POMC User's Guide

Michele Chiari, Francesco Pontiggia

28.04.2025

Abstract

POMC is tool suite that includes two tools: POMC and POPACheck.

POMC is a model checker that models procedural programs as pushdown automata, and checks them against formulas of Precedence-Oriented Temporal Logic (POTL). POPACheck is an extension of POMC towards probabilistic recursive programs, modelled as probabilistic Operator Precedence Automata (pOPA).

It supports model checking of Linear Temporal Logic (LTL), and a fragment of POTL, 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 is 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 their input and output formats, and also describes at a high level its architecture and source code.

1 Introduction

1.1 POTL

Precedence-Oriented Temporal Logic (POTL) [6, 7] 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. POTL is based on the family of Operator Precedence Languages (OPL) [10], a subclass of deterministic context-free languages. POTL is strictly more expressive than LTL and other temporal logics based on subfamilies of context-free languages, such as CaRet [3] and NWTL [2]. In particular, POTL reasons on an algebraic structure equipped with, besides the usual linear order, a binary nesting relation between word positions, which can be one-to-one, one-to-many, or many-to-one. Such a relation is more general than the one found in Nested Words [4], because the latter may only be one-to-one. POTL can be applied to the specification of several kinds of requirements on procedural programs with exceptions.

1.2 POMC

POMC contains two different model checking engines for POTL. The explicit-state engine employs an automata-based model checking procedure for POTL. This procedure consists of building an Operator Precedence Automaton (OPA), the class of pushdown automata that identifies OPL, accepting the language denoted by a given POTL formula. The size of the generated automaton is exponential in the length of the formula,

which is asymptotically comparable with other linear-time temporal logic formalisms such as LTL, CaRet, and NWTL. Given a POTL formula φ and an input OPA modeling some system, POMC builds the OPA equivalent to $\neg\varphi$, computes its intersection with the input OPA, and checks the emptiness of the resulting OPA. Both the OPA construction and the intersection are done on-the-fly. The explicit-state engine has been implemented for the infinite-word case too, using ωOPBA instead of OPA.

POMC also contains a SMT-based model checking engine for POTL formulas. It consists of a bounded SMT encoding of a tree-shaped tableau for POTL [5]. The tableau is complete: is the provided bound is sufficiently large, both truth and falseness of a formula can be proved. For the time being, this engine only supports finite-word model checking.

POMC also supports providing input models in MiniProc, a simple procedural programming language with exceptions. MiniProc programs are automatically translated into equivalent OPA. The SMT-based engine only supports MiniProc programs as inputs.

1.3 POPACheck

pOPA [11] are a class of probabilistic pushdown automata based on OPLs. 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.

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 [11]. 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 [8]. 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 [13]. OVI has been introduced originally by Winkler and Katoen [15] 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 POMC and 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 POMC and POPACheck input files in Sections 3, 4.1, and 4.2. 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 Debian-based GNU/Linux distribution, 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, because some features used by POPACheck are buggy in older versions. The current version of the tool (29.04.2025) has been fully tested with Z3 version 4.8.12 and Ubuntu 24.04/10. Note that version 4.8.12 is the one installed by default by the APT package index on Ubuntu 24.04/10. On the other hand, we experienced some issues on Ubuntu 22.04 with the default version (still 4.8.12), where Z3 sometimes returns error `Z3: invalid argument`. Please report to the POMC team in case you experience some issue.

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

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

This command automatically clones and builds also the bindings from haskell-z3.

2.1   POMC

POMC can be executed on an input file `file.pomc` as follows:

```
$ stack exec -- pomc file.pomc
```

---

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

By default, POMC will perform infinite-word model checking. The optional arguments `--finite` and `--infinite` can be used to control this behavior manually. POMC uses the explicit-state engine by default. To use the SMT engine, use the flag `--smt=k`, where `k` is a positive integer indicating the maximum length of the encoding. For the time being, it can only be used together with `--finite`. So for instance, to check an input file with the SMT-based engine type:

```
$ stack exec -- pomc --finite --smt=200 file.pomc
```

Type `stack exec -- pomc --help` to see all available command-line options.

Directory `eval` contains several POMC input files. Such files contain POTL formulas and OPA to be checked against them. For more details on the format of POMC input files, see Section 3.

Directory `eval` also contains the Python script `mcbench.py`, which may be useful to evaluate POMC input files, as it also prints a summary of the resources used by POMC. It must be executed with a subdirectory of `~/path/to/POMC-sources` as its working directory. If invoked with no arguments, it executes POMC on all input files in the current working directory with the infinite-word semantics and explicit-state engine. E.g.,

```
$ cd ~/path/to/POMC-sources/eval
$ ./mcbench.py opa-cav
```

evaluates all `*.pomc` files in directory `~/path/to/POMC-sources/eval/opa-cav`. The script can also be invoked with POMC files as its arguments, which are then evaluated. E.g.,

```
$ cd ~/path/to/POMC-sources/eval/opa-cav
$ ./mcbench.py 1-generic-small.pomc 2-generic-medium.pomc
```

executes POMC on files `1-generic-small.pomc` and `2-generic-medium.pomc`. `mcbench.py` can be invoked with the following optional flags:

`-s, --smt <#k>` Use the SMT engine with the given value of `k`

`-f, --finite` Only check finite execution traces (infinite-word model checking is the default)

`-i, --iters <#iters>` Number of iterations of the benchmarks to be performed. The final table printed by the script contains the mean time and memory values computed on all iterations. (Default: 1)

`-j, --jobs <#jobs>` Number of benchmarks to be run in parallel. If you provide a value greater than 1, make sure you have enough CPU cores on your machine. (Default: 1)

`-t, --timeout <timeout>` Timeout for benchmarks in seconds

`-M, --max-mem <limit>` Memory limit for benchmark in MiB

`-m, --ms` Output time in milliseconds instead of seconds.

`--csv <file>` Write results in CSV format in the given file

`-v, --verbose <level>` Verbosity level can be 0 (no additional info), 1 (print POMC output, e.g. counterexamples), or 2 (print POMC output and time/memory statistics).

	call	ret	han	exc
call	$\lessdot$	$\doteq$	$\lessdot$	$\gtrdot$
ret	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
han	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$
exc	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

	call	ret	han	exc	stm
call	$\lessdot$	$\doteq$	$\lessdot$	$\gtrdot$	$\lessdot$
ret	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
han	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	$\lessdot$
exc	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
stm	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

(a) OPM $M_{\mathbf{call}}$ (b) OPM $M_{\mathbf{stm}}$

Figure 1

2.2 POPACheck

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 [12] 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 [17, 16] 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 POMC Input/Output Format

POMC takes in input plain text files of two possible formats.

3.1 Providing input models as OPA

The first input format contains a requirement specification in terms of a list of POTL formulas, and an OPA to be checked against them. This format is only supported by the explicit-state engine. An input file must be as follows:

```
formulas = FORMULA [, FORMULA ...] ;
prec = SL PR SL [, SL PR SL ...] ;
opa:
  initials = STATE_SET ;
  finals = STATE_SET ;
  deltaPush = (STATE, AP_SET, STATE_SET)
              [, (STATE, AP_SET, STATE_SET) ...] ;
  deltaShift = (STATE, AP_SET, STATE_SET)
              [, (STATE, AP_SET, STATE_SET) ...] ;
  deltaPop = (STATE, STATE, STATE_SET)
              [, (STATE, STATE, STATE_SET) ...] ;
```

where `STATE_SET` is either a single state, or a space-separated list of states, surrounded by parentheses. States are non-negative integer numbers (e.g. `(0 1 ...)`). `AP_SET` is a space-separated list of atomic propositions, surrounded by parentheses (e.g. `(call p1)` or `("call" "p1")`). In more detail:

- `prec` is followed by a comma-separated list of precedence relations between structural labels, that make up an Operator Precedence Matrix. The list is terminated by a semicolon. Precedence relations (`PR`) can be one of `<`, `=`, or `>`, which respectively mean $\lessdot$, $\doteq$, and $\gtrdot$. Structural labels (`SL`) can be any sequence of alphabetic characters.

- `formulas` is followed by a comma-separated, semicolon-terminated list of POTL formulas. The syntax of such formulas is defined later in this section.

- `opa` is followed by the explicit description of an OPA or an ωOPBA. The list of initial and final states must be given, as well as the transition relations. Whether the given automaton is to be interpreted as an OPA or ωOPBA is decided by the `--finite` and `--infinite` command-line arguments.

Additionally, POMC input files may contain C++-style single-line comments starting with \\, and C-style multi-line comments enclosed in `/*` and `*/`.

External files can be included with

```
include = "path/to/file.inc";
```

where the path is relative to the `pomc` file location.

POTL formulas can be written by using the operators in the "POMC Operator" column of Table 1, following the same syntax rules as in [6]. Normal and structural labels can be expressed as normal atomic propositions.

Once POMC is executed on an input file in the format above, it checks whether the given OPA satisfies the given formulas, one by one.

Consider the example input file `1-generic-small.pomc`, reported below:

```
prec = call < call, call = ret, call < han, call > exc,
       ret > call,  ret > ret,  ret > han,  ret > exc,
```

```
        han < call,  han > ret,  han < han,  han = exc,
        exc > call,  exc > ret,  exc > han,  exc > exc;

formulas = G ((call And pb And (T Sd (call And pa)))
                --> (PNu exc Or XNu exc));

opa:
  initials = 0;
  finals = 10;
  deltaPush =
    (0, (call pa),   1),
    (1, (han),       2),
    (2, (call pb),   3),
    (3, (call pc),   4),
    (4, (call pc),   4),
    (6, (call perr), 7),
    (8, (call perr), 7);
  deltaShift =
    (4, (exc),       5),
    (7, (ret perr),  7),
    (9, (ret pa),    11);
  deltaPop =
    (4, 2, 4),
    (4, 3, 4),
    (4, 4, 4),
    (5, 1, 6),
    (7, 6, 8),
    (7, 8, 9),
    (11, 0, 10);
```

First, OPM $M_{\mathbf{call}}$ from [6] (Figure 1a) is chosen.

The meaning of the formula `G ((call And pb And (T Sd (call And pa)))` `-> (PNu exc Or XNu exc))`, or $\square\big((\mathbf{call} \wedge \mathrm{p}_B \wedge Scall(\top, \mathrm{p}_A)) \implies CallThr(\top)\big)$, is explained in the paper.

POMC will check the OPA against the formula, yielding the following output:

```
Model Checking
Formula: G ((("call" And "pb") And (T Sd ("call" And "pa")))
                --> ((PNu "exc") Or (XNu "exc")))
Input OPA state count: 12
Result:  True
Elapsed time: 14.59 s


Total elapsed time: 14.59 s (1.4593e1 s)
```

Indeed, the OPA does satisfy the formula. POMC also outputs the time taken by each acceptance check and, when a formula is rejected, a (partial) counterexample trace.

Group	POTL Operator	POMC 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	–
	$\bigcirc_H^d$	HNd	Prefix	–
	$\bigcirc_H^u$	HNu	Prefix	–
	$\ominus_H^d$	HBd	Prefix	–
	$\ominus_H^u$	HBu	Prefix	–
	$\Diamond$	F, Eventually	Prefix	–
	$\Box$	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 1: This table contains all currently supported POTL 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.

```
PROGRAM := <DECL; ...> FUNCTION <FUNCTION ...>
DECL := TYPE IDENTIFIER <, IDENTIFIER ...>
TYPE := bool | uINT | sINT | uINT[INT] | sINT[INT]
FUNCTION := IDENTIFIER (<FARG, ...>) { <DECL; ...> STMT; <STMT; ...> }
FARG := TYPE IDENTIFIER | TYPE & IDENTIFIER
STMT := LVALUE = BEXPR
      | LVALUE = *
      | while (GUARD) { <STMT; ...> }
      | if (GUARD) { <STMT; ...> } else { <STMT; ...> }
      | try { <STMT; ...> } catch { <STMT; ...> }
      | IDENTIFIER(<EXPR, ...>)
      | throw
GUARD := * | EXPR
LVALUE := IDENTIFIER | IDENTIFIER[EXPR]
EXPR := EXPR || CONJ | CONJ
CONJ := CONJ && BTERM | BTERM
BTERM := IEXPR COMP IEXPR | IEXPR
COMP := == | != | < | <= | > | >=
IEXPR := IEXPR + PEXPR | IEXPR - PEXPR | PEXPR
PEXPR := PEXPR * ITERM | PEXPR / ITERM | ITERM
ITERM := !ITERM | (EXPR) | IDENTIFIER | IDENTIFIER[EXPR] | LITERAL
LITERAL := <+|-> INTuINT | <+|-> INTsINT | true | false
```

Figure 2: MiniProc syntax.

3.2 Providing MiniProc input models

The second kind of input files also contain POTL formulas, and a program in the *MiniProc* language to be checked against them. MiniProc is a simplified procedural programming language, where variables are all fixed-size (note that MiniProc is not Turing-complete, so any use of the word 'program' when referring to it is a deliberate abuse of terminology). This limitation allows POMC to translate every MiniProc program into an OPA, that is then checked against the supplied formulas. This kind of input files have this form:

```
formulas = FORMULA [, FORMULA ...] ;
program:
PROGRAM
```

The syntax of MiniProc programs is reported in Figure 2. In the definition, non-terminal symbols are uppercase, and keywords lowercase. Parts surrounded by angle brackets are optional, and ellipses mean that the enclosing group can be repeated zero or more times. An IDENTIFIER is any sequence of letters, numbers, or characters '.', ':' and '_', starting with a letter or an underscore.

The program starts with a variable declaration, which must include all global variables used in the program. Variables can be Boolean, or of signed or unsigned fixed-width integer types, or fixed-size arrays thereof. Then, a sequence of functions are defined, the first one being the entry-point to the program. Functions can have formal parameters that are passed by value or by value-result,[2] the latter being marked

---

[2]When a parameter is passed by value-result, the actual parameter is copied into the formal parameter

9

with the `&` symbol. Function bodies consist of semicolon-separated statements, which start after zero or more lists of local variables. Assignments, while loops and ifs have the usual semantics. The try-catch statement executes the catch block whenever an exception is thrown by any statement in the try block (or any function it calls). Exceptions are thrown by the `throw` statement, and they are not typed (i.e., there is no way to distinguish different kinds of exceptions). Functions can be called by prepending their name to actual parameters enclosed in parentheses. Actual parameters passed by value-result can only be variable names. Expressions can be made of the usual arithmetic operations when they involve integer variables, and arrays can be indexed by integer expressions enclosed in square brackets, both for assigning and reading them. Integer literals can be specified by a decimal number followed by the type of the literal (e.g., `u8` for an 8-bit unsigned integer, `s16` for a 16-bit signed integer, etc.), possibly preceded by its sign. Boolean expressions can contain comparisons between integers, and can be composed through the logical and (`&&`), or (`||`) and negation (`!`) operators.

POMC automatically translates such programs into OPA or ωOPBA, depending on whether finite- or infinite-word model checking has been chosen. The way this is done is detailed in Appendix A.

It is possible to declare *modules* by including a double colon (`::`) in function names. E.g., function `A::B::C()` is contained in module `A::B`, which is contained in `A`. In the OPA resulting from the program, the module names hold whenever a contained function is called or returns. This is useful for referring to multiple functions at once in POTL formulas, hence drastically reducing formula length and closure size.

When providing input models as programs, it is possible to use MiniProc expressions as atomic propositions in POTL formulas by using the syntax

```
[ IDENTIFIER | EXPR ]
```

where `IDENTIFIER`, which is optional, is a function name and `EXPR` is any MiniProc expression as defined in Figure 2. The expression will be evaluated in the scope of the specified function, or in the global scope if none is given; it will evaluate to false during the execution of all other functions. The expression may only refer to variables either global or local to the specified function, and an error is raised otherwise.

An example input file is given below:

```
formulas = G ((call And pb And (call Sd (call And pa)))
               --> (PNu exc Or XNu exc));

program:
var foo;

pa() {
  foo = false;
  try {
    pb();
  } catch {
    pc();
  }
}
```

---

when the function is called and, when the function returns, the value of the formal parameter is copied back into the actual parameter (which must be a variable).

$$prog \coloneqq [decl; \dots] \ func \ [func \dots]$$
$$decl \coloneqq type \ identifier \ [, identifier \dots]$$
$$type \coloneqq \texttt{bool} \mid \texttt{u}int \mid \texttt{s}int \mid \texttt{u}int[int] \mid \texttt{s}int[int]$$
$$func \coloneqq f(type \ [\&]x_1 \ [, type \ [\&]x_2 \dots])$$
$$\{[decl; \dots] \ block\}$$
$$stmt \coloneqq lval = e$$
$$\mid lval = \texttt{Distribution(}\dots\texttt{)}$$
$$\mid lval = e_1\{e_2 : e_3\}[e_4\{e_5 : e_6\} \dots]e_n$$
$$\mid [\texttt{query}] \ f(e_1 \mid lval_1 \ [, e_2 \mid lval_2 \dots])$$
$$\mid \texttt{if } (e) \ \{block\} \ \texttt{else} \ \{block\}$$
$$\mid \texttt{while } (e) \ \{block\}$$
$$\mid \texttt{observe } (e)$$
$$block \coloneqq stmt; [stmt \dots;]$$
$$lval \coloneqq identifier \mid identifier[e]$$

Figure 3: MiniProb syntax.

```
pb() {
  if (foo) {
    throw;
  } else {}
}

pc() { }
```

POMC prints the following:

```
Model Checking
Formula: G ((("call" And "pb") And ("call" Sd ("call" And "pa")))
  --> ((PNu "exc") Or (XNu "exc")))
Input OPA state count: 28
Result:  True
Elapsed time: 803.7 ms


Total elapsed time: 803.7 ms (8.0370e-1 s)
```

4 POPACheck Input/Output Language

POPACheck analyzes programs written in MiniProb, a simple probabilistic programming language (Fig. 3). 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 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`.

4.1 Model Check Queries

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

$$\text{probabilistic query: } q;$$

where q is one of the queries of Table 2. When a formula is needed, it has to be placed on a new line with syntax:

$$\text{formula: } f;$$

where f follows the syntax of Table 3. For some technical reasons explained in [11], 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.

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.

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:

$$\varphi \left\{ P \right\} \theta$$

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:

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 2: Available queries.

```
            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~~	–
	$\diamondsuit$ (LTL)	F, Eventually	Prefix	–
	$\bigcirc$ (LTL)	N	Prefix	–
	$\mathcal{U}$ (LTL)	U	Infix	Right
	$\square$ (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 3: 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.**

4.2 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.

5   Some experiments with POMC

In this section we report the results of some experiments provided in the `eval` directory. The experiments were executed on a laptop with a 2.2 GHz Intel processor and 15 GiB of RAM, running Ubuntu GNU/Linux 20.04. Here we only report results with the explicit-state engine.

These are only a few of the experiments shipped with this repository, and this section is intended to provide a sample of them, so it will not be updated frequently.

5.1   Directory `automata/opa-cav`

This directory contains a few programs modeled as OPA, on which POMC proves or disproves some interesting specifications. The resources employed by POMC on such

	Benchmark name	# states	Time (ms)	Memory (KiB)		Result
				Total	MC only	
1	generic small	12	867	70,040	10,166	True
2	generic medium	24	673	70,064	4,043	False
3	generic larger	30	1,014	70,063	14,160	True
4	Jensen	42	305	70,050	3,154	True
5	unsafe stack	63	1,493	109,610	43,177	False
6	safe stack	77	637	70,089	7,234	True
7	unsafe stack neutrality	63	5,286	383,312	167,654	True
8	safe stack neutrality	77	840	70,077	16,773	True

Table 4: Results of the evaluation.

tasks are reported in Table 4. If you wish to repeat such experiments, you may run the following commands:

```
$ cd ~/path/to/POMC-sources/eval
$ ./mcbench.py -f automata/opa-cav
```

Generic procedural programs. Formula

$$\Box\big((\mathbf{call} \wedge \mathrm{p}_B \wedge Scall(\top, \mathrm{p}_A)) \implies CallThr(\top)\big)$$

means that whenever procedure p_B is executed and at least one instance of p_A is on the stack, p_B is terminated by an exception. We checked it against three OPA representing some simple procedural programs with exceptions and recursive procedures. The formula holds on benchmarks no. 1 and 3, but not on no. 2.

Stack Inspection. [9] contains an example Java program for managing a bank account, which uses the security framework of the Java Development Kit to enforce user permissions. The program allows the user to check the account balance, and to withdraw money. To perform such tasks, the invoking program must have been granted permissions `CanPay` and `Debit`, respectively. We modeled such program as an OPA (bench. 4), and proved that the program enforces such security measures effectively by checking it against the formula

$$\Box(\mathbf{call} \wedge \mathtt{read} \implies \neg(\top\, \mathcal{S}_\chi^d\, (\mathbf{call} \wedge \neg\mathtt{CanPay} \wedge \neg\mathtt{read})))$$

meaning that the account balance cannot be read if some function in the stack lacks the `CanPay` permission (a similar formula checks the `Debit` permission).

Exception Safety. [14] is a tutorial on how to make exception-safe generic containers in C++. It presents two implementations of a generic stack data structure, parametric on the element type `T`. The first one is not exception-safe: if the constructor of `T` throws an exception during a pop action, the topmost element is removed, but it is not returned, and it is lost. This violates the strong exception safety [1] requirement that each operation is rolled back if an exception is thrown. The second version of the data structure instead satisfies such requirement.

While exception safety is, in general, undecidable, it is possible to prove the stronger requirement that each modification to the data structure is only committed once no

Formula	Time (ms)	Memory (KiB) Tot.	MC	Result
$\chi_F^d \mathrm{p}_{Err}$	1.1	70,095	175	False
$\bigcirc^d(\bigcirc^d(\mathbf{call} \wedge \chi_F^u \mathbf{exc}))$	21.0	70,095	1,290	False
$\bigcirc^d(\mathbf{han} \wedge (\chi_F^d(\mathbf{exc} \wedge \chi_P^u \mathbf{call})))$	42.2	70,088	2,297	False
$\square(\mathbf{exc} \implies \chi_P^u \mathbf{call})$	10.7	70,099	839	True
$\top \, \mathcal{U}_\chi^d \, \mathbf{exc}$	2.2	70,093	121	False
$\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^u \, \mathbf{exc}))$	4.3	70,094	113	False
$\square((\mathbf{call} \wedge \mathrm{p}_A \wedge (\neg \mathbf{ret} \, \mathcal{U}_\chi^d \, \mathrm{WRx})) \implies \chi_F^u \mathbf{exc})$	3,257.7	238,833	102,582	True
$\bigcirc^d(\bigcirc^u \mathbf{call})$	0.7	70,094	139	False
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\ominus^u \mathbf{call})))$	3.4	70,108	126	False
$\chi_F^d(\bigcirc^d(\ominus^u \mathbf{call}))$	1.3	70,096	137	False
$\square((\mathbf{call} \wedge \mathrm{p}_A \wedge CallThr(\top)) \implies CallThr(\mathrm{e}_B))$	7,793.7	402,420	173,639	False
$\diamond(\bigcirc_H^d \mathrm{p}_B)$	2.1	70,097	114	False
$\diamond(\ominus_H^d \mathrm{p}_B)$	2.8	70,097	114	False
$\diamond(\mathrm{p}_A \wedge (\mathbf{call} \, \mathcal{U}_H^d \, \mathrm{p}_C))$	594.9	77,806	29,786	True
$\diamond(\mathrm{p}_C \wedge (\mathbf{call} \, \mathcal{S}_H^d \, \mathrm{p}_A))$	676.6	96,296	37,949	True
$\square((\mathrm{p}_C \wedge \chi_F^u \mathbf{exc}) \implies (\neg \mathrm{p}_A \, \mathcal{S}_H^d \, \mathrm{p}_B))$	—	—	—	OOM
$\square(\mathbf{call} \wedge \mathrm{p}_B \implies \neg \mathrm{p}_C \, \mathcal{U}_H^u \, \mathrm{p}_{Err})$	198.2	70,088	10,606	True
$\diamond(\bigcirc_H^u \mathrm{p}_{Err})$	1.1	70,093	114	False
$\diamond(\ominus_H^u \mathrm{p}_{Err})$	1.2	70,089	114	False
$\diamond(\mathrm{p}_A \wedge (\mathbf{call} \, \mathcal{U}_H^u \, \mathrm{p}_B))$	10.3	70,105	115	False
$\diamond(\mathrm{p}_B \wedge (\mathbf{call} \, \mathcal{S}_H^u \, \mathrm{p}_A))$	10.8	70,095	115	False
$\square(\mathbf{call} \implies \chi_F^d \mathbf{ret})$	3.0	70,095	112	False
$\square(\mathbf{call} \implies \neg \bigcirc^u \mathbf{exc})$	1.9	70,106	113	False
$\square(\mathbf{call} \wedge \mathrm{p}_A \implies \neg CallThr(\top))$	110.7	70,094	4,937	False
$\square(\mathbf{exc} \implies \neg(\ominus^u(\mathbf{call} \wedge \mathrm{p}_A) \vee \chi_P^u(\mathbf{call} \wedge \mathrm{p}_A)))$	28.9	70,095	112	False
$\square((\mathbf{call} \wedge \mathrm{p}_B \wedge (\mathbf{call} \, \mathcal{S}_\chi^d \, (\mathbf{call} \wedge \mathrm{p}_A))) \implies CallThr(\top)$	926.1	70,104	13,310	True
$\square(\mathbf{han} \implies \chi_F^u \mathbf{ret})$	17.0	70,079	1,252	True
$\top \, \mathcal{U}_\chi^u \, \mathbf{exc}$	7.7	70,101	121	True
$\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^u \, \mathbf{exc}))$	44.6	70,104	2,376	True
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^u \, \mathbf{exc})))$	123.7	70,090	5,261	False
$\square(\mathbf{call} \wedge \mathrm{p}_C \implies (\top \, \mathcal{U}_\chi^u \, \mathbf{exc} \wedge \chi_P^d \mathbf{han}))$	92.9	70,096	1,346	False
$\mathbf{call} \, \mathcal{U}_\chi^d \, (\mathbf{ret} \wedge \mathrm{p}_{Err})$	1.8	70,107	114	False
$\chi_F^d(\mathbf{call} \wedge ((\mathbf{call} \vee \mathbf{exc}) \, \mathcal{S}_\chi^u \, \mathrm{p}_B))$	10.8	70,086	117	False
$\bigcirc^d(\bigcirc^d((\mathbf{call} \vee \mathbf{exc}) \, \mathcal{U}_\chi^u \, \mathbf{ret}))$	5.3	70,094	114	False

Table 5: Results of the additional experiments on OPA "generic larger".

more exceptions can be thrown. We modeled both versions as OPA, and checked such requirement with the following formula:

$$\square(\mathbf{exc} \implies \neg((\ominus^u \texttt{modified} \vee \chi_P^u \texttt{modified}) \wedge \chi_P^u(\texttt{Stack :: push} \vee \texttt{Stack :: pop})))$$

POMC successfully found a counterexample for the first implementation (5), and proved the safety of the second one (6).

Additionally, we proved that both implementations are *exception neutral* (7, 8), i.e. they do not block exceptions thrown by the underlying types.

5.2 Directory `automata/opa-more`

This directory contains more experiments devised with the purpose of testing all POTL operators, also in order to find the most critical cases. In fact, the complexity of POTL model checking is exponential in the length of the formula. This is of

Benchmark name	# states	Time (s)	Memory (KiB)		Result
			Total	MC only	
doubleHan	22	52.96	2,091,256	869,661	False
jensen	1236	1.97	73,712	17,339	True
simpleExc	19	65.42	3,278,876	1,353,000	False
simpleExcNoHan	12	37.72	1,510,524	656,422	False
simpleIfElse	28	27.62	942,280	383,231	False
simpleIfThen	28	30.67	1,046,584	415,648	False
simpleWhile	16	0.09	73,768	3,251	True
stackSafe	340	31.51	653,616	265,363	True
stackUnsafe	162	16.48	532,736	224,573	False

Table 6: Results of the evaluation of `miniproc` files.

course unsurprising, since it subsumes logics such as LTL and NWTL, whose model checking is also exponential. Actually, model checking is feasible for many specifications useful in practice. There are, however, some cases in which the exponentiality of the construction becomes evident.

In Table 5 we show the results of model checking numerous POTL formulas on one of the OPA representing generic procedural programs. Some of them are checked very quickly, while others require a long execution time and a very large amount of memory. POMC runs out of memory on one of such formulas. We were able to run it in 367 seconds on a server with a 2.0 GHz 16-core AMD CPU and 500 GB of RAM. If you wish to repeat such experiments, you may run the following commands:

```
$ cd ~/path/to/POMC-sources/eval
$ ./mcbench.py -f opa-more/generic-larger
```

Of course, a machine with an appropriate amount of RAM is needed.

5.3 Directory `miniproc/finite`

This directory contains a few verification tasks in which the model has been expressed as a MiniProc program. Each file in this directory contains multiple formulas.

`jensen.pomc`, `stackUnsafe.pomc` and `stackSafe.pomc` contain the same tasks as those with the same name described in Section 5.1. This time, however, models are expressed as MiniProc programs, and the resulting OPA contain many more states.

Other files contain simpler programs, checked against all formulas form Table 5.

Table 6 reports the results of such experiments. When more than one formula is checked in a single file, the reported result is True only if all formulas are verified, False if at least one of them is not.

6 Source Code

The POMC suite is open source. The source code is contained in directory `src/Pomc/`. We describe the contents of each file below.

Parse This directory contains the parser for input files.

18

Check.hs This file contains the data structures and functions that implement the translation of POTL formulas into OPA. The `check` and `fastcheck` functions build the OPA and check for string acceptance. `makeOpa` returns a thunk containing an un-evaluated OPA, which is built on-the-fly while the calling context evaluates the transition functions.

DoubleSet.hs a data structure used by the SCC-finding algorithm.

Encoding.hs contains a data structure that represents a set of POTL formulas as a bit vector. We use it to encode OPA states in a memory-efficient form in Check.hs.

GStack.hs contains a custom implementation of a LIFO stack for the ωOPBA emptiness algorithms.

LogUtils.hs contains some logging-related functions.

MaybeMap.hs contains another helper data structure for the emptiness algorithms.

MiniProc.hs contains code that translates MiniProc programs into OPA.

ModelChecker.hs contains the model checking launcher functions, and a data structure to represent the input OPA to be checked explicitly. It calls `makeOpa` to translate the negation of the specification into an equivalent OPA, creates a thunk representing an un-evaluated intersection of the two OPA, and then uses the reachability algorithm from Satisfiability.hs to determine emptiness.

Opa.hs contains an implementation of OPA, which is used to test string acceptance.

OpaGen.hs contains a simple automated OPA generator (still experimental).

Potl.hs defines the datatype for POTL formulas.

Prec.hs defines the data type for precedence relations.

Prop.hs defines the data type for atomic propositions.

PropConv.hs contains dome functions useful to change the representation of atomic propositions from strings to unsigned integers. This is used by other parts of the program to achieve better performances, as strings are represented as lists of char in Haskell, which is quite inefficient.

Satisfiability.hs contains the reachability algorithms used in the model checker to decide OPA emptiness. They can also be use to decide satisfiability of a formula.

SatUtil.hs contains utility data structures for the satisfiability algorithms.

SCCAlgorithm.hs contains the implementation of the algorithm for finding strongly connected components in ωOPBA employed for the emptiness check.

SetMap.hs contains another helper data structure for satisfiability.

State.hs contains the data type used to represent OPA states.

TimeUtils.hs contains functions used to measure time.

TripleHashTable.hs contains a hash table used in the emptiness check.

Z3Encoding.hs contains the SMT-based engine.

The source code of POPACheck is contained in directory `src/Pomc/`.

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 [11]).

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 [12].

6.1 Test Suite

The `test` directory contains regression tests based on the HUnit provider of the Tasty[3] 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.

---

[3] https://github.com/UnkindPartition/tasty

Acknowledgements

References

[1] D. Abrahams. Exception-Safety in Generic Components. In *Generic Programming*, pages 69–79. Springer, 2000. `doi:10.1007/3-540-39953-4\_6`.

[2] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. *LMCS*, 4(4), 2008. `doi:10.2168/LMCS-4(4:11)2008`.

[3] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS 2004*, pages 467–481. Springer, 2004. `doi:10.1007/978-3-540-24730-2\_35`.

[4] R. Alur and P. Madhusudan. Adding nesting structure to words. *JACM*, 56(3), 2009. `doi:10.1145/1516512.1516518`.

[5] Michele Chiari, Luca Geatti, Nicola Gigante, and Matteo Pradella. Smt-based symbolic model-checking for operator precedence languages. In *CAV '24*, volume 14681 of *LNCS*, pages 387–408. Springer, 2024. `doi:10.1007/978-3-031-65627-9_19`.

[6] 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`.

[7] 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`.

[8] 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`.

[9] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *Proc. '99 IEEE Symp. on Security and Privacy*, pages 89–103, 1999. `doi:10.1109/SECPRI.1999.766902`.

[10] 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`.

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

[12] 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.

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

[14] Herb Sutter. Exception-safe generic containers. *C++ Report*, 1997. URL: https://ptgmedia.pearsoncmg.com/imprint_downloads/informit/aw/meyerscddemo/DEMO/MAGAZINE/SU_FRAME.HTM.

[15] 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.

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

[17] 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.

A From MiniProc to OPA

A MiniProc program can be converted to an equivalent OPA or ωOPBA. This is done in two stages: first, we build an *extended* OPA whose transitions are labeled with Boolean expressions and assignments; then, we convert such OPA to a normal one, ready for model checking. Note that this construction is outdated as it does not explain how we deal with things such as integer variables and function arguments, but it should still give a good overview of the process.

A.1 Extended OPA

Given a MiniProc program P and the set I_P of identifiers in P, we call $L_P = BExp_P \cup Ass_P$ the set of labels on P, where $BExp_P$ and Ass_P are resp. the sets of Boolean expressions and assignments on I_P. We build the extended OPA

$$\mathcal{A}_P^E = (\Sigma_P, M_{\mathbf{call}}, Q_P^E, \{q_0\}, \{q_f\}, \delta_P^E)$$

with $\Sigma_P = \Sigma_{\mathbf{call}} \cup L_P$. Q_P and δ_P^E are built inductively on the program structure. For each statement s in P, we define the set of entry state/label pairs $En_s \subseteq Q_P \times L_P$. Each entry state is labeled with an element form either $BExp_P$ or Ass_P, but not both.

Functions For each function f in P we define a set of entry states $En_f = En_s$, where s is the first statement in the function's body; we also add transitions and states $q_f^l \overset{\mathbf{ret}\ f}{\dashrightarrow} q_f^r$, to which we link the last statement in f, and $q_f^t \overset{\mathbf{exc}}{\dashrightarrow} q_f^e$, which implements `throw` statements.

Function Call For a call s to function f, we add $q_s \overset{\mathbf{call}\ f\ l}{\longrightarrow} q$ for all $(q, l) \in En_f$, and $q_f^t \overset{q_s}{\Longrightarrow} q_{f'}^t$, where f' is the function containing s. Let s' be the successor of s: we add $q_f^r \overset{q_s\ l}{\Longrightarrow} q$ for all $(q, l) \in En_s$.

Assignments For each assignment s we add $q_s \xrightarrow{\textbf{stm } s} q_s$, and set $En_s = Ex_s = \{(q_s, \top)\}$. Let s' be the successor of s: we add $q_s \xRightarrow{(q_s, l)} q$ for all $(q, l) \in En_s$.

If-then-else For each statement s of the form `if` b_s `then {` $s_1; \dots; s_n$ `} else {` $s_{n+1}; \dots; s_m$ `}` we have $En_s = \{(q, b_s \wedge l) \mid (q, l) \in En_{s_1}\} \cup \{(q, \neg b_s \wedge l) \mid (q, l) \in En_{s_{n+1}}\}$.

While For a statement s of the form `while` b_s `{` $s_1; \dots; s_n$ `}` we set $En_s = \{(q, b_s \wedge l) \mid (q, l) \in En_{s_1}\} \cup \{(q, \neg b_s \wedge l) \mid (q, l) \in En_{s_{n+1}}\}$, where s_{n+1} is the successor of s. Also, both s_{n+1} and s itself are considered as successors of s_n, and their entry sets are merged.

Throw For a `throw` statement s in a function f we just set $En_s = \{(q_f^t, \top)\}$.

Try-Catch For a statement s in function f of the form `try {` $s_1; \dots; s_n$ `} catch {` $s_{n+1}; \dots; s_m$ `}`, we add a new state q_s and set $En_s = \{(q_s, \top)\}$, and a push transition $q_s \xrightarrow{\textbf{han } l} q$ for each $(q, l) \in En_{s_1}$ that installs the handler. We first deal with the case when an exception is caught. We add pop transitions $q_f^e \xRightarrow{q_s \, l} q$ for each $(q, l) \in En_{s_{n+1}}$ that pop the handler when an exception is thrown in the try block, and pass the execution flow to the catch block. Then, statement s_m is linked to the entry states of s', the first statement after s (how this is done depends on what kind of statement s_m is). For the case when no exception is thrown, we add a shift transition that simulates a dummy `throw` statement t after s_n, to uninstall the handler. When lowering s_n, we consider t as its next statement, add states q_t and q_t', and set $En_t = \{(q_t, \top)\}$. Then we add $q_t \xdashrightarrow{\textbf{exc } dummy} q_t'$, and $q_t' \xRightarrow{q_s \, l} q$ for all $(q, l) \in En_{s'}$, which pop the handler and continue the execution with the first statement after s.

Finally, if f_0 is the first function listed in the MiniProc program, we add transitions $q_0 \xrightarrow{\textbf{call } f_0 \, l} q$ for all $(q, l) \in En_{f_0}$, and $q_{f_0}^r \xRightarrow{q_0} q_f$.

A.2 From extended OPA to OPA

We expand states of $\mathcal{A}_P^E$ with all possible variable valuations, to obtain OPA

$$\mathcal{A}_P = (\Sigma_{\textbf{call}} \times I_P, M_{\textbf{call}}, Q_P, \{q_0\} \times \{0,1\}^{|I_P|}, \{q_f\} \times \{0,1\}^{|I_P|}, \delta_P),$$

where $Q_P \subseteq Q_P^E \times \{0,1\}^{|I_P|}$. Each state is a pair (q, v) with $q \in Q_P^E$ and v is a bitvector representing a possible valuation of variables that hold in q. By $v \models l$ we mean that the variable valuation $v \in \{0,1\}^{|I_P|}$ satisfies Boolean expression $l \in BExp_P$; if $l = (x := e) \in Ass_P$ with $x \in I_P$ and $e \in BExp_P$ we mean $v \models x \iff e$. By $\text{vars}(v)$ we denote the set of variables satisfied by $v \in \{0,1\}^{|I_P|}$. We define $Q_P := \bigcup_{i \in \mathbb{N}} Q_P^i$ inductively through the following equations:

$$Q_P^0 := \{q_0\} \times \{0,1\}^{|I_P|}$$
$$Q_P^{n+1} := \{(q, v) \mid q' \in Q_P^n, (q', a \, l, q) \in \delta_P^E, v \models l\}$$

This is implemented through a depth-first visit of $\mathcal{A}_P^E$, from which we derive

$$\delta_P := \{q \stackrel{a \;\; \mathrm{vars(v)}}{\longrightarrow} q' \mid q \stackrel{a \; l}{\longrightarrow} q' \in \delta_P^E, \; q, q' \in Q_P\}$$
$$\cup \{q \stackrel{a \;\; \mathrm{vars(v)}}{\dashrightarrow} q' \mid q \stackrel{a \; l}{\dashrightarrow} q' \in \delta_P^E, \; q, q' \in Q_P\}$$
$$\cup \{q \stackrel{p}{\Longrightarrow} q' \mid q \stackrel{p \; l}{\Longrightarrow} q' \in \delta_P^E, \; q, q', p \in Q_P\}$$

Note that $\mathcal{A}_P$ has size exponential in $|I_P|$ in the worst case, but not in general, since only reachable variable assignments are considered.