

RWTH AACHEN

The present work was submitted to
LEHR- UND FORSCHUNGSGEBIET INFORMATIK 2
and communicated by Jürgen Giesl

Bachelorthesis

Integrating KoAT's Push-Button Probabilistic Cost Analysis into Caesar

Thesis advisor:	Prof. Dr. Jürgen Giesl
Second examiner:	Dr. Tim Quatmann
Supervisors:	Philipp Schroer, Nils Lommen

Author:	Mats Eblenkamp
Matriculation number:	445395
Email:	mats.eblenkamp@rwth-aachen.de
Submission date:	15th April 2026

Abstract

Probabilistic programs model randomized algorithms and stochastic processes, but verifying their correctness and runtime behavior remains challenging. Caesar is a deductive verification infrastructure for probabilistic programs that requires users to provide pre-expectations as annotations. Finding these manually can be complex and error-prone. KoAT is a tool for automated cost complexity analysis of probabilistic integer programs that can automatically infer upper bounds on expected runtimes. In this thesis, we develop and implement a fully automatic translation from Caesar’s input language HeyVL into KoAT’s input format, enabling users to invoke KoAT at the push of a button to obtain runtime bounds. We formally prove the correctness of our translation, showing that KoAT’s result constitutes a valid upper bound on the original program’s expected runtime. We empirically evaluated our implementation and showed that our translation produces consistent results compared to KoAT’s native analysis.

Contents

1	Introduction	1
2	Probabilistic Programs	5
2.1	pGCL	6
2.1.1	Syntax	6
2.2	Markov Decision Processes	9
2.3	Expected Rewards	11
2.4	Operational MDP Semantics	13
2.5	HeyVL	18
2.5.1	Post-Expectations	18
2.5.2	Pre-Expectations	19
2.5.3	(Co)procedures	19
3	Analyzing HeyVL programs with KoAT	21
3.1	The JANI Format	21
3.1.1	Symbolic MDPs	22
3.1.2	The JANI Translation	24
3.2	The KoAT Format	33
3.3	Translating JANI to KoAT’s input	35
3.3.1	Preprocessing	35
3.3.2	Implementation Details	37
3.4	Soundness	39
4	Evaluation	41
4.1	Experimental Setup	41
4.2	Results and Discussion	42
4.2.1	Performance	43
4.2.2	Result Quality	44
4.2.3	Correctness	45
4.2.4	Summary	45
5	Conclusion and Future Work	47
A	Evaluation Results	51

List of Figures

1	Translation Pipeline	2
2	Example MDP representing a pGCL program	10
3	Rules for the small-step execution relation $\rightarrow$	15
4	Operational Markov chain $\mathcal{M}_{\langle C, \sigma \rangle}$ with program C from Example 1 and σ any program state with $\sigma(x) = 3$	17
5	Definition of the Function <i>trnsf</i>	25
6	SM_C for Example Program C from Example 1	27
7	Definition of the Function $\tau : \mathcal{T}_1 \rightarrow \mathcal{T}_2$	28
8	Comparison of the two HeyVL encodings for the 21 benchmarks that contain probability distributions. The top row represents encoding via distribution expressions, the bottom row represents encoding via probabilistic choices. .	42
9	Summary results for all 44 benchmarks across the three analysis pipelines. .	43
10	KoAT runtime as a function of program size. Timeouts are excluded.	44
11	Comparing computed upper bounds with known pre-expectations of se- lected benchmarks. Benchmarks with non-trivial postexpectations unequal to 0 are marked with $\dagger$	46
12	Asymptotic Results	52
13	Exact Results HeyVL	53
14	Exact Results KoAT vs. eco-imp	54
15	Total Time	55

Chapter 1

Introduction

Probabilistic programs are programs that contain any sort of nondeterministic or random behavior. They are used in randomized algorithms, communication protocols, models of physical and biological processes, machine learning, and artificial intelligence [SBK⁺23]. When writing probabilistic programs, unnoticed bugs can easily occur. For example, bugs that are not found in testing because they have a low probability of occurring. As such, automated verification and runtime analysis of these programs are especially relevant to ensure correctness.

Caesar is a deductive verification infrastructure for probabilistic programs that is being developed at the Software Modeling and Verification Group (MOVES) at RWTH Aachen University in cooperation with Saarland University, Denmark Technical University, and University College London [Vera]. It can verify whether a program does what it is supposed to by reasoning about expected values of probabilistic programs, for example, the expected value of some variable or some probability [SBK⁺23]. In particular, it can also verify upper or lower bounds on the expected runtime of a program [SBK⁺23]. For this, **Caesar** uses a mixture of different verification techniques and proof rules to make the verification process relatively automated on one hand, while also allowing the user to have some influence on the process on the other hand [SBK⁺23].

For **Caesar**'s verification, users need to annotate the program with a pre-expectation. Such a pre-expectation encodes an upper or lower bound on the expected value of a given post-expectation. For runtime verification, this pre-expectation serves as a specification of the program's expected cost. Users also need to provide loop invariants for all loops in the program, which need to hold for all loop iterations [SBK⁺23, SHK25]. Finding correct pre-expectations and invariants manually can be quite complex and cumbersome [SHK25], since it requires a thorough understanding of the program, is error-prone, and does not scale well to larger programs.

On the other hand, we have **KoAT**, a tool for complexity analysis of probabilistic integer programs. **KoAT** is able to automatically infer upper bounds on the expected runtimes for many probabilistic integer programs [MHG21a, LG24]. Consider the following simple example:

```
while ( $2 \leq x$ ) {  
   $\{x := x - 1\}[1/2]\{x := x - 2\}$ ;  
  reward 1  
}
```

This program decrements x by either 1 or 2 in each loop iteration with equal probability, terminating when $x < 2$. The exact expected runtime is $(\frac{2}{3}) \cdot x$. When fed with the translated version of this program, KoAT automatically infers an upper bound of $|x|$. This is a valid pre-expectation that Caesar can use directly, with no manual effort required. This example will serve as our running example throughout the thesis. This means, when looking for a program’s pre-expectation in Caesar to verify its runtime, feeding the program into KoAT can often automatically provide a correct pre-expectation for Caesar to use, without the hassle of the user having to compute one themselves.

The goal of this thesis is to create and implement an automatic translation from Caesar’s input language HeyVL to KoAT’s input format to allow users to call KoAT on a HeyVL program at the push of a button and thus receive an upper bound on the expected runtime.

The translation process proceeds in two steps, as depicted in Figure 1. Caesar already supports a translation from HeyVL to the JANI format [Verb]. JANI is an intermediate model representation originally designed for probabilistic model checkers such as Storm [Hen18]. We utilize Caesar’s translation and JANI’s similar structure to the KoAT input and develop a second step that translates the JANI model into a KoAT input file. After that, we invoke KoAT to automatically compute an upper bound on the program’s expected runtime.

The research questions of our thesis are

- **RQ1 - Correctness:**
Can our translation be used to automatically derive sound upper bounds on expected runtimes that are consistent with known pre-expectations of HeyVL programs?
- **RQ2 - Result quality:**
To what extent does our implementation produce the same asymptotic and exact complexity results as running KoAT directly on a manually optimized KoAT input?
- **RQ3 - Performance:**
How does the runtime of our pipeline compare to that of KoAT and *eco-imp*, the cost analysis tool that is the origin of our benchmark set?

We will start by introducing probabilistic programs and their semantics defined over Markov decision processes (MDPs) in Chapter 2. We will also define expected runtimes for our context and how they are measured, as well as how they relate to HeyVL’s pre-expectations.

In Chapter 3 we will then move on to formally define Caesar’s automatic translation into the JANI model and prove that the expected runtime of the original program is bounded above by that of the resulting JANI model. We go on to introduce KoAT and its input format before developing a new translation from JANI to KoAT and proving the expected runtime remains the same. This already answers **RQ1**, i.e., the formal correctness of the translation pipeline.

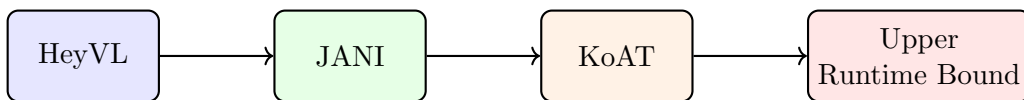


Figure 1: Translation Pipeline

We implemented the full translation pipeline in Rust and evaluate it on a set of 44 benchmarks in [Chapter 4](#), comparing results and runtimes against both a direct KoAT encoding and the `eco-imp` tool. This addresses **RQ2** and **RQ3**. Lastly, we select a few benchmarks and compare their results to each program’s actual pre-expectation to gather further empirical support for the correctness of our translation (**RQ1**).

Chapter 2

Probabilistic Programs

Probabilistic programs are programs that contain any sort of nondeterministic or random behavior. It is important to distinguish between randomness and nondeterminism in this context. Randomness is always assigned a probability. For example, in a fair coin toss, the probability for either heads or tails is 0.5. Nondeterminism, on the other hand, does not allow for statements about the probability of a certain outcome but only that it is possible and thus can either happen or not. For this reason we differentiate between the concepts of nondeterministic choice and probabilistic choice in probabilistic programs. In practice, probabilistic programs can be used, for example, to model stochastic processes in fields such as machine learning and artificial intelligence, but also for randomized algorithms, communication protocols, or to model physical processes [SBK⁺23].

The reasoning why assigning probabilities to specific events makes sense is relatively intuitive. Consider, for instance, we want to model a process where we know the statistical probability of a certain event (e.g., through extensive testing). Then it is obvious that we should use this probability in our model for the most realistic result.

The usefulness of a concept like nondeterminism may be harder to grasp. After all, what good is a program that operates by no specific heuristic and there is no way to at least know which outcome is the most likely? However, programs including nondeterminism usually use schedulers, which choose what the next action should be, according to some heuristic based on the current and previous program states. A scheduler could, for instance, always choose the path that affects the runtime in the best or worst possible way to find a best- or worst-case expected runtime of a program, without any information about the likelihood of the outcome. This makes nondeterministic choices useful if one has no statistical knowledge about the probability of a certain outcome and just knows that different outcomes could be possible. It can thus be used to model unknown environmental influences or unknown decisions. For example if someone has written two different subprograms to solve a problem and has not decided on one yet, they could insert a nondeterministic choice between the two subprograms and consider the worst (or best) case outcome.

In this chapter, we will first introduce the probabilistic language pGCL and then go on to define MDPs and expected rewards over them. We will need those to formally prove the correctness of our translation in [Chapter 3](#). We use pGCL to formalize the semantics of probabilistic programs and of our translation in this thesis because it has well-established semantics over MDPs, and our implementation only focuses on an executable part of HeyVL that can be encoded in pGCL as well and vice versa. HeyVL itself is a verification language for probabilistic programs, rather than a probabilistic programming language.

We will further explain what HeyVL does and what its pre-expectations are exactly, towards the end of this chapter.

2.1 pGCL

pGCL is a probabilistic programming language featuring both probabilistic choice and nondeterminism. It is named after Dijkstra's Guarded Command Language (GCL), which only includes nondeterministic features. In addition to usual programming statements like variable assignments, if-statements, or while-loops, which are included in GCL, pGCL also allows for probabilistic choices and random assignments. Consider [Example 1](#), which was already introduced in [Chapter 1](#). It subtracts either 1 or 2 from x in every step and counts the number of loop iterations until termination with a reward statement. We will explain its formal syntax and semantics in the following.

Example 1 (pGCL Program). An example program that looks like this:

```
while ( $2 \leq x$ ) {
   $\{x := x - 1\} [1/2] \{x := x - 2\};$ 
  reward 1
}
```

2.1.1 Syntax

For the formal definition of a probabilistic GCL, we adopt the one from [\[Kam19\]](#) and extend it by a reward statement as in [\[BKMW24\]](#):

Definition 1 (pGCL Syntax). Let $\mathbf{Vars}$ be a countable set of program variables and let $\mathbf{Vals}$ be a countable set of values. In our case, we set $\mathbf{Vals} = \mathbb{Z} \cup \{\mathbf{true}, \mathbf{false}\}$ as the set of integers and Booleans because that is what our translation supports. The reason is that KoAT only takes integer variables, so we can only allow values that can be encoded as integers. Furthermore, let $\Sigma = \{\sigma \mid \sigma : \mathbf{Vars} \rightarrow \mathbb{Z}\}$ be the set of program states for the entirety of this thesis, where each state stores a value for every variable. The set of programs in our pGCL is then described by this grammar and denoted by pGCL:

C	$\longrightarrow$	skip	(effectless program)
		$x := E$	(assignment)
		$x \approx \mu$	(random assignment)
		reward r	(collect reward)
		$C; C$	(sequential composition)
		$\{C\} [p] \{C\}$	(probabilistic choice)
		$\{C\} \square \{C\}$	(nondeterministic choice)
		if $(\varphi) \{C\}$ else $\{C\}$	(conditional choice)
		while $(\varphi) \{C\}$	(while loop)

Where

1. $x \in \mathbf{Vars}$ is a program variable.
2. φ is a *Boolean expression* over program variables. We write $\sigma \models \varphi$ and say σ satisfies φ if φ evaluated on σ is **true**. That means every $x \in \mathbf{Vars}$ that appears in φ is substituted by $\sigma(x)$ and then the expression is evaluated.
3. $E : \Sigma \rightarrow \mathbb{Z}$ is an *arithmetic expression* over program variables that maps every program state to a value.
4. $\mu : \mathbf{Vals} \rightarrow [0, 1] \cap \mathbb{Q}$ is a *distribution expression* representing a *probability distribution*¹. It maps a countable set of values $\mathbf{Vals}$ to rational probabilities with $\sum_{v \in \mathbf{Vals}} \mu(v) = 1$. Note that every probability distribution over values is a discrete probability distribution, since $\mathbf{Vals}$ is required to be countable.
5. $r : \Sigma \rightarrow \mathbb{N}$ is a non-negative *polynomial expression* over program variables that maps every state to a non-negative reward value
6. $p : \Sigma \rightarrow [0, 1] \cap \mathbb{Q}$ is a *probability expression* over program variables that maps every program state to a rational number, expressing a probability.

We will first focus on explaining the deterministic constructs before going into more detail on the others.

The *skip* program is effectless and simply terminates in the same state it started in, without making any changes.

The *reward* statement collects a reward r . In [Section 2.3](#), we will define a semantics for expected rewards, i.e., the expected value of rewards accumulated during each possible program execution. The expected rewards can for example be used to measure expected runtimes, like in [Example 1](#), where the reward counts the number of loop iterations before termination.

The *assignment* $x := E$ assigns x the value $E(\sigma)$ for the current program state σ . All other variables remain unaltered.

The *sequential composition* $C_1; C_2$ executes C_1 first, and after its termination executes C_2 .

The *conditional choice* $\text{if } (\varphi) \{C_1\} \text{ else } \{C_2\}$ evaluates φ in the current program state and either executes C_1 if φ evaluates to **true** or else executes C_2 if it evaluates to **false**.

Lastly, the construct $\text{while } (\varphi) \{C\}$ evaluates φ and executes C if the result is **true**, similar to the if-statement. After termination of C though, it loops back to the beginning and checks the condition again, this time with the program state resulting from the loop's last execution. It only terminates when φ evaluates to **false**. It is possible that this never happens and the loop runs forever without termination.

While most statements discussed so far should be familiar from classical programming languages and behave as one might expect, there are also three new constructs introduced. We will explain how they work in the following:

¹We differ here from the definition of [\[Kam19\]](#), where distribution expressions depend on the program state. The reason is that our pGCL definition is supposed to represent HeyVL's executable part and HeyVL's distributions only allow constant inputs (with the exception of *flip* that we will address in [Section 2.5](#))

Probabilistic choice

$$\{C_1\} [p] \{C_2\}$$

The probabilistic choice construct evaluates to a rational probability $p(\sigma)$ based on the current program state σ . It then executes program C_1 with this very probability and program C_2 with the remaining probability $1 - p(\sigma)$.

Example 2. The program

$$\{x := x - 1\} [1/2] \{x := x - 2\}$$

increases or decreases variable x by 1, both with a probability of $1/2$. Since the probability expression $1/2$ does not depend on any variables, it remains independent of the current program state. A different example where this is not the case would be

$$\{x := x - 1\} [1/x+1] \{x := x - 2\}.$$

Here the probability of each subprogram depends on x , so if the probability $1/2$ in [Example 1](#) was replaced by $1/x+1$, the probability that x is decreased by 1 would become a bit greater with every loop iteration.

Nondeterministic choice

$$\{C_1\} \square \{C_2\}$$

Similar to the probabilistic choice construct, the nondeterministic choice construct also executes either program C_1 or program C_2 . However, in contrast to probabilistic choice, a nondeterministic choice has no probability assigned to either case. This simply means, that while both choices are *possible*, neither one is more likely to happen than the other, nor do they explicitly have the same likelihood. In [Section 2.2](#) we will introduce schedulers, which resolve nondeterministic choices by deciding which subprogram will be executed.

Example 3. For example the program

$$\{x := y\} \square \{y := x\}$$

could have the outcome that x assumes the value of y or the other way around. A statement about which event is more likely to occur is not possible.

Random assignments

$$x \approx \mu$$

A random assignment evaluates the distribution expression, by assigning x a value sampled by distribution μ . That means a value $v \in Vals$ gets chosen with probability $\mu(v)$.

Example 4. The program

$$x \approx \text{Unif}(1, 3)$$

assigns x a value of 1, 2 or 3, each with probability $1/3$. Here $\text{Unif}(1, 3)$ is a concrete uniform probability distribution.

2.2 Markov Decision Processes

To understand the operational semantics of pGCL, as defined later in [Section 2.4](#), we first need to introduce Markov chains and Markov decision processes, which will be the basis of our operational semantics.

Markov chains are transition systems featuring probability distributions for successor states and are used to model probabilistic processes. However, Markov chains require concrete probability distributions and cannot handle nondeterministic choices. Since our definition for pGCL includes nondeterministic choice, we will introduce a more general variant of Markov chains, namely *Markov decision processes* (MDPs) that model nondeterministic behavior in addition to probabilistic behavior.

Definition 2 (Markov Decision Process (MDP)).² A *Markov Decision Process* is a tuple $\mathcal{M} = (\mathcal{L}, \mathcal{Act}, \mathcal{GT}, l_{init})$, with

- $\mathcal{L}$ being a countable set of locations,
- $\mathcal{Act}$ being a set of actions modeling different nondeterministic choices,
- $\mathcal{GT}$ being the set of general transitions. A general transition $g \in \mathcal{GT}$ is a set of transitions $t \in \mathcal{T} = \mathcal{L} \times \mathcal{Act} \times [0, 1] \times \mathcal{L}$, where we define $\mathcal{T}$ as the set of all possible transitions. For all transitions $t_i = (l_i, \alpha_i, p_i, l'_i) \in g$
 - The start location l_i and the action α_i have to be the same, so every general transition g has a common start location l_g and action α_g
 - The probabilities of all transitions must sum to either 0 or 1:

$$\sum_{t_i \in g} p_i \in \{0, 1\}$$

- l_{init} being the initial location.

An action α_g is *enabled* in location l_g if and only if $\sum_{t_i \in g} p_i = 1$. Let $\mathcal{Act}(l)$ be the set of enabled actions when in location l . Then it is required that for any location l that there has to be at least one action enabled, i.e., $\mathcal{Act}(l) \neq \emptyset$. For every transition $t = (l, \alpha, p, l')$ with $p > 0$, we call each l' an α -*successor* of l .

Intuitively, an MDP operates as follows: The locations are traversed via the transitions. First the initial location l_{init} is entered. When entering a location l , the next action α is chosen nondeterministically from the set of enabled actions $\mathcal{Act}(l)$. Executing this action then chooses a transition $t = (l, \alpha, p, l')$ with probability $p \in (0, 1] \cap \mathbb{Q}$ ³, leading to a new successor location l' .

Example 5. We depict an MDP as a transition system, as in [Figure 2a](#) representing the program in [Figure 2b](#). The program contains a loop that terminates with a probability of $1/2$ in every loop. With a probability of again, $1/2$ the loop is iterated once more and a reward is added. A nondeterministic choice decides whether the reward is 1 or 2.

²We present here a simplified version of the definition in [\[BKL08\]](#) that omits atomic propositions of states, since those are not needed for our purposes. We combine it with the notation of a PIP from [\[MHG21a\]](#) that we will later relate to MDPs.

³While theoretically possible, we ignore real number probabilities here for algorithmic purposes.

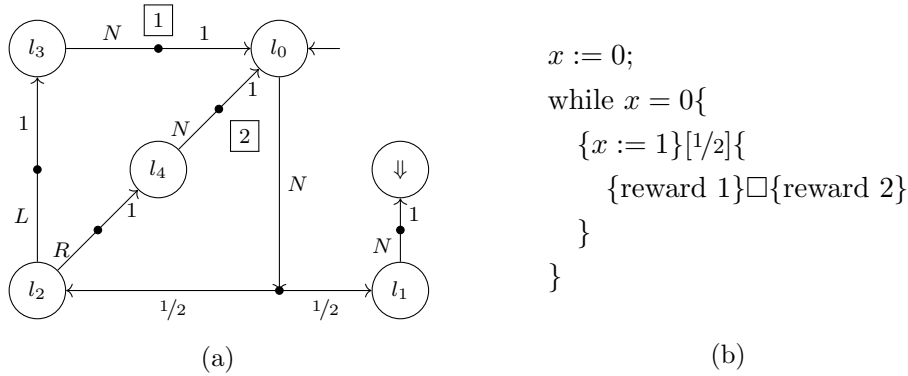


Figure 2: Example MDP representing a pGCL program

The MDP shown in Figure 2a is $\mathcal{M} = (\{l_0, l_1, l_2, l_3, l_4\}, \{L, N, R\}, \mathcal{GT}, l_0)$. The locations are represented by circles with their names written in them. The initial location l_0 is marked by an incoming arrow that does not originate from another location. From there, a transition to l_1 and l_2 labeled with the respective probabilities marks the probabilistic choice to either set $x = 1$ and terminate in the next step (l_1) or collect a reward (l_2).

In general, a transition $t = (l, \alpha, p, l')$ with $p > 0$ is represented by an edge from l to l' labeled with the action α and the respective probability p . Here, action N is used if there is only one choice available, and actions L and R represent the nondeterministic choice between l_3 and l_4 . From both l_3 and l_4 a deterministic transition with probability 1 returns to the initial location and repeats the loop. The boxes next to the edges represent the edges' rewards.

By traversing through an MDP's locations, we create paths consisting of the traversed locations. The following definitions of paths, schedulers and path probabilities are taken from Definition 2 of [BKMW24] but adjusted to include transitions.

Definition 3 (Paths). Let $\mathcal{M} = (\mathcal{L}, \mathcal{Act}, \mathcal{GT}, l_{init})$ be an MDP.

1. Then a *path* is a finite, non-empty sequence of locations

$$\pi = l_0 \xrightarrow{t_0} l_1 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} l_n$$

such that there is $t_i = (l_i, \alpha_i, p_i, l_{i+1})$ with $\alpha_i \in \mathcal{Act}(l_i)$ and $p_i > 0$ for each $i \in \{0, \dots, n-1\}$.

2. The set of paths with length n that start at location $l_0 \in \mathcal{L}$ is defined as

$$\text{Paths}_{\mathcal{M}}^n(l_0) = \{\pi \mid \pi = l_0 \xrightarrow{t_0} \dots \xrightarrow{t_{n-1}} l_n \text{ is a path in } \mathcal{M}\}.$$

3. We define the set of all finite paths starting in $l_0 \in \mathcal{L}$ as

$$\text{Paths}_{\mathcal{M}}(l_0) = \bigcup_{n \in \mathbb{N}} \text{Paths}_{\mathcal{M}}^n(l_0).$$

4. We write $\text{Paths}_{\mathcal{M}}^n$ instead of $\text{Paths}_{\mathcal{M}}^n(l_{init})$ and $\text{Paths}_{\mathcal{M}}$ instead of $\text{Paths}_{\mathcal{M}}(l_{init})$, respectively.

For nondeterministic actions, there needs to be some function to decide which one should be taken. We cannot just pick one at random because that would mean assigning a probability to the choice. This is where schedulers come into play.

Definition 4 (Scheduler).⁴ A scheduler for an MDP $\mathcal{M} = (\mathcal{L}, \mathcal{Act}, \mathcal{GT}, l_{init})$ is defined as

$$\mathfrak{S} : \mathcal{L} \rightarrow \mathcal{Act}$$

and maps locations to enabled actions. We denote the set of all schedulers of $\mathcal{M}$ as *Scheds* as $\mathcal{M}$ will always be clear from context.

Upon entering a location $l \in \mathcal{L}$, the scheduler chooses $\alpha = \mathfrak{S}(l)$ deterministically as the next action to resolve the nondeterminism between all enabled actions from $\mathcal{Act}(l)$. Then the next location l' is chosen between all possible transitions $t = (l, \alpha, p, l')$ with probability p . In this way, one can model different schedulers to look at different pathways through the MDP. For example a scheduler could always decide on the action that results in the longest runtime, i.e., aiming for worst-case expected runtimes.

After resolving the nondeterminism of MDPs through schedulers, we can compute a path's probability:

Definition 5 (Path Probabilities). For an MDP $\mathcal{M} = (\mathcal{L}, \mathcal{Act}, \mathcal{GT}, l_{init})$ and a path $\pi = l_0 \xrightarrow{t_0} \dots \xrightarrow{t_{n-1}} l_n$, with $t_i = (l_i, \mathfrak{S}(l_i), p_i, l_{i+1})$ the path's probability in $\mathcal{M}$ under a scheduler $\mathfrak{S}$ is defined as

$$\text{Prob}_{\mathfrak{S}}(\pi) = \prod_{i=0}^{n-1} p_i.$$

Markov Chains

If we now consider an MDP $\mathcal{M}$ with $|\mathcal{Act}(S)| = 1$, we remove the possibility of nondeterministic choice through actions. As you may recall, an MDP without the ability to model nondeterminism is called a *Markov chain* (MC).

Having only one action also leaves only one possible scheduler, namely the one that maps every location to the same action. Thus we will omit the scheduler and the transitions' actions in any notations regarding Markov chains and consider them implied.

2.3 Expected Rewards

As seen in the example MDP in [Figure 2a](#), MDPs can also be equipped with reward statements, to allow for modeling of resource consumption or runtime. For this, an MDP is assigned a reward function. For the definitions of rewards and expected rewards, we again follow [\[BKMW24\]](#) with slight modifications and additions, most notably that our reward function assigns rewards to transitions instead of states. We do this to later better show the relation between MDPs and KoAT's input format.

Definition 6 (Reward Function). A function

$$\text{rew} : \mathcal{GT} \rightarrow \mathbb{N}$$

for an MDP $\mathcal{M} = (\mathcal{L}, \mathcal{Act}, \mathcal{GT}, l_{init})$ assigns a natural number reward to every general transition. For a transition $t \in g$ we define

$$\text{rew}(t) = \text{rew}(g).$$

⁴In contrast to [\[BKMW24\]](#)'s scheduler definition, ours depends only on the current state.

The reward function can be extended to paths, where we sum up all rewards of the transitions along a path, to get a cumulative reward. Formally,

$$\text{rew}(l_0 \xrightarrow[t_0]{} \dots \xrightarrow[t_{n-1}]{} l_n) = \sum_{i=0}^{n-1} \text{rew}(t_i).$$

To evaluate the rewards of an MDP, both for specific paths and the MDP overall, we define expected rewards. Note that the maximal/minimal *expected* reward is not necessarily the highest or lowest *possible* reward (i.e., there could be a specific path with a higher/lower actual reward).

Definition 7 (Expected Rewards). Let $\mathcal{M} = (\mathcal{L}, \text{Act}, \mathcal{GT}, l_{\text{init}})$ be an MDP with reward function $\text{rew} : \mathcal{GT} \rightarrow \mathbb{N}$ and a scheduler $\mathfrak{S} \in \text{Scheds}$. Then we define:

1. The expected reward from $l \in \mathcal{L}$ after n steps under $\mathfrak{S}$ as

$$\text{ER}_{\mathfrak{S}}^n(\mathcal{M}, l, \text{rew}) = \sum_{\pi \in \text{Paths}_{\mathcal{M}}^n(l)} \text{Prob}_{\mathfrak{S}}(\pi) \cdot \text{rew}(\pi).$$

It goes through all paths of length n starting in l and sums up the product of every path's reward (as defined in [Definition 6](#)) with its probability, where every non-deterministic choice is resolved by $\mathfrak{S}$ (as defined in [Definition 5](#)).

2. The expected reward from $l \in \mathcal{L}$ under $\mathfrak{S}$ as

$$\text{ER}_{\mathfrak{S}}(\mathcal{M}, l, \text{rew}) = \sup_{n \in \mathbb{N}} \text{ER}_{\mathfrak{S}}^n(\mathcal{M}, l, \text{rew}).$$

It takes the supremum of the expected reward for paths of length n over all possible path lengths n . That means it is the least upper bound on the expected reward of all path lengths or more intuitively, the largest expected reward possible for a path starting in l .

3. The expected reward from $\mathcal{M}$ under $\mathfrak{S}$ as

$$\text{ER}_{\mathfrak{S}}(\mathcal{M}, \text{rew}) = \text{ER}_{\mathfrak{S}}(\mathcal{M}, l_{\text{init}}, \text{rew}).$$

It is simply the expected reward for the whole MDP that results from its initial location.

4. The minimal expected reward from $\mathcal{M}$ as

$$\text{MinER}(\mathcal{M}, \text{rew}) = \inf_{\mathfrak{S} \in \text{Scheds}} \text{ER}_{\mathfrak{S}}(\mathcal{M}, \text{rew}).$$

It takes the infimum of the MDP's expected reward over all schedulers, i.e., it looks for the scheduler that always makes its decisions such that the smallest possible expected reward results.

5. The maximal expected reward from $\mathcal{M}$ as

$$\text{MaxER}(\mathcal{M}, \text{rew}) = \sup_{\mathfrak{S} \in \text{Scheds}} \text{ER}_{\mathfrak{S}}(\mathcal{M}, \text{rew}).$$

The same intuition as for the minimal expected reward applies here, only that it looks for the scheduler that maximizes the expected reward.

Consider the MDP from Figure 2a again. Its minimal expected reward is obviously achieved by the scheduler that always chooses the left action when in location l_2 , formally $\mathfrak{S}_{min}(l_2) = L$. This results in a reward of 1 in every iteration. On the other hand, the maximal expected reward is achieved by always choosing the right option ($\mathfrak{S}_{max}(l_2) = R$) and going to location l_4 with a reward of 2. To actually compute the minimal and maximal expected rewards, we need to compute $\text{ER}_{\mathfrak{S}}(\mathcal{M}, l_0, \text{rew}) = \sup_{n \in \mathbb{N}} \text{ER}_{\mathfrak{S}}^n(\mathcal{M}, l_0, \text{rew})$ for both schedulers $\mathfrak{S}_{min}$ and $\mathfrak{S}_{max}$ respectively. The infinite path that always goes from l_0 to l_2 obviously results in the highest reward for both cases and is the supremum of all expected rewards of finite paths. Its expected reward is

$$1/2 \cdot 0 + 1/4 \cdot r + 1/8 \cdot 2r + \dots = \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i \cdot ir = r.$$

We generalize the formula here with a reward constant r , since the collected reward per loop iteration in Figure 2a is either always 1 for the minimum case or always 2 for the maximum case.

The explanation for the formula is the following: In the first iteration, the program either terminates with probability $1/2$ and a collected reward of 0 or collects a reward r and does another iteration. Here again we can terminate with probability $1/2$. The resulting path would thus have a probability of $1/2 \cdot 1/2 = 1/4$ and a total reward of r . Otherwise another iteration is executed, again collecting reward r and so on.

With that we get $\text{MinER}(\mathcal{M}, \text{rew}) = \text{ER}_{\mathfrak{S}_{min}}(\mathcal{M}, l_0, \text{rew}) = 1$ for $r = 1$ and $\text{MaxER}(\mathcal{M}, \text{rew}) = \text{ER}_{\mathfrak{S}_{max}}(\mathcal{M}, l_0, \text{rew}) = 2$ for $r = 2$.

If we look at expected rewards as modeling expected runtimes (which is our main viewpoint in this thesis), the maximal expected reward can be seen as a worst-case expected runtime. Because of this, this kind of nondeterminism is also called *demonic* nondeterminism. On the other hand, the minimal expected reward reflects the best-case expected runtime and is thus called *angelic* nondeterminism as a complement to the worst case. Note that there are differing definitions of angelic and demonic choice. Different sources might describe them the exact opposite as we do here, e.g., in [BKMW24] or [Kam19]. For instance, if one uses rewards not to encode some kind of cost, but rather a benefit that is desirable to get as high as possible, it would make sense to call the nondeterminism that achieves this angelic instead of demonic and the other way around. However, since we are focused on runtimes here and higher runtimes are generally seen as undesirable, we will stick to the convention defined first.

2.4 Operational MDP Semantics

The reason for our introduction of MDPs here, is that they can be used to represent programs, as we already saw in Figure 2. We will now take a look at a formal way to derive an MDP representing the semantics of a pGCL program. We follow the definition of [BKMW24] of the operational MDP of pGCL, adapted to our notation to remain consistent. The countable set of configurations is defined as

$$\text{Conf} = (\text{pGCL} \cup \{\downarrow\}) \times \Sigma \cup \{\perp\}.$$

pGCL is the set of all pGCL programs and Σ the set of all program states as defined in Definition 1. A configuration $\mathfrak{c} = (C, \sigma) \in \text{Conf}$ denotes that we want to execute program

C on program state σ . Configurations of form $(\Downarrow, \tau)$ mark *final* configurations with *final state* τ . $\perp$ marks a sink configuration that is reached from every final configuration and is needed because we need to be able to extend each finite path infinitely long, for our definition of expected rewards to make sense.

We define the smallest relation

$$\rightarrow \subseteq \text{Conf} \times \{L, R, N\} \times [0, 1] \times \text{Conf}$$

that satisfies the rules in Figure 3 as the *small step execution relation* as in [BKMW24] and follow their notation, where $\mathbf{c} \xrightarrow{\alpha, p} \mathbf{c}'$ is written instead of $(\mathbf{c}, \alpha, p, \mathbf{c}')$. We will explain the rules in more detail shortly. For now note that an inference rule of form $\frac{A}{B}$ can be read as "if A holds, B holds too". If the space above the line is empty, it simply means B always holds without any necessary prerequisites.

Definition 8 (Operational Markov Decision Process). The *operational MDP* of a pGCL program C_0 on an input σ_0 is the MDP $\mathcal{M}_{\langle C_0, \sigma_0 \rangle} = (\text{Conf}, \{L, R, N\}, \mathcal{GT}, \langle C_0, \sigma_0 \rangle)$, where

- Conf is the set of locations,
- $\{L, R, N\}$ is the set of actions,
- $\mathcal{GT}$ is the set of general transitions induced by $\rightarrow$, defined as

$$\mathcal{GT} = \{g(l, \alpha) \mid l \in \text{Conf}, \alpha \in \{L, R, N\}, g(l, \alpha) \neq \emptyset\},$$

with

$$g(l, \alpha) = \{(l, \alpha, p, l') \in \rightarrow \mid p \in (0, 1], l' \in \text{Conf}\}$$

- and $\langle C_0, \sigma_0 \rangle$ is the initial location

We define the *operational Markov chain* $\mathcal{M}_{\langle C_0, \sigma_0 \rangle} = (\text{Conf}, \{N\}, \mathcal{GT}, \langle C_0, \sigma_0 \rangle)$ of a pGCL program C_0 that does not contain nondeterminism analogously to the operational MDP. The transitions follow the same rules from Figure 3 with the exclusion of nondeterministic choice and the omission of action N in every rule.

The intuition of a configuration $\mathbf{c} = \langle C, \sigma \rangle \in \text{Conf}$, is that C stores the program that is left to execute and σ obviously stores the variables' values. Every transition advances one step further in the execution of the program. For nondeterministic and probabilistic choices, there can be multiple outgoing transitions. There are three possible actions L, N, R to encode nondeterministic choice. For deterministic choices, only N is enabled and for nondeterministic choices of form $\{C_1\} \square \{C_2\}$, L represents choice C_1 and R choice C_2 . The transition probability function also assigns probability 1 to all non-probabilistic transitions.

We will now explain all the rules from Figure 3 in detail.

1. These rules make sure that the sink configuration $\perp$ can be reached from every final state and that, once in $\perp$, every transition remains there.
2. The **skip** program does nothing, so the transition simply enters the final configuration without altering the program state σ .
3. The assignment alters the program state by assigning the value of E to x . This is marked by the notation $\sigma[x \mapsto E(\sigma)]$, which represents the state that maps every variable $y \neq x$ to $\sigma(y)$ and x to $E(\sigma)$. After the assignment the program should terminate, so the configuration entered is also final again.

1. Final configurations:

$$\frac{}{\langle \Downarrow, \sigma \rangle \xrightarrow{N,1} \perp} \quad \frac{}{\perp \xrightarrow{N,1} \perp}$$

2. Effectless Program:

$$\frac{}{\langle \text{skip}, \sigma \rangle \xrightarrow{N,1} \langle \Downarrow, \sigma \rangle}$$

3. Assignment:

$$\frac{}{\langle x := E, \sigma \rangle \xrightarrow{N,1} \langle \Downarrow, \sigma[x \mapsto E(\sigma)] \rangle}$$

4. Random Assignment:

$$\frac{}{\langle x := \mu, \sigma \rangle \xrightarrow{N, \mu(v)} \langle \Downarrow, \sigma[x \mapsto v] \rangle} \quad \forall v \in Vals \text{ with } \mu(v) > 0$$

5. Reward:

$$\frac{}{\langle \text{reward } r, \sigma \rangle \xrightarrow{N,1} \langle \Downarrow, \sigma \rangle}$$

6. Sequential Composition:

$$\frac{\langle C_1, \sigma \rangle \xrightarrow{\alpha, p(\sigma)} \langle \Downarrow, \sigma' \rangle}{\langle C_1; C_2, \sigma \rangle \xrightarrow{\alpha, p(\sigma)} \langle C_2, \sigma' \rangle} \quad \frac{\langle C_1, \sigma \rangle \xrightarrow{\alpha, p(\sigma)} \langle C'_1, \sigma' \rangle}{\langle C_1; C_2, \sigma \rangle \xrightarrow{\alpha, p(\sigma)} \langle C'_1; C_2, \sigma' \rangle}$$

7. Probabilistic Choice:

$$\frac{C_1 \neq C_2}{\langle \{C_1\} [p] \{C_2\}, \sigma \rangle \xrightarrow{N, p(\sigma)} \langle C_1, \sigma \rangle} \quad \frac{C_1 \neq C_2}{\langle \{C_1\} [p] \{C_2\}, \sigma \rangle \xrightarrow{N, 1-p(\sigma)} \langle C_2, \sigma \rangle}$$

$$\frac{}{\langle \{C\} [p] \{C\}, \sigma \rangle \xrightarrow{N,1} \langle C, \sigma \rangle}$$

8. Nondeterministic Choice:

$$\frac{}{\langle \{C_1\} \square \{C_2\}, \sigma \rangle \xrightarrow{L,1} \langle C_1, \sigma \rangle} \quad \frac{}{\langle \{C_1\} \square \{C_2\}, \sigma \rangle \xrightarrow{R,1} \langle C_2, \sigma \rangle}$$

9. Conditional Choice:

$$\frac{\sigma \models \varphi}{\langle \text{if } (\varphi) \{C_1\} \text{ else } \{C_2\}, \sigma \rangle \xrightarrow{N,1} \langle C_1, \sigma \rangle} \quad \frac{\sigma \models \neg \varphi}{\langle \text{if } (\varphi) \{C_1\} \text{ else } \{C_2\}, \sigma \rangle \xrightarrow{N,1} \langle C_2, \sigma \rangle}$$

10. While Loop:

$$\frac{\sigma \models \varphi}{\langle \text{while } (\varphi) \{C\}, \sigma \rangle \xrightarrow{N,1} \langle C; \text{while } (\varphi) \{C\}, \sigma \rangle} \quad \frac{\sigma \models \neg \varphi}{\langle \text{while } (\varphi) \{C\}, \sigma \rangle \xrightarrow{N,1} \langle \Downarrow, \sigma \rangle}$$

Figure 3: Rules for the small-step execution relation $\rightarrow$.

4. The random assignment samples a value from probability distribution μ and assigns it to x . So the program state σ is changed to $\langle \Downarrow, \sigma[x \mapsto v] \rangle$ for every $v \in \text{Vals}$ with $\mu(v) > 0$.
5. The **reward** statement accumulates reward $r(\sigma)$. However, this is encoded in the reward function (see Definition 9), not in the MDP itself, so there is no difference to **skip** for the transition rules. All it does is enter a final configuration without altering the state.
6. The sequential composition rules are the first ones with conditions that need to hold for the rule to apply. They can be seen as a case distinction. In the first case executing C_1 alone terminates with action α , probability $p(\sigma)$ and new program state σ' . Because of this, it can be deduced that executing one step on $C_1; C_2$ should also lead to program state σ' with the same action α and the same probability $p(\sigma)$. Furthermore, only C_2 is left to execute, so the new configuration is $\langle C_2, \sigma' \rangle$. In the other case, executing C_1 alone does not terminate but lead to a different configuration $\langle C'_1, \sigma' \rangle$. Here, C'_1 first needs to be executed until termination and then C_2 , so the rule dictates $\langle C'_1; C_2, \sigma' \rangle$ as the resulting configuration.
7. The probabilistic choice rules also have a case distinction, namely that either programs C_1 and C_2 differ, or they are the same. In the first case, program C_1 is executed with probability $p(\sigma)$ and program C_2 with probability $1 - p(\sigma)$. Thus there is one transition leading to C_1 and discarding C_2 with probability $p(\sigma)$ and another transition discarding C_1 and leading to C_2 with probability $1 - p(\sigma)$. The third rule depicts the case that $C_1 = C_2$ and is very straightforward. No matter if the probabilistic choice picks the first or the second program, it will be C in both cases, so the transition just goes to configuration $\langle C, \sigma \rangle$ with probability 1. In all three rules, the program state does not change, since it is only decided which subprogram is to be executed next.
8. For the nondeterministic transitions no conditions are necessary. Like mentioned in the beginning, the different actions exist to encode these two rules. One transition chooses the "left" program C_1 with action L , the other the "right" program C_2 with action R . The state σ remains unaltered again.
9. The conditional choice needs another case distinction. If φ is satisfied, C_1 is executed with probability 1 (and action N , since it is a deterministic transition), otherwise C_2 is executed with probability 1.
10. The while loop also checks whether its condition φ is satisfied. If it is not, the loop does not get executed anymore and thus leads to termination ($\Downarrow$). If φ is satisfied, the loop body C is executed and then the loop condition is checked again. Thus the transition goes to configuration $\langle C; \text{while}(\varphi)\{C\}, \sigma \rangle$. Since this step only checks φ , σ again doesn't get altered.

As mentioned, the reward statements do not change anything about the MDP's transitions but are only relevant for the corresponding reward function:

Definition 9 (Reward Function for the Operational MDP). The reward function for the operational MDP $\mathcal{M}_{\langle C_0, \sigma_0 \rangle}$ of a program C_0 on input σ_0 is defined as

$$\text{rew}_{\text{pGCL}}(\langle C, \sigma \rangle, \alpha, p, \langle C', \sigma' \rangle) = \begin{cases} r(\sigma) & \text{if } C = \text{reward } r \\ & \text{or } C = \text{reward } r; C' \\ 0 & \text{else.} \end{cases}$$

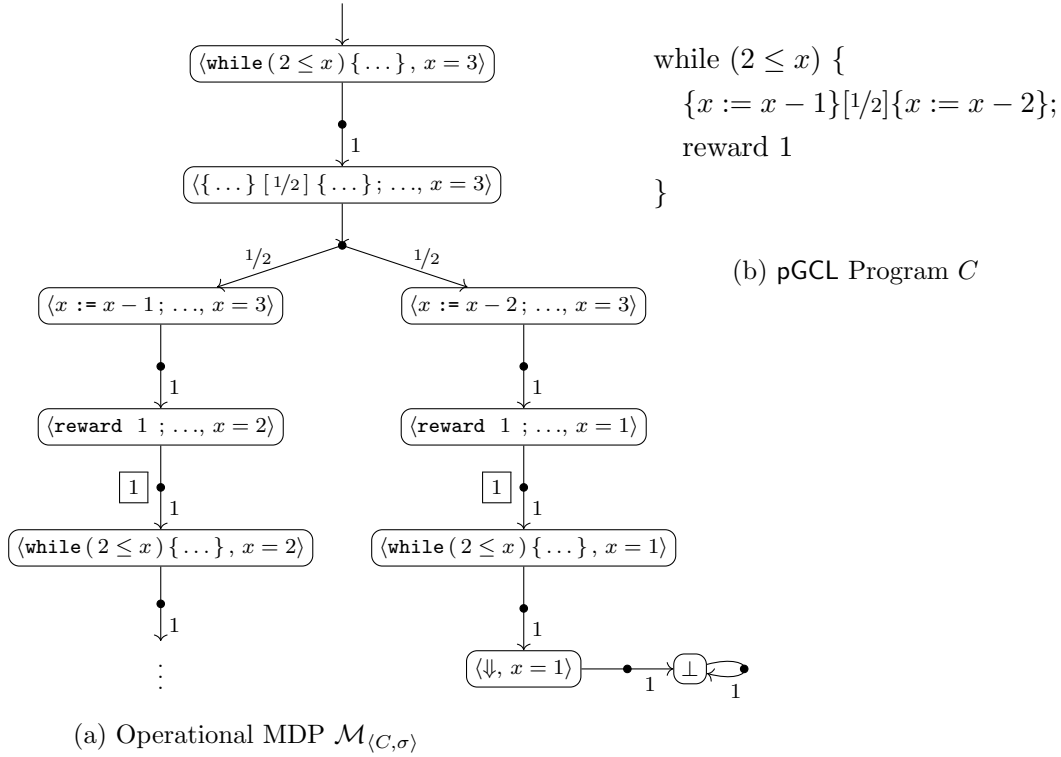


Figure 4: Operational Markov chain $\mathcal{M}_{\langle C, \sigma \rangle}$ with program C from [Example 1](#) and σ any program state with $\sigma(x) = 3$

Example 6 (Operational Markov Chain). Consider the operational Markov chain $\mathcal{M}_{\sigma}^C$ of program C from [Example 1](#) with an initial input state σ that satisfies $\sigma(x) = 3$, depicted in [Figure 4](#). Since C does not contain any nondeterminism, a Markov chain suffices here.

First the while loop's condition is evaluated. Since $2 \leq \sigma(x) = 3$ holds, we apply rule 10 and enter the loop with probability 1. From there a transition is created that splits the graph based on the probabilistic choice rule with probability $1/2$ each. The next two locations both assign a new value to x as described in rule 3. Notice that while the program `reward 1` is identical in both successors, they still remain separate, since the value of x after the assignment differs based on the previous state. From both locations we return to the program `while(2 ≤ x)`, with x reduced by 1 or 2 respectively, compared to state σ in the beginning. Both transitions are also labeled with 1 in a box to represent the reward function from [Definition 9](#). Now on the left side, another loop is executed, since the condition still holds. On the right side, the other transition from rule 10 is applied and we end in a final configuration with program state $\sigma' = \sigma[x \mapsto 1]$. Lastly, the transitions in rule 1 lead from the final configuration to the sink and from the sink to itself.

Notice that the operational MDP differs from the representation of [Figure 2b](#)'s example code in [Figure 2a](#). Since the operational MDP contains the program states in its configurations, there is not just one configuration for every program statement but rather a different one for every program statement and variable assignment combination. This results in a while loop not necessarily looping back to the same location but rather creating a long (possibly infinite) path and in general, allowing for infinitely many configurations (this is not the case in our example though, since it terminates for every possible path). Furthermore, the MDP depends on the input state σ .

For our translation, this is a big problem, since we need a finite representation of a pGCL

program for our translation, that does not depend on a specific input state, but functions the same for every σ . We will address this problem in [Chapter 3](#) by introducing symbolic MDPs - our formal representation of the JANI model, mentioned in [Chapter 1](#). This symbolic MDP is constructed similarly to the operational MDP but is modified in a way that supports our needs.

2.5 HeyVL

Caesar relies on its own intermediate verification language (IVL) HeyVL as its input format [\[SBK⁺23\]](#). HeyVL is designed to encode probabilistic verification problems. That means instead of being a probabilistic programming language itself, it rather acts as an intermediate language that different probabilistic languages (like pGCL) can be translated to for verification. A key aspect of HeyVL is that it is based on its own real-valued first-order logic HeyLo. This allows reasoning about direct quantities such as specific probabilities or expected values, opposed to Boolean first-order logic that traditional IVLs use and that only allow reasoning about Boolean predicates [\[SBK⁺23\]](#). HeyLo can then again be transferred into a problem for an SMT Solver to verify the program for all possible inputs.

A HeyVL program consists of one or multiple *(co)procedures*. Such a (co)procedure consists of a body S , a pre-expectation ϕ and a post-expectation ψ , where S is essentially a probabilistic program (which may contain additional verification statements) and ϕ and ψ are HeyLo formulae, which for simplification purposes can be thought of as expressions over the variables.

2.5.1 Post-Expectations

The post-expectation has an influence on the expected reward of the program. At termination it is collected additionally to the previous rewards. To account for this in our translation, we extend the operational reward function rew_{pGCL} from [Definition 9](#) by the post-expectation according to [\[BKMW24\]](#):

Definition 10 (Reward Function induced by Postexpectation). Let ψ be the post-expectation of a program, then we define the reward function $\text{rew}_\psi : \mathcal{GT} \rightarrow \mathbb{N}$ of the operational MDP $\mathcal{M}_{\langle C_0, \sigma_0 \rangle}$ as

$$\begin{aligned} \text{rew}_\psi(\langle C, \sigma \rangle, \alpha, p, \langle C', \sigma' \rangle) &= \begin{cases} \psi(\sigma) & \text{if } C = \Downarrow \\ \text{rew}_{\text{pGCL}}(\langle C, \sigma \rangle, \alpha, p, \langle C', \sigma' \rangle) & \text{else.} \end{cases} \\ &= \begin{cases} \psi(\sigma) & \text{if } C = \Downarrow \\ r(\sigma) & \text{if else } C = \text{reward } r \\ & \text{or } C = \text{reward } r; C' \\ 0 & \text{else.} \end{cases} \end{aligned}$$

For example, if $\psi = x$ and the program terminates in state σ with $\sigma(x) = 5$, then the final transition from $\langle \Downarrow, \sigma \rangle$ to $\perp$ collects reward $\sigma(x) = 5$ in addition to all rewards collected during execution.

2.5.2 Pre-Expectations

The central concept in HeyVL's verification model is the *pre-expectation*. Informally, given a program C and a post-expectation ψ , the *weakest pre-expectation* $\text{ert}\llbracket C \rrbracket(\psi)$ maps every initial state σ_0 to the expected value of ψ accumulated over the execution of C on σ . This includes any rewards collected along the way. An expression ϕ is a valid pre-expectation for coprocs if $\phi(\sigma_0) \geq \text{ert}\llbracket C \rrbracket(\psi)(\sigma_0)$ holds for all states σ . This means the pre-expectation has to be an upper bound on this expected value. For our MDP semantics, this is exactly the maximal expected reward from [Definition 7](#) for operational MDPs: $\text{ert}\llbracket C \rrbracket(\psi)(\sigma_0) = \text{MaxER}(\mathcal{M}_{\langle C, \sigma_0 \rangle}, \text{rew}_\psi)$ [SBK⁺23].

For runtime analysis, the post-expectation is set to $\psi = 0$, so $\text{ert}\llbracket C \rrbracket(0)(\sigma_0)$ is simply the expected sum of rewards collected during execution, i.e., the *expected runtime*. This corresponds to $\text{MaxER}(\mathcal{M}_{\langle C, \sigma_0 \rangle}, \text{rew}_{\text{pGCL}})$, since $\text{rew}_0 = \text{rew}_{\text{pGCL}}$ by definition.

2.5.3 (Co)procedures

A procedure verifies if the expected value of the post ψ is *lower* bound by the pre ϕ , i.e., if $\phi(\sigma_0) \leq \text{ert}\llbracket C \rrbracket(\psi)(\sigma_0)$ for all possible input states σ_0 . A coprocedure, on the other hand, verifies an *upper* bound on the expected value of ψ , by checking if $\phi(\sigma_0) \geq \text{ert}\llbracket C \rrbracket(\psi)(\sigma_0)$ for all σ_0 . Since we are only concerned with upper bounds in this thesis, pre-expectations will always concern coprocedures for us.

The pGCL program from [Example 1](#) written in HeyVL as a coprocedure can be seen in [Example 7](#). Here, we want to verify that the **pre** $(2/3) \cdot \text{init_x}$ is an upper bound on the runtime, indicated by the **post** 0.

Example 7.

```

coproc linear(init_x: UInt) -> (x: UInt)
  pre (2/3) * init_x
  post 0
{
  var prob_choice: Bool = false
  x = init_x
  while 2 <= x {
    prob_choice = flip(1/2)
    if prob_choice {
      x = x - 1
    } else {
      x = x - 2
    }
    reward 1
  }
}

```

At first glance it looks a lot more complicated but we will quickly go through the differences and explain them, to see that it is the same program just written in a different way.

In the coproc head, we see the input variable `init_x` and the output variable `x`, both unsigned integers. The additional variable `init_x` is needed in HeyVL, because input variables cannot be altered. It is used to initialize `x` in the procedure body, which contains the actual probabilistic program. Another additional variable in the HeyVL program is `prob_choice`. It is needed to encode probabilistic choice, which isn't directly supported by HeyVL. The pGCL subprogram $\{ x := x - 1 \} [1/2] \{ x := x - 2 \}$ written in HeyVL is

```

prob_choice = flip(1/2)
if prob_choice {
    x = x - 1
} else {
    x = x - 2
}.

```

Here, `flip(1/2)` does the probabilistic choice and returns either **true** or **false**, each with a probability of 0.5. The result is then stored in the `prob_choice` variable and used in an if-condition to execute one of the two subprograms. Note that `flip` is the only distribution expression in HeyVL that can depend on variables and not just constants. However, as one can see in the above example, it can be encoded as a probabilistic choice in our [Definition 1](#) of pGCL, so we do not need to alter our definition of distribution expressions to account for it.

Every iteration of the loops body decreases x by an expected value of $1/2 \cdot 1 + 1/2 \cdot 2 = 3/2$ and the loop terminates when $x < 2$. This means the program requires an expected number of approximately $\frac{x}{3/2} = \frac{2}{3}$ iterations until termination. Since, x is set to `init_x` before the loop, the pre-expectation $2/3 * \text{init_x}$ as annotated in [Example 7](#) is therefore a tight upper bound on the expected runtime.

So, as one can see, the HeyVL program from [Example 7](#) results in exactly the same expected rewards on termination as the pGCL program from [Example 1](#), it just needs a few extra lines of code.

In addition to the programming statements from pGCL, HeyVL also supports verification statements such as *assert* or *assume* [\[SBK+23\]](#). However, in this thesis, we will only focus HeyVL's executable fragment that *includes* while loops but *excludes* verification statements, i.e., we will only look at programs that can also be written in pGCL, as discussed previously.

Chapter 3

Analyzing HeyVL programs with KoAT

Now we are back to our goal to feed a program written in HeyVL into KoAT to receive an upper bound on the expected runtime. For that, we develop a fully automated translation from a HeyVL program into the KoAT format in this chapter. For the formal definitions and proofs, we again use the pGCL encoding of HeyVL as defined in [Definition 1](#).

We will start by formally defining symbolic MDPs and Caesar’s automatic translation to said symbolic MDPs. They are our formal representation of the JANI model. We will then prove that Caesar’s translation correctly upper bounds the expected rewards of the original program.

We go on to introduce KoAT and its input format, before defining our translation to said format and proving that it correctly maintains the expected rewards of symbolic MDPs. In the end, this will give us a full definition and proof of correctness for our entire translation pipeline as depicted in [Figure 1](#).

3.1 The JANI Format

First, we need a formal representation of the probabilistic input program, which we can then translate into a model representing the KoAT format. As we saw in [Section 2.2](#) an MDP can be a suitable representation for a probabilistic program. We also saw how to derive the operational MDP of any pGCL program in [Section 2.4](#). However, we also already addressed a few of its issues there. The operational MDP stores the program states, i.e., the variables’ values, in its locations, resulting in possibly infinite locations and a different MDP for every input state σ . That is why we will introduce the so-called JANI model in this section, a format that represents MDPs and other automata encoded in json [\[Hen18, jm\]](#). Thankfully, Caesar already features a translation from HeyVL into JANI [\[Verb\]](#), which returns Markov Chains or Markov Decision Processes in this format. While originally intended for model checkers like Storm [\[Hen18\]](#), we can make use of this structure in our translation, since it is very similar to the way KoAT programs are encoded. Since there is no formal definition for the translation from HeyVL into JANI yet, we will formally define the specific model that results from the translation. Moreover, we will prove that its expected reward upper bounds that of the program.

3.1.1 Symbolic MDPs

We first define the general concept of *symbolic MDPs* that represent the way MDPs are encoded in JANI generally, before focusing on the symbolic MDP that results from Caesar's translation in [Definition 15](#).

Definition 11 (Symbolic MDP).¹

We define a symbolic MDP (SMDP) as

$$\mathcal{SM} = (\mathcal{L}, \text{Vars}, \mathcal{Act}, \mathcal{GT}, l_{init})$$

where

- $\mathcal{L}$ is the set of locations,
- Vars is the set of variables,
- $\mathcal{Act}$ is the set of actions,
- and $\mathcal{GT}$ is the set of general transitions. A general transition $g \in \mathcal{GT}$ is a finite non-empty set of transitions. A transition t is here defined as $t = (l, \alpha, p, \varphi, \eta, l')$ where
 - $l \in \mathcal{L}$ is the start location and $l' \in \mathcal{L}$ is the destination of the transition,
 - $\alpha \in \mathcal{Act}$ is the transition's action,
 - $p : \Sigma \rightarrow [0, 1] \cap \mathbb{Q}$ is its probability expression mapping states to probabilities,
 - $\varphi : \Sigma \rightarrow \{\mathbf{true}, \mathbf{false}\}$ is a Boolean expression over Vars , representing the transition's guard,
 - $r : \Sigma \rightarrow \mathbb{N}$ is a non-negative polynomial reward expression, representing the reward of the transition,
 - $\eta : \text{Vars} \rightarrow \mathbb{Z}[\text{Vars}]$ is a variable update function that maps every variable to a polynomial expression over the set of variables. We also write $\eta(\sigma) = \sigma'$, where each variable's polynomial expression is evaluated according to the values in σ and the results are stored in σ' .
- l_{init} is the initial location.

Instead of $t = (l, \alpha, p, \varphi, r, \eta, l')$, we also use a similar notation as in [Section 2.4](#) and write $l \xrightarrow{\alpha, p, \varphi, r, \{x := E, y := F, \dots\}} l'$, where the set $\{x := E, y := F, \dots\}$ represents η and $x := E$ is written when $\eta(x) = E$. We omit most parts of the transition t if possible, as to not clutter the notation unnecessarily. When only some attributes are given explicitly, it will be clear from context which are meant.

¹We use the symbolic Markov automaton from Definition 20 in [\[Hen18\]](#) as a rough template for our symbolic MDP but adjust it for our purposes.

If omitted, the following values are implied for the attributes:

- $\alpha = N$
- $p = 1$
- $\varphi = \mathbf{true}$
- $r = 0$
- $x := x$ for every variable $x \in Vars$

Since the transitions of a symbolic MDP contain expressions that depend on the program state, we need to slightly redefine paths and reward functions on SMDPs. To include program states, we again define a set of configurations, similar as for the operational MDP in [Section 2.4](#). We define it as a combination between locations from $\mathcal{L}$ and program states from Σ :

$$\mathbf{Conf}_{\text{symp}} = \mathcal{L} \times \Sigma$$

Definition 12 (Symbolic MDP Path). Let $\mathcal{SM} = (\mathcal{L}, Vars, Act, l_{init}, \mathcal{GT})$ be a symbolic MDP

1. Then we define a path on $\mathcal{SM}$ not just between locations, but between configurations $\langle l, \sigma \rangle \in \mathbf{Conf}_{\text{symp}}$:

$$\pi = \langle l_0, \sigma_0 \rangle \xrightarrow{t_0} \dots \xrightarrow{t_{n-1}} \langle l_n, \sigma_n \rangle$$

where for each $t_i = (l_i, \alpha_i, p_i, \varphi_i, r_i, \eta_i, l'_i)$ we have $\varphi_i(\sigma_i) = \mathbf{true}$ and $\eta_i(\sigma_i) = \sigma_{i+1}$. This is needed, because for specific program runs, we need to keep track of the variables' values, which are not encoded in SMDP locations, like they were for operational MDPs.

2. We thus also define the set of all paths that begin in location l_0 with program state σ_0 as

$$\mathbf{Paths}_{\mathcal{SM}}^n(l_0, \sigma_0) = \{\pi \mid \pi = \langle l_0, \sigma_0 \rangle \xrightarrow{t_0} \dots \xrightarrow{t_{n-1}} \langle l_n, \sigma_n \rangle \text{ is a path in } \mathcal{M}\}.$$

3. The set $\mathbf{Paths}_{\mathcal{SM}}(l_0, \sigma_0)$ is defined analogously to [Definition 3](#). We then write $\mathbf{Paths}_{\mathcal{SM}}^n(\sigma_0)$ instead of $\mathbf{Paths}_{\mathcal{SM}}^n(l_{init}, \sigma_0)$ and $\mathbf{Paths}_{\mathcal{SM}}(\sigma_0)$ instead of $\mathbf{Paths}_{\mathcal{SM}}(l_{init}, \sigma_0)$.
4. A symbolic scheduler $\mathfrak{S} : \mathbf{Conf}_{\text{symp}} \rightarrow Act$ now also takes configurations $\langle l, \sigma \rangle$ as input, depending not just on the location but also on the current program state.
5. We define the path probability for π , with $t_i = (l_i, \mathfrak{S}(\langle l_i, \sigma_i \rangle), p_i, \varphi_i, r_i, \eta_i, l'_i)$ and symbolic scheduler $\mathfrak{S}$ as

$$\mathbf{Prob}_{\mathfrak{S}}(\pi) = \prod_{i=0}^{n-1} p_i(\sigma_i).$$

With that, we can define the reward function of a symbolic MDP.

Definition 13 (Symbolic Reward Function). Let RE be the set of all reward expressions $r : \Sigma \rightarrow \mathbb{N}$. Then we define *the* reward function $\text{rew}_{\mathcal{SM}} : \mathcal{GT} \rightarrow RE$ of a symbolic MDP $\mathcal{SM} = (\mathcal{L}, \text{Vars}, \text{Act}, l_{\text{init}}, \mathcal{GT})$ as

$$\text{rew}_{\mathcal{SM}}(l, \alpha, p, \varphi, r, \eta, l') = r$$

It maps every transition to its reward expression r . The reward of a path is then

$$\text{rew}_{\mathcal{SM}}(\langle l_0, \sigma_0 \rangle \xrightarrow{t_0} \dots \xrightarrow{t_{n-1}} \langle l_n, \sigma_n \rangle) = \sum_{i=0}^{n-1} \text{rew}_{\mathcal{SM}}(t_i)(\sigma_i) = \sum_{i=0}^{n-1} r_i(\sigma_i),$$

where each transition's reward expression r_i is evaluated according to the program state σ_i .

Definition 14 (Expected Rewards for symbolic MDPs). The expected rewards for symbolic MDPs are defined analogously to [Definition 7](#). The only difference is that here the initial state σ also needs to be included, so

$$\text{ER}_{\mathfrak{S}}^n(\mathcal{SM}, l, \sigma, \text{rew}) = \sum_{\pi \in \text{Paths}_{\mathfrak{S}}^n(l, \sigma)} \text{Prob}_{\mathfrak{S}}(\pi) \cdot \text{rew}(\pi).$$

and all other definitions analogously. We omit $\text{rew}_{\mathcal{SM}}$ in the arguments when it is implied.

3.1.2 The JANI Translation

The specific symbolic MDP that Caesar creates from an input program and encodes in the JANI model is defined like this:

Definition 15 (Symbolic MDP of a pGCL program). Let $C \in \text{pGCL}$ be a pGCL program with post-expectation ψ . Then we define *the* symbolic MDP of C as

$$\mathcal{SM}_C = (\mathcal{L}_C, \text{Vars}_C, \text{Act}, l_{\text{init}}, \mathcal{GT}_C)$$

with

- $\mathcal{L}_C$ is the set of locations induced by $\mathcal{GT}_C$
- Vars_C is the set of all variables that appear in C ,
- $\text{Act} = \{L, N, R\}$,
- $l_{\text{init}} = l_0$,
- $\mathcal{GT}_C = \text{trns}(l_0, C, \Downarrow) \cup \{\{\Downarrow \xrightarrow{r=\psi} \perp\}\}$ with trns defined in [Figure 5](#)

We define here the trns function that represents the algorithmic approach to how the transitions in Caesar's translation are built recursively in the actual implementation. We will see that trns as defined here, is very similar to the rules from [Figure 3](#). Essentially there have only been modifications in the output model to make it finite and independent of specific program states. However, a key difference is that the transitions defined by trns include a guard, a probability expression, a reward expression and an update function, as the effects of these cannot be encoded in the locations anymore, without a concrete program state. Instead, they are evaluated over paths like we saw in [Definition 12](#)

Let $GenTrans$ be the set of all possible general transitions. We then define the recursive function $trns l : \mathcal{L} \times \mathbf{pGCL} \times \mathcal{L} \rightarrow \mathcal{P}(GenTrans)$ for all program statements as following:

1. Skip:

$$trns l(l, \text{skip}, l') = \{\{l \rightarrow l'\}\}$$

2. Assignment:

$$trns l(l, x := E, l') = \{\{l \xrightarrow{\{x:=E\}} l'\}\}$$

3. Random Assignment:

$$trns l(l, x \approx \mu, l') = \{\{l \xrightarrow{p=\mu(v), \{x:=v\}} l'\} \mid \forall v \in Vals : \mu(v) > 0\}$$

4. Reward:

$$trns l(l, \text{reward } r, l') = \{\{l \xrightarrow{r} l'\}\}$$

5. Sequential Composition:

$$\begin{aligned} trns l(l, C_1; C_2, l'') &= GT_1 \cup GT_2 \quad \text{where} \\ GT_1 &= trns l(l, C_1, l'), \\ GT_2 &= trns l(l', C_2, l''), \\ l' &\text{ is a fresh location} \end{aligned}$$

6. Probabilistic Choice:

$$\begin{aligned} \text{if } C_1 \neq C_2 : trns l(l, \{ C_1 \} [p] \{ C_2 \}, l') &= \{\{l \xrightarrow{p} l_1, l \xrightarrow{1-p} l_2\}\} \cup GT_1 \cup GT_2 \quad \text{where} \\ GT_1 &= trns l(l_1, C_1, l'), \\ GT_2 &= trns l(l_2, C_2, l'), \\ l_1, l_2 &\text{ are fresh locations} \\ \text{else } : trns l(l, \{ C_1 \} [p] \{ C_1 \}, l'') &= \{\{l \rightarrow l'\}\} \cup GT \quad \text{where} \\ GT &= trns l(l', C_1, l'') \\ l' &\text{ is a fresh location} \end{aligned}$$

7. Nondeterministic Choice:

$$\begin{aligned} trns l(l, \{ C_1 \} \square \{ C_2 \}, l') &= \{\{l \xrightarrow{L} l_1\}, \{l \xrightarrow{R} l_2\}\} \cup GT_1 \cup GT_2 \quad \text{where} \\ GT_1 &= trns l(l_1, C_1, l'), \\ GT_2 &= trns l(l_2, C_2, l'), \\ l_1, l_2 &\text{ are fresh locations} \end{aligned}$$

8. Conditional Choice:

$$\begin{aligned} trns l(l, \text{if } (\varphi) \{ C_1 \} \text{ else } \{ C_2 \}, l') &= \{\{l \xrightarrow{\varphi} l_1\}, \{l \xrightarrow{\neg\varphi} l_2\}\} \cup GT_1 \cup GT_2 \quad \text{where} \\ GT_1 &= trns l(l_1, C_1, l'), \\ GT_2 &= trns l(l_2, C_2, l'), \\ l_1, l_2 &\text{ are fresh locations} \end{aligned}$$

9. While Loop:

$$\begin{aligned} trns l(l, \text{while } (\varphi) \{ C \}, l'') &= \{\{l \xrightarrow{\varphi} l'\}, \{l \xrightarrow{\neg\varphi} l''\}\} \cup GT \quad \text{where} \\ GT &= trns l(l', C, l), \\ l' &\text{ is a fresh location} \end{aligned}$$

Figure 5: Definition of the Function $trns l$

and [Definition 13](#), because the path configurations add the program states necessary for evaluation.

The *trnsf* function works as follows: It takes a starting location l , a pGCL program C and a destination location l' and returns a set of general transitions that form a path from l to l' while encoding the semantics of C in the transitions.

The explanation behind the function is relatively intuitive and follows the same logic as the small-step relation rules from [Figure 3](#), so will go over them more quickly:

The *skip* just creates an "empty" transition from l to l' , i.e., a transition where all expressions have their default value as defined in [Definition 11](#) and nothing gets altered.

Assignment and *reward* act similarly but add an assignment function or reward expression, respectively, that corresponds to the program's intended semantics.

Random Assignment creates a single general transition containing multiple transitions that each set x to a different value v with probability $\mu(v)$. Since μ is discrete, this set of transitions is finite.

For the *sequential composition* between locations l and l'' , a fresh location l' is created as a midway point and *trnsf* is called recursively to first create transitions for program C_1 from l to l' and then go on from l' to l'' according to C_2 .

For *probabilistic choice*, *trnsf* creates two fresh locations l_1 and l_2 and then transitions from l to l_1 and l_2 with the respective probability expressions p and $1-p$. These transitions are grouped together in one general transition, because they have the same start location, action, guard and reward. Additionally to those two transitions, *trnsf* is again called recursively from l_1 and l_2 with programs C_1 and C_2 and destination location l' .

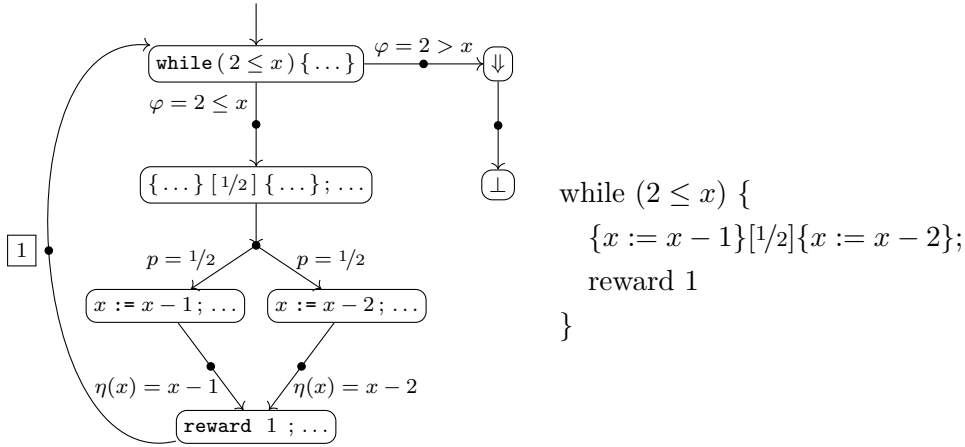
Nondeterministic choice and *conditional choice* are treated very similarly to probabilistic choice, the only difference being that the transitions are obviously labeled with the correct actions or guards instead of probability expressions. They also each create an own general transition for each transition, because the transitions have different actions or guards, respectively, assigned and thus need to be in their own general transitions.

Finally, for the *while loop* with start l and destination l'' , *trnsf* first does a conditional choice based on its guard. If it is not satisfied, program C can be skipped, and an immediate transition to l'' is created. For the other case, we create a transition to a fresh location l' and then call *trnsf* recursively with l as destination, because after executing C , we want to loop back to the starting state l , of course.

Example 8 (Symbolic MDP of a pGCL program). The example in [Figure 6](#) shows the symbolic MDP for our main [Example 1](#). We label the locations with the programs they represent, i.e., the program that will be executed from this point on. We only label the transitions when attributes differ from the default values defined in [Definition 11](#), i.e., if left out, the values $\alpha = N$, $p = 1$, $\varphi = \mathbf{true}$, $r = 0$ and $\eta(x) = x$ are implied.

As we can see, the symbolic MDP's structure is similar to the operational MDP from [Figure 4](#). However, there are also key differences, which result from the fact that the symbolic MDP does not depend on program states. For example, there is only one location for every program, so all *reward 1* locations and all *while*($2 \leq x$) locations, were merged, respectively. The edges now also need to include information like the guard φ and variable assignment η , since there is no program state σ stored in the locations that can imply which path needs to be taken, or what effects a path has.

Because every program has only one location and we only consider finite programs, our symbolic MDP of a program is also always finite. In contrast, as mentioned previously, the operational MDP can have an infinite number of locations through a while loop that creates a new location in every iteration. Here, as we can see in the graph, the symbolic

Figure 6: $\mathcal{SM}_C$ for Example Program C from Example 1

MDP actually *loops* back to the original while location instead of creating new ones for every possible variable assignment.

To show that the expected rewards of the symbolic MDP of a program are the same as the expected rewards of its operational MDP, we will first prove the following lemma, which states that the reward and probability of a path remain the same after the Caesar translation.

Lemma 1 (Expected Reward Remains). Let $\mathcal{M}_{\langle C_0, \sigma_0 \rangle} = (\text{Conf}, \{L, N, R\}, \mathcal{GT}_{\mathcal{M}}, \langle C_0, \sigma_0 \rangle)$ be the operational MDP of a pGCL program C_0 with starting state σ_0 and post-expectation ψ . Let rew_{ψ} be its reward function.

Let $\mathcal{SM}_{C_0} = (\mathcal{L}, \text{Vars}, \{L, N, R\}, \mathcal{GT}_{\mathcal{SM}}, l_0)$ be the symbolic MDP of C_0 with reward function $\text{rew}_{\mathcal{SM}_{C_0}}$.

Let $\mathfrak{S} : \text{Conf} \rightarrow \{L, N, R\}$ be a scheduler for $\mathcal{M}_{\langle C_0, \sigma_0 \rangle}$ and $\mathfrak{S}' : \mathcal{L} \times \Sigma \rightarrow \{L, N, R\}$ a symbolic scheduler for $\mathcal{SM}_{C_0}$ with $\mathfrak{S}'(\langle l_C, \sigma \rangle) = \mathfrak{S}(\langle C, \sigma \rangle)$ for every program $C \in \text{pGCL}$ and state $\sigma \in \Sigma$.

Then there exists a mapping $f : \text{Paths}_{\mathcal{M}_{\langle C_0, \sigma_0 \rangle}} \rightarrow \text{Paths}_{\mathcal{SM}_{C_0}}(\sigma_0)$ that maps every $\pi \in \text{Paths}_{\mathcal{M}_{\langle C_0, \sigma_0 \rangle}}$ to a path $f(\pi) = \pi' \in \text{Paths}_{\mathcal{SM}_{C_0}}(\sigma_0)$ with

1. the same path probability: $\text{Prob}_{\mathfrak{S}}(\pi) = \text{Prob}_{\mathfrak{S}'}(\pi')$,
2. and the same reward: $\text{rew}_{\psi}(\pi) = \text{rew}_{\mathcal{SM}_{C_0}}(\pi')$.

Proof.

We first define $\tau : \mathcal{T}_1 \rightarrow \mathcal{T}_2$ in Figure 7 on page 28 as a mapping of singular transitions from $\mathcal{M}_{\langle C_0, \sigma_0 \rangle}$ to $\mathcal{SM}_{C_0}$, where $\mathcal{T}_1$ is the set of transitions of $\mathcal{M}_{\langle C_0, \sigma_0 \rangle}$ and $\mathcal{T}_2$ is the set of transitions of $\mathcal{SM}_{C_0}$. We reason why τ is well-defined in Figure 7, by explaining why each transition that it maps to exists in $\mathcal{T}_2$.

We then create an injective function $f : \text{Paths}_{\mathcal{M}_{\langle C_0, \sigma_0 \rangle}} \rightarrow \text{Paths}_{\mathcal{SM}_{C_0}}(\sigma_0)$ that uses τ to map each path π in $\mathcal{M}_{\langle C_0, \sigma_0 \rangle}$ to an equivalent path π' in $\mathcal{SM}_{C_0}$ that encodes the same program run on C_0 . Then we show that 1. $\text{Prob}_{\mathfrak{S}}(\pi) = \text{Prob}_{\mathfrak{S}'}(\pi')$ and 2. $\text{rew}_{\psi}(\pi) = \text{rew}_{\mathcal{SM}_{C_0}}(\pi')$ hold for every path length n , by induction. We define f as

$$f(\langle C_0, \sigma_0 \rangle \xrightarrow{t_0} \langle C_1, \sigma_1 \rangle \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} \langle C_n, \sigma_n \rangle) = \langle l_{C_0}, \sigma_0 \rangle \xrightarrow{\tau(t_0)} \langle l_{C_1}, \sigma_1 \rangle \xrightarrow{\tau(t_1)} \dots \xrightarrow{\tau(t_{n-1})} \langle l_{C_n}, \sigma_n \rangle.$$

1. Final Configuration:

$$\tau(\langle \Downarrow, \sigma \rangle \xrightarrow{N,1} \perp) = \Downarrow \xrightarrow{r=\psi} \perp$$

This transition always exists, because it is part of [Definition 15](#).

2. Skip: $C = \text{skip}$

$$\tau(\langle C, \sigma \rangle \xrightarrow{N,1} \langle C', \sigma \rangle) = l_C \rightarrow l_{C'}$$

Exists, because $\{l_C \rightarrow l_{C'}\} \in \text{trns}(l_C, \text{skip}, l_{C'})$.

3. Assignment: $C = x := E$

$$\tau(\langle C, \sigma \rangle \xrightarrow{N,1} \langle C', \sigma[x \mapsto E(\sigma)] \rangle) = l_C \xrightarrow{\{x:=E\}} l_{C'}$$

Exists, because $\{l_C \xrightarrow{\{x:=E\}} l_{C'}\} \in \text{trns}(l_C, x := E, l_{C'})$.

4. Random Assignment: $C = x \approx \mu$

$$\tau(\langle C, \sigma \rangle \xrightarrow{N, \mu(v)} \langle C', \sigma[x \mapsto v] \rangle) = l_C \xrightarrow{p=\mu(v), \{x:=v\}} l_{C'}$$

Exists, because $\{l_C \xrightarrow{p=\mu(v), \{x:=v\}} l_{C'}\} \in \text{trns}(l_C, x \approx \mu, l_{C'})$.

5. Reward: $C = \text{reward } r$

$$\tau(\langle C, \sigma \rangle \xrightarrow{N,1} \langle C', \sigma \rangle) = l_C \xrightarrow{r} l_{C'}$$

Exists, because $\{l_C \xrightarrow{r} l_{C'}\} \in \text{trns}(l_C, \text{reward } r, l_{C'})$.

6. Sequential Composition: $C = C_1; C_2$

$$\begin{aligned} \tau(\langle C_1; C_2, \sigma \rangle \xrightarrow{\alpha, p(\sigma)} \langle C_2, \sigma' \rangle) &= l_C \xrightarrow{\alpha, p, \eta(\sigma)=\sigma'} l_{C_2} \\ \tau(\langle C_1; C_2, \sigma \rangle \xrightarrow{\alpha, p(\sigma)} \langle C'_1; C_2, \sigma' \rangle) &= l_C \xrightarrow{\alpha, p, \eta(\sigma)=\sigma'} l_{C'_1; C_2} \end{aligned}$$

Exist, because $\text{trns}(l_C, C_1; C_2, l_{C'})$ recursively calls $\text{trns}(l_C, C_1, l_{C_2})$ or $\text{trns}(l_C, C_1, l_{C'_1; C_2})$, respectively, for which C_1 then again is one of the cases 1.-10. described here and thus creates the correct transition, as shown for all other respective cases. If C_1 is again a sequential composition, trns will simply be called until one of the other cases is reached, because we only consider finite programs.

7. Probabilistic Choice: $C = \{C_1\} [p] \{C_2\}$

$$\begin{aligned} \tau(\langle \{C_1\} [p] \{C_2\}, \sigma \rangle \xrightarrow{N, p(\sigma)} \langle C_1, \sigma \rangle) &= l_C \xrightarrow{p} l_{C_1} && \text{for } C_1 \neq C_2 \\ \tau(\langle \{C_1\} [p] \{C_2\}, \sigma \rangle \xrightarrow{N, 1-p(\sigma)} \langle C_2, \sigma \rangle) &= l_C \xrightarrow{1-p} l_{C_2} && \text{for } C_1 \neq C_2 \\ \tau(\langle \{C_1\} [p] \{C_1\}, \sigma \rangle \xrightarrow{N,1} \langle C_1, \sigma \rangle) &= l_C \rightarrow l_{C_1} \end{aligned}$$

Exist, because $\{l_C \xrightarrow{p} l_{C_1}, l_C \xrightarrow{1-p} l_{C_2}\} \in \text{trns}(l_C, \{C_1\} [p] \{C_2\}, l_{C'})$ for $C_1 \neq C_2$ and $\{l_C \rightarrow l_{C_1}\} \in \text{trns}(l_C, \{C_1\} [p] \{C_1\}, l_{C'})$.

8. Nondeterministic Choice: $C = \{C_1\} \sqcap \{C_2\}$

$$\begin{aligned} \tau(\langle \{C_1\} \sqcap \{C_2\}, \sigma \rangle \xrightarrow{L,1} \langle C_1, \sigma \rangle) &= l_C \xrightarrow{L} l_{C_1} \\ \tau(\langle \{C_1\} \sqcap \{C_2\}, \sigma \rangle \xrightarrow{R,1} \langle C_2, \sigma \rangle) &= l_C \xrightarrow{R} l_{C_2} \end{aligned}$$

Exist, because $\{l_C \xrightarrow{L} l_{C_1}, l_C \xrightarrow{R} l_{C_2}\} \in \text{trns}(l_C, \{C_1\} \sqcap \{C_2\}, l_{C'})$.

9. Conditional Choice: $C = \text{if } (\varphi) \{C_1\} \text{ else } \{C_2\}$

$$\begin{aligned} \tau(\langle \text{if } (\varphi) \{C_1\} \text{ else } \{C_2\}, \sigma \rangle \xrightarrow{N,1} \langle C_1, \sigma \rangle) &= l_C \xrightarrow{\varphi} l_{C_1} \\ \tau(\langle \text{if } (\varphi) \{C_1\} \text{ else } \{C_2\}, \sigma \rangle \xrightarrow{N,1} \langle C_2, \sigma \rangle) &= l_C \xrightarrow{\neg\varphi} l_{C_2} \end{aligned}$$

Exist, because $\{l_C \xrightarrow{\varphi} l_{C_1}, l_C \xrightarrow{\neg\varphi} l_{C_2}\} \in \text{trns}(l_C, \text{if } (\varphi) \{C_1\} \text{ else } \{C_2\}, l_{C'})$.

10. While Loop: $C = \text{while } (\varphi) \{C\}$

$$\begin{aligned} \tau(\langle \text{while } (\varphi) \{C\}, \sigma \rangle \xrightarrow{N,1} \langle C_1; \text{while } (\varphi) \{C\}, \sigma \rangle) &= l_C \xrightarrow{\varphi} l_{C_1; C} \\ \tau(\langle \text{while } (\varphi) \{C\}, \sigma \rangle \xrightarrow{N,1} \langle C', \sigma \rangle) &= l_C \xrightarrow{\neg\varphi} l_{C'} \end{aligned}$$

Exist, because $\{l_C \xrightarrow{\varphi} l_{C_1; C}, l_C \xrightarrow{\neg\varphi} l_{C'}\} \in \text{trns}(l_C, \text{while } (\varphi) \{C\}, l_{C'})$.

Now we prove the lemma by induction over the path length n .

Base Case. For $n = 1$, let $\pi = \langle C_0, \sigma_0 \rangle \xrightarrow[t_0]{} \langle C_1, \sigma_1 \rangle \in \text{Paths}_{\mathcal{M}_{\langle C_0, \sigma_0 \rangle}}^1$ with p_0 being the probability expression of t_0 .

1.

$$\begin{aligned} \text{Prob}_{\mathfrak{S}}(\pi) &= \text{Prob}_{\mathfrak{S}}(\langle C_0, \sigma_0 \rangle \xrightarrow[t_0]{} \langle C_1, \sigma_1 \rangle) \\ &= \begin{cases} p(\sigma_0) & \text{if } C_0 = \{C_1\}[p]\{C'\} \\ 1 - p(\sigma_0) & \text{else if } C_0 = \{C'\}[p]\{C_1\} \\ p_0(\sigma_0) & \text{else if } C_0 = x : \approx \mu \\ & \text{or } C_0 = C'_1; C'_2 \\ 1 & \text{else.} \end{cases} \end{aligned}$$

$$\begin{aligned} \text{Prob}_{\mathfrak{S}'}(\pi') &= \text{Prob}_{\mathfrak{S}'}(f(\pi)) \\ &= \text{Prob}_{\mathfrak{S}'}(\langle l_{C_0}, \sigma_0 \rangle \xrightarrow[\tau(t_0)]{} \langle l_{C_1}, \sigma_1 \rangle) \\ &= \begin{cases} \text{Prob}_{\mathfrak{S}'}(\langle l_{C_0}, \sigma_0 \rangle \xrightarrow{p} \langle l_{C_1}, \sigma_1 \rangle) & \text{if } C_0 = \{C_1\}[p]\{C'\} \\ \text{Prob}_{\mathfrak{S}'}(\langle l_{C_0}, \sigma_0 \rangle \xrightarrow{1-p} \langle l_{C_1}, \sigma_1 \rangle) & \text{else if } C_0 = \{C'\}[p]\{C_1\} \\ \text{Prob}_{\mathfrak{S}'}(\langle l_{C_0}, \sigma_0 \rangle \xrightarrow{p_0} \langle l_{C_1}, \sigma_1 \rangle) & \text{else if } C_0 = x : \approx \mu \\ & \text{or } C_0 = C'_1; C'_2 \\ \text{Prob}_{\mathfrak{S}'}(\langle l_{C_0}, \sigma_0 \rangle \xrightarrow{1} \langle l_{C_1}, \sigma_1 \rangle) & \text{else.} \end{cases} \\ &= \begin{cases} p(\sigma_0) & \text{if } C_0 = \{C_1\}[p]\{C'\} \\ 1 - p(\sigma_0) & \text{else if } C_0 = \{C'\}[p]\{C_1\} \\ p_0(\sigma_0) & \text{else if } C_0 = x : \approx \mu \\ & \text{or } C_0 = C'_1; C'_2 \\ 1 & \text{else.} \end{cases} \end{aligned}$$

$$\Rightarrow \text{Prob}_{\mathfrak{S}}(\pi) = \text{Prob}_{\mathfrak{S}'}(f(\pi)) \text{ for all } \pi \in \text{Paths}_{\mathcal{M}_{\langle C_0, \sigma_0 \rangle}}^1$$

2.

$$\begin{aligned} \text{rew}_{\psi}(\pi) &= \text{rew}_{\psi}(\langle C_0, \sigma_0 \rangle \xrightarrow[t_0]{} \langle C_1, \sigma_1 \rangle) \\ &= \text{rew}_{\psi}(t_0) \\ &= \begin{cases} \psi(\sigma_0) & \text{if } C_0 = \Downarrow \\ r(\sigma_0) & \text{if else } C_0 = \text{reward } r \\ & \text{or } C_0 = \text{reward } r; C_1 \\ 0 & \text{else.} \end{cases} \end{aligned}$$

$$\begin{aligned}
\text{rew}_{\mathcal{SM}_{C_0}}(\pi') &= \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi)) \\
&= \text{rew}_{\mathcal{SM}_{C_0}}(\langle l_{C_0}, \sigma_0 \rangle \xrightarrow[\tau(t_0)]{} \langle l_{C_1}, \sigma_1 \rangle) \\
&= \begin{cases} \text{rew}_{\mathcal{SM}_{C_0}}(\langle \Downarrow, \sigma_0 \rangle \xrightarrow{r=\psi} \langle \perp, \sigma_1 \rangle) & \text{if } C = \Downarrow \\ \text{rew}_{\mathcal{SM}_{C_0}}(\langle l_{C_0}, \sigma_0 \rangle \xrightarrow{r=r} \langle l_{C_1}, \sigma_1 \rangle) & \text{else if } C = \text{reward } r \\ & \text{or } C = \text{reward } r; C_1 \\ \text{rew}_{\mathcal{SM}_{C_0}}(\langle l_{C_0}, \sigma_0 \rangle \xrightarrow{r=0} \langle l_{C_1}, \sigma_1 \rangle) & \text{else.} \end{cases} \\
&= \begin{cases} \psi(\sigma_0) & \text{if } C = \Downarrow \\ r(\sigma_0) & \text{else if } C = \text{reward } r \\ & \text{or } C = \text{reward } r; C_1 \\ 0 & \text{else.} \end{cases}
\end{aligned}$$

$\Rightarrow \text{rew}_\psi(\pi) = \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi))$ for all $\pi \in \text{Paths}_{\mathcal{M}_{\langle C_0, \sigma_0 \rangle}}^1$

We first show the base case - a path of length $n = 1$ - for the probability. A singular transition in $\mathcal{M}_{\langle C_0, \sigma_0 \rangle}$ can be created by any of the rules in [Figure 3](#). The only transitions having a probability that is not 1 emerge from probabilistic choice, random assignment, or sequential composition. All other possible program statements are thus covered by the else case, while the other probabilities arise from the definitions of probabilistic choice and sequential composition.

For the probability of $f(\pi)$ we then simply plug in the definition for τ of all transitions with probabilities that are not p and then use the definition of path probabilities for symbolic MDPs to receive the same probability as for π .

For the reward of paths of length 1 we again consider the relevant cases, which in this case are only the transition where postcondition ψ is applied and reward statements. When plugging in the definitions of τ and of rewards of symbolic paths, we can see that the reward of π is the same as the one of $f(\pi)$.

Induction Hypothesis. Assume 1. $\text{Prob}_\Theta(\pi) = \text{Prob}_{\Theta'}(\pi')$ and 2. $\text{rew}_\psi(\pi) = \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi))$ hold for some arbitrary n , and $\pi \in \text{Paths}_{\mathcal{M}_{\langle C_0, \sigma_0 \rangle}}^n$.

Induction Step. Let the hypothesis hold for some n . Let $\pi = \langle C_0, \sigma_0 \rangle \xrightarrow{t_0} \dots \xrightarrow{t_n} \langle C_{n+1}, \sigma_{n+1} \rangle \in \text{Paths}_{\mathcal{M}_{\langle C_0, \sigma_0 \rangle}}^{n+1}$ and $\pi^{-1} = \langle C_0, \sigma_0 \rangle \xrightarrow{t_0} \dots \xrightarrow{t_{n-1}} \langle C_n, \sigma_n \rangle$.

1.

$$\begin{aligned}
&\text{Prob}_\Theta(\pi) \\
&= \text{Prob}_\Theta(\pi^-) \cdot p_n && \text{(Definition 5)} \\
&= \text{Prob}_{\Theta'}(f(\pi^-)) \cdot p_n && \text{(ind. hypothesis)} \\
&= \text{Prob}_{\Theta'}(f(\pi^-)) \cdot \text{Prob}_\Theta(\langle C_n, \sigma_n \rangle \xrightarrow{t_n} \langle C_{n+1}, \sigma_{n+1} \rangle) \\
&= \text{Prob}_{\Theta'}(f(\pi^-)) \cdot \begin{cases} p(\sigma_n) & \text{if } C_n = \{C_{n+1}\}[p]\{C'\} \\ 1 - p(\sigma_n) & \text{else if } C_n = \{C'\}[p]\{C_{n+1}\} \\ p_n(\sigma_n) & \text{else if } C_n = x : \approx \mu \\ & \text{or } C_n = C'_1; C'_2 \\ 1 & \text{else.} \end{cases}
\end{aligned}$$

$$\begin{aligned}
&= \text{Prob}_{\mathfrak{S}'}(f(\pi^-)) \cdot \begin{cases} \text{Prob}_{\mathfrak{S}'}(\langle l_{C_n}, \sigma_n \rangle \xrightarrow{p} \langle l_{C_{n+1}}, \sigma_{n+1} \rangle) & \text{if } C_n = \{C_{n+1}\} [p] \{C'\} \\ \text{Prob}_{\mathfrak{S}'}(\langle l_{C_n}, \sigma_n \rangle \xrightarrow{1-p} \langle l_{C_{n+1}}, \sigma_{n+1} \rangle) & \text{else if } C_n = \{C'\} [p] \{C_{n+1}\} \\ \text{Prob}_{\mathfrak{S}'}(\langle l_{C_n}, \sigma_n \rangle \xrightarrow{p_n} \langle l_{C_{n+1}}, \sigma_{n+1} \rangle) & \text{else if } C_n = x : \approx \mu \\ & \text{or } C_n = C'_1; C'_2 \\ \text{Prob}_{\mathfrak{S}'}(\langle l_{C_n}, \sigma_n \rangle \xrightarrow{1} \langle l_{C_{n+1}}, \sigma_{n+1} \rangle) & \text{else.} \end{cases} \\
&= \text{Prob}_{\mathfrak{S}'}(f(\pi^-)) \cdot \text{Prob}_{\mathfrak{S}'}(\langle l_{C_n}, \sigma_n \rangle \xrightarrow{\tau(t_n)} \langle l_{C_{n+1}}, \sigma_{n+1} \rangle) \\
&= \text{Prob}_{\mathfrak{S}'}(f(\pi))
\end{aligned}$$

2.

$$\begin{aligned}
&\text{rew}_{\psi}(\pi) \\
&= \text{rew}_{\psi}(\pi^-) + \text{rew}_{\psi}(t_n) && \text{(Definition 6)} \\
&= \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi^-)) + \text{rew}_{\psi}(t_n) && \text{(ind. hypothesis)} \\
&= \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi^-)) + \begin{cases} \psi(\sigma_n) & \text{if } C_n = \Downarrow \\ r(\sigma_n) & \text{else if } C_n = \text{reward } r \\ & \text{or reward } r; C_{n+1} \\ 0 & \text{else} \end{cases} && \text{(Definition 9)} \\
&= \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi^-)) + \begin{cases} \text{rew}_{\mathcal{SM}_{C_0}}(\langle \Downarrow, \sigma_n \rangle \xrightarrow{r=\psi} \langle \perp, \sigma_{n+1} \rangle) & \text{if } C_n = \Downarrow \\ \text{rew}_{\mathcal{SM}_{C_0}}(\langle l_{C_n}, \sigma_n \rangle \xrightarrow{r=r} \langle l_{C_{n+1}}, \sigma_{n+1} \rangle) & \text{else if } C_n = \text{reward } r \\ & \text{or } C_n = \text{reward } r; C_{n+1} \\ \text{rew}_{\mathcal{SM}_{C_0}}(\langle l_{C_n}, \sigma_n \rangle \xrightarrow{r=0} \langle l_{C_{n+1}}, \sigma_{n+1} \rangle) & \text{else.} \end{cases} \\
&= \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi^-)) + \text{rew}_{\mathcal{SM}_{C_0}}(\langle l_{C_n}, \sigma_n \rangle \xrightarrow{\tau(t_n)} \langle l_{C_{n+1}}, \sigma_{n+1} \rangle) && \text{(Definition 13)} \\
&= \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi^-)) + \text{rew}_{\mathcal{SM}_{C_0}}(f(t_n)) && \text{(Def. } f) \\
&= \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi)) && \text{(Definition 13)}
\end{aligned}$$

For the induction step of a path's probability, we can first split the probability of π up into the probability of truncated path π^- , multiplied by the the n 'th transition's probability p_n . We can then use the induction hypothesis to show that $\text{Prob}_{\mathfrak{S}}(\pi^-) = \text{Prob}_{\mathfrak{S}'}(f(\pi^-))$. For p_n we use the same case distinction as in the base case. In the last step, we can combine the definition of the probability for symbolic paths with the definition of f and come to the conclusion that the hypothesis holds for paths of length $n + 1$, if it holds for length n .

For rewards, we apply a very similar pattern, only here we add the reward of t_n up to the reward of π^- , since we defined rewards of a path over the sum of its transitions.

Thus we have proved the assumption by induction over the pathlength n . $\square$

Theorem 1. Given a program C_0 with post-expectation ψ and starting state σ_0 , let $M_{\langle C_0, \sigma_0 \rangle}$ be its operational MDP with reward function rew_{ψ} and let $\mathcal{SM}_{C_0}$ be its symbolic MDP with reward function $\text{rew}_{\mathcal{SM}_{C_0}}$. Then the maximal expected reward of $M_{\langle C_0, \sigma_0 \rangle}$ is bound by the maximal expected reward of $\mathcal{SM}_{C_0}$ with input σ_0 , i.e.,

$$\text{MaxER}(\mathcal{M}_{\langle C_0, \sigma_0 \rangle}, \text{rew}_{\psi}) \leq \text{MaxER}(\mathcal{SM}_{C_0}, \sigma_0).$$

Proof.

$$\begin{aligned}
& \text{MaxER}(\mathcal{M}_{\langle C_0, \sigma_0 \rangle}, \text{rew}_\psi) \\
&= \sup_{\mathfrak{S} \in \text{Scheds}} \text{ER}_{\mathfrak{S}}(\mathcal{M}_{\langle C_0, \sigma_0 \rangle}, \text{rew}_\psi) && (\text{Definition 7}) \\
&= \sup_{\mathfrak{S} \in \text{Scheds}} \sup_{n \in \mathbb{N}} \text{ER}_{\mathfrak{S}}^n(\mathcal{M}_{\langle C_0, \sigma_0 \rangle}, l_{\text{init}}, \text{rew}_\psi) && (\text{Definition 7}) \\
&= \sup_{\mathfrak{S} \in \text{Scheds}} \sup_{n \in \mathbb{N}} \sum_{\pi \in \text{Paths}_{\overline{\mathcal{M}}_{\langle C_0, \sigma_0 \rangle}}^n} \text{Prob}_{\mathfrak{S}}(\pi) \cdot \text{rew}_\psi(\pi) && (\text{Definition 7}) \\
&= \sup_{\mathfrak{S} \in \text{Scheds}} \sup_{n \in \mathbb{N}} \sum_{\pi \in \text{Paths}_{\overline{\mathcal{M}}_{\langle C_0, \sigma_0 \rangle}}^n} \text{Prob}_{\mathfrak{S}}(f(\pi)) \cdot \text{rew}_{\mathcal{SM}_{C_0}}(f(\pi)) && (\text{Lemma 1}) \\
&\leq \sup_{\mathfrak{S} \in \text{Scheds}} \sup_{n \in \mathbb{N}} \sum_{\pi' \in \text{Paths}_{\overline{\mathcal{SM}_{C_0, \sigma_0}}}^n} \text{Prob}_{\mathfrak{S}}(\pi') \cdot \text{rew}_{\mathcal{SM}_{C_0}}(\pi') \\
&= \text{MaxER}(\mathcal{SM}_{C_0, \sigma_0}). && (\text{Definition 13})
\end{aligned}$$

Most of the steps simply follow from the definition of expected rewards until we arrive at the sum over all paths. We then use [Lemma 1](#) to show that the paths' probabilities and rewards stay the same after mapping them with f . Since f is well defined, we know that $f(\pi) = \pi'$ is a valid path in $\mathcal{SM}_{C_0}$ for every π . However, when we take the supremum over the paths of $\mathcal{SM}_{C_0}$, there could be other paths that lead to larger expected rewards², which is why an inequality is used here. We then substitute the definition of expected rewards for symbolic MDPs to receive $\text{MaxER}(\mathcal{SM}_{C_0, \sigma_0})$. $\square$

²One could prove that this is not the case and the expected rewards are indeed the same, by showing bijectivity of f . Here, we leave this out, though, since the inequality suffices for us because KoAT computes an upper bound in the end anyway.

3.2 The KoAT Format

Having learned about the intermediate step of our translation, we will now look at the goal format we want to end up with, namely the one of KoAT. KoAT is a tool from LuFG i2 Aachen, that automatically infers upper bounds on the expected runtime of probabilistic integer programs. It does so by a modular approach, i.e., by computing bounds for individual general transitions and summing them up for a total expected runtime bound on the whole program [MHG21a, LG24].

We will first see the formal definition of a probabilistic integer program (PIP), which is very similar to the symbolic MDP we introduced before. Then we will also take a look at the input syntax KoAT accepts, before developing our translation from the JANI model to the KoAT model in Section 3.3.

The set of all variables in a program $\mathcal{V}$ includes the set of program variables $\mathcal{PV} \subseteq \mathcal{V}$. The remaining variables $\mathcal{V} \setminus \mathcal{PV}$ are called *temporary variables*. The set of constraints $\mathcal{C}$ contains (in-)equalities between polynomials and their conjunction. Formally, $e_1 \sim e_2 \in \mathcal{C}$ for all polynomials $e_1, e_2 \in \mathbb{Z}[\mathcal{V}]$ and relations $\sim \in \{=, \neq, \leq, <, \geq, >\}$. We differ here slightly from the source, which only uses $\leq$ in the definition, which still allows for constructs equivalent to the other relations, though. Additionally, $c_1 \wedge c_2 \in \mathcal{C}$ for all constraints $c_1, c_2 \in \mathcal{C}$. We then define probabilistic integer programs as in [MHG21a]:

Definition 16 (Probabilistic Integer Program (PIP)). A probabilistic integer program is defined as $(\mathcal{PV}, \mathcal{L}, \mathcal{GT}, l_0)$ where

- $\mathcal{PV} \subseteq \mathcal{V}$ is a finite set of program variables,
- $\mathcal{L}$ is a finite non-empty set of program locations,
- $\mathcal{GT}$ is a finite non-empty set of general transitions. A general transition $g \in \mathcal{GT}$ is a finite non-empty set of transitions $t = (l, p, \tau, c, \eta, l') \in g$, where
 - l is the start location and l' is the target location,
 - $p : \mathbb{Z}[\mathcal{V}] \rightarrow [0, 1] \cap \mathbb{Q}$ is the probability expression of t ,
 - $\tau \in \mathcal{C}$ is the guard of t ,
 - $c \in \mathbb{Z}[\mathcal{V}]$ is the transition's non-negative cost polynomial and KoAT's equivalent of a reward function,
 - $\eta : \mathcal{PV} \rightarrow \mathbb{Z}[\mathcal{V}]$ is the update function of t . It maps every program variable to an update polynomial.
 - The probabilities of all transitions of g must add up to 1. Additionally, all transitions $t \in g$ must start at the same location l and have the same guard τ and the same cost c . So every general transition g can also be thought of as having a start location l_g , a guard τ_g and c_g .
- l_0 is the initial location of the program. l_0 must not be reached by any transition, i.e., it can not be any transition's target location

Since PIPs share the same structure as symbolic MDPs without explicit actions, we define expected rewards and MaxER for PIPs analogously to Definition 14. The differences being that schedulers select among enabled general transitions without an explicit action and a general transition's reward is substituted by its cost under PIPs. Under this definition MaxER(P, σ_0) is well defined for a PIP P with initial state σ_0 .

The input format KoAT uses to encode a PIP stems from the Termination and Complexity Competition [MHG21b] and looks like this for our pGCL program from [Example 1](#):

Example 9.

```
(GOAL EXPECTEDCOMPLEXITY)
(STARTTERM (FUNCTIONSYMBOLS 10))
(VAR x)
(RULES
  10(x) -{0}> 11(x)
  11(x) -{0}> 12(x) :|: x >= 2
  12(x) -{0}> 0.5:13(x-1) :+: 0.5:13(x-2)
  13(x) -{1}> 11(x)
)
```

Example 10. Note that it can also be written more efficiently like this. While they are not the same program, they are equivalent with respect to their runtime complexity.

```
(GOAL EXPECTEDCOMPLEXITY)
(STARTTERM (FUNCTIONSYMBOLS 10))
(VAR x)
(RULES
  10(x) -{0}> 11(x)
  11(x) -{1}> 0.5:11(x-1) :+: 0.5:11(x-2) :|: x >= 2
)
```

The transition from 10 to 11 is still needed, because, as mentioned previously, the start location 10 must not be any transition's target location.

We will now explain the syntax of the input format and relate it to the formal definition of a PIP.

- The **GOAL** is not really part of the program, but simply indicates if it is probabilistic. **EXPECTEDCOMPLEXITY** is used if the program contains any probabilistic choices, else **COMPLEXITY** is used if it does not
- **STARTTERM (FUNCTIONSYMBOLS 10)** denotes the initial location 10 of the PIP
- **VAR** denotes the set of variables used, in this case $\mathcal{V} = \{x\}$
- Lastly, **RULES** encodes all general transitions from $\mathcal{GT}$
- A general transition $g \in \mathcal{GT}$ has the form **LHS -{COST}> RHS (:|: GUARD)**
 - **LHS** denotes the start location l_g , along with the variables that are updated by η
 - The **COST** represents the general transition's cost c_g . If omitted, the cost defaults to 1
 - **GUARD** encodes the guard τ_g of the general transition. If omitted, the guard defaults to **true**
 - **RHS** then specifies the transitions $t \in g$, in particular:

- * the target location l' ,
 - * the probability p of the transition, written in front of the location (e.g., $0.5:l0(\dots)$). Probabilities can also be written as a fraction enclosed by square brackets, like this $[1/2]$
 - * the updated variables according to η ,
 - so every term is a polynomial expression from $\mathbb{Z}[\mathcal{V}]$,
 - the i 'th variable in LHS will be assigned a value according to the i 'th expression after the target location. So a rule $l1(x,y) \rightarrow l2(x, x+1)$ would keep x the same and assign the value $x + 1$ to y
 - * The transitions are separated by $:+:$
 - * If g is a singleton, i.e., it only contains one transition, we leave out the implied probability $p = 1$
- The set of locations $\mathcal{L}$ is not explicitly stated but implied by the transitions in RULES

We simply define the expected rewards of PIPs in the same way as we did for symbolic MDPs.

3.3 Translating JANI to KoAT's input

Having defined the JANI model (our symbolic MDP) and KoAT's input format (the PIP), one can see why we chose to use Caesar's JANI translation as a basis for our translation, as the definitions show a lot of similarities. This makes a lot of the translation steps easier, but there are still some small nuances and specific cases we need to consider. So in this section we will explain in detail how our translation from JANI to KoAT works and what special cases to look out for.

Let $\mathcal{SM}_C = (\mathcal{L}, \text{Vars}, \text{Act}, l_{init}, \mathcal{GT})$ be the symbolic MDP of a pGCL program C as defined in [Definition 15](#) for this entire section.

3.3.1 Preprocessing

The first thing our implementation does, is to check whether the program contains non-determinism and is a proc (as opposed to a coproc). If so, a warning is issued, because a proc verifies a lower bound on the expected runtime, while KoAT always gives an upper bound on the expected runtime. While KoAT could, of course, simply provide the upper bound of a translated proc, this is usually not the intended result the user expects when writing a proc. Thus it is highly recommended that only coprocs are given as inputs for the translation.

Before the actual translation begins, we need to alter the JANI model to simplify the translation process. Guards in the JANI model can be arbitrary boolean expressions over the program variables. KoAT however only allows constraints (guards) that are (in-)equalities and conjunctions between them, as explained in the beginning of [Section 3.2](#). To be able to still translate all guards that can appear in the JANI model, we derived a solution consisting of two steps that alter the JANI model itself, before the actual translation steps are executed:

In the first step, all guards of all transitions are transformed to be in Disjunctive Normal Form (DNF). DNF is a form consisting of disjunctions between formulas that only contain

conjunctions: $(\psi_{11} \wedge \dots) \vee (\psi_{21} \wedge \dots) \vee \dots$ with ψ_{ij} being simple (in-)equalities. Every Boolean formula can be transformed into DNF.

After transforming the guards into DNF, there is still the issue that KoAT does not allow any disjunctions. The solution for this is to create multiple transitions that differ only in the guard, one for each conjunction in the DNF.

Formally, consider a general transition $g = \{t_1, \dots, t_m\} \in \mathcal{GT}$ from the JANI model with $t_j = (l, \alpha, p_j, \varphi, r, \eta_j, l'_j)$ for $1 \leq j \leq m$. Let $\varphi' = \bigvee_{i=1}^n c_i$ be the original guard φ in DNF with $c_i \in \mathcal{C}$ being a conjunctive constraint for $1 \leq i \leq n$, as allowed by KoAT. Then we remove g from the set of general transitions and create n new general transitions, one for each conjunction c_i . Each new general transition is simply defined as $g_i = \{t_{i1}, \dots, t_{im}\}$ with $t_{ij} = (l, \alpha_i, p_j, c_i, r, \eta_j, l'_j)$ and a fresh action α_i . This means that we basically create a nondeterministic choice between transitions that all do the same thing, i.e., have the same locations, probability, reward, and assignments, but with different constraints.

To explain why this solution works and does not alter the expected reward of the program, we look at the different cases that can happen when the program runs and location l is reached:

1. No constraint c_i is satisfied. Then φ' is unsatisfied, and because they are equivalent, φ is, too. This means the original general transition g would not have been activated, and neither will any of the new transitions, since all their constraints are unsatisfied.
2. One constraint c_i is satisfied, while all others are unsatisfied. Since φ' is a conjunction over all constraints, one satisfied constraint is enough for φ' , and thus also φ , to be satisfied. That means the original general transition g would be activated and execute one of its transitions t_j with probability p_j after resolving nondeterminism. After the transformation, t_j doesn't exist anymore, but t_{ij} is activated (because c_i is satisfied). Because t_{ij} has the same effects and probability of execution as t_j , the result is the same as if t_j was still there.
3. Lastly, two or more constraints are satisfied. Obviously, φ' and φ are also satisfied then, and g would be able to be executed. Now, though, we have multiple active general transitions in its place. However, this is not an issue, because there will simply be a nondeterministic choice between the general transitions of which the guards are satisfied, and only one will be executed. Since, apart from the action and constraint, they are all the same as each other and as g , the effects will always be the same.

Thus, this transformation changes nothing about the runtime complexity of the program and simply expresses JANI transitions in a way KoAT can handle them. This altered version of the JANI model is then fed into the translation pipeline to receive a KoAT output.

We define *the* probabilistic integer program of a pGCL program in [Definition 17](#)

Definition 17 (PIP of a pGCL program). Let $\mathcal{SM}_C = (\mathcal{L}_1, Vars, Act, l_{init}, \mathcal{GT}_1)$ be the symbolic MDP of a program C with guards that has undergone the aforementioned transformation. We then define the PIP that then results from the translation by a mapping $pip(\mathcal{SM}_C) = (\mathcal{PV}, \mathcal{L}_2, \mathcal{GT}_2, l_0)$ with

- $\mathcal{PV} = Vars$
- $\mathcal{L}_2 = \mathcal{L}_1 \cup \{l_{start}\}$ with $l_{start} \notin \mathcal{L}_1$
- $\mathcal{GT}_2 = \{\delta(g) \mid g \in \mathcal{GT}_1\} \cup \{t_{start}\}$ with
 - $t_{start} = (l_{start}, 1, \mathbf{true}, 0, \eta_0, l_{init})$,
 - $\eta_0(x) = x$ for all program variables $x \in \mathcal{PV}$,
 - $\delta(\{t_1, \dots, t_n\}) = \{\delta(t_1), \dots, \delta(t_n)\}$ for all general transitions $g = \{t_1, \dots, t_n\} \in \mathcal{GT}_1$,
 - and $\delta(l, \alpha, p, \varphi, r, \eta, l') = (l, p, \varphi, r, \eta, l')$ for all transitions $t = (l, \alpha, p, \varphi, r, \eta, l') \in \mathcal{T}_1$, where $\mathcal{T}_1$ is the set of all transitions of $\mathcal{SM}_C$.
- $l_0 = l_{start}$

We will now explain this translation and further details that needed to be considered for its implementation.

3.3.2 Implementation Details

We start with the initial location l_{start} of the PIP. Against intuition, this is not simply the starting location l_{init} from $\mathcal{SM}_C$, but we rather introduce a new location l_{start} and a default transition from l_{start} to l_{init} . This is because KoAT's start term is not allowed to be called recursively. So, an artificial transition without any effect or cost is created, to prevent any issues if l_{init} is called recursively, i.e., is the destination of any transition.

The set of locations from the JANI model is adopted, and only the location l_{start} is added. The locations $\Downarrow$ and $\perp$ needed to be renamed to *down* and *sink* in the implementation, because KoAT does not support those symbols as locations.

For the program variables $\mathcal{PV}$ we simply copy the variables from $Vars$. However, one should note that the variables' types may be converted, since KoAT only supports integers. For example, we express a boolean variable as an integer variable and set it to 0 for **false**, and 1 for **true**. This is not reflected in encoding of the variables themselves, since KoAT just assumes all variables are integers, but we will see the implications of this on the transition's guards.

The set of general transitions $\mathcal{GT}_2$ for $pip(\mathcal{SM}_C)$, also copies all general transitions from $\mathcal{GT}_1$, only removing the actions, which are implicit in PIPs, and adding the start transition t_{start} .

For the GOAL, our translation code goes through every transition and checks if a probabilistic transition is included. If yes, the GOAL is set to **EXPECTEDCOMPLEXITY**, else it is set to **COMPLEXITY**.

Transition Guards

In our implementation, we modify the general transitions slightly further than defined in Definition 17. Because of the transformation of the symbolic MDP in ??, the input guard φ is always a conjunction of atoms. First it is checked if any of the atoms equate to **false**.

If so, the guard is unsatisfiable, and the entire transition is disregarded, as it will never be executed. If any of the atoms equate to **true**, the atom is simply left out, because it does not have any influence on the guard. After that, the remaining atoms simply stay the same and are connected by conjunctions again. If any of the atoms contain booleans, e.g., **b == true**, **true** gets translated to 1 and **false** to 0, as we need to encode all variables as integers. It is obvious that none of these changes influence the program's runtime complexity.

Cost Function

Lastly, when running the translation, there is a choice between the uniform cost-model and the reward cost-model. The uniform cost model ignores all reward statements in the code and thus the reward r of every transition and simply assigns cost 1 to every transition. The only exception to this being the artificial edge from l_{start} to l_{init} , which always has cost 0 as to not affect the program. The uniform cost-model can basically be seen as a step counter. Of course, when applying this model, the result of KoAT will not be a correct expected reward as defined previously, so we only consider the reward cost-model in this thesis.

The reward cost-model is the one used in [Definition 17](#) that simply keeps reward r of the JANI transition as the cost of the PIP transition. This is also the default and recommended model, as Caesar uses the rewards to verify its pre-expectation too, and users who write HeyVL code including reward statements most likely want KoAT's cost analysis to be based on the same reward/cost function.

Example 11 (PIP of our symbolic MDP). Let $\mathcal{SM}_C$ be the symbolic MDP of program C from [Example 1](#), as depicted in [Figure 6](#). Then the PIP $pip(\mathcal{SM}_C)$ that finally results from our entire translation pipeline and is fed into KoAT looks like this:

```
(GOAL EXPECTEDCOMPLEXITY)
(STARTTERM (FUNCTIONSYMBOLS 10))
(VAR x)
(RULES
  10(x) -{0}> 11(x)
  11(x) -{0}> 12(x) :|: x >= 2
  11(x) -{0}> down(x) :|: x < 2
  12(x) -{0}> (1/2):13(x-1) :+: (1/2):14(x-2)
  13(x) -{0}> 15(x-1)
  14(x) -{0}> 15(x-2)
  15(x) -{1}> 11(x)
  down(x) -{0}> sink(x)
)
```

If we compare this to [Example 9](#) or even [Example 10](#), we can see that our version is far from the ideal representation of C in this input format. As we will see in [Chapter 4](#), this blowup in general transitions is also the biggest flaw of our implementation, as it increases runtimes and can lead to timeouts for larger programs. However, it is still a correct representation, since it has the same runtime complexity as [Example 9](#) and [Example 10](#). We will also see the formal correctness of KoAT's upper runtime bound in the following [Section 3.4](#).

3.4 Soundness

Finally, we will conclude this chapter by proving the overall correctness of our translation pipeline displayed in [Figure 1](#).

Relating MaxER to KoAT's Expected Runtime.

KoAT operates on PIPs and computes an *expected runtime bound* as defined in Definition 3 of [\[LG24\]](#). The expected runtime bound $\mathcal{RB}$ is a function that assigns every general transition g an upper bound on the expected number of times g is executed under the worst-case scheduler. Formally, $(\mathcal{RB}(g))(\sigma_0) \geq \sup_{\mathfrak{S}} \mathbb{E}_{\mathfrak{S}, \sigma_0}(\mathcal{R}_g)$. The sum of $\mathcal{RB}(g)$ over all general transitions g is then *the* expected runtime complexity of the entire program, i.e., $\sum_{g \in \mathcal{GT}} ((\mathcal{RB}(g))(\sigma_0)) \geq \sup_{\mathfrak{S}} \mathcal{R}_{\mathfrak{S}, \sigma_0}$ for all input states σ_0 . Here, $\mathcal{R}_{\mathfrak{S}, \sigma_0}$ is the total expected number of times any transition is executed. In our setting, general transitions carry costs corresponding to the reward expressions from $\mathcal{SM}_C$, so that $\sup_{\mathfrak{S}} \mathcal{R}_{\mathfrak{S}}$ corresponds to our maximal expected rewards $\text{MaxER}(\text{pip}(\mathcal{SM}_C), \sigma_0)$. Both take the supremum over all schedulers of the expected collected cost. We therefore write $\mathcal{CA}(\text{pip}(\mathcal{SM}_C))$ for the total bound $\sum_{g \in \mathcal{GT}} \mathcal{RB}(g)$ returned by KoAT.

Lemma 2. Let C be a pGCL program with post-expectation ψ and initial state σ_0 . Let $\mathcal{SM}_C$ be its symbolic MDP with reward function $\text{rew}_{\mathcal{SM}_C}$ and $P = \text{pip}(\mathcal{SM}_C)$. Then the maximal expected reward of $\mathcal{SM}_C$ is the same as the one of P . Formally,

$$\text{MaxER}(\mathcal{SM}_C, \sigma_0) = \text{MaxER}(\text{pip}(\mathcal{SM}_C), \sigma_0).$$

Proof.

The PIP $\text{pip}(\mathcal{SM}_C)$ differs from $\mathcal{SM}_C$ only in the addition of a fresh start location l_{start} and a single transition t_{start} with cost 0 and probability 1. Every path in $\mathcal{SM}_C$ starting in l_{init} with state σ_0 corresponds bijectively to a path in $\text{pip}(\mathcal{SM}_C)$ extended by the initial transition t_{start} . Since t_{start} contributes neither a reward, nor a probability that would change the expected reward of a path, the maximal expected rewards are identical. $\square$

Theorem 2 (Soundness). Let C be a pGCL program with post-expectation ψ and input σ_0 . Then, given $\text{pip}(\mathcal{SM}_C)$ as input, KoAT provides a valid upper bound on the maximal expected reward of the operational MDP of C . Formally,

$$\text{MaxER}(\mathcal{M}_{\langle C_0, \sigma_0 \rangle}, \text{rew}_{\psi}) \leq \mathcal{CA}(\text{pip}(\mathcal{SM}_C))(\sigma_0), \quad (3.1)$$

where $\mathcal{CA}$ is KoAT's cost approximation.

Proof.

$$\begin{aligned} \text{MaxER}(\mathcal{M}_{\langle C_0, \sigma_0 \rangle}, \text{rew}_{\psi}) &\leq \text{MaxER}(\mathcal{SM}_{C_0}, \text{rew}_{\mathcal{SM}_{C_0}}) && \text{(Theorem 1)} \\ &= \text{MaxER}(\text{pip}(\mathcal{SM}_{C_0}), \text{rew}_{\mathcal{SM}_{C_0}}) && \text{(Lemma 2)} \\ &\leq \mathcal{CA}(\text{pip}(\mathcal{SM}_C))(\sigma_0) \square \\ &\quad \text{(Definition 3, [LG24], originally from [MHG21a])} \end{aligned}$$

Corollary 1. The bound $\mathcal{CA}(\text{pip}(\mathcal{SM}_C))(\sigma_0)$ returned by KoAT is a valid pre-expectation to the corresponding coproc of program C in Caesar for all possible inputs σ_0 . That is, $\mathcal{CA}(\text{pip}(\mathcal{SM}_C))(\sigma_0) \geq \text{MaxER}(\mathcal{M}_{\langle C_0, \sigma_0 \rangle}, \text{rew}_{\psi})$ for all σ_0 , since we established in [Section 2.5](#) that $\text{MaxER}(\mathcal{M}_{\langle C_0, \sigma_0 \rangle}, \text{rew}_{\psi}) = \text{ert}[\llbracket C \rrbracket](\psi)(\sigma_0)$.

This completes the formal justification of our translation pipeline. Together, [Theorem 1](#) and [Theorem 2](#) establish that KoAT's result on our translated program constitutes a correct upper bound on the original program's maximal expected rewards, and by [Corollary 1](#) a valid pre-expectation for its corresponding HeyVL coproc.

Chapter 4

Evaluation

4.1 Experimental Setup

We implemented the translation in Rust and created a pipeline that follows the process from [Figure 1](#), calling commands from both, **Caesar** and **KoAT**, in the process. Our code receives a **HeyVL** file and returns **KoAT**’s result of the translated file completely automated. It is available online on Zenodo [\[Ebl26\]](#). In this chapter, we will evaluate the performance of our implementation regarding the runtime of the translation code, the runtime of **KoAT** on the translated file and the result **KoAT** returns.

To this end, we took 44 probabilistic benchmarks originally from the **eco-imp** tool [\[Ava20\]](#), which are also distributed with **KoAT** along with directly translated versions in **KoAT**’s own format [\[mth21\]](#). We then translated the **eco-imp** benchmarks with the help of **Caesar**’s **pgcl2heyvl** tool [\[Verd\]](#) and some manual adjustments into **HeyVL**.

The **HeyVL** benchmarks translated by **pgcl2heyvl** encoded probability distributions as a combination of probabilistic choices and assignments. Since random assignments with actual distribution expressions are translated more efficiently by our code, we manually added them in all possible files, replacing the probabilistic choice constructs. However, to test whether there are any noticeable differences between the two versions, we kept the original files without any distributions in a separate set as alternatives.

We thus have three different formats for each benchmark: a **HeyVL** format that is the ideal representation, the **KoAT** format and the **eco-imp** format. For some benchmarks containing probability distributions, we have an additional **HeyVL** format that encodes the distributions with probabilistic choices and assignments.

We then ran all tools on their respective inputs with a timeout of 60 seconds, timed their runtimes and noted their outputs. This enables a comparison of our translation with **KoAT** and **eco-imp**. **eco-imp** is a different tool for automated expected cost analysis of probabilistic programs, using a modular approach [\[AMS20\]](#). The detailed results of our benchmark analysis can be found in [Appendix A](#).

We classify results into asymptotic complexity classes as follows. **KoAT** returns upper bounds as polynomial expressions over its internal argument names (e.g., $2 * Arg_0 + 3 * Arg_1 + 5$), which we rewrite to match the actual program variables’ names. Since **KoAT**’s bounds hold for arbitrary integer inputs, they are expressed over the absolute values of variables (e.g., $2 * |x| + 3 * |y| + 5$). **KoAT** also classifies these bounds into asymptotic complexity classes directly. We adopt the same classification and apply it to the **eco-imp** results as well. $\mathcal{O}(1)$ is the class for constant bounds, $\mathcal{O}(n)$ is used when the result is linear

in its input variables and $\mathcal{O}(n^2)$ if it contains a quadratic term such as $|x^2|$ or $|x| * |y|$. We write ∞ if a tool determines that no finite upper bound exists.

All commands were performed in WSL on a freshly restarted machine running Windows 11 with an Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz and 8GB RAM. For KoAT’s executions, we used the static binary with version KoAT2 v2.1.0 available at [Lom25]. For our translation, we used the rewards cost-model. Our evaluation can be replicated by executing the command `run-benchmarks` in the project “caesar-koat-translation” [Ebl26] to receive a JSON file with the results. Executing `generate-latex` then returns a tex file that displays the results in tables and scatter plots. Detailed information on how to execute the available commands can be found in the README.

4.2 Results and Discussion

We classified each analysis run as a *success*, a *timeout*, an *error* or ∞ and counted how often each result occurred for the respective tools/encodings in Figure 8 and Figure 9. A run is counted as a success when an asymptotic bound unequal to infinity was returned before the timeout of 60 seconds. If the process took longer, we counted a timeout. If it terminated before the timeout but could not find a finite bound, we counted it as ∞ . In the two HeyVL cases where an error occurred, KoAT exited with non-zero status: “signal: 11 (SIGSEGV) (core dumped)”. KoAT itself produced the error “Segmentation fault (core dumped)” for the benchmark “bayesian” in our setup, while our pipeline produced a timeout.

For every success, we also classified each file’s result in one of the three asymptotic complexity classes as described previously. Lastly, we measured the average and median time of the entire pipeline (i.e., combined runtime of the translation and KoAT for HeyVL files and execution time of KoAT/eco-imp, respectively). For this, only benchmarks that were successful in every pipeline were included, as to not skew the results. All standard deviations are reported as sample standard deviations¹.

¹i.e., using Bessel’s correction, dividing by $n - 1$

Figure 8: Comparison of the two HeyVL encodings for the 21 benchmarks that contain probability distributions. The top row represents encoding via distribution expressions, the bottom row represents encoding via probabilistic choices.

Tool	Success	Timeout	Error	∞	$O(1)$	$O(n)$	$O(n^2)$	Avg Time	Median Time	Std Dev
Caesar + KoAT	12	1	0	8	0	9	3	2.47 s	2.25 s	1.43 s
Caesar (no dis) + KoAT	10	9	2	0	0	9	1	5.29 s	4.74 s	3.56 s

First, we will look at the difference between the HeyVL encodings for the benchmarks that contained probability distributions. The results are displayed in Figure 8. We can see that using HeyVL’s distribution expressions instead of probabilistic choices when possible, results in faster runtimes, fewer timeouts and errors, and a few more successes. The reason for this might be that distribution expressions produce fewer general transitions in the translation, making it easier for KoAT to analyze the result. However, there are also many cases where this version terminated without having found a real upper bound. Interestingly though, for some of these cases, the version without distribution expressions did find a solution; see Figure 13 (in particular: C4B_t15, fcall, hyper, sprd-walk). A detailed analysis of when and why this behavior occurs is left as future work at this point. Since the distribution expression encoding produces fewer timeouts and more successes overall for our benchmarks, we will mainly use this encoding in the remainder of the evaluation.

In Figure 9 we can now see the summarized results of our evaluation for all 44 benchmarks. As mentioned above, for our translation pipeline, we use the HeyVL encoding with distribution expressions when possible. We only included the 27 benchmarks in the average and median runtimes and the standard deviation, that were successful on every one of the three tools. The Caesar + KoAT row again represents our entire implemented pipeline.

4.2.1 Performance

We will firstly address the runtime of our implementation in comparison to KoAT and eco-imp. We can see that our implementation takes longer on successful benchmarks than KoAT on its own. While the relative overhead is significant (roughly $6\times$), the absolute difference remains small at only ≈ 2.6 seconds on average. However, one should also note that the standard deviation is relatively high, so there are cases that do take much longer. Our pipeline also produces 7 timeouts, which do not occur for direct KoAT execution. The actual runtime of our translation has very little impact on this and is negligible, with an average time of ≈ 0.2 seconds. The overhead mostly comes from KoAT’s runtime on the translated file. We attribute this overhead primarily to the larger number of transitions and variables in the translated files. The reason is that HeyVL needs more lines of code to encode a program and also more than double the number of variables, since it needs an *init_x* for every *x* and an additional boolean for its flip statements, as explained in Section 2.5.

Figure 9: Summary results for all 44 benchmarks across the three analysis pipelines.

Tool	Success	Timeout	Error	∞	$O(1)$	$O(n)$	$O(n^2)$	Avg Time	Median Time	Std Dev
Caesar + KoAT	28	7	0	9	6	18	4	2.92 s	1.58 s	3.75 s
direct KoAT	35	0	1	8	7	24	4	0.34 s	0.28 s	0.21 s
eco-imp	39	5	0	0	6	28	4	0.02 s	0.02 s	0.03 s

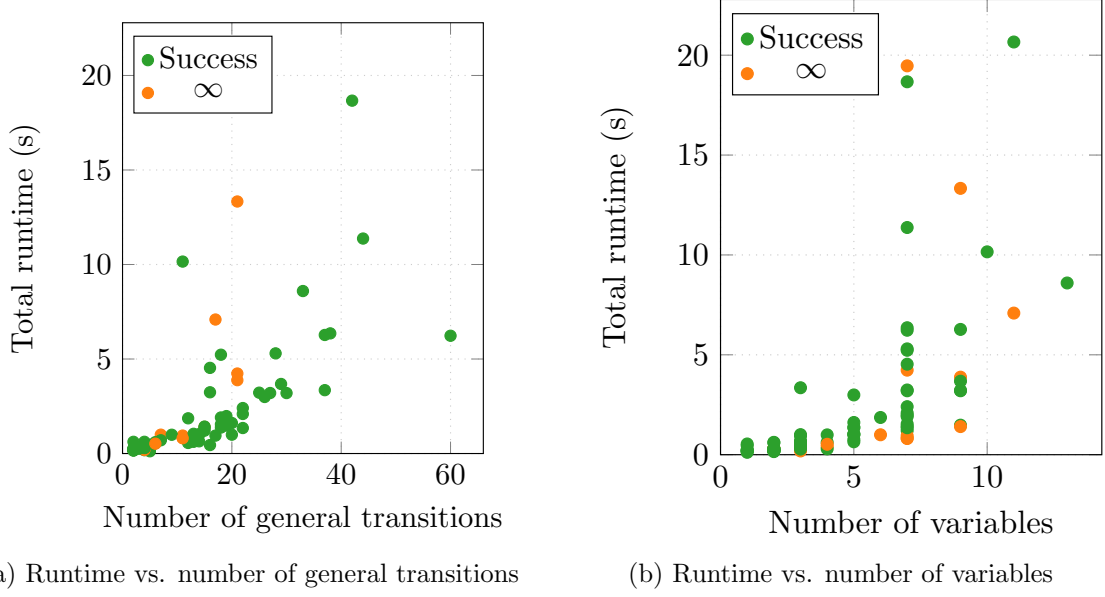


Figure 10: KoAT runtime as a function of program size. Timeouts are excluded.

To test our hypothesis, we created two scatter plots in Figure 10 that display all benchmarks that resulted in *success* or ∞ . This includes the direct KoAT files, as well as the translated versions of both HeyVL encodings. For every KoAT file, we counted the number of general transitions, as well as the number of variables and mapped those to the x-axis, respectively. The y-axis represents the runtime of the respective benchmark in seconds. The scatter plots indeed suggest a positive correlation between the number of general transitions and KoAT’s runtime, as well as a positive correlation between the number of variables and KoAT’s runtime. While correlation does not imply causation, it is likely that the added complexity of larger programs is the cause for increased runtimes. Additionally, HeyVL benchmarks that caused a timeout have an average of 58.0 transitions and 10.5 variables per translated KoAT file, compared to 23.3 transitions and 6.6 variables for successful HeyVL benchmarks. This further supports the hypothesis that larger programs are a primary driver of longer runtimes and timeouts.

eco-imp’s execution of the benchmarks is by far the fastest of all three approaches and produces the most successes (see Figure 9). Since the benchmarks originate from the eco-imp project though, eco-imp performs particularly well in these instances, concerning both runtime and results. This bias limits the comparability to the other approaches. Nevertheless, our implementation is still able to return comparable result quality on 24 of 25 benchmarks where both tools succeed (see Figure 12). Additionally, there are 3 instances where our pipeline was able to infer a bound, while eco-imp timed out.

4.2.2 Result Quality

Beyond runtime, we also compare the quality of results. An important insight we gain here is that, if we do not time out, our implementation always produces the same asymptotic result that KoAT’s direct call produces (see Figure 12). There is only one exception to this, namely the “multirace” benchmark. Here, our pipeline even produces better results than

the original KoAT file, since KoAT was able to return a finite bound for our translation, while returning ∞ for KoAT’s original benchmark. This means in some rare cases our approach can even provide a better asymptotic result.

Additionally, we can compare the exact results of Caesar + KoAT in Figure 13 and direct KoAT in Figure 14. Here we count 2 instances where the result was the same (“ber”, “no_loop”), 1 case where only Caesar + KoAT got a result (“multirace”) and 8 cases where only KoAT got a result. More important are the 25 cases where they both returned results that differed. In 21 of these 25 cases, our translation was able to produce a tighter result, with lower constants than KoAT. Direct KoAT only performed better on 4 benchmarks (“C4B_t61”, “pr_seq_bin”, “pol07”, “rdseq1”). We can thus conclude that for exact results, our translation pipeline is even preferable to a direct execution of KoAT in many cases. Why this is the case remains an open question and is left for future work.

4.2.3 Correctness

To see if the exact results we produce are correct upper bounds, we selected a few HeyVL benchmarks from Caesar’s example set [SD25], that included correct preexpectations and returned results for our implementation. Some of these benchmarks also originate from the same eco-imp set we used previously. In Figure 11 we compare the results of our analysis to the actual pre-expectations, to empirically verify whether our implementation can automatically infer correct pre-expectations for HeyVL programs. While our focus remains on expected runtime analysis, we also included 3 wp-benchmarks that had a non-trivial post-expectation unequal to 0, to see how well our implementation performs there.

The computed bounds are valid upper bounds and are consistent with the exact preexpectations. For the ert-benchmarks (i.e., the ones with postexpectation 0), the computed bounds are structurally similar to the known preexpectations. They typically only differ in constant factors or variable signs. For instance, our pipeline returns $2|x| + 2|n|$ for “fcall”, while the exact pre-expectation is $2 * (n - x)$. Both expressions are linear in the same variables, and the difference is explained by KoAT’s use of absolute values, which conservatively handles negative inputs.

For “geo1”, while ∞ is not a useful result to use as a pre-expectation, it is still technically a correct upper bound. The other bounds of the wp-benchmarks returned results that were not as tight. Overall, this part of the evaluation supports our claim that the results of our implementation are correct upper bounds on the expected runtime of the inputs.

4.2.4 Summary

To summarize and answer our research questions from the beginning:

RQ1 (Correctness): Our results confirm that the implementation produces sound upper bounds. If KoAT returns a finite result, it constitutes a valid preexpectation.

RQ2 (Result quality): Our pipeline consistently matches KoAT’s asymptotic results, and in one case even improves upon them. Concerning exact results, we even return tighter outputs than KoAT itself in a majority of cases.

RQ3 (Performance): While our implementation introduces a roughly $6\times$ relative overhead on KoAT’s direct runtime, mainly due to larger translated files, it remains practical for smaller programs.

Figure 11: Comparing computed upper bounds with known pre-expectations of selected benchmarks. Benchmarks with non-trivial postexpectations unequal to 0 are marked with $\dagger$.

Benchmark	pre	exact
brp1 $\dagger$	$([toSend \leq 4] * (totalFailed + 1)) + ([!(toSend \leq 4)] * \infty)$	$3 * toSend * maxFailed + 3 * sent * maxFailed + 3 * toSend + 3 * sent + 3 * maxFailed + 3 * failed + 5 * totalFailed + 3$
condand	$m + n$	$ n + m + 1$
fcall	$2 * (n - x)$	$2 * x + 2 * n $
geo1 $\dagger$	$c + 1$	∞
hyper	$5 * (n - x)$	$8.68 * x + 8.68 * n + 16.36$
linear01	$0.6 * x$	$ x $
prdwalk	$1.14286 * ((n + 4) - x)$	$8.66 * x + 8.66 * n + 5.43$
rdwalk	$2 * ((n + 1) - x)$	$2 * x + 2 * n + 2$
refute-brp5_bmc $\dagger$	$totalFailed + 1$	$3 * toSend * maxFailed + 3 * sent * maxFailed + 3 * toSend + 3 * sent + 3 * maxFailed + 3 * failed + 5 * totalFailed + 3$

We can thus conclude that our implementation successfully bridges Caesar and KoAT, enabling automated expected runtime analysis for HeyVL programs. While performance limitations remain for larger programs, the results demonstrate that our translation is both correct and practically applicable.

Chapter 5

Conclusion and Future Work

In this thesis, we developed a fully automatic translation from HeyVL into KoAT’s input format and formally proved its soundness. If KoAT returns a bound on the translated program, it constitutes a valid upper bound on the original program’s expected runtime. Furthermore, we ran our implementation of the translation on the benchmark set of the tool *eco-imp* and evaluated the performance of the translation. We compared it with KoAT’s execution on its own version of the benchmarks, as well as with *eco-imp*.

We now revisit our research questions and summarize our findings:

RQ1 - *Correctness*:

We formally proved that KoAT’s output on our translated programs constitutes a sound upper bound on the original program’s expected runtime. This is further supported empirically by our evaluation, where all computed bounds were consistent with valid pre-expectations.

RQ2 - *Result quality*:

We found that our pipeline consistently produces asymptotically equivalent results to running KoAT on its own benchmark versions. In one case our pipeline even improves upon it. For the majority of cases where our pipeline produced results, the exact result was even tighter than KoAT’s own version, showcasing an improved analysis result.

RQ3 - *Performance*:

Our implementation introduces a relatively high runtime overhead of approximately 2.6 seconds on average for successful benchmarks compared to direct KoAT execution. Our research indicates that this can be attributed to the higher complexity and larger size of our translated inputs. In particular, we were unable to compete with *eco-imp*’s almost instant runtimes.

The current implementation has two notable limitations: firstly, we only considered an executable fragment of HeyVL that can be expressed by pGCL code. This excludes HeyVL’s verification statements from our translation. Secondly, as the evaluation shows, the translated files are significantly larger than handwritten KoAT inputs, which leads to performance problems and timeouts.

While our pipeline is currently implemented as a standalone tool that just calls Caesar’s commands, a natural next step would be to further integrate the pipeline directly into Caesar’s codebase, also written in Rust. This would allow users to call our commands directly via Caesar.

Additionally, the translation could be optimized to reduce the size of the translated files, for example, by eliminating the auxiliary variables from HeyVL that KoAT does not need or by collapsing multiple transitions in a single general transition.

All formats (i.e., HeyVL, JANI and KoAT) support distribution expressions [Verc, Hen18, MHG21a]. However, Caesar’s translation into JANI currently encodes a distribution in a general transition as a probabilistic choice between different assignments. Thus, a logical next step would be to add support for translating distributions into JANI directly and then from JANI to KoAT as well, which could increase KoAT’s performance further.

Bibliography

- [AMS20] Martin Avanzini, Georg Moser, and Michael Schaper. A modular cost analysis for probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, November 2020.
- [Ava20] Martin Avanzini. eco-imp benchmarks. https://gitlab.inria.fr/mavanzin/ecoimp/-/tree/master/examples/NCH-18?ref_type=heads, 2020. Accessed: 2026-04-12.
- [BKL08] C. Baier, J.P. Katoen, and K.G. Larsen. *Principles of Model Checking*. The MIT Press. MIT Press, 2008.
- [BKMW24] Kevin Batz, Benjamin Lucien Kaminski, Christoph Matheja, and Tobias Winkler. Jp: Mdp. fp. pp: Characterizing total expected rewards in markov decision processes as least fixed points with an application to operational semantics of probabilistic programs. In *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part I*, pages 255–302. Springer, 2024.
- [Ebl26] Mats Eblenkamp. caesar-koat-translation: Artifact for bachelor’s thesis ‘integrating koat’s push-button probabilistic cost analysis into caesar’. <https://doi.org/10.5281/zenodo.19520566>, 2026.
- [Hen18] Hans Christian Hensel. *The probabilistic model checker Storm : symbolic methods for probabilistic model checking*. Dissertation, RWTH Aachen University, Aachen, 2018. Veröffentlicht auf dem Publikationsserver der RWTH Aachen 2019; Dissertation, RWTH Aachen University, 2018.
- [jm] jani model. jani-model. https://docs.google.com/document/d/1BDQIzPBtscxJFF1DUEPIo8ivKHgXT8_X6hz5quq7jK0/edit?tab=t.0. Accessed: 2026-04-15.
- [Kam19] Benjamin Lucien Kaminski. *Advanced Weakest Precondition Calculi for Probabilistic Programs*. PhD thesis, RWTH Aachen, 2019.
- [Lom25] Nils Lommen. Koat releases. <https://github.com/aprove-developers/KoAT2-Releases/releases>, 2025. Accessed: 2026-04-12.
- [LG24] Nils Lommen, Éléonore Meyer, and Jürgen Giesl. Control-flow refinement for complexity analysis of probabilistic programs in koat, 2024.
- [MHG21a] Fabian Meyer, Marcel Hark, and Jürgen Giesl. Inferring expected runtimes of probabilistic integer programs using expected sizes. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction*

- and Analysis of Systems*, pages 250–269, Cham, 2021. Springer International Publishing.
- [MHG21b] Fabian Meyer, Marcel Hark, and Jürgen Giesl. Inferring expected runtimes using sizes website, 2021.
- [mth21] mthark. Koat2 probabilistic benchmarks, release v1.1. <https://github.com/aprove-developers/KoAT2-Releases/releases/tag/v1.1>, 2021. Accessed: 2026-04-12.
- [SBK⁺23] Philipp Schröder, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. A deductive verification infrastructure for probabilistic programs. *Proc. ACM Program. Lang.*, 7(OOPSLA2):2052–2082, 2023.
- [SD25] Philipp Schröder and Umut Yiğit Dural. Caesar pgcl examples in heyvl. <https://github.com/moves-rwth/caesar/tree/main/pgcl/examples-heyvl>, 2025. Accessed: 2026-04-13.
- [SHK25] Philipp Schröder, Darion Haase, and Joost-Pieter Katoen. Error Localization, Certificates, and Hints for Probabilistic Program Verification via Slicing (Extended Version), December 2025. arXiv:2512.20214 [cs].
- [Vera] Caesar Verifier. About caesar. <https://www.caesarverifier.org/about>. Accessed: 2026-01-15.
- [Verb] Caesar Verifier. Caesar: Model checking. <https://www.caesarverifier.org/docs/model-checking>. Accessed: 2026-04-15.
- [Verc] Caesar Verifier. Heyvl distribution expressions. <https://www.caesarverifier.org/docs/stdlib/distributions>. Accessed: 2026-04-14.
- [Verd] Caesar Verifier. pgcl2heyvl frontend. <https://www.caesarverifier.org/docs/pgcl>. Accessed: 2026-04-12.

Appendix A

Evaluation Results

Figure 12: Asymptotic Results

Benchmark	Caesar + KoAT	direct KoAT	eco-imp
2drwalk	TO	∞	$\mathcal{O}(n)$
C4B_t09	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
C4B_t13	∞	∞	$\mathcal{O}(n)$
C4B_t15	∞	$\mathcal{O}(n)$	TO
C4B_t19	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
C4B_t30	∞	∞	-
C4B_t61	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
bayesian	TO	ERR	$\mathcal{O}(n)$
ber	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
bin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
complex	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	TO
condand	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
cooling	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
coupon	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
cowboy_duel	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
cowboy_duel_3way	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
fcall	∞	$\mathcal{O}(n)$	$\mathcal{O}(n)$
filling	TO	∞	$\mathcal{O}(n)$
geo	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
hyper	∞	$\mathcal{O}(n)$	$\mathcal{O}(n)$
linear01	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
miner	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
multirace	$\mathcal{O}(n^2)$	∞	$\mathcal{O}(n)$
no_loop	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
pol04	TO	∞	$\mathcal{O}(n^2)$
pol05	∞	∞	$\mathcal{O}(n^2)$
pol06	TO	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$
pol07	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	TO
prdwalk	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
prnes	$\mathcal{O}(n)$	$\mathcal{O}(n)$	TO
prseq	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
prseq_bin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
prspeed	TO	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
race	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
rdbub	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
rdseq1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
rdspeed	∞	$\mathcal{O}(n)$	$\mathcal{O}(n)$
rdwalk	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
rfind_lv	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
rfind_mc	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
robot	TO	$\mathcal{O}(n)$	$\mathcal{O}(n)$
roulette	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
sprdwalk	∞	$\mathcal{O}(n)$	$\mathcal{O}(n)$
trader	∞	∞	TO

Figure 13: Exact Results HeyVL

Benchmark	Caesar + KoAT	Caesar (no dis) + KoAT
2drwalk	TO	-
C4B.t09	$44 * x $	$41 * x $
C4B.t13	∞	-
C4B.t15	∞	$2 * y + 4 * x + 1$
C4B.t19	$2 * k + 4 * i + 251$	-
C4B.t30	∞	TO
C4B.t61	$50 * l + 336$	$50 * l + 336$
bayesian	TO	-
ber	$2 * x + 2 * n $	-
bin	$ n + 2$	TO
complex	$18 * N * M + 2 * w + 36 * N + 6 * y $	ERR
condand	$ n + m + 1$	-
cooling	$ mt + st + t + 1$	$2 * t + mt + st $
coupon	25	-
cowboy_duel	3	-
cowboy_duel_3way	7	-
fcall	∞	$2 * x + 2 * n $
filling	TO	TO
geo	5	-
hyper	∞	$8.68 * x + 8.68 * n + 16.36$
linear01	$ x $	-
miner	$9 * n $	-
multirace	$4 * m * n + 8 * m + 8 * n + 17$	TO
no_loop	15	-
pol04	TO	-
pol05	∞	TO
pol06	TO	-
pol07	$9 * n ^2 + 33 * n + 18$	-
prdwalk	$2 * x + 2 * n + 6$	$8.66 * x + 8.66 * n + 5.43$
prnes	$0.06 * y + 71.73 * n + 55.56$	-
prseq	$3.75 * y + 3.45 * x + 6.90$	$3.75 * y + 3.45 * x + 6.90$
prseq.bin	$3.75 * x + 4.05 * y + 3.75$	$12 * x * y + 12 * x ^2 + 145.05 * y + 168.75 * x + 288.95$
prspeed	TO	-
race	$ h + t + 6$	TO
rdbub	$2 * n ^2$	ERR
rdseq1	$5 * x + y + 2$	-
rdspeed	∞	TO
rdwalk	$2 * x + 2 * n + 2$	-
rfind_lv	2	-
rfind_mc	$ k $	-
robot	TO	-
roulette	$4.93 * n + 98.67$	TO
sprdwalk	∞	$2 * x + 2 * n $
trader	∞	TO

Figure 14: Exact Results KoAT vs. eco-imp

Benchmark	direct KoAT	eco-imp
2drwalk	∞	$2(1 + -d + n)$
C4B.t09	$128 * x + 169$	$41(x)$
C4B.t13	∞	$[y0](1.25(x) + (y))$
C4B.t15	$3 * x + y + 1$	TO
C4B.t19	$2 * k + 6 * i + 101$	$2(-100 + i) + (51 + i + k)$
C4B.t30	∞	-
C4B.t61	$12 * l + 1$	$10(-7 + l) + 7$
bayesian	ERR	$4(n)$
ber	$2 * x + 2 * n $	$2(n + -x)$
bin	$0.20 * n + 1.80$	$1.00(n)$
complex	$48 * N * M + 12 * N + 2 * W + 8 * Y $	TO
condand	$ n + m $	$2(m)$
cooling	$ mt + st + t $	$2.10(1 + t) + (mt + -st)$
coupon	105.17	25
cowboy_duel	3.60	1.20
cowboy_duel_3way	12.50	3.46
fcall	$2 * x + 2 * n $	$2(n + -x)$
filling	∞	$0.60(1 + volToFill) + 0.60(volToFill)$
geo	7	5
hyper	$5 * x + 5 * n $	$5.18(-1 + n + -x)$
linear01	$4 * x $	$(-1 + x)$
miner	$9 * N + 3.50$	$7.50(n)$
multirace	∞	$[m0]4(mn + n)$
no_loop	15	5
pol04	∞	$7.50(x) + 4.50(x^2)$
pol05	∞	$0.62(x) + 1.25(x^2)$
pol06	0	$[minPrice0]2(-minPrice + -minPricesPrice + sPrice + sPrice^2)$
pol07	$4 * N ^2 + 14 * N + 15$	TO
prdwalk	$2.29 * x + 2.29 * n + 9.14$	$2.67(n + -x)$
prnes	$0.11 * Y + 136.96 * N + 105.26$	TO
prseq	$3.90 * X + 4.50 * Y $	$[y10](3(-2 + x + -y) + 3(-2 + x + -y) + 1.50(-9 + y))$
prseq_bin	$2.40 * Y + 1.80 * X + 1.80$	$[y10](3.20(-2 + x + -y) + 3.20(-2 + x + -y) + 1.50(-9 + y))$
prspeed	$21.33 * x * y + 21.33 * x * m + 21.33 * y * n + 21.33 * m * n + 4 * y + 4 * m + 2.67 * x + 2.67 * n $	$2(m + -y) + 1.33(-2 + n + -x)$
race	$1.33 * H + 1.33 * T + 13$	$1.83(3 + -h + t) + (z)$
rdbub	$5.33 * N ^2 + 12 * N + 5$	$(n) + (n^2)$
rdseq1	$3 * x + y + 1$	$[y0](2.25(x) + (y))$
rdspeed	$2.67 * x + 2.67 * n $	$2(m + -y) + 1.33(-2 + n + -x)$
rdwalk	$4 * X + 4 * N + 4$	$2(1 + n + -x)$
rfind_lv	4	2
rfind_mc	$2 * k + 3$	(k)
robot	$2424 * N + 2487$	$1.25(N)$
roulette	$34.53 * n + 690.67$	$4.93(21 + -n)$
sprdwalk	$2 * x + 2 * n $	$2(n + -x)$
trader	∞	TO

Figure 15: Total Time

Benchmark	Caesar + KoAT	direct KoAT	eco-imp
2drwalk	TO	20.72 s	0.07 s
C4B_t09	1.48 s	0.38 s	0.02 s
C4B_t13	4.23 s	0.33 s	0.02 s
C4B_t15	3.88 s	0.29 s	TO
C4B_t19	1.37 s	0.30 s	0.02 s
C4B_t30	1.41 s	0.26 s	0.03 s
C4B_t61	0.65 s	0.22 s	0.02 s
bayesian	TO	ERR	0.01 s
ber	1.21 s	0.20 s	0.01 s
bin	1.86 s	0.15 s	0.02 s
complex	8.60 s	0.76 s	TO
condand	1.02 s	0.19 s	0.01 s
cooling	3.68 s	0.46 s	0.05 s
coupon	3.35 s	0.54 s	0.01 s
cowboy_duel	0.95 s	0.26 s	0.01 s
cowboy_duel_3way	6.23 s	1.00 s	0.01 s
fcall	0.82 s	0.32 s	0.01 s
filling	TO	1.00 s	0.02 s
geo	1.00 s	0.18 s	0.01 s
hyper	0.95 s	0.20 s	0.01 s
linear01	0.56 s	0.17 s	0.01 s
miner	5.30 s	0.70 s	0.02 s
multirace	20.66 s	0.40 s	0.03 s
no_loop	0.45 s	0.12 s	0.00 s
pol04	TO	0.19 s	0.02 s
pol05	19.47 s	0.33 s	0.03 s
pol06	TO	0.99 s	0.06 s
pol07	1.91 s	0.41 s	TO
prdwalk	4.53 s	0.28 s	0.02 s
prnes	2.99 s	0.60 s	TO
prseq	2.09 s	0.30 s	0.02 s
prseq_bin	2.40 s	0.31 s	0.03 s
prspeed	TO	0.62 s	0.03 s
race	3.24 s	0.18 s	0.06 s
rdbub	5.23 s	0.46 s	0.02 s
rdseq1	1.61 s	0.30 s	0.02 s
rdspeed	7.09 s	0.29 s	0.03 s
rdwalk	1.04 s	0.26 s	0.01 s
rfind_lv	0.62 s	0.21 s	0.01 s
rfind_mc	1.54 s	0.28 s	0.01 s
robot	TO	10.15 s	0.02 s
roulette	18.67 s	0.62 s	0.13 s
sprdwalk	0.81 s	0.20 s	0.01 s
trader	13.33 s	0.52 s	TO