POMC User's Guide

Michele Chiari

Abstract

POMC is an implementation of the automaton construction for formulas of Precedence-Oriented Temporal Logic (POTL), and the model checking procedure thereof. 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

Precedence-Oriented Temporal Logic (POTL) [5] is a novel temporal logic formalism based on the family of Operator Precedence Languages (OPL), 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.

Besides some results concerning its expressiveness, we introduced an automata-based model checking procedure for POTL. This procedure consists in 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.

POMC is a tool that implements the automaton construction for POTL, and a model checking procedure for it. For the time being, only the construction for finite words has been implemented. 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.

We used POMC to prove some interesting properties of programs which we modeled as OPA. All such experiments are contained in `pomc` files in the `opa` and `opa-more` subdirectories.

If you wish to examine the input formulas and OPA to such experiments more carefully, or to write your own, we describe the format of POMC input files in Section 3. We also demonstrate the use of the tool with a few experiments in Section 4. Finally, Section 5 contains a high-level description of the source code.

2 Quick-Start Guide

POMC has been developed in the Haskell programming language, and it has been packaged with the Haskell Tool Stack[1]. POMC can be built from sources by typing the following commands in a shell:

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

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

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

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. E.g.,

```
$ cd ~/path/to/POMC-sources/eval
$ python3 mcbench.py opa
```

evaluates all `*.pomc` files in directory `~/path/to/POMC-sources/eval/opa`. 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
$ python3 mcbench.py 1-generic-fig.2.pomc
                                  2-generic-medium.pomc
```

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

`-iter <#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)

`-jobs <#jobs>` Number of benchmarks to be run in parallel. If you provide a value greater than 1, make sure you increase the number of cores available to the VM accordingly. (Default: 1)

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

3 POMC Input/Output Format

POMC takes in input plain text files of the following format:

```
prec = Mcall | SL PR SL [, SL PR SL ...] ;
formulas = FORMULA [, FORMULA ...] ;
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 either `Mcall`, or a comma-separated list of precedence relations between structural labels, that make up an Operator Precedence Matrix. The list is terminated by a semicolon. If `Mcall`, then OPM $M_{\mathbf{call}}$ from [5] (reported in Figure 1) is used. Otherwise, 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. It is possible to use the wildcard `*` to match any structural label not explicitly specified, e.g., `call < *` means that the structural label `call` yields precedence to any other structural label, and `* < *` means that any not previously specified structural label yields precedence to all other structural labels. You must always specify `* > #`.

- `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. The list of initial and final states must be given, as well as the transition relations.

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

POTL formulas can be written by using the operators in the "POMC Operator" column of Table 1, following the same syntax rules as in the paper.

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-fig.2.pomc`, reported below:

```
prec = Mcall;
```

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.

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

Figure 1: OPM $M_{\mathbf{call}}$.

```
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 [5] (Figure 1) is chosen. The meaning of the formula `G ((call And pb And (T Sd (call And pa))) -> (PNu exc Or XNu exc))`, or $\square\big((\mathbf{call} \wedge \mathrm{p}_B \wedge \mathit{Scall}(\top, \mathrm{p}_A)) \implies \mathit{CallThr}(\top)\big)$, is explained in the paper.

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

```
_______________________________________________________
Evaluating file opa/1-generic-fig.2.pomc ...

Model Checking
Formula: G ((("call" And "pb") And (T Sd ("call" And "pa")))
                --> ((PNu "exc") Or (XNu "exc")))
```

	Benchmark name	# states	Time (s)	Mem. (MB)	Result
1	generic (Fig. 2)	12	14	126	True
2	generic medium	24	15	119	False
3	generic larger	30	16	233	True
4	Jensen	42	1	70	True
5	unsafe stack	63	46	185	False
6	safe stack	77	44	70	True
7	unsafe stack neutrality	63	62	1,787	True
8	safe stack neutrality	77	29	126	True

Table 2: Results of the evaluation.

```
Input OPA state count: 12
Result:  True
Elapsed time: 14.59 s


Total elapsed time: 14.59 s (1.4593e1 s)

Max memory used (KB): 129152
```

Indeed, the OPA does satisfy the formula. POMC additionally outputs the time taken by each acceptance check, and the maximum resident memory used.

4 Some experiments

In this section we report the results of some experiments, that are provided in the `eval` directory.

4.1 Directory `opa`

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 tasks are reported in Table 2. If you wish to repeat such experiments, you may run the following commands:

```
$ cd ~/artifact/contents
$ python3 mcbench.py opa
```

Generic procedural programs. Formula

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

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. [6] 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

$$\square(\mathbf{call} \land \mathtt{read} \implies \neg(\top \, \mathcal{S}^d_\chi \, (\mathbf{call} \land \neg\mathtt{CanPay} \land \neg\mathtt{read})))$$

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

$$\square(\mathbf{exc} \implies \neg((\circleddash^u \mathtt{modified} \lor \chi^u_P \mathtt{modified}) \land \chi^u_P(\mathtt{Stack::push} \lor \mathtt{Stack::pop})))$$

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.

4.2 Directory `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 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 Tables 3, 4 and 5 we show the results of model checking numerous POTL formulas on three 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. Because of the memory requirements of some of such formulas, these results were run on a server with a 2.0 GHz 16-core AMD CPU and 500 GB of

Formula	Time (s)
$\chi_F^d \mathrm{p}_{err}$	0.043
$\bigcirc^d(\bigcirc^d(\mathbf{call} \land \chi_F^u \mathbf{exc}))$	0.094
$\bigcirc^d(\mathbf{han} \land (\chi_F^d(\mathbf{exc} \land \chi_P^u \mathbf{call})))$	0.889
$\square(\mathbf{exc} \implies \chi_P^u \mathbf{call})$	0.051
$\top \,\mathcal{U}_\chi^d\, \mathbf{exc}$	0.006
$\bigcirc^d(\bigcirc^d(\top \,\mathcal{U}_\chi^d\, \mathbf{exc}))$	0.021
$\square((\mathbf{call} \land \mathrm{p}_A \land (\neg \mathbf{ret} \,\mathcal{U}_\chi^d\, \mathrm{WRx})) \implies \chi_F^u \mathbf{exc})$	214
$\bigcirc^d(\bigcirc^u \mathbf{call})$	0.001
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\ominus^u \mathbf{call})))$	0.003
$\chi_F^d(\bigcirc^d(\ominus^u \mathbf{call}))$	0.081
$\square((\mathbf{call} \land \mathrm{p}_A \land (\bigcirc^u \mathbf{exc} \lor \chi_F^u \mathbf{exc})) \implies (\bigcirc^u \mathrm{e}_B \lor \chi_F^u \mathrm{e}_B))$	541
$\Diamond(\bigcirc_H^d \mathrm{p}_B)$	0.053
$\Diamond(\ominus_H^d \mathrm{p}_B)$	0.010
$\Diamond(\mathrm{p}_A \land (\mathbf{call} \,\mathcal{U}_H^d\, \mathrm{p}_C))$	0.36
$\Diamond(\mathrm{p}_C \land (\mathbf{call} \,\mathcal{S}_H^d\, \mathrm{p}_A))$	0.252
$\square((\mathrm{p}_C \land \chi_F^u \mathbf{exc}) \implies (\neg \mathrm{p}_A \,\mathcal{S}_H^d\, \mathrm{p}_B))$	567
$\square(\mathbf{call} \land \mathrm{p}_B \implies \neg \mathrm{p}_C \,\mathcal{U}_H^u\, \mathrm{p}_{err})$	0.137
$\Diamond(\bigcirc_H^u \mathrm{p}_{err})$	0.003
$\Diamond(\ominus_H^u \mathrm{p}_{err})$	0.002
$\Diamond(\mathrm{p}_A \land (\mathbf{call} \,\mathcal{U}_H^u\, \mathrm{p}_B))$	0.032
$\Diamond(\mathrm{p}_B \land (\mathbf{call} \,\mathcal{S}_H^u\, \mathrm{p}_A))$	0.031
$\bigcirc^d \mathbf{call}$	0.001
$\bigcirc^d(\bigcirc^d \mathbf{call})$	0.001
$\bigcirc^d(\bigcirc^d(\bigcirc^d \mathbf{call}))$	0.001
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\bigcirc^d \mathbf{call})))$	0.004
$\square(\mathbf{call} \implies \chi_F^d \mathbf{ret})$	0.095
$\square(\mathbf{call} \implies \neg \bigcirc^u \mathbf{exc})$	0.003
$\square(\mathbf{call} \land \mathrm{p}_A \implies \neg(\bigcirc^u \mathbf{exc} \lor \chi_F^u \mathbf{exc}))$	0.647
$\square(\mathbf{exc} \implies \neg(\ominus^u(\mathbf{call} \land \mathrm{p}_A) \lor \chi_P^u(\mathbf{call} \land \mathrm{p}_A)))$	0.676
$\square((\mathbf{call} \land \mathrm{p}_B \land (\mathbf{call} \,\mathcal{S}_\chi^d\, (\mathbf{call} \land \mathrm{p}_A))) \implies \bigcirc^u \mathbf{exc} \lor \chi_F^u \mathbf{exc}$	32.4
$\square(\mathbf{han} \implies \chi_F^u \mathbf{ret})$	0.090
$\top \,\mathcal{U}_\chi^u\, \mathbf{exc}$	0.045
$\bigcirc^d(\bigcirc^d(\top \,\mathcal{U}_\chi^u\, \mathbf{exc}))$	0.205
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\top \,\mathcal{U}_\chi^u\, \mathbf{exc})))$	0.694
$\square(\mathbf{call} \land \mathrm{p}_C \implies (\top \,\mathcal{U}_\chi^u\, \mathbf{exc} \land \chi_P^d \mathbf{han}))$	5.06
$\mathbf{call} \,\mathcal{U}_\chi^d\, (\mathbf{ret} \land \mathrm{p}_{err})$	0.145
$\chi_F^d(\mathbf{call} \land ((\mathbf{call} \lor \mathbf{exc}) \,\mathcal{S}_\chi^u\, \mathrm{p}_B))$	1.44
$\bigcirc^d(\bigcirc^d((\mathbf{call} \lor \mathbf{exc}) \,\mathcal{U}_\chi^u\, \mathbf{ret}))$	0.332
Total	1366
Maximum memory used (MB)	18,956.048

Table 3: Results of the additional experiments on OPA "generic (Fig. 2)".

Formula	Time (s)
$\chi_F^d \mathrm{p}_{err}$	0.006
$\bigcirc^d(\bigcirc^d(\mathbf{call} \wedge \chi_F^u \mathbf{exc}))$	0.543
$\bigcirc^d(\mathbf{han} \wedge (\chi_F^d(\mathbf{exc} \wedge \chi_P^u \mathbf{call})))$	3.24
$\square(\mathbf{exc} \implies \chi_P^u \mathbf{call})$	0.055
$\top \, \mathcal{U}_\chi^d \, \mathbf{exc}$	0.017
$\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^d \, \mathbf{exc}))$	0.041
$\square((\mathbf{call} \wedge \mathrm{p}_A \wedge (\neg \mathbf{ret} \, \mathcal{U}_\chi^d \, \mathrm{WRx})) \implies \chi_F^u \mathbf{exc})$	616
$\bigcirc^d(\bigcirc^u \mathbf{call})$	0.001
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\ominus^u \mathbf{call})))$	0.009
$\chi_F^d(\bigcirc^d(\ominus^u \mathbf{call}))$	0.008
$\square((\mathbf{call} \wedge \mathrm{p}_A \wedge (\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc})) \implies (\bigcirc^u \mathrm{e}_B \vee \chi_F^u \mathrm{e}_B))$	673
$\Diamond(\bigcirc_H^d \mathrm{p}_B)$	0.005
$\Diamond(\ominus_H^d \mathrm{p}_B)$	0.007
$\Diamond(\mathrm{p}_A \wedge (\mathbf{call} \, \mathcal{U}_H^d \, \mathrm{p}_C))$	1.29
$\Diamond(\mathrm{p}_C \wedge (\mathbf{call} \, \mathcal{S}_H^d \, \mathrm{p}_A))$	1.48
$\square((\mathrm{p}_C \wedge \chi_F^u \mathbf{exc}) \implies (\neg \mathrm{p}_A \, \mathcal{S}_H^d \, \mathrm{p}_B))$	2253
$\square(\mathbf{call} \wedge \mathrm{p}_B \implies \neg \mathrm{p}_C \, \mathcal{U}_H^u \, \mathrm{p}_{err})$	0.310
$\Diamond(\bigcirc_H^u \mathrm{p}_{err})$	0.002
$\Diamond(\ominus_H^u \mathrm{p}_{err})$	0.001
$\Diamond(\mathrm{p}_A \wedge (\mathbf{call} \, \mathcal{U}_H^u \, \mathrm{p}_B))$	0.035
$\Diamond(\mathrm{p}_B \wedge (\mathbf{call} \, \mathcal{S}_H^u \, \mathrm{p}_A))$	0.028
$\bigcirc^d \mathbf{call}$	0.001
$\bigcirc^d(\bigcirc^d \mathbf{call})$	0.001
$\bigcirc^d(\bigcirc^d(\bigcirc^d \mathbf{call}))$	0.001
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\bigcirc^d \mathbf{call})))$	0.003
$\square(\mathbf{call} \implies \chi_F^d \mathbf{ret})$	0.037
$\square(\mathbf{call} \implies \neg \bigcirc^u \mathbf{exc})$	0.004
$\square(\mathbf{call} \wedge \mathrm{p}_A \implies \neg(\bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}))$	0.367
$\square(\mathbf{exc} \implies \neg(\ominus^u(\mathbf{call} \wedge \mathrm{p}_A) \vee \chi_P^u(\mathbf{call} \wedge \mathrm{p}_A)))$	0.622
$\square((\mathbf{call} \wedge \mathrm{p}_B \wedge (\mathbf{call} \, \mathcal{S}_\chi^d \, (\mathbf{call} \wedge \mathrm{p}_A))) \implies \bigcirc^u \mathbf{exc} \vee \chi_F^u \mathbf{exc}$	0.046
$\square(\mathbf{han} \implies \chi_F^u \mathbf{ret})$	0.152
$\top \, \mathcal{U}_\chi^u \, \mathbf{exc}$	0.215
$\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^u \, \mathbf{exc}))$	1.30
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^u \, \mathbf{exc})))$	3.79
$\square(\mathbf{call} \wedge \mathrm{p}_C \implies (\top \, \mathcal{U}_\chi^u \, \mathbf{exc} \wedge \chi_P^d \mathbf{han}))$	5.84
$\mathbf{call} \, \mathcal{U}_\chi^d \, (\mathbf{ret} \wedge \mathrm{p}_{err})$	0.014
$\chi_F^d(\mathbf{call} \wedge ((\mathbf{call} \vee \mathbf{exc}) \, \mathcal{S}_\chi^u \, \mathrm{p}_B))$	0.856
$\bigcirc^d(\bigcirc^d((\mathbf{call} \vee \mathbf{exc}) \, \mathcal{U}_\chi^u \, \mathbf{ret}))$	0.053
Total	3607
Maximum memory used (MB)	94,535.576

Table 4: Results of the additional experiments on OPA “generic medium”.

Formula	Time (s)
$\chi_F^d \mathrm{p}_{err}$	0.003
$\bigcirc^d(\bigcirc^d(\mathbf{call} \land \chi_F^u \mathbf{exc}))$	0.517
$\bigcirc^d(\mathbf{han} \land (\chi_F^d(\mathbf{exc} \land \chi_P^u \mathbf{call})))$	0.243
$\square(\mathbf{exc} \implies \chi_P^u \mathbf{call})$	0.054
$\top \, \mathcal{U}_\chi^d \, \mathbf{exc}$	0.008
$\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^d \, \mathbf{exc}))$	0.018
$\square((\mathbf{call} \land \mathrm{p}_A \land (\neg \mathbf{ret} \, \mathcal{U}_\chi^d \, \mathrm{WRx})) \implies \chi_F^u \mathbf{exc})$	728
$\bigcirc^d(\bigcirc^u \mathbf{call})$	0.001
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\ominus^u \mathbf{call})))$	0.001
$\chi_F^d(\bigcirc^d(\ominus^u \mathbf{call}))$	0.008
$\square((\mathbf{call} \land \mathrm{p}_A \land (\bigcirc^u \mathbf{exc} \lor \chi_F^u \mathbf{exc})) \implies (\bigcirc^u \mathrm{e}_B \lor \chi_F^u \mathrm{e}_B))$	1843
$\diamond(\bigcirc_H^d \mathrm{p}_B)$	0.016
$\diamond(\ominus_H^d \mathrm{p}_B)$	0.006
$\diamond(\mathrm{p}_A \land (\mathbf{call} \, \mathcal{U}_H^d \, \mathrm{p}_C))$	0.446
$\diamond(\mathrm{p}_C \land (\mathbf{call} \, \mathcal{S}_H^d \, \mathrm{p}_A))$	0.466
$\square((\mathrm{p}_C \land \chi_F^u \mathbf{exc}) \implies (\neg \mathrm{p}_A \, \mathcal{S}_H^d \, \mathrm{p}_B))$	1764
$\square(\mathbf{call} \land \mathrm{p}_B \implies \neg \mathrm{p}_C \, \mathcal{U}_H^u \, \mathrm{p}_{err})$	0.132
$\diamond(\bigcirc_H^u \mathrm{p}_{err})$	0.002
$\diamond(\ominus_H^u \mathrm{p}_{err})$	0.001
$\diamond(\mathrm{p}_A \land (\mathbf{call} \, \mathcal{U}_H^u \, \mathrm{p}_B))$	0.028
$\diamond(\mathrm{p}_B \land (\mathbf{call} \, \mathcal{S}_H^u \, \mathrm{p}_A))$	0.027
$\bigcirc^d \mathbf{call}$	0.001
$\bigcirc^d(\bigcirc^d \mathbf{call})$	0.002
$\bigcirc^d(\bigcirc^d(\bigcirc^d \mathbf{call}))$	0.001
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\bigcirc^d \mathbf{call})))$	0.008
$\square(\mathbf{call} \implies \chi_F^d \mathbf{ret})$	0.030
$\square(\mathbf{call} \implies \neg \bigcirc^u \mathbf{exc})$	0.003
$\square(\mathbf{call} \land \mathrm{p}_A \implies \neg(\bigcirc^u \mathbf{exc} \lor \chi_F^u \mathbf{exc}))$	0.743
$\square(\mathbf{exc} \implies \neg(\ominus^u(\mathbf{call} \land \mathrm{p}_A) \lor \chi_P^u(\mathbf{call} \land \mathrm{p}_A)))$	0.622
$\square((\mathbf{call} \land \mathrm{p}_B \land (\mathbf{call} \, \mathcal{S}_\chi^d \, (\mathbf{call} \land \mathrm{p}_A))) \implies \bigcirc^u \mathbf{exc} \lor \chi_F^u \mathbf{exc}$	31.9
$\square(\mathbf{han} \implies \chi_F^u \mathbf{ret})$	0.362
$\top \, \mathcal{U}_\chi^u \, \mathbf{exc}$	0.241
$\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^u \, \mathbf{exc}))$	1.39
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^u \, \mathbf{exc})))$	4.80
$\square(\mathbf{call} \land \mathrm{p}_C \implies (\top \, \mathcal{U}_\chi^u \, \mathbf{exc} \land \chi_P^d \mathbf{han}))$	5.78
$\mathbf{call} \, \mathcal{U}_\chi^d \, (\mathbf{ret} \land \mathrm{p}_{err})$	0.027
$\chi_F^d(\mathbf{call} \land ((\mathbf{call} \lor \mathbf{exc}) \, \mathcal{S}_\chi^u \, \mathrm{p}_B))$	0.728
$\bigcirc^d(\bigcirc^d((\mathbf{call} \lor \mathbf{exc}) \, \mathcal{U}_\chi^u \, \mathbf{ret}))$	1.85
Total	4385
Maximum memory used (MB)	60,318.568

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

RAM (the fact that the CPU has 16 cores is actually irrelevant, as POMC only runs on one for the time being).

If you wish to repeat such experiments, you may run the following commands:

```
$ cd ~/artifact/contents
$ python3 mcbench.py opa-more
```

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

5 Source Code

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

App.hs The application's main function. The main function parses the input files and calls the appropriate functionalities (such as the model checker) to fulfill their requests.

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.

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

Example.hs contains the OPM used in examples in various POTL papers, including $M_{\mathbf{call}}$ from Figure 1.

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).

Parse.hs contains a parser for POMC input files.

PotlV2.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.

RPotl.hs defines an intermediate representation for POTL formulas. All formulas must be translated into this representation before converting them to OPA with the functions from Check.hs.

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.

Util.hs contains various functions used in other parts of the code.

References

[1] Abrahams, D.: Exception-Safety in Generic Components. In: Generic Programming. pp. 69–79. Springer (2000). https://doi.org/10.1007/3-540-39953-4_6

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

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

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

[5] Chiari, M., Mandrioli, D., Pradella, M.: POTL: A first-order complete temporal logic for operator precedence languages. CoRR **abs/1910.09327** (2019), http://arxiv.org/abs/1910.09327

[6] Jensen, T., Le Metayer, D., Thorn, T.: Verification of control flow based security properties. In: Proc. '99 IEEE Symp. on Security and Privacy. pp. 89–103 (1999). https://doi.org/10.1109/SECPRI.1999.766902

[7] Sutter, H.: Exception-safe generic containers. C++ Report (1997), https://ptgmedia.pearsoncmg.com/imprint_downloads/informit/aw/meyerscddemo/DEMO/MAGAZINE/SU_FRAME.HTM