th element of Σ:

$$\frac{(\Sigma, e_1) \Downarrow_o v_1 \quad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 \circ e_2) \Downarrow_o v_1 \circ v_2}$$

$$\frac{}{(\Sigma, (d, k)) \Downarrow_o d(\sigma_k)}$$

Again the whole set of rules for $\Downarrow_o$ can be found in [SPM$^+$]. An example illustrating the distribution extraction process can be found in Figure 2.2.

2.4.2 Final Result

By proceeding with a structural induction, the main theorem for the soundness is derived:

Theorem 2.4.1

Let $(0, H, p) \Downarrow_s \{x\}$*, where* x *is a finite program. Then* $(\Sigma, H_0, p) \Downarrow_c b$ *if and only if* $(\Sigma, x) \Downarrow_o b$*.*

It states that after the distribution extraction of a program p resulting in an expression tree, for any draw sequence Σ, the concrete evaluation of this program yields the same result as the evaluation of the extracted distribution. The soundness proof shows that the concrete semantics evaluates to the same output as the evaluation of the extracted distribution for all syntactical constructs of *PROBCORE*. To give an impression how the proof works, Figure 2.4 exemplary shows the proof for loops. The mentioned Lemma 2 states soundness for conditions.

while $c\ s$ There are two inference rules concerning the symbolic semantics of **while** loops, so we must prove that both are sound.

First consider the rule WHILE0. If it applies, we must have $(\Sigma, x) \Downarrow_o$ **false** for $(H_s, c) \Downarrow_s \{x\}$, and thus (by lemma 2) $(H_c, c) \Downarrow_c$ **false**. Then $(\Sigma, H_c, \textbf{while}cs) \rightarrow^* (\Sigma, H_c, \textbf{skip})$. But by assumption, $(n, H_s, \textbf{while}cs) \rightarrow (n, H_s, \textbf{skip})$, so the inductive statement holds.

Second, consider the rule WHILE in the symbolic semantics. It is identical to the corresponding rule for **while**, so by induction this case is established.

Figure 2.4: Proof for while loops [SPM+]

Chapter 3

Optimizations

One important reason for slicing probabilistic programs by Bayesian networks is that we can exploit statistical knowledge to optimize these and therefore reducing the verification effort. As we will see, probabilistic assertions in our optimized model can either directly be verified for the case that the optimizations provide a simple Bernoulli distribution or by applying a sampling approach.

3.1 Arithmetic Operations on Common Distributions

The verification effort can be reduced by reducing the number of nodes that have to be considered in a Bayesian network. One way of achieving this is to combine nodes that represent common distributions which are associated by an arithmetic operation to one single node. Figure 3.1 shows such a reduction. Assume that we have two random variables X_1 and X_2 with $X_1 \sim \mathcal{N}_1(\mu_{X_1} = 1, \sigma^2_{X_1} = 16)$ and $X_2 \sim \mathcal{N}_2(\mu_{X_2} = 5, \sigma^2_{X_2} = 9)$, then the sum of these is also a Gaussian distribution with $X_1 + X_2 = X_3 \sim \mathcal{N}_3(\mu_{X_1} + \mu_{X_2} = 6, \sigma^2_{X_1} + \sigma^2_{X_2} = 25)$. *MAYHAP* exhibits such reductions for

a rich variety of distributions and arithmetic operations. Still, the catalog of this kind of reductions is to be extended in future work.

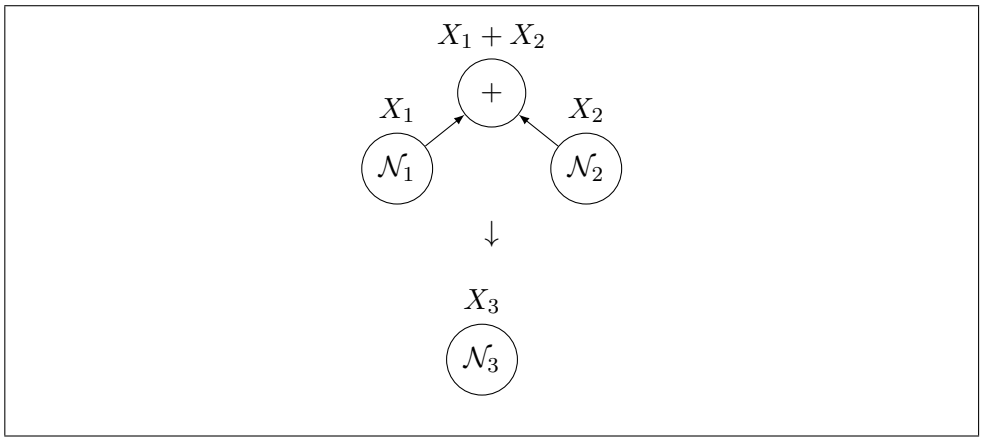

Figure 3.1: Reduction of the sum of two Gaussians

3.2 Simplifying Inequalities

Some cases require to check whether a probabilistic value is smaller than some constant. In this case the *cumulative distribution function* (CDF) of the distribution which the probabilistic value depends on can be exploited. The CDF $CDF_X(x)$ of a random variable say X is given by $CDF_X(x) = Pr(X < x)$. Intuitively spoken it gives the probability that the outcome of X is less than some real-valued x. Therefore instead of checking $X < c$, we compute $CDF_X(x)$. This process is depicted in Figure 3.2. Assume for example that X_1 is a random variable with $X_1 \sim \mathcal{N}(1, 5)$.

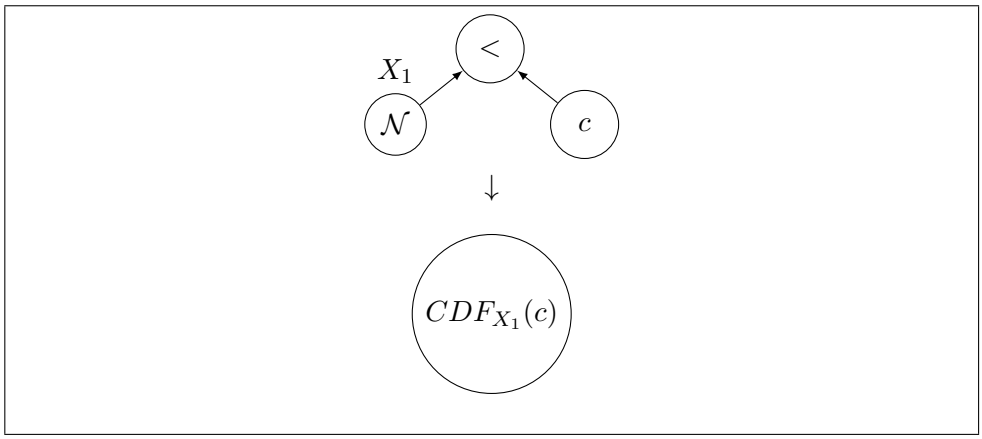

Figure 3.2: Reduction of a probabilistic inequality

3.3 Central Limit Theorem

The *Central Limit Theorem* (CLT) states that the sum of a large amount of independent random variables that are identically distributed and have a finite expected value and variance converges to a normal distribution. *MAYHAP* exploits that when the sum of such random variables is calculated, e.g. in an iterative process, to transform such a sum into a Gaussian distribution. Figure 3.3 illustrates that for a distribution D satisfying the necessary conditions for the application of the CLT.

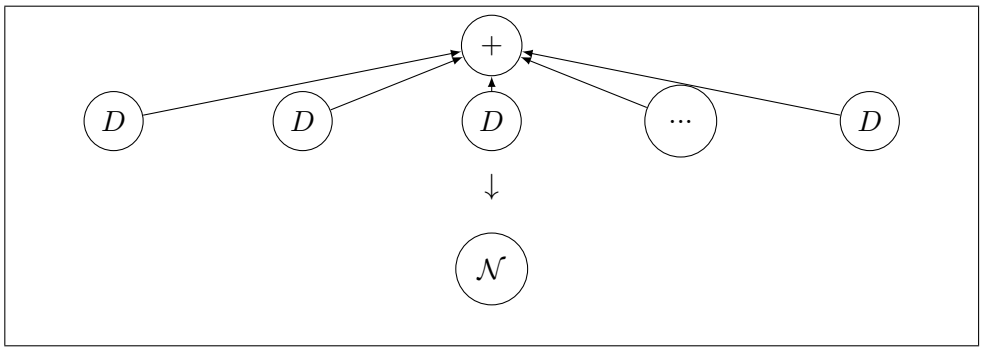

Figure 3.3: Transformation of a huge sum of random variables into a Gaussian distribution

3.4 Sampling Approach

While there exist probabilistic assertions that can be verified directly on the optimized Bayesian network without using sampling, this of course does not always apply. When direct verification is not possible *MAYHAP* uses statistical verification on the optimized Bayesian network. Since probabilistic assertions represent conditions evaluating to a Boolean value, they can be modeled as Bernoulli random variables. Assume that X_i with $i \in \{1, ..., n\}$ are n samples of a probabilistic assertion, then *MAYHAP* uses $p_\sim = \frac{1}{n} \sum_{i=1}^{n} X_i$ as an approximation of the probability p of the corresponding Bernoulli distribution. In order to ensure that $p_\sim$ is a good approximation of the actual p, the probability that $p_\sim \in [p - \epsilon, p + \epsilon]$ where ϵ is the desired accuracy is determined. The probabilistic assertion is verified as true when this probability is at least as big as $1-\alpha$ stating that with probability α the estimation is not accurate enough:

$$Pr(p_\sim \in [p - \epsilon, p + \epsilon]) \geq 1 - \alpha \text{ (i)}$$

When using *MAYHAP* a programmer can input α and ϵ for the verification of a probabilistic assertion. A priori it is not clear how many samples n are needed to satisfy the desired accuracy and confidence. So to ensure that (i) holds, the two-sided *Chernoff bound* is used that gives an upper bound for the probability that $p_\sim$ differs more from p than ϵ:

$$Pr(\mid p_{\sim} - p \mid \geq \epsilon p) \leq 2e^{-\frac{\epsilon^2 np}{2+\epsilon}}$$

$Pr(\mid p_{\sim} - p \mid \geq \epsilon p)$ is α by definition. Setting $p = 1$ which results in the worst case for the sampling procedure and reformulating this as an inequality depending on the number of samples n, we obtain for the minimum number of samples:

$$n \geq \frac{2+\epsilon}{\epsilon^2} ln(\frac{2}{\alpha})$$

Since this approach only calculates an upper bound for the number of samples, obtaining them in this manner can be extended by checking after every iteration whether (i) is satisfied. Such an extension is left to future work.

Chapter 4

Evaluation

4.1 MAYHAP

Before we have a look at the results, we briefly examine the architecture and some of the implementation details of *MAYHAP*. The latter compiles C and C++ code using the LLVM compiler infrastructure and applies the procedure that we have presented. Programmers can simply add passert(e) to their code in order to notify that a probabilistic assertion shall be checked. Furthermore, one can specify the desired accuracy ϵ and confidence α.

4.2 Results

MAYHAP was tested on some benchmarks of representative problem domains in probabilistic computing, namely sensors, differential privacy and approximate computing. Figure 4.1 from [SPM+14] elaborates what the particular benchmarks compute and which kind of probabilistic assertions are tested. *MAYHAP*s approach is measured by comparing it to naive stress testing which is explained at the beginning of chapter two. All programs were tested with a confidence of $\alpha = 0.05$ and an accuracy of $\epsilon = 0.01$ which according to Chernoffs bound results in 74147 samples.

Program	Description and passert	Time (seconds)			Optimization Counts		
		Baseline	Analysis	Sampling	Arith	Dist Op	CLT
gpswalk	Location sensing and velocity calculation passert: Velocity is within normal walking speed	537.0	1.6	59.0	1914	0	1
salary	Calculate average of concrete obfuscated salaries passert: Obfuscated mean is close to true mean	150.0	2.5	< 0.1	3	1	1
salary-abs	salary with abstract salaries drawn from a distribution passert: As above	87.0	20.0	0.2	5003	1	1
kmeans	Approximate clustering passert: Total distance is within threshold	1.8	0.3	< 0.1	2149	300	0
sobel	Approximate image filter passert: Average pixel difference is small	37.0	2.8	< 0.1	7880	0	1
hotspot	Approximate CMOS thermal simulation passert: Temperature error is low	422.0	4.7	28.0	1	24064	1
inversek2j	Approximate robotics control passert: Computed angles are close to inputs	4.8	< 0.1	< 0.1	901	200	1

Figure 4.1: Benchmarks

Figure 4.2 from [SPM+14] depicts the runtime of stress testing (B) which in each case defines the reference for the remaining runtimes of a non-optimized (N) and an optimized (O) verification scheme of *MAYHAP*. Furthermore, *MAYHAP*'s runtimes, non-optimized and optimized, are divided into the time needed for the analysis and the time spent executing the optimized representation. We can observe that, except for the benchmark $salary - abs$, the verification of the unoptimized representation provides massive advantages compared to stress testing. This is due to the fact that redundant deterministic parts are removed and we only consider slices of the input program. The impact of the optimizations is strongly dependent on the benchmark instance. While these heavily reduce the runtime on *salary*, *salary-abs* and *sobel*, the runtime even slightly increases on *hotspot* compared to non-optimized verification. Still, the results suggest that the additional runtime needed for the optimization steps, is worth it and can highly improve the verification performance.

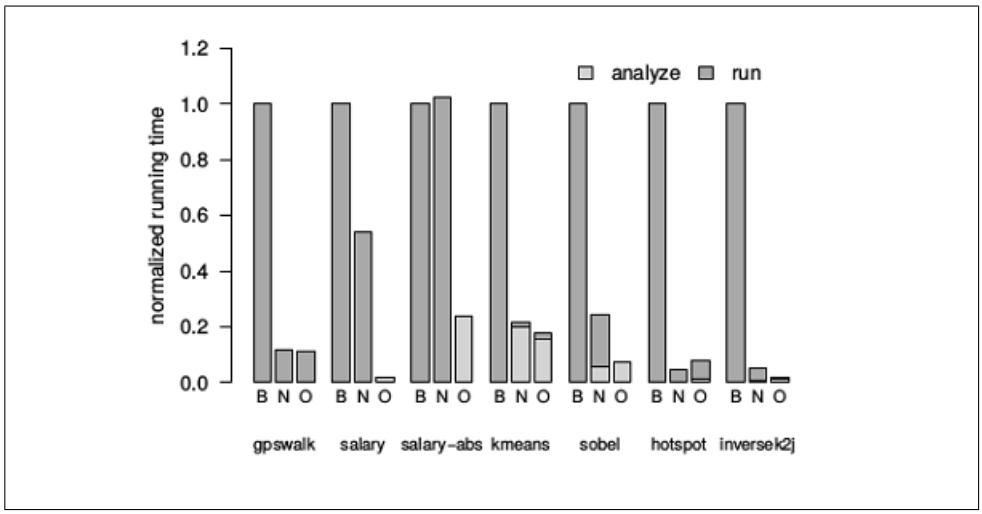

Figure 4.2: Results

4.3 Conclusion

After we have introduced the theoretical foundations that are required to understand the main approach, we presented the latter. We then formalized the distribution extraction and asserted the main theorem stating that the concrete semantics evaluates a given *PROBCORE* program identically as the symbolic semantics. Furthermore, we sketched how the soundness is shown. The evaluation suggests that the presented approach outperforms naive stress testing of probabilistic programs by removing redundant deterministic computation and symbolically executing probabilistic slices of the program.
In addition to that, we have seen that the presented notions can be extended at multiple working points, e.g. the bandwidth and depth of the used optimizations. Hence, future work will probably advance the verification process.

Bibliography

[Big] Big-step structural operational semantics. `http://fsl.cs.illinois.edu/images/b/b3/CS522-Spring-2011-PL-book-bigstep.pdf/`.

[Sam] Adrian Sampson. Expressing and verifying probabilistic assertions. `https://www.youtube.com/watch?v=84qpGAKIc4M`.

[Sma] Small-step structural operational semantics. `http://fsl.cs.illinois.edu/images/7/74/CS522-Spring-2011-PL-book-smallstep.pdf/`.

[SPH84] Micha Sharir, Amir Pnueli, and Sergiu Hart. Verification of probabilistic programs. *SIAM Journal on Computing*, 13(2):292–314, 1984.

[SPM+] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. Probabilistic assertions: Extended semantics and proof. *http://research.microsoft.com/pubs/211410/passert-aux.pdf*.

[SPM+14] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 14. ACM, 2014.