# Better Predicates and Heuristics for Improved Commutativity Synthesis

Adam Chen[1][0000−0002−5159−6532]*, Parisa Fathololumi[1][0000−0002−8602−2862], Mihai Nicola[1][0000−0003−0204−1626], Jared Pincus[2][0000−0001−6708−5262], Tegan Brennan[1][0000−0002−9988−8630], and Eric Koskinen[1][0000−0001−7363−634X]

[1] Stevens Institute of Technology
{achen19,pfathol1,lnicola,tbrenna5,ekoskine}@stevens.edu
[2] Boston University pincus@bu.edu

**Abstract.** Code commutativity has increasingly many applications including proof methodologies for concurrency, reductions, automated parallelization, distributed systems and blockchain smart contracts. While there has been some work on automatically generating commutativity conditions through abstraction refinement, the performance of such refinement algorithms critically depends on (i) the universe of predicates and (ii) the choice of the next predicate during search, and thus far this has not been examined in detail.

In this paper, we improve commutativity synthesis by addressing these under-explored requirements. We prune the universe of predicates through a combination of better predicate generation, new *a priori* syntactic filtering, and through dynamic reduction of the search space. We also present new predicate selection heuristics: one based on look-ahead, and one that utilizes model counting to greedily cover the search space. Our work is embodied in the new commutativity synthesis tool Servois2, a generational improvement over the state-of-the-art tool Servois. Servois2 is implemented in a faster language and has support for CVC5 and Z3. We contribute new, non-trivial commutativity benchmarks. All of the new features in Servois2 are shown to either increase performance (geomean 3.58× speedup) or simplify the conditions generated, when compared against Servois. We also show that our look-ahead heuristic leads to better scaling with respect to the number of predicates.

## 1 Introduction

Commutativity of data structure methods and program code applies to a wide variety of contexts, ranging from proof methodologies for concurrency (*e.g.* Siever [1], CIVL [2], Anchor [3]) to exploiting multicore (*e.g.* parallelizing compilers [4], transactional memory [5] declarative programming [6,7], scalable systems [8]) to distributed systems (*e.g.* CRDTs [9] and blockchain [10,11]).

Accordingly, there have been a variety of techniques and tools for reasoning about commutativity, including program analysis [4], sampling [12], random

*Corresponding author.

interpretation [13], and abstract interpretation [11]. A recent workshop[‡] exemplifies the rising interest in commutativity.

In many contexts program code does not always commute, and it is therefore helpful to specify the *conditions* under which code commutes. In a hashtable, for example, inserting key $k$ commutes with removing key $k'$ only when $k \neq k'$. Bansal *et al.* [14,15] introduced an abstraction-refinement method for automatically synthesizing such commutativity conditions. The idea is to recursively test the logical space using an SMT solver, accumulating conditions that imply commutativity or non-commutativity in disjunctive normal form. While the authors provided a proof-of-concept abstraction-refinement algorithm, they did not explore deeper performance considerations including more aggressive search heuristics, semantic predicate treatment, model counting optimizations, scalability, etc.

This paper focuses on how commutativity condition refinement can be improved by addressing the key search parameters. By addressing these performance considerations, we improve speed and scalability, as well as quality of outputted conditions. We encapsulate our results in a new tool that is a generation improvement in synthesizing commutativity conditions.

**Contributions.** Our work improves the state-of-the-art in the following ways:

*1. Predicate semantics.* (Sec. 3) In the state of the art, predicates must be built by manually writing terms, and are then mildly filtered and used without any information as to how one predicate relates to another. Refinement is exponential in the number of predicates so it is important to focus on important predicates. To that end, we improve the treatment of predicates by both syntactically and then semantically filtering redundant predicates. We next show how the information from filtering can be used to construct a lattice of predicates, ordered by implication, and use this lattice to better filter predicates and prune the state space during search. We also automatically extract terms from the input problem's pre/post relations.

*2. Search heuristics.* (Sec. 4) A key step in the algorithm is choosing the next predicate to divide the search space. We implement two new heuristics:

- `poke2`: A new predicate selection heuristic which avoids redundant SMT work, while also using the information obtained more directly. Consequently, it performs at most half as many SMT queries, if not fewer, than Servois's original implementation.
- `mcMax`: A heuristic that employs model counting to more quickly cover the search space. Model counting is the problem of computing the number of models (*i.e.* distinct assignments to variables) that satisfy a given predicate [16]. As many predicates have infinitely many solutions, model-counting constraint solvers return the number of solutions for a given predicate within a given bound [17,18]. The `mcMax` heuristic takes a quantitative approach to predicate selection by leveraging model-counting to greedily pick predicates based on the largest covering of the state space, making choices based on

[‡] `https://pldi22.sigplan.org/home/cora-2022`

approximate finite-domain information, yet maintaining soundness of the overall infinite-domain algorithm.

*3. New implementation and pragmatic concerns.* (Sec. 5) We implemented SERVOIS2 in OCaml, exploiting the expected performance benefits of OCaml over Python. Our implementation is parametric on SMT solver, now supporting CVC4, CVC5 and Z3. We therefore inherit the expanded theory support, expanding the domains in which commute conditions can be synthesized. SERVOIS2 can now, for example, synthesize commute conditions for Strings operations like `hasChar` and `concat`. We also support interruption, emitting a sound but incomplete condition, allowing SERVOIS2 to be used in a larger variety of new settings. Finally, SERVOIS2 has a more well-defined API (as an OCaml type), allowing one to use it as a library. SERVOIS2 is publicly available at: `github.com/veracity-lang/servois2`. The artifact, which contains a copy of the code, is available at: `https://www.doi.org/10.5281/zenodo.7935263`.

*4. Evaluation.* (Sec. 6) In order to show that our approaches improve performance in practice, we introduced new, non-trivial benchmarks that Bansal *et al.* [14]'s tool SERVOIS struggles to solve. We evaluated all of SERVOIS2's new approaches, including the `poke2` and `mcMax` heuristics, in comparison to a faithful re-implementation of the `poke` heuristic in our new OCaml implementation on both the original benchmarks and our new ones. We also compared the performance improvements between heuristics with additional options for tuning the synthesis (see Sec. 6). Our experiments demonstrate that our approaches do give a substantial speedup—3.58× (geometric mean) faster. In cases that involve theories where model counting can be done efficiently (strings, linear integer arithmetic, integer arrays), `mcMax` is often able to offer better performance. Furthermore, `poke2` scales approximately linearly with the number of state variables, while the other heuristics (including all those in the prior work) diverge after only a few variables. Finally, given this wide variety of options, we used a portfolio approach, running each case with all options (solvers, heuristics, approaches to terms, etc.) and reporting back the first one to finish.

## 2 Background: Commutativity synthesis

We begin with a brief review of abstraction-refinement commutativity condition synthesis, emphasizing key steps.

Suppose we have an abstract data type (ADT) method call $m(\bar{a})/r_m$, with method name $m$, taking argument vector $\bar{a}$ and returning value $r_m$. Similarly, consider a second method call $n(\bar{b})/r_n$. We say these method calls *commute* from an initial ADT state $\sigma$, provided that when methods are applied in either order, they lead to the same final ADT state, and will have observed the same return values along the way. We notate this $m \bowtie_{\sigma,\bar{a},\bar{b},r_m,r_n} n$, with subscripts omitted when the context is clear. A commutativity *condition* is a logical formula $\varphi_m^n(\sigma, \bar{a}, \bar{b})$ describing the conditions on the initial ADT state $\sigma$ (and parameters $\bar{a}$ and $\bar{b}$) under which $m$ and $n$ always commute. (A non-commutativity condition $\tilde{\varphi}$ describes conditions when they always do not commute.) As an example, a

```
REFINE^m_n(H, P){
  if valid(H ⇒ m ⋈̂ n) then
    φ := φ ∨ H;
  else if valid(H ⇒ m ⋈̸̂ n) then
    φ̃ := φ̃ ∨ H;
  else
    let χc = counterexs. to ⋈̂
        χnc = counterexs. to ⋈̸̂ in
    let p = CHOOSE(H, P, χc, χnc) in
    REFINE^m_n(H ∧ p, P \ {p});
    REFINE^m_n(H ∧ ¬p, P \ {p});
}
```

```
main (spec, terms){
  φ := false;  φ̃ := false;
  let P = BUILD(terms)in
  try {REFINE^m_n(true, P); }
  catch (Interrupt e) {skip;}
  return(φ, φ̃); }
```

Fig. 1: The commutativity condition REFINE algorithm [14].

Set ADT with methods `insert`$(x)$ and `remove`$(y)$, a sufficient commutativity condition would be $\varphi^{\texttt{remove}}_{\texttt{insert}} \equiv x \neq y$.

We synthesize a commutativity condition $\varphi$ via the REFINE algorithm[14], which takes as input, an ADT specification, with methods' pre/post conditions written in SMTLIB. The algorithm uses the binary operator $\hat{\bowtie}$, which is defined as $\bowtie$ on a lifted (total) version of the ADT; we omit the full details as they are not relevant to our improvements on the work. When run on a given pair of methods $m$ and $n$, the output of the algorithm is a pair $(\varphi^n_m, \tilde{\varphi}^n_m)$ of commutativity/non-commutativity conditions. Consider as an example input, an ADT for a hashtable that has three variables representing the state: a `size` integer, a Set over sort $E$ of `keys`, and a finite array `H` mapping elements of sort $E$ to sort $F$. Then, for each method, *e.g.*, `put`$(k, v)$, the input ADT specification includes a pre-condition (in this case true) and a post-condition relating the pre-state with input vector (`size`, `keys`, `H`, $k$, $v$) to a tuple of new values with return value (in this case, true or false representing success) (`size_new`, `keys_new`, `H_new`, $r$). The REFINE algorithm as output synthesizes commutativity conditions for the input method pair. In the case of the hashtable example, the solution for the commutativity synthesis of two calls of the same method `put`$(k_1, v_1)$ and `put`$(k_2, v_2)$ generated by the algorithm is $\varphi \equiv (v_1 = v_2 \wedge \texttt{H}[k_1] = v_2 \wedge k_1 \in \texttt{H}) \vee \ldots$ (truncated). Other cases (disjuncts) omitted for brevity.

The REFINE algorithm is presented in Fig. 1. The algorithm recursively partitions the logical space along conjunctions of predicates, which are selected from a set of predicates $\mathcal{P}$. When the algorithm finds a region of the state space $H$ that is a sufficient condition for commutativity (or *mutatis mutandis* non-commutativity), it adds it to an accumulated DNF logical commutativity condition. Otherwise, the recursive calls use counterexamples to select a predicate $p$ that differentiates the two counterexamples $\chi_c$ and $\chi_{nc}$. This predicate is conjunctively added to $H$ and used in the children recursive calls, and similarly for its negation. Fig. 2 illustrates this process of partitioning the logical space through the use of differentiating counterexamples.

This process continues until a necessary and sufficient commutativity condition is found, or all combinations of predicates are exhausted. Typically, exhaustion of predicates is unlikely as there are exponentially many combinations of them. Furthermore, the algorithm can theoretically be interrupted (*e.g.* after a timeout) to yield a sound commutativity condition.

While the REFINE algorithm is a somewhat straightforward form of abstraction-refinement, the effectiveness of the technique and implementation thereof critically depends on how predicates are handled, selected, pruned, etc. We now discuss these details and how SERVOIS2 improves on each of them.

## 3 Semantic Treatment of Predicates

REFINE has worst-case exponential runtime in the number of predicates. While a CHOOSE function that picks good predicates helps, it is still important to generate a small set $\mathcal{P}$ of relevant predicates and be selective during each recursive call. At the very least, reducing the number of predicates gives linear improvements on runtime, as SMT solvers, as well as CHOOSE, must handle every predicate in $\mathcal{P}$. In this section, we describe better methods of reducing the size of this set $\mathcal{P}$.

*(a) Improved predicate filtering.* In SERVOIS, after the initial list of predicates was built from manually provided terms, the SMT solver was queried twice for each predicate, and any predicate that was tautologically true or false was discarded. We retain this functionality, but first perform an additional syntactic layer of filtering by dropping any predicate that is:

1. A reflexive operation on two identical terms,
2. An operation between two constants, or
3. A symmetric case of another predicate already included.

Since all of these filters are done purely syntactically, we save SMT work.

*(b) Pruning by exploiting implication.* We next determine which predicates imply other predicates. This can be done via syntactic implication rules such as $x > y \Rightarrow x+n > y+n$. As a benefit, we are able to compute the closure of the logical implication relation, and are able to sort predicates into equivalence classes. Thus by removing redundant predicates, the size of the set of predicates can be reduced.

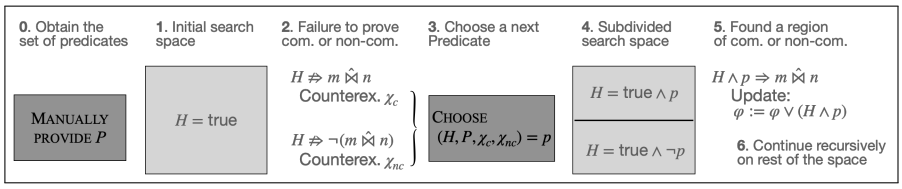



Fig. 2: REFINE recursively divides the logical search space using $\mathcal{P}$.

Given logical implication relations, we can build a lattice out of the partially ordered set of predicates, ordered by the $\Rightarrow$ relation. This lattice information can be used to dynamically prune predicates that become redundant during runtime due to selection of other, related predicates. For example, consider the following LIA benchmark `multiVarA` $\bowtie$ `multiVarB`. (Technically SERVOIS/SERVOIS2 inputs are given as ADT pre/post specifications, but we write this example as code illustration purposes.)

```
int x, y;
bool multiVarA() {if(x>0) { x = 2*x + y; }; return true}
bool multiVarB() {if(x>y) { x = x - 2*y; } else { x = x - y; }; return true}
```

Here, the lattice identifies implication chains such as $0 > (2x + y) \Rightarrow (2x + y) \leq 0 \Rightarrow 2 > (2x + y) \Rightarrow (2x + y) \leq 2$, or $0 > (2x + y) \Rightarrow 0 \neq (2x + y)$. During search, we are able to use these chains to efficiently pruning the predicate lattice, effectively reducing the height of the lattice and thus width of the search tree.

Fig. 3 illustrates our modifications to the original algorithm in Fig. 1. Starting from `main`, we perform automated predicate generation PREDGEN, which we will discuss below. From this set $\mathcal{P}$, we construct the lattice $\mathcal{L}$ with MKLAT. Within REFINE, we parameterize CHOOSE by $\mathcal{L}$, allowing the CHOOSE heuristics discussed in Sec. 4 to make choices based on $\mathcal{L}$. Finally, we prune the search space by using RMUPPER to remove all predicates that are weaker, *i.e.* higher in the lattice, than the selected predicate pair $(p, \neg p)$. We may do similarly with the predicates stronger than the negation (RMLOWER). As a result, recursive calls will not have to consider any predicates that are already entailed by $H$.

Constructing the lattice can be costly as the size of the relation is quadratic in the number of predicates. Furthermore, syntactic rules cannot discover all implications, so an SMT solver must be invoked if more precision is desired. As we will see, it is not always worth this overhead. We have thus kept our lattice treatment as an optional feature. When disabled, the lattice simply behaves as a set of predicates (*i.e.* any predicate is only related to itself), à la Fig. 1, and assume that the set of predicates is closed under negation[§].

*(c) Automatic predicate generation via term extraction.* SERVOIS requires the programmer to manually provide *terms* with each method in the ADT specification, which can be error-prone and tedious. These terms are then used to build predicates by using boolean relations such as $=, >$, etc. We are able to automatically generate the terms for synthesizing the predicates by traversing the method specification (state variables, method arguments, pre/post-condition), and extracting basic expressions (categorized by type). The expressions are then combined with predefined operations for each type (*e.g.* in-/equality for Integers, membership/subset/etc. for Sets, contains/prefix/etc. for Strings, ...) to generate the predicates. With this approach, we generate enough predicates to establish a sufficiently granular search space across all of our benchmarks and

[§]To satisfy closure, we include negations of all predicates. This comes at no performance loss, as such additions can be skipped over by CHOOSE. This is valid because REFINE recurses upon the negation of the chosen predicate.

```
REFINE^m_n(H, L){
  if valid(H ⇒ m ⋈̂ n) then
    φ := φ ∨ H;
  else if valid(H ⇒ m ⋈̸̂ n) then
    φ̃ := φ̃ ∨ H;
  else
    let χc = counterexs. to ⋈̂
        χnc = counterexs. to ⋈̸̂ in
    let p = CHOOSE(H, P, χc, χnc) in
    let L' = L \ {p, ¬p} in
    REFINE^m_n(H ∧ p, RMLOWER(
               RMUPPER(L', p), ¬p));
    REFINE^m_n(H ∧ ¬p, RMLOWER(
               RMUPPER(L', ¬p), p));
}

main (spec){
  φ := false;  φ̃ := false;
  let P = PREDGEN(spec) in
  try {REFINE^m_n(true, MKLAT(P)); }
  catch (Interrupt e) {skip;}
  return (φ, φ̃);
}
RMUPPER(L, p){
  return L \ {p' | p ⇒ p' ∧ p ≠ p'};
}
RMLOWER(L, p){
  return L \ {p' | p' ⇒ p ∧ p ≠ p'};
}
```

Fig. 3: Our modified algorithm. Fig. 1 is recovered when taking $\mathcal{L}$ to be the trivial lattice over negation completion.

we are not limited in how exhaustively terms are provided. Manual-vs-automatic term extraction leads to different sets and quantities of predicates, which may affect how conditions are expressed. In Sec. 6 we discuss the performance impact.

*(d) Syntax-based generation of predicates.* Once predicates are automatically generated, it is natural to consider whether more complex predicates can be generated. While the original tool only considered predicates on two given terms, we found that often, compound terms that may not be provided or directly in the specification's syntax would be present in commutativity conditions. We added the expansion of terms with known and provided functions to allow for the automated generation of compound terms and predicates. Thus more complex commutativity conditions could be expressed, and the user does not have to already have specific predicates in mind when listing terms. Due to the exponential nature of syntax expansion, we get a greatly increased number of predicates. We found that this increase was too detrimental to performance to be practical—two or more iterations often times out. However there were still some test cases that benefited from performing one or two iterations, and the approach would likely be beneficial with improved pruning.

## 4 Search Heuristics

At each step, the REFINE algorithm must CHOOSE a predicate that differentiates the commutative and non-commutative examples. While any implementation of CHOOSE maintains the soundness of REFINE, due to the exponential nature of the number of subsets of predicates, choosing a "good" predicate is important both to efficiency and quality of the form of the emitted condition. We refer to a CHOOSE strategy as a "heuristic". Bansal *et al.* [14] describe a heuristic—referred to as poke—which performs a greedy one-step look-ahead.

```
(1)  CHOOSE_poke2(H, P, χc, χnc){
(2)    let P' = DiffingPreds(P, χc, χnc)in
(3)    let weight(p) =
(4)      let p' = if([[p]]_χc) then p else !p in
(5)      if valid(H ∧ p' ⇒ m ⋈̂ n) then
(6)        return 0;
(7)      else if valid(H∧ !p' ⇒ m ⋈̸̂ n) then
(8)        return 0;
(9)      else
(10)       letχ'c = counterexs. to ⋈̂ in
(11)       let χ'nc = counterexs. to ⋈̸̂ in
(12)       return Length(
(13)            DiffingPreds(P', χ'c, χ'nc))
(14)   in list_min(weight, P')
(15) }
(16) DiffingPreds(P, χc, χnc){
(17)   return filter((fun p → [[p]]_χc ≠ [[p]]_χnc), P);
(18) }
```

Fig. 4: Pseudocode for our poke2 heuristic for choosing which predicate to recurse upon. Here, $\mathcal{P}$ may be obtained from $\mathcal{L}$ by taking the underlying set of predicates. `list_min`$(f, \mathcal{P})$ returns the element of $\mathcal{P}$ that minimizes $f$.

In this section we introduce two new predicate selection heuristics called CHOOSE$_{\texttt{poke2}}$ and CHOOSE$_{\texttt{mcMax}}$ (or simply, poke2 and mcMax) that, as we show in Sec. 6, perform better than the CHOOSE$_{\texttt{poke}}$ of Bansal *et al.* [14], with trade-offs to consider between the two of them.

### 4.1 The poke2 heuristic

We begin by formalizing the poke2 heuristic, and compare it to the previous poke heuristic. When the SMT solver is invoked with the "valid()" queries in REFINE, we obtain two satisfying counterexamples: $\chi_c$ for commutativity and $\chi_{nc}$ for non-commutativity. poke and poke2 share the common behavior to then proceed with two steps: (1) Test each predicate to see which hold in which counterexample (if either); this can be done in the same SMT query that was used for valid. This testing lets one find predicates that differ between the commutative counterexample and the non-commutative counterexample. This is summarized in *DiffingPreds*; the pseudocode for this subroutine is given at the bottom of Fig. 4. (2) Next, perform a partial look-ahead on each of these predicates—however, the way this is done differs between the heuristics.

– **The poke2 heuristic.** The full pseudocode for poke2 is given in Fig. 4. The partial look-ahead is encapsulated in the *weight* function. If a predicate was true in the commutative case then we can tentatively conjoin it with the commutativity condition and its inverse with the non-commutativity condition (*mutatis mutandis* for false—keeping track of which case is done on Line 4), then query the solver (Lines 5 and 7) to see how many predicates still

```
(1) CHOOSE_mcMax(H, P, χc, χnc){
(2)  let P' = DiffingPreds(P, χc, χnc) in
(3)  let cover(p) = #([[p]]) / #(Σ)
(4)      cover(¬p) = 1 − cover(p) in
(5)  let P'' = list_max(cover, P') in
(6)  return first(P'')
(7) }
```

(a) mcMax

```
(1) CHOOSE_mcMax-poke2(H, P, χc, χnc)
(2)  let P' = DiffingPreds(P, χc, χnc) in
(3)  let cover(p) = #([[p]]) / #(Σ)
(4)      cover(¬p) = 1 − cover(p) in
(5)  let P'' = list_max(cover, P') in
(6)  return list_min(weight, P'')
(7) }
```

(b) mcMax-poke2

Fig. 5: Pseudocode for mcMax heuristics. As before, $\mathcal{P}$ is obtained by taking the underlying set of $\mathcal{L}$.

differentiate the two cases. We define the number of remaining differentiating predicates to be the weight of the predicate (Lines 9-11). Finally we pick the differentiating predicate that results in the fewest remaining differentiating predicates in the look-ahead (Line 12). In the case that two predicates have the same number of new differentiating predicates, we prefer the simpler (measured in number of atoms) one (not shown).

– **The poke heuristic.** By contrast, in poke the predicate and its inverse were tested with both the commutative case and the non-commutative case, irrespective of whether it was true or false in the commutative case. This resulted in many degenerate return values, which not only increased SMT time, but also could pick less beneficial predicates.

There is no need to prove correctness of poke2, as the REFINE algorithm is correct for any implementation of CHOOSE that picks a differentiating predicate. Our evaluation of poke2 is thus based on runtime (more detail in Section 6.1).

### 4.2 The mcMax heuristic

For theories where model counting is efficiently supported by existing tools, we introduce an additional heuristic called mcMax. mcMax uses model counting to determine the number of satisfying solutions for each constraint on the state space. It then uses this count to quantify how well each predicate covers the state space and picks the predicate with the best coverage.

Model counting requires a finite domain, so we treat state variables as finite on a bounded domain, *e.g.*, treating integers as fixed-length bit vectors. Recall that any implementation of CHOOSE is sound, so bounding the domain (temporarily as a heuristic hint) does not threaten soundness. For such fixed-length bit representations, we ideally require a bit width bound that is large enough to properly differentiate between coverage ratios. Experimentally, we found that a bound as low as 4-bit representation of integers was sufficient for LIA constraints with relatively small coefficients and a length of 4 was sufficient for string constraints.

We now describe how mcMax proceeds using the pseudocode given in Fig. 5. The mcMax heuristic starts off on Line 2 in a similar manner as poke heuristic by

constructing the subset of differentiating predicates $\mathcal{P}'$ from the two satisfying counterexamples. In the next step (Lines 3-4), we calculate the coverage ratio for both $p$ and its complement $\neg p$ as the fraction of their corresponding models' count. Finally, the predicate found to represent the largest state region is chosen (Line 5). Recall that REFINE traverses both the given predicate *and* its negation (shown in the recursive calls in Fig. 1). In the case that execution is interrupted, we observe the first recursive call may be explored, while the second is not. In these cases `mcMax` often leads to a better (higher coverage ratio) predicate compared to a non-model-counting heuristic, since we greedily pick the larger conjuncts.

To overcome the arbitrary first choice among equally covering predicates at line 6 in Fig.5a, the variant `mcMax-poke2` equips `mcMax` with the weight-based ranking of predicates from `poke2`. Whenever the list of maximal coverage predicates returned by `mcMax` has at least two candidates, the predicate selection is turned over to `poke2` applied to the candidate list.

## 5 Implementation

SERVOIS2 is implemented in OCaml and is publicly released under the MIT License[¶]. The tool has an underlying representation for SMT expressions, and parses input YAML files and the SMTLIB2 expressions within them. Examples of the SERVOIS/SERVOIS2 input format are available in the repository. The output commutativity condition is also an SMTLIB2 expression, but may be further constrained: since it is always in disjunctive normal form, and we add one conjunct at a time, we may model disjunctive normal form as a list of conjuncts, which are in turn lists of atoms. The lattice is implemented as a module parameterized by any module exposing an ordering relation, and is encoded as a graph with vertices stored in a map and two edge sets: that of covering elements and that of elements covered by it.

*Model counting.* For counting the solutions satisfied by each predicate, we use the state-of-the-art model-counting constraint solver ABC [18] that, among other strengths, allows for passing the specific domain bound along with the model-counting query. ABC supports precise solving of model-counting queries over strings, booleans, and linear integer arithmetic. We memorize the counting results in an association list to reuse them in subsequent calls of $\text{CHOOSE}_{\texttt{mcMax}}$.

*Model counting for integer arrays.* We expand the applicability of `mcMax` heuristics to predicates over array terms by adopting a method similar to the state-of-the-art model counter for bounded array constraints MCBAT [19]. This approach involves applying a sequence of model-count preserving reductions from the theory of arrays to the theory of uninterpreted functions and linear integer arithmetic before dispatching the query to ABC. While MCBAT focuses on formulas that are universally quantified over index variables, our procedure below addresses quantifier-free array constraints with terms $a[i]$ representing the value stored in the array $a$ at index $i$.
Consider for example, the problem of counting the solutions $\langle x, i, j\rangle$ satisfying

[¶] `https://github.com/veracity-lang/servois2`

the predicate $(x[i] \geq x[j] - 1)$, where $i, j$ are integer variables and $x$ represents arrays of size 4. We accomplish the task in three stages. First, we translate the predicate into a list of linear integer arithmetic constraints that are conjoined into a formula. Then we count the number of satisfying solutions $\langle x_i, x_j, i, j \rangle$ by running ABC on this query:

$$(i \geq 0) \wedge (i < 4) \wedge (j \geq 0) \wedge (j < 4) \wedge (i = j \Rightarrow x_i = x_j) \wedge (x_i \geq x_j - 1)$$

Finally, we obtain the total model count by multiplying the translated query result with the value domain size twice, once for each of the unaccounted and implicitly unconstrained array values.

The reductions below summarize the steps of our model counting procedure for formulas with integer array constraints:

1. Replace all compound array index expressions $e$ with fresh variables $i$ and add corresponding constraints of the form $e = i$. Perform the replacement from the outermost expression inwardly. Consider, for example, the term $x[k + j - 2] > 3$ occuring in the query. We first replace the access term $k + j - 2$ by a fresh variable $i$, and then introduce an additional constraint $i = k + j - 2$ which captures this replacement.
2. Add array bounds constraints for each array index variable $i$.
3. Perform Ackermann's reduction:
   - Replace all occurrences of array index terms $a[i]$ with fresh variables $a_i$, keeping track of the replaced mappings for each array variable $a$.
   - Add functional consistency constraints for each array variable and each pair of array index terms occurring in the query, i.e. $(i = j) \Rightarrow (a_i = a_j)$.
4. Dispatch the set of constraints to ABC and obtain the model-count $\#mc_{tr}$. Thus far, there are only minimal differences to the approach in [19].
5. Identify the unaccounted mappings for each array variable and compute the partial model-count by considering their summation and the unconstrained value domain: $\#mc_{unacc} = |\mathbb{Z}|^{unacc}$.
6. Obtain final model count as $\#mc = (\#mc_{tr} \ * \ \#mc_{unacc})$.

*Additional solvers & theories.* SERVOIS was hardcoded to work with CVC4 [20]. We have parameterized SERVOIS2 by SMT solver via OCaml modules and extended support for CVC5 [21] and for Z3 [22]. While mostly an implementation detail, this does allow us to leverage the additional strength of the other solvers. For example, CVC4 (as of version 1.8) did not have good support for modulus and division. Both CVC5 and Z3 are able to support such operations, and SERVOIS2 is able to generate commutativity/non-commutativity conditions for modular arithmetic examples.

With expanded solver support, SERVOIS2 can tackle more theories, including ones for which specialized solvers are useful. Neither *bit-vectors* nor *strings* were supported in the original release of SERVOIS, but SERVOIS2 can synthesize commutativity conditions for both theories. As an example, we showed that SERVOIS2 is capable of inferring that bit-vector negation always commutes with

itself. We also benchmarked a few string examples, such as substr $\bowtie$ hasChar, as they also demonstrate the usefulness of model counting.

*Early termination.* The following theorem is presented in Bansal *et al.* [14]:

**Theorem 1.** *For each* $\textsc{Refine}_n^m$ *iteration:* $\varphi \Rightarrow m \mathbin{\hat{\bowtie}} n$*, and* $\tilde{\varphi} \Rightarrow m \mathbin{\hat{\not\bowtie}} n$*.*

Thus, if updates to $\varphi$ and $\tilde{\varphi}$ are atomic, then terminating the algorithm at any point will yield valid conditions. We take advantage of this in Servois2 by allowing timeouts: the algorithm gracefully terminates after a designated time by outputting the incomplete (yet valid) conditions $\varphi, \tilde{\varphi}$.

This proves useful in practice, as not all commutativity conditions may be expressible in terms of the predicates available; a necessary and sufficient condition for synthesis of a complete commutativity condition via the Refine algorithm is given in Bansal *et al.* [14]. In such cases, the algorithm must finish its exponential run-time, only to determine that no complete commutativity condition is expressible. Even if the algorithm does terminate, after a certain point, the commutativity condition may be more complex than is useful. Thus it is usually more useful to cut the execution short and report only the most important few disjuncts of the commutativity conditions. In Sec. 6.2 we describe an instance of both a case where the algorithm does not terminate and a case where the algorithm terminates, but we still may obtain a reasonable condition by limiting the execution time.

## 6 Evaluation

We evaluated whether Servois2 improved over the state-of-the-art Servois in terms of performance (speed) and expressivity. All experiments below were run on a machine with an AMD EPYC 7452 32-Core CPU, 128GB RAM, Ubuntu 20.04, and OCaml 4.14.0.

*Benchmarks.* Our suite of 68 benchmarks begins with those used to evaluate Servois in the prior work [14]. Since the core goal of our work is to improve performance, we have pruned down this set, removing those benchmarks for which all tested heuristics can synthesize a condition after zero or one iteration(s). For example, we omitted the counter and accumulator examples because the conditions generated were either true/false or a single atom. We also removed all similar method pairs with simple commutativity conditions from the remaining data structures: sets, hashtables (HT), and stacks (Sta).

In addition to these benchmarks, we contribute new benchmarks for strings (Str) and linear integer arithmetic calculations (LIA), and a benchmark based on rigid motions on hexagons (DiH, for "dihedral"). These serve to show the application of model counting, which works best on these domains. The model counter is not applicable to the other data sets due to presence of custom data declarations. It could also be run on the counter and accumulator benchmarks, but we do not expect that to be illustrative due to triviality.

Moreover, we used Veracity[‖] project [6] benchmarks as additional nontrivial benchmarks. There are 26 reported benchmarks in Veracity that use commutativity synthesis. We have also implemented some new benchmarks, *e.g.* Solidity examples translated to Veracity, to demonstrate various aspects of our improvements in addition to more speedup. We elaborate on the usage of Veracity benchmarks in Sec. 6.2.

### 6.1 Performance results

We would ideally compare the performance of Servois2 versus Servois, but since Servois2 is written in OCaml, and Servois is written in Python, there is an obvious speedup from compilation, and indeed we found Servois2 to be at least twice as fast even using the same heuristics and on the same inputs. (As an example: the Hashtable put/put example was the slowest running benchmark—it took 5.31s with Servois using `poke`, and 2.61s with Servois2 using the same heuristic.)

However, our work is not aimed at comparing Python vs OCaml, so we instead benchmark across our new heuristics (`poke2` and `mcMax`) and features in comparison to a faithful re-implementation of Servois's `poke` in OCaml. The re-implementation was created by manually translating the source code of Servois.

**Comparison to `poke` baseline.** To test the variety of features we have added, we ran each benchmark with all combinations of features:

- The heuristics `poke`, `poke2`, and `mcMax`/`mcMax-poke2` (when applicable).
- With each of the cvc4, cvc5, and z3 solvers.
- With and without automatic term extraction (Sec. 3).

We report the configuration with the best performance in Table 1[**], using `poke` with cvc4 and no lattice, no term extraction as a reference point for comparison. The heuristic and solver is given, then whether term extraction was performed (notated TG). The geometric mean of the speedup ratio of the best configuration over `poke` was 3.58×. Note that this speedup is conservative, as the several benchmarks that timed out with `poke` (and did not with Servois2) are excluded. We also report the change in the complexity of the synthesized commutativity condition in $\Delta$A, indicating the change in the number of atoms in the synthesized condition. The full generated conditions are omitted; note that if synthesis terminates with a complete condition, the generated conditions will be logically equivalent, but sometimes the order of the terms changed. (†) indicates the cases where the tool terminated with an incomplete condition. We terminated the benchmarks at 120s, and indicate the ones that still did not finish within this time with **T**. A few benchmarks could not be run under cvc4, and those are marked with ☢. All benchmarks whose `poke` baseline took less than 1 second to execute were omitted from the table due to triviality.

[‖] `http://www.veracity-lang.org`

[**] mVarA and mVarB are short for multiVarA and multiVarB

| **Benchmark** | poke (s) | Srv2 (s) | **Spdup** | **$\Delta$A** | **Best Configuration** |
|---|---|---|---|---|---|
| DiH: motion ⋈ motion | ☢ | 4.71 | n/a | n/a | mcmax, z3 |
| HT: put ⋈ put | 2.22 | 1.28 | 1.73× | 0 | poke2, cvc4 |
| LIA: mVarA ⋈ mVarB | 16.23 | 3.48 | 4.66× | 0 | poke2, cvc5 |
| LIA: sum ⋈ multiVarSum | † 8.45 | † 8.36 | 1.01× | 0 | poke, cvc4 |
| LIA: sum ⋈ posSum | 4.38 | 0.71 | 6.21× | 0 | poke2, cvc4 |
| Str2: set ⋈ concat | † 9.89 | † 6.66 | 1.48× | -1 | poke2, z3, TG |
| Str3: read ⋈ write | † 5.31 | † 0.86 | 6.21× | 43 | poke, cvc5 |
| Str: hasChar ⋈ concat | 2.93 | 0.57 | 5.14× | 0 | mcmax, cvc5, TG |
| Str: substr ⋈ hasChar | 1.75 | 0.74 | 2.35× | 0 | mcmaxpoke2, cvc4 |
| Vcy: array-disjoint | 1.16 | 0.46 | 2.52× | 0 | poke2, cvc4, TG |
| Vcy: array1 | 1.51 | 0.51 | 2.96× | -2 | mcmax, cvc4 |
| Vcy: array2 | 2.91 | 0.98 | 2.97× | 0 | poke2, cvc4 |
| Vcy: array3 | 1.92 | 0.53 | 3.62× | 0 | poke2, z3 |
| Vcy: auction3 | 76.66 | 23.17 | 3.31× | 8 | poke2, cvc4 |
| Vcy: auction4 | 1.79 | 0.34 | 5.26× | 0 | poke2, z3, TG |
| Vcy: dict | 13.37 | 2.82 | 4.74× | 0 | poke2, cvc5 |
| Vcy: even-odd | ☢ | † 1.02 | n/a | n/a | poke2, cvc5 |
| Vcy: ht-add-put | 7.63 | 3.37 | 2.26× | 0 | poke2, cvc4 |
| Vcy: ht-cond-mem-get | † 1.28 | † 1.19 | 1.08× | -2 | poke2, cvc5 |
| Vcy: ht-cond-size-get | 1.66 | 0.71 | 2.34× | 0 | poke2, cvc4 |
| Vcy: ht-simple | 38.84 | 18.78 | 2.07× | 4 | poke2, cvc4 |
| Vcy: linear-bool | 3.15 | 0.90 | 3.50× | -2 | mcmax, cvc4 |
| Vcy: linear-cond | 2.09 | 1.30 | 1.61× | -1 | poke2, z3 |
| Vcy: loop-amt | † 11.35 | † 0.30 | 37.83× | 10 | mcmax, z3 |
| Vcy: loop-inter | 7.83 | 2.40 | 3.26× | -21 | mcmax, cvc5 |
| Vcy: matrix | 3.17 | 0.24 | 13.21× | 0 | poke2, z3 |
| Vcy: nested-counter_1 | 1.12 | 0.51 | 2.20× | 0 | poke2, cvc4 |
| Vcy: nested-counter_2 | † 4.74 | † 1.42 | 3.34× | -19 | mcmax, cvc5 |
| Vcy: nonlinear | 7.44 | 0.59 | 12.61× | 0 | poke2, z3 |
| Vcy: pullPayment | 7.69 | 1.59 | 4.84× | 0 | poke2, z3 |
| Vcy: simple | † 7.69 | † 2.29 | 3.36× | 26 | poke2, cvc4 |
| Vcy: standardToken2 | **T** | 11.31 | n/a | n/a | poke2, z3, TG |
| Vcy: standardToken3 | **T** | 1.45 | n/a | n/a | poke2, z3, TG |
| Vcy: standardToken4 | 2.34 | 0.35 | 6.69× | 0 | poke2, z3 |
| Vcy: standardToken5 | **T** | 13.25 | n/a | n/a | poke2, z3 |

Table 1: Total number of benchmarks: 68 (33 trivial ones omitted)
Benchmarks that previously timed out: 3
Benchmarks that previously crashed: 2
† means condition generated was incomplete. **$\Delta$A** is change in atom count.
**T** indicates time out (set at 120s). ☢ indicates cannot be run.

The mcMax heuristic only applies to ADTs with theories supported by the model counter ABC[23] extended with our procedure in Sec. 5, hence our results using that heuristic are limited to the String, LIA, and Dihedral ADTs, as well as the Veracity benchmarks. Our extension for integer arrays allowed for the use of mcMax on the majority of the Veracity benchmarks. In some cases, mcMax provides a significant speedup over poke and even poke2. For example, in the hasChar ⋈ concat benchmark, mcMax is 2.89× as fast as poke2 (not shown) and over 5.41× as fast as poke, with the same configuration aside from heuristic. In other cases, such as in the sum ⋈ multiVarSum analysis, mcMax underperforms

| Benchmark | Best Non-Latt. | Best Latt. | Spdup | $\Delta$A | Best Config. | Latt Cnstr. |
|---|---|---|---|---|---|---|
| DiH: motion $\bowtie$ motion | 4.71 | n/a | n/a | n/a | n/a | **T** |
| HT: put $\bowtie$ put | 1.28 | 1.16 | 1.11× | 0 | `poke2`, CVC4 | 0.31 |
| LIA: mVarA $\bowtie$ mVarB | 3.48 | 1.14 | 3.07× | 0 | `poke2`, CVC4 | 10.39 |
| LIA: sum $\bowtie$ multiVarSum | † 8.36 | † 3.69 | 2.27× | -53 | `mcmax`, CVC4 | 1.87 |
| Str2: set $\bowtie$ concat | † 6.66 | n/a | n/a | n/a | n/a | **T** |
| Vcy: auction3 | 23.17 | n/a | n/a | n/a | n/a | **T** |
| Vcy: dict | 2.82 | 2.59 | 1.09× | 0 | `poke2`, CVC5 | 34.09 |
| Vcy: even-odd | † 1.02 | † 0.98 | 1.04× | 0 | `poke2`, CVC5 | 1.46 |
| Vcy: ht-add-put | 3.37 | 3.14 | 1.07× | 0 | `poke2`, CVC4 | 7.50 |
| Vcy: ht-cond-mem-get | † 1.19 | n/a | n/a | n/a | n/a | **T** |
| Vcy: ht-simple | 18.78 | 18.06 | 1.04× | 0 | `poke2`, CVC4 | 100.86 |
| Vcy: linear-cond | 1.30 | 1.06 | 1.23× | 0 | `poke2`, z3 | 1.44 |
| Vcy: loop-inter | 2.40 | n/a | n/a | n/a | n/a | **T** |
| Vcy: nested-counter_2 | † 1.42 | n/a | n/a | n/a | n/a | **T** |
| Vcy: pullPayment | 1.59 | n/a | n/a | n/a | n/a | **T** |
| Vcy: simple | † 2.29 | † 2.11 | 1.09× | 0 | `poke2`, CVC4 | 5.51 |
| Vcy: standardToken2 | 11.31 | 10.31 | 1.10× | 0 | `poke2`, z3, TG | 113.51 |
| Vcy: standardToken3 | 1.45 | n/a | n/a | n/a | n/a | **T** |
| Vcy: standardToken5 | 13.25 | n/a | n/a | n/a | n/a | **T** |

Table 2: Comparison of runtimes for cases where lattice construction and model counting is applicable. Note that in many cases, lattice construction always times out or errors. Lattice timeout was defined at 300s for Veracity benches and 30s for other benches. Those rows are marked with n/a. `mVarA` is short for `multiVarA`.

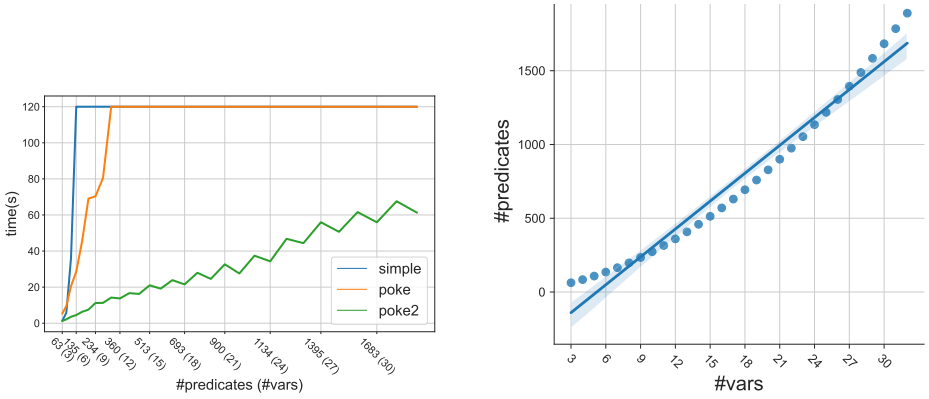
compared to `poke2` and `poke`.

The performance of `mcMax` seems to depend on the methods considered, but there are cases where it can significantly improve run time. In future work, we hope to explore additional model-counting heuristics such as bisecting the search space rather than greedily covering it.

**Extraction of terms.** The original SERVOIS tool required users to provide *terms*, sacrificing some degree of automation which is inconvenient and error-prone. As described in Sec. 3, SERVOIS2 now can automatically extract terms from the method specifications. As shown in Table 1, denoted by TG, the automated term extraction can even outperform manually provided terms.

In addition, by automatically extracting terms, our approach is another step closer to a fully automated commutativity synthesizer—the user does not have to do the manual work of providing terms. We believe that in conjunction with comparable performance, this makes automated term extraction preferable.

**Predicate lattice.** We also evaluated the performance using the predicate lattice approach outlined in Sec. 3. In practice, we found that the overhead of lattice construction using SMT queries was typically too high, and it did not substantially improve synthesis time in most cases. When using syntactic rules (using a preliminary set of inference rules and axioms), we did not discover




(a) Performance versus increased variables/predicates.

(b) Predicates versus variables.

Fig. 6: Experimental results on the scalability of general-purpose heuristics

enough implications to be useful in any cases, with still substantial, albeit greatly reduced overhead. However, we did find that some LIA examples were improved by using the SMT implication lattice. `sum` $\bowtie$ `multiVarSum` in particular saw a 2.27× speedup from 8.36s to 3.69s, which is a substantial speedup even accounting for the lattice construction time of 1.87s. In `multiVarA` $\bowtie$ `multiVarB`, the discovery of logically equivalent predicates filtered more than half of the initial list of predicates—from 280 (including negations of predicates) to 106. For the complete results, see Table 2. While it remains unclear whether the predicate lattice can be used for performance gains in most cases, the preliminary results suggest that further work may yield larger gains in difficult cases.

**Scalability.** Consider the toy example below, where we have a possibly ordered set described by $x$ variables and we want to compute another element $a$ that can potentially be added to the set after applying a marginal decrement. In this (somewhat artificial) example, the number of predicates increases (Fig. 6b) with the number of variables, while the commutativity problem has a straightforward solution: $(b = 0) \vee ((b \neq 0) \wedge (a \geq 0) \wedge (a - b \geq 0) \bigwedge_{i=1}^{n-1}(x_i < x_{i+1}))$.

```
(1) int a, b, x_1, x_2, ..., x_n;
(2) bool sum(){
(3)     a := (a - b); return true; }
(4) bool multiVarSum(){
(5)     if (a>0 && x_1<x_2 && x_2<x_3 &&  ... && x_{n-1}<x_n){
(6)        a := (x_n + a); return true;
(7)     } else {
(8)        a := (x_n - a); return true; }}
```

Although `mcMax` shows promising results, due to reduced applicability to cases where efficient model counting is supported, we did not consider it for this particular experiment. Our focus here is on general-purpose heuristics. Fig. 6a reports the results of our experiments running the heuristics `simple` (presented in [14]), `poke`, and `poke2` on the above example, with increasingly many variables (up to 30) and, consequently, increasingly many predicates. For the precise ADT specification, refer to the "`lia_scale_var_template`" file in the artifact or Github provided in Sec. 1. We observe an impressive performance benefit from the `poke2` heuristic. Firstly, observe that starting from only a small number of predicates, `poke2` proved to be one order of magnitude faster than `poke` which timed out early in our experiment. Secondly, the increase in the number of state variables $x$ is roughly linear with the increase of `poke2` synthesis time. And lastly, the `poke2` heuristic led to synthesizing the utmost simple condition, namely the one humanly inferred.

### 6.2 Case study: `commute` blocks in Veracity

To show Servois2's applicability, we present the case study of its use in the Veracity[††] project [6], recalled below. The original Servois lacked features (*e.g.* solvers, theories, early termination) and performance to be used in such a setting. Despite the use of Servois2 in Veracity, the improvements described in the current paper are orthogonal.

Veracity is a parallelizing compiler for a language in which programmers directly express conditions under which sequential blocks of code commute [6]. Expanding programs with such commutativity annotations enables parallelization of sequential code that has dataflow dependencies, which previously could not be parallelized. We omit the finer details as it is outside the scope of commutativity synthesis. Consider for example, the following Veracity benchmark `even-odd` includes a `commute` statement, with a blank commutativity condition to be synthesized (or provided by the user):

```
commute (_) { { if(x%2==0) x:=x+y;}        (1)
              { x:=x+y; }          }       (2)
```

Veracity needs Servois2 in order to synthesize the following commutativity condition for these program fragments labeled (1) and (2): $y = 0 \vee (y \neq 0 \wedge x\%2 = x + y)$. In more detail, Servois2 is used by first having the Veracity compiler translate the program code into methods, say $block_1()$ and $block_2()$ on an ADT whose state are the program variables `x` and `y`. Then, the synthesized commutativity conditions are translated back and inserted in place of the "`_`" in the Veracity `commute` block. Unfortunately the original Servois's limited support for solvers/theories (as well as limited performance) prevents it from synthesizing a commutativity condition for this benchmark. The divergent behavior of Servois on some benchmarks was a further impediment to its use in Veracity.

A few selected benchmarks are shown in Table 3. These benchmarks are illustrative of the different kinds of typical output from Servois2. Most cases

[††] `http://www.veracity-lang.org`

| Program | Time | Inferred Conditions |
|---|---|---|
| dict | 3.82 | i != r && c + x != y \|\| c + x == y |
| ht-simple | 30.64 | x + a != z && 3 == tbl[z] && y != z |
| loop-amt | >120.00 | 0 == i && amt == i_pre && ctr - 1 > i_pre && i_pre <= amt && 0 != i_pre && i_pre <= ctr && amt != amt_pre && ctr - 1 > amt_pre && amt_pre <= amt && 0 != amt_pre && amt_pre <= ctr && ctr - 1 != 1 && 1 != ctr && 1 != amt && 1 == ctr + amt \|\| ... \|\| amt == i && 1 == ctr && 1 != amt && 1 == ctr + amt |

Table 3: A selected subset of Veracity benchmarks. (Times are in seconds.)

were similar to the `dict` example, terminating in a few seconds with a sensible result. The `ht-simple` case takes more time. The condition is complete, but due to the longer time, it may be worth terminating the algorithm early and only receiving one or two of the disjuncts, especially if they cover the most common cases. Finally, `loop-amt` is a case that is not amenable to commutativity inference and it would be better to terminate sooner and allow the user to attempt a different approach.

Unlike direct ADT benchmarks, those derived from Veracity programs involve the composition of numerous effects and thus involve complex commutativity conditions. Consequently, most of the Veracity benchmarks make substantially more complex queries to Servois2 than the handwritten ADT specifications. We thus used all of the Veracity benchmarks to test the different configurations, as mentioned before and shown in Table 1.

New non-trivial benchmarks were manually translated into the Veracity programming language. These were various combinations of functions from the SmartContract/Auction, Solidity/StandardToken, and Solidity/PullPayment source codes. Most of these new benchmarks perform better on the new heuristic `poke2` compared to the previously presented approach `poke`. Also, for several of them, `poke` did not terminate, so we had to use the early termination feature to synthesize the commutativity condition within a specific time frame. For StandardToken, for example, after executing `TransferForm` $\bowtie$ `Approve` with using `poke` and 120s timeout, we get an incomplete condition; however, with `poke2`, we can get a complete condition in about 10 seconds with a reasonable number of atoms.

## 7 Conclusion and Future Work

We have shown a more mature and performant method of automatically synthesizing commutativity conditions in the Servois2 synthesizer. Our results confirm what one might expect: that more advanced heuristics and better treatment of predicates leads to overall performance improvement. Furthermore, we have released a far more usable tool that has already been used in recent work [6] and is ready to be integrated into other commutativity settings such as proof methodologies [1,3,2] or distributed systems [9,11]. There are several directions for future work in this space, discussed below.

*Algorithmic improvements.* We saw great improvements in the performance of the heuristic in keeping track of which predicates aligned with the commutative (resp. non-commutative) case. The algorithm is currently agnostic to which condition is being pursued, and it may be possible to tag such information in the recursive calls, leading to similar improvements in performance. Furthermore, the disjunctive nature of the algorithm may be amenable to parallelization. However, it is unclear whether the actual reasoning is amenable to parallelization or if it is not worth the overhead.

*Extended use of model counting.* `mcMax` uses the model-counting solver ABC [23], which targets string, LIA, and boolean constraints, but we could also use other model counters with support for other theories. Approximate model counters [24] are a promising avenue for handling model-counting queries across additional theories, and the integration of such a model counter might lead to further applicability of `mcMax`.

The `mcMax` heuristic provides one model counting heuristic to inform predicate selection, but we hypothesize that additional heuristics might provide advantages on different benchmarks, for example, by maximizing partitioning rather than covering. Given the promising results of `mcMax`, we plan to pursue a more extensive evaluation of model-counting heuristics.

Model counting might find an additional use in cases where our commutativity analysis terminates early. Using model counting, we can determine what portion of the input domain is covered by the resulting commutativity and non-commutativity conditions, augmenting our analysis with additional reliability information in cases of early termination. It also may be possible to use this information to determine when to terminate.

*Improving the use of the predicate lattice.* Our experiments indicate that the overhead of lattice construction is significant. Thus for the lattice to be practical, one would need to both increase its performance benefit and decrease the overhead from construction. Although Refine prunes predicates based on the lattice, none of the current heuristics use information about implication chains, and there may be even more gains to be had by using lattices. There are also more sophisticated approaches to building the lattice data structure of logical implications, such as by using the framework GreenTrie [25]. The number of queries can be reduced through semantic reasoning and caching sub-formulas. This could reduce the overhead of lattice construction, thus making their use more appealing.

*Acknowledgements.* This work is supported in part by NSF Award #2008633 and #2107169.

## References


1. A. Farzan and A. Vandikas, "Reductions for safety proofs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–28, 2019.
2. B. Kragl and S. Qadeer, "The civl verifier," in *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2021, pp. 143–152.
3. C. Flanagan and S. N. Freund, "The anchor verifier for blocking and non-blocking concurrent software," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, 2020.
4. M. C. Rinard and P. C. Diniz, "Commutativity analysis: A new analysis technique for parallelizing compilers," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 6, pp. 942–991, November 1997. [Online]. Available: citeseer.ist.psu.edu/rinard97commutativity.html
5. A. Spiegelman, G. Golan-Gueta, and I. Keidar, "Transactional data structure libraries," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 682–696, 2016.
6. A. Chen, P. Fathololumi, E. Koskinen, and J. Pincus, "Veracity: Declarative multicore programming with commutativity)," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 186:1–186:31, October 2022. [Online]. Available: https://doi.org/10.1145/3563349
7. P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August, "Commutative set: A language extension for implicit parallel programming," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 1–11.
8. A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 4, pp. 1–47, 2015.
9. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," Ph.D. dissertation, Inria–Centre Paris-Rocquencourt; INRIA, 2011.
10. T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: ACM, 2017, pp. 303–312. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087835
11. G. Pîrlea, A. Kumar, and I. Sergey, "Practical smart contract sharding with ownership and commutativity analysis," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 1327–1341. [Online]. Available: https://doi.org/10.1145/3453483.3454112
12. T. Gehr, D. Dimitrov, and M. T. Vechev, "Learning commutativity specifications," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, 2015, pp. 307–323. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21690-4_18
13. F. Aleen and N. Clark, "Commutativity analysis for software parallelization: letting program transformations see the big picture," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, M. L. Soffa and M. J. Irwin, Eds. ACM, 2009, pp. 241–252.
14. K. Bansal, E. Koskinen, and O. Tripp, "Automatic generation of precise and useful commutativity conditions," in *Tools and Algorithms for the*

*Construction and Analysis of Systems - 24th International Conference, TACAS 2018*, ser. Lecture Notes in Computer Science, D. Beyer and M. Huisman, Eds., vol. 10805. Springer, 2018, pp. 115–132. [Online]. Available: https://doi.org/10.1007/978-3-319-89960-2_7

15. K. Bansal, E. Koskinen, and O. Tripp, "Synthesizing precise and useful commutativity conditions," *Journal of Automated Reasoning*, vol. 64, no. 7, pp. 1333–1359, 2020.
16. C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting," in *Handbook of satisfiability*. IOS press, 2021, pp. 993–1014.
17. J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, "Effective lattice point counting in rational convex polytopes," *Journal of symbolic computation*, vol. 38, no. 4, pp. 1273–1302, 2004.
18. A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 255–272.
19. A. Molavi, T. Schneider, M. Downing, and L. Bang, "Mcbat: Model counting for constraints over bounded integer arrays," in *Software Verification: 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20–21, 2020, Revised Selected Papers 13*. Springer, 2020, pp. 124–143.
20. C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 171–177. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_14
21. H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24
22. L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.
23. A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 255–272.
24. S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Approximate model counting," in *Handbook of Satisfiability*. IOS Press, 2021, pp. 1015–1045.
25. X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 177–187. [Online]. Available: https://doi.org/10.1145/2771783.2771806