# Automatic Generation of Precise and Useful Commutativity Conditions

Kshitij Bansal[1], Eric Koskinen[2], and Omer Tripp[1]

[1] Google, Inc.
[2] Stevens Institute of Technology[†]

**Abstract.** Reasoning about commutativity between data-structure operations has been, and remains, an important problem with applications including parallelizing compilers, optimistic parallelization and, more recently, Ethereum smart contracts. There have been research results on automatic generation of commutativity conditions, yet we are unaware of any fully automated technique to generate conditions that are both sound and effective (i.e., not overly conservative).
We take a first step in this direction. We have designed such a technique, driven by an algorithm that iteratively refines a conservative approximation of the commutativity (and non-commutativity) condition for a pair of methods into an increasingly precise version. The algorithm terminates if/when the entire state space has been considered, and can be aborted at any time to obtain a partial yet sound commutativity condition. We have generalized our work to left-/right-movers [25] and proved relative completeness. We describe aspects of our technique that lead to *useful* commutativity conditions, including how predicates are selected during refinement and heuristics that impact the output shape of the condition. We have implemented our technique in a prototype open-source tool called SERVOIS. Our algorithm produces quantifier-free queries that are dispatched to a back-end SMT solver. We evaluate SERVOIS through two case studies: (i) We synthesize commutativity conditions for a range of data structures including Set, HashTable, Accumulator, Counter, and Stack. (ii) We consider an Ethereum smart contract called BlockKing, and show that SERVOIS can detect serious concurrency-related vulnerabilities and guide developers to construct robust and efficient implementations.

## 1 Introduction

Reasoning about the conditions under which data-structure operations commute is an important problem. The ability to derive sound yet effective commutativity conditions unlocks the potential of multicore architectures, including parallelizing compilers [28,32], speculative execution (*e.g.* transactional memory [17]), peephole partial-order reduction [36], futures, etc. Another important application domain that has emerged recently is Ethereum smart contracts: efficient execution of such contracts hinges on exploiting their commutativity [12].

[†]Support in part by NSF CCF Award #1618059 and CCF Award #1421126.

Intuitively, commutativity is an important property because linearizable data-structure operations that commute can be executed concurrently: their effects don't interfere with each other in an observable way. When using a linearizable HashTable, for example, knowledge that `put(x,'a')` commutes with `get(y)` provided that $\texttt{x} \neq \texttt{y}$ enables significant parallelization opportunities as both can be performed concurrently. Indeed, it's important for the commutativity condition to be sufficiently granular so that parallelism can be exploited effectively [10]. At the same time, to make safe use of a commutativity condition, it must be sound [22,21]. Achieving both of these goals using manual reasoning is burdensome and error prone.

In light of that, researchers have investigated ways of verifying user-provided commutativity conditions [20] as well as synthesizing such conditions automatically, *e.g.* based on random interpretation [6], profiling [31] or sampling [16]. None of these approaches, however, meets the goal of computing a commutativity condition that is both *sound* and *granular* in a *fully automated* manner.

In this paper, we take a first step in this direction. We present a refinement-based technique for generating commutativity conditions. Our technique builds on well-known descriptions and representations of abstract data types (ADTs) in terms of logical $(Pre_m, Post_m)$ specifications [18,14,15,7,26,24] for each method $m$. Our algorithm iteratively relaxes under-approximations of the commutativity *and* non-commutativity conditions of methods $m$ and $n4$, starting from $\mathsf{false}$, into increasingly precise versions. At each step, we conjunctively subdivide the symbolic state space into regions, searching for areas where $m$ and $n$ commute and where they don't. Counterexamples to both the positive side and the negative side are used in the next symbolic subdivision. Throughout this recursive process, we accumulate the commutativity condition as a growing disjunction of these regions. The output of our procedure is a logical formula $\varphi_m^n$ specifies when method $m$ commutes with method $n$. We have proven that the algorithm is sound, and can also be aborted at any time to obtain a partial, yet useful [31,17], commutativity condition. We show that, under certain conditions, termination is guaranteed (relative completeness).

We address several challenges that arise in using an iterative refinement approach to generating precise and useful commutativity conditions. First, we show how to pose the commutativity question in a way that does not introduce additional quantifiers. We also show how to generate the predicate vocabulary for expressing the condition $\varphi_m^n$, as well as how to choose the predicates throughout the refinement loop. A further question that we address is how predicate selection impacts the conciseness and readability of the generated commutativity conditions. Finally, we have generalized our algorithm to left-/right-movers [25], a more precise version of commutativity.

We have implemented our approach as the Servois tool, whose code and documentation are available online [2]. Servois is built on top of the CVC4 SMT solver [8]. We evaluate Servois through two case studies. First, we generate commutativity conditions for a collection of popular data structures, including Set, HashTable, Accumulator, Counter, and Stack. The conditions typically combine multiple theories, such as sets, integers, arrays, etc. We show the conditions to be comparable in granularity to manually specified conditions [20]. Second, we consider BlockKing [29], an Ethereum smart contract, with its known vulnerability. We demonstrate how a developer can be guided by SERVOIS to create a more robust implementation.

**Contributions.** In summary, this paper makes the following contributions:

- The first sound and precise technique to automatically generate commutativity conditions (Sec. 4).
- Proof of soundness and relative completeness (Sec. 4).
- An implementation that takes an abstract code specification and automatically generates commutativity conditions using an SMT solver (Sec. 5).
- A novel technique for selecting refinement predicates that improves scalability and the simplicity of the generated formulae (Sec. 5).
- Demonstrated efficacy for several key data structures (Sec. 6.1) as well as the BlockKing Ethereum smart contract [29]. (Sec. 6.2).

**Related work.** The closest to our contribution in this paper is a recent technique by Gehr *et al.* [16] for learning, or inference, of commutativity conditions based on black-box sampling. They draw concrete arguments, extract relevant predicates from the sampled set of examples, and then search for a formula over the predicates. There are no soundness or completeness guarantees.

Both Aleen and Clark [6] and Tripp *et al.* [31] identify sequences of actions that commute (via random interpretation and dynamic analysis, respectively). However, neither technique yields an explicit commutativity condition. Kulkarni et al. [23] point out that varying degrees of commutativity specification precision are useful. Kim and Rinard [20] use Jahob to verify manually specified commutativity conditions of several different linked data structures. Commutativity specifications are also found in dynamic analysis techniques [13].

More distantly related is the work on synthesizing program implementations, such as CEGIS [30] and synchronization synthesis [35,34]. Our algorithm uses a form of counterexample-guided abstraction refinement [9].

## 2 Example

Consider the Set ADT, whose state consists of a single set $S$ that stores an unordered collection of unique elements. Specifying commutativity conditions is generally nontrivial, more importantly it is easy to miss subtle corner cases. Additionally, it has to be done pairwise for all methods. For ease of illustration, we will focus on a relatively simple Set example and one pair of operations: (i) `contains(`$x$`)/bool`, a side-effect-free check whether the element $x$ is in $S$; and (ii) `add(`$y$`)/bool` adds $y$ to $S$ if it is not already there and returns `true`, or otherwise returns `false`. `add` and `contains` clearly commute if they refer to different elements in the set. There is another case that is less obvious: `add` and `contains` commute if they refer to the same element $e$, as long as in the pre-state $e \in S$. In this case, under both orders of execution, `add` and `contains` leave the set unmodified and return false and true, respectively. The algorithm we describe in this paper takes 3.6s to automatically produce a precise logical formula $\varphi$ that captures this commutativity condition, *i.e.* the disjunction of the two cases above: $\varphi \equiv x \neq y \vee (x = y \wedge x \in S)$. The algorithm also generates the conditions under which the methods *do not* commute: $\tilde{\varphi} \equiv x = y \wedge x \notin S$. These are precise, since $\varphi$ is the negation of $\tilde{\varphi}$.

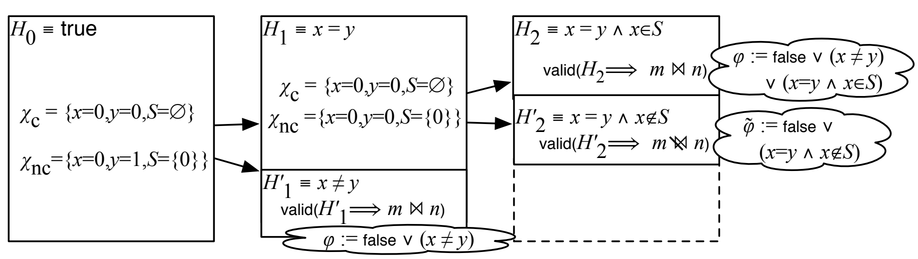
A more complicated commutativity condition generated by our tool, Servois, for BlockKing (Sec. 6.2) is for method enter(val$_1$,sendr$_1$,bk$_1$...) and completed in 1.4s. It does not commute with itself enter(val$_2$,sendr$_2$,bk$_2$...) if and only if:

$$\bigvee \begin{cases} \mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{sendr}_1 \neq \mathsf{sendr}_2 \\ \mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{sendr}_1 = \mathsf{sendr}_2 \wedge \mathsf{val}_1 \neq \mathsf{val}_2 \\ \mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{sendr}_1 = \mathsf{sendr}_2 \wedge \mathsf{val}_1 = \mathsf{val}_2 \wedge \mathsf{bk}_1 \neq \mathsf{bk}_2 \end{cases}$$

Eliding the details for now, this disjunction effectively enumerates the non-commutativity cases. As we will see in Sec. 6.2, this condition directly identifies a vulnerability in BlockKing.

Capturing precise conditions such as these by hand, and doing so for many pairs of operations, is tedious and error prone. This paper instead presents a way to automate this. Our algorithm recursively subdivides the state space via predicates until, at the base case, regions are found that are either entirely commutative or else entirely non-commutative.

Returning to our Set example, the conditions we incrementally generate are denoted $\varphi$ and $\tilde{\varphi}$, respectively. The following diagram illustrates how our algorithm proceeds to generate the commutativity conditions for methods `add` and `contains`.




In this diagram, each subsequent panel depicts a partitioning of the state space into regions of commutativity ($\varphi$) or non-commutativity ($\tilde{\varphi}$). The counterexamples $\chi_c, \chi_{nc}$ give values for the arguments $x$, $y$ and the current state of the set $S$.

We denote by $H$ the logical formula that describes the current state space at a given recursive call. As expected, we begin with $H_0 = \mathsf{true}$, $\varphi = \mathsf{false}$, and $\tilde{\varphi} = \mathsf{false}$. There are three cases for a given $H$: (i) $H$ describes a precondition for $m$ and $n$ in which they *always* commute; (ii) $H$ describes a precondition for $m$ and $n$ in which they *never* commute; or (iii) neither of the above. The latter case drives the algorithm to subdivide the region by choosing a new predicate.

We detail the run of this refinement loop on our earlier Set example. We reserve elaborating on other subtleties and challenges to the later sections (some are briefly described in the outline below). At each step of the algorithm, we determine which case we are in via carefully designed validity queries to an SMT solver. For $H_0$, it returns the commutativity counterexample described above: $\chi_c = \{x = 0, y = 0, S = \emptyset\}$ as well as the non-commutativity counterexample $\chi_{nc} = \{x = 0, y = 1, S = \{0\}\}$. Since, therefore, $H_0 = \mathtt{true}$ is neither a commutativity nor a non-commutativity condition, we must refine $H_0$ into regions (or stronger conditions). In particular, we would like to perform a *useful* subdivision: Divide $H_0$ into an $H_1$ that allows $\chi_c$ but disallows $\chi_{nc}$, and an $H'_1$ that allows $\chi_{nc}$ but not $\chi_c$. To this end, we must choose a predicate $p$ (from a suitable set of predicates $\mathcal{P}$, discussed later), such that $H_0 \wedge p \Rightarrow \chi_c$ while $H_0 \wedge \neg p \Rightarrow \chi_{nc}$ (or vice versa). The predicate $x = y$ satisfies this property. The algorithm then makes the next two recursive calls, adding $p$ as a conjunct to $H$, as shown in the second column of the diagram above: one with $H_1 \equiv \mathtt{true} \wedge x = y$ and one with $H'_1 \equiv \mathtt{true} \wedge x \neq y$. Taking the $H'_1$ case, our algorithm makes another SMT query and finds that $x \neq y$ implies that `add` always commutes with `contains`. At this point, it can update the commutativity condition $\varphi$, letting $\varphi := \varphi \vee H'_1$, adding this $H'_1$ region to the growing disjunction. On the other hand, $H_1$ is neither a sufficient commutativity nor a sufficient non-commutativity condition, and so our algorithm, again, produces the respective counterexamples: $\chi_c = \{x = 0, y = 0, S = \emptyset\}$ amd $\chi_{nc} = \{x = 0, y = 0, S = \{0\}\}$. In this case, our algorithm selects the predicate $x \in S$, and makes two further recursive calls: one with $H_2 \equiv x = y \wedge x \in S$ and another with $H'_2 \equiv x = y \wedge x \notin S$. In this case, it finds that $H_2$ is a sufficiently strong precondition for commutativity, while $H'_2$ is a strong enough precondition for non-commutativity. Consequently, $H_2$ is added as a new conjunct to $\varphi$, yielding $\varphi \equiv x \neq y \vee (x = y \wedge x \in S)$. Similarly, $\tilde{\varphi}$ is updated to be: $\tilde{\varphi} \equiv (x = y \wedge x \notin S)$.

No further recursive calls are made so the algorithm terminates and, as we show in Lemma 1, we have obtained a precise (complete) commutativity/non-commutativity specification: $\varphi \vee \tilde{\varphi}$ is valid.

While the algorithm outlined so far is a relatively standard refinement loop, the above generated conditions were not immediate. We now discuss the challenges involved in enabling generation of sound yet useful conditions.

**Outline.** A first question is how to pose the underlying commutativity queries for each subsequent $H$, which are discharged to an SMT solver. We describe how to pose these queries in a way that avoids the introduction of additional quantifiers, so that we can remain in fragments for which the solver has complete decision procedures. Thus, if the data structure can be encoded using theories that are decidable, then the queries we pose to the SMT solver are guaranteed to be decidable as well. $Pre_m/Post_m$ specifications that are partial would introduce quantifier alternation, but we show how this can be avoided by, instead, transforming them into total specifications.

Guarantees (Sec. 4). We have proven that our algorithm is sound, and it produces sound commutativity conditions, even if aborted or the ADT description involves undecidable theories. We further show that termination implies completeness, and specify broad conditions that imply termination (*i.e.* relative completeness).

A second challenge is to prioritize predicates during the refinement loop. This choice impacts not only the algorithm's performance, but also the quality (or conciseness) of the resulting conditions. Our choice of next predicate $p$ is governed by two requirements. First, for progress, $p/\neg p$ must eliminate the counterexamples to commutativity/non-commutativity due to the last iteration. This may still leave multiple choices, and we propose two heuristics – called *simple* and *poke*—with different trade-offs to break ties. This is discussed in Sec. 5, along with other practical considerations.

We conclude with an evaluation on a range of popular data structures (Sec. 6.1) and a case study on boosting the security of an Ethereum smart contract (Sec. 6.2).

## 3 Preliminaries

**States, actions, methods.** We will work with a state space denoted $\Sigma$, with decidable equality and a set of *actions* $A$. For each $\alpha \in A$, we have a transition function $(\!|\alpha|\!) : \Sigma \rightharpoonup \Sigma$. We denote a single transition as $\sigma \xrightarrow{\alpha} \sigma'$. We assume that each such action arc completes in finite time. Let $\mathfrak{T} \equiv (\Sigma, A, (\!|\bullet|\!))$. We say that two *actions* $\alpha_1$ and $\alpha_2$ *commute* [13], denoted $\alpha_1 \bowtie \alpha_2$, provided that $(\!|\alpha_1|\!) \circ (\!|\alpha_2|\!) = (\!|\alpha_2|\!) \circ (\!|\alpha_1|\!)$. Note that $\bowtie$ is with respect to $\mathfrak{T} = (\Sigma, A, (\!|\bullet|\!))$. Our formalism, implementation, and evaluation all extend to a more fine-grained notion of commutativity: an asymmetric version called left-movers and right-movers [25], where a method commutes in one direction and not the other. For ease of presentation, the formal detail in the body of this paper discusses only commutativity, but a discussion of how our technique generalizes can be found in Apx B. Also, in our evaluation (Sec. 6) we show left-/right-mover conditions that were generated by our implementation.

An action $\alpha \in A$ is of the form $m(\bar{x})/\bar{r}$, where $m$, $\bar{x}$ and $\bar{r}$ are called a *method*, *arguments* and *return values* respectively. As a convention, for actions corresponding to a method $n$, we use $\bar{y}$ for arguments and $\bar{s}$ for return values. In our context, the set of methods will be finite, inducing a finite partitioning of $A$. We refer to an action, say $m(\bar{a})/\bar{v}$, as *corresponding* to method $m$ (where $\bar{a}$ and $\bar{v}$ are vectors of values). The set of actions corresponding to a method $m$, denoted $A_m$, might be infinite as the arguments and return values may be from an infinite domain. A given method corresponds to many possible actions.

**Definition 1 (Method Commutativity).** *For $m$ and $n$,*

$$m \bowtie n \equiv \forall \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\ m(\bar{x})/\bar{r} \bowtie n(\bar{y})/\bar{s}$$

Above we have quantified over all actions corresponding to $m$ and $n$. That is, the quantification $\forall \bar{x}\bar{r}$ means $\forall m(\bar{x})/\bar{r} \in A_m$, i.e., all vectors of arguments and return values that constitute an action in $A_m$.

**Abstract specifications.** We describe the actions of a method $m$ symbolically as pre-condition $Pre_m$ and post-condition $Post_m$. Pre-conditions are logical formulae over method arguments and the initial state, and post-conditions are over method arguments, and return values, initial state and final state:

$$[\![Pre_m]\!] : \bar{x} \to \Sigma \to \mathbb{B} \qquad [\![Post_m]\!] : \bar{x} \to \bar{r} \to \Sigma \to \Sigma \to \mathbb{B}$$

Given $(Pre_m, Post_m)$ for every method $m$, we define a transition system $\mathfrak{T} = (\Sigma, A, (\!| \bullet |\!))$ such that $\sigma \xrightarrow{m(\bar{x})/\bar{r}} \sigma'$ *iff* $[\![Pre_m]\!]\ \bar{x}\ \sigma$ and $[\![Post_m]\!]\ \bar{x}\ \bar{r}\ \sigma\ \sigma'$.

Since our approach works on deterministic transition systems, we have implemented a check (discussed in Sec. 6) that ensures the input transition system is deterministic. Deterministic specifications were sufficient to model the Set, HashTable, Accumulator, Counter, and Stack data structures. This is unsurprising given the inherent difficulty of creating efficient concurrent implementations of nondeterministic operations, whose effects are hard to characterize. Reducing nondeterministic data-structure methods to deterministic ones through symbolic partial determinization [5,11] is left as future work.

**Logical commutativity formulae.** We will work with, and generate, a commutativity condition for methods $m$ and $n$ as logical formulae over initial states and the arguments/return values of the methods. We denote a logical commutativity formula as $\varphi$ and assume a decidable interpretation of formulae: $[\![\varphi]\!] : (\sigma, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \to \mathbb{B}$. (We tuple the arguments for brevity.) The first argument is the initial state. Commutativity *post-* and *mid-*conditions can also be written [20], but here, for simplicity, we focus only *pre-*conditions. Throughout this paper, we may write $[\![\varphi]\!]$ as simply $\varphi$ when it is clear from context that $\varphi$ is meant to be interpreted.

We say that $\varphi_m^n$ is a *sound commutativity condition*, and $\hat{\varphi}_m^n$ a sound *non-*commutativity condition resp., for $m$ and $n$ provided that

$$\forall \sigma \bar{x} \bar{y} \bar{r} \bar{s}.\ [\![\varphi_m^n]\!]\ \sigma\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s} \Rightarrow m(\bar{x})/\bar{r} \bowtie n(\bar{y})/\bar{s}, \text{ and}$$
$$\forall \sigma \bar{x} \bar{y} \bar{r} \bar{s}.\ [\![\hat{\varphi}_m^n]\!]\ \sigma\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s} \Rightarrow \neg(m(\bar{x})/\bar{r} \bowtie n(\bar{y})/\bar{s}), \text{ resp.}$$

### 3.1 Commutativity without Quantifier Alternation

Def. 1 requires showing equivalence between different compositions of potentially partial functions. That is, $(\!|\alpha_1|\!) \circ (\!|\alpha_2|\!) = (\!|\alpha_2|\!) \circ (\!|\alpha_1|\!)$ if and only if:

$$\forall \sigma_0\ \sigma_1\ \sigma_{12}.\ (\!|\alpha_1|\!)\sigma_0 = \sigma_1 \wedge (\!|\alpha_2|\!)\sigma_1 = \sigma_{12} \quad \Rightarrow \quad \exists \sigma_3.\ (\!|\alpha_2|\!)\sigma_0 = \sigma_3 \wedge (\!|\alpha_1|\!)\sigma_3 = \sigma_{12}$$
$$(\textit{and a symmetric case for the other direction})$$

Even when the transition relation can be expressed in a decidable theory, because of $\forall\exists$ quantifier alternation in the above encoding (which is undecidable in general), any procedure requiring such a check would be incomplete. SMT solvers are particularly poor at handling such constraints.

We observe that when the transition system is specified as $Pre_m$ and $Post_m$ conditions, and the $Post_m$ condition is *consistent* with $Pre_m$, then it is possible to avoid quantifier alternation. By consistent we mean that whenever $Pre_m$ holds, there is always some state and return value for which $Post_m$ holds.

$$\forall \bar{a}\ \sigma.\quad Pre_m(\bar{a}, \sigma) = \mathsf{true} \ \Rightarrow\ \exists \sigma'\ \bar{r}.\ Post_m(\bar{a}, \bar{r}, \sigma, \sigma').$$

In particular, this assumption holds for all of the specifications in the examples we considered (see Sec. 6). This allows us to perform a simple transformation on transition systems to a lifted domain, and enforce a definition of commutativity in the lifted domain denoted $m \mathrel{\hat{\bowtie}} n$ that is equivalent to Eqn. 1. This new definition requires only *universal* quantification, and as such, is better suited to automation of SMT-backed algorithms (*e.g.* Sec. 4).

For lack of space, we have deferred the technical details of this lifting transformation denoted LIFT to Apx. A. If the reader prefers, it suffices instead to simply think of a lifted transition system $\hat{\mathfrak{T}} = (\hat{\Sigma}, A, (\!|\bullet|\!))$ as a normal transition system, except with the guarantee that $(\!|\bullet|\!)$ is total. The transformation involves introducing a special state called $Err$, and mapping any undefined states in $(\!|\bullet|\!)$ to $Err$, as well as mapping $Err$ to $Err$. We also use notations $(\widehat{Pre_m}, \widehat{Post_m})$ and $\hat{\bowtie}$ when considering commutativity in the lifted domain.

## 4 Iterative Refinement

We now present an iterative refinement strategy that, when given a lifted abstract transition system, generates the commutativity and the non-commutativity conditions. We then discuss soundness and relative completeness and, in Secs. 5 and 6, several challenges involved in enabling this refinement to generate precise and useful commutativity conditions.

```
Refine_n^m(H, P) {
  if valid(H ⇒ m ⋈̂ n) then
    φ := φ ∨ H;
  else if valid(H ⇒ m ⋈̸̂ n) then
    φ̃ := φ̃ ∨ H;
  else
    let (χ_c, χ_nc) = counterexs. to ⋈̂ and ⋈̸̂
    let p = Choose(H, P, χ_c, χ_nc) in
      Refine_n^m(H ∧ p, P \ {p});
      Refine_n^m(H ∧ ¬p, P \ {p});
}
main {
  φ := false;   φ̃ := false;
  try { Refine_n^m(true, P); }
  catch (InterruptedExn e) { skip; }
  return(φ, φ̃);
}
```

**Fig. 1.** Algorithm for generating commutativity $\varphi$ and non-commutativity $\tilde{\varphi}$.

The refinement algorithm symbolically searches the state space for regions where the operations commute (or do not commute) in a conjunctive manner, adding on one predicate at a time. We add each subregion $H$ (described conjunctively) in which commutativity always holds to a growing disjunctive description of the commutativity condition $\varphi$, and each subregion $H$ in which commutativity never holds to a growing disjunctive description of the non-commutativity condition $\tilde{\varphi}$.

The algorithm in Fig. 1 begins by setting $\varphi = \mathsf{false}$ and $\tilde{\varphi} = \mathsf{false}$. REFINE begins a symbolic binary search through the state space $H$, starting from the entire state: $H = \mathsf{true}$. It also may use a collection of predicates $\mathcal{P}$ (discussed later). At each iteration, REFINE checks whether the current $H$ represents a region of space for which $m$ and $n$ always commute: $H \Rightarrow m \mathbin{\hat{\bowtie}} n$ (described below). If so, $H$ can be disjunctively added to $\varphi$. It may, instead be the case that $H$ represents a region of space for which $m$ and $n$ never commute: $H \Rightarrow m \mathbin{\hat{\not\bowtie}} n$. If so, $H$ can be disjunctively added to $\tilde{\varphi}$. If neither of these cases hold, we have two counterexamples. $\chi_{\mathrm{c}}$ is the counterexample to commutativity, returned if the validity check on Line 2 fails. $\chi_{\mathrm{nc}}$ is the counterexample to *non*-commutativity, returned if the validity check on Line 4 fails.

We now need to subdivide $H$ into two regions. This is accomplished by selecting a new predicate $p$ via the CHOOSE method. For now, let the method CHOOSE and the choice of predicate vocabulary $\mathcal{P}$ be parametric. REFINE is sound regardless of the behavior of CHOOSE. Below we give the conditions on CHOOSE that ensure relative completeness, and in Sec. 6 we discuss our particular strategy. Regardless of what $p$ is returned by CHOOSE, two recursive calls are made to REFINE, one with argument $H \wedge p$, and the other with argument $H \wedge \neg p$.

The refinement algorithm generates commutativity conditions that are in disjunctive normal form. Hence, any (finite) logical formula can be represented. This logical language is more expressive than previous commutativity logics that, because they were designed for run-time purposes, were restricted to conjunctions of inequalities [23] and boolean combinations of predicates over finite domains [13].

**Checking a candidate $H_m^n$.** Our algorithm involves checking whether ($H_m^n \Rightarrow m \mathbin{\hat{\bowtie}} n$) or ($H_m^n \Rightarrow m \mathbin{\hat{\not\bowtie}} n$). As shown in Sec. 3.1, we can check whether $H_m^n$ specifies conditions under which $m \bowtie n$ via an SMT query that does not introduce quantifier alternation. For brevity, we define:

$$\mathsf{valid}(H_m^n \;\Rightarrow\; m \mathbin{\hat{\bowtie}} n) \;\equiv\; \mathsf{valid}\begin{pmatrix} \forall \hat{\sigma}_0 \; \bar{x} \; \bar{y} \; \bar{r} \; \bar{s}. \quad H_m^n(\hat{\sigma}_0, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \;\Rightarrow \\ m(\bar{x})/\bar{r} \;\; n(\bar{y})/\bar{s} \;\; \hat{\sigma}_0 = n(\bar{y})/\bar{s} \;\; m(\bar{x})/\bar{r} \;\; \hat{\sigma}_0 \end{pmatrix}$$

Above we assume as a black box an SMT solver providing $\mathsf{valid}$. Here we have lifted the universal quantification within $\hat{\bowtie}$ outside the implication.

We can similarly check whether $H_m^n$ is a condition under which $m$ and $n$ *do not* commute. First, we define negative analogs of commutativity:

$$\begin{aligned} \alpha_1 \mathbin{\not\bowtie} \alpha_2 \;&\equiv \forall \hat{\sigma}_0.\; \hat{\sigma}_0 \neq Err \;\Rightarrow\; \llparenthesis \alpha_2 \rrparenthesis \; \llparenthesis \alpha_1 \rrparenthesis \; \hat{\sigma}_0 \neq \llparenthesis \alpha_1 \rrparenthesis \; \llparenthesis \alpha_2 \rrparenthesis \; \hat{\sigma}_0 \\ m \mathbin{\hat{\not\bowtie}} n \;&\equiv \forall \bar{x} \; \bar{y} \; \bar{r} \; \bar{s}.\; m(\bar{x})/\bar{r} \mathbin{\not\bowtie} n(\bar{y})/\bar{s} \end{aligned}$$

We thus define a check for when $\varphi_m^n$ is a *non*-commutativity condition with:

$$\mathsf{valid}(H_m^n \;\Rightarrow\; m \mathbin{\hat{\not\bowtie}} n) \;\equiv\; \mathsf{valid}\begin{pmatrix} \forall \hat{\sigma}_0 \; \bar{x} \; \bar{y} \; \bar{r} \; \bar{s}.\; H_m^n(\hat{\sigma}_0, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \;\Rightarrow\; \hat{\sigma}_0 \neq Err \;\Rightarrow \\ m(\bar{x})/\bar{r} \;\; n(\bar{y})/\bar{s} \;\; \hat{\sigma}_0 \neq n(\bar{y})/\bar{s} \;\; m(\bar{x})/\bar{r} \;\; \hat{\sigma}_0 \end{pmatrix}$$

The following theorem shows that $\varphi$ is a sound approximation of when $m \mathbin{\hat{\bowtie}} n$ always holds (and similarly for $\tilde{\varphi}$).

**Theorem 1 (Soundness).** *Each* $\text{REFINE}_n^m$ *iter.,* $\varphi \Rightarrow m \mathbin{\hat{\bowtie}} n$*, and* $\tilde{\varphi} \Rightarrow m \mathbin{\hat{\not\bowtie}} n$*.*

Soundness holds regardless of what Choose returns (not surprising since updates to $\varphi$ and $\tilde{\varphi}$ are guarded by validity checks) and even when the theories used to model the underlying data-structure are incomplete. Next we show that termination implies completeness:

**Lemma 1.** *If* $\textsc{Refine}_n^m$ *terminates, then* $\varphi \vee \tilde{\varphi}$.

**Theorem 2 (Conditions for Termination).** $\textsc{Refine}_n^m$ *terminates if:*

1. **(expressiveness)** *the state space* $\Sigma$ *is partitionable into a finite set of regions* $\Sigma_1, ..., \Sigma_N$*, each described by a finite conjunction of predicates* $\psi_i$*, such that either* $\psi_i \Rightarrow m \mathrel{\hat{\bowtie}} n$ *or* $\psi_i \Rightarrow m \mathrel{\hat{\not\bowtie}} n$*; and*
2. **(fairness)** *for every* $p \in \mathcal{P}$*,* Choose *eventually picks* $p$ *(note that this does not imply that* $\mathcal{P}$ *is finite),*

Note that while these conditions ensure termination, the bound on the number of iterations depends on the predicate language and behavior of Choose.

## 5 The Servois tool and practical considerations

**Input.** We use an input specification language building on YAML (which has parser and printer support for all common programming languages) with SMTLIB as the logical language. This format is human readable and can be automatically generated relatively easily, thus enabling the integration with other tools [18,14,15,7,26,24]. See Apx. D.1 for the Counter ADT specification, which was derived from the *Pre* and *Post* conditions used in earlier work [20].

The states (state) of a transition system describing an ADT are encoded as list of variables (each as a name, type pair), and each method (method) specification requires a list of argument types (args), return type (return), and *Pre* (requires) and *Post* (ensures) conditions. For an example (Counter [20]), see Apx. D.1.

**Implementation.** We have developed the open-source Servois tool [3], which implements Refine, Lift, predicate generation, and a method for selecting predicates (Choose) discussed below. Servois uses CVC4 [8] as a backend SMT solver. Servois begins by performing some pre-processing on the input transition system. It checks that the transition system is deterministic. Next, in case the transition system is partial, Servois performs the Lift transformation (Sec. 3.1 and Apx. A). An example of Lift applied to Counter is in Apx. D.2.

Next, Servois automatically generates the predicate language (in addition to user-provided hints). As discussed in Sec. 4, if the predicate vocabulary is not sufficiently expressive, then the algorithm would not be able to converge on precise commutativity and non-commutativity conditions. We generate predicates by using terms and operators that appear in the specification, and generating well-typed atoms not trivially true or false. As we demonstrate in Sec. 6, this strategy works well in practice. An intuitive explanation is that the *Pre* and *Post* formulas suffice to express the footprint of an operation, and so the atoms comprising them are an effective vocabulary to express when operations do, or do not, interfere.

**Predicate selection (Choose).** Even though the number of computed predicates is relatively small, since our algorithm is exponential in number of predicates it is essential to be able to identify *relevant* predicates for the algorithm. To this end, in addition to filtering trivial predicates, we prioritize predicates based on the *two* counterexamples generated by the validity checks in Refine. Predicates that distinguish between the given counter examples are tried first (call these *distinguishing* predicates). More formally, Choose must return a predicate such that $\chi_c \Rightarrow\ H \wedge p$ and $\chi_{nc} \Rightarrow\ H \wedge \neg p$. This guarantees progress on both recursive calls. When combined with a heuristic to favor less complex atoms, this ensured termination in a reasonable amount of time on our examples. We refer to this as the *simple heuristic*.

Though this produced precise conditions, they were not always very concise. While not an issue from a correctness standpoint, it is desirable for human understanding, and inspection purposes. We thus introduced a new heuristic which significantly improves the *qualitative* aspect of our synthesis algorithm. We found that doing a lookahead (recurses on each predicate one level deep, or *poke*) and computing the number of distinguishing predicates for the two branches as a good indicator of *importance* of the predicate. More precisely, we pick the predicate with lowest sum of remaining number of distinguishing predicates by the two calls. As an aside, those familiar with decision tree learning, might see a connection with the notion of entropy gain. This requires more calls to the SMT solver at each call, but it cuts down the total number of branches to be explored. Also, all individual queries were relatively simple for CVC4. The heuristic converges much faster to the relevant predicates, and produces smaller, concise conditions.

## 6 Case studies

### 6.1 Common Data-Structures

We applied Servois to Set, HashTable, Accumulator, Counter, and Stack. The generated commutativity conditions for these data structures typically combine multiple theories, such as sets, integers and arrays. We used the quantifier-free integer theory in SMTLIB to encode the abstract state and contracts for the Counter and Accumulator ADTs. For Set, the theory of finite sets for tracking elements along with integers to track size; for HashTable, finite sets to track keys, and arrays for the HashMap itself. For Stack, we observed that for the purpose of pairwise commutativity it is sufficient to track the behavior of boundedly many top elements. Since two operations can *at most* either pop the top two elements or push two elements, tracking four elements is sufficient. The full specifications in Servois input format can be found in the Appendix D.

Depending on the pair of methods, the number of predicates generated by PGen were (count after filtering in parentheses): Counter: 25-25 (12-12), Accu-

| | Meth. $m(\bar{x})$ Meth. $n(\bar{y})$ | Simple | Poke | $\varphi_n^m$ generated by Poke heuristic |
|---|---|---|---|---|
| | **Operations** | Qs (time) | Qs (time) | |
| **Counter** | decrement ⋈ decrement | 3 (0.11) | 3 (0.11) | true |
| | increment ▷ decrement | 10 (0.36) | 34 (0.91) | ¬(0 = c) |
| | decrement ▷ increment | 3 (0.11) | 3 (0.12) | true |
| | decrement ⋈ reset | 2 (0.10) | 2 (0.10) | false |
| | decrement ⋈ zero | 6 (0.19) | 26 (0.66) | ¬(1 = c) |
| | increment ⋈ increment | 3 (0.12) | 3 (0.11) | true |
| | increment ⋈ reset | 2 (0.09) | 2 (0.10) | false |
| | increment ⋈ zero | 10 (0.30) | 34 (0.86) | ¬(0 = c) |
| | reset ⋈ reset | 3 (0.11) | 3 (0.11) | true |
| | reset ⋈ zero | 9 (0.24) | 30 (0.69) | 0 = c |
| | zero ⋈ zero | 3 (0.11) | 3 (0.11) | true |
| **Acum.** | increase ⋈ increase | 3 (0.11) | 3 (0.11) | true |
| | increase ⋈ read | 13 (0.31) | 28 (0.63) | c + x1 = c |
| | read ⋈ read | 3 (0.09) | 3 (0.09) | true |
| **Set** | add ⋈ add | 10 (0.40) | 140 (4.47) | [y1 = x1 ∧ y1 ∈ S] ∨ [¬(y1 = x1)] |
| | add ⋈ contains | 10 (0.42) | 122 (3.63) | [x1 ∈ S] ∨ [¬(x1 ∈ S) ∧ ¬(y1 = x1)] |
| | add ⋈ getsize | 6 (0.21) | 31 (0.93) | x1 ∈ S |
| | add ⋈ remove | 6 (0.28) | 66 (2.28) | ¬(y1 = x1) |
| | contains ⋈ contains | 3 (0.18) | 3 (0.16) | true |
| | contains ⋈ getsize | 3 (0.13) | 3 (0.13) | true |
| | contains ⋈ remove | 17 (0.57) | 160 (4.81) | [S\{x1} = {y1}] ∨ [¬(S\{x1} = {y1}) ∧ y1 ∈ {x1} ∧ ...] ∨ [...] |
| | getsize ⋈ getsize | 3 (0.12) | 3 (0.13) | true |
| | getsize ⋈ remove | 13 (0.39) | 37 (1.03) | ¬(y1 ∈ S) |
| | remove ⋈ remove | 21 (0.75) | 192 (6.47) | [S\{y1} = {x1}] ∨ [¬(S\{y1} = {x1}) ∧ y1 ∈ {x1} ∧ ...] ∨ [...] |
| **HashTable** | get ⋈ get | 3 (0.17) | 3 (0.15) | true |
| | get ⋈ haskey | 3 (0.14) | 3 (0.14) | true |
| | put ▷ get | 13 (0.47) | 74 (2.37) | [H[x1=...] = H ∧ y1 ∈ keys] ∨ [¬(H[x1=...] = H) ∧ ¬(y1 = x1)] |
| | get ▷ put | 10 (0.37) | 48 (1.54) | [H[y1] = y2] ∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)] |
| | remove ▷ get | 3 (0.17) | 3 (0.16) | true |
| | get ▷ remove | 13 (0.45) | 40 (1.23) | ¬(y1 = x1) |
| | get ⋈ size | 3 (0.14) | 3 (0.14) | true |
| | haskey ⋈ haskey | 3 (0.14) | 3 (0.14) | true |
| | haskey ⋈ put | 10 (0.37) | 52 (1.63) | [y1 ∈ keys] ∨ [¬(y1 ∈ keys) ∧ ¬(y1 = x1)] |
| | haskey ⋈ remove | 17 (0.59) | 44 (1.36) | [x1 ∈ keys ∧ ¬(y1 = x1)] ∨ [¬(x1 ∈ keys)] |
| | haskey ⋈ size | 3 (0.14) | 3 (0.14) | true |
| | put ⋈ put | 24 (0.97) | 357 (13.50) | [H[y1] = y2 ∧ x2 = H[x1] ∧ ...] ∨ [H[y1] = y2 ∧ x2 = H[x1] ∧ ...] ∨ [...] |
| | put ⋈ remove | 6 (0.30) | 33 (1.26) | ¬(y1 = x1) |
| | put ⋈ size | 6 (0.29) | 23 (0.82) | x1 ∈ keys |
| | remove ⋈ remove | 21 (0.89) | 192 (6.95) | [keys\{x1} = {y1}] ∨ [¬(keys\{x1} = {y1}) ∧ y1 ∈ {x1} ∧ ...] ∨ [...] |
| | remove ⋈ size | 13 (0.45) | 37 (1.13) | ¬(x1 ∈ keys) |
| | size ⋈ size | 3 (0.14) | 3 (0.14) | true |
| **Stack** | clear ⋈ clear | 3 (0.13) | 3 (0.13) | true |
| | clear ⋈ pop | 2 (0.10) | 2 (0.11) | false |
| | clear ⋈ push | 2 (0.12) | 2 (0.11) | false |
| | pop ⋈ pop | 6 (0.23) | 20 (0.62) | nextToTop = top |
| | push ▷ pop | 72 (2.14) | 115 (3.53) | ¬(0 = size) ∧ top = x1 |
| | pop ▷ push | 34 (0.99) | 76 (2.21) | y1 = top |
| | push ⋈ push | 13 (0.58) | 20 (0.72) | y1 = x1 |

**Fig. 2.** Automatically generated commutativity conditions are shown in column $\varphi_n^m$. When right-moverness (▷) conditions are same for a pair of methods, we show them together in one row (⋈). **Qs** specifies the number of SMT queries made, and running time in seconds in parentheses.

```
int warrior, warriorGold, warriorBlock, callback_result, king, kingBlock;

void enter(int val, int sendr, int bk, int rnd) {
   if (val < 50) { send(sendr,val); return; }
   warrior = sendr;
   warriorGold = val;
   warriorBlock = bk // write global variables
   rpc_call("random number generator",__callback,res);
   // Another call to enter() can execute while waiting for RPC
   function __callback(int res_RN) {
      // Most recent writer to warrior now reaps benefit of every callback
      if (modFun(warriorBlock) == res_RN) {
         king = warrior; // winner
         kingBlock = warriorBlock; } } }
```

**Fig. 3.** Simplified code for BlockKing in a C-like language.

mulator: 1-20 (0-20), Set: 17-55 (17-34), HashTable: 18-36 (6-36), Stack: 41-61 (41-42). We did not provide any hints to the algorithm for this case study.

On all our examples, the *simple* heuristic terminated with precise commutativity conditions. In Fig. 2, we give the number of solver queries and total time (in paren.) consumed by this heuristic. The experiments were run on a 2.53 GHz Intel Core 2 Duo machine with 8 GB RAM. The conditions in Fig.2 are those generated by the *poke* heuristic, and interested reader may compare them with the simple heuristic in Apx D.

On the theoretical side, our Choose implementation is fair (satisfies condition 2 of Thm. 2, as Lines 9-10 of the algorithm remove from $\mathcal{P}$ the predicate being tried). From our experiments we conclude that our choice of predicates satisfies condition 1 of Thm. 2.

**Validation.** Although our algorithm is sound, we manually validated the implementation of Servois by examining its output and comparing the generated commutativity conditions with those reported by prior studies. In the case of Accumulator and Counter, our commutativity conditions were identical to those given in [20]. For the Set data structure, the work of [20] used a less precise Set abstraction, so we instead validated against the conditions of [23]. As for HashTable, we validated that our conditions match those by Dimitrov *et al.* [13]. Our implementation is relatively complete because it (and the examples we examined) satisfy the requirements of Thm. 2 as follows.

### 6.2 The BlockKing Ethereum smart contract

We further validated our approach by examining a real-world situation in which non-commutativity opens the door for attacks that exploit interleavings. We examined "smart contracts", which are programs written in the Solidity programming language [4] and executed on the Ethereum blockchain [1]. Eliding many details, smart contracts are like objects, and blockchain participants can invoke methods on these objects. Although the initial intuition is that smart contracts are executed sequentially, practitioners and academics [29] are increasingly realizing that the blockchain is a concurrent environment due to the fact the execution of one actor's *smart contract* can be split across multiple blocks, with other actors' smart contracts interleaved. Therefore, the execution model of the blockchain has been compared to that of concurrent objects [29]. Unfortunately, many smart contracts are not written with this in mind, and attackers can exploit interleavings to their benefit.

As an important example of this, we study the BlockKing smart contract. Fig. 3 provides a simplification of its description, as discussed in [29]. This is a simple game in which the players—each identified by an address sendr—participate by making calls to BlockKing.enter(), sending money val to the contract. (The grey variables are external input that we have lifted to be parameters. bk reflects the caller's current block number and rnd is the outcome of a random number generation, described shortly.) The variables on Line 1 are globals, writable in any call to enter. On Line 4 there is a trivial case when the caller hasn't put enough value into the game, and the money is simply returned. Otherwise, the caller stores their address and value into the shared state. A random number is then generated and, since this requires complex algorithms, it is done via a remote procedure call to a third-party on Line 8, with a callback method provided on Line 10. If the randomly generated number is equal to a modulus of the current block number, then the caller is the winner, and warrior's (caller's) details are stored to king and kingBlock on Line 14.

The fact that random number generation is done via an RPC call means that players' invocations of enter can be interleaved. Moreover, these calls all write sendr and val to shared variables, so the RPC callback will always roll the dice for whomever most recently wrote to warriorBlock. An attacker can use this to leverage other players' investments into the game to increase his/her own chance to win.

We now explore how Servois can aid a programmer in developing a more secure implementation. We observe that, as in traditional parallel programming contexts, if smart contracts are commutative then these interleavings are not problematic. Otherwise, there is cause for concern. To this end, we translated the BlockKing game into Servois format (see Apx. E.2). Servois took 1.4s (on machine with 2.4 GHz Intel Core i5 processor and 8 GB RAM) and yielded the following *non-commutativity* condition for two calls to `enter`:

$$\texttt{enter}(\mathsf{val}_1, \mathsf{sendr}_1, \mathsf{bk}_1, \mathsf{rnd}_1) \;\; \hat{\not\bowtie} \;\; \texttt{enter}(\mathsf{val}_2, \mathsf{sendr}_2, \mathsf{bk}_2, \mathsf{rnd}_2)$$
$$\textit{iff}$$
$$\bigvee \begin{cases} \mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{sendr}_1 \neq \mathsf{sendr}_2 \\ \mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{sendr}_1 = \mathsf{sendr}_2 \wedge \mathsf{val}_1 \neq \mathsf{val}_2 \\ \mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{sendr}_1 = \mathsf{sendr}_2 \wedge \mathsf{val}_1 = \mathsf{val}_2 \wedge \mathsf{bk}_1 \neq \mathsf{bk}_2 \end{cases}$$

This disjunction effectively enumerates cases under which they contract calls *do not* commute. Of particular note is the first disjunct. From this first disjunct, whenever $\mathsf{sendr}_1 \neq \mathsf{sendr}_2$, the calls will not commute. Since in practice $\mathsf{sendr}_1$

will always be different from $\mathsf{sendr}_2$ (two different callers) and $\mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50$ is the non-trivial case, the operations will almost never commute. This should be immediate cause for concern to the developer.

A commutative version of BlockKing would mean that there are no interleavings to be concerned about. Indeed, a simple way to improve commutativity is for each player to write their respective sendr and val to distinct shared state, perhaps via a hashtable keyed on sendr. To this end, we created a new version enter_fixed (see Apx. E.1 and the input to SERVOISin Apx. E.3). SERVOIS generated the following *non*-commutativity condition after 3.5s.

$$\texttt{enter\_fixed}(\mathsf{val}_1, \mathsf{sendr}_1, \mathsf{bk}_1, \mathsf{rnd}_1) \;\not\bowtie\; \texttt{enter\_fixed}(\mathsf{val}_2, \mathsf{sendr}_2, \mathsf{bk}_2, \mathsf{rnd}_2)$$
$$\textit{iff}$$
$$\bigvee \begin{cases} \mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{val}_1 = \mathsf{val}_2 \wedge \mathsf{bk}_1 \neq \mathsf{bk}_2 \wedge \mathsf{sendr}_1 = \mathsf{sendr}_2 \\ \mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{val}_1 \neq \mathsf{val}_2 \wedge \mathsf{sendr}_1 = \mathsf{sendr}_2 \\ \mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{md}(\mathsf{bk}_2) = \mathsf{rnd}_2 \wedge \mathsf{md}(\mathsf{bk}_1) = \mathsf{rnd}_1 \wedge \mathsf{sendr}_1 \neq \mathsf{sendr}_2 \end{cases}$$

In the above non-commutativity condition, md is shorthand for modFun. In the first two disjuncts above, $\mathsf{sendr}_1 = \mathsf{sendr}_2$ which is, again, a case that will not occur in practice. All that remains is the third disjunct where $\mathsf{md}(\mathsf{bk}_2) = \mathsf{rnd}_2$ and $\mathsf{md}(\mathsf{bk}_1) = \mathsf{rnd}_1$. This corresponds to the case where *both* players have won. In this case, it is acceptable for the operations to not commute, because whomever won more recently will store their address and block to the shared state king and kingBlock.

In summary, if we assume that $\mathsf{sendr}_1 \neq \mathsf{sendr}_2$, the non-commutativity of the original version is $\mathsf{val}_1 \geq 50 \vee \mathsf{val}_2 \geq 50$ (very strong). By contrast, the non-commutativity of the fixed version is $\mathsf{val}_1 \geq 50 \wedge \mathsf{val}_2 \geq 50 \wedge \mathsf{md}(\mathsf{bk}_2) = \mathsf{rnd}_2 \wedge \mathsf{md}(\mathsf{bk}_1) = \mathsf{rnd}_1$. We have thus demonstrated that the commutativity (and non-commutativity) conditions generated by SERVOIS can help developers understand the model of interference between two concurrent calls.

## 7 Conclusions and Future Work

This paper demonstrates that it is possible to automatically generate sound and effective commutativity conditions, a task that has so far been done manually or without soundness. Our commutativity conditions are applicable in a variety of contexts including transactional boosting [17], open nested transactions [27], and other non-transactional concurrency paradigms such as race detection [13], parallelizing compilers [28,33], and, as we show, robustness of Ethereum smart contracts [29]. It has been shown that understanding the commutativity of data-structure operations provides a key avenue to improved performance [10] or ease of verification [22,21].

One avenue for future research is how to leverage the internal state of the SMT solver (beyond counterexamples) in order to generate new predicates, perhaps using existing techniques from other contexts [19]. We also plan to explore strategies to compute commutativity conditions directly from the program's code, without the need for an intermediate abstract representation [33].

## References


1. Ethereum. https://ethereum.org/.
2. Servois homepage. http://cs.nyu.edu/~kshitij/projects/servois.
3. Servois source code. https://github.com/kbansal/servois.
4. Solidity programming language. https://solidity.readthedocs.io/en/develop/.
5. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, May 1991.
6. 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)*, pages 241–252. ACM, 2009.
7. M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, 2005.
8. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806, pages 171–177. Springer, July 2011.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.
10. 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 Trans. Comput. Syst.*, 32(4):10, 2015.
11. B. Cook and E. Koskinen. Making prophecies with decision predicates. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 399–410, 2011.
12. 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*, PODC '17, pages 303–312, New York, NY, USA, 2017. ACM.
13. D. Dimitrov, V. Raychev, M. T. Vechev, and E. Koskinen. Commutativity race detection. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 33. ACM, 2014.
14. G. W. Ernst and W. F. Ogden. Specification of abstract data types in modula. *ACM Trans. Program. Lang. Syst.*, 2(4):522–543, Oct. 1980.
15. L. Flon and J. Misra. A unified approach to the specification and verification of abstract data types. In *Proc. Specifications of Reliable Software Conf., IEEE Computer Society*, 1979.
16. 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*, pages 307–323, 2015.
17. M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'08)*, 2008.
18. C. A. R. Hoare. Software pioneers. In M. Broy and E. Denert, editors, *Software Pioneers*, chapter Proof of Correctness of Data Representations, pages 385–396. Springer-Verlag New York, Inc., New York, NY, USA, 2002.
19. Y. Hu, C. Barrett, and B. Goldberg. Theory and algorithms for the generation and validation of speculative loop optimizations. In *Proceedings of the 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM '04)*, pages 281–289. IEEE Computer Society, Sept. 2004.

20. D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 528–541. ACM, 2011.
21. E. Koskinen and M. J. Parkinson. The push/pull model of transactions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, Portland, OR, USA, June, 2015*, 2015.
22. E. Koskinen, M. J. Parkinson, and M. Herlihy. Coarse-grained transactions. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 19–30. ACM, 2010.
23. M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 542–555. ACM, 2011.
24. K. R. M. Leino. Specifying and verifying programs in spec#. In *Proceedings of the 6th International Perspectives of Systems Informatics, Andrei Ershov Memorial Conference, PSI 2006*, page 20, 2006.
25. R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
26. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.
27. Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007*, pages 68–78. ACM, 2007.
28. M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, November 1997.
29. I. Sergey and A. Hobor. A concurrent perspective on smart contracts. In *1st Workshop on Trusted Smart Contracts*, 2017.
30. A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation PLDI 2008*, pages 136–148, 2008.
31. O. Tripp, R. Manevich, J. Field, and M. Sagiv. Janus: Exploiting parallelism via hindsight. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 145–156, New York, NY, USA, 2012. ACM.
32. O. Tripp, G. Yorsh, J. Field, and M. Sagiv. Hawkeye: Effective discovery of dataflow impediments to parallelization. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 207–224, New York, NY, USA, 2011. ACM.
33. O. Tripp, G. Yorsh, J. Field, and M. Sagiv. HAWKEYE: effective discovery of dataflow impediments to parallelization. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*, pages 207–224, 2011.
34. M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 125–135, 2008.
35. M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 327–338, 2010.
36. C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 382–396, 2008.

# Appendix

## A Lifting transition systems

**Definition 2 (Lifted transition function).** *For* $\mathfrak{T} = (\Sigma, A, (\!|\bullet|\!))$*, we lift* $\mathfrak{T}$ *to* $\hat{\mathfrak{T}} = (\hat{\Sigma}, A, (\!|\!|\bullet|\!|\!))$ *where* $\hat{\Sigma} = \Sigma \cup \{Err\}$*,* $Err \notin \Sigma$*, and* $(\!|\!|\alpha|\!|\!) : \hat{\Sigma} \to \hat{\Sigma}$*, as:*

$$(\!|\!|\alpha|\!|\!)\hat{\sigma} \equiv \begin{cases} Err & \text{if } \hat{\sigma} = Err \\ (\!|\alpha|\!)\hat{\sigma} & \text{if } \hat{\sigma} \in \mathbf{dom}((\!|\alpha|\!)) \\ Err & \text{otherwise} \end{cases}$$

Intuitively, $(\!|\!|\alpha|\!|\!)$ wraps $(\!|\alpha|\!)$ so that $Err$ loops back to $Err$, and the (potentially partial) $(\!|\alpha|\!)$ is made to be total by mapping elements to $Err$ when they are undefined in $(\!|\alpha|\!)$. Note that it is not necessary to lift the actions (or, indeed, the methods), but only the states and transition function. Once lifted, for a given state $\hat{\sigma}_0$, the question of *some* successor state becomes equivalent to *all* successor states because there is exactly one successor state.

**Abstraction.** Pre-/post-conditions $(Pre_m, Post_m)$ are suitable for specifications of potentially partial transition systems. One can translate these into a new pair $(\widehat{Pre_m}, \widehat{Post_m})$ that induces a corresponding lifted transition system that (i) is total and (ii) remains deterministic. These lifted specifications have types over lifted state spaces:

$$[\![\widehat{Pre_m}]\!] : \bar{x} \to \hat{\Sigma} \to \mathbb{B} \qquad [\![\widehat{Post_m}]\!] : \bar{x} \to \bar{r} \to \hat{\Sigma} \to \hat{\Sigma} \to \mathbb{B}$$

Our implementation performs this lifting via translation from $(Pre_m, Post_m)$ to:

$$\widehat{Pre_m}(\bar{x}, \hat{\sigma}) \equiv \ \mathsf{true}$$

$$\widehat{Post_m}(\bar{x}, \bar{r}, \hat{\sigma}, \hat{\sigma}') \equiv \ \bigvee \begin{cases} \hat{\sigma} = Err \wedge \hat{\sigma}' = Err \\ \hat{\sigma} \neq Err \wedge Pre_m(\bar{x}, \hat{\sigma}) \wedge \hat{\sigma}' \neq Err \wedge Post_m(\bar{x}, \bar{r}, \hat{\sigma}, \hat{\sigma}') \\ \hat{\sigma} \neq Err \wedge \neg Pre_m(\bar{x}, \hat{\sigma}) \wedge \hat{\sigma}' = Err \end{cases}$$

(We abuse notation, giving $\hat{\sigma}$ as an argument to $Pre_m$, etc.) It is easy to see that the lifted transition system induced by this translation $(\hat{\Sigma}, (\!|\!|\bullet|\!|\!))$ is of the form given in Def. 2. In the Appendix, we show how our tool transforms a counter specification (Apx. D.1) into an equivalent lifted version (Apx. D.2). Notice that this specification is now a total transition system.

We use the notation $\hat{\bowtie}$ to mean $\bowtie$ but over lifted transition system $\hat{\mathfrak{T}}$. Since $\hat{\bowtie}$ is over total, determinsitic transition functions, $\alpha_1 \mathrel{\hat{\bowtie}} \alpha_2$ is equivalent to:

$$\forall \hat{\sigma}_0.\ \hat{\sigma}_0 \neq Err \ \Rightarrow \ (\!|\!|\alpha_2|\!|\!)\ (\!|\!|\alpha_1|\!|\!)\ \hat{\sigma}_0 = (\!|\!|\alpha_1|\!|\!)\ (\!|\!|\alpha_2|\!|\!)\ \hat{\sigma}_0 \qquad (1)$$

The equivalence above is in terms of state equality. Importantly, this is a universally quantified formula that translates to a ground satisfiability check in an SMT solver (modulo the theories used to model the data structure). In our refinement algorithm (Sec. 4), we will use this format to check whether candidate logical formulae describe commutative subregions.

**Lemma 2.** $m \bowtie n$ *if and only if* $m \mathrel{\hat{\bowtie}} n$*. (*All proofs in Apx. C.*)*

## B Right-/Left-movers

**Definition 3 (Action right-mover [25]).** *We say that an action* $\alpha_1$ moves to the right of *action* $\alpha_2$ *commute, denoted* $\alpha_1 \triangleright \alpha_2$*, provided that* $(\!|\alpha_2|\!) \circ (\!|\alpha_1|\!) \subseteq (\!|\alpha_1|\!) \circ (\!|\alpha_2|\!)$.

Note that left-movers can be defined as right-movers, but with arguments swapped.

**Definition 4 (Method right-mover).** *For* $m$ *and* $n$*,*

$$m \;\triangleright\; n \;\;\equiv\;\; \forall \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\ m(\bar{x})/\bar{r} \triangleright n(\bar{y})/\bar{s}$$

A *logical right-mover condition* denoted $\boldsymbol{\Psi}_m^n$ has the same type as a commutativity condition and, again $[\![\boldsymbol{\Psi}_m^n]\!]$ denotes interpretations of $\boldsymbol{\Psi}_m^n$. Moreover, we say that $\boldsymbol{\Psi}_m^n$ is a right-mover condition for $m$ and $n$ provided that $\forall \sigma_0\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\ [\![\boldsymbol{\Psi}_m^n]\!]\ \sigma_0\ (m(\bar{x})/\bar{r})\ (n(\bar{y})/\bar{s}) = \mathsf{true} \Rightarrow m \;\triangleright\; n$ and similar for a *non*-right-mover condition.

**Checking whether** $H_m^n \Rightarrow m \;\hat{\triangleright}\; n$**.** After performing the lifting transformation, we again are able to reduce the question of whether a formula $H_m^n$ is a right-mover condition to a validity check that does not introduce quantifier alternation.

$$\mathsf{valid} \left( \begin{array}{l} \forall \hat{\sigma}_0\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}. \\ \quad \varphi_m^n(\hat{\sigma}_0, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \;\Rightarrow \\ \quad \hat{\sigma}_0 \neq Err \;\Rightarrow \\ \quad (\!|n(\bar{y})/\bar{s}|\!)\ (\!|m(\bar{x})/\bar{r}|\!)\ \hat{\sigma}_0 \neq Err \;\Rightarrow \\ \quad (\!|n(\bar{y})/\bar{s}|\!)\ (\!|m(\bar{x})/\bar{r}|\!)\ \hat{\sigma}_0 = (\!|m(\bar{x})/\bar{r}|\!)\ (\!|n(\bar{y})/\bar{s}|\!)\ \hat{\sigma}_0. \end{array} \right)$$

Notice that this is a generalization of the validity check for commutativity.

## C Proofs

**Lemma 2.**

*Proof.* Follows from classical reasoning, functional extensionality and case analysis on totality-vs-partiality.

**Theorem 1.**

*Proof.* By induction. Initially, $\mathsf{false}$ is a suitable condition for when commutativity holds. $\mathsf{false}$ is also a suitable condition under which commutativity does not hold. At each iteration, $\varphi$ or $\tilde{\varphi}$ may be updated (not both, but for soundness this does not matter). Consider $\varphi$. It must also be the case that $(\varphi \vee H) \Rightarrow m \hat{\bowtie} n$ because we know that $\varphi \Rightarrow m \hat{\bowtie} n$ (from the previous iteration) and that $H \Rightarrow m \hat{\bowtie} n$ (from the $\mathsf{valid}$ check at Line 2). Analogous reasoning for $\tilde{\varphi}$.

**Lemma 1.**

*Proof.* The recursive calls of the Refine algorithm induce a *binary* tree $T$, where nodes are labeled by the conjunction of predicates. If Refine terminates, then $T$ is finite, and each node is labeled with a finite conjunction $p_0 \wedge ... \wedge p_n$.

*Claim.* The disjunction of all leaf node labels is valid. *Pf.* By induction on the tree. Base case: a single-node tree has label $\mathsf{true}$. Inductive case: for every new node created, labeled with a new conjunct $... \wedge p$, there is a sibling node with label $... \wedge \neg p$.

Each leaf node of tree $T$, labeled with conjunction $\gamma$, arises from Refine reaching a base case where, by construction, the conjunction $\gamma$ is disjunctively added to either $\varphi$ or $\tilde{\varphi}$. Since Refine terminates, *all* conjunctions are added to either $\varphi$ or $\tilde{\varphi}$, and thus $\varphi \vee \tilde{\varphi}$ must be valid.

**Theorem 2.**

*Proof.* By contradiction. As in the proof for Lemma 1, we represent the algorithm's execution as a binary tree $T$, induced by the recursive Refine calls, whose nodes are labeled by the conjunction of predicates (Lines 9 and 10 in Algorithm 1). Assume there exists an infinite path along $T$, and let its respective labels be $\pi = p_0, p_0 \wedge p_1, p_0 \wedge p_1 \wedge p_2, ....$

*Claim.* There is no finite prefix of $\pi$ that contains all the predicates $\psi_i$. *Pf.* Had there been such a prefix $\varpi$, by the expressiveness assumption the running condition $H$ would satisfy one of the validity checks at lines 2 and 4 within, or immediately after, $\varpi$. This is because $H$ would be equal to, or stronger than, the conjunction of the predicates $\psi_i$. This would have made $\pi$ finite, as $\pi$ is extended only if both of the validity checks fail, where we assume $\pi$ is infinite.

By the above claim, at least one of the predicates $\psi_i$ is not contained in any finite prefix of $\pi$. This contradicts the fairness assumption, whereby any predicate $p \in \mathcal{P}$ is chosen after finitely many Choose invocations (provided the algorithm hasn't terminated).

# D Data Structure Representations

## D.1 Counter

```
# Counter data structure's abstract definition

name: counter

state:
  - name: contents
    type: Int

states_equal:
  definition: (= contents_1 contents_2)

methods:
  - name: increment
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 0)
    ensures: |
      (and (= contents_new (+ contents 1))
           (= result true))
    terms:
      Int: [contents, 1, (+ contents 1)]
  - name: decrement
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 1)
    ensures: |
      (and (= contents_new (- contents 1))
           (= result true))
    terms:
      Int: [contents, 1, (- contents 1), 0]
  - name: reset
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 0)
    ensures: |
      (and (= contents_new 0)
           (= result true))
    terms:
      Int: [contents, 0]
  - name: zero
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 0)
    ensures: |
      (and (= contents_new contents)
           (= result (= contents 0)))
    terms:
      Int: [contents, 0]

predicates:
  - name: "="
```

type: [Int, Int]

- `decrement` ⋈ `decrement`
  Simple:
  true
  Poke:
  true
- `increment` ▷ `decrement`
  Simple:
  [1 = `contents`]
  ∨ [¬(1 = `contents`) ∧ ¬(0 = `contents`)]
  Poke:
  ¬(0 = `contents`)
- `decrement` ▷ `increment`
  Simple:
  true
  Poke:
  true
- `decrement` ⋈ `reset`
  Simple:
  false
  Poke:
  false
- `decrement` ⋈ `zero`
  Simple:
  ¬(1 = `contents`)
  Poke:
  ¬(1 = `contents`)
- `increment` ⋈ `increment`
  Simple:
  true
  Poke:
  true
- `increment` ⋈ `reset`
  Simple:
  false
  Poke:
  false
- `increment` ⋈ `zero`
  Simple:
  [1 = `contents`]
  ∨ [¬(1 = `contents`) ∧ ¬(0 = `contents`)]
  Poke:
  ¬(0 = `contents`)
- `reset` ⋈ `reset`
  Simple:
  true
  Poke:
  true
- `reset` ⋈ `zero`
  Simple:
  ¬(1 = `contents`) ∧ 0 = `contents`
  Poke:
  0 = `contents`
- `zero` ⋈ `zero`
  Simple:
  true
  Poke:
  true

## D.2 Counter (lifted, auto-generated)


```
methods:
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not err_new) (>= contents 0)\n\
    \ (and (= contents_new (+ contents 1))\n      (= result true))\n)\n    (and (not\
    \ err) err_new (not (>= contents 0)\n)))"
  name: increment
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 1
    - (+ contents 1)
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not err_new) (>= contents 1)\n\
    \ (and (= contents_new (- contents 1))\n      (= result true))\n)\n    (and (not\
    \ err) err_new (not (>= contents 1)\n)))"
  name: decrement
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 1
    - (- contents 1)
    - 0
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not err_new) (>= contents 0)\n\
    \ (and (= contents_new 0)\n      (= result true))\n)\n    (and (not err) err_new\
    \ (not (>= contents 0)\n)))"
  name: reset
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 0
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not err_new) (>= contents 0)\n\
    \ (and (= contents_new contents)\n      (= result (= contents 0)))\n)\n    (and\
    \ (not err) err_new (not (>= contents 0)\n)))"
  name: zero
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 0
name: counter
predicates:
- name: '='
  type:
  - Int
  - Int
state:
- name: contents
  type: Int
- name: err
```

```
  type: Bool
states_equal:
  definition: '(or (and err_1 err_2) (and (not err_1) (not err_2)

    (= contents_1 contents_2)

    ))'
```

## D.3 Accumulator

```
# Accumulator abstract definition

name: accumulator

state:
  - name: contents
    type: Int

options:


states_equal:
  definition: (= contents_1 contents_2)

methods:
  - name: increase
    args:
      - name: n
        type: Int
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (and (= contents_new (+ contents n))
           (= result true))
    terms:
      Int: [$1, contents, (+ contents $1)]
  - name: read
    args: []
    return:
      - name: result
        type: Int
    requires: |
      true
    ensures: |
      (and (= contents_new contents)
           (= result contents))
    terms:
      Int: [contents]

predicates:
  - name: "="
    type: [Int, Int]
```

- increase ⋈ increase
  Simple:
  true
  Poke:
  true

- increase ⋈ read
  Simple:
  [x1 = contents ∧ contents + x1 = contents]
  ∨ [¬(x1 = contents) ∧ contents + x1 = contents]
  Poke:
  contents + x1 = contents
- read ⋈ read
  Simple:
  true
  Poke:
  true

## D.4 Set

```
name: set

preamble: |
  (declare-sort E 0)

state:
  - name: S
    type: (Set E)
  - name: size
    type: Int

states_equal:
  definition: (and (= S_1 S_2) (= size_1 size_2))

methods:
  - name: add
    args:
      - name: v
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (ite (member v S)
           (and (= S_new S)
                (= size_new size)
                (not result))
           (and (= S_new (union S (singleton v)))
                (= size_new (+ size 1))
                result))
    terms:
      E: [$1]
      Int: [size, 1, (+ size 1)]
      (Set E): [S, (singleton $1), (union S (singleton $1))]
  - name: remove
    args:
      - name: v
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (ite (member v S)
           (and (= S_new (setminus S (singleton v)))
                (= size_new (- size 1))
                result)
           (and (= S_new S)
                (= size_new size)
                (not result)))
```

```
    terms:
      E: [$1]
      Int: [size, 1, (- size 1)]
      (Set E): [S, (singleton $1), (setminus S (singleton $1))]
  - name: contains
    args:
      - name: v
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (and (= S_new S)
           (= size_new size)
           (= (member v S) result))
    terms:
      E: [$1]
      Int: [size]
      (Set E): [S, (singleton $1), (setminus S (singleton $1))]
  - name: getsize
    args: []
    return:
      - name: result
        type: Int
    requires: |
      true
    ensures: |
      (and (= S_new S)
           (= size_new size)
           (= size result))
    terms:
      Int: [size]

predicates:
  - name: "="
    type: [Int, Int]
  - name: "="
    type: [E, E]
  - name: "="
    type: [(Set E), (Set E)]
  - name: "member"
    type: [E, (Set E)]
```

- $\texttt{add} \bowtie \texttt{add}$
  Simple:
  $[\texttt{y1} = \texttt{x1} \wedge \texttt{y1} \in \texttt{S}]$
  $\vee\ [\neg(\texttt{y1} = \texttt{x1})]$
  Poke:
  $[\texttt{y1} = \texttt{x1} \wedge \texttt{y1} \in \texttt{S}]$
  $\vee\ [\neg(\texttt{y1} = \texttt{x1})]$
- $\texttt{add} \bowtie \texttt{contains}$
  Simple:
  $[\texttt{y1} = \texttt{x1} \wedge \texttt{y1} \in \texttt{S}]$
  $\vee\ [\neg(\texttt{y1} = \texttt{x1})]$
  Poke:
  $[\texttt{x1} \in \texttt{S}]$
  $\vee\ [\neg(\texttt{x1} \in \texttt{S}) \wedge \neg(\texttt{y1} = \texttt{x1})]$
- $\texttt{add} \bowtie \texttt{getsize}$
  Simple:
  $\texttt{x1} \in \texttt{S}$
  Poke:
  $\texttt{x1} \in \texttt{S}$
- $\texttt{add} \bowtie \texttt{remove}$
  Simple:
  $\neg(\texttt{y1} = \texttt{x1})$
  Poke:
  $\neg(\texttt{y1} = \texttt{x1})$

– contains ⋈ contains
Simple:
true
Poke:
true
– contains ⋈ getsize
Simple:
true
Poke:
true
– contains ⋈ remove
Simple:
$[\mathtt{y1} = \mathtt{x1} \wedge 1 = \mathtt{size} \wedge \neg(\mathtt{y1} \in \mathtt{S})]$
$\vee [\mathtt{y1} = \mathtt{x1} \wedge \neg(1 = \mathtt{size}) \wedge \neg(\mathtt{y1} \in \mathtt{S})]$
$\vee [\neg(\mathtt{y1} = \mathtt{x1})]$
Poke:
$[\mathtt{S} \backslash \{\mathtt{x1}\} = \{\mathtt{y1}\}]$
$\vee [\neg(\mathtt{S} \backslash \{\mathtt{x1}\} = \{\mathtt{y1}\}) \wedge \mathtt{y1} \in \{\mathtt{x1}\} \wedge \neg(\mathtt{y1} \in \mathtt{S})]$
$\vee [\neg(\mathtt{S} \backslash \{\mathtt{x1}\} = \{\mathtt{y1}\}) \wedge \neg(\mathtt{y1} \in \{\mathtt{x1}\})]$
– getsize ⋈ getsize
Simple:
true
Poke:
true
– getsize ⋈ remove
Simple:
$[1 = \mathtt{size} \wedge \neg(\mathtt{y1} \in \mathtt{S})]$
$\vee [\neg(1 = \mathtt{size}) \wedge \neg(\mathtt{y1} \in \mathtt{S})]$
Poke:
$\neg(\mathtt{y1} \in \mathtt{S})$
– remove ⋈ remove
Simple:
$[1 = \mathtt{size} \wedge \mathtt{y1} = \mathtt{x1} \wedge \neg(\mathtt{y1} \in \mathtt{S})]$
$\vee [1 = \mathtt{size} \wedge \neg(\mathtt{y1} = \mathtt{x1})]$
$\vee [\neg(1 = \mathtt{size}) \wedge \mathtt{y1} = \mathtt{x1} \wedge \neg(\mathtt{y1} \in \mathtt{S})]$
$\vee [\neg(1 = \mathtt{size}) \wedge \neg(\mathtt{y1} = \mathtt{x1})]$
Poke:
$[\mathtt{S} \backslash \{\mathtt{y1}\} = \{\mathtt{x1}\}]$
$\vee [\neg(\mathtt{S} \backslash \{\mathtt{y1}\} = \{\mathtt{x1}\}) \wedge \mathtt{y1} \in \{\mathtt{x1}\} \wedge \neg(\mathtt{y1} \in \mathtt{S})]$
$\vee [\neg(\mathtt{S} \backslash \{\mathtt{y1}\} = \{\mathtt{x1}\}) \wedge \neg(\mathtt{y1} \in \{\mathtt{x1}\})]$

## D.5 HashTable

```
# Hash table data structure's abstract definition

name: HashTable

preamble: |
  (declare-sort E 0)
  (declare-sort F 0)

state:
  - name: keys
    type: (Set E)
  - name: H
    type: (Array E F)
  - name: size
    type: Int

states_equal:
  definition: |
    (and (= keys_1 keys_2)
         (= H_1 H_2)
         (= size_1 size_2))

methods:
  - name: haskey
```

```
  args:
    - name: k0
      type: E
  return:
    - name: result
      type: Bool
  requires: |
    true
  ensures: |
    (and (= keys_new keys)
         (= H_new H)
         (= size_new size)
         (= (member k0 keys) result)
     )
  terms:
    Int: [size]
    E: [$1]
    (Set E): [keys]
    (Array E F): [H]
- name: remove
  args:
    - name: v
      type: E
  return:
    - name: result
      type: Bool
  requires: |
    true
  ensures: |
    (ite (member v keys)
         (and (= keys_new (setminus keys (singleton v)))
              (= size_new (- size 1))
              (= H_new H)
              result)
         (and (= keys_new keys)
              (= size_new size)
              (= H_new H)
              (not result)))
  terms:
    Int: [size, 1, (- size 1)]
    E: [$1]
    (Set E): [keys, (singleton $1), (setminus keys (singleton $1))]
    (Array E F): [H]
- name: put
  args:
    - name: k0
      type: E
    - name: v0
      type: F
  return:
    - name: result
      type: Bool
  requires: |
    true
  ensures: |
    (ite (member k0 keys)
         (and (= keys_new keys)
              (= size_new size)
              (ite (= v0 (select H k0))
                   (and (not result)
                        (= H_new H))
                   (and result
                        (= H_new (store H k0 v0)))))
         (and (= keys_new (insert k0 keys))
              (= size_new (+ size 1))
              result
              (= H_new (store H k0 v0))))
  terms:
```

```
      Int: [size, 1, (+ size 1)]
      E: [$1]
      F: [$2, (select H $1), ]
      (Set E): [keys, (insert $1 keys)]
      (Array E F): [H, (store H $1 $2)]
  - name: get
    args:
      - name: k0
        type: E
    return:
      - name: result
        type: F
    requires: |
      (member k0 keys)
    ensures: |
      (and (= keys_new keys)
           (= H_new H)
           (= size_new size)
           (= (select H k0) result)
           )
    terms:
      Int: [size]
      E: [$1]
      F: [(select H $1)]
      (Set E): [keys]
      (Array E F): [H]
  - name: size
    args: []
    return:
      - name: result
        type: Int
    requires: |
      true
    ensures: |
      (and (= keys_new keys)
           (= H_new H)
           (= size_new size)
           (= size result))
    terms:
      Int: [size]
      (Set E): [keys]
      (Array E F): [H]

predicates:
  - name: "="
    type: [Int, Int]
  - name: "="
    type: [E, E]
  - name: "="
    type: [F, F]
  - name: "="
    type: [(Set E), (Set E)]
  - name: "="
    type: [(Array E F), (Array E F)]
  - name: "member"
    type: [E, (Set E)]
```

- get ⋈ get
  Simple:
  true
  Poke:
  true
- get ⋈ haskey
  Simple:
  true
  Poke:
  true

- put ▷ get
  Simple:
  [x2 = H[y1] ∧ y1 ∈ keys]
  ∨ [¬(x2 = H[y1]) ∧ ¬(y1 = x1)]
  Poke:
  [H[x1=x2] = H ∧ y1 ∈ keys]
  ∨ [¬(H[x1=x2] = H) ∧ ¬(y1 = x1)]
- get ▷ put
  Simple:
  [H[y1] = y2]
  ∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)]
  Poke:
  [H[y1] = y2]
  ∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)]
- remove ▷ get
  Simple:
  true
  Poke:
  true
- get ▷ remove
  Simple:
  [1 = size ∧ ¬(y1 = x1)]
  ∨ [¬(1 = size) ∧ ¬(y1 = x1)]
  Poke:
  ¬(y1 = x1)
- get ⋈ size
  Simple:
  true
  Poke:
  true
- haskey ⋈ haskey
  Simple:
  true
  Poke:
  true
- haskey ⋈ put
  Simple:
  [y1 = x1 ∧ y1 ∈ keys]
  ∨ [¬(y1 = x1)]
  Poke:
  [y1 ∈ keys]
  ∨ [¬(y1 ∈ keys) ∧ ¬(y1 = x1)]
- haskey ⋈ remove
  Simple:
  [y1 = x1 ∧ 1 = size ∧ ¬(y1 ∈ keys)]
  ∨ [y1 = x1 ∧ ¬(1 = size) ∧ ¬(y1 ∈ keys)]
  ∨ [¬(y1 = x1)]
  Poke:
  [x1 ∈ keys ∧ ¬(y1 = x1)]
  ∨ [¬(x1 ∈ keys)]
- haskey ⋈ size
  Simple:
  true
  Poke:
  true
- put ⋈ put
  Simple:
  [x2 = y2 ∧ x2 = H[y1] ∧ y1 ∈ keys]
  ∨ [x2 = y2 ∧ x2 = H[y1] ∧ ¬(y1 ∈ keys) ∧ ¬(y1 = x1)]
  ∨ [x2 = y2 ∧ ¬(x2 = H[y1]) ∧ ¬(y1 = x1)]
  ∨ [¬(x2 = y2) ∧ ¬(y1 = x1)]
  Poke:
  [H[y1] = y2 ∧ x2 = H[x1] ∧ size + 1 = 1 ∧ y1 ∈ keys]
  ∨ [H[y1] = y2 ∧ x2 = H[x1] ∧ size + 1 = 1 ∧ ¬(y1 ∈ keys) ∧ ¬(y1 = x1)]
  ∨ [H[y1] = y2 ∧ x2 = H[x1] ∧ ¬(size + 1 = 1) ∧ x1 ∈ keys]
  ∨ [H[y1] = y2 ∧ x2 = H[x1] ∧ ¬(size + 1 = 1) ∧ ¬(x1 ∈ keys) ∧ ¬(y1 = x1)]
  ∨ [H[y1] = y2 ∧ ¬(x2 = H[x1]) ∧ ¬(y1 = x1)]
  ∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)]

- put ⋈ remove
  Simple:
  ¬(y1 = x1)
  Poke:
  ¬(y1 = x1)
- put ⋈ size
  Simple:
  x1 ∈ keys
  Poke:
  x1 ∈ keys
- remove ⋈ remove
  Simple:
  [1 = size ∧ y1 = x1 ∧ ¬(y1 ∈ keys)]
  ∨ [1 = size ∧ ¬(y1 = x1)]
  ∨ [¬(1 = size) ∧ y1 = x1 ∧ ¬(y1 ∈ keys)]
  ∨ [¬(1 = size) ∧ ¬(y1 = x1)]
  Poke:
  [keys\{x1} = {y1}]
  ∨ [¬(keys\{x1} = {y1}) ∧ y1 ∈ {x1} ∧ ¬(y1 ∈ keys)]
  ∨ [¬(keys\{x1} = {y1}) ∧ ¬(y1 ∈ {x1})]
- remove ⋈ size
  Simple:
  [1 = size ∧ ¬(x1 ∈ keys)]
  ∨ [¬(1 = size) ∧ ¬(x1 ∈ keys)]
  Poke:
  ¬(x1 ∈ keys)
- size ⋈ size
  Simple:
  true
  Poke:
  true

## D.6 Stack

```
# Stack definition

name: stack

preamble: |
  (declare-sort E 0)

state:
  - name: size
    type: Int
  - name: top
    type: E
  - name: nextToTop
    type: E
  - name: secondToTop
    type: E
  - name: thirdToTop
    type: E

states_equal:
  definition:
    (and (= size_1 size_2)
         (or (= size_1 0)
             (and (= size_1 1) (= top_1 top_2))
             (and (= top_1 top_2) (= nextToTop_1 nextToTop_2))))

methods:
  - name: push
    args:
      - name: v
        type: E
    return:
```

```yaml
      - name: result
        type: Bool
    requires: |
      (>= size 0)
    ensures: |
      (and (= size_new (+ size 1))
           (= top_new v)
           (= nextToTop_new top)
           (= secondToTop_new nextToTop)
           (= thirdToTop_new secondToTop)
           (= result true))
    terms:
      Int: [size, 1, (+ size 1)]
      E: [top, nextToTop, secondToTop, thirdToTop, $1]
  - name: pop
    args: []
    return:
      - name: result
        type: E
    requires: |
      (>= size 1)
    ensures: |
      (and (= size_new (- size 1))
           (= result top)
           (= top_new nextToTop)
           (= nextToTop_new secondToTop)
           (= secondToTop_new thirdToTop))
    terms:
      Int: [size, 1, (- size 1), 0]
      E: [top, nextToTop, secondToTop, thirdToTop]
  - name: clear
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= size 0)
    ensures: |
      (and (= size_new 0)
           (= result true))
    terms:
      Int: [size, 0]
      E: [top, nextToTop, secondToTop, thirdToTop]

predicates:
  - name: "="
    type: [Int, Int]
  - name: "="
    type: [E, E]
```

- `clear` ⋈ `clear`
  Simple:
  true
  Poke:
  true
- `clear` ⋈ `pop`
  Simple:
  false
  Poke:
  false
- `clear` ⋈ `push`
  Simple:
  false
  Poke:
  false
- `pop` ⋈ `pop`
  Simple:

nextToTop = top
Poke:
nextToTop = top
- push ▷ pop
Simple:
[1 = size ∧ nextToTop = top ∧ nextToTop = thirdToTop ∧ nextToTop = x1]
∨ [1 = size ∧ nextToTop = top ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = x1]
∨ [1 = size ∧ ¬(nextToTop = top) ∧ nextToTop = thirdToTop ∧ nextToTop = secondToTop ∧ top = x1]
∨ [1 = size ∧ ¬(nextToTop = top) ∧ nextToTop = thirdToTop ∧ ¬(nextToTop = secondToTop) ∧ top = x1]
∨ [1 = size ∧ ¬(nextToTop = top) ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = secondToTop ∧ top = x1]
∨ [1 = size ∧ ¬(nextToTop = top) ∧ ¬(nextToTop = thirdToTop) ∧ ¬(nextToTop = secondToTop) ∧ top = x1]
∨ [¬(1 = size) ∧ ¬(0 = size) ∧ nextToTop = thirdToTop ∧ nextToTop = secondToTop ∧ top = x1]
∨ [¬(1 = size) ∧ ¬(0 = size) ∧ nextToTop = thirdToTop ∧ ¬(nextToTop = secondToTop) ∧ top = x1]
∨ [¬(1 = size) ∧ ¬(0 = size) ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = secondToTop ∧ top = x1]
∨ [¬(1 = size) ∧ ¬(0 = size) ∧ ¬(nextToTop = thirdToTop) ∧ ¬(nextToTop = secondToTop) ∧ top = x1]
Poke:
¬(0 = size) ∧ top = x1
- pop ▷ push
Simple:
[nextToTop = y1 ∧ nextToTop = top]
∨ [¬(nextToTop = y1) ∧ nextToTop = thirdToTop ∧ nextToTop = secondToTop ∧ y1 = top]
∨ [¬(nextToTop = y1) ∧ nextToTop = thirdToTop ∧ ¬(nextToTop = secondToTop) ∧ y1 = top]
∨ [¬(nextToTop = y1) ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = secondToTop ∧ y1 = top]
∨ [¬(nextToTop = y1) ∧ ¬(nextToTop = thirdToTop) ∧ ¬(nextToTop = secondToTop) ∧ y1 = top]
Poke:
y1 = top
- push ⋈ push
Simple:
[thirdToTop = y1 ∧ thirdToTop = x1]
∨ [¬(thirdToTop = y1) ∧ y1 = x1]
Poke:
y1 = x1

# E textsfBlockKing

## E.1 Fixed version of **BlockKing**

The following is simplified pseudo-code for a *fixed* version of BlockKing in a C-like language.

```
struct storage {
   int warrior;
   int warriorGold;
   int warriorBlock;
   int res;
};

hashtable[int,struct storage] scratch = ...;
int king, kingBlock;

void enter_fixed(int val, int sendr, int bk, int rnd) {
   if (val < 50) { send(sendr,val); return; }
   scratch[sendr].warrior = sendr;
   scratch[sendr].warriorGold = val;
   scratch[sendr].warriorBlock = bk;
   // callback generates the random number in scratch[sendr]
   rpc_call("random number generator",__callback,scratch[sendr].res);
   function __callback() {
      if (modFun(scratch[sendr].warriorBlock) == scratch[sendr].res) {
         king = scratch[sendr].warrior; // winner
         kingBlock = scratch[sendr].warriorBlock; } } }
```

## E.2 BlockKing: YML representation

```
name: blockking

preamble: |
  (declare-fun modFn (Int) Int)

state:
  - name: warrior
    type: Int
  - name: warriorGold
    type: Int
  - name: warriorBlock
    type: Int
  - name: king
    type: Int
  - name: kingBlock
    type: Int

methods:
  - name: enter
    args:
      - name: msg_value
        type: Int
      - name: msg_sender
        type: Int
      - name: block_number
        type: Int
      - name: random
        type: Int
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (and result
```

```
      (ite (< msg_value 50)
           (states_equal warrior warriorGold warriorBlock
                         king kingBlock err
                         warrior_new warriorGold_new warriorBlock_new
                         king_new kingBlock_new err_new)
           (and (= warrior_new msg_sender)
                (= warriorGold_new msg_value)
                (= warriorBlock_new block_number)
                (ite (= (modFn warriorBlock_new) random)
                     (and (= king_new warrior_new)
                          (= kingBlock_new warriorBlock_new))
                     (and (= king_new king)
                          (= kingBlock_new kingBlock))
                )
           )
      )
      )
```

Predicates:

```
(= x1 y1)
(= x2 y2)
(= x3 y3)
(= (modFn x3) x4)
(= (modFn y3) y4)
(< x1 50)
(< y1 50)
```

## E.3 BlockKing Fixed: YML representation

```yaml
name: blockking_fixed

preamble: |
  (declare-fun modFn (Int) Int)

state:
  - name: warrior
    type: (Array Int Int)
  - name: warriorGold
    type: (Array Int Int)
  - name: warriorBlock
    type: (Array Int Int)
  - name: king
    type: Int
  - name: kingBlock
    type: Int

methods:
  - name: enter
    args:
      - name: msg_value
        type: Int
      - name: msg_sender
        type: Int
      - name: block_number
        type: Int
      - name: random
        type: Int
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (and result
      (ite (< msg_value 50)
```

```
        (states_equal warrior warriorGold warriorBlock
                      king kingBlock err
                      warrior_new warriorGold_new warriorBlock_new
                      king_new kingBlock_new err_new)
        (and (= warrior_new (store warrior msg_sender msg_sender))
             (= warriorGold_new (store warriorGold msg_sender msg_value))
             (= warriorBlock_new (store warriorBlock msg_sender block_number))
             (ite (= (modFn (select warriorBlock_new msg_sender)) random)
                  (and (= king_new (select warrior_new msg_sender))
                       (= kingBlock_new (select warriorBlock_new msg_sender)))
                  (and (= king_new king)
                       (= kingBlock_new kingBlock))
             )
        )
   )
   )
```

Predicates:

```
(= x1 y1)
(= x2 y2)
(= x3 y3)
(= (modFn x3) x4)
(= (modFn y3) y4)
(< x1 50)
(< y1 50)
```