Journal of Automated Reasoning (2020) 64:1333–1359
https://doi.org/10.1007/s10817-020-09573-w

Synthesizing Precise and Useful Commutativity Conditions

Kshitij Bansal[1] · Eric Koskinen[2] · Omer Tripp[3]

Received: 20 June 2020 / Accepted: 27 June 2020 / Published online: 29 August 2020

Abstract

Reasoning about commutativity between data-structure operations is an important problem with many applications. In the sequential setting, commutativity can be used to reason about the correctness of refactoring, compiler transformations, and identify instances of nondeterminism. In parallel contexts, commutativity dates back to the database (Weihl in IEEE Trans Comput 37(12):1488–1505, 1988) and compilers (Rinard and Diniz in ACM Trans Program Lang Syst 19(6):942–991, 1997) communities and, more recently, appears in optimistic parallelization (Herlihy and Koskinen in Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming, 2008), dynamic concurrency (Tripp et al. in Proceedings of the 33rd ACM SIGPLAN conference on programming language design and implementation, PLDI '12, New York, NY, USA, ACM, pp 145–156, 2012; Dimitrov et al. in Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation, 2014), scalable systems (Clements et al. in ACM Trans Comput Syst 32(4):10, 2015) and even smart contracts (Dickerson et al. in Proceedings of the ACM symposium on principles of distributed computing, PODC '17, New York, NY, USA, ACM, pp 303–312, 2017). 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 (Lipton in Commun ACM 8(12):717–721, 1975) 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-

This article extends our prior work [3], with the addition of proofs (Theorems 1 and 2), support for left- and right-movers (Sect. 6), and a new set of applications (Sect. 9), including memories and lock-based synchronization, transactional memory, distributed systems, refactoring, verification, and code synthesis.

K. Bansal was partially supported by NSF award #1228768. E. Koskinen was partially supported by NSF CCF Award #1421126, CCF Award #1618542, and CCF Award #1813745, and some of the work was done while he was at New York University and at IBM Research. K. Bansal was at New York University when part of the work was completed. The work by O. Tripp was done prior to him joining Amazon (while he was at IBM Research and Google).

Extended author information available on the last page of the article

source tool SERVOIS. Our algorithm produces quantifier-free queries that are dispatched to a back-end SMT solver. We evaluate SERVOIS first by synthesizing commutativity conditions for a range of data structures including Set, HashTable, Accumulator, Counter, and Stack. We then show several applications of our work including reasoning about memories and locks, finding vulnerabilities in Ethereum smart contracts, improving transactional memory performance, distributed applications, code refactoring, verification, and synthesis.

Keywords Commutativity · Abstraction refinement · Synthesis

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 can improve correctness of sequential programs, eg in validating refactoring transformations like code motion [16], and further unlocks the potential of multicore architectures, including parallelizing compilers [31,38], speculative execution (e.g. transactional memory [19]), peephole partial-order reduction [41], futures, etc. In recent years, another important application domain has emerged: Ethereum [15] smart contracts. Efficient execution of such contracts hinges on exploiting their commutativity [11] and block-wise concurrency can lead to vulnerabilities [32]. 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 [9]. At the same time, to make safe use of a commutativity condition, it must be sound [24,25]. 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 [23] as well as synthesizing such conditions automatically, e.g. based on random interpretation [2], profiling [37] 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 specifications [6,14,17,20,27,29] denoted $(Pre_m, Post_m)$ 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 [19,37], 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. We next show that our algorithm can be generalized to left-/right-movers [28], a more precise version of commutativity.

We have implemented our approach as the SERVOIS tool, whose code and documentation are available online [33]. SERVOIS is built on top of the CVC4 SMT solver [7]. We first evaluate SERVOIS through by generating 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 [23].

We then explore a range of applications of how the commutativity conditions we use can be employed in numerous contexts. We consider reasoning about memories and locks. We also consider BlockKing [32], an Ethereum smart contract, with its known vulnerability. We demonstrate how a developer can be guided by SERVOIS to create a more robust implementation. Furthermore, we describe how SERVOIS can aid transactional memory, distributed systems, code refactoring, verification and synthesis.

Contributions In summary, this paper makes the following contributions:

- The first sound and precise technique to automatically generate commutativity conditions (Sect. 5).
- Proofs of soundness and relative completeness (Sect. 5).
- A generalization to left- and right-movers (Sect. 6).
- An implementation that takes an abstract code specification and automatically generates commutativity conditions using an SMT solver (Sect. 7).
- A novel technique for selecting refinement predicates that improves scalability and the simplicity of the generated formulae (Sect. 7).
- Demonstrated efficacy for several key data structures (Sect. 8).
- Application of our work to a variety of contexts (Sect. 9).

An earlier version of this work was previously published. [3].

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 [2] and Tripp et al. [37] identify sequences of actions that commute (via random interpretation and dynamic analysis, respectively). However, neither technique yields an explicit commutativity condition. Kulkarni et al. [26] point out that varying degrees of commutativity specification precision are useful. Kim and Rinard [23] use Jahob to verify manually specified commutativity conditions of several different linked data structures. Commutativity is also of interest in the feature-oriented programming commutativity. Chechik et al. [8] discuss how to identify non-commutativity by examining so-called feature trees. If two operations in this tree may write to the same shared variables, the authors conservatively conclude that these operations are non-commutative. Commutativity specifications are also found in dynamic analysis techniques [12]. More distantly related is work on synthesis of programs [35] and of synchronization [39,40].

2 Example

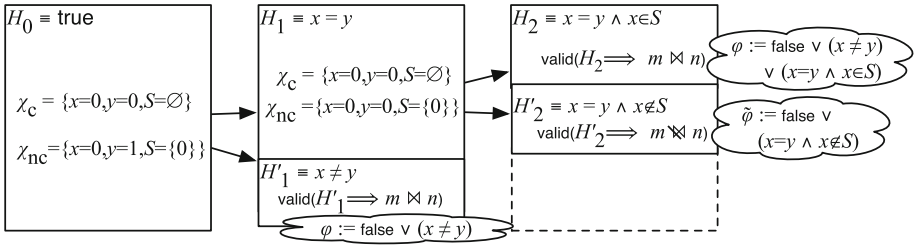
Specifying commutativity conditions is generally nontrivial and it is easy to miss subtle corner cases. Additionally, it has to be done pairwise for all methods. For ease of illustration, we will focus on the relatively simple Set ADT, whose state consists of a single set S that stores an unordered collection 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 SERVOISin 1.4s for Ethereum smart contract BlockKing. This contract has a method called enter(val_1, sendr_1, bk_1…) (Fig. 5, Sect. 8) which *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 Sect. 8, 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 (Sect. 4).

For H_0, it returns the commutativity counterexample: $\chi_c = \{x = 0, y = 0, S = \emptyset\}$ as well as the non-commutativity counterexample $\chi_{nc} = \{x = 0, y = 1, S = \{0\}\}$. Since, therefore, $H_0 = \mathsf{true}$ 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 $H_0 \wedge p \Rightarrow \chi_c$ 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 $H_1 \equiv \mathsf{true} \wedge x = y$ 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: $\chi_c = \{x = 0, y = 0, S = \emptyset\}$ and $\chi_{nc} = \{x = 0, y = 0, S = \{0\}\}$. 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 $H_2' \equiv x = y \wedge x \notin S$. It now 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 (Lemma 1).

Challenges and 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.

(Section 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.

(Section 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.

(Section 6) We have generalized our theory and algorithm to be able to synthesis conditions for left- and right-movers [28], an asymmetric version of commutativity.

(Section 7) 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.

(Section 8) We next provide an evaluation on a range of popular data structures including a Counter, Stack, HashTable, Accumulator, and Set.

(Section 9) We show several applications of our work, including memories and lock-based synchronization, a case study on the security of an Ethereum smart contract, and examples of how SERVOIS commutativity conditions can be employed in transactional memory, distributed systems, code refactoring, verification, and code synthesis.

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* [12], 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 [28], where a method commutes in one direction and not the other. We return to this in Sect. 6. Also, in our evaluation (Sect. 8) 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: $[\![Pre_m]\!] : \bar{x} \rightarrow \Sigma \rightarrow \mathbb{B}$. Post-conditions are over method arguments, and return values, initial state and final state: $[\![Post_m]\!] : \bar{x} \rightarrow \bar{r} \rightarrow \Sigma \rightarrow \Sigma \rightarrow \mathbb{B}$. 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 (Sect. 8) 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 [1,10] 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, *i.e.* that $[\![\varphi]\!] : (\sigma, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \to \mathbb{B}$. (We tuple the arguments for brevity.) The first argument is the initial state. Commutativity *post*- and *mid*-conditions can also be written [23] 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

Definition 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).

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

That is, the procedure terminates for every Pre_m, which holds in particular for all of the specifications in the examples we considered (see Sect. 8). 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 $m \mathbin{\hat{\bowtie}} n$ that is equivalent to Definition 1. This new definition (inspired by "type lifting") requires only *universal* quantification, and as such, is better suited to SMT-backed algorithms (Sect. 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:

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

(For simplicity of presentation, 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 Definition 2. Later, we show that our tool transforms a counter specification (Fig. 2) into an an equivalent lifted version that is total (Fig. 3).

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:

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

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

5 Iterative Refinement

We now present an iterative refinement strategy that, when given a lifted abstract transition system, generates the conditions for commutativity and non-commutativity. We then discuss soundness and relative completeness and, in Sects. 7 and 8, 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: $H \Rightarrow m \hat{\bowtie} n$ (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: $H \Rightarrow m \hat{\not\bowtie} n$. 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_n^m(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_n^m(H ∧ p, P \ {p});
10       REFINE_n^m(H ∧ ¬p, P \ {p});
11 }
12 main { φ := false;  φ̃ := false;
13          try { REFINE_n^m(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 Sect. 8 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 $H \wedge \neg p$. Since we branch on each predicate, the algorithm is exponential in the number of predicates in the worst case. In Sect. 7 we discuss prioritizing predicates to make this practical in practice.

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 [26] and boolean combinations of predicates over finite domains [12].

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 Sect. 4, 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:

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

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_m^n is a condition under which m and n *do not* commute. First, we define negative analogs of commutativity:

$$\alpha_1 \mathbin{\hat{\not\bowtie}} \alpha_2 \equiv \forall \hat{o}_0.\ \hat{o}_0 \neq \mathsf{err} \Rightarrow (\!|\alpha_2|\!)\ (\!|\alpha_1|\!)\ \hat{o}_0 \neq (\!|\alpha_1|\!)\ (\!|\alpha_2|\!)\ \hat{o}_0$$
$$m \mathbin{\hat{\not\bowtie}} n \equiv \forall \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\ m(\bar{x})/\bar{r} \mathbin{\hat{\not\bowtie}} n(\bar{y})/\bar{s}$$

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

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

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

Proof We focus on the commutativity condition φ case; $\tilde{\varphi}$ is analogous. Initially, $\varphi = \mathsf{false}$ is a sound condition for when commutativity holds. The REFINE_n^m algorithm proceeds by iteratively updating φ, constructing a DNF formula of the following shape:

$$\varphi = \mathsf{false} \vee (p_0^0 \wedge \cdots \wedge p_{N_0}^0) \vee \cdots \vee (p_0^M \wedge \cdots \wedge p_{N_M}^M)$$

where there are some M disjuncts, each consisting of some N_i (for $i \in [1, M]$) predicate conjuncts. For soundness, it suffices to show that each disjunct's conjunction is a valid commutativity condition. Fix one such conjunction $H = p_0 \wedge \cdots \wedge p_{N_i}$. This conjunction is accumulated as the first parameter on the REFINE_n^m call stack, in the recursive call made on Lines 9 and 10. φ is only updated on Line 3 and it does so by adding this new conjunction H. However, the addition of H is guarded by $\mathsf{valid}(H \Rightarrow m \hat{\bowtie} n)$ on Line 2 and thus each H must be a sound commutativity condition. □

Because all additions to the (non)commutativity conditions are immediately proceeded by a valid check, soundness depends only on simple local condition rather than a complicated invariant. Note also that 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 1 *If* REFINE_n^m *terminates, then* $\varphi \vee \tilde{\varphi}$.

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 $p_0 \wedge \cdots \wedge p_n$.

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

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

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

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 $\pi = p_0, p_0 \wedge p_1, p_0 \wedge p_1 \wedge p_2, \ldots$.

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

By the above claim, at least one of the predicates ψ_i is not contained in any finite prefix of π. 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). □

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

6 Right-/Left-Movers

We now describe how the formalism and algorithm presented thus far can be extend to a more fine-grained notion of commutativity: an asymmetric version called left-movers and right-movers [28], where a method commutes in one direction and not the other.

Definition 3 (*Right-mover* [28]) We say that an **action** α_1 *moves to the right of* action α_2 commute, denoted $\alpha_1 \triangleright \alpha_2$, provided that $(\!|\alpha_2|\!) \circ (\!|\alpha_1|\!) \subseteq (\!|\alpha_1|\!) \circ (\!|\alpha_2|\!)$. For **methods** m and n,

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

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

We also extend right-movers to lifted transition systems, as in Sect. 4. We use $\hat{\triangleright}$ to mean $\triangleright$ but over lifted transition systems $\hat{\mathfrak{T}}$.

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

$$\begin{aligned}
&\mathsf{valid}(\Psi_m^n \Rightarrow m\ \hat{\triangleright}\ n) \\
&\equiv \mathsf{valid}\left(\begin{array}{l}
\forall \hat{\sigma}_0\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}. \\
\quad \Psi_m^n(\hat{\sigma}_0, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \ \Rightarrow\ \ \hat{\sigma}_0 \neq \mathsf{err}\ \Rightarrow \\
\quad [\![n(\bar{y})/\bar{s}]\!]\ [\![m(\bar{x})/\bar{r}]\!]\ \hat{\sigma}_0 \neq \mathsf{err}\ \Rightarrow \\
\quad [\![n(\bar{y})/\bar{s}]\!]\ [\![m(\bar{x})/\bar{r}]\!]\ \hat{\sigma}_0 = [\![m(\bar{x})/\bar{r}]\!]\ [\![n(\bar{y})/\bar{s}]\!]\ \hat{\sigma}_0.
\end{array}\right)
\end{aligned}$$

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

Lemma 2 *If* $\mathsf{valid}(H_m^n \Rightarrow m\ \hat{\triangleright}\ n)$ *and* $\mathsf{valid}(H_m^n \Rightarrow n\ \hat{\triangleright}\ m)$ *then* $\mathsf{valid}(H_m^n \Rightarrow m \bowtie n)$.

We define $\hat{}\textsc{Refine}_n^m$ to be the same algorithm as in Fig. 1, except that $\mathsf{valid}(H_m^n \Rightarrow m\ \hat{\bowtie}\ n)$ is replaced with $\mathsf{valid}(H_m^n \Rightarrow m\ \hat{\triangleright}\ n)$. Then,

Theorem 3 (Left-mover soundness) *For each* $\hat{}\textsc{Refine}_n^m$ *iteration:* $\Psi_m^n \Rightarrow m\ \hat{\triangleright}\ n$, *and* $\tilde{\Psi}_m^n \Rightarrow m\ \hat{\not\triangleright}\ n$.

Proof Similar to Theorem 1. □

6.1 The Full Lay of the Land

Recall that commutativity implies both-moverness while non-commutativity implies either non-left-, non-right- or non-both-moverness. We use the following notation for the three subcases of a non-commutativity condition $\tilde{\varphi}$:

$\varphi_{\not\triangleright\triangleleft}$: left-, but *non*-right- mover condition.
$\varphi_{\triangleright\not\triangleright}$: right-, but *non*-left- mover condition.
$\varphi_{\not\triangleright\not\triangleright}$: *non*-left- and *non*-right- mover condition.

In some cases it may be helpful to distinguish these cases. Therefore, we now describe how to combine the $\hat{\triangleright}\text{REFINE}_n^m$ and $\hat{\triangleleft}\text{REFINE}_n^m$ algorithms so that we can fully identify all cases. Let us denote $\hat{\triangleright}\text{REFINE}_n^m(H, \mathcal{P})$ to be the algorithms above, starting from state space H and constructing/returning right-moverness/non-right-moverness pair (ψ, ψ').

```
1 let (ψ_r, ψ̃_r) = ▷̂REFINE_n^m(true, P) in
2 let (ψ_l, ψ̃_l) = ◁̂REFINE_n^m(true, P) in
3   φ̃ := ψ̃_r ∨ ψ̃_l;
4   φ_◁▷ := ψ̃_r ∧ ψ̃_l;
5   φ_⋈ := ψ_r ∧ ψ_l;
6   φ_◁◁ := ψ̃_r ∧ ψ_l;
7   φ_▷▷ := ψ_r ∧ ψ̃_l;
```

The above calls to REFINE divide the state space into four quadrants. Properties $\varphi_{\bowtie}$, $\varphi_{\triangleleft\triangleleft}$, $\varphi_{\triangleright\triangleright}$ and $\varphi_{\triangleleft\triangleright}$ are all defined via conjunction. Meanwhile, the weaker non-commutativity $\tilde{\varphi}$ is defined with disjunction. While $\tilde{\varphi}$ maintains DNF form, the other formulas have lost it. This can be rectified through formula manipulation. Alternatively, one could explore modifications to the original REFINE algorithm that query the state space at this more precise granularity. We leave this to future work.

7 The SERVOIS Tool and Practical Considerations

We have developed the open-source tool SERVOIS [34]. SERVOIS uses CVC4 [7] as a backend SMT solver. It begins by parsing an input ADT specification and performing some pre-processing, discussed below. It subsequently implements REFINE, LIFT, predicate generation, and a method for selecting predicates (CHOOSE) discussed below.

Input We use an input specification language building on YAML (which has parser and printer support for many programming languages) with SMTLIB as the logical language. This can be automatically generated relatively easily, thus enabling the integration with other tools [6,14,17,20,27,29]. Specifically, the input is specified by the following:

- `name`: name of the object being modeled.
- `state`: a list of `name`, `type` where name is name of the fields, and type the SMTLIB type being used to model the corresponding field.
- `states_equal`: a SMT formula denoting when two states should be considered equal.
- `methods`: specification of the methods as pre and post conditions. Specifically, following things need to be specified for a method:
  - `args`: a list of `name`, `type` where name is name of an argument, and type the SMTLIB type being used to model the corresponding argument.
  - `return`: a list of `name`, `type` where name is name of a return value, and type the SMTLIB type being used to model the corresponding return value.
  - `requires`: a SMT formula denoting the precondition of the method.
  - `ensures`: a SMT formula denoting the postcondition of the method.
  - `terms`: optionally, terms that should be used to generate predicates.

An example input for the Counter ADT specification can be seen in Fig. 2. It was derived from the *Pre* and *Post* conditions used in earlier work [23]. The states of a transition system

```
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]
```

Fig. 2 An example of the input ADT specification for Counter

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. Notice that the Counter specification (Fig. 2) involves methods that have preconditions (e.g. `increment`) and therefore, the specification is not total. SERVOIS performs the LIFT transformation[1] as described in Sect. 4. An example of LIFT applied to Counter is in Fig. 3. Notice that the state space has been augmented with `err` and post-conditions of all methods now account for `err`. Moreover, `states_equal` has also been amended.

Predicate Generation (PGEN) Next, SERVOIS automatically generates the predicate language 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 (Sect. 5). We generate predicates by using terms and operators that appear in the specification, and generating well-typed atoms not trivially true or false. For example, if `size, 1, (size+1)` are terms of sort $\mathbb{Z}$ that appear in the formula along with the predicates = and $\geq$, then we generate ($\texttt{size} = 1$), ($\texttt{size} \geq 1$), etc (a total of 18 predicates in this case). We filter out those that are trivial. As we demonstrate in Sect. 8, our predicate generation 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 $\chi_c \Rightarrow H \wedge p$ 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 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 the 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) 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. We call this strategy "*poke*." 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 step, 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.

8 Evaluation

We applied SERVOIS to Set, HashTable, Accumulator, Counter, and Stack. The generated commutativity conditions for these data structures typically combine multiple theories, such

[1] https://github.com/kbansal/servois/blob/master/src/lift.py.

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

    ))'
```

Fig. 3 An example of the ADT specification after lifting has been performed. (`terms` have been elided for lack of space.)

as sets, integers and arrays. We now discuss how we represented each of the data structures. All evaluation data is available on the SERVOIS homepage [33].

1. **Counter** The Counter involves typical operations: `increment`, `decrement`, `zero` test and `reset`. The counter can be thought of as an integer that can take only non-negative values (the state is modeled in our language as one variable of sort Int, and non-negativity is enforced using preconditions).
   In particular, this breaks the assumption that a successor state always exists doesn't hold (Axiom 1). We get around this assumption in our encoding by introducing an Err state as described in Sect. 3. Our abstract definition encodes this as pair of variables, one of sort Int (for contents), and other sort Bool (for Err).
2. **Accumulator** This is a simple data structure which holds a natural number, initially zero. There are only two methods: `add`(int x), which increments internal state by x; and `read()`, which returns the internal value. We represent the internal state using a single variable of sort Int.
3. **Set** This is a Set data structure with four operations: `add`, `remove`, `contains` and `getsize`. We encode the state in SMT using a two variables with Set sort [4] and size as Int sort. The elements of the Set are of an uninterpreted sort. The operations are straightforward. `add` and `remove` each take one argument and return true or false. The return value is true if and only if the data structure is modified (e.g., `add`(x) returns true if x is not in the data structure before the call). `contains` and `getsize` do not modify the state of the data structure.
4. **Hashtable** We use the SMTLIB theories of arrays and finite sets to encode a Hashtable. The state is represented as a tuple (`keys`, `H`). `keys` keeps track of the set of keys on which the Hashtable is currently defined, this is encoded as a set of an uninterpreted sort: (Set E). The values corresponding to keys in the Hashtable is encoded using the array theory: `H` is an array from E to another uninterpreted sort for the values. `get` takes a single argument of sort E, if it is not in keys it goes to an error state. Otherwise, it returns the value in the Hashtable. `put` takes two arguments: a (key,value) pair and updates the Hashtable. Similar to Set, it returns true (resp. false) if the data structure is modified (resp. not modified). `remove` takes a single argument, the key to be removed if it exists. Other methods are self-explanatory.
5. **Stack** We use an abstraction which tracks only the top two values and the size. We observe that stack operations `push` and `pop` only modify the top element. Thus for the purposes of commutativity conditions we are looking at the value of only top two elements are of importance, other that we check that size is the same. A subtle point here though is that even though top two values can be modified, we need to track up to four values since two pop calls can cause the third from top value to become part of the top two values.

Depending on the pair of methods, the number of predicates generated by PGEN were as follows: We did not provide any hints to the algorithm for this case study. On all our

ADT	Predicates	After filtering
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

examples, the *simple* heuristic terminated with precise commutativity conditions. In Fig. 4, 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. 4 are those generated by the *poke* heuristic, and interested reader may compare them with the simple heuristic in [5]. On the theoretical side, our CHOOSE implementation is fair (satisfies condition 2 of Theorem 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 Theorem 2.

Evaluation of Various Abstract Data Structures 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 [23]. For the Set data structure, the work of [23] used a less precise Set abstraction, so we instead validated against the conditions of [26]. As for HashTable, we validated that our conditions match those by Dimitrov et al. [12].

9 Applications

Commutativity has played a central role in a number of application domains. In this section we discuss use cases that emerge from our automated reasoning about commutativity, as performed by SERVOIS.

9.1 Memory, Locks and Commutativity

We now apply SERVOIS to the setting of memories and locks. We work with a set of variables *Vars* (of uninterpreted sort) and a memory $\texttt{mem} : \mathit{Vars} \to \mathbb{Z}$. First, we consider the simple setting where there are two operations: $\texttt{read}(x)/v$ and $\texttt{write}(y)/w$, similar to a Hashtable but with fewer methods. Not surprisingly, SERVOIS generated the commutativity conditions including the fact that `write` operations on the same variable commute provided that the value being written is the same.

Next, we introduce locks and mix them with memory. Let $\mathcal{L}_\perp$ be the type of locks. We use τ to represent a lock value (e.g. a thread IDs) with a special distinct lock $\perp$ to indicate unlocked. Finally, we maintain a mapping $\texttt{locked} : \mathit{Vars} \to \mathcal{L}_\perp$ to indicate whether a given memory location is locked and, if so, by whom.

To represent these types and methods in SERVOIS, we exploited the underlying SMT solver's ability to declare sorts and to declare datatypes such as a Pair. Technically, `locked` maps variables to a Pair, consisting of a boolean flag (for $\perp$) and a sort for thread IDs. We define methods $\texttt{lock}(\tau_i, x)$ and $\texttt{unlock}(\tau_i, x)$ which, respectively, represent thread τ_i locking and unlocking the variable x, following the usual semantics of locks. We then further define $\texttt{tryread}(\tau_i, x)$ which attempts to read the value of x but can only do so if τ_i holds the lock and returns -1 otherwise. We similarly define $\texttt{trywrite}(\tau_i, x, v)$.

For commutativity in this context, there are many different cases to consider. For example, when considering only `tryread` and `trywrite`, SERVOIS generated a commutativity condition that had 11 different cases. This cases include:

	$m(\bar{x})$		$n(\bar{y})$	**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. 4 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 [5]

1. `tryread` $\bowtie$ `trywrite` when operating on different variables.
2. `tryread` $\bowtie$ `trywrite` when the value being written is the same as what's already there.
3. `tryread` $\bowtie$ `trywrite` when it is the same thread, same variable, but the lock is not held. (In this case both the read and write will fail.)
4. `tryread` $\bowtie$ `trywrite` when they are different threads.

Running SERVOIS, we also obtained commutativity conditions such as:

Method pair	Example commutativity case
`tryread` $\bowtie$ `tryread`	*always*
`trywrite`(x, τ_i) $\bowtie$ `lock`(y, τ_j)	when $x \neq y$ and the lock for y is not held.
`lock`(x, τ_i) $\bowtie$ `lock`(y, τ_j)	whenever $x \neq y$.
`lock`(x, τ_i) $\bowtie$ `lock`(y, τ_j)	whenever $x = y$ and the lock is held by anyone, including a third party.
`lock`(x, τ_i) $\bowtie$ `lock`(y, τ_j)	whenever $x = y$ and the lock is unheld, but $\tau_i = \tau_j$.
`lock`(x, τ_i) $\bowtie$ `unlock`(y, τ_j)	whenever $x \neq y$.
`lock`(x, τ_i) $\bowtie$ `unlock`(y, τ_j)	whenever $\tau_i = \tau_j$ and $x = y$, but the lock is held by someone else.
`lock`(x, τ_i) $\bowtie$ `unlock`(y, τ_j)	for different threads with $x = y$, but it is already locked by τ_i.
`lock`(x, τ_i) $\bowtie$ `unlock`(y, τ_j)	for different threads with $x = y$, but locked by a third party.

9.2 Ethereum Smart Contracts and BlockKing

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 [36] and executed on the Ethereum blockchain [15]. 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 [32] 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 [32]. 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. Figure 5 provides a simplification of its description, as discussed in [32]. 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.

```
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. 5 Simplified code for BlockKing in a C-like language

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 SERVOIS source code [34]). 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.

Fixed version of BlockKing. 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, shown in Fig. 6. (YML versions of these two programs, `blockking.yml` and `blockking_fixed.yml`, can be found in our source code repository [34].) 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

```
1  struct storage {
2     int warrior;
3     int warriorGold;
4     int warriorBlock;
5     int res;
6  };
7
8  hashtable[int,struct storage] scratch = ...;
9  int king, kingBlock;
10
11 void enter_fixed(int val, int sendr, int bk, int rnd) {
12    if (val < 50) { send(sendr,val); return; }
13    scratch[sendr].warrior = sendr;
14    scratch[sendr].warriorGold = val;
15    scratch[sendr].warriorBlock = bk;
16    // callback generates the random number in scratch[sendr]
17    rpc_call("random number generator",__callback,scratch[sendr].res);
18    function __callback() {
19       if (modFun(scratch[sendr].warriorBlock) == scratch[sendr].res) {
20          king = scratch[sendr].warrior; // winner
21          kingBlock = scratch[sendr].warriorBlock; } } }
```

Fig. 6 Our fixed version of BlockKing in a C-like language

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.

9.3 Transactional Memory

The output of SERVOIS can be used for speculative concurrent execution, e.g. transactional memory. Specifically, transactional boosting [19] is a form of transactional memory in which transactions consist of operations on shared highly-concurrent objects rather than directly on a shared memory. This strategy permits conflict to be defined—based on commutativity—at the level of abstract data types rather than at the overly conservative level of memory read/write operations. It has been shown to improve performance [19]. Here is an example of an execution of two boosted transactions:

Thread 1 :	`begin` $v = sk.\mathsf{pop}();$	$ht.\mathsf{put}(5, v);$	`commit`
Thread 2 :	`begin`	$w = ht.\mathsf{get}(6);$	`commit`

The above transactions perform operations on a shared Stack sk and Hashtable ht, which each must be implemented as linearizable objects with no shared state beyond the state of the objects. Even though these are concurrent transactions, the boosting methodology permits concurrent operations on the same objects, provided that the operations *commute*. As we

have seen in Sect. 8, these hashtable methods indeed commute because they are operating on distinct keys.

The impact of boosting on the performance of transactional memory critically depends on the quality and completeness of the specification, which controls the extent to which spurious rollbacks are avoided. SERVOIS is unique among availably synthesis tools in its ability to (i) always generate commutativity conditions, even if not fully precise, for arbitrary data structures, and (ii) ensure the soundness of the generated conditions, so that none of the guarantees of the underlying transactional memory system is lost.

9.4 Testing for Interactions Between Code Blocks

The importance of representing concurrent data structures according to their guarantees has already been established by past studies, eg in the context of exposing impediments to loop parallelization [38]. Again here, as with transactional memory, there is the challenge of automatically coming up with a commutativity specification. The HawkEye approach is to emulate concrete-level reads and writes w.r.t. the abstract representation of a concurrent ADT implementation. This yields approximate commutativity conditions, whereas SERVOIS enables an alternative that is more precise while still practical (see discussion above). As an illustrative example, consider concurrent calls to Counter's decrement method. SERVOIS is able to establish that these always commute, as show in Sect. 8, yet writing to the value field of the counter to decrement it would trigger a spurious conflict by HawkEye.

To show this, we extended our implementation of the Counter, to further support a write(v) operation, which directly wrote to the counter. Re-running SERVOIS, we found that:

- incr $\bowtie$ decr as long as the counter was not 0 (as in the original Counter).
- incr $\bowtie$ incr in all cases (as in the original Counter).
- incr $\bowtie$ write *never*.
- decr $\bowtie$ write *never*.
- write $\bowtie$ write when both are writing the same value (as in a Memory).

9.5 Parallel and Distributed Systems

As the BlockKing example (Sect. 9.2) illustrates, a major aspect of reasoning about the correctness of parallel and distributed programs concerns the conditions under which concurrent execution of the code is safe. As long as these conditions meet the intentions of the developer, and are enforced at the implementation level via appropriate synchronization, concurrent execution is safe. From a formal standpoint, this amounts to commutativity checking.

SERVOIS exposes a convenient interface for this type of reasoning. The ability to focus on operations of interest, using custom logical vocabularies, enables granular yet accurate reasoning about their interactions. Here is an example, adapted from Xiao et al. [43]:

```
1 int max = 0;
2 int y = 0;
3 foreach (Row row in input) {
4    int x = row["x"].Integer;
5    if(max < x) {
6      max = x;
7      y = row["y"].Integer; }
```

Given two instances of the body of the `foreach` loop, SERVOIS is able to report that commutativity is not generally guaranteed. Indeed, while simply searching for the maximal value is an operation that can be evaluated in a distributed fashion, using that value (`x`) to extract another column value (`y`) yields deterministic output only if that other value is the same across rows the share the maximal value.

9.6 Refactoring

A common use case in code maintenance and refactoring is to move statements [16]. That occurs, for example, when trying to increase reuse via method extraction; hoisting statements outside a loop structure for improved performance; or changing the order of statements to organize the code better. In all of these cases, there is the danger of introducing unintended side effects because of latent interactions between statements that have changed their relative position.

IDE-integrated refactoring tools often take care of the change itself, but without being able to (fully) reason about the correctness of the transformation. Ultimately it is up to the developer to approve the change (eg via a dialog that presents the before and after versions of the code).

An example follows:

```
1 int max = 0;
2 int y = 0;
3 foreach (String e in input) {
4     { "X" ≠ e }
5     Data x = map.Get("X");
6     Data y = map.Get(e);
7     Compute(x, y); }
```

It is safe to hoist `Data x = map.get("x");` outside the loop given the knowledge that $\forall$`e` $\in$ `input.`$\forall$`"X"` $\neq$ `e`. However, this is beyond what a standard compiler or refactoring tool is able to establish. Within this use case, SERVOIS could suggest that `"X"` $\neq e$ is a sufficient condition for the hoisting operation, which the developer can then review and approve.

SERVOIS is well positioned to integrate with IDE refactoring tools, in that it's feasible to create reusable vocabularies of predicates for popular languages, libraries and data structures, and with these, have SERVOIS perform resource-bounded reasoning when a refactoring request is made. The unique advantage of SERVOIS, beyond features already mentioned above, is in its ability to always generate sound commutativity conditions, even if stopped before reaching a stable solution.

9.7 Verification and Test Case Generation

Testing and verification are yet another opportunity to apply SERVOIS. Specifically, given an encoding of a data structure and its associated operations, the conditions that SERVOIS computes for commutativity become a correctness checking objective. As a concrete example, SERVOIS can integrate with dynamic analysis tools to guide checking of the implementation both under conditions where method calls are expected to commute and where they are expected to interfere.

Here is an illustrative example:

```
1 Object PutIfAbsent(Map m, Entry e) {
2           m.Put(e.Key, e.Value); }
```

Per specification, two calls to `PutIfAbsent` commute either if the `Maps` are different or if the `Entrys` differ in their `Key` field or if the `Values` are the same. With this information given by SERVOIS, a testing tool can check for the behavior when `PutIfAbsent` calls interfere. That would uncover a bug, whereby

```
PutIfAbsent(k,v1); PutIfAbsent(k,v2);
```

yields a state where `k` is mapped to `v2` rather than `v1`.

Integrations of this sort bridge the step down from specification to implementation. SERVOIS synthesizes commutativity conditions at the specification level, which are discharged to a concrete-level checking tool for testing or verification against the implementation. Again in this use case, the ability to generate conditions within a limited time budget, and while reusing predicate vocabularies across verification tasks, makes SERVOIS a practical alternative that is easy to integrate with algorithms for verification or testing.

9.8 Code Synthesis

An analogous use case to correctness checking with SERVOIS operating at the specification level is code synthesis. In this scenario, the conditions computed by SERVOIS are become the specification for synchronization synthesis.
Consider the following program:

```
1 k1 := placement(v1);
2 k2 := placement(v2);
3 { k1 ≠ k2 } // φ^n_m
4 if (*)
5    put(k1, v1); put(k2, v2);
6 else
7    put(k2, v2); put(k1, v1);
```

The second part of this program (from Line 4 onward) is attempting to exploit the commutativity of `put` and `put` operating on different keys, represented by commutativity condition φ_m^n used as a precondition. The synthesis question becomes: *what implementation of* `placement` *ensures that* $k_1 \neq k_2$?

Synthesis algorithms like that by Itzhaky et al. [22] can then be used to generate an admissible implementation of `placement`.

Synchronization synthesis need to be fully optimal, but (i) the synthesis algorithm should be reasonably efficient (not taking too long to complete), (ii) the synthesized synchronization should be sound, and finally (iii) the algorithm should be (close to) complete, not failing too often to synthesize synchronization. The SERVOIS design, and guarantees, meet all three of these requirements.

10 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 [30], and other non-transactional concurrency paradigms such as race detection [12], parallelizing compilers [31,38], and, as we show, robustness of Ethereum smart contracts [32]. It has been shown that understanding the commutativity of data-structure operations provides a key avenue to improved performance [9] or ease of verification [24,25].

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 [38]. Finally, it may be worth exploring how our approach of ensuring totality applies to other properties such as safety and liveness.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**, 253–284 (1991)
2. Aleen,F., Clark, N.: 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), pp. 241–252. ACM (2009)
3. Bansal, K., Koskinen, E., Tripp, O.: Automatic generation of precise and useful commutativity conditions. In: Tools and Algorithms for the Construction and Analysis of Systems—24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Greece, April 2018, Proceedings, Part II (2017)
4. Bansal, K., Reynolds, A., Barrett, C., Tinelli, C.: A new decision procedure for finite sets and cardinality constraints in SMT. In: Proceedings of the 8th International Joint Conference on Automated Reasoning, vol. 9706, pp. 82–98. Springer (2016)
5. Bansal, K.: Decision Procedures for Finite Sets with Cardinality and Local Theory Extensions. Ph.D. Thesis, New York University (Jan. 2016)
6. Barnett, M., Leino, K.R.M., Schulte, W.: 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, pp. 49–69 (2005)
7. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.), Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11), vol. 6806, pp. 171–177. Springer (July 2011)
8. Chechik, M., Stavropoulou, I., Disenfeld, C., Rubin, J.: FPH: efficient non-commutativity analysis of feature-based systems. In: Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings, pp. 319–336 (2018)
9. Clements, A.T., Kaashoek, M.F., Zeldovich, N., Morris, R.T., Kohler, E.: The scalable commutativity rule: designing scalable software for multicore processors. ACM Trans. Comput. Syst. **32**(4), 10 (2015)
10. Cook, B., Koskinen, E.: 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, pp. 399–410 (2011)
11. Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding concurrency to smart contracts. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC '17, New York, NY, USA, pp. 303–312 (2017). ACM
12. Dimitrov, D., Raychev, V., Vechev, M.T., Koskinen, E.: Commutativity race detection. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom, June 09–11, 2014, p. 33. ACM (2014)
13. Dimitrov, D., Raychev, V., Vechev, M., Koskinen, E.: Commutativity race detection. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14) (2014)
14. Ernst, G.W., Ogden, W.F.: Specification of abstract data types in modula. ACM Trans. Program. Lang. Syst. **2**(4), 522–543 (1980)

15. Ethereum. https://ethereum.org/. Accessed 26 Aug 2020
16. Ettinger, R.: Program sliding. In: ECOOP 2012—Object-Oriented Programming—26th European Conference, Beijing, China, June 11–16, 2012. Proceedings, pp. 713–737 (2012)
17. Flon, L., Misra, J.: A unified approach to the specification and verification of abstract data types. In: Proceedings of Specifications of Reliable Software Conference. IEEE Computer Society (1979)
18. Gehr, T., Dimitrov, D., Vechev, M.T.: Learning commutativity specifications. In: Computer Aided Verification—27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I, pp. 307–323 (2015)
19. Herlihy, M., Koskinen, E.: 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. Hoare, C.A.R.: Software pioneers. In: Broy, M., Denert, E. (eds.) Proof of Correctness of Data Representations, pp. 385–396. Springer, New York (2002)
21. Hu, Y., Barrett, C., Goldberg, B.: 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), pp. 281–289. IEEE Computer Society (Sept. 2004)
22. Itzhaky, S., Gulwani, S., Immerman, N., Sagiv, M.: A simple inductive synthesis methodology and its applications. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, New York, NY, USA, pp. 36–46 (2010). ACM
23. Kim, D., Rinard, M.C.: 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, pp. 528–541. ACM (2011)
24. Koskinen, E., Parkinson, M.J., Herlihy, M.: Coarse-grained transactions. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 19–30. ACM (2010)
25. Koskinen, E., Parkinson, M.J.: The push/pull model of transactions. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, Portland, OR, USA, June (2015)
26. Kulkarni, M., Nguyen, D., Prountzos, D., Sui, X., Pingali, K.: Exploiting the commutativity lattice. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 542–555. ACM (2011)
27. Leino, K.R.M.: Specifying and verifying programs in spec#. In: Proceedings of the 6th International Perspectives of Systems Informatics, Andrei Ershov Memorial Conference, PSI 2006, p. 20 (2006)
28. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
29. Meyer, B.: Applying "design by contract". IEEE Comput. **25**(10), 40–51 (1992)
30. Ni, Y., Menon, V., Adl-Tabatabai, A., Hosking, A.L., Hudson, R.L., Moss, J.E.B., Saha, B., Shpeisman, T.: Open nesting in software transactional memory. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, pp. 68–78. ACM (2007)
31. Rinard, M.C., Diniz, P.C.: Commutativity analysis: a new analysis technique for parallelizing compilers. ACM Trans. Program. Lang. Syst. **19**(6), 942–991 (1997)
32. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: 1st Workshop on Trusted Smart Contracts (2017)
33. Servois homepage. http://cs.nyu.edu/~kshitij/projects/servois. Accessed 26 Aug 2020
34. Servois source code. https://github.com/kbansal/servois. Accessed 26 Aug 2020
35. Solar-Lezama, A., Jones, C. G., Bodík, R.: Sketching concurrent data structures. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation PLDI 2008, pp. 136–148 (2008)
36. Solidity programming language. https://solidity.readthedocs.io/en/develop/. Accessed 26 Aug 2020
37. Tripp, O., Manevich, R., Field, J., Sagiv, M.: Janus: exploiting parallelism via hindsight. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, New York, NY, USA, pp. 145–156 (2012). ACM
38. Tripp, O., Yorsh, G., Field, J., Sagiv, M.: 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, part of SPLASH 2011, Portland, OR, USA, October 22–27, 2011, pp. 207–224 (2011)
39. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 327–338 (2010)

40. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, pp. 125–135 (2008)
41. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 382–396. Springer, Berlin (2008)
42. Weihl, W.: Commutativity-based concurrency control for abstract data types. IEEE Trans. Comput. **37**(12), 1488–1505 (1988)
43. Xiao, T., Zhang, J., Zhou, H., Guo, Z., McDirmid, S., Lin, W., Chen, W., Zhou, L.: Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs. In: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, New York, NY, USA, pp. 44–53 (2014). ACM

Affiliations

Kshitij Bansal[1] · Eric Koskinen[2] · Omer Tripp[3]

✉ Omer Tripp
omertrip@amazon.com

Kshitij Bansal
kbk@google.com

Eric Koskinen
eric.koskinen@stevens.edu

1 Google Inc., Mountain View, CA, USA

2 Stevens Institute of Technology, Hoboken, NJ, USA

3 Amazon.com, East Palo Alto, CA, USA