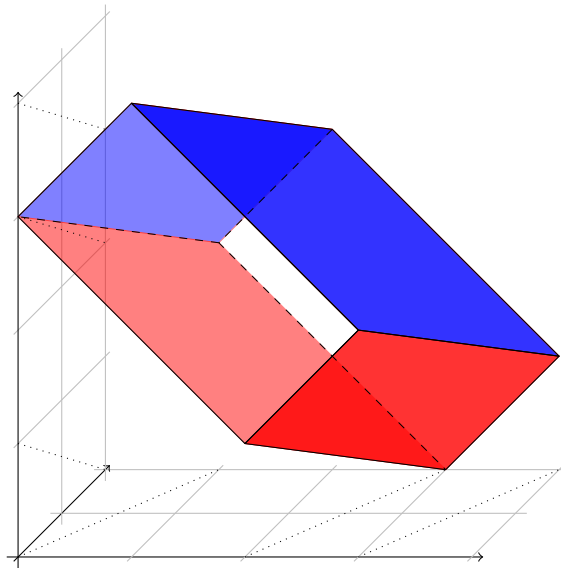


Invariant Generation for Linear Probabilistic Programs

Friedrich Gretz

09.12.2010



Supervised by
Prof. Dr. Ir. Joost-Pieter Katoen
and Prof. Dr. Jürgen Giesl

Acknowledgements

At this point I would like to thank Joost-Pieter Katoen for his advice, patience and support. My thanks also go to Thomas Noll for a short crash course in semantics and all the folks at i2 for a pleasant time.

I am very grateful to Larissa Meinicke who helped a lot through discussions during her visit to Aachen. Also, I wish to thank Annabelle McIver for explaining me several details of pGCL via email.

I really appreciate the help of Arthur C. Norman and Thomas Sturm who readily answered my questions concerning REDUCE and REDLOG.

Last but not least I thank Denise and my family for their love and support.

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 09.12.2010

Friedrich Gretz

Contents

1	Introduction	1
2	probabilistic Guarded Command Language	7
2.1	Syntax	7
2.2	Expectations	10
2.3	Semantics	13
2.3.1	Semantics of a loop	15
2.3.2	A note on the “liberal” assumption	18
2.4	A Proof Example	19
3	Invariant generation	21
3.1	Standard programs	22
3.2	Probabilistic programs	26
4	Implementation	33
4.1	Tutorial	33
4.2	Overview	36
4.3	Parser	36
4.4	From probabilistic verification conditions to FO-formulas	38
4.5	Quantifier elimination and simplification	41
5	Experiments	45
5.1	Binomial Update	45
5.2	Biased Coin	47
6	Conclusion	51
6.1	Summary	51
6.2	Outlook	52

1 Introduction

The meaning of a proposition is the method of its verification.

Moritz Schlick, 1936

In this chapter we learn what verification is, how logical inference works and why loop-invariant annotations are essential. We briefly give an overview over the current research and give an outlook on this thesis.

Verification is a branch of computer science in which we try to formally prove that a given “system” behaves correctly. In this context a system may be a software program, a hardware circuit or any other thing that exposes some behaviour. The definition of what is the correct behaviour is given by the system designer.

In contrast to testing, verification is a tedious and expensive task. Therefore it is rarely applied in the areas where failures do not harm a user’s safety. For example if a computer game crashes due to a programming error, this has no severe consequences - the player has just to wait for the next patch. Things are different for safety critical systems like control software of rockets, airplanes, trains or the circuit design of massively produced computer chips etc. An error in the design or implementation of such systems may lead to loss of life or the financial breakdown of a manufacturer. Here testing does not suffice because each test only shows that the system works in one particular case but does not show that it will work in all cases. This justifies the application of verification techniques.

In order to establish a formal argument why a given system is correct (or not) we first need to find a representation of the system that allows for mathematical reasoning - usually this will be some graph depicting the system’s behaviour. The same is true for the specification of correct behaviour: we have to translate an intuitive claim like “this controller will close the train’s doors before the train moves” to a mathematical entity like a logical formula or an automaton.

Depending on the system and its specification different formal approaches are useful. They differ not only by their application area but also by their automation level and therefore their time and effort. One popular approach is “model checking” [1] another one is “logical inference”.

Model checking, in short, is a technique where the user formulates a model of the system and its properties. The model incorporates all possible behaviour of the system, i.e. it is a representation of all states of the system and transitions between them. The properties define the desired behaviour using temporal logic. A software, called the “model checker”, then *automatically* verifies if the properties are satisfied by the model and provides counterexamples if they are not. The basic underlying idea is that every possible execution of the system is examined and if a

1 Introduction

state that violates a property is reached during execution, this is reported by the model checker. One major drawback of this technique is that the number of states in the model depends not only on the number of variables in the system but also on their domains: the more different values a variable can take the more states there are. For big systems the state space is said to “explode”. Even worse is the fact that variables with infinite domains (like integers or real numbers) cannot be modelled and it is hard and sometimes impossible to abstract from these domains without changing the relevant properties of the model.

Logical inference was introduced in the seminal papers of Robert W. Floyd [7] and C. A. R. Hoare [10]. Their idea is to define certain axioms about a programming language (i.e. define what is the effect of an assignment, what is the behaviour of conditional choice, etc. . .) and then prove the desired property of a program in that language by annotating every program line with a statement that is true for that line. The truth of such an annotation is *inferred* from the truth of the previous annotation and the effect of the program code between them which is dictated by known axioms. Usually such a proof requires human ingenuity and cannot be done fully automatically. This is a drawback, of course. On the other hand we can prove any property that is deducible using the axioms and our knowledge of mathematics. For example, there is no need to restrict our program variables to a finite domain.

The rest of this thesis deals with a technique to aid program verification through logical inference.

Before we turn our attention to the actual topic of our work let us consider the following example program and explain the concept of inference more intuitively.

Claim 1.1. We claim that when the program in Listing 1.1 is executed and successfully stops, it computes the product between some number x and a non-negative integer y without the use of the multiplication operation.

Listing 1.1: Small program that computes $x \cdot y$ using only the $+$ operation

```
result := 0;
counter := 0;
while(counter < y) {
    result := result + x;
    counter := counter + 1;
}
```

We might somehow *see* that the Claim 1.1 is true but we can as well prove it. Therefore we assume that y is a non-negative integer and we show that $result = x \cdot y$ holds when the program ends. In order to show this, we use logical statements or *annotations* between the program lines. Such an annotation is supposed to be true after the previous line of code has been executed and before the program proceeds to the next line. In order to show our claim we assume the truth of the very first annotation and then the truth of subsequent annotations follows from the truth of the previous annotation and our intuition about what the effect of the executed line of code is¹.

¹We will give a precise definition in Chapter 2.

Listing 1.2: Correctness proof for the previous program

```

1   $\langle 0 \leq y \rangle$ 
2  result := 0;
3   $\langle result = 0 \wedge 0 \leq y \rangle$ 
4  counter := 0;
5   $\langle result = 0 \wedge counter = 0 \wedge 0 \leq y \rangle$ 
6   $\langle result = x \cdot counter \wedge counter \leq y \rangle$ 
7  while(counter < y) {
8       $\langle result = x \cdot counter \wedge counter \leq y \wedge counter < y \rangle$ 
9       $\langle result + x = x \cdot counter + x \wedge counter < y \rangle$ 
10      $\langle result + x = x \cdot counter + x \wedge counter + 1 \leq y \rangle$ 
11     result := result + x;
12      $\langle result = x \cdot (counter + 1) \wedge counter + 1 \leq y \rangle$ 
13     counter := counter + 1;
14      $\langle result = x \cdot counter \wedge counter \leq y \rangle$ 
15 }
16  $\langle result = x \cdot counter \wedge counter \leq y \wedge counter \geq y \rangle$ 
17  $\langle result = x \cdot counter \wedge counter = y \rangle$ 
18  $\langle result = x \cdot y \rangle$ 

```

Listing 1.2 shows a proof of Claim 1.1. So what is the reasoning behind this proof? First we assume that y is non-negative. After the variable *result* is set to zero we can additionally assert that $result = 0$. Then, analogously we proceed with *counter* (line 5).

Now comes the part that requires ingenuity: we rewrite $result = 0 = x \cdot 0 = x \cdot counter$ and $0 = counter \leq y$. The purpose is to express relations between the variables. If we can show that this relation is preserved by the while-loop (lines 7-14) we can use this relation in the following way: when the loop terminates (line 16) we know that the loop's guard is false and so $counter < y$ does not hold anymore. Together with our relation from line 6 we know that *counter* must be equal to y and obtain our desired property in line 18.

How can we see that the annotation from line 6 is indeed preserved by the loop?

Case 1: $counter \geq y$ holds and the while-loop is not entered so we jump to line 16. The variables were not changed, hence it is still valid.

Case 2: $counter < y$ holds and the program enters the while-loop. Additionally we know that the inequality is strict ($counter < y$, line 8) and we can add x on both sides of the equality without changing the truth of the statement (line 9). In line 10 we use the fact that y is an integer. After the assignment in line 11 we replace $result + x$ by $result$ and after execution of line 13 replace $counter + 1$ by $counter$. Now we see that our assertion from line 6 still holds.

Our proof is finished: we have successfully shown that the given program computes $x \cdot y$ for some number x and a non-negative integer y . The basic ingredient in this proof is the assertion from line 6 which is preserved by the loop. In lines 16-18 we easily deduce our claim from this assertion together with the assumption that the loop terminates².

²The reader might object that we did not prove the termination of the loop. This is true. Termination requires a separate proof. Termination proofs define another research area and are not treated here. Also note that Claim 1.1 did not say that the program terminates but if it does, it produces the right result.

1 Introduction

This example not only shows how logical inference works but also it motivates our work: the most crucial part in constructing a proof is to find an assertion like the one in line 6. These assertions are called *invariants* because they are not changed by the execution of the loop which they annotate. The main hindrance to construct correctness proofs automatically is that, in general, it is impossible to find such invariants fully automatically. Nevertheless it would be nice to support the user’s search for an applicable invariant with a tool. In that way the proofs can be carried out semi-automatically.

In this thesis we present a new tool that discovers invariants of a certain “shape”. The novelty in our work is that we implemented the first tool that generates invariants for *probabilistic programs*. These are programs where the choice between two alternatives can be random according to some discrete distribution. The simplest example looks like this:

Listing 1.3: Probabilistic choice

```
(x := 0 [0.5] x := 1);
```

In Listing 1.3 the variable x is set to either 0 or 1, each with probability 0.5. In other words we might say, that this line represents the flipping of a fair coin. There are several reasons why we might want to have random events in our programs: One reason is that randomisation makes the worst case behaviour of an algorithm unlikely. For example, sorting algorithms or scheduling algorithms often depend on choices made during their execution. For some input instances these choices might be optimal while for others the same choices might slow down the algorithm. In order to level out the difference between best-case runtime and worst-case runtime, the choices are made randomly. Another reason for probabilistic algorithms is that they might solve a “hard” problem very fast with only a low error probability. Finally, there are problems that cannot be solved by a deterministic algorithm.

Before we proceed to explain our approach in detail in the next chapters, we briefly discuss the related work and place ours into a bigger picture.

Just as there are several approaches to verification in general, there are at least two approaches to find loop-invariants, that are distinguished in literature: “abstract interpretation” and “constraint-based” methods. We will not confuse the reader with a discussion of how those methods differ. We just state that abstract interpretation methods are usually considered to have better scalability³ but are not complete. This means they discover some invariants but not all of them. In contrary, constraint-based methods are complete but can only be applied to smaller problems at their current state of the art.

Constraint-based invariant generation for non-probabilistic⁴ programs was introduced, amongst others, in a paper by Colón, Sankaranarayanan and Sipma [4]. Further work was carried out by [9, 11, 16].

The main idea is that we can construct constraints, that decide if a given annotation is invariant for a loop in a given program. The main task is to solve these constraints and thereby find all satisfying annotations. And hence the name of this approach.

Katoen, McIver, Meinicke and Morgan [12] have lifted the method of [4] to probabilistic

³They still work reasonably well for “larger” programs.

⁴Programs that do not use probabilistic choice will be referred to as *standard*, *qualitative* or *non-probabilistic* throughout this thesis.

programs. In the scope of this diploma thesis we have built the first tool that generates invariants for probabilistic programs based on the work in [12].

The rest of this thesis is divided as follows: in the next chapter we set out the terms and definitions which are used to talk about probabilistic programs and their invariants. Constraint-based invariant generation is explained in Chapter 3. In Chapter 4 we discuss the design of our tool and highlight important implementation details. The application of the tool to some examples is shown in Chapter 5 and finally we conclude this thesis with a discussion of the work done so far and give an outlook on future work in Chapter 6.

2 probabilistic Guarded Command Language

Penny: I give up. He's impossible!
Sheldon: I can't be impossible; I exist! I believe what you meant to say is, 'I give up. He's improbable'.

The Big Bang Theory
Season 03, Episode 03

This chapter introduces the syntax and semantics of a probabilistic programming language. We will learn how properties of probabilistic programs can be proven using annotations that we call “expectations” [14].

This thesis is centered around a language called pGCL [14]. pGCL stands for “probabilistic Guarded Command Language”. It is a variation of Dijkstra’s GCL [6] enriched by a language construct for probabilistic choice (retaining nondeterministic choice at the same time). We do not consider a real language like Java or C and instead use the much simpler and clutter-free pGCL in order to show how our approach works in principle.

2.1 Syntax

The only data-type in pGCL are the real numbers. In order to manipulate real-valued program variables we have the assignment construct:

$$x := E;$$

Here, x is the name of a variable and E some numerical expression which is built from numbers, variable names and arithmetical operators.

Our language has three constructs for expressing a choice in the control flow. First, there is the conditional choice:

$$\mathbf{if}(G) \{prog_1\} \mathbf{else} \{prog_2\}$$

As usual, the *guard* G is a predicate that depends on some program variables. G is evaluated and if it holds, then $prog_1$ is executed, otherwise $prog_2$.

2 probabilistic Guarded Command Language

Second, we define the probabilistic choice:

$$\{prog_1\} [p] \{prog_2\}$$

The meaning is that either $prog_1$ is executed with probability p , or, with probability $1 - p$, $prog_2$ is executed.

The third kind of choice is nondeterministic choice. Nondeterminism is the ability to choose between alternatives without specifying how the decision should be made. Thus any decision strategy complies with nondeterministic choice. The syntax is:

$$\{prog_1\} [] \{prog_2\}$$

Nondeterminism can be used to abstract away decisions which you cannot influence or when you do not exactly know how they are made. For example, an algorithm might depend on a user's choice, say the pushing of one button or another. We do not know what the user will do, but we know that the one or the other button will be pressed. Using nondeterminism you can specify the user's behaviour and incorporate it into your algorithm in order to reason about the "whole" system.

It is important to realise that nondeterminism is *not* the same as the uniform distribution. This becomes apparent when a decision is made repeatedly: According to the uniform distribution, in the long run, the user has to press each button exactly 50 percent of the time. But according to the nondeterministic choice she has more freedom and can press the buttons according to some other ratio or even press the same button all the time.

Another application of nondeterminism is expressing probability ranges. Usually you do not know the exact probability of an event. Instead of setting a probability to a fixed value, say 0.95, one would rather assume it is somewhere between e.g. 0.93 and 0.97. This can be expressed in the following way:

$$\{\{event_1\} [0.93] \{event_2\}\} [] \{\{event_1\} [0.97] \{event_2\}\}$$

We have seen all three kinds of choice and continue with the language construct for repetition. It is expressed in the usual way using a while-loop:

$$\mathbf{while} (G) \{prog\}$$

Repeatedly, the guard G is evaluated and, if it is true, $prog$ is executed until eventually G is found to be false and the loop terminates.

Sequential composition is used to execute one program after another:

$$prog_1; prog_2$$

We have collected the essential features of pGCL. Finally, there are two more constructs that consist of just a single keyword. The simplest language construct in pGCL is

skip

with the effect that nothing happens. **skip** is rather useless on its own but becomes useful in the context of a choice, for instance:

Listing 2.1: The binomial update program

```

1  x := 0;
2  n := 0;
3  while (n - M + 1 <= 0) {
4      (x := x + 1 [p] skip);
5      n := n + 1;
6  }

```

With probability $1 - p$, x is not changed in an iteration of the loop. In principle, we could replace line 4 either by

$$\{x := x + 1;\} [p] \{\}$$

or

$$\{x := x + 1;\} [p] \{x := x\}$$

but both alternatives are not as explicit and readable as the simple keyword **skip**.

We can model failure (or abnormal termination) of a program using the keyword:

abort

Again, the construct becomes useful in context of some choice like:

Listing 2.2: The use of improper termination

```

1  x := 0;
2  n := 0;
3  while (n - M + 1 < 0) {
4      {x := x + 1;} [p] {skip;}
5      n := n + 1;
6  }
7  if (x-1 <= 0) {
8      abort;
9  }

```

This program has a non-zero chance — depending on p — to not terminate (properly).

Definition 2.1 (pGCL syntax). The pGCL syntax is summarized in the subsequent grammar.

$$\begin{aligned}
 \textit{prog} &:= \textit{prog prog} \mid \textit{stmt}; \mid \textit{choice} \mid \textit{loop} \\
 \textit{stmt} &:= \mathbf{abort} \mid \mathbf{skip} \mid x := E \\
 \textit{choice} &:= \mathbf{if} (G) \{\textit{prog}\} \mathbf{else} \{\textit{prog}\} \mid \textit{prog} [p] \textit{prog} \mid \textit{prog} [] \textit{prog} \\
 \textit{loop} &:= \mathbf{while} (G) \{\textit{prog}\}
 \end{aligned}$$

Let Var denote the set of all program variables. Then, $x \in Var$. E is a numerical expression built from numbers, program variables and the arithmetical operators as follows:

$$E := c \in \mathbb{R} \mid x \in Var \mid E + E \mid E \cdot E \mid E - E \mid E/E.$$

G is a first-order quantifier-free formula

$$G := E \leq 0 \mid E < 0 \mid \neg G \mid G \wedge G \mid G \vee G.$$

2 probabilistic Guarded Command Language

We will call such formulas *predicates*, from now on. Finally, p is either a number or some expression that is a probability

$$p := c \in [0, 1] \mid E.$$

For the subsequent analysis we restrict ourselves to *linear* programs¹.

Definition 2.2 (Linear pGCL program). A pGCL program is called *linear* if it satisfies the following syntactical restrictions:

- the expression E at the right-hand side of an assignment is linear, i.e. E has the form

$$E = a_1x_1 + a_2x_2 + \dots + a_nx_n + b, \quad \begin{array}{l} a_1, \dots, a_n, b \in \mathbb{R} \\ x_1, \dots, x_n \in \text{Var} \\ n = |\text{Var}| \end{array}$$

- the atomic propositions of a guard G are linear inequalities, and
- the parameter p of a probabilistic choice must not depend on any program variable, e.g. we allow

$$p = 5q \text{ where } q \text{ is not a variable name}$$

but disallow

$$p = 1/x \text{ where } x \text{ is a variable name.}$$

Despite our limitation to linear programs our language remains sufficiently expressive. In fact, it is Turing complete because it subsumes the WHILE-language [15]. Another justification for the restriction to linear programs is the fact that non-probabilistic program analysis is often limited to the linear case as well [4].

2.2 Expectations

From our point of view, the semantics of a language defines the way in which we prove a property of a program. But in order to understand the proof, first we have to consider the probabilistic counterpart for annotations as we have seen them in Chapter 1. Let's look more closely on (qualitative) annotations:

Listing 2.3: Qualitative annotations

```
1  ⟨x ≥ 12⟩
2  x := x+1;
3  ⟨x ≥ 13⟩
```

We have learned in the introduction what this means: “Assuming x is at least 12 and line 2 is executed and terminates, we can infer that x is at least 13 in the end.” Now, how would we annotate the following?

$$\{x := x + 1;\} [0.4] \{x := x - 1;\}$$

¹Do not confuse our “linear programs” with an optimisation problem called linear programming.

We have to incorporate probability into our otherwise purely qualitative annotations. McIver and Morgan [14] introduced *expectations* to solve this problem. An expectation, as we know it from stochastics in the discrete case, is a sum of events multiplied by their probability:

$$E(X) = \sum_{x \in \text{supp}(X)} x \cdot p(x)$$

More precisely the expectation E is a function of a discrete random variable X . Each summand is the product of a number x and its probability. In order to make the sum countable the summation is done only for those x that have a positive probability, i.e. lie in the support of X . Analogously, we can annotate our programs using sums of predicates multiplied by a numerical value.

Definition 2.3 (Expectation). An *expectation* is an annotation that maps a state to a non-negative number

$$E : S \rightarrow \mathbb{R}_0^+$$

and has the following syntax

$$E = \sum_i [pred_i] \cdot w_i.$$

Each $pred_i$ is a boolean combination of linear inequalities, cf. Definition 2.1. Predicates are embedded into the real domain where $[true] = 1$ and $[false] = 0$. Each predicate is weighted by a real valued expression w_i . The integer i is an index that runs from zero to some $k \in \mathbb{N}$ which means that there are finitely many summands.

Logical consequence between standard annotations turns into the “less than or equal to” relation between expectations. Let’s consider an example:

Listing 2.4: Quantitative annotations - Expectations

- 1 $\langle [x \geq 12] \cdot 1 \rangle$
- 2 $\{ \mathbf{x} := \mathbf{x} + 1; \} [0.4] \{ \mathbf{x} := \mathbf{x} - 1; \}$
- 3 $\langle [x \geq 13] \cdot 0.4 + [x \geq 11] \cdot 0.6 \rangle$
- 4 $\langle [x \geq 11] \cdot 1 \rangle$

Assuming that $x \geq 12$ holds initially, we compute the expectation in line 3. Line 4 is a consequence because $([x \geq 13] \cdot 0.4 + [x \geq 11] \cdot 0.6) \leq ([x \geq 11] \cdot 1)$ for all $x \in \mathbb{R}$, cf. Figure 2.1.

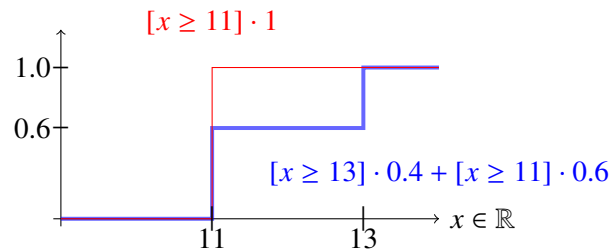


Figure 2.1: $([x \geq 13] \cdot 0.4 + [x \geq 11] \cdot 0.6) \leq ([x \geq 11] \cdot 1)$

2 probabilistic Guarded Command Language

So far we looked at annotated programs and read them from top to bottom, i.e. we looked at some assumption and found out where we can get with that by the end of the program. But that is not what we actually would like to do in verification! Instead we — the system designer — claim some *post-condition*, i.e. a property which is supposed to hold at the *end* of the program, and during the verification procedure we try to find a proof that shows under which assumption our claim is true. A successful proof is one that yields a *pre-condition*, i.e. an assumption before the first line of the program, which is a tautology or a condition that is trivially satisfied, for example, $0 \leq N$ where N is an unknown but positive constant by design.

Therefore we reconsider our example from Listing 2.3 and read it as follows: “ $x \geq 13$ is established by the assignment in line 2, if $x \geq 12$ was true before”. The same reasoning directly applies to the example in the introduction (Listing 1.2, page 2).

In the probabilistic case the reasoning becomes “what is the least probability that a post-condition is satisfied?” or “what is the expected value of a program variable at the end?”

The example from Listing 2.4 then might be annotated as following:

Listing 2.5: Expectation of outcome $x \geq 13$

```

1   $\langle [x \geq 12] \cdot 0.4 + [x \geq 14] \cdot 0.6 \rangle$ 
2   $\{x := x+1;\} [0.4] \{x := x-1;\}$ 
3   $\langle [x \geq 13] \cdot 1 \rangle$ 

```

From the *pre-expectation* in line 1 we know that if $x \geq 14$ initially holds, then surely $x \geq 13$ will be true after executing line 2. If $12 \leq x < 14$ is true initially, then there is still a chance to establish the *post-expectation* with probability 0.4. Note that the post-expectation $x \geq 13$ is purely qualitative. We can also write a purely quantitative post-expectation, in order to find out the expected value of the program variable x .

Listing 2.6: Expectation of x

```

1   $\langle [true] \cdot (x+1) \cdot 0.4 + [true] \cdot (x-1) \cdot 0.6 \rangle$ 
2   $\{x := x+1;\} [0.4] \{x := x-1;\}$ 
3   $\langle [true] \cdot x \rangle$ 

```

Line 1 simplifies to $[true] \cdot (x - 0.2)$. This is the expected value of x . For example, if we run the program from Listing 2.6 again and again where x is initially 10, we expect an average outcome of $x = 9.8$. Imagine that Listing 2.6 is an abstract representation of a game where x is your budget. Then the average outcome $x - 0.2$ means that in the long run the player loses 0.2 units per round.

In principle, it is possible to compute a pre-expectation for a more general post-expectation like the one in Listing 2.7, line 3. But it is not clear what this post-expectation means. Neither does it establish a single proposition nor does it ask for the expected value of a variable. Therefore we can give no meaning to the computed pre-expectation.

Listing 2.7: Expectation nonsense

```

1   $\langle [x \geq 12] \cdot 0.16 + [x \geq 14] \cdot 0.24 + [x \geq 24] \cdot 0.24 + [x \geq 26] \cdot 0.36 \rangle$ 
2   $\{x := x+1;\} [0.4] \{x := x-1;\}$ 
3   $\langle [x \geq 13] \cdot 0.4 + [x \geq 25] \cdot 0.6 \rangle$ 

```

Post-expectations should be chosen in such a way that other annotations, possibly with more than one summand, that occur in the proof for a program can always be interpreted as intermediate steps on the way to a purely qualitative post-expectation (cf. Listing 2.5) or a purely quantitative post-expectation (cf. Listing 2.6).

We have given an intuition about annotations and proceed to the formal definition of the language’s semantics.

2.3 Semantics

We have defined the syntax of pGCL and conveyed intuition for the meaning of each language construct. Probabilistic annotations — called expectations — were introduced and we have used them as pre- and post-expectations in order to express a property of a program, e.g. that a predicate is established with some probability or that a variable has a certain expected value at the end of the program. Still a formal definition is missing that tells us what is the meaning of a pGCL program. Let us therefore look once more on standard programs and remind ourselves how semantics is defined within the non-probabilistic setting [6].

Given a program with n program variables, every evaluation of the variables constitutes a program *state*. The program’s *state space* is \mathbb{R}^n . A postcondition P characterises a subset $F \subseteq \mathbb{R}^n$ of desired final states. The semantics dictates how to find a characterisation Q of the largest set $I \subseteq \mathbb{R}^n$ of initial states, such that the execution of the program started at a state in I leads to a state in F . Then Q is called the *weakest precondition* for the given program with postcondition P . It is the weakest because it imposes the weakest possible constraints on the states and thus characterises the largest possible set of initial states.

Additionally, the precondition should exclude only those states from which the program’s execution terminates in final state that violates the postcondition. Initial states which lead to an infinite computation are not excluded. This property is called “liberal”. It is briefly discussed in Section 2.3.2.

In summary, the semantics of a standard program is given in terms of the *weakest liberal precondition*, wlp for short. Given a program $prog$ and a postcondition P , the $wlp(prog, P)$ characterises all initial states of $prog$ such that if $prog$ is started in one of them and assuming that $prog$ terminates, a final state is reached that satisfies P . From a system designer’s point of view, let Q be a characterisation of the intended initial states, then a successful correctness proof shows that $Q \Rightarrow wlp(prog, P)$. Since wlp *transforms* one condition into another, Dijkstra calls it a *predicate transformer*.

By analogy, McIver and Morgan [14] coined the term *greatest liberal pre-expectation* for probabilistic programs. Their *expectation transformer* is also denoted by wlp (weakest liberal pre-expectation), in order to remind us of the analogy to standard programs. The examples in Section 2.2 show that for qualitative P , the pre-expectation $wlp(prog, P)$ is a discrete probability distribution. For purely quantitative post-expectations, say x , $wlp(prog, [true] \cdot x)$ is the expected value of x .

After this short introduction, definitions are in order that lay the basis for pGCL semantics.

2 probabilistic Guarded Command Language

Definition 2.4 (Expectation space). An *expectation space*

$$\mathbb{E} := (\mathbb{R}^n \rightarrow \mathbb{R}_0^+, \leq)$$

is the set of all mappings from states into the non-negative real domain (cf. Definition 2.3). The at-most relation is inherited point-wise from the ordering on the real numbers:

$$\begin{aligned} P &\leq Q \\ \Leftrightarrow \forall x_1, \dots, x_n \in \mathbb{R} : P(x_1, \dots, x_n) &\leq Q(x_1, \dots, x_n) \end{aligned}$$

Having defined an expectation space, we can view programs as functions that transform expectations. For example, given a pGCL program $prog$, the aforementioned expectation transformer $wlp(prog, \cdot)$ computes for any given post-expectation P the greatest liberal pre-expectation $wlp(prog, P)$.

Definition 2.5 (Expectation transformer). An *expectation transformer* is a map between expectations. Thus every pGCL program induces an expectation transformer. The set of all transformers is denoted by

$$\mathbb{T} := (\mathbb{E} \leftarrow \mathbb{E}, \leq)$$

where the functional arrow is written backwards in order to remind us that we compute a pre-expectation from a post-expectation in a backwards manner. Once again, the ordering is inherited point-wise from the underlying objects. For $wlp(prog, \cdot)$ this means:

$$\begin{aligned} wlp(prog_1, \cdot) &\leq wlp(prog_2, \cdot) \\ \Leftrightarrow \forall E \in \mathbb{E} : wlp(prog_1, E) &\leq wlp(prog_2, E) \end{aligned}$$

Table 2.1 gives a summary of expectation transformations for each language construct. For a given language construct and a given expectation P , we ask for the weakest liberal pre-expectation. The answer is given in the right column of Table 2.1.

The program **skip** does not change the program's state, thus P itself is the pre-expectation. The wlp for **abort** is always 1 independent of the expectation P because **abort** does not properly terminate and due to the liberal assumption in wlp the pre-expectation is $[true]$. The semantics for assignment is the same as in the non-probabilistic setting but lifted to expectations instead of boolean formulae. This means that each occurrence of the variable in each boolean formula and weight is replaced by the expression on the right hand side. For example

$$wlp(x := 5x + 8y + 1, [-x + y \leq 0] \cdot (2x)) = [-5x - 7y - 1 \leq 0] \cdot (10x + 16y + 2).$$

The pre-expectation for sequential composition can be computed recursively.

When a program's execution reaches an **if** statement, the boolean guard G is either true or false. In case it is true, the pre-condition is

$$wlp(prog_1, P) = 1 \cdot wlp(prog_1, P) + 0 \cdot wlp(prog_2, P).$$

Otherwise we consider the **else** branch and the pre-expectation becomes

$$wlp(prog_2, P) = 0 \cdot wlp(prog_1, P) + 1 \cdot wlp(prog_2, P).$$

Table 2.1: The semantics of the pGCL language

prog	wlp(prog, P)
skip	P
abort	1
x:=E	P[x/E]
prog ₁ ;prog ₂	wlp(prog ₁ , wlp(prog ₂ ,P))
if (G) {prog ₁ } else {prog ₂ }	[G] · wlp(prog ₁ ,P) + [¬G] · wlp(prog ₂ ,P)
prog ₁ [p] prog ₂	p · wlp(prog ₁ ,P) + (1-p) · wlp(prog ₂ ,P)
prog ₁ [] prog ₂	min { wlp(prog ₁ ,P), wlp(prog ₂ ,P) }
while (G) {prog ₁ }	greatest expectation Q such that Q = [G] · wlp(prog ₁ , Q) + [¬G] · P

So, depending on the guard we obtain

$$wlp(\mathbf{if} (G) \{prog_1\} \mathbf{else} \{prog_2\}, P) = [G] \cdot wlp(prog_1, P) + [\neg G] \cdot wlp(prog_2, P)$$

in general.

The structure of the pre-expectation is similar for probabilistic choice but instead of predicates $[G]$ and $[\neg G]$ we use probabilities p and $1 - p$ as factors for the two alternatives. If we consider a purely qualitative program, then the weights in the expectation would always evaluate to either 0 or 1 — it is only the probabilistic choice that contributes other values.

Finally, nondeterminism is seen as *demonic* choice. This means that in the wlp computation we assume that the nondeterminism is resolved by a “demon” who tries to minimize our chances to establish our desired property. Therefore we compute the wlp of both alternatives and take the minimum of the two as the pre-expectation of the nondeterministic choice. This is why in our claims we talk about the *least* probability of an outcome or the *least* expected value of a variable. Adopting demonic choice, we guarantee the highest possible probability or value for the worst case scenario.

2.3.1 Semantics of a loop

Unfortunately, there is no purely syntactic definition of the weakest liberal pre-expectation for a loop. The semantics of a while-loop is an expectation transformer defined as the greatest fixed point f of $\Phi : \mathbb{T} \rightarrow \mathbb{T}$, where

$$f \text{ is an abbreviation for } wlp(\mathbf{while} (G) \{prog_1\}, \cdot)$$

and

$$\Phi(f) := [G] \cdot f(wlp(prog_1, \cdot)) + [\neg G] \cdot (wlp(\mathbf{skip}, \cdot)).$$

It is the *greatest* fixed point because of the liberal assumption which means that the pre-expectation of a non-terminating loop is always 1 (and not 0).

The fixed point characterisation does not provide a way to compute a pre-condition for the while-loop fully automatically — in fact there is none. This follows from the fact that pGCL is

2 probabilistic Guarded Command Language

Turing complete and Rice's theorem. Instead we rely on so-called loop invariant annotations, invariants for short. In the standard case, an annotation is invariant of the loop if its truth value is not changed by the loop. In other words, if the invariant is true before the loop-body is executed we require that it remains true after the loop-body was executed once. By induction, the invariant remains true after the loop-body was repeatedly executed and finally the loop terminates. This enables us to formulate pre- and postconditions regardless of the loop-body code and the number of its repeated executions! The precondition is just the invariant and the postcondition is the invariant conjuncted with the loop's negated guard. Listing 2.8 shows how a loop is annotated using an invariant I .

Listing 2.8: Invariant annotation

```
1  ⟨Θ⟩
2  ⟨I⟩
3  while (G) {
4      ⟨I ∧ G⟩
5      loop_body;
6  }
7  ⟨I ∧ ¬G⟩
```

Definition 2.6 (Invariant). We call an annotation I *loop-invariant* if the following two conditions hold:

- Initiation: I holds before the loop

$$\Theta \Rightarrow I$$

- Consecution: I holds after one execution of the loop body (before another repetition begins or the program execution continues after the loop)

$$I \wedge G \Rightarrow wlp(\text{loop_body}, I)$$

The above conditions are called verification conditions. If we can verify that a given annotation I satisfies the above conditions, we have shown that I is indeed invariant.

As we already mentioned in Section 2.2, the implication relation between predicates turns into the less than or equal to relation between expectations. Analogously, an invariant is an expectation whose value is not decreased by the execution of the loop-body. A significant difference to Definition 2.6 is that there is no probabilistic counter-part of the initiation condition. This means we do not require that the invariant expectation has a certain minimal initial value. The reason for this is that the definition of the pGCL semantics is given in terms of predicate transformations which are applied backwards: from post-expectation to pre-expectation. We have not defined a probabilistic analogon to the standard “strongest postcondition” which is used in forward analysis. In fact, we could also drop the initiation condition in Definition 2.6. But there is a notion of a strongest postcondition for standard programs which we did not discuss here and furthermore the initiation condition sorts out all those annotations that would satisfy the consecution condition only because the premise is false. Such annotations would not be “wrong” but simply meaningless. That is why in the standard case we establish that an invariant is true

initially and stays true thereafter. In the case of probabilistic invariants the value does not need to stay the same throughout the execution of the loop — it just must not decrease.

Definition 2.7 (Probabilistic invariant). We call an expectation \mathcal{I} *loop-invariant* if the following two conditions hold:

- Consecution: $\mathcal{I} \cdot [G] \leq wlp(\text{loop_body}, \mathcal{I})$
- Boundedness: $0 \leq \mathcal{I} \leq 1$

The last condition is new and ensures that the expectation \mathcal{I} is non-negative and bound from above.

Without these bounds we would run into technical problems. To show this, the following two examples from [14] are briefly examined².

Listing 2.9: “Geometric distribution” program

```

b := 1;
n := 0;
while (b >= 1) {
    (b := 1 [0.5] b := 0);
    n := n + 1;
}

```

The program in Listing 2.9 assigns n a value in \mathbb{N} according to the geometrical distribution

$$P(n = k) = p(1 - p)^{k-1}$$

with $p = 0.5$. This program shows why mixed sign expectation are a bad idea in general, cf. Definition 2.3. Assume the post-expectation

$$P = [true] \cdot ((-2)^n).$$

Then the pre-expectation

$$wlp(\text{prog}, P),$$

where prog is the program in Listing 2.9, amounts to

$$\sum_{n \in \mathbb{N}} \frac{(-2)^n}{2^n}.$$

But this term is undefined. In order to avoid such problems — particularly with invariants — the lower boundedness requirement enforces same-sign (non-negative) invariants.

The subsequent “random walk” program in Listing 2.10 motivates an upper bound for invariants.

²We use a shorthand notation for choice here: when both alternatives consist of one statement only, then the whole choice construct is written as one statement in parenthesis. This is a bit shorter and corresponds to the notation in [12].

2 probabilistic Guarded Command Language

Listing 2.10: “Random walk” program

```
n := 1;
while (n > 0) {
    (n := n + 1 [0.5] n := n - 1);
}
```

Without the upper boundedness condition

$$\mathcal{I} = [true] \cdot (n)$$

is invariant. It satisfies the consecution condition

$$[n \neq 0] \cdot (n) \leq [true] \cdot 0.5 \cdot (n + 1) + [true] \cdot 0.5 \cdot (n - 1) = [true] \cdot (n)$$

and is non-negative because the least value of n is zero. However \mathcal{I} leads to a false reasoning. If we ask for the expected value of n we compute

$$wlp(prog, [true] \cdot (n)) = 1$$

which is wrong because the program always ends up with $n = 0$.

Nevertheless unbounded invariants can be useful but require other constraints depending on the program which assure that reasoning with them is sound. For the tool development we stick with the requirement of an upper bound. The price that we pay is that our tool cannot generate interesting invariants for e.g. the “geometric distribution” program. In this context, “interesting” means that the invariant helps to prove that the expected value of n is $1/p$. Such an invariant cannot be bounded because a proof requires to consider all possible $n \in \mathbb{N}$ and their probabilities.

We fixed the verification conditions that tell us when an annotation is loop-invariant but how can we actually find such an annotation? In particular, how can we find annotations that help us in the overall proof? For example $\mathcal{I} = true$ is always a solution but rather useless. This is the main problem of logical inference. Usually one has to guess the invariant for each new program. Even once an annotation was guessed that is supposed to be invariant, this claim has to be proven by hand which is quite tedious. This becomes even worse in the case of probabilistic annotations that feature not only predicates but also weights that add more degrees of freedom.

As mentioned before there cannot be an algorithm that finds all invariants completely automatically for all programs. Nevertheless, methods were developed that assist the human user in an interactive fashion.

In the next chapter we will describe how exactly those constraints are computed.

2.3.2 A note on the “liberal” assumption

In the literature (cf. [6, 14]), there are two notions of weakest preconditions: one with and one without the assumption that the program at hand terminates. The latter being denoted by wp . Of course, if the program terminates, the notions coincide. Otherwise the computed preconditions will be different. For example, the wp of **abort** is always 0 because it does not properly terminate and therefore no given postcondition or post-expectation can be established. The wlp of

abort is always 1 because any post-expectation could be established if only **abort** would terminate (which it does not)³. Since the “non-liberal” pre-expectation requires a termination proof we stick to the liberal and simpler notion. The methods presented here are only used to show partial correctness.

2.4 A Proof Example

We conclude this chapter with a short proof example. Let us, once again, look at an example taken from [12]. Listing 2.11 shows a pGCL program that assigns a natural number from $\{0, \dots, M\}$, $M \in \mathbb{N}$ to x according to the binomial distribution.

Listing 2.11: “Binomial update” program assigns x some value in $\{0, \dots, M\}$

```

x := 0;
n := 0;
while (n - M + 1 <= 0) {
    (x := x + 1 [p] skip);
    n := n + 1;
}

```

Claim 2.1. Our claim is that the expected value of x is $M \cdot p$ (on termination of the program):

$$E(x) = M \cdot p$$

We will use our probabilistic logical inference method to prove this claim.

Listing 2.12: Correctness proof for the program of Listing 2.11

```

1   $\langle [0 \leq M] \cdot (p) \rangle$ 
2   $\langle [0 \geq 0 \wedge 0 \leq M] \cdot (\frac{1}{M} \cdot 0 + p) \rangle$ 
3  x := 0;
4   $\langle [x \geq 0 \wedge 0 \leq M] \cdot (\frac{1}{M}x + p) \rangle$ 
5   $\langle [x \geq 0 \wedge x - 0 \leq 0 \wedge 0 \leq M] \cdot (\frac{1}{M}x - p\frac{1}{M} \cdot 0 + p) \rangle$ 
6  n := 0;
7   $\langle [x \geq 0 \wedge x - n \leq 0 \wedge n \leq M] \cdot (\frac{1}{M}x - p\frac{1}{M}n + p) \rangle$ 
8  while (n - M - 1 <= 0) {
9      (x := x + 1 [p] skip);
10     n := n + 1;
11 }
12  $\langle [x \geq 0 \wedge x - n \leq 0 \wedge n \leq M \wedge n > M - 1] \cdot (\frac{1}{M}x - p\frac{1}{M}n + p) \rangle$ 
13  $\langle [true] \cdot (\frac{1}{M}x) \rangle$ 

```

We can prove Claim 2.1 using annotations as in Listing 2.12. There, we prove that the expected value of $\frac{1}{M}x$ is p . This immediately yields our claim because the expectation is a linear function

³If the premise of an implication is false, then the implication itself is valid

2 probabilistic Guarded Command Language

and we can therefore multiply both sides of

$$E\left(\frac{1}{M}x\right) = p$$

with M and obtain the desired result.

Note, that the proof relies on the assumption that $0 \leq M$, which is trivially true, because “by design” M is a positive constant.

We will show in Chapter 5 how we can find the invariant expectation in line 7. All annotations above line 7 are just applications of the wlp semantics given earlier in Table 2.1. The reasoning behind the step from line 12 to line 13 is: n is initially zero and is only increased by one, thus n is some integer that fulfills

$$n \leq M \wedge n > M - 1.$$

Therefore $n = M$ holds and

$$\frac{1}{M}x - p\frac{1}{M}n + p$$

can be simplified to

$$\frac{1}{M}x.$$

Additionally, for any boolean formula φ it holds, that

$$\varphi \Rightarrow \text{true}.$$

This is why the expectation in line 12 is less than the expectation in line 13.

3 Invariant generation

In this chapter we explain in greater detail how constraint-based invariant generation works. We will first follow [4] and explain invariant generation for non-probabilistic programs. Afterwards we explain the idea how probabilistic programs can be analysed within the same framework [12].

Previously, we have seen examples which show that finding loop invariants is essential to prove a program's property. In general the invariant \mathcal{I} can be any arbitrary boolean combination of inequalities of non-linear and non-polynomial functions like

$$\mathcal{I} = (\exp(x) \cdot \sqrt{y} < 17) \vee (\cos(x) \cdot z^3 \geq 0 \wedge y = -5).$$

However the presented approach is limited to the class of *linear* functions.

Definition 3.1 (Linear Invariant). For *standard* programs, an annotation is called a *linear invariant* if

- it is invariant, cf. Definition 2.6, and
- its atoms are linear inequalities.

For *probabilistic* programs, an annotation is called a *linear invariant* if

- it is invariant, cf. Definition 2.7,
- its predicates' atoms are linear inequalities, and
- its weights are linear expressions.

Strictly speaking, when we write “invariant” annotations we mean so-called *inductive* annotations. This distinction can be found in the book by Manna and Pnueli [13].

Definition 3.2 (Inductive annotation). If a standard / probabilistic annotation satisfies the conditions from Definition 2.6 / 2.7 in all program states and not only the reachable ones, it is called *inductive*.

Let us consider an example.

Listing 3.1: Invariant annotation

```
1 // assume x <= 0
2  $\mathcal{I} = \langle x \leq 10 \rangle$ 
3 while (x < 10) {
4     x := x + 1;
5 }
```

3 Invariant generation

If x takes only integer values in the context of the program in Listing 3.1, then clearly \mathcal{I} is an invariant. It is not inductive because there is an unreachable state, e.g. $x = -0.1$ such that the execution of the loop would terminate at $x = 10.9$ and violate \mathcal{I} .

At first sight one may think that inductive annotations are too restrictive. Nevertheless they are the ones which we are interested in and this has the following reason. Inductive annotations obviously are also invariant but furthermore they are independent of the knowledge of the reachable state space. This makes it easier to find them because we neglect all those invariants that e.g. would rely on assumptions about the integrity of variables, etc. Note, that from now on we stick with the common convention and use “invariant” synonymously with “inductive”.

The last missing definition is that of a template.

Definition 3.3 (Template). A *template* is an annotation where some (or all) coefficients are left unspecified. For example:

$$\mathcal{T} = [\alpha \cdot x + 1 \geq 0] \cdot (\delta \cdot x - \epsilon \cdot y) + [y + \beta \leq 0] \cdot (0.85)$$

is an expectation template where $x, y \in \text{Var}$ and $\alpha, \beta, \delta, \epsilon$ are real valued parameters of the template. Assigning values to these parameters yields an *instantiation* of the template, e.g.

$$\mathcal{T}[\alpha/0.5, \beta/0, \delta/1, \epsilon/2] = [0.5x + 1 \geq 0] \cdot (x - 2y) + [y \leq 0] \cdot (0.85).$$

3.1 Standard programs

We have defined all the parts we need: a language, its semantics, linear (inductive) invariants and templates. It is time to present the technique to synthesise invariants due to Colón et al. [4]. Note that, in their paper they use a slightly different notation. We explain the details using the following simple program:

Listing 3.2: A simple program

```
n := 0;
while (n - M + 1 <= 0) {
    n := n + 1;
}
```

To start with, the presented approach needs two ingredients. First, a program with one loop like the one given above. Second, a template for the desired linear invariant. This means we do not search for some completely unknown invariant but instead we assume it should have a particular “shape”. More precisely, we assume it is a linear inequality build of the program’s variables and coefficients. Our goal is to find constraints on the template’s parameters such that any satisfying evaluation yields a linear invariant. Let us consider the following template for our simple program:

$$\mathcal{T} = \alpha \cdot n + \beta \leq 0.$$

For this program, \mathcal{T} is the most general linear template that we can choose. At this stage we do not know which values are admissible for α and β . The aim is to find constraints on those two parameters that tell us which values to choose for α and β in order to obtain an invariant.

To make things clearer we immediately reveal the results. We will find out that e.g. $\mathcal{T}[\alpha/1, \beta/0]$ is not invariant, whereas $\mathcal{T}[\alpha/-1, \beta/0]$ is invariant. In particular, we will see that every choice for α and β which satisfies

$$\beta \leq 0 \wedge \alpha \leq -\frac{\beta}{M}$$

yields an invariant for the while-loop in Listing 3.2.

Now, let us go back to the starting point: given a program and a template, one has to find the verification condition for this template. Here we apply the wlp-semantics in order to find the right-hand side of the consecution condition. With $\mathcal{T} = \alpha \cdot n + \beta \leq 0$ and the program in Listing 3.2, the initiation condition from Definition 2.6 becomes

$$VC_{init} = \forall n \in \mathbb{R} : \underbrace{n = 0}_{\Theta} \Rightarrow \underbrace{\alpha \cdot n + \beta \leq 0}_I$$

and the (more interesting) consecution condition turns into

$$VC_{consec} = \forall n \in \mathbb{R} : \underbrace{\alpha \cdot n + \beta \leq 0}_I \wedge \underbrace{n - M + 1 \leq 0}_G \Rightarrow \underbrace{\alpha \cdot n + \alpha + \beta \leq 0}_{wlp(n:=n+1, \alpha \cdot n + \beta \leq 0)}$$

Remember that variables in pGCL are always real valued and since we are looking for an inductive instantiation of \mathcal{T} , the universal quantification of n is over the real numbers (even though n takes only integer values in the context of the analysed program).

For later use we define the conjunction VC in order to refer to both conditions simultaneously:

$$VC = VC_{init} \wedge VC_{consec}$$

The sought parameters depend only on the constraints expressed by VC above. That is why the whole approach is called constraint-based. Having formulated the verification conditions we are finished, in the sense that our solution, i.e., the set of admissible values of α and β is implicitly given by the verification conditions. Any instantiation of α and β that makes VC valid, yields an invariant.

In order to extract explicit constraints for the parameters of \mathcal{T} , Colón et al. [4] suggest the following two-phase approach.

1. The universally quantified verification conditions are transformed into first-order logic formulas where the program variables are not present any more and fresh existentially quantified parameters are introduced.
2. Elimination of the existential quantifiers from the formulas results in quantifier-free first-order formulas that only depend on the template's parameters (here α and β).

For the first step, one has to make sure that the verification conditions are in conjunctive normal form (CNF). We remove implication as usual¹ and discover that VC is already in CNF.

$$VC \equiv \forall n \in \mathbb{R} : \underbrace{(n \neq 0 \vee \alpha \cdot n + \beta \leq 0)}_{VC_{init}} \wedge \underbrace{(-\alpha \cdot n - \beta < 0 \vee -n + M - 1 < 0 \vee \alpha \cdot n + \alpha + \beta \leq 0)}_{VC_{consec}}$$

¹ $a \Rightarrow b \equiv \neg a \vee b$

3 Invariant generation

Then, using the equivalence

$$\begin{aligned}
VC &\equiv \forall n \in \mathbb{R} : \bigwedge_i \bigvee_j \text{ineq}_{ij} \\
&\equiv \forall n \in \mathbb{R} : \bigwedge_i \neg \bigwedge_j \neg \text{ineq}_{ij} \\
&\equiv \bigwedge_i \forall n \in \mathbb{R} : \neg \bigwedge_j \neg \text{ineq}_{ij}
\end{aligned}$$

we rewrite each clause as a system of negated linear strict and non-strict inequalities:

- $\forall n \in \mathbb{R} : \begin{pmatrix} 1 \cdot n + 0 & \leq & 0 \\ -1 \cdot n + 0 & \leq & 0 \end{pmatrix} \wedge (-\alpha \cdot n - \beta < 0)$
- $\forall n \in \mathbb{R} : \begin{pmatrix} \alpha \cdot n + \beta & \leq & 0 \\ 1 \cdot n - M + 1 & \leq & 0 \end{pmatrix} \wedge (-\alpha \cdot n - (\alpha + \beta) < 0)$

This representation of the verification conditions allows for the application of Motzkin's Transposition Theorem [12, 2]. This theorem provides a method to express the universally quantified verification conditions above using existential quantification. It is a generalisation of "Farkas Lemma" which was used in [4].

Theorem 3.1 (Motzkin's Transposition Theorem). Given the set of (non-strict and strict) linear inequalities over real-valued variables x_1, \dots, x_n

$$\begin{aligned}
S &:= \begin{bmatrix} \alpha_{(1,1)}x_1 & + \dots + \alpha_{(1,n)}x_n & + \beta_1 \leq 0 \\ \vdots & & \vdots \\ \alpha_{(m,1)}x_1 & + \dots + \alpha_{(m,n)}x_n & + \beta_m \leq 0 \end{bmatrix} \\
T &:= \begin{bmatrix} \alpha_{(m+1,1)}x_1 & + \dots + \alpha_{(m+1,n)}x_n & + \beta_{m+1} < 0 \\ \vdots & & \vdots \\ \alpha_{(m+k,1)}x_1 & + \dots + \alpha_{(m+k,n)}x_n & + \beta_{m+k} < 0 \end{bmatrix},
\end{aligned}$$

in which $\alpha_{(1,1)}, \dots, \alpha_{(m+k,n)}$ and $\beta_1, \dots, \beta_{m+k}$ are real-valued, we have that S and T simultaneously are *not* satisfiable (i.e. they have no solution in x_1, \dots, x_n) if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, \dots, \lambda_{m+k}$ such that either

$$0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, \quad \dots, \quad 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, \quad 1 = \left(\sum_{i=1}^{m+k} \lambda_i \beta_i \right) - \lambda_0,$$

or at least one coefficient λ_i for i in the range $[m+1 \dots m+k]$ is non-zero and

$$0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, \quad \dots, \quad 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, \quad 0 = \left(\sum_{i=1}^{m+k} \lambda_i \beta_i \right) - \lambda_0.$$

This theorem characterises the *non-satisfiability* of S and T . This is why we negated the inequalities before writing them as an inequality system. According to the theorem our verification conditions become:

$$\begin{aligned} \varphi := \exists \lambda_0, \dots, \lambda_3 : & \quad (\lambda_0 \geq 0 \wedge \dots \wedge \lambda_3 \geq 0 \wedge 0 = \lambda_1 - \lambda_2 - \alpha \lambda_3 \\ & \quad \wedge 1 = -\beta \lambda_3 - \lambda_0) \\ \vee & \quad (\lambda_0 \geq 0 \wedge \dots \wedge \lambda_3 \geq 0 \wedge \lambda_3 \neq 0 \wedge 0 = \lambda_1 - \lambda_2 - \alpha \lambda_3 \\ & \quad \wedge 0 = -\beta \lambda_3 - \lambda_0) \end{aligned}$$

$$\begin{aligned} \psi := \exists \lambda_0, \dots, \lambda_3 : & \quad (\lambda_0 \geq 0 \wedge \dots \wedge \lambda_3 \geq 0 \wedge 0 = \alpha \lambda_1 + \lambda_2 - \alpha \lambda_3 \\ & \quad \wedge 1 = (\beta \lambda_1 + (-M + 1)\lambda_2 + (-\alpha - \beta)\lambda_3) - \lambda_0) \\ \vee & \quad (\lambda_0 \geq 0 \wedge \dots \wedge \lambda_3 \geq 0 \wedge \lambda_3 \neq 0 \wedge 0 = \alpha \lambda_1 + \lambda_2 - \alpha \lambda_3 \\ & \quad \wedge 0 = (\beta \lambda_1 + (-M + 1)\lambda_2 + (-\alpha - \beta)\lambda_3) - \lambda_0) \end{aligned}$$

This completes the first step of the technique by Colón et al. We proceed with step two. Quantifier elimination is the transformation of a quantified formula into an equivalent formula which is quantifier-free and contains only free variables. If all variables were quantified, then the result is either *true* or *false*. This operation can be seen as a simplification step. Quantifier elimination for first-order formulas over a real closed field is doubly exponential in the number of quantifiers in the worst-case [3, 5]. The tool REDLOG² performs this operation automatically and — for our input instances — reasonably fast. The formulas φ and ψ are given to REDLOG and it computes the following equivalent quantifier-free formulas:

$$\beta \leq 0 \vee \alpha \cdot \beta < 0 \wedge \alpha > 0$$

and

$$\begin{aligned} & (\alpha \cdot M + \beta \leq 0 \vee \alpha < 0 \\ \vee & \quad \alpha^2 \cdot M^2 - \alpha^2 \cdot M + 2 \cdot \alpha \cdot \beta \cdot M - \alpha \cdot \beta + \beta^2 > 0 \wedge \alpha = 0 \\ \vee & \quad \alpha^2 \cdot M - \alpha^2 + \alpha \cdot \beta \leq 0 \wedge \alpha \cdot M - \alpha + \beta > 0). \end{aligned}$$

This result is not very readable and can be further simplified using the fact that both formulas have to be true simultaneously and M is a positive constant. We end up with the following constraints on α and β :

$$\beta \leq 0 \wedge \alpha \leq -\frac{\beta}{M} \tag{3.1}$$

This concludes the second step. Above is a compact and explicit representation of the verification conditions. Every satisfying instantiation of α and β yields an invariant annotation. It is up to the user to choose the parameters. Constraint (3.1) is depicted in Figure 3.1: every pair (α, β) that lies in the shaded area satisfies this constraint. If we were to prove a property, say “whenever the program terminates, n is no less than zero”, we choose $\alpha = -1$ and $\beta = 0$ and use the invariant $-n \leq 0$ in the proof. We might as well want to show an upper bound on n and choose $\beta = -1$ and $\alpha = \frac{1}{M}$ and use the invariant $\frac{1}{M} \cdot n - 1 \leq 0 \Leftrightarrow n \leq M$. Of course, this is a toy

²<http://redlog.dolzmann.de/>

3 Invariant generation

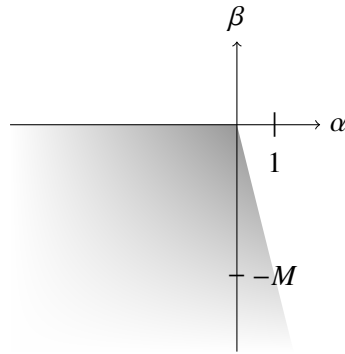


Figure 3.1: All points in the shaded region satisfy formula (3.1)

example but it clearly shows the methods behind invariant generation and possible application of invariants.

Figure 3.2 visualises the explained constraint-based linear invariant generation approach. It is important to point out that all transformations from the verification conditions to the constraints on the template's parameters are equivalence transformations. This means that our method is complete in the following sense:

Theorem 3.2 (Completeness of constraint-based invariant generation). Given a standard linear program and a linear template \mathcal{T} . The method due to Colón et al. [4] yields constraints that characterise precisely the invariant instantiations of the template. In particular, if \mathcal{T} does not have any invariant instantiations, then the constraints are unsatisfiable.

Theorem 3.2 does neither state that we necessarily discover all linear invariants because that depends on the given template nor does it claim that linear programs have only linear invariants.

In the presentation of the invariant generation method we had to compute the wlp of a loop body. In order to do this, nested loops are not allowed. Certainly this is a limitation because in practice nested loops do occur. However there are many interesting programs with only one loop that have to be mastered before we try to analyse nested loops.

3.2 Probabilistic programs

In the approach shown above the verification conditions were first-order formulas. It was shown how invariants can be derived from those formulas using Motzkin's Transposition Theorem and quantifier elimination.

In the case of probabilistic programs, the verification conditions of an invariant expectation become numerical expressions which we relate using inequality, as explained in the preceding chapter.

Our aim now is to find a way to transform the verification conditions of probabilistic invariants, i.e. inequalities between expectations, into first-order formulas. Then we are able to reuse the technique above. But is it possible to encode that one expectation is no greater than another

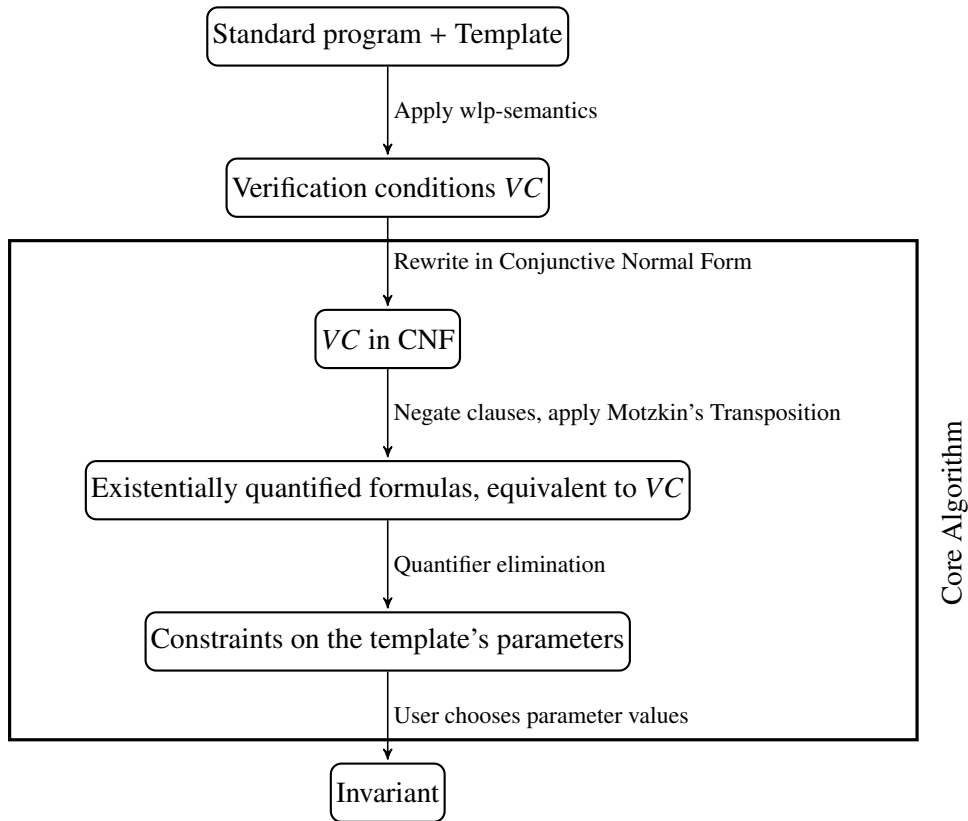


Figure 3.2: Method that generates invariants from templates

3 Invariant generation

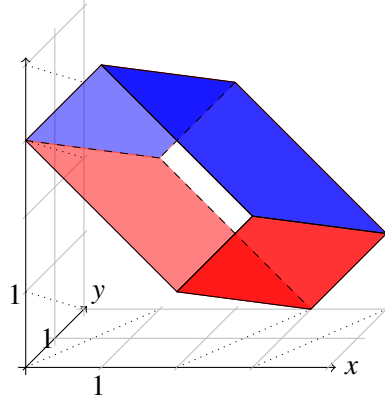


Figure 3.3: Comparison of expectations

using only boolean predicates? Yes, it is. We demonstrate this using the following inequality between expectations:

$$\begin{aligned}
 & [0 \leq 2x - y \leq 4 \wedge 0 \leq y \leq 2] \cdot (-x + 3) \\
 & \quad + [4 < 2x - y \leq 6 \wedge 0 \leq y \leq 2] \cdot (x - y - 1) \\
 \leq & [0 \leq 2x - y \leq 2 \wedge 0 \leq y \leq 2] \cdot (x - y + 3) \\
 & \quad + [2 < 2x - y \leq 6 \wedge 0 \leq y \leq 2] \cdot (-x + 5)
 \end{aligned} \tag{3.2}$$

In order to help understanding, we visualise the surfaces that are represented by the two expectations, cf. Figure 3.3. It happens to be that the four faces constitute the sides of a prism in 3-dimensional space. The inequality holds, if we can show that for every pair (x, y) the blue faces are not below the red ones.

First, observe that the predicates of expectation (3.2) are mutually exclusive, i.e. if $0 \leq 2x - y \leq 4 \wedge 0 \leq y \leq 2$ holds, then $4 < 2x - y \leq 6 \wedge 0 \leq y \leq 2$ is false and the other way round. The same is true for the predicates of expectation (3.3). This means that we can reason about each summand separately because for all values of x and y only one summand contributes a weight to the expectation's total value.

Definition 3.4 (Disjoint Normal Form). An expectation $E = [P_1] \cdot w_1 + \dots + [P_n] \cdot w_n$ is in *disjoint normal form* (DNF), if its predicates are pairwise disjoint, i.e.

$$\forall 1 \leq i < j \leq n : P_i \wedge P_j \equiv \text{false}$$

The idea behind the transformation of an inequality between expectations (as above) to a boolean formula is to compare the expectations' weights in a pairwise fashion. For example, if the predicate $0 \leq 2x - y \leq 4 \wedge 0 \leq y \leq 2$ from (3.2) and $0 \leq 2x - y \leq 2 \wedge 0 \leq y \leq 2$ from (3.3) both hold, then, accordingly $-x + 3$ has to be at most $x - y + 3$. Formally, this can be written as a formula:

$$(0 \leq 2x - y \leq 4 \wedge 0 \leq y \leq 2 \wedge 0 \leq 2x - y \leq 2 \wedge 0 \leq y \leq 2) \Rightarrow (-x + 3) \leq (x - y + 3).$$

In that way we can express for all four pairs of predicates that the corresponding weight from expectation 3.2 is not greater than the one from expectation 3.3. Additionally it must be ensured that if none of the predicates of 3.3 are satisfied while there is one predicate in 3.2 which is true, then the weight must not be greater than zero.

Lemma 3.1. Given two expectations in disjoint-normal form

$$\mathcal{I} = [P_1] \cdot u_1 + \dots + [P_M] \cdot u_M$$

$$\mathcal{J} = [Q_1] \cdot w_1 + \dots + [Q_K] \cdot w_K$$

The inequality $\mathcal{I} \leq \mathcal{J}$ holds if and only if

$$\begin{aligned} \forall x_1, \dots, x_n \in \mathbb{R} : & \bigwedge_{m \in \overline{M}} \bigwedge_{k \in \overline{K}} (P_m \wedge Q_k \Rightarrow (u_m - w_k \leq 0)) \\ & \wedge \bigwedge_{m \in \overline{M}} \left(P_m \wedge \left(\bigwedge_{k \in \overline{K}} \neg Q_k \right) \Rightarrow u_m \leq 0 \right) \end{aligned}$$

holds, where \overline{X} is the set of indices $\{1, 2, \dots, X\}$.

Using Lemma 3.1, the inequality between expectations (3.2) and (3.3) is transformed into the following formula:

$$\begin{aligned} \forall x, y \in \mathbb{R} : & ((0 \leq 2x - y \leq 4 \wedge 0 \leq y \leq 2) \wedge (0 \leq 2x - y \leq 2 \wedge 0 \leq y \leq 2) \\ & \Rightarrow -2x + y \leq 0) \\ \wedge & ((0 \leq 2x - y \leq 4 \wedge 0 \leq y \leq 2) \wedge (2 < 2x - y \leq 6 \wedge 0 \leq y \leq 2) \\ & \Rightarrow -2 \leq 0) \\ \wedge & ((4 < 2x - y \leq 6 \wedge 0 \leq y \leq 2) \wedge (0 \leq 2x - y \leq 2 \wedge 0 \leq y \leq 2) \\ & \Rightarrow -4 \leq 0) \\ \wedge & ((4 < 2x - y \leq 6 \wedge 0 \leq y \leq 2) \wedge (2 < 2x - y \leq 6 \wedge 0 \leq y \leq 2) \\ & \Rightarrow 2x - y - 6 \leq 0) \\ \wedge & ((0 \leq 2x - y \leq 4 \wedge 0 \leq y \leq 2) \wedge \neg(0 \leq 2x - y \leq 2 \wedge 0 \leq y \leq 2) \\ & \wedge \neg(2 < 2x - y \leq 6 \wedge 0 \leq y \leq 2) \Rightarrow -x + 3 \leq 0) \\ \wedge & ((4 < 2x - y \leq 6 \wedge 0 \leq y \leq 2) \wedge \neg(0 \leq 2x - y \leq 2 \wedge 0 \leq y \leq 2) \\ & \wedge \neg(2 < 2x - y \leq 6 \wedge 0 \leq y \leq 2) \Rightarrow x - y - 1 \leq 0) \end{aligned}$$

This is a first-order formula just like the verification conditions *VC* in the previous section, consequently it can be further transformed (cf. Figure 3.2, Core Algorithm) and after the quantifier elimination we are given the very simple and satisfying result:

true

which means that the inequality holds, as confirmed by Figure 3.3.

3 Invariant generation

Note that we rely on the fact that both expectations are in disjoint normal norm. Of course, an expectation is not always given in this form. But there is a straight-forward transformation of an expectation into an equivalent one in DNF. If an expectation E is not in DNF this means, that there are summands $[P_i] \cdot w_i$ and $[P_j] \cdot w_j$ which “overlap”, i.e. there are states that satisfy both predicates. For a DNF we have to find a predicate that characterizes this “overlapping” region, a predicate for states in P_i but not in P_j and one for states in P_j but not in P_i . Naturally, we can then replace the two summands with three: $[P_i \wedge \neg P_j] \cdot w_i$, $[P_j \wedge \neg P_i] \cdot w_j$ and $[P_i \wedge P_j] \cdot (w_i + w_j)$. In general we apply the following theorem:

Theorem 3.3 (Transformation to DNF). Given an expectation of the form

$$E = [P_1] \cdot w_1 + \dots + [P_n] \cdot w_n.$$

Then an equivalent expectation in DNF can be written as:

$$\sum_{I \in \mathcal{P}(\bar{n}) \setminus \emptyset} \left(\left[\bigwedge_{i \in I} P_i \wedge \neg \left(\bigwedge_{j \in \mathcal{P}(\bar{n}) \setminus I} P_j \right) \right] \cdot \left(\sum_{i \in I} w_i \right) \right)$$

where \bar{n} is the index set $\{1, \dots, n\}$ and $\mathcal{P}(\cdot)$ denotes the power set.

One can see immediately that this transformation yields an exponentially longer expectation. The practical impacts and possible alternatives are discussed in Chapter 4.

At this point we save the reader from an analysis of a probabilistic program because the verification conditions become totally unreadable after the transformation to DNF and application of Lemma 3.1. Instead we proceed to the next chapters where we finally discuss our implementation and show how our tool saves us a lot of work.

We summarise the explained method for probabilistic invariant generation by means of a flowchart in Figure 3.4. Note, how we reuse techniques from non-probabilistic invariant generation of the previous section. Since all transformations on the way from probabilistic verification conditions to first order formulas are equivalence transformations, again we conclude that the overall approach is complete in the sense of Theorem 3.2.

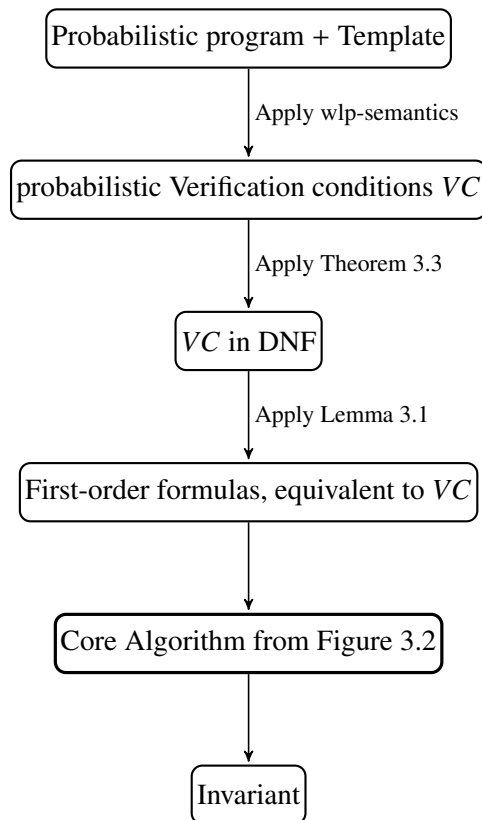


Figure 3.4: Method that generates *probabilistic* invariants from templates

4 Implementation

This chapter describes how our tool works. In particular, differences between the approach in [12] and the concrete implementation are pointed out.

Our tool needs only two prerequisites: a Java interpreter and an installation of the “REDUCE Computer Algebra System”¹ which provides the aforementioned package “REDLOG”. All of the software is freely available. Before we introduce any further implementation details, a brief tutorial shows the workflow of our tool.

4.1 Tutorial

Let us first have a look at the tool from a user’s perspective. Started in interactive mode we are presented a prompt.

Enter a command. Type ‘m’ to see a list of available commands.

Entering ‘m’ gives us a list of all commands available and we will explain them one by one in this tutorial.

Please choose an action:

```
[l] load a pGCL file
[g] generate a program graph for the currently loaded program
[t] enter a template invariant
[a] enter assumptions
[i] generate constraints for a valid invariant
[q] quit
```

To start with, we need a program with a loop. Therefore we press ‘l’ and get a list of all programs that are in the default folder for pGCL programs. Here let us choose, for example, the binomial update program. We could, as well type a path to the program file which should be loaded. The tool notifies the user that the file has been successfully loaded.

```
pgcl_programs/binomial_update_alt.pgcl successfully parsed!
```

At this point, one might like to verify that the pGCL program was parsed correctly and the data structures built by our parser correspond to the program’s control flow graph. Pressing ‘g’ gives us a textual representation of a graph. This text can be copied into the Graphviz software which then generates a picture of this graph, cf. Figure 4.1. The location without incoming edges is

¹REDUCE homepage: <http://www.reduce-algebra.com/>

4 Implementation

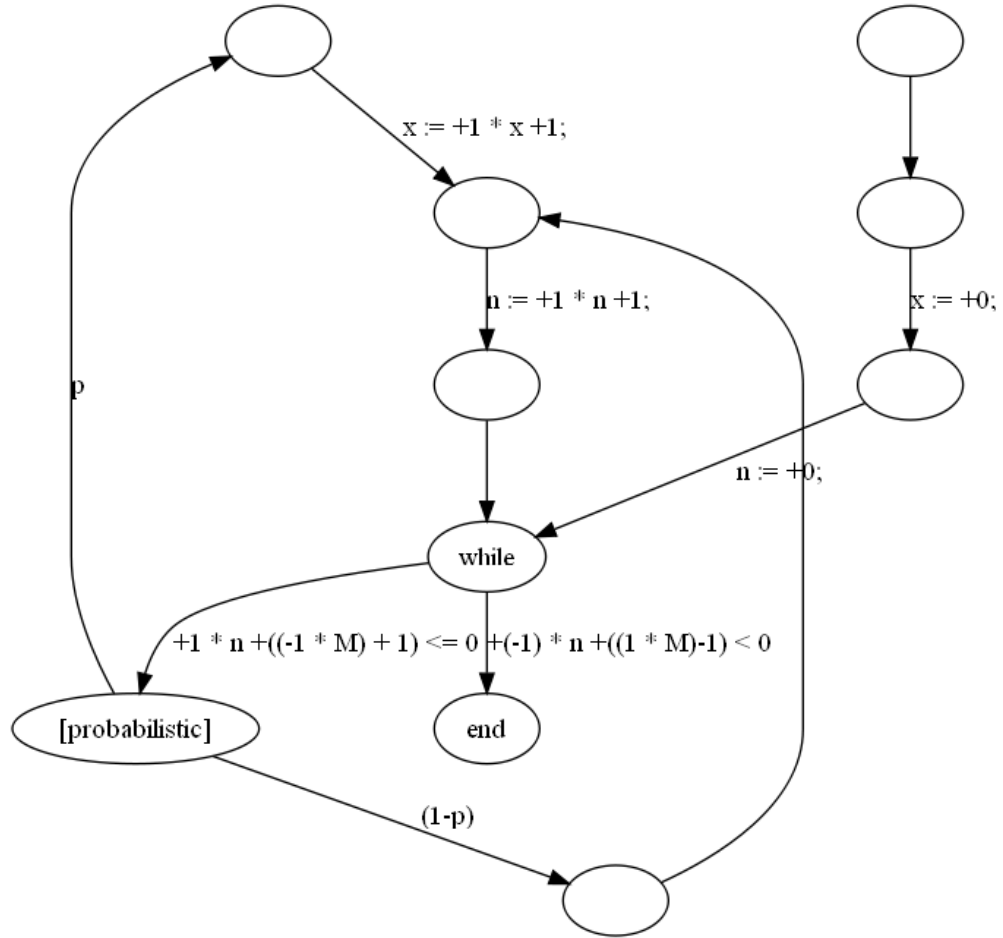


Figure 4.1: Program graph for the Binomial Update program, generated from internal data structures

the starting location. We follow the control flow from this location on. There are different kinds of edge labels. Some edges are labelled with an assignment like “ $n := n + 1;$ ” which means that this assignment takes place when the program moves from the edge’s source location to its target location. Edges can also be labelled with a guard. For instance, in order to move from the “while” location to the “end” location the condition “ $-n + M - 1 < 0$ ” must be satisfied. Edges, that leave a location where a choice is made probabilistically, are labelled with the probability of being taken. Unlabelled edges have no effect or condition. Having convinced ourselves that the program graph looks as intended we proceed to the next step and enter a template. To do so, type ‘t’ at the prompt and enter a parameterised expectation².

²In the current version of the tool a template consists only of one summand because this is enough for all examples considered.

Enter a template of the form [linear term ≤ 0 and linear term $< 0 \dots$
 $[\alpha \cdot x \leq 0] \cdot (\beta)$

Using the template $\mathcal{T} = [\alpha \cdot x \leq 0] \cdot (\beta)$, we enter 'i' and the tool responds with

```
18.11.2010 16:14:30 constraintGenerator.DirectConstraintGenerator...
INFO: Quantifier Elimination using REDLOG took 140 ms
18.11.2010 16:14:30 java.util.logging.LogManager$RootLogger log
INFO: Consecution condition yields:
```

$\alpha \leq 0$ or $\beta \cdot p \leq 0$

```
18.11.2010 16:14:30 constraintGenerator.DirectConstraintGenerator...
INFO: Quantifier Elimination using REDLOG took 138 ms
18.11.2010 16:14:30 java.util.logging.LogManager$RootLogger log
INFO:  $0 \leq 1$  condition yields:
```

$\beta \geq 0$

```
18.11.2010 16:14:30 constraintGenerator.DirectConstraintGenerator...
INFO: Quantifier Elimination using REDLOG took 158 ms
18.11.2010 16:14:30 java.util.logging.LogManager$RootLogger log
INFO:  $1 \leq 1$  condition yields:
```

$\beta - 1 \leq 0$

Here, we see a small part of the log messages which are produced during execution. It shows the time that it took to get the quantifier-free formulas from REDLOG and for each verification condition we see the corresponding constraint on the template's parameters. For instance, if we choose some negative α and some negative β , then the consecution condition from Definition 2.7 is satisfied but the (lower) boundedness condition is violated because it requires $\beta \geq 0$. For this particular example we can already see that $\alpha = -1$ and $\beta = 1$ is a satisfying assignment which yields the invariant

$$\mathcal{T}[\alpha/-1, \beta/1] = [x \geq 0] \cdot (1).$$

It tells us that the variable x is positive, if it was positive before the execution of the loop — independent of the probability p .

The last feature of our tool is to support the user in finding such satisfying assignments. This feature is used by entering the command 'a'. Then the three generated constraints are combined into one formula (using conjunction) and REDLOG's formula simplification method *rlgsn* is applied to that formula. Additionally, the user may provide so called assumptions. For example we know that p is a probability and therefore lies between zero and one and we might assume that β should be negative (by only looking at the consecution condition):

4 Implementation

```
Enter your assumption in REDLOG syntax, i.e. a comma separated l...
0<=p, p<=1, beta<0
Simplifying constraints with your assumption yields:
    false
```

The constraints are simplified to *false* because our assumption $\beta < 0$ contradicts them (cf. lower boundedness constraint). From this we learn that $\beta \geq 0$ must hold if the formula is satisfiable at all. Using such assumptions one can approach the solution step by step when the constraints are unmanageable otherwise.

We have presented our interactive tool and have shown its workflow for invariant generation. In the following we take a look “under the hood” and discuss the implementation. In particular, we point out the differences to the methods in [12] which were explained in the previous chapter. Problems and ideas for future work are also discussed.

4.2 Overview

Figure 4.2 gives an overview over the control flow of the tool chain. We will take a look at the functionality of the parser and the transformations in the subsequent sections. The figure shows how REDLOG is incorporated as a black box for two tasks: quantifier elimination and simplification modulo assumptions. Currently the communication is done by generating a REDLOG input file and starting REDLOG with Java’s `Runtime.exec()` method. The result is then extracted from the returned `InputStream` object. Although this approach to data exchange is not efficient for frequent communication, it is fast enough for our tasks. More than that, it provides the developer with the possibility to look into the the generated input, change it and replay the interaction with REDLOG independently of our tool. Together with the detailed log-file, this helps debugging and testing the tool.

4.3 Parser

In addition to the grammar rules (cf. Definition 2.1) we require that each pGCL file begins with a declaration of variables. The declaration is simply a sequence of assignments. The variable declaration helps the parser to distinguish program variables from constants.

We used the popular parser generator tool ANTLR³ to build our parser. The benefits of doing so are the flexibility of ANTLR which generates the parser from grammar and tree grammar files. This way changes to the language can be easily incorporated. And of course we saved a considerable amount of time since the only thing we had to implement ourselves were the data structures that are instantiated during the parsing process — the lexical analysis, parsing and error handling are done completely by the ANTLR generated code.

The textual pGCL file is represented by its control flow graph, also called program graph. This facilitates the straight forward computation of wlp for the loop body and a given template. The wlp computation is implemented recursively.

³<http://www.antlr.org/>

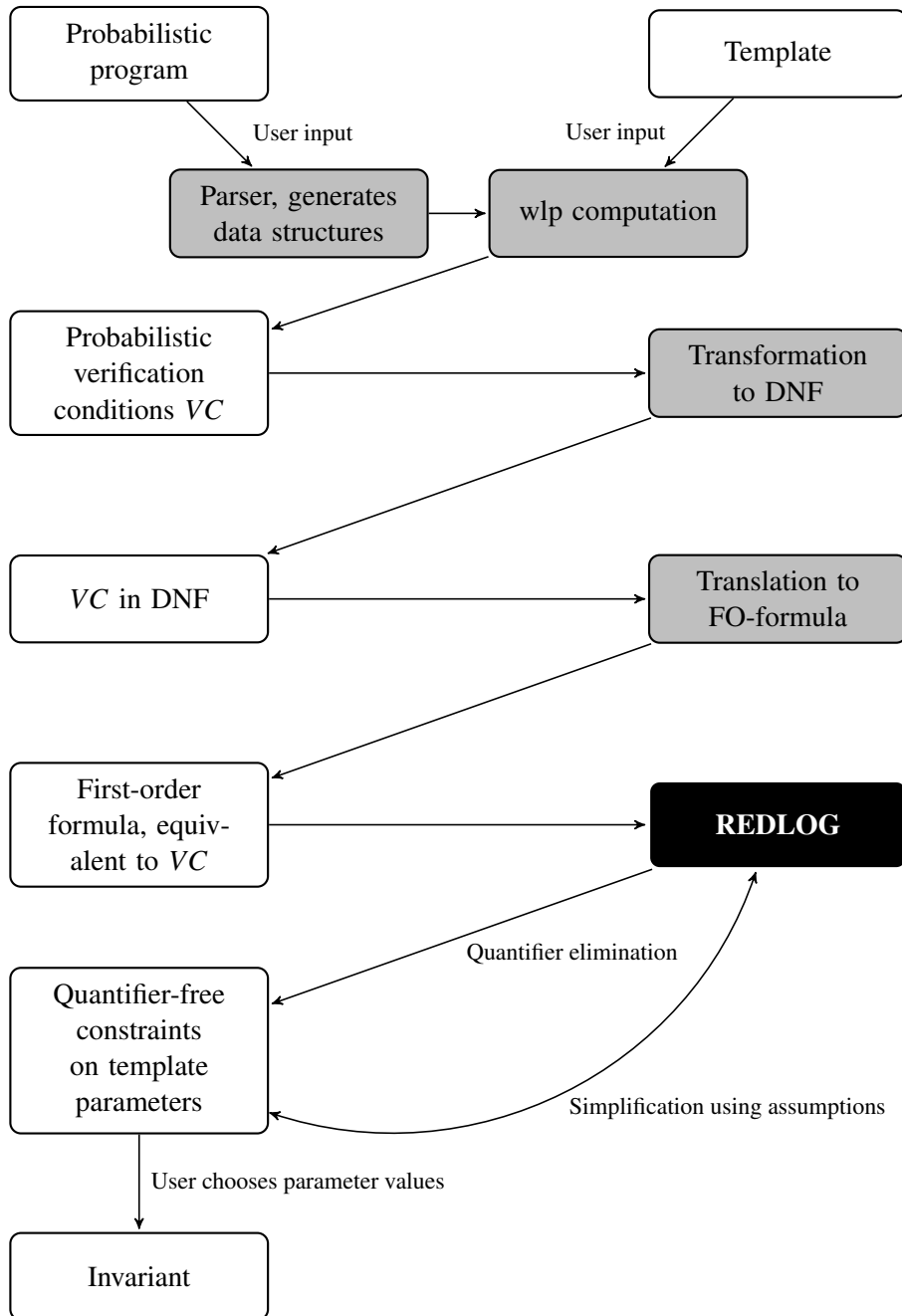


Figure 4.2: Program flow for probabilistic invariant generation. White nodes represent data, grey nodes represent operations of our tool. The external tool REDLOG is shown as a “black box”.

4 Implementation

Since the verification condition for an invariant depend only on the while-loop, the pGCL code before and after the loop is neglected during the invariant generation process. Starting at the “while” location, the loop is traversed forward. According to the wlp semantics (cf. Table 2.1) the wlp expression is recursively nested until the same “while” location is reached again. From there the template is returned and the wlp transformations are applied to it until we are back at the starting point.

We give an example for the automated wlp computation. Consider the program graph in Figure 4.1. Assume we were to compute the pre-expectation

$$E = wlp(x := x + 1[p]skip; n := n + 1, \mathcal{T}),$$

where $\mathcal{T} = [\alpha \cdot x \leq 0] \cdot (\beta)$.

The first statement of the loop body is the probabilistic choice. We apply the semantics and get

$$E = p \cdot wlp(x := x + 1; n := n + 1, \mathcal{T}) + (1 - p) \cdot wlp(skip; n := n + 1, \mathcal{T}).$$

Then we resolve the sequential composition which gives us

$$E = p \cdot wlp(x := x + 1, wlp(n := n + 1, \mathcal{T})) + (1 - p) \cdot wlp(skip, wlp(n := n + 1, \mathcal{T})).$$

Both paths have reached the while location again. Now we go backwards and actually apply the transformations to the template. First n is increased:

$$\begin{aligned} E &= p \cdot wlp(x := x + 1, wlp(n := n + 1, [\alpha \cdot x \leq 0] \cdot (\beta))) \\ &\quad + (1 - p) \cdot wlp(skip, wlp(n := n + 1, [\alpha \cdot x \leq 0] \cdot (\beta))) \\ &= p \cdot wlp(x := x + 1, [\alpha \cdot x \leq 0] \cdot (\beta)) + (1 - p) \cdot wlp(skip, [\alpha \cdot x \leq 0] \cdot (\beta)) \end{aligned}$$

Of course, this has no effect, because n is not present in the template. The next step is to increase x (only in the first summand) while the second summand stays as it is because **skip** has no effect.

$$E = p \cdot [\alpha \cdot x + \alpha \leq 0] \cdot (\beta) + (1 - p) \cdot [\alpha \cdot x \leq 0] \cdot (\beta)$$

And this is the final result which is inserted into the consecution condition.

4.4 From probabilistic verification conditions to FO-formulas

In the original paper [12] an expectation’s predicate is always a conjunction of inequalities. We instead allow any boolean combination of inequalities for the expectations’ predicates, cf. Definition 2.3. This facilitates a more compact representation of expectations and therefore increases the efficiency of the transformation to DNF. The difference is best explained with a small example. Assume we were to compute the DNF for the following expectation:

$$\begin{aligned} E &= [y \geq -1 \wedge y - x + 2 \geq 0 \wedge y + x + 2 \geq 0] \cdot (0.5) \\ &\quad + [y \leq 1 \wedge y + x - 2 \leq 0 \wedge y - x - 2 \leq 0] \cdot (0.5) \end{aligned} \tag{4.1}$$

4.4 From probabilistic verification conditions to FO-formulas

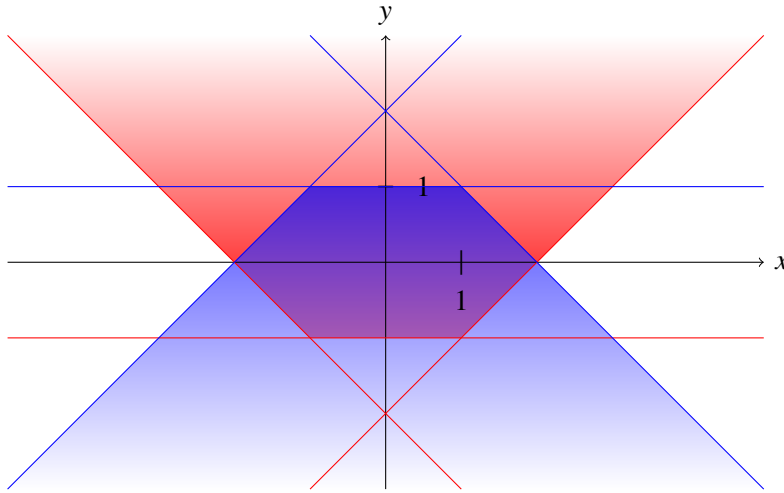


Figure 4.3: Expectation 4.1

Figure 4.3 gives us an idea for which points on the x - y plane E has a non-zero value. The application of Theorem 3.3 yields an expectation with $2^n - 1$ summands, where n is the number of summands of the original expectation. In this case we obtain 3 summands:

$$\begin{aligned}
 E_{DNF} = & [y \geq -1 \wedge y - x + 2 \geq 0 \wedge y + x + 2 \geq 0 \\
 & \wedge (y > 1 \vee y + x - 2 > 0 \vee y - x - 2 > 0)] \cdot (0.5) \\
 & + [y \leq 1 \wedge y + x - 2 \leq 0 \wedge y - x - 2 \leq 0 \\
 & \wedge \neg(y \geq -1 \wedge y - x + 2 \geq 0 \wedge y + x + 2 \geq 0)] \cdot (0.5) \\
 & + [y \geq -1 \wedge y - x + 2 \geq 0 \wedge y + x + 2 \geq 0 \\
 & \wedge y \leq 1 \wedge y + x - 2 \leq 0 \wedge y - x - 2 \leq 0] \cdot (1)
 \end{aligned}$$

Of course we can represent E_{DNF} as well using only conjunction. But with conjunctions of linear inequalities a predicate can only characterise convex spaces. This means we have to characterise each of the 13 coloured convex areas in Figure 4.3 separately. For larger expectations this would lead to unmanageably long expectations in DNF.

Given the DNF of the expectations on the right and left hand sides of the inequality relation it was proposed to transform this (numerical) verification condition into FO-formulas (which are not numerical any more but either true or false). The idea in [12], which was implemented in the tool, is to compare the expectations summand-wise, cf. Lemma 3.1. In a nutshell, the idea is, if a predicate is true on the left hand side and one is true on the right hand side then the corresponding weights have to satisfy the “at most” relation. Additionally, if a predicate is satisfied on the left hand side and none on the right hand side, then the weight must not exceed zero. The size of the resulting formula can be expressed in the number of summands of the expectations. Say the first one has M summands and the second has K . Then the resulting formula will have $M \cdot K$ clauses for the pairwise comparison plus an additional M subformulas for the “non-positivity” condition. An interesting question is whether we really need all these transformations which give

4 Implementation

us an exponential blow up. Maybe there is a different way to translate probabilistic verification conditions to first-order formulas which gives more compact formulas.

Recently we found a new way. It has not been implemented yet, so we can only speculate on its effectiveness. But clearly the implementation of this new transformation approach belongs to the very near future work.

Theorem 4.1. Given two expectations *not necessarily* in disjoint-normal form

$$\mathcal{I} = [P_1] \cdot u_1 + \dots + [P_M] \cdot u_M$$

$$\mathcal{J} = [Q_1] \cdot w_1 + \dots + [Q_K] \cdot w_K$$

The inequality $\mathcal{I} \leq \mathcal{J}$ holds if and only if φ is valid where

$$\begin{aligned} \varphi = & \forall x_1, \dots, x_n \in \mathbb{R} \exists y_1, \dots, y_M, z_1, \dots, z_K \in \mathbb{R} : \\ & \bigwedge_{m \in \overline{M}} (P_m \Rightarrow (y_m = u_m) \wedge \neg P_m \Rightarrow (y_m = 0)) \\ & \wedge \bigwedge_{k \in \overline{K}} (Q_k \Rightarrow (z_k = w_k) \wedge \neg Q_k \Rightarrow (z_k = 0)) \\ & \wedge \sum_{m \in \overline{M}} y_m \leq \sum_{k \in \overline{K}} z_k \end{aligned}$$

and where x_1, \dots, x_n are the program variables, $y_1, \dots, y_M, z_1, \dots, z_K$ are fresh variables, i.e. they neither occur in the analysed program as variables nor as constants and \overline{X} denotes the set of indices $\{1, 2, \dots, X\}$.

Proof.

Case 1: $\mathcal{I} \leq \mathcal{J}$ does hold. We have to show that φ is a tautology. Therefore, fix an arbitrary tuple $x = (x_1, \dots, x_n)$ and show that there exist $y_1, \dots, y_M, z_1, \dots, z_K$ such that φ is true.

Case 1.1: $P_m(x) = \text{true}$. Then there exists $y_m = u_m$ and hence $P_m \Rightarrow (y_m = u_m) \wedge \neg P_m \Rightarrow (y_m = 0)$ is fulfilled.

Case 1.2: $P_m(x) = \text{false}$. Then there exists $y_m = 0$ and again $P_m \Rightarrow (y_m = u_m) \wedge \neg P_m \Rightarrow (y_m = 0)$ is fulfilled.

The same reasoning applies to Q_k and z_k . But then

$$\sum_{m \in \overline{M}} y_m = \mathcal{I}(x) \quad \text{and} \quad \sum_{k \in \overline{K}} z_k = \mathcal{J}(x)$$

and by assumption it follows that

$$\sum_{m \in \overline{M}} y_m \leq \sum_{k \in \overline{K}} z_k.$$

Thus φ is valid.

4.5 Quantifier elimination and simplification

Case 2: $\mathcal{I} \leq \mathcal{J}$ does *not* hold. By assumption there exists a tuple of values $x^* = (x_1^*, \dots, x_M^*)$ such that

$$\mathcal{I}(x^*) > \mathcal{J}(x^*).$$

We have to show that φ is unsatisfiable. Therefore, consider the tuple x^* .

Case 2.1: $P_m(x^*) = \text{true}$. Then y_m has to be equal to u_m since otherwise φ is already false.

Case 2.2: $P_m(x^*) = \text{false}$. Then y_m has to be equal to 0 since otherwise φ is already false.

The same reasoning applies to Q_k and z_k . But then

$$\sum_{m \in \overline{M}} y_m = \mathcal{I}(x^*) \quad \text{and} \quad \sum_{k \in \overline{K}} z_k = \mathcal{J}(x^*)$$

and by assumption it follows that

$$\sum_{m \in \overline{M}} y_m > \sum_{k \in \overline{K}} z_k.$$

Thus there is now way to satisfy φ .

□

The improvement which comes with Theorem 4.1 is that the size of the FO-formula is significantly reduced. Again, the resulting formula is a conjunction of clauses but the number of clauses is just linear in M and K (the number of summands)! From a theoretical point of view, this need not be an improvement, because, as mentioned earlier, the space and time complexity for quantifier elimination is dominated by the number of quantifiers [3, 5]. Nevertheless in practise, we do not artificially construct worst-case formulas. Thus, it might turn out that our new encoding of the verification conditions is shorter (and therefore can be computed more efficiently) and at the same time does not impair REDLOG's performance.

4.5 Quantifier elimination and simplification

A major difference between the methods in Chapter 3 and our tool is that the tool does not transform the universally quantified first-order formula to an existential one, cf. item 1, page 23.

This saves the tool a lot of effort, namely computing the conjunctive normal form which causes an exponential blow up in the size of the formula. Additionally the Motzkin Transposition itself can increase the formula size. The increase in the number of quantified variables and formula length can have negative impacts on REDLOG's performance. Let us look more closely at an example from the previous chapter. There we computed the following verification condition VC :

$$VC \equiv \forall n \in \mathbb{R} : (n \neq 0 \vee \alpha \cdot n + \beta \leq 0) \wedge (-\alpha \cdot n - \beta < 0 \vee -n + M - 1 < 0 \vee \alpha \cdot n + \alpha + \beta \leq 0)$$

Elimination of quantifiers directly from VC using REDLOG yields:

$$\beta \leq 0 \wedge (\alpha^2 \cdot M - \alpha^2 + \alpha \cdot \beta < 0 \vee \alpha \leq 0) \wedge (\alpha^2 \cdot M + \alpha \cdot \beta \leq 0 \vee \alpha \leq 0)$$

4 Implementation

In the example we followed the original approach by Colón et al. [4] and transformed VC into the equivalent formula $\varphi \wedge \psi$ where

$$\begin{aligned} \varphi := \exists \lambda_0, \dots, \lambda_3 : & \quad (\lambda_0 \geq 0 \wedge \dots \wedge \lambda_3 \geq 0 \wedge 0 = \lambda_1 - \lambda_2 - \alpha \lambda_3 \\ & \quad \wedge 1 = -\beta \lambda_3 - \lambda_0) \\ & \quad \vee (\lambda_0 \geq 0 \wedge \dots \wedge \lambda_3 \geq 0 \wedge \lambda_3 \neq 0 \wedge 0 = \lambda_1 - \lambda_2 - \alpha \lambda_3 \\ & \quad \wedge 0 = -\beta \lambda_3 - \lambda_0) \\ \psi := \exists \lambda_0, \dots, \lambda_3 : & \quad (\lambda_0 \geq 0 \wedge \dots \wedge \lambda_3 \geq 0 \wedge 0 = \alpha \lambda_1 + \lambda_2 - \alpha \lambda_3 \\ & \quad \wedge 1 = (\beta \lambda_1 + (-M + 1)\lambda_2 + (-\alpha - \beta)\lambda_3) - \lambda_0) \\ & \quad \vee (\lambda_0 \geq 0 \wedge \dots \wedge \lambda_3 \geq 0 \wedge \lambda_3 \neq 0 \wedge 0 = \alpha \lambda_1 + \lambda_2 - \alpha \lambda_3 \\ & \quad \wedge 0 = (\beta \lambda_1 + (-M + 1)\lambda_2 + (-\alpha - \beta)\lambda_3) - \lambda_0) \end{aligned}$$

Applying REDLOG's quantifier elimination methods to this formula gives:

$$\begin{aligned} & (\beta \leq 0 \vee \alpha \cdot \beta < 0 \wedge \alpha > 0) \wedge (\alpha \cdot M + \beta \leq 0 \vee \alpha < 0) \\ \vee & \quad \alpha^2 \cdot M^2 - \alpha^2 \cdot M + 2 \cdot \alpha \cdot \beta \cdot M - \alpha \cdot \beta + \beta^2 > 0 \wedge \alpha = 0 \\ \vee & \quad \alpha^2 \cdot M - \alpha^2 + \alpha \cdot \beta \leq 0 \wedge \alpha \cdot M - \alpha + \beta > 0) \end{aligned}$$

VC is equivalent to $\varphi \wedge \psi$ and therefore the results of the quantifier elimination procedure are equivalent as well. However, in practise there is a difference in the size of the representation and its readability. Unfortunately, neither of the results is minimal in size but we can say that the first result is more readable.

So far, we considered just the length of the formulas before and after the Motzkin Transposition. But they also differ in their structure. The first one is universally quantified over all program variables, the latter is existentially quantified over auxiliary variables. The number of these lambdas depends on the number of inequalities to which the transposition is applied. One might wonder if quantifier elimination over existential formulas is so much faster that the increase in size due to Motzkin's Transposition is compensated. But this is not the case since the complexity of quantifier elimination for universally quantified formulas is the same as for existentially quantified ones. That is because an algorithm for elimination of existential quantifiers can be directly applied to a universally quantified formula (and vice versa) using the equivalence:

$$\forall x_1, \dots, x_n : \psi \equiv \neg \exists x_1, \dots, x_n : \neg \psi$$

The example above leads us to a key issue of the implementation of our invariant generation method. Even though we can obtain some equivalent quantifier-free formula it is not minimal. For example:

$$\beta \leq 0 \wedge (\alpha^2 \cdot M - \alpha^2 + \alpha \cdot \beta < 0 \vee \alpha \leq 0) \wedge (\alpha^2 \cdot M + \alpha \cdot \beta \leq 0 \vee \alpha \leq 0)$$

is equivalent to (assuming $M > 0$)

$$\beta \leq 0 \wedge \alpha \leq -\frac{\beta}{M}.$$

4.5 Quantifier elimination and simplification

We tried to use REDLOG’s simplification methods *rlsimpl*, *rlgsc* and *rlgsd*. However, none of those is able to reduce the size of the formula (in fact *rlgsd* would increase the size of the formula in this particular case).

It is not a big surprise that simplifying (quantifier-free) first-order formulas is not easily done. The problem of minimizing boolean formulas is already NP-hard. When dealing with “few” boolean variables methods like the Karnaugh map or the Quine-McCluskey algorithm can be applied. Currently, we do not know whether there are algorithms that can minimize an FO-formula or if at least heuristics exist that perform better than those offered by REDLOG.

In the tutorial we have already shown one possibility to mitigate the problem with cluttered formulas. Our tool offers an interface which can be used to access REDLOG’s *rlgsn* command and simplify formulas modulo assumptions (command ‘a’, described in the tutorial, page 35). This enables the user to insert assumptions and see how the constraints change. For example, we might be interested to instantiate the template with a positive α . Simplifying

$$\beta \leq 0 \wedge (\alpha^2 \cdot M - \alpha^2 + \alpha \cdot \beta < 0 \vee \alpha \leq 0) \wedge (\alpha^2 \cdot M + \alpha \cdot \beta \leq 0 \vee \alpha \leq 0)$$

modulo $\alpha > 0$ results in

$$\alpha^2 \cdot M - \alpha^2 + \alpha \cdot \beta < 0 \wedge \alpha^2 \cdot M + \alpha \cdot \beta \leq 0 \wedge \beta \leq 0.$$

And because we already assumed a positive α we can divide by α , which yields

$$\alpha \cdot M - \alpha + \beta < 0 \wedge \alpha \cdot M + \beta \leq 0 \wedge \beta \leq 0.$$

This formula is already simpler and further assumptions can be made if necessary. Currently it is up to the user to ensure that his assumptions are consistent. Contradictory assumptions may lead to a wrong simplification result. We might implement a satisfiability check to prevent this.

Assumptions also allow to provide the simplifier with additional knowledge. For example, the user might know that M is always a positive constant or, if a probability parameter p occurs, the user can add the fact that p is between zero and one which might lead to a simplification. In fact, such knowledge could be encoded into the verification conditions by the tool itself. However we need to introduce some form of “declarations” in our pGCL files so that the tool is aware of the additional facts.

So far, we discussed the translation of verification conditions to first-order formulas and whether or not these formulas should be preprocessed with Motzkin’s Transposition before quantifier elimination. However, we did not pinpoint the reason why only linear invariants are considered. The reason for this is, that the Motzkin Transposition requires linear inequalities as input. With the circumvention of the Motzkin Transposition we can give up the restriction to linear templates because the quantifier elimination tool REDLOG accepts formulas with polynomial inequalities as well. Thus in principle, our approach can be used to reason about polynomial templates. We were not able to implement the necessary data structures in our tool yet because polynomial invariants were originally beyond the scope of this project. In future we will extend the tool as necessary and try to generate polynomial invariants, too. We are particularly interested to see how our the straight-forward extension performs in comparison to more elaborated techniques, cf. [16].

5 Experiments

5.1 Binomial Update

Throughout the thesis we extensively discussed the “binomial update” program. In Chapter 2 it served as the motivating example for probabilistic invariants. There, we explained how an invariant can be used to prove that the expected value of x is indeed $M \cdot p$ according to the expected average of the binomial distribution. In that example the invariant was given and we paid a greater attention to its use within a proof. But how do we actually discover this invariant? To answer this question let us have a final look at the program.

Listing 5.1: “Binomial update” program assigns x some value in $\{0, \dots, M\}$

```
x := 0;
n := 0;
while (n - M + 1 <= 0) {
    (x := x + 1 [p] skip);
    n := n + 1;
}
```

When searching for invariants it might be a bad idea to start with a very general template because the constraints returned by the tool will depend on many parameters and therefore can be quite complex. Just to illustrate this issue we enter the following template:

$$\mathcal{T} = [a + x \geq 0 \wedge x - b \leq 0 \wedge c + n \geq 0 \wedge n - d \leq 0] \cdot (e \cdot x + f \cdot n + g)$$

The template facilitates the search of upper and lower bounds for the variables x and n and an apt weight which can depend on the variables and some constant g . Within two minutes the tool responds with a constraint that is 8690 characters long (slightly more than two pages). Clearly, we do not want to work with that.

Instead we iteratively approach our goal. First, we start with simple, qualitative invariants which might help us to limit a variable’s range. We will use the fact that if qualitative \mathcal{I} and \mathcal{J} are invariant, then so is $\mathcal{I} \wedge \mathcal{J}$. In this way we increasingly strengthen our invariant. After the discovery of a qualitative invariant, we can proceed and reason about its weight, thereby making it lesser¹. McIver and Morgan call this approach “modular reasoning” [14].

To start with, we look at the program text and imagine what information we need in a proof about the expected value of x . Certainly x plays a role and we can use templates like

$$\mathcal{T} = [x + a \geq 0] \cdot (1) \quad \text{or} \quad \mathcal{T} = [a \cdot x \geq 0] \cdot (1)$$

¹Remember, we mentioned in Chapter 2 that “weak” in the qualitative setting translates to “great” in the quantitative setting. Analogously, “strong” corresponds to “small”.

5 Experiments

to find out that x is invariantly non-negative. The same can be done for n . Even better, we discover that x is always at most n using the template

$$\mathcal{T} = [a \cdot x + b \cdot n \geq 0] \cdot (1).$$

Finally, an upper bound for n is given by the guard. The highest possible value at the beginning of an iteration is $M - 1$. We can prove that $n \leq M$ is invariant using a template without parameters. This means we enter the hypothetical invariant directly into our tool and since there are no parameters the resulting constraints can only be *true* or *false*. In this case the result is, of course, *true*.

So far we have a standard invariant that establishes a relation between variables and imposes lower and upper bounds on them. However, if we want to prove something about the expected value of x we have to include it into the weight. Let us consider the template

$$\mathcal{T} = [x \geq 0 \wedge x - n \leq 0 \wedge n - M \leq 0] \cdot (a \cdot x + b \cdot n + c).$$

The constraint for the consecution condition provides a hint for further refinement of this template. The constraint is

$$a \cdot p + b \geq 0 \vee M - 1 < 0.$$

Remember that M is the number of loop iterations. Thus M is certainly greater than one. Therefore the first inequality must hold. Let us assume $b = -a \cdot p$. This simplifies the template to

$$\mathcal{T} = [x \geq 0 \wedge x - n \leq 0 \wedge n - M \leq 0] \cdot (a \cdot x - a \cdot p \cdot n + c).$$

If we re-run the tool with this simplified template we see that the consecution condition is satisfied and we now need to make sure that $(a \cdot x - a \cdot p \cdot n + c)$ is between zero and one. In order to find useful a and c we consider the values of x and n before the first iteration of the while-loop and after termination. In the beginning all variables are set to zero. So we know that

$$0 \leq c \leq 1$$

must hold initially. Upon termination we do not exactly know the value of x but we know that $n = M$ holds. Hence we want that

$$0 \leq a \cdot x - a \cdot p \cdot M + c \leq 1$$

holds. Remember that the proof objective is the expected value of x . It is desirable that $-a \cdot p \cdot M + c$ cancel. Then $a \cdot x$ remains which should be bounded between zero and one. This reasoning suggests to choose

$$a = \frac{1}{M} \quad \text{and consequently} \quad c = p.$$

Thereby we scale x and n down to $[0, \dots, 1]$ and make sure that by the end of the program $\frac{1}{M} \cdot x$ remains while the rest of the term cancels. Verifying this choice with our tool gives us the desired response *true*. The construction of the proof with the invariant

$$\mathcal{I} = [x \geq 0 \wedge x - n \leq 0 \wedge n - M \leq 0] \cdot \left(\frac{1}{M} \cdot x - \frac{p}{M} \cdot n + p \right)$$

has already been explained in Chapter 2.

Note that the constraints do not dictate exactly the invariants that we have found. We could as well show that e.g. $x \geq -5.5$ is invariant. Unfortunately our tool does not know what we want to prove about the program and which invariants are useful for this. Therefore it is up to the user to decide how to choose the parameters. For example, the program initially assigns zero to x and afterwards x is possibly increased but never decreased so $x \geq -5.5$ is too weak and one would choose rather $x \geq 0$. Another rule of thumb is to choose parameter values such that an inequality between two parameters is satisfied with equality.

5.2 Biased Coin

The next example is a program that simulates a “biased” coin flip by repeatedly flipping a fair coin. In the source code, the variable b is assigned boolean constants for better readability. But in fact, b is a real valued variable and the constants are replaced by their numerical counterparts by the parser.

Listing 5.2: “Biased Coin” program assigns x

```
//init :
x:= p;
b:= true;
//loop while b is true:
while (b - 1>= 0) {
  (b := false [0.5] b := true);
  //if b is true
  if (b - 1 >= 0) {
    x:= 2*x;
    if (x - 1 >= 0) {
      x:= x-1;
    } else {
      skip;
    }
  }
  else if (x - 0.5 >= 0) {
    x:= 1;
  }
  else {
    x:= 0;
  }
}
```

Our aim is to prove that the probability of the event $x = 1$ is p . Therefore we need an invariant that will ensure $x = 1$ after the loop and which has a worth of p at initialisation. It is plain to see that the loop can only terminate if $b < 1$ (i.e. $b = false$) but in that case the variable x is set to

5 Experiments

either zero or one in the loop body. Testing our hypothesis

$$[\neg(b - 1 \geq 0) \Rightarrow (x = 0 \vee x = 1)] \cdot (1)$$

with the tool we learn that our observation is correct. We see that this annotation simplifies to

$$[x = 0] \cdot (1) + [x = 1] \cdot (1)$$

after termination of the loop because then surely the guard of the while-loop does not hold any more. Now we need to express the weight in terms of x in order to make sure that $[x = 0] \cdot (1) + [x = 1] \cdot (1)$ implies $[x = 1] \cdot (1)$ and that $wlp(x := p; b := true, \mathcal{I}) = p$. Hence we choose

$$\mathcal{I} = [\neg(b - 1 \geq 0) \Rightarrow (x = 0 \vee x = 1)] \cdot (x)$$

and see whether our tool will return *true* for all constraints. Apparently this is not the case. This is because we only know that if the guard is false, then x is either zero or one but otherwise we do not anything about x . So the consecution condition is satisfied but the boundedness conditions are both violated.

Let us start anew and try to discover an invariant which limits the range of x . Just like in the previous section we use the simple template

$$\mathcal{T} = [x - a \leq 0] \cdot (1)$$

to find out that x is invariantly between zero and one. This qualitative invariant we can insert into \mathcal{I} and obtain

$$\mathcal{I} = [x \geq 0 \wedge x - 1 \leq 0 \wedge \neg(b - 1 \geq 0) \Rightarrow (x = 0 \vee x = 1)] \cdot (x).$$

The modified predicate ensures that x is bounded as required by the verification conditions. Indeed our tool reports that all conditions are met and we can successfully carry out the proof that

$$Pr(x = 1) \geq p$$

as shown in Listing 5.3.

Listing 5.3: Correctness proof for the “Biased Coin” program

```

⟨[true] · (p)⟩
x := p;
⟨[0 ≤ x ∧ x ≤ 1] · (x)⟩
⟨[0 ≤ x ∧ x ≤ 1 ∧ false ⇒ (x = 1 ∨ x = 0)] · (x)⟩
b := true;
⟨[0 ≤ x ∧ x ≤ 1 ∧ (b < 1) ⇒ (x = 1 ∨ x = 0)] · (x)⟩
while (b - 1 >= 0) {
    (b := false [0.5] b := true);
    //if b is true
    if (b - 1 >= 0) {
        x := 2*x;
        if (x - 1 >= 0) {
            x := x-1;
        } else {
            skip;
        }
    }
    else if (x - 0.5 >= 0) {
        x := 1;
    }
    else {
        x := 0;
    }
}
⟨[0 ≤ x ∧ x ≤ 1 ∧ (b < 1) ⇒ (x = 1 ∨ x = 0) ∧ b < 1] · (x)⟩
⟨[x = 1] · (x) + [x = 0] · (x)⟩
⟨[x = 1] · (1)⟩

```


6 Conclusion

6.1 Summary

In this thesis we presented the probabilistic programming language pGCL. A key feature of this language is the probabilistic choice between alternatives which is made according to some discrete distribution. The semantics of pGCL was given in terms of the expectation transformer wlp. We explained how we can reason about a program's properties by means of expectations. The properties, that were considered, established a least probability for some event (expressed by a predicate) or the least expected value of a program variable.

We highlighted the importance of invariant annotations for the proof of such properties. These invariants are difficult to synthesise which motivates the development of algorithms that automate invariant generation as much as possible. We treated the constraint-based approach for invariant generation for standard programs due to Colón et al. [4]. This lays the basis for an extension to invariant generation for probabilistic programs by Katoen et al. [12]. Within the scope of our diploma project we implemented the first tool for invariant generation for probabilistic programs which is based on the approach in [12]. The key points of the implementation were discussed.

We explained how verification conditions are generated from the program text and a template. These verification conditions determine whether an instantiation of a template is invariant or not. Our goal was to find the set of all invariant instances of the template.

For this purpose, we presented a translation from verification conditions to (universally quantified) first-order formulas over the real numbers. This translation included the computation of a so-called disjoint normal form for expectations and subsequently, a comparison of expectations in a pairwise fashion. We discussed the complexity of these steps and proposed an enhanced encoding of the verification condition which can be computed exponentially faster.

Finally, we extracted explicit constraints on the template's parameters by applying quantifier elimination to FO-formulas, which were mentioned above. For the elimination procedure we rely on a tool called REDLOG. We discussed practical issues of quantifier elimination including performance and the non-minimality of the result. Consequently, we identified the need in minimisation techniques for FO-formulas over the real domain.

In order to alleviate the problems with unreadable constraints we suggested to make use of REDLOG's formula simplification method. It accepts assumptions about the formula which may help to reduce the size of the constraints significantly.

Given the quantifier-free constraints for the template's parameters it is up to the user to choose values. In general there are infinitely many possible choices. We recommended to choose "simple" parameters whenever possible. In particular we chose parameters in such a way that inequalities between them were satisfied with equality.

6.2 Outlook

Many questions are open yet and more work has to be done to understand the benefits and limits of invariant generation. Among other things the future work includes the following points.

- Currently it is up to the user to formulate a template which serves as input to our tool and in the end the user has to choose parameter values. More case studies need to be considered in order to derive better heuristics which assist the user in his choices.
- The restriction to linear invariants apparently hinders us from the analysis of more interesting programs. The extension of our methods to polynomial invariants is one of the main goals in the future.
- The previous point opens another application of invariant generation. Recent publications e.g. [8] suggest that one can establish an upper bound for the time complexity using invariants. Therefore a counter, say i , is introduced that is increased with every iteration of the loop. If an algorithm has a running time complexity of n^3 where n determines the input size, then we hope to find invariants of the form $i \leq n^3$ which prove this time bound.
- We have seen that the generated constraints often have an overly complex representation. This negatively affects the usability of the tool. Therefore we need to find new methods for simplifying first-order quantifier-free formulas which make the results more readable.
- For standard programs there is a straight-forward correspondence between predicate transformer semantics and operational semantics: In each execution step the program moves from one set of states to another. The program annotations characterise these sets. This is different for probabilistic programs. In the operational model, a transition leads from one distribution over program states to another but our annotations are expectations. It would be interesting to find a correspondence between these two.
- A long term-goal is to give up the restriction to programs with one loop only. However, nested loops impose a big challenge already in the non-probabilistic setting.

We hope that our work is a valuable contribution to the area of logical inference that dealt mostly with non-probabilistic programs so far. Our tool demonstrates the use of constraint-based invariant generation methods for probabilistic programs. Still many questions are unresolved in this area and we hope that some of them might be tackled with suggestions made in this thesis.

Bibliography

- [1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
- [2] Adi Ben-Israel. Motzkin’s transposition theorem, and the related theorems of farkas, gordan and stiemke.
- [3] Christopher W. Brown and James H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *Proceedings of the 2007 international symposium on Symbolic and algebraic computation, ISSAC ’07*, pages 54–60, New York, NY, USA, 2007. ACM.
- [4] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification*, pages 420–432. Springer Verlag, 2003.
- [5] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1-2):29 – 35, 1988.
- [6] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [7] Robert W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceedings of Symposium on Applied Mathematics*, volume 19, pages 19–32. A.M.S., 1967.
- [8] Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Proceedings of the 20th international conference on Computer Aided Verification, CAV ’08*, pages 370–384, Berlin, Heidelberg, 2008. Springer-Verlag.
- [9] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *In PLDI*, 2008.
- [10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [11] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *In IMACS Intl. Conf. on Applications of Computer Algebra*, 2004.
- [12] Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll Morgan. Linear-invariant generation for probabilistic programs. In *Static Analysis Symposium (SAS). LNCS*. Springer-Verlag, 2010.

Bibliography

- [13] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
- [14] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag, 2004.
- [15] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
- [16] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases, 2004.