POPACheck User's Guide

Michele Chiari, Francesco Pontiggia

28.04.2025

Abstract

POPACheck is an extension of the POMC model checker towards probabilistic recursive programs, modelled as probabilistic Operator Precedence Automata (pOPA). It supports model checking of temporal logic formalism Linear Temporal Logic (LTL), and additionally a fragment of Precedence-Oriented Temporal Logic (POTL), a logic for context-free properties such as pre/post-conditions. The fragment is called POTLfχ. Given a pOPA and a formula in either LTL or POTLfχ, POPACheck can solve both qualitative and quantitative model check queries. Additionally, it can compute (approximately) the termination probability of the pOPA. Computing termination probabilities are a central problem in probabilistic pushdown model checking, preliminary to any kind of analysis, and amounts at computing the Least Fixed Point solution of a Positive Polynomial System (PPS). This document is a reference guide to its input and output formats, and also describes at a high level its architecture and source code.

1 Introduction

pOPA [6] are a class of probabilistic pushdown automata based on the family of Operator Precedence Languages (OPLs) [5], a subclass of deterministic context-free languages. While they do not read an input, which would make any model checking problem undecidable, the (infinite-length) traces of state labels collected in the paths of a given pOPA constitute an OPL.

Precedence-Oriented Temporal Logic (POTL) [2, 3] is an established temporal logic formalism for expressing many fundamental properties on programs with recursive procedures, such as partial and total correctness, and Hoare-style pre/post conditions. Given a POTL formula, the POMC [1] tool can translate it into an Operator Precedence Büchi Automaton (OPBA), the class of (nondeterministic, non probabilistic) Pushdown Automata recognizing OPLs.

POPACheck exploits the fact that OPLs are closed by Boolean operations (e.g., intersection, complementation . . .). Roughly speaking, POPACheck:

- takes as input a formula and a program in a custom Domain-Specific Language called MiniProb.
- translates the program into a (explicitly represented) pOPA.
- uses POMC modules to translate the formula into an OPBA.
- model-checks the pOPA against the OPBA via automata-based model checking, i.e., via a cross-product.

Involved technicalities arise due to the facts that:

- pOPAs are equipped with an unbounded stack, hence they are infinite-state models.
- we do not perform determinization of the specification OPBA, as canonical in probabilistic model checking.

We'll skip their treatment here, and refer to [6]. We just mention that pOPA infinite runs can be represented (or 'summarized') by a finite-state Markov Chain called *support chain.* The support chain of a pOPA can be computed by solving (nonlinear) Positive Polynomial Systems (PPSs) of equations for their Least Fixed Point. Solutions to these systems are called *termination probabilities.* Due to their nonlinearity, they cannot be computed exactly, i.e. solutions may be irrational, and not even expressible by radicals [4]. Our tool deals with this issue by computing sound lower and upper rational bounds to termination probabilities. While it computes lower bound always via numerical methods, it offers two approaches for upper bounds: one is purely numerical, and it is called Optimistic Value Iteration (OVI); the other one relies on the SMT solver Z3 [8]. OVI has been introduced originally by Winkler and Katoen [9] in the tool Pray.

Similar equation systems arise in quantitative model checking. Likewise, POPACheck computes lower and upper bounds to the Least Fixed Point solutions of these systems - in this case, always via OVI. This means that for quantitative model check queries POPACheck will return a lower and an upper bound to the satisfaction probability.

We show how to use POPACheck in Section 2. If you wish to examine the input formulas and OPA for the experiments more carefully, or to write your own, we describe the format of POPACheck input files in Sections 3, 4, and 5. Finally, Section 6 contains a high-level description of the source code.

2 Quick-Start Guide

POPACheck, just like POMC, has been developed in the Haskell programming language, and packaged with the Haskell Tool Stack[1]. POPACheck has a few dependencies:

- Z3 for solving (nonlinear) equations systems.
- BLAS/LAPACK, GSL and GLPK for approximating solutions to PPSs via iterative fixpoint numerical methods (Newton's method), which are used in the Haskell hmatrix package.

On a UNIX-like system, they can be installed by running:

```
sudo apt install libz3-dev libgsl0-dev liblapack-dev libatlas-base-dev
```

This link contains some information on how to install hmatrix dependencies on other systems. Haskell bindings to Z3 are hosted on the Github repository haskell-z3.

The Z3 library requires special care, as it might conflict with the underlying OS. The current version of the tool (14.04.2025) has been fully tested with Z3 version 4.8.12 and Ubuntu 24. Note that version 4.8.12 is the one installed by default by the

[1] https://www.haskellstack.org/

APT package index on Ubuntu 24. On the other side, we experienced some issues on Ubuntu 22, where Z3 sometimes returns error `Z3: invalid argument`. Please report to the POPACheck team in case you experience some issue.

After having resolved the dependencies, POPACheck can be built from sources by typing the following commands in a shell:

```
$ cd ~/path/to/POPACheck-sources
$ stack setup
$ stack build
```

Then, POPACheck can be executed on an input file `file.pomc` as follows:

```
$ stack exec popacheck -- file.pomc {args}
```

POPACheck stack commands take a few arguments:

- `--noovi` [default: False]. When set, POPACheck uses Z3 instead of OVI for computing upper bounds to termination probabilities. As the experimental evaluation of [7] suggests, this leads almost always to timeouts, and will probably will be removed in then near future.
- `--newton` [default: False]. When set, POPACheck uses Newton's iterative method for computing lower bounds to termination probabilities, and to fractions in quantitative model checking. The default method is Value Iteration with Gauss-Seidel updates. We refer to PreMo publications [11, 10] for a detailed description of these two numerical methods. In our experiments, the Newton method tends to be slightly faster than Gauss-Seidel Value Iteration, hence we encourage users to set this flag.
- `-verbose` [default: 0]. Logging level. 0 = no logging, 1 = show info, 2 = debug mode.

Directory `eval` contains the Python script `probbench.py`, which may be useful to evaluate POPACheck input files, as it also prints a summary of the resources used by POPACheck. It must be executed with a subdirectory of `~/path/to/POPACheck-sources` as its working directory, and either `--print` to print the results in the shell, or `--raw_csv file_name` for saving results in .csv format in `file_name`.

```
$ cd ~/path/to/POPACheck-sources/eval
$ ./probbench.py prob/established/qualitative/schelling --print
```

evaluates all `*.pomc` files in directory
`~/path/to/POPACheck-sources/eval/prob/established/qualitative/schelling`.

3 Input Language

POPACheck analyzes programs written in MiniProb, a simple probabilistic programming language (Fig. 1). MiniProb programs are written in files with extension `.pomc`. MiniProb supports (un)signed integer variables of arbitrary width (u8 is an 8-bit unsigned type) and fixed-size arrays. Functions take parameters by value or value-result (with &). Actual parameters can only be variable identifiers for value-result parameters, and any expression if passed by value. Expressions consist of variables, array

$$\begin{aligned}
\mathit{prog} &:= [\mathit{decl}; \ldots]\ \mathit{func}\ [\mathit{func} \ldots] \\
\mathit{decl} &:= \mathit{type}\ \mathit{identifier}\ [, \mathit{identifier} \ldots] \\
\mathit{type} &:= \texttt{bool} \mid \texttt{u}\mathit{int} \mid \texttt{s}\mathit{int} \mid \texttt{u}\mathit{int}[\mathit{int}] \mid \texttt{s}\mathit{int}[\mathit{int}] \\
\mathit{func} &:= f\,\texttt{(}\mathit{type}\ [\&]x_1\ [, \mathit{type}\ [\&]x_2 \ldots]\texttt{)} \\
&\quad\ \{[\mathit{decl}; \ldots]\ \mathit{block}\} \\
\mathit{stmt} &:= \mathit{lval} = e \\
&\quad \mid \mathit{lval} = \texttt{Distribution(}\ldots\texttt{)} \\
&\quad \mid \mathit{lval} = e_1\{e_2 : e_3\}[e_4\{e_5 : e_6\} \ldots]e_n \\
&\quad \mid [\texttt{query}]\ f\,\texttt{(}e_1 \mid \mathit{lval}_1\ [, e_2 \mid \mathit{lval}_2 \ldots]\texttt{)} \\
&\quad \mid \texttt{if (}e\texttt{)}\ \{\mathit{block}\}\ \texttt{else}\ \{\mathit{block}\} \\
&\quad \mid \texttt{while (}e\texttt{)}\ \{\mathit{block}\} \\
&\quad \mid \texttt{observe (}e\texttt{)} \\
\mathit{block} &:= \mathit{stmt}; [\mathit{stmt} \ldots ;] \\
\mathit{lval} &:= \mathit{identifier} \mid \mathit{identifier}[e]
\end{aligned}$$

Figure 1: MiniProb syntax.

indexing, integer constants, and the usual arithmetic and Boolean operators, including comparisons. Boolean operators handle integers (0 means false, everything else true). Programs may sample from `Bernoulli(`e_1, e_2`)`, which returns 1 with probability $p = e_1/e_2$, and 0 with probability $1 - p$, or from `Uniform(`e_1, e_2`)`, which samples uniformly among integers from e_1 to $e_2 - 1$. Random assignments of the form $x = e_1\{e_2/e_3\}e_4$ mean that x is assigned the value of e_1 with probability e_2/e_3, and e_4 with probability $1 - e_2/e_3$. Finally, functions can `query` the distribution on value-result parameters of another function, and condition on a Boolean expression with `observe`.

4 Model Check Queries

A model check query must be put at the beginning of a `.pomc` file, before the program. It follows the syntax:

`probabilistic query:` q`;`

where q is one of the queries of Table 1. When a formula is needed, it has to be placed on a new line with syntax:

`formula:` f`;`

where f follows the syntax of Table 2. For some technical reasons explained in [6], POPACheck does not support the whole POTL logic. In a single line, this is due to the fact that the model check algorithm avoids determinization of the specification automata. Though, POPACheck supports full LTL. To get an idea of queries, consider inspecting different experiments in `eval/prob/established/`, where the same programs are verified against different queries.

query	explanation	Formula?
`approximate`	what is the termination probability of the program?	No
`qualitative`	Does the program satisfy f almost surely?	Yes
`quantitative`	What is the probability that the program satisfies f?	Yes

Table 1: Available queries.

Plain reachability queries are supported through the LTL `Eventually` operator at the moment. We plan to optimize it in future work, as they could be encoded as a termination query.

Additionally, POPACheck supports also the query `unfold&export`, which constructs a Markov Chain in explicit Storm format for a given program by unfolding the program's stack. Argument `maxDepth` to the stack command specifies the maximum stack depth to unfold [default: `100`]. When `maxDepth` is reached, recursion is not unfolded anymore, and a simple self-loop is added. Note that `.pomc` programs may have infinite recursion. We use this feature for testing purposes, and do not advertise users to try it out.

4.1 An example: Pre/Post Conditions.

With POTLfχ it is possible to express and check automatically pre/post conditions on recursive programs. Consider the following Hoare triple:

$$\varphi \; \{P\} \; \theta$$

where P is a potentially recursive program. We want to check whether, if φ holds at a call of P (*pre-condition*), then θ holds at the corresponding return (*post-condition*). This requirement cannot be expressed with LTL as it is a context-free requirement, but it can be expressed with POTLfχ via:

```
                probabilistic query:  qualitative;
formula:  G ((call And P And φ) Implies (XNu (ret And P And θ)))
```

which means that *always* (`G`), *if* the program is in a state calling P and where φ holds, *then* (`Implies`) this call has a matching return (`XNu (ret And P)`) where θ holds. Note that this formula does not hold almost surely if P has non zero probability of non terminating - nonterminating runs do not have a matching return.

Group	POTL (or LTL) Operator	POPACheck Operator	Notation	Associativity
Unary	$\neg$	`~`, `Not`	Prefix	–
	$\bigcirc^d$	`PNd`	Prefix	–
	$\bigcirc^u$	`PNu`	Prefix	–
	$\ominus^d$	`PBd`	Prefix	–
	$\ominus^u$	`PBu`	Prefix	–
	χ_F^d	`XNd`	Prefix	–
	χ_F^u	`XNu`	Prefix	–
	~~χ_P^d~~	~~`XBd`~~	~~Prefix~~	–
	~~χ_P^u~~	~~`XBu`~~	~~Prefix~~	–
	~~$\ominus_H^d$~~	~~`HNd`~~	~~Prefix~~	–
	~~$\ominus_H^u$~~	~~`HNu`~~	~~Prefix~~	–
	~~$\ominus_H^d$~~	~~`HBd`~~	~~Prefix~~	–
	~~$\ominus_H^u$~~	~~`HBu`~~	~~Prefix~~	–
	$\Diamond$ (LTL)	`F`, `Eventually`	Prefix	–
	$\bigcirc$ (LTL)	`N`	Prefix	–
	$\mathcal{U}$ (LTL)	`U`	Infix	Right
	$\Box$ (LTL)	`G`, `Always`	Prefix	–
POTL Binary	$\mathcal{U}_\chi^d$	`Ud`	Infix	Right
	$\mathcal{U}_\chi^u$	`Uu`	Infix	Right
	~~$\mathcal{S}_\chi^d$~~	~~`Sd`~~	~~Infix~~	~~Right~~
	~~$\mathcal{S}_\chi^u$~~	~~`Su`~~	~~Infix~~	~~Right~~
	~~$\mathcal{U}_H^d$~~	~~`HUd`~~	~~Infix~~	~~Right~~
	~~$\mathcal{U}_H^u$~~	~~`HUu`~~	~~Infix~~	~~Right~~
	~~$\mathcal{S}_H^d$~~	~~`HSd`~~	~~Infix~~	~~Right~~
	~~$\mathcal{S}_H^u$~~	~~`HSu`~~	~~Infix~~	~~Right~~
Prop. Binary	$\wedge$	`And`, `&&`	Infix	Left
	$\vee$	`Or`, `||`	Infix	Left
	$\oplus$	`Xor`	Infix	Left
	$\implies$	`Implies`, `-->`	Infix	Right
	$\iff$	`Iff`, `<-->`	Infix	Right

Table 2: All POTL and LTL operators, in descending order of precedence. Operators listed on the same line are synonyms. Operators in the same group have the same precedence. Note that operators are case sensitive. **Operators not in the fragment POTLfχ supported by POPACheck are crossed out.**

5 Interpreting the output

The output of running a query is quite verbose at the moment. For example:

```
$ stack exec -- popacheck prob/established/quantitative/schelling/Q03.pomc -
-newton
```

prints

```
Quantitative Probabilistic Model Checking
Query: G (((call And alice) And [| (p == [4]4)]) --> (~ (XNu obs)))
Result:  (5064173399 % 5660011320,6145 % 6868)
Floating Point Result:  (0.8947284930518479,0.894729178800233)
Elapsed time: 20.74 s (total), 2.3883e-2 s (upper bounds), 2.8956e-
2 s (PAST certificates), 1.1778e0 s (graph analysis),1.1905e1 s
(upper bounds with OVI for quant MC),7.5453e-4 s (eq system for quant MC).
Input pOPA state count: 311
Support graph size: 682
Equations solved for termination probabilities: 1230
Non-trivial equations solved for termination probabilities: 266
SCC count in the support graph: 1117
Size of the largest SCC in the support graph: 24
Largest number of non trivial equations in an SCC in the Support Graph: 52
Size of graph G: 44
Equations solved for quant mc: 893036
Non-trivial equations solved for quant mc: 68226
SCC count in quant mc weight computation: 336410
Size of the largest SCC in quant mc weight computation: 144
Largest number of non trivial equations in an SCC in quant mc weight computation: 7318
```

Most of lines just print statistics about the experiment. An user may only read

```
Floating Point Result:  (0.8947284930518479,0.894729178800233)
```

which are, respectively, a lower and an upper bound to the probability that the Schelling model satisfies formula Q03. It might be of general interest to inspect the overall number of equations solved (i.e., the size of the PPS), 893036 in this case, or the number of states in the input model, 311.

6 Source Code

POPACheck is open source. The source code of POPACheck is contained in directory `src/Pomc/Prob`. We refer to the documentation of POMC for the other modules. We describe the contents of each file below.

FixPoint.hs contains the data structures to represent sparse PPSs, and vectors of solutions (termination probabilities). Given a PPS, it keeps track of those equations that are not solved, and allows to obtain a lower bound to their Least Fixed Point solution via either the Gauss-Seidel method or the Newton's method.

GGraph.hs contains the implementation of main qualitative and quantitative model checking routines. It also contains some procedures for building the cross-product between the formula's automaton and the support chain of the pOPA (what is called graph G in [6]).

GReach.hs contains functions for exploring edges in graph G that underpin support edges in the summary chain of the pOPA. For qualitative model checking, it offers a simple reachability algorithm for building these edges. For quantitative model checking, it offers a SCC-based algorithm for computing both lower and upper bounds to the fraction associated with each edge via OVI. This amounts at solving PPSs strictly resembling those for termination probabilities.

MiniProb.hs contains the implementation of the MiniProb programming language.

OVI.hs contains our implementation of Optimistic Value Iteration (OVI) for computing upper bounds to the Least Fixed Point solution of PPSs.

ProbEncoding.hs contains routines for generating a Bitvector encoding of formulae satisfied in a summary edge in the cross-product graph.

ProbModelChecking.hs exposes all our probabilistic model checking APIs.

ProbUtils.hs contains various utility functions.

SupportGraph.hs contains a function for building the support graph of an input pOPA, an intermediate formalism for the computation of the support chain.

Z3Termination.hs contains routines for computing termination probabilities of a pOPA, either via OVI or via Z3, and certifying via Z3 that such probabilities are exactly equal to one when needed, according to the semialgorithm of [7].

6.1 Test Suite

The `test` directory contains regression tests based on the HUnit provider of the Tasty[2] framework. They can be run with

```
$ stack test
```

but note that some of them may take a very long time or exhaust your memory. To learn how to execute just some of them, please consult the `README.md` file in the `test` directory.

Acknowledgements

We are thankful to Tobias Winkler and Prof. Joost-Pieter Katoen (RWTH Aachen) for the fruitful discussions and for the advice on implementing OVI.

[2]`https://github.com/UnkindPartition/tasty`

References

[1] Michele Chiari, Dino Mandrioli, Francesco Pontiggia, and Matteo Pradella. A model checker for operator precedence languages. *ACM Trans. Program. Lang. Syst.*, 45(3), 2023. doi:10.1145/3608443.

[2] Michele Chiari, Dino Mandrioli, and Matteo Pradella. Model-checking structured context-free languages. In *CAV '21*, volume 12760 of *LNCS*, page 387–410. Springer, 2021. doi:10.1007/978-3-030-81688-9_18.

[3] Michele Chiari, Dino Mandrioli, and Matteo Pradella. A first-order complete temporal logic for structured context-free languages. *Log. Methods Comput. Sci.*, 18:3, 2022. doi:10.46298/LMCS-18(3:11)2022.

[4] Kousha Etessami and Mihalis Yannakakis. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM*, 56(1):1:1–1:66, 2009. doi:10.1145/1462153.1462154.

[5] D. Mandrioli and M. Pradella. Generalizing input-driven languages: Theoretical and practical benefits. *Computer Science Review*, 27:61–87, 2018. doi:10.1016/j.cosrev.2017.12.001.

[6] Francesco Pontiggia, Ezio Bartocci, and Michele Chiari. Model checking probabilistic operator precedence automata. *CoRR*, 2024. URL: https://doi.org/10.48550/arXiv.2404.03515.

[7] Francesco Pontiggia, Ezio Bartocci, and Michele Chiari. POPACheck: a Model Checker for probabilistic Pushdown Automata, 2025. URL: https://arxiv.org/abs/2502.03956, arXiv:2502.03956.

[8] Microsoft Research. Z3. URL: https://github.com/Z3Prover/z3.

[9] Tobias Winkler and Joost-Pieter Katoen. Certificates for probabilistic pushdown automata via optimistic value iteration. In *TACAS'23*, volume 13994 of *LNCS*, pages 391–409. Springer, 2023. doi:10.1007/978-3-031-30820-8_24.

[10] Dominik Wojtczak. *Recursive probabilistic models : efficient analysis and implementation*. PhD thesis, University of Edinburgh, UK, 2009. URL: https://hdl.handle.net/1842/3217.

[11] Dominik Wojtczak and Kousha Etessami. PReMo: An analyzer for probabilistic recursive models. In *TACAS'07*, volume 4424 of *LNCS*, pages 66–71. Springer, 2007. doi:10.1007/978-3-540-71209-1_7.