POMC User's Guide

Michele Chiari

Abstract

POMC is a model checker that models procedural programs as pushdown automata and checks them against formulas of Precedence-Oriented Temporal Logic (POTL). 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 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 [4]. 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 [2], 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.

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 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. 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 [6]. 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.

We used POMC to prove some interesting properties of programs which we modeled both as OPA and MiniProc programs. Such experiments are contained in `pomc` files in the `eval` subdirectory.

We show how to use POMC 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 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 packaged with the Haskell Tool Stack[1]. It requires the development files for the Z3 Theorem Prover library to be built (package `libz3-dev` on Debian-based systems). 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
```

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.,

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

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

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

	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	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

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

Figure 1

```
$ 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).

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 [5]. 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 [5] (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 $\square\big((\mathbf{call} \wedge \mathrm{p}_B \wedge \mathit{Scall}(\top, \mathrm{p}_A)) \implies \mathit{CallThr}(\top)\big)$, is explained in the paper.

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

```
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.

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:

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.

```
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 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

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

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();
  }
}

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

pc() { }
```

POMC prints the following:

	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 2: Results of the evaluation.

```
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 Some experiments

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.

4.1 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 tasks are reported in Table 2. 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

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

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. [7] 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

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

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. [8] 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:

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

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 `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 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 3 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.

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

Table 3: Results of the additional experiments on OPA “generic larger”.

4.3 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 4.1. 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 3.

Table 4 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.

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 4: Results of the evaluation of `miniproc` files.

5 Source Code

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

Parse This directory contains the parser for input files.

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 `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.

Acknowledgements

We are grateful to Davide Bergamaschi for developing an early prototype of this tool, and to Francesco Pontiggia for implementing the model checking algorithms for infinite words and performance optimizations.

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 and P. Madhusudan. Adding nesting structure to words. *JACM*, 56(3), 2009. doi:10.1145/1516512.1516518.

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

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

[4] 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.

[5] M. Chiari, D. Mandrioli, and M. 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.

[6] M. Chiari, L. Geatti, N. Gigante, and M. 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.

[7] 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.

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

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 $L_P = \mathit{BExp}_P \cup \mathit{Ass}_P$ 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 $\mathit{En}_s \subseteq Q_P \times L_P$. 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 $\mathit{En}_f = \mathit{En}_s$, 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 $q_s \xrightarrow{\mathbf{call}\ f\ l} q$ for all $(q, l) \in \mathit{En}_f$, 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 $(q, l) \in \mathit{En}_s$.

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 $En_s = \{(q^t_f, \top)\}$.

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 $q^e_f \xRightarrow{q_s\ l} q$ 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 $q_t \overset{\textbf{exc}\ \mathit{dummy}}{\dashrightarrow} q'_t$, 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 $q^r_{f_0} \xRightarrow{q_0} q_f$.

A.2 From extended OPA to OPA

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

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

where $Q_P \subseteq Q^E_P \times \{0,1\}^{|I_P|}$. Each state is a pair (q, v) with $q \in Q^E_P$ 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 $\mathrm{vars}(v)$ we denote the set of variables satisfied by $v \in \{0,1\}^{|I_P|}$. We define $Q_P := \bigcup_{i \in \mathbb{N}} Q^i_P$ inductively through the following equations:

$$Q^0_P := \{q_0\} \times \{0,1\}^{|I_P|}$$
$$Q^{n+1}_P := \{(q, v) \mid q' \in Q^n_P,\ (q', a\, l, q) \in \delta^E_P,\ v \models l\}$$

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

$$\begin{aligned}\delta_P :=& \{q \overset{a \ \ \text{vars(v)}}{\longrightarrow} q' \mid q \xrightarrow{a\ l} q' \in \delta_P^E,\ q, q' \in Q_P\} \\ & \cup \{q \overset{a \ \ \text{vars(v)}}{\dashrightarrow} q' \mid q \overset{a\ l}{\dashrightarrow} q' \in \delta_P^E,\ q, q' \in Q_P\} \\ & \cup \{q \overset{p}{\Longrightarrow} q' \mid q \overset{p\ l}{\Longrightarrow} q' \in \delta_P^E,\ q, q', p \in Q_P\}\end{aligned}$$

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.