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 is 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.

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 [27] 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 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 [30,34], speculative execution (*e.g.* transactional memory [19]),

*This work was partially supported by NSF award #1228768. Author was at New York University when part of the work was completed.

†Support in part by NSF CCF Award #1421126, and CCF Award #1618542. Some of the research was done while author was at IBM Research.

‡Some of the research was done while author was at IBM Research.

peephole partial-order reduction [37], futures, etc. Another important application domain that has emerged recently is Ethereum [1] smart contracts: efficient execution of such contracts hinges on exploiting their commutativity [14] and block-wise concurrency can lead to vulnerabilities [31]. Intuitively, commutativity is an important property because linearizable data-structure operations that commute can be executed concurrently: their effects do not 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. Indeed, it's important for the commutativity condition to be sufficiently granular so that parallelism can be exploited effectively [12]. At the same time, to make safe use of a commutativity condition, it must be sound [24,23]. 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 [22] as well as synthesizing such conditions automatically, *e.g.* based on random interpretation [6], profiling [33] or sampling [18]. None of these approaches, however, meet the goal of computing a commutativity condition that is both *sound* and *granular* in a *fully automated* manner.

In this paper, we present a refinement-based technique for synthesizing 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 [20,16,17,10,28,26] for each method m. Our algorithm iteratively relaxes under-approximations of the commutativity *and* non-commutativity conditions of methods m and n, starting from 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 φ_m^n which 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 [33,19], 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 φ_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 [27], 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 [11]. 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 [22]. Second, we consider BlockKing [31], 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. 5).
- Proof of soundness and relative completeness (Sec. 5).
- An implementation that takes an abstract code specification and automatically generates commutativity conditions using an SMT solver (Sec. 6).
- A novel technique for selecting refinement predicates that improves scalability and the simplicity of the generated formulae (Sec. 6).
- Demonstrated efficacy for several key data structures as well as the BlockKing Ethereum smart contract [31] (Sec. 7).

An extended version of this paper can be found in [8].

Related work. The closest to our contribution in this paper is a recent technique by Gehr *et al.* [18] 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.* [33] identify sequences of actions that commute (via random interpretation and dynamic analysis, respectively). However, neither technique yields an explicit commutativity condition. Kulkarni et al. [25] point out that varying degrees of commutativity specification precision are useful. Kim and Rinard [22] use Jahob to verify manually specified commutativity conditions of several different linked data structures. Commutativity specifications are also found in dynamic analysis techniques [15]. More distantly related is work on synthesis of programs [32] and of synchronization [36,35].

2 Example

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 the relatively simple Set ADT, whose state consists of a single set S that stores an unordered collection of unique elements. Let us consider 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

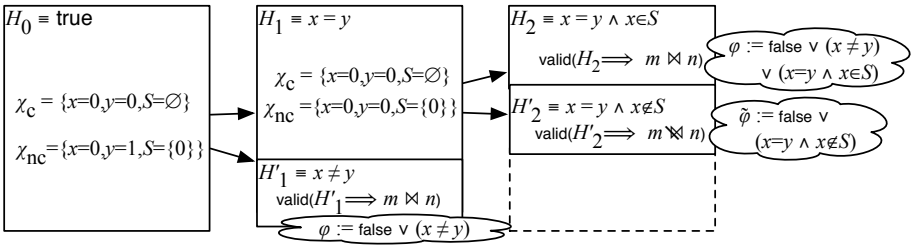
another case that is less obvious: `add` and `contains` commute if they refer to the same element e, as long as in the pre-state $e \in S$. In this case, under both orders of execution, `add` and `contains` leave the set unmodified and return false and true, respectively. The algorithm we describe in this paper completes within a few seconds, producing a precise logical formula φ 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 φ is the negation of $\tilde{\varphi}$.

A more complicated commutativity condition is generated by our tool SERVOIS in 1.4s for Ethereum smart contract BlockKing. Method enter(val_1,sendr_1,bk_1...) (Fig. 3, Sec. 7) does not commute with itself enter(val_2,sendr_2,bk_2...) *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 enumerates the non-commutativity cases and, as discussed in Sec. 7, directly identifies a vulnerability.

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 φ and $\tilde{\varphi}$, respectively. The following diagram illustrates how our algorithm proceeds to generate the commutativity conditions for `add` and `contains` (abbreviated as m and n).



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

We denote by H the logical formula that describes the current state space at a given recursive call. 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 now detail the run of this refinement loop on our earlier Set example. We elaborate on the other challenges that arise in later sections. At each step

of the algorithm, we determine which case we are in via carefully designed validity queries to an SMT solver (Sec. 4). For H_0, it returns the commutativity counterexample: $\chi_{\text{c}} = \{x = 0, y = 0, S = \emptyset\}$ as well as the non-commutativity counterexample $\chi_{\text{nc}} = \{x = 0, y = 1, S = \{0\}\}$. Since, therefore, $H_0 = \mathsf{true}$ 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 χ_{c} but disallows χ_{nc}, and an H_1' that allows χ_{nc} but not χ_{c}. So we must choose a predicate p (from a suitable set of predicates $\mathcal{P}$, discussed later), such that $H_0 \wedge p \Rightarrow \chi_{\text{c}}$ while $H_0 \wedge \neg p \Rightarrow \chi_{\text{nc}}$ (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 $H_1 \equiv \mathsf{true} \wedge x = y$ and one with $H_1' \equiv \mathsf{true} \wedge x \neq y$. 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 φ, 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: $\chi_{\text{c}} = \{x = 0, y = 0, S = \emptyset\}$ and $\chi_{\text{nc}} = \{x = 0, y = 0, S = \{0\}\}$. 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 φ, 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 we have obtained a precise (complete) commutativity/non-commutativity specification: $\varphi \vee \tilde{\varphi}$ is valid (Lem. 2).

Challenges & outline. While the algorithm outlined so far is a relatively standard refinement, the above generated conditions were not immediate. We now discuss challenges involved in generating sound *and* useful conditions.

(Sec. 4) A first question is how to pose the underlying commutativity queries for each subsequent H 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.

(Sec. 5) We have proved that our algorithm is sound even if aborted or the ADT description involves undecidable theories. We further show that termination implies completeness, and specify broad conditions that imply termination.

(Sec. 6) Another challenge is to prioritize predicates during the refinement loop. This choice impacts not only the algorithm's performance, but also the quality/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.

(Sec. 7) We conclude with an evaluation on a range of popular data structures and a case study on boosting the security of an Ethereum smart contract.

3 Preliminaries

States, actions, methods. We will work with a state space Σ, 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* α_1 and α_2 *commute* [15], 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 [27], where a method commutes in one direction and not the other. Details can be found in [8]. Also, in our evaluation (Sec. 7) 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. 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 arguments and return values may be from an infinite domain.

Definition 1. *Methods m and n* commute*, denoted $m \bowtie n$ provided that* $\forall \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\ m(\bar{x})/\bar{r} \bowtie n(\bar{y})/\bar{s}$.

The quantification $\forall \bar{x}\bar{r}$ above 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 symbolically describe the actions of a method m as pre-condition Pre_m and post-condition $Post_m$. Pre-conditions are logical formulae over method arguments and the initial state: $[\![Pre_m]\!] : \bar{x} \to \Sigma \to \mathbb{B}$. Post-conditions are over method arguments, and return values, initial state and final state: $[\![Post_m]\!] : \bar{x} \to \bar{r} \to \Sigma \to \Sigma \to \mathbb{B}$. Given $(Pre_m, Post_m)$ for every method m, we define a transition system $\mathfrak{T} = (\Sigma, A, (\!|\bullet|\!))$ such that $\sigma \xrightarrow{m(\bar{a})/\bar{v}} \sigma'$ *iff* $[\![Pre_m]\!]\ \bar{a}\ \sigma$ and $[\![Post_m]\!]\ \bar{a}\ \bar{v}\ \sigma\ \sigma'$.

Since our approach works on deterministic transition systems, we have implemented an SMT-based check (Sec. 7) that ensures the input transition system is deterministic. Deterministic specifications were sufficient in our examples. 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 [5,13] is left as future work.

Logical commutativity formulae. We will 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 φ 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 [22] but here, for simplicity, we focus on commutativity *pre*-conditions. We may write $[\![\varphi]\!]$ as φ when it is clear from context that φ is meant to be interpreted.

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

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

4 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:

$$\forall \sigma_0\ \sigma_1\ \sigma_{12}.\ (\!|\alpha_1|\!)\sigma_0 = \sigma_1 \wedge (\!|\alpha_2|\!)\sigma_1 = \sigma_{12}\ \ \Rightarrow\ \ \exists \sigma_3.\ (\!|\alpha_2|\!)\sigma_0 = \sigma_3 \wedge (\!|\alpha_1|\!)\sigma_3 = \sigma_{12}$$
(*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 (*i.e.* for which the procedure does not abort).

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

That is, the procedure terminates for every Pre_m, which holds in particular for all of the specifications in the examples we considered (see Sec. 7). 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 $m \hat{\bowtie} n$ that is equivalent to Def. 1. This new definition (inspired by "type lifting") requires only *universal* quantification, and as such, is better suited to SMT-backed algorithms (Sec. 5).

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

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

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 $(|\alpha|)$. 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 is total and remains deterministic. These lifted specifications have types over lifted state spaces: $[\![\widehat{Pre_m}]\!] : \bar{x} \to \hat{\Sigma} \to \mathbb{B}$ and $[\![\widehat{Post_m}]\!] : \bar{x} \to \bar{r} \to \hat{\Sigma} \to \hat{\Sigma} \to \mathbb{B}$. Our implementation performs this lifting via translation denoted Lift from $(Pre_m, Post_m)$ to:

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

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

(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 [8], we show how our tool transforms a counter specification into an equivalent lifted version that is total.

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, $\alpha_1 \hat{\bowtie} \alpha_2$ is equivalent to:

$$\forall \hat{\sigma}_0.\ \hat{\sigma}_0 \neq \mathsf{err} \ \Rightarrow \ (\!|\alpha_2|\!)\ (\!|\alpha_1|\!)\ \hat{\sigma}_0 = (\!|\alpha_1|\!)\ (\!|\alpha_2|\!)\ \hat{\sigma}_0 \tag{1}$$

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. 5), we will use this format to check whether candidate logical formulae describe commutative subregions.

Lemma 1. *$m \bowtie n$ if and only if $m \hat{\bowtie} n$. (*All proofs in [8].*)*

5 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. 6 and 7, challenges in generating precise *and* useful commutativity conditions.

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 φ, 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: $H \Rightarrow m \mathrel{\hat{\bowtie}} n$ (described below). If so, H can be disjunctively added to φ. It may, instead be the case that H represents a region of space for which m and n never commute: $H \Rightarrow m \mathrel{\hat{\not\bowtie}} n$. If so, H can be disjunctively added to $\tilde{\varphi}$. If neither of these cases hold, we have two counterexamples. χ_{c} is the counterexample to commutativity, returned if the validity check on Line 2 fails. χ_{nc} is the counterexample to *non*-commutativity, returned if the validity check on Line 4 fails.

```
1  REFINE^m_n(H, P) {
2      if valid(H ⇒ m ⋈̂ n) then
3        φ := φ ∨ H;
4      else if valid(H ⇒ m ⋈̸̂ n) then
5        φ̃ := φ̃ ∨ H;
6      else
7        let (χc, χnc) = counterexs. to ⋈̂ and ⋈̸̂
8        let p = CHOOSE(H, P, χc, χnc) in
9          REFINE^m_n(H ∧ p, P \ {p});
10         REFINE^m_n(H ∧ ¬p, P \ {p});
11 }
12 main {  φ := false;   φ̃ := false;
13         try { REFINE^m_n(true, P); }
14         catch (InterruptedExn e) { skip; }
15         return(φ, φ̃); }
```

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

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. 7 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 algorithm is exponential in the number of predicates. In Sec. 6 we discuss prioritizing predicates.

The refinement algorithm generates commutativity conditions 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 [25] and boolean combinations of predicates over finite domains [15].

Checking a candidate H^n_m. Our algorithm involves checking whether $(H^n_m \Rightarrow m \mathrel{\hat{\bowtie}} n)$ or $(H^n_m \Rightarrow m \mathrel{\hat{\not\bowtie}} n)$. As shown in Sec. 4, we can check whether H^n_m specifies conditions under which $m \bowtie n$ via an SMT query that does not

introduce quantifier alternation. For brevity, we define:

$$\mathsf{valid}(H^n_m \;\Rightarrow\; m \mathrel{\hat{\bowtie}} n) \;\equiv\; \mathsf{valid}\left(\begin{array}{l}\forall \hat{\sigma}_0\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\quad H^n_m(\hat{\sigma}_0, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \;\Rightarrow \\ \qquad m(\bar{x})/\bar{r}\ \ n(\bar{y})/\bar{s}\ \ \hat{\sigma}_0 = n(\bar{y})/\bar{s}\ \ m(\bar{x})/\bar{r}\ \ \hat{\sigma}_0\end{array}\right)$$

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

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

$$\begin{array}{rl}\alpha_1 \mathrel{\hat{\not\bowtie}} \alpha_2 & \equiv \forall \hat{\sigma}_0.\ \hat{\sigma}_0 \neq \mathsf{err} \;\Rightarrow\; (\!|\alpha_2|\!)\ (\!|\alpha_1|\!)\ \hat{\sigma}_0 \neq (\!|\alpha_1|\!)\ (\!|\alpha_2|\!)\ \hat{\sigma}_0 \\ m \mathrel{\hat{\not\bowtie}} n & \equiv \forall \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\ m(\bar{x})/\bar{r} \mathrel{\hat{\not\bowtie}} n(\bar{y})/\bar{s}\end{array}$$

We thus define a check for when φ^n_m is a *non*-commutativity condition with:

$$\mathsf{valid}(H^n_m \;\Rightarrow\; m \mathrel{\hat{\not\bowtie}} n) \;\equiv\; \mathsf{valid}\left(\begin{array}{l}\forall \hat{\sigma}_0\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\ H^n_m(\hat{\sigma}_0, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \;\Rightarrow\; \hat{\sigma}_0 \neq \mathsf{err} \;\Rightarrow \\ \qquad m(\bar{x})/\bar{r}\ \ n(\bar{y})/\bar{s}\ \ \hat{\sigma}_0 \neq n(\bar{y})/\bar{s}\ \ m(\bar{x})/\bar{r}\ \ \hat{\sigma}_0\end{array}\right)$$

Theorem 1 (Soundness). *For each* $\textsc{Refine}^m_n$ *iteration:* $\varphi \Rightarrow m \mathrel{\hat{\bowtie}} n$*, and* $\tilde{\varphi} \Rightarrow m \mathrel{\hat{\not\bowtie}} n$*.*

All proofs available in [8]. Soundness holds regardless of what Choose returns and even when the theories used to model the underlying data-structure are incomplete. Next we show termination implies completeness:

Lemma 2. *If* $\textsc{Refine}^m_n$ *terminates, then* $\varphi \vee \tilde{\varphi}$*.*

Theorem 2 (Conditions for Termination). $\textsc{Refine}^m_n$ *terminates if 1.* **(expressiveness)** *the state space* Σ *is partitionable into a finite set of regions* $\Sigma_1, ..., \Sigma_N$*, each described by a finite conjunction of predicates* ψ_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.

6 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 can be automatically generated relatively easily, thus enabling the integration with other tools [20,16,17,10,28,26]. In [8], we show the Counter ADT specification, which was derived from the *Pre* and *Post* conditions used in earlier work [22]. The states of a transition system describing an ADT are encoded as list of variables (each as a name/type pair), and each method specification requires a list of argument types, return type, and *Pre*/*Post* conditions. Again, the Counter example can be seen in [8]. .

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 [11] 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. 4). An example of LIFT applied to Counter is in [8].

Next, SERVOIS automatically generates the predicate language (PGEN) in addition to user-provided hints. If the predicate vocabulary is not sufficiently expressive, then the algorithm would not be able to converge on precise commutativity and non-commutativity conditions (Sec. 5). 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. 7, this strategy works well in practice. Intuitively, *Pre* and *Post* formulas suffice to express the footprint of an operation. 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). CHOOSE must return a predicate such that $\chi_{\mathrm{c}} \Rightarrow H \wedge p$ and $\chi_{\mathrm{nc}} \Rightarrow H \wedge \neg p$. This guarantees progress on both recursive calls. When combined with a heuristic to favor less complex atoms, this ensured timely termination on our examples. We refer to this as the *simple* heuristic.

Though this produced precise conditions, they were not always very concise, which is desirable for human understanding, and inspection purposes. We thus introduced a new heuristic which significantly improves the *qualitative* aspect of our algorithm. We found that doing a lookahead (recurse 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.

7 Case studies

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

	$m(\bar{x})$	$n(\bar{y})$	**Simple** Qs (time)	**Poke** Qs (time)	φ_n^m **(Poke)**
Counter	`decrement`	$\bowtie$ `decrement`	3 (0.1)	3 (0.1)	true
	`increment`	$\triangleright$ `decrement`	10 (0.3)	34 (0.9)	$\neg(0 = c)$
	`decrement`	$\triangleright$ `increment`	3 (0.1)	3 (0.1)	true
	`decrement`	$\bowtie$ `reset`	2 (0.1)	2 (0.1)	false
	`decrement`	$\bowtie$ `zero`	6 (0.1)	26 (0.6)	$\neg(1 = c)$
	`increment`	$\bowtie$ `increment`	3 (0.1)	3 (0.1)	true
	`increment`	$\bowtie$ `reset`	2 (0.0)	2 (0.1)	false
	`increment`	$\bowtie$ `zero`	10 (0.3)	34 (0.8)	$\neg(0 = c)$
	`reset`	$\bowtie$ `reset`	3 (0.1)	3 (0.1)	true
	`reset`	$\bowtie$ `zero`	9 (0.2)	30 (0.6)	$0 = c$
	`zero`	$\bowtie$ `zero`	3 (0.1)	3 (0.1)	true
Acum.	`increase`	$\bowtie$ `increase`	3 (0.1)	3 (0.1)	true
	`increase`	$\bowtie$ `read`	13 (0.3)	28 (0.6)	$c + x_1 = c$
	`read`	$\bowtie$ `read`	3 (0.0)	3 (0.0)	true
Set	`add`	$\bowtie$ `add`	10 (0.4)	140 (4.4)	$(y_1 = x_1 \wedge y_1 \in S) \vee \neg(y_1 = x_1)$
	`add`	$\bowtie$ `contains`	10 (0.4)	122 (3.6)	$x_1 \in S \vee (\neg(x_1 \in S) \wedge \neg(y_1 = x_1))$
	`add`	$\bowtie$ `getsize`	6 (0.2)	31 (0.9)	$x_1 \in S$
	`add`	$\bowtie$ `remove`	6 (0.2)	66 (2.2)	$\neg(y_1 = x_1)$
	`contains`	$\bowtie$ `contains`	3 (0.1)	3 (0.1)	true
	`contains`	$\bowtie$ `getsize`	3 (0.1)	3 (0.1)	true
	`contains`	$\bowtie$ `remove`	17 (0.5)	160 (4.8)	$S \setminus \{x_1\} = \{y_1\} \vee (... \wedge y_1 \in \{x_1\}) \vee ...$
	`getsize`	$\bowtie$ `getsize`	3 (0.1)	3 (0.1)	true
	`getsize`	$\bowtie$ `remove`	13 (0.3)	37 (1.0)	$\neg(y_1 \in S)$
	`remove`	$\bowtie$ `remove`	21 (0.7)	192 (6.4)	$S \setminus \{y_1\} = \{x_1\} \vee (... \wedge y_1 \in \{x_1\}) \vee ...$
HashTable	`get`	$\bowtie$ `get`	3 (0.1)	3 (0.1)	true
	`get`	$\bowtie$ `haskey`	3 (0.1)	3 (0.1)	true
	`put`	$\triangleright$ `get`	13 (0.4)	74 (2.3)	$(H[x_1 \leftarrow x_2] = H \wedge y_1 \in keys)$ $\vee(\neg(H[x_1 \leftarrow x_2] = H) \wedge \neg(y_1 = x_1))$
	`get`	$\triangleright$ `put`	10 (0.3)	48 (1.5)	$[H[y_1] = y_2] \vee [\neg(H[y_1] = y_2) \wedge \neg(y_1 = x_1)]$
	`remove`	$\triangleright$ `get`	3 (0.1)	3 (0.1)	true
	`get`	$\triangleright$ `remove`	13 (0.4)	40 (1.2)	$\neg(y_1 = x_1)$
	`get`	$\bowtie$ `size`	3 (0.1)	3 (0.1)	true
	`haskey`	$\bowtie$ `haskey`	3 (0.1)	3 (0.1)	true
	`haskey`	$\bowtie$ `put`	10 (0.3)	52 (1.6)	$[y_1 \in keys] \vee [\neg(y_1 \in keys) \wedge \neg(y_1 = x_1)]$
	`haskey`	$\bowtie$ `remove`	17 (0.5)	44 (1.3)	$[x_1 \in keys \wedge \neg(y_1 = x_1)] \vee [\neg(x_1 \in keys)]$
	`haskey`	$\bowtie$ `size`	3 (0.1)	3 (0.1)	true
	`put`	$\bowtie$ `put`	24 (0.9)	357 (13.5)	$... \vee (\neg(H[y_1] = y_2) \wedge \neg(y_1 = x_1))$
	`put`	$\bowtie$ `remove`	6 (0.3)	33 (1.2)	$\neg(y_1 = x_1)$
	`put`	$\bowtie$ `size`	6 (0.2)	23 (0.8)	$x_1 \in keys$
	`remove`	$\bowtie$ `remove`	21 (0.8)	192 (6.9)	$[keys \setminus \{x_1\} = \{y_1\}] \vee [...]$
	`remove`	$\bowtie$ `size`	13 (0.4)	37 (1.1)	$\neg(x_1 \in keys)$
	`size`	$\bowtie$ `size`	3 (0.1)	3 (0.1)	true
Stack	`clear`	$\bowtie$ `clear`	3 (0.1)	3 (0.1)	true
	`clear`	$\bowtie$ `pop`	2 (0.1)	2 (0.1)	false
	`clear`	$\bowtie$ `push`	2 (0.1)	2 (0.1)	false
	`pop`	$\bowtie$ `pop`	6 (0.2)	20 (0.6)	$nextToTop = top$
	`push`	$\triangleright$ `pop`	72 (2.1)	115 (3.5)	$\neg(0 = size) \wedge top = x_1$
	`pop`	$\triangleright$ `push`	34 (0.9)	76 (2.2)	$y_1 = top$
	`push`	$\bowtie$ `push`	13 (0.5)	20 (0.7)	$y_1 = x_1$

Fig. 2. Automatically generated commutativity conditions (φ_n^m). Right-moverness ($\triangleright$) conditions identical for a pair of methods denoted by $\bowtie$. **Qs** denotes number of SMT queries. Running time in seconds. Longer conditions have been truncated, see [7].

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

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

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 [9] 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. All evaluation data is available on our website [2].

Depending on the pair of methods, the number of predicates generated by PGen were (count after filtering in parentheses): Counter: 25-25 (12-12), Accumulator: 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 [7]. 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.

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 [22]. For the Set data structure, the work of [22] used a less precise Set abstraction, so we instead validated against the conditions of [25]. As for HashTable, we validated that our conditions match those by Dimitrov *et al.* [15].

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 exe-

cuted 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 [31] 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 [31]. Unfortunately, many smart contracts are not written with this in mind, and attackers can exploit interleavings to their benefit.

As an example, we study the BlockKing smart contract. Fig. 3 provides a simplification of its description, as discussed in [31]. 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 3 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 5, with a callback method provided on Line 7. 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 10.

Since random number generation is done via an RPC, 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 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 [8]). 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`:

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

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 sendr_1 will always be different from 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 [8]). SERVOIS generated the following *non*-commutativity condition after 3.5s.

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

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/block to the shared king/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.

8 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 [19], open nested transactions [29], and other non-transactional concurrency paradigms such as race detection [15], parallelizing compilers [30,34], and, as we show, robustness of Ethereum smart contracts [31]. It has been shown that understanding the commutativity of data-structure operations provides a key avenue to improved performance [12] or ease of verification [24,23].

This work opens several avenues of future research. For instance, leveraging the internal state of the SMT solver (beyond counterexamples) in order to generate new predicates [21]; automatically building abstract representation or making inferences such as one we made for the stack example; and exploring strategies to compute commutativity conditions directly from the program's code, without the need for an intermediate abstract representation [34].

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. K. Bansal. *Decision Procedures for Finite Sets with Cardinality and Local Theory Extensions*. PhD thesis, New York University, Jan. 2016.
8. K. Bansal, E. Koskinen, and O. Tripp. Automatic generation of precise and useful commutativity conditions (extended version). *CoRR*, abs/1802.08748, 2018. https://arxiv.org/abs/1802.08748.
9. K. Bansal, A. Reynolds, C. Barrett, and C. Tinelli. A new decision procedure for finite sets and cardinality constraints in SMT. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*, pages 82–98. Springer, 2016.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.

20. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
28. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.
29. 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.
30. 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.
31. I. Sergey and A. Hobor. A concurrent perspective on smart contracts. In *1st Workshop on Trusted Smart Contracts*, 2017.
32. 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.
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.