Decomposing Data Structure Commutativity Proofs with *mn*-Differencing

Eric Koskinen[1(✉)] and Kshitij Bansal[2]

[1] Stevens Institute of Technology, Hoboken, USA
eric.koskinen@stevens.edu
[2] Google, Inc., Menlo Park, USA

Abstract. Commutativity of data structure methods is of ongoing interest in contexts such as parallelizing compilers, transactional memory, speculative execution and software scalability. Despite this interest, we lack effective theories and techniques to aid commutativity verification.

In this paper, we introduce a novel decomposition to improve the task of verifying method-pair commutativity conditions from data structure implementations. The key enabling insight—called *mn*-differencing—defines the precision necessary for an abstraction to be fine-grained enough so that commutativity of method implementations in the abstract domain entails commutativity in the concrete domain, yet can be less precise than what is needed for full-functional correctness. We incorporate this decomposition into a proof rule, as well as an automata-theoretic reduction for commutativity verification. Finally, we discuss our simple proof-of-concept implementation and experimental results showing that *mn*-differencing leads to more scalable commutativity verification of some simple examples.

1 Introduction

For an object o, with state σ and methods m, n, etc., let $\bar{x}$ and $\bar{y}$ denote argument vectors and $m(\bar{x})/\bar{r}$ denote a method signature, including a vector of corresponding return values $\bar{r}$. *Commutativity* of two methods, denoted $m(\bar{x})/\bar{r} \bowtie n(\bar{y})/\bar{s}$, are circumstances where operations m and n, when applied in either order, lead to the same final state and agree on the intermediate return values $\bar{r}$ and $\bar{s}$. A *commutativity condition* is a logical formula $\varphi_m^n(\sigma, \bar{x}, \bar{r})$ indicating, for a given state σ, whether the methods will always commute, as a function of parameters.

Commutativity conditions are typically much smaller than full specifications, yet they are powerful: it has been shown that they are an enabling ingredient in correct, efficient concurrent execution in the context of parallelizing compilers [39], optimistic parallelism [33], transactional memory [16,24,29,30,36], race detection [17], speculative execution, features [13], layered concurrent programs [31], etc. More broadly, a paper from the systems community [15] found

E. Koskinen—Supported in part by NSF award #1813745 and #2008633.

F. Henglein et al. (Eds.): VMCAI 2021, LNCS 12597, pp. 81–103, 2021.
https://doi.org/10.1007/978-3-030-67067-2_5

that, when software fragments are implemented so that they commute, better scalability is achieved. Commutativity captures independence and when combined with linearizability proofs (*e.g.*, [12,43]) enables concurrent execution. Naturally, it is important that commutativity be correct and, in recent years, growing effort has been made toward reasoning about commutativity conditions automatically. At present, these works are either unsound [4,21] or else they rely on data structure specifications as intermediaries [7,27] which, interestingly, can lead to unsound commutativity conclusions (see Sect. 2).

Our goal in this paper is to improve the task of verifying a commutativity property φ_m^n directly from the data-structure source code of methods m and n. Toward this goal, we first provide a straight-forward way to formulate the problem as a multi-trace (2-safety) question, *i.e.*, relating the behaviors in one circumstance with those in another. This first automata-theoretic reduction (called $\textsc{Reduce}_m^n$) is a product program, but with the pre-condition strengthened by only considering reachable data-structure states and the post-condition weakened to observational equivalence. Although $\textsc{Reduce}_m^n$ is sound, it does not employ any commutativity-specific abstractions and, thus, reachability solvers struggle to verify the resulting encoding, for lack of the ability to decompose the problem in a manner suitable to commutativity.

The key idea of this paper is a decomposition geared toward improving commutativity verification. We introduce the concept of an mn-differencing abstraction (α, R_α) which gives a requirement for how precise an abstraction α must be so that one can reason in that abstract domain and relate abstract post-states with R_α, and yet entail return value agreement in the concrete domain. Intuitively, R_α captures the differences between the behavior of pairs of operations when applied in either order (*e.g.* how `push` and `pop` effect the top element of stack), while abstracting away state reads or mutations that would be the same, regardless of the order in which they are applied (*e.g.* those elements deeper in the stack that are untouched). R_α relations capture return value agreement, but they do not quite capture commutativity. We show the pieces fit together by combining R_α with a relation C that tracks the unmodified, cloned portion of the state and an ADT-specific observational equivalence relation I_β. Proving that I_β is an observational equivalence relation is then done using a separate ADT-specific abstraction β.

We then return to algorithms, introducing a second reduction $\text{DA}\textsc{Reduce}_m^n$ that exploits mn-differencing. $\text{DA}\textsc{Reduce}_m^n$ emits two reachability tasks: automata $\mathcal{A}_\mathbf{A}(m, n, \varphi_m^n, I)$ and $\mathcal{A}_\mathbf{B}(I)$, thus allowing reachability analyses to synthesize separate abstractions (α, R_α) and C for $\mathcal{A}_\mathbf{A}(m, n, \varphi_m^n, I)$ and β for $\mathcal{A}_\mathbf{B}(I)$. Moreover, $\mathcal{A}_\mathbf{B}(I)$ is independent of m, n and φ_m^n, so it can be proved safe once and then reused for every subsequent φ_m^n query.

We implement our reductions in a simple prototype tool called CityProver, on top of Ultimate [22] and CPAchecker [10]. CityProver takes as input simple data structures in C (with integers, structs, arrays) and a candidate formula φ_m^n. It then uses the reductions to discover a proof that φ_m^n is a valid commutativity condition or else produce a counterexample. We report encouraging preliminary results verifying commutativity properties of some simple data structures such as a memory cell, counter, two-place Set, array stack, array queue and rudimentary hash table. In all examples, CityProver was able to discover α, R_α, C and β automatically. In some cases we manually provided simple I relations. Since we reduce to automaton reachability, there was no need for any other user input (such as invariants, preconditions, predicates, lemmas, etc.). We further consider the merits of users providing I relations as opposed to the pre/post-specifications in prior work [7,27], and discuss benefits pertaining to soundness, automation, simplicity and usability. Finally, our experiments show that *mn*-differencing improves commutativity verification. DAReduce$_m^n$ performs better than Reduce$_m^n$: it is typically faster and suffers from timeouts less frequently.

Contributions. In summary, our contributions are:

- A reduction Reduce$_m^n$ that strengthens the pre-condition to reachable ADT states and weakens the post-condition to observational equivalence. (Sect. 4)
- A decomposition of commutativity reasoning that gives a requirement for how precise an abstraction must be to entail concrete commutativity. (Sect. 5)
- An improved reduction DAReduce$_m^n$, which exploits *mn*-differencing and observational equivalence relations. (Sect. 6)
- A proof-of-concept implementation, that uses these reductions to verify candidate commutativity conditions. (Sect. 7)
- Preliminary experiments showing that DAReduce$_m^n$ out-performs Reduce$_m^n$ on some simple numeric data structures such as a memory cell, counter, two-place Set, array stack, array queue and rudimentary hash table. (Sect. 7)

Some results have been abridged. An extended version is available [28].

Our verified commutativity conditions can be used with existing concurrent implementations (compilers [39], graph algorithms [33,36], STM [16,24], etc.). Moreover, with some further research, they could be combined with linearizability proofs and used inside parallelizing compilers. We believe this to be a promising direction for future work.

Limitations. *mn*-differencing is defined semantically and could be applied to a wide range of programs, parametric data-structures, etc. Our implementation relies on underlying reachability solvers which are typically limited to programs with simple arrays and simple pointers, with limited support for quantified invariants. Thus, although *mn*-differencing and DAReduce$_m^n$ support ADTs with parameterized sizes (such as ArrayStack), our experiments instead compared Reduce$_m^n$-*vs*-DAReduce$_m^n$ for (infinite state) ADTs of a fixed size. We were also limited by these tools' capability of performing permutation reasoning (*e.g.* limited disjunctive power).

2 Overview

Motivating Examples. Consider the SimpleSet data structure shown at the left of Fig. 1. This data structure is a simplification of a Set, capable of storing up to two natural numbers using private integers a and b. Value -1 is reserved to indicate that nothing is stored in the variable. Method add(x) checks to see if

```
class SimpleSet {
 private int a, b, sz;
 SimpleSet() { a=b=−1; sz=0; }
 void add(uint x) {
   if(sz == 0) { a=x; sz++; ret; }
   if(a==x || b==x) { ret; }
   if(a==−1) { a=x; sz++; ret; }
   if(b==−1) { b=x; sz++; ret; }
   ret;
 }
 bool isin(uint y) { ret (a==y||b==y);}
 int getsize() { ret sz; }
 void clear() { a=−1; b=−1; sz = 0; }
}
```

```
class ArrayStack {
 private int A[MAX], top;
 ArrayStack() { top = −1; }
 bool push(int x) {
  if (top==MAX−1) ret false;
  A[top++] = x; ret true;
 }
 int pop() {
  if (top == −1) ret −1;
  else ret A[top−−]; }
 bool isempty() { ret (top==−1); }
}
```

Fig. 1. On the left, a SimpleSet data structure, capable of storing up to two natural numbers (using integer fields a and b) and tracking the size sz of the Set. On the right, a simple ArrayStack, that implements a stack using an array A and a top index.

there is space available and that x is not already in the Set, and then stores x in an open slot (either a or b). **ret** means return. Methods isin(y), getsize() and clear() are straightforward.

A commutativity condition, written as a logical formula φ_m^n, describes the conditions under which two methods $m(\bar{x})$ and $n(\bar{y})$ commute, in terms of the argument values and the state of the data structure σ. Two methods isin(x) and isin(y) always commute because neither modifies the ADT, so we say $\varphi_{\mathsf{isin}(x)}^{\mathsf{isin}(y)} \equiv \mathsf{true}$. The commutativity condition of methods add(x) and isin(y) is more involved: $\varphi_{\mathsf{add}(x)}^{\mathsf{isin}(y)} \equiv x \neq y \vee (x = y \wedge \mathsf{a} = x) \vee (x = y \wedge \mathsf{b} = x)$. This condition specifies three situations (disjuncts) in which the two operations commute. In the first case, the methods are operating on different values. Method isin(y) is a read-only operation and since $y \neq x$, it is not affected by an attempt to insert x. Moreover, regardless of the order of these methods, add(x) will either succeed or not (depending on whether space is available) and this operation will not be affected by isin(y). In the other disjuncts, the element being added is already in the Set, so method invocations will observe the same return values regardless of the order and no changes (that could be observed by later methods) will be made by either of these methods. Note that there can be multiple concrete ways of representing the same semantic data structure state: $\mathsf{a} = 5 \wedge \mathsf{b} = 3$ is the same as $\mathsf{a} = 3 \wedge \mathsf{b} = 5$. Other commutativity conditions include: $\varphi_{\mathsf{isin}(y)}^{\mathsf{clear}} \equiv (\mathsf{a} \neq y \wedge \mathsf{b} \neq y)$, $\varphi_{\mathsf{isin}(y)}^{\mathsf{getsize}} \equiv \mathsf{true}$, $\varphi_{\mathsf{add}(x)}^{\mathsf{clear}} \equiv \mathsf{false}$, $\varphi_{\mathsf{clear}}^{\mathsf{getsize}} \equiv \mathsf{sz} = 0$ and $\varphi_{\mathsf{add}(x)}^{\mathsf{getsize}} \equiv \mathsf{a} = x \vee \mathsf{b} = x \vee (\mathsf{a} \neq x \wedge \mathsf{a} \neq -1 \wedge \mathsf{b} \neq x \wedge \mathsf{b} \neq -1)$.

As a second running example, let us consider an array based implementation of Stack, given at the right of Fig. 1. ArrayStack maintains array A for data, a top index to indicate end of the stack, and has operations push and pop.

The capacity of $\mathsf{ArrayStack}$, MAX is parametric. The commutativity condition $\varphi^{\mathsf{pop}}_{\mathsf{push}(x)} \equiv \mathsf{top} > -1 \wedge \mathsf{A}[\mathsf{top}] = x \wedge \mathsf{top} < \mathsf{MAX}$ captures that they commute provided that there is at least one element in the stack, the top value is the same as the value being pushed and that there is enough space to push.

The above examples illustrate that commutativity conditions, even for small data-structures, can quickly become tricky to reason about. Nonetheless, correctness of these conditions is important to avoid unsafe concurrency when they are used in parallelization strategies [13,15,16,24,31,33,36]. Some prior works have described unsound methods for verifying commutativity [4,21] and others [7,27] have built upon ADT specifications which, as we discuss below, can lead to unsound commutativity conditions.

What's Hard About This Problem? Toward proving that a candidate φ^n_m is a commutativity condition for $m(\bar{x}) \bowtie n(\bar{y})$, one can begin by posing the problem as 2-safety [14], perhaps using Hoare quadruple notation [44] below on the left:

Hoare Quad.

$$\begin{array}{l|l}
\multicolumn{2}{l}{\{\varphi^n_m(\sigma_1) \wedge \sigma_1 = \sigma_2\}} \\
r^1_m := m(\bar{a}); & r^2_n := n(\bar{b}); \\
r^1_n := n(\bar{b}); & r^2_m := m(\bar{a}); \\
\multicolumn{2}{l}{\{r^1_m = r^2_m \wedge r^1_n = r^2_n \wedge \sigma'_1 = \sigma'_2\}}
\end{array}$$

Example

$$\begin{array}{l|l}
\multicolumn{2}{l}{\{\varphi^{\mathsf{pop}()}_{\mathsf{push}(x)} \wedge \sigma_1 = \sigma_2\}} \\
r^1_m := \mathsf{push}(x); & r^2_n := \mathsf{pop}(); \\
r^1_n := \mathsf{pop}(); & r^2_m := \mathsf{push}(x); \\
\multicolumn{2}{l}{\{r^1_m = r^2_m \wedge r^1_n = r^2_n \wedge \sigma'_1 = \sigma'_2\}}
\end{array}$$

Intuitively, this Hoare quadruple (similar to a product program [8] or self-composition [9,41]) involves two *copies* of the program, shown on either side of the vertical bar. The pre-condition is a relation on the states of these two programs, as is the post-condition. For commutativity, we start by letting the precondition require that the commutativity condition φ^n_m holds and that the two programs begin in the same ADT states. Meanwhile, the post condition asserts that return values will agree and that the post-states are equivalent. Above on the right is an example: $\mathsf{ArrayStack}$ with $\varphi^{\mathsf{pop}()}_{\mathsf{push}(x)} \equiv \mathsf{A}[\mathsf{top}] = x \wedge \mathsf{top} > 1 \wedge \mathsf{top} < \mathsf{MAX}$. Running an existing tool (*e.g.* a product program [8] and Ultimate [22]) yields a counterexample, with starting state: $\mathsf{A} = [z, y, x, \alpha] \wedge \mathsf{top} = 2$. The counterexample shows that in this case the post states are different. Depending on the order methods are applied, one reaches either $\mathsf{A} = [z, y, x, \alpha] \wedge \mathsf{top} = 2$ or else $\mathsf{A} = [z, y, x, x] \wedge \mathsf{top} = 2$. Our knowledge of stack semantics tells us that these are the same state (because the value in the 3rd array slot does not matter), but automated tools do not know these states are equivalent: concrete equality is too strict. Similarly, for $\mathsf{SimpleSet}$ $\varphi^{\mathsf{add}(y)}_{\mathsf{add}(x)} \equiv x \neq y$ we would obtain a counterexample complaining that $(\mathsf{a} = x \wedge \mathsf{b} = y)$ is different from $(\mathsf{a} = y \wedge \mathsf{b} = x)$.

It appears we need a better notion of equality for the post-states. We might then be tempted to exploit specifications, as is done in prior work [7,27]. Then we can ask whether $Post_m(Post_n(\sigma)) = Post_n(Post_m(\sigma))$[1]. Unfortunately, it is unclear what precision is appropriate for commutativity. Let's take, for example, a coarse specification such as $\{\mathsf{true}\}\mathsf{push}(x)\{\mathsf{true}\}$. Using this in our Hoare

[1] Note: as discussed in Sect. 3, we employ the technique discussed in Sect. 4 of Bansal *et al.* [7] to avoid the need for under-approximation or quantifier alternation.

quadruple, we might conclude a post-relation true, seemingly indicating that all post-states are related. We would thus be inclined to incorrectly conclude that any φ_m^n is a valid commutativity condition. When specifications are too coarse like this one, Bansal *et al.* [7] would incorrectly synthesize commutativity condition $\varphi_{\mathsf{push}(x)}^{\mathsf{push}(y)} \equiv \mathsf{true}$. The problem is that abstraction does not capture effects of $\mathsf{push}(x)$ that are relevant to commutativity.

Meanwhile, fine-grained specifications can be close to what is needed for full-functional correctness and it is not clear that we need this level of granularity: much of the post-condition is irrelevant to commutativity. When considering $\mathsf{push}(x)$ and pop, the interaction is limited to the top element of the stack (as well as whether the stack is empty or full), whereas the deeper part of the stack is the same regardless of the order of these methods.

Decomposition and Reductions for Commutativity. We now summarize the challenges and contributions of our work in the context of these examples.

(Section 4). We first observe that we do not strictly need pre/post specifications for commutativity verification and, instead, can work with observational equivalence relations. As a simple start, we describe a straight-forward reduction $\textsc{Reduce}_m^n$ from verifying commutativity conditions of an ADT to an automaton reachability problem. $\textsc{Reduce}_m^n$ emits an automaton $\mathcal{A}(\varphi_m^n)$ whose safety entails that φ_m^n is a valid commutativity condition for methods m and n. To this end, the reduction (i) ensures that we only concern ourselves with commutativity from an over-approximation of the *reachable* states of the object and (ii) weakens the post-condition to a notion of *observational* equivalence. While $\textsc{Reduce}_m^n$ is sound, it does not lead to scalable tools: reachability solvers struggle to decompose the problem.

(Section 5). The main question we ask in this paper is: *What is the right abstraction granularity for commutativity?* Not knowing this has hindered prior works as well as the performance of $\textsc{Reduce}_m^n$. First, the necessary precision depends on methods under consideration. For example, when concerned with return values arising in commutativity of $\mathsf{SimpleSet}$'s $\mathsf{isin}\ (y)/\ \mathsf{clear}$, it is sufficient to use an abstraction that ignores sz. We only need to reason about whether y is stored in a or b. We can use, *e.g.*, an abstraction with predicates $\mathsf{a} = y$ and $\mathsf{b} = y$ (along with their negations). This also ignores all other possible values for a and b: for showing return value agreement, the only relevant aspect of the state is whether or not y is in the set. Similarly, for $\mathsf{ArrayStack}$ $\mathsf{push}(x)/\mathsf{pop}()$, we only need to consider the top value and we can abstract away deeper parts of the stack, that are untouched in either method order. While, on the other hand, for $\mathsf{pop}() \bowtie \mathsf{pop}()$, the second-from-top also matters.

Formally, we give a requirement for an abstraction α and a relation R_α in that domain, that it be precise enough so that reasoning about return value agreement in the abstract domain faithfully covers reasoning about agreement in the concrete domain. We call this pair (α, R_α) an mn-differencing abstraction and relation. For the SimpleSet example with isin (x)/ clear $()$, we can define α to be based on the above mentioned predicates, and then use the relation:

$$R_\alpha(\sigma_1, \sigma_2) \equiv (\mathsf{a} = x)_1 \vee (\mathsf{b} = x)_1 \Leftrightarrow (\mathsf{a} = x)_2 \vee (\mathsf{b} = x)_2, \tag{1}$$

i.e. the relation that tracks whether σ_1 and σ_2 agree on those predicates. (Subscripts mean that the predicate holds in the correspondingly numbered state.) R_α is a relation on abstract states and summarizes the possible pairs of post-states that will have agreed on return values. For methods push(x)/pop on ArrayStack, we can define an abstraction α with predicates $\{\mathsf{top} \geq 0, \mathsf{A}[\mathsf{top}] = x\}$, and use

$$R_\alpha \equiv (\mathsf{top} \geq 0)_1 = (\mathsf{top} \geq 0)_2 \quad \wedge \quad (\mathsf{A}[\mathsf{top}] = x)_1 = (\mathsf{A}[\mathsf{top}] = x)_2 \tag{2}$$

This abstraction simply tracks that the ArrayStack is non-empty and whether the top element is x or not. Meanwhile, the remaining portion of the state is *identical* between the two states because it came from cloning the reachable initial state. The equivalence reasoning can easily be tracked with direct, inductive equality: a cloned & unmodified frame relation C. For this example, $C \equiv \forall i < \mathsf{top}_1.\ \mathsf{A}_1[i] = \mathsf{A}_2[i]$. We will later see that our algorithms and tools will be able to synthesize these (α, R_α) mn-differencing abstractions/relations and cloned frame C.

While so far we have addressed return values, states that are related by $R_\alpha \wedge C$ may not necessarily be observationally equivalent. We show the pieces fit together by working with observational equivalence *relations*. For reasoning about this equivalence, we use a separate abstraction β, more geared toward relational equivalence, and a relation I_β in that abstract domain. For the ArrayStack and SimpleSet examples, we can use the following such relations:

$$I_{AS}(\sigma_1, \sigma_2) \equiv \mathsf{top}_1 = \mathsf{top}_2 \wedge (\forall i. 0 \leq i \leq \mathsf{top}_1 \Rightarrow \mathsf{A}_1[i] = \mathsf{A}_2[i]) \tag{3}$$

$$I_{SS}(\sigma_1, \sigma_2) \equiv ((\mathsf{a}_1 = \mathsf{a}_2 \wedge \mathsf{b}_1 = \mathsf{b}_2) \ \vee \ (\mathsf{a}_1 = \mathsf{b}_2 \wedge \mathsf{b}_1 = \mathsf{a}_2)) \ \wedge \ (\mathsf{sz}_1 = \mathsf{sz}_2) \tag{4}$$

I_{AS} says that the two states agree on the (ordered) values in the Stack. (top_1 means the value of top in σ_1, etc.) For SimpleSet, I_{SS} specifies that two states are equivalent provided that they are storing the same values—perhaps in different ways—and they agree on the size. These observational equivalence relations can sometimes be inferred and, otherwise, are typically compact. Crucially, however, unlike pre/post specifications, if I is an observational equivalence relation, then it is guaranteed to lead to sound commutativity conclusions. Putting it all together, our decomposition can be posed as a proof rule on the right.

$$\boxed{\dfrac{\begin{array}{ll} (i) & : \{I_\beta\}[s_m]^1(\bar{x}) \mid [s_m]^2(\bar{y})\{I_\beta\} \\ (ii) & : \{Rch \wedge \varphi_m^n\} \begin{smallmatrix}[s_m]^1(\bar{x});\\ [s_n]^1(\bar{x})\end{smallmatrix} \mid \begin{smallmatrix}[s_n]^2(\bar{y});\\ [s_m]^2(\bar{y})\end{smallmatrix} \{R_\alpha \wedge C\} \\ (iii) & : (R_\alpha \wedge C) \implies I_\beta \end{array}}{\varphi_m^n \text{ is a commut. cond. for } m(\bar{x}) \bowtie n(\bar{y})}}$$

The notation $[s_m]^1(\bar{x})$ means the implementation of method m, under a (standard) translation [8,9,41] to act on the σ_1 copy of the state, with arguments $\bar{x}$. Premise (i) incorporates observational equivalence, while (ii) summarizes mn-differencing. Notice that premise (i) does not involve φ_m^n, R_α or C. An outcome of this decomposition is that automated reasoning about I_β (which pertains to all methods of the ADT) can be separated from reasoning about $R_\alpha \wedge C$ (which pertains to a given triple m, n, φ_m^n). Consequently, (i) can be done once globally for the ADT, and then (ii) and (iii) can be done for each new commutativity validity query. Note also that $R_\alpha \wedge C$ is typically *stronger* than I_β (and hence can imply I_β) because it is more specialized to the m/n method-pair under consideration. Meanwhile, I_β is a weaker relation characterizing overall ADT equivalence.

(Section 6). We next describe an improved reduction $\text{DA}\textsc{Reduce}_m^n$, which employs our decomposition. $\text{DA}\textsc{Reduce}_m^n$ emits a pair of automata $\mathcal{A}_{\mathbf{A}}(m, n, \varphi_m^n, I)$ and $\mathcal{A}_{\mathbf{B}}(I)$, such that if we prove both are safe then φ_m^n must be a valid commutativity condition. Again, this separation allows tools to synthesize (α, R_α) and C separately from β and I_β.

(Section 7). Finally, we describe a proof-of-concept implementation of $\textsc{Reduce}_m^n$ and $\text{DA}\textsc{Reduce}_m^n$, and employing Ultimate and CPAchecker as reachability solvers. We report experiments comparing the performance of the two reductions, when applied to some simple ADTs including the above SimpleSet, ArrayStack, Queue and a rudimentary HashTable. While $\text{DA}\textsc{Reduce}_m^n$ has some initial overhead, its use of mn-differencing abstractions appears to enable it to perform better than $\textsc{Reduce}_m^n$.

3 Preliminaries

We work with a simple model of a (sequential) object-oriented language. We will denote an object by o. Objects can have member fields $o.a$ and, for the purposes of this paper, we assume them to be integers, structs or integer arrays. Methods are denoted $o.m(\bar{x}), o.n(\bar{y}), \ldots$ where $\bar{x}$ is a vector of the arguments. We often will omit the o. We use the notation $m(\bar{x})/\bar{r}$ to refer to the return variables $\bar{r}$. We use $\bar{a}$ to denote a vector of argument *values*, $\bar{u}$ to denote a vector of return *values* and $m(\bar{a})/\bar{u}$ or $n(\bar{b})/\bar{v}$ to denote a corresponding invocation of a method which we call an *action*. Methods' source code is parsed from C into control-flow automata (CFA) [23] using assume to represent branching and loops. (See [28] for details on our CFA-based implementation.) Edges are labeled with straight-line ASTs consisting of assume, assignment, and sequential composition. We use s_m to refer to the source code of object method m. For simplicity, we assume that one object method cannot call another, and that all object methods terminate.

Commutativity and Commutativity Conditions. We fix a single object o, denote that object's concrete state space Σ, and assume decidable equality. We denote $\sigma \xrightarrow{m(\bar{a})/\bar{u}} \sigma'$ for the big-step semantics in which the arguments are provided, the entire method is reduced and return values given in $\bar{u}$. For lack of space, we omit the small-step semantics $[\![s]\!]$ of individual statements. For the big-step semantics, we assume that such a successor state σ' is always defined (total) and is unique (determinism). Programs can be transformed so these conditions hold, via wrapping [7] and prophecy variables[2] [3], respectively.

[2] For example, we can use prophecy variables to translate a method such as `int m(a) { if (nondet()) x := a; }` into one that has does not have nondeterminism in its transition system: `int m(a, rho) { if (rho) x := a; }`.

Definition 1 (Observational equivalence for commutativity (*e.g.* [29]). *We define relation $\simeq \subseteq \Sigma \times \Sigma$ as the following greatest fixpoint*

$$\frac{\forall m(\bar{a}) \in M.\ \sigma_1 \xrightarrow{m(\bar{a})/\bar{r}_1} \sigma_1' \quad \sigma_2 \xrightarrow{m(\bar{a})/\bar{r}_2} \sigma_2' \quad \bar{r}_1 = \bar{r}_2 \quad \sigma_1' \simeq \sigma_2'}{\sigma_1 \simeq \sigma_2}$$

The above co-inductive definition expresses that two states σ_1 and σ_2 of an object are observationally equivalent $\simeq$ provided that, when any given method invocation $m(\bar{a})$ is applied to both σ_1 and σ_2, then the respective return values agree. Moreover, the resulting post-states maintain the $\simeq$ relation. A (logical) observational equivalence *relation* I is a formula such that $[\![I]\!] \Rightarrow \simeq$. I_{AS} from the previous section is one such relation. A counterexample to observational equivalence is a finite sequence of method operations $m_1(\bar{a}_1), ..., m_k(\bar{a}_k)$ applied to both σ_1 and σ_2 such that for $m_k(\bar{a}_k)$, the return values disagree, *i.e.*, $\bar{r}_1^k \neq \bar{r}_2^k$.

We next use observational equivalence to define commutativity. As is typical [7,17] we define commutativity first at the layer of an action, which are particular *values*, and second at the layer of a method, which includes a quantification over all of the possible values for the arguments and return variables.

Definition 2 (Commutativity of m *and* n). *For values $\bar{a}, \bar{b}$, we say* **actions** *$m(\bar{a})$ and $n(\bar{b})$ commute, denoted $m(\bar{a}) \bowtie n(\bar{b})$, if for all $\sigma, \bar{u}_1, \bar{u}_2, \bar{v}_1, \bar{v}_2, \sigma_m, \sigma_n, \sigma_{mn}, \sigma_{nm}$ such that $\sigma \xrightarrow{m(\bar{a})/\bar{u}_1} \sigma_m \xrightarrow{n(\bar{b})/\bar{v}_1} \sigma_{mn}$ and $\sigma \xrightarrow{n(\bar{b})/\bar{v}_2} \sigma_n \xrightarrow{m(\bar{a})/\bar{u}_2} \sigma_{nm}$, then $(\bar{u}_1 = \bar{u}_2 \wedge \bar{v}_1 = \bar{v}_2 \wedge \sigma_{mn} \simeq \sigma_{nm})$.* **Methods** *$m$ and n commute denoted $m \bowtie n$ provided that $\forall \bar{a}\ \bar{b}.\ m(\bar{a}) \bowtie n(\bar{b})$.*

The quantification $\forall \bar{a}\ \bar{b}$, etc. means vectors of all possible argument values. Our work extends to a more fine-grained notion of commutativity: an asymmetric version called left-movers and right-movers [34], where a method commutes in one direction and not the other.

We will work with commutativity conditions for methods m and n as logical formulae over initial states and the arguments of the methods. We denote a logical commutativity formula as φ_m^n and assume a computable interpretation of formulae: $[\![\varphi_m^n]\!] : (\sigma, \bar{x}, \bar{y}) \rightarrow \mathbb{B}$. (We tuple the arguments for brevity.) The first argument is the initial state. Commutativity *post-* and *mid*-conditions can also be written over return values [27] but here, for simplicity, we focus on commutativity *pre*-conditions. We may write $[\![\varphi_m^n]\!]$ as φ_m^n when it is clear from context that φ_m^n is meant to be interpreted.

Definition 3 (Commutativity Condition). *Logical formula φ_m^n is a* commutativity condition *for m and n provided that $\forall \sigma\ \bar{a}\ \bar{b}.\ [\![\varphi_m^n]\!]\ \sigma\ \bar{a}\ \bar{b} \Rightarrow m(\bar{a}) \bowtie n(\bar{b})$.*

4 One-Shot Reduction to Reachability

We now take a first stab at the goal of reducing commutativity verification to reachability (*i.e.*, verifying non-reachability of an error location). The problems do not exactly align because commutativity verification is instead defined over an object implementation, method pairs, a formula φ_m^n and a notion of equivalence for objects. We thus pose commutativity as reachability intuitively as follows:

$$\begin{array}{ll}
\text{Pre-condition :} & \left\{\begin{array}{l}
\sigma_1.\mathsf{init}();\\
\mathsf{while}(*)\{[s_m]^1(\bar{a});\ \text{where } m(\bar{a}) \ \mathsf{chosen\ nondeterministically}\ \}\\
\mathsf{assume}(\varphi_m^n(\sigma_1,\bar{a},\bar{b}));\\
\sigma_2 = \sigma_1.\mathsf{clone}();
\end{array}\right.\\
\text{Product :} & \left\{\begin{array}{l|l}
r_m^1 := [s_m]^1(\bar{a}); & r_n^2 := [s_n]^2(\bar{b});\\
r_n^1 := [s_n]^1(\bar{b}); & r_m^2 := [s_m]^2(\bar{a});
\end{array}\right.\\
\text{Post-condition :} & \left\{\begin{array}{l}
\mathsf{assert}(r_m^1 = r_m^2 \wedge r_n^1 = r_n^2);\\
\mathsf{while}(*)\{\ \mathsf{for\ any}\ m(\bar{a})\ \mathsf{chosen\ nondetermistically:}\\
\quad r_1 = [s_m]^1(\bar{a}); r_2 = [s_m]^2(\bar{a});\\
\quad \mathsf{assert}(r_1 = r_2);\}
\end{array}\right.
\end{array}$$

(A formalization can be found in the extended version [28].) Ignoring the pre/-post conditions, in the above quadruple, we have used a product program [8], which encodes two programs (one for each order of method implementations s_m and s_n), each applied to a replica of the state σ, similar to self-composition and other techniques [6,8,9,18–20,40,41].

Strengthening the Pre-condition for Reachable ADT States σ_1 and σ_2. Above the pre-condition: (i) loops, symbolically applying an arbitrary number of method implementations on σ_1, (ii) assumes φ_m^n of the resulting state and (iii) duplicates that state to σ_2. This has the effect that σ_1 and σ_2 will be identical, restricted to only reachable ADT states, and φ_m^n will hold. That is, the precondition can be thought of as: $\{\textbf{Reachable}(\sigma_1) \wedge \varphi_m^n(\sigma_1,\bar{a},\bar{b}) \wedge \sigma_1 = \sigma_2\}$. Verification tools will typically over-approximate **Reachable**.

Weakening the Post-condition to Observational Equivalence. Meanwhile, the post-condition asserts return value agreement, and then loops, symbolically executing a nondeterministically chosen method and argument values on both σ_1 and σ_2, and then asserting that return values agree. Thus the post-condition ensures return value agreement and that there is no sequence of methods that could be applied to both of them, witnessing further disagreement. That is, the postcondition can be thought of as: $\{r_m^1 = r_m^2 \wedge r_n^1 = r_n^2 \wedge \textbf{ObsEq}(\sigma_1,\sigma_2)\}$.

Formally, $\textsc{Reduce}_m^n(\varphi_m^n, m, n, M)$ is a transformation over an input object implementation CFA to an output CFA automaton $\mathcal{A}(\varphi_m^n)$ with an error state q_{er}. We prove that if q_{er} is unreachable in the output encoding $\mathcal{A}(\varphi_m^n)$, then φ_m^n is a valid commutativity condition for m and n. That is, if $\mathcal{A}(\varphi_m^n)$ is safe, then φ_m^n is a commutativity condition. (Detail in the extended version [28]).

Example. Figure 2 is a pseudo-code illustration of $\mathcal{A}(\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)})$, the output generated when $\textsc{Reduce}_m^n$ is applied to methods $\mathsf{add}(x)$ and $\mathsf{isin}\,(y)$ of SimpleSet from Sect. 2. When a candidate formula $\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)}$ is supplied and a program analysis tool for reachability is applied, the tool performs the reasoning necessary for commutativity. In sum, $\textsc{Reduce}_m^n$ uses the implementation of the ADT itself (including other methods such as clear) in order to symbolically represent reachable states s1 and s2 for the Pre-condition and require that the post-state pairs be observationally equivalent in the Post-condition.

Multiple Commutations. Relational reasoning is needed for post-state equivalence but, when commutativity proofs are used in (most) compilers or runtime systems, only one method ordering will actually be executed. The pair-wise commutativity proofs generalize to multiple commutations due to the fact that each possible post-state in one pair's proof is another possible reachable initial state for another pair.

```
SimpleSet s1 = new SimpleSet();                          Pre-cond.
while(*) { int t = *; assume (t>0); switch(*) {
  case 1: [add]^1(t); case 2: [isin]^1(t);
  case 3: [size]^1(); case 4: [clear]^1(); }}
int x = *; int y = *;
assume( φ^{isin(y)}_{add(x)}(s1,x,y) );
SimpleSet s2 = s1.clone();
-------------------------------------------------------------------
r^1_m = [add]^1(x); | r^2_n = [isin]^2(y);                   Quad.
r^1_n = [isin]^1(y); | r^2_m = [add]^2(x);
assert(r^1_m=r^2_m && r^1_n=r^2_n);
-------------------------------------------------------------------
                                                          Post-cond.
while(true){ int t=*; assume(t>0); switch(*) {
  case 1: assert([add]^1(t) == [add]^2(t));
  case 2: assert([isin]^1(t) == [isin]^2(t));
  case 3: assert([clear]^1() == [clear]^2());
  case 4: assert([size]^1() == [size]^2()); } }
```

Fig. 2. REDUCE^n_m applied to add(x)/ isin (y).

While REDUCE^n_m is sound we show in Sect. 7 that tools don't scale well at proving the safety of REDUCE^n_m's output. In the next Sect. 5 we describe an abstraction targeted at proving commutativity to better enable automated reasoning. In the subsequent Sect. 6, we employ that abstraction in an improved reduction DAREDUCE^n_m.

5 Decomposing Commutativity with *mn*-differencing

The problem with reductions like REDUCE^n_m, is that general-purpose reachability tools do not know how to find the right abstraction for commutativity reasoning and those tools end up veering toward searching for unnecessarily intricate abstractions for full-functional verification. We now present a decomposition to mitigate this problem.

Consider the ArrayStackpush(x)/pop example and a (symbolic) state such as $[a, b, c]$ with top $= 2$ and condition $\varphi^{\mathsf{pop}}_{\mathsf{push}(x)} \equiv x = \mathsf{A}[\mathsf{top}]$. When applying a general-purpose reachability solver to REDUCE^n_m, it will consider deep stack values such as a and b because those values could be reachable in a post-state after a further sequence of pop operations. But the solver is actually doing unnecessary work and is not inherently capable of noticing that those deep stack values will be the same, regardless of the order that push(x) and pop are applied.

Even with the sophisticated and automatic abstraction techniques available in today's tools, we do not currently have a notion of what is the *right* abstraction for commutativity and consequently, today's tools often end up diverging searching for an overly precise abstraction. In this section we address this problem and answer the question: how coarse-grained can an abstraction be, while still being fine-grained enough to reason about commutativity?

The idea of mn-differencing can be visualized via the diagram on the right. We start with two states σ_1 and σ_2 that are exactly equal. The product program leads to post states σ_1' and σ_2'. For these post states, we require return value agreement, denoted $\mathcal{X} \equiv r_m^1 = r_m^2 \wedge r_n^1 = r_n^2$. Next, we have an abstraction α, specific to this m/n pair, and a product program in this abstract domain.

$$\begin{tikzcd}
 & \hat{\sigma}_1 \arrow[rr, "\hat{r}_m^1 := \hat{m}(\bar{a}); \hat{r}_n^1 := \hat{n}(\bar{b})"] \arrow[dd, dash, dashed] & & \hat{\sigma}_1' \arrow[dd, dash, dashed, "R_\alpha"] \\
\sigma_1 \arrow[ur, "\alpha"] \arrow[rr, "r_m^1 := m(\bar{a}); r_n^1 := n(\bar{b})"] \arrow[dd, dash, dashed, "(\sigma_1 = \sigma_2)"] & & \sigma_1' \arrow[ur, "\alpha"] \arrow[dd, dash, dashed, "\mathcal{X}"] & \\
 & \hat{\sigma}_2 \arrow[rr, "\hat{r}_n^2 := \hat{n}(\bar{b}); \hat{r}_m^2 := \hat{m}(\bar{a})"'] & & \hat{\sigma}_2' \\
\sigma_2 \arrow[ur, "\alpha"] \arrow[rr, "r_n^2 := n(\bar{b}); r_m^2 := m(\bar{a})"'] & & \sigma_2' \arrow[ur, "\alpha"'] &
\end{tikzcd}$$

The key idea is that (i) relation R_α relates abstract post-states whose return values agree in the abstract domain, and (ii) α is required to be precise enough that return values agree for all state pairs in the concretization of R_α. We can then check whether an initial assumption of φ_m^n on σ_1 implies such an R_α, *i.e.*, checking return value agreement using α which is just precise enough to do so. For $\mathsf{isin}\,(x)/\,\mathsf{clear}$, define an abstraction α with predicates $\{\mathsf{a} = x, \mathsf{a} \neq x, \mathsf{b} = x, \mathsf{b} \neq x\}$ that tracks whether x is in the set. Then

$$R_\alpha^1(\sigma_1, \sigma_2) \equiv (\mathsf{a} = x)_1 \vee (\mathsf{b} = x)_1 \Leftrightarrow (\mathsf{a} = x)_2 \vee (\mathsf{b} = x)_2,$$

i.e. the relation that tracks if σ_1 and σ_2 agree on those predicates. Meanwhile, for $\mathsf{pop}/\mathsf{pop}$ on $\mathsf{ArrayStack}$, we can define a different α with predicates $\{\mathsf{top} > 1, \mathsf{A}[\mathsf{top} - 1] = \mathsf{A}[\mathsf{top}]\}$, and use the relation

$$R_\alpha^2 \equiv (\mathsf{top} > 1)_1 = (\mathsf{top} > 1)_2 \quad \wedge \quad (\mathsf{A}[\mathsf{top} - 1] = \mathsf{A}[\mathsf{top}])_1 = (\mathsf{A}[\mathsf{top} - 1] = \mathsf{A}[\mathsf{top}])_2$$

This relation characterizes state pairs which agree on the stack having at least two elements, and agree that the top and penultimate elements are the same. As we will see, these abstractions and relations, although they are quite weak, are just strong enough so that they capture whether return values will agree.

5.1 Formal Definition

We now formalize mn-differencing. Where noted below, some definitions are omitted and can be found in the extended version [28]. First, we define a set of state pairs denoted $\mathsf{posts}(\sigma, m, \bar{a}, n, \bar{b})$ to be the set of all pairs of post-states (each denoted $(\sigma_{mn}, \sigma_{nm})$) originating from σ after the methods are applied in the two alternate orders:

$$\mathsf{posts}(\sigma, m, \bar{a}, n, \bar{b}) \equiv \{(\sigma_1, \sigma_2) \mid \sigma \xrightarrow{m(\bar{a})/\bar{r}_m^1} \sigma' \xrightarrow{n(\bar{b})/\bar{r}_n^1} \sigma_1 \wedge \sigma \xrightarrow{n(\bar{b})/\bar{r}_n^2} \sigma'' \xrightarrow{m(\bar{a})/\bar{r}_m^2} \sigma_2\}$$

We also define return value agreement denoted $\mathsf{rvsagree}(\sigma, m, \bar{a}, n, \bar{b})$ as a predicate indicating that all such post-states originated from σ agree on return values:

$$\begin{aligned}&\mathsf{rvsagree}(\sigma, m, \bar{a}, n, \bar{b}) \equiv \\ &\quad \forall \bar{r}^1_m, \bar{r}^1_n, \bar{r}^2_n, \bar{r}^2_m \text{ such that } (\sigma \xrightarrow{m(\bar{a})/\bar{r}^1_m} \sigma_1 \xrightarrow{n(\bar{b})/\bar{r}^1_n} \sigma_1' \wedge \sigma \xrightarrow{n(\bar{b})/\bar{r}^2_n} \sigma_2 \xrightarrow{m(\bar{a})/\bar{r}^2_m} \sigma_2')., \\ &\quad \bar{r}^1_m = \bar{r}^2_m \wedge \bar{r}^2_n = \bar{r}^1_n\end{aligned}$$

Definition 4 (*mn*-differencing Abstraction (α, R_α)). *For an object with state space Σ, and two methods m and n. Let $\alpha : \Sigma \to \Sigma^\alpha$ be an abstraction of the states, and $\gamma : \Sigma^\alpha \to \mathcal{P}(\Sigma)$ the corresponding concretization. A relation $R_\alpha \subseteq \Sigma^\alpha \times \Sigma^\alpha$ with its abstraction (α, R_α) is an mn-differencing abstraction if*

$$\begin{aligned}&\forall \sigma_1^\alpha, \sigma_2^\alpha \in \Sigma^\alpha . R_\alpha(\sigma_1^\alpha, \sigma_2^\alpha) \wedge \forall \sigma\ \bar{a}\ \bar{b}.\ \mathit{posts}(\sigma, m, \bar{a}, n, \bar{b}) \in \gamma(\sigma_1^\alpha) \times \gamma(\sigma_2^\alpha) \Rightarrow \\ &\quad \mathit{rvsagree}(\sigma, m, \bar{a}, n, \bar{b})\end{aligned}$$

The above definition requires that α be a precise enough abstraction so that R_α can discriminate in the abstract domain between pairs of post-states where return values will have agreed versus disagreed in the concrete domain.

A relation R_α may not hold for every initial state σ. For example, the above R^2_α for pop/pop does not hold when the stack is empty. Hence, we need to ask whether R_α holds, under the assumption that φ^n_m holds in the pre-condition. We say that φ^n_m *implies* (α, R_α) if

$$\forall \sigma\ \bar{a}\ \bar{b}.\ \varphi^n_m(\sigma, \bar{a}, \bar{b}) \Rightarrow \forall (\sigma_1, \sigma_2) \in \mathsf{posts}(\sigma, m, \bar{a}, n, \bar{b}) \Rightarrow R_\alpha(\alpha(\sigma_1), \alpha(\sigma_2))$$

For SimpleSet isin $(x)/$ clear, if we let $\varphi^{\mathsf{clear}}_{\mathsf{isin}\,(x)} \equiv \mathsf{a} \neq x \wedge \mathsf{b} \neq x$, this will imply R^1_α in the posts. Let's see why. If this commutativity condition $\varphi^{\mathsf{clear}}_{\mathsf{isin}\,(x)}$ holds, then x will not be in the set. Neither method adds x to the set and an abstract domain, tracking only whether $\mathsf{a} = x$ and $\mathsf{b} = x$ hold, will lead to post states that agree on whether x is in the set and this carries over to the agreeing on whether x is in the set in the concrete domain.

Cloned and Untouched Frame. The components of the state that are abstracted away by an *mn*-differencing abstraction include portions of the state that are *unmodified* in either method order (or are both modified in the same way). For example, the deeper elements of ArrayStack remain untouched regardless of the order that methods push and pop are applied. We refer to these state components via a *cloning relation* $C \subseteq \Sigma \times \Sigma$, that we use in conjunction with R_α. Because this relation $C(\sigma_1, \sigma_2)$ always holds when $\sigma_1 = \sigma_2$, and the two method orderings both begin from the same starting point σ, a program analysis can begin with the fact $C(\sigma_0, \sigma_0)$ and then inductively prove that $\mathsf{posts}(\sigma_0, m, \bar{a}, n, \bar{b}) \Rightarrow C$. The cloning relation can instead be thought of as simply a strengthening of R_α, but we present it here separately to emphasize that C captures components of the states that are directly equal, whereas R_α may abstract away unequal components.

Post-state Equivalence. $R_\alpha \wedge C$ is specific to the method pair under consideration and, as such, it can exploit the particular specific effects of the method pair. For example, the R_α for SimpleSet clear/clear can simply say that both sets are empty. On the other hand, these relations alone are not enough to characterize commutativity. States that are R_α-related are not necessarily equivalent. What's needed is to show that method-pair-specific $R_\alpha \wedge C$ relation on the post-states of this method is *strong enough* to imply so that they are observationally equivalent.

We achieve this by using ADT-specific (rather than mn-specific) logical observational equivalence *relations* I_β and separate abstractions I_β there for.

The standard concept of observational equivalence relations [11] is visualized on the right. Importantly, we can use an abstraction β here that is separate from α; this will become useful in the subsequent sections. Formally, I_β is an *observational equivalence relation* iff: $\forall \sigma_1^\beta, \sigma_2^\beta \in \Sigma^\beta.\ I_\beta(\sigma_1^\beta, \sigma_2^\beta) \Rightarrow \forall \sigma_1 \in \delta(\sigma_1^\beta), \sigma_2 \in \delta(\sigma_2^\beta).\ \sigma_1 \simeq \sigma_2$. Relations I_{SS} and I_{AS}, defined earlier, are such relations.

$$\begin{tikzcd}
& \hat\sigma_1 \arrow[rr, "\hat m'(\bar a)/r_1"] \arrow[dd, dashed, "I_\beta"] & & \hat\sigma'_1 \arrow[dd, dashed, "(r_1 = r_2) \wedge I_\beta"] \\
\sigma_1 \arrow[ur, "\beta"] \arrow[rr, "m'(\bar a)/r_1"] \arrow[dd, dashed] & & \sigma'_1 \arrow[ur, "\beta"] \arrow[dd, dashed] & \\
& \hat\sigma_2 \arrow[rr, "\hat m'(\bar a)/r_2"] & & \hat\sigma'_2 \\
\sigma_2 \arrow[ur, "\beta"] \arrow[rr, "m'(\bar a)/r_2"] & & \sigma'_2 \arrow[ur, "\beta"] &
\end{tikzcd}$$

5.2 Connecting the Pieces Together

Finally, we connect R_α and C with I_β and show that they can be used to reason about whether φ_m^n is a valid commutativity condition. The idea is summarized in the proof rule on the right. (Soundness of the rule is given in the extended version [28].) The first judgment (i), presented as a Hoare quadruple, ensures that I_β is an observational equivalence relation. This judgment can be concluded once per ADT and, subsequently, I_β can be used repeatedly, whenever we wish to verify a new commutativity condition via the other judgments. The second judgment (ii) starts from a reachable ADT state where the commutativity condition φ_m^n holds, and has a post-relation $R_\alpha \wedge C$ consisting of an mn-differencing relation, along with a *cloned, untouched frame* C. Finally judgment (iii) combines the mn-differencing abstraction R_α with the cloned aspects of the state C to imply I_β.

$$\boxed{\dfrac{\begin{array}{ll} (i) & : \{I_\beta\}[s_m]^1(\bar x) \mid [s_m]^2(\bar y)\{I_\beta\} \\ (ii) & : \{Rch \wedge \varphi_m^n\} \begin{smallmatrix}[s_m]^1(\bar x);\\ [s_n]^1(\bar x)\end{smallmatrix} \mid \begin{smallmatrix}[s_n]^2(\bar y);\\ [s_m]^2(\bar y)\end{smallmatrix} \{R_\alpha \wedge C\} \\ (iii) & : (R_\alpha \wedge C) \implies I_\beta \end{array}}{\varphi_m^n \text{ is a commut. cond. for } m(\bar x) \bowtie n(\bar y)}}$$

Although an $R_\alpha \wedge C$ may *imply* an I_β, this does not mean that R_α is itself an observational equivalence relation. $R_\alpha \wedge C$ is typically stronger than I_β, but specific to the method-pair. For clear / clear , $R_\alpha \wedge C$ could relate SimpleSet states that are empty. While this implies the SimpleSet observational equivalence relation I_{SS} (Eq. 4 in Sect. 2), this R_α is of course *not* an observational equivalence relation: as soon as add(x) is added to both states, the relation is violated. What's important is simply that R_α implies I_β and, *separately*, that I_β itself is an observational equivalence relation.

Semantically, this decomposition can always be done because we can use $\simeq$ as the notion of observational equivalence and an overly precise R_α. Logically, however, completeness depends on whether a logical observational equivalence relation exists and can be expressed in the assertion language. We leave delving into the details of the assertion language (*e.g.* heap logics) as future work.

6 *mn*-differencing for Automata-Based Verification

As we show in Sect. 7, Reduce_m^n given in Sect. 4 yields encodings for which general-purpose reachability solvers quickly diverge: the abstractions they search for become tantamount to what's needed for full-functional verification. In this section, we employ mn-differencing to introduce an improved reduction called DAReduce_m^n that decomposes the reasoning into two phases: (**A**) finding a sufficient R_α and frame C that implies I_β and then (**B**) proving that I_β is an observational equivalence relation. The output of DAReduce_m^n are a pair of output automata $\mathcal{A}_\mathbf{A}(m, n, \varphi_m^n, I)$ and $\mathcal{A}_\mathbf{B}(I)$, which informally can be thought of as follows:

$\underline{\mathcal{A}_\mathbf{A}(m, n, \varphi_m^n, I)}$

```
$\sigma_1$.init();
while($*$){$\sigma_1.m(\bar{a})$; where $m(\bar{a})$ chosen nondet.}
assume($\varphi_m^n(\sigma_1, \bar{a}, \bar{b})$);   $\sigma_2 = \sigma_1$.clone();
$r_m^1 := \sigma_1.m(\bar{a})$; | $r_n^2 := \sigma_2.n(\bar{b})$;
$r_n^1 := \sigma_1.n(\bar{b})$;  | $r_m^2 := \sigma_2.m(\bar{a})$;
assert($r_m^1 = r_m^2 \wedge r_n^1 = r_n^2$); //$R_\alpha$
assert($I(\sigma_1, \sigma_2)$); //$R_\alpha \wedge C \implies I$
```

$\underline{\mathcal{A}_\mathbf{B}(I)}$

```
assume($I(\sigma_1, \sigma_2)$);
let $m(\bar{a})$ chosen nondet. in
  $r_1 = \sigma_1.m(\bar{a}); r_2 = \sigma_2.m(\bar{a})$;
assert($r_1 = r_2 \wedge I(\sigma_1, \sigma_2)$);
```

DAReduce_m^n is formalized as a transformation over CFAs in the extended version [28]. Unlike Reduce_m^n, $\mathcal{A}_\mathbf{A}(m, n, \varphi_m^n, I)$ ends early with assertions that return values agree and that I must hold. Thus, an analysis on $\mathcal{A}_\mathbf{A}(m, n, \varphi_m^n, I)$ will construct an abstraction α and an mn-differencing relation R_α, as well as a cloned frame C such that $R_\alpha \wedge C \Rightarrow I$. Meanwhile, $\mathcal{A}_\mathbf{B}(I)$ is designed so that a safety proof on $\mathcal{A}_\mathbf{B}(I)$ entails that I is an observational equivalence relation. A pre-condition that assumes I, and then a nondeterministic choice of any ADT method m with nondeterministically selected method arguments $\bar{a}$. To prove that I is an observational equivalence relation, a reachability solver will synthesize an appropriate abstraction β for I in $\mathcal{A}_\mathbf{B}(I)$. If both $\mathcal{A}_\mathbf{A}(m, n, \varphi_m^n, I)$ and $\mathcal{A}_\mathbf{B}(I)$ are safe, then φ_m^n is a valid commutativity condition (as shown in the extended version [28].)

DAReduce_m^n improves over Reduce_m^n by decomposing the verification problem with separate abstraction goals, making it more amenable to automation (see Sect. 7). Moreover, as in the proof rule (Sect. 5), a proof of safety of $\mathcal{A}_\mathbf{B}(I)$ can be done once for the entire ADT. Then, for a given method pair and candidate condition φ_m^n, one only needs to prove the safety of $\mathcal{A}_\mathbf{A}(m, n, \varphi_m^n, I)$.

Automation. Synthesis of α, R_α, C and β is automated. The definition of $\mathcal{A}_\mathbf{B}(I)$ can be amended so that a reachability solver could potentially infer I. The below amended version encodes the search for a (relational) observational equivalence as the search for a (non-relational) loop invariant.

$$\text{amended } \mathcal{A}_{\mathbf{B}}(I) : \begin{cases} \mathsf{while(true)\{let}\;\; m(\bar{a})\;\; \mathsf{chosen\ nondeterministically\ in} \\ \quad r_1 = \sigma_1.m(\bar{a}); r_2 = \sigma_2.m(\bar{a}); \\ \quad \mathsf{assert}(r_1 = r_2);\} \end{cases}$$

7 Evaluation

Our goals were to evaluate (1) whether mn-differencing abstractions ease commutativity verification, *i.e.*, whether DAReduce^n_m outperforms Reduce^n_m, and (2) how automated our strategy can be.

We implemented a proof-of-concept tool called CityProver[3]. CityProver takes, as input, C-style source code, using structs for object state. Examples are included with the CityProver release. We have written them as C macros so that our experiments focus on commutativity rather than testing existing tools' inter-procedural reasoning power. Also provided as input to CityProver is a commutativity condition φ^n_m and the method names m and n. CityProver then implements Reduce^n_m and DAReduce^n_m via a program transformation.

Experiments. We created some small examples (with integers, structs and arrays) and ran CityProver on them. Our experiments were run on a Quad-Core Intel(R) Xeon(R) CPU E3-1220 v6 at 3.00 GHz, inside a QEMU VM. We began with single-field objects including: (M) a Memory cell; (A) an Accumulator with increment, decrement, and a check whether the value is 0; and (C) a Counter that also has a clear method. For each object, we considered some example method pairs with both a valid commutativity condition and an incorrect commutativity condition (to check that the tool discovers a counterexample).

ADT	Methods	$\varphi^{n(y_1)}_{m(x_1)}$	Exp.	Reduce^n_m		DAReduce^n_m	
				cpa	**ult**	**cpa**	**ult**
M	rd ⋈ wr	$\mathsf{s1.x} = y_1$	✓	1.4 ✓	0.7 ✓	3.9 ✓	1 ✓
M	rd ⋈ wr	true	χ	1.4 χ	0.2 χ	1.3 χ	0.2 χ
M	wr ⋈ wr	$y_1 = x_1$	✓	1.4 ✓	0.5 ✓	3.9 ✓	0.8 ✓
M	wr ⋈ wr	true	χ	1.3 χ	0.3 χ	2.4 χ	0.4 χ
M	rd ⋈ rd	true	✓	1.4 ✓	0.6 ✓	3.9 ✓	1 ✓
A	dec ⋈ isz	$\mathsf{s1.x} > 1$	✓	1.5 ✓	2.2 ✓	4 ✓	2.6 ✓
A	dec ⋈ isz	true	χ	1.5 χ	0.7 χ	1.2 χ	0.6 χ
A	dec ⋈ inc	$\mathsf{s1.x} > 1$	✓	1.5 ✓	1.3 ✓	4.1 ✓	1.7 ✓
A	dec ⋈ inc	true	✓	1.5 ✓	1.2 ✓	4 ✓	1.5 ✓
A	inc ⋈ isz	$\mathsf{s1.x} > 1$	✓	1.5 ✓	3.3 ✓	4.1 ✓	2.9 ✓
A	inc ⋈ isz	true	χ	1.6 χ	0.7 χ	1.2 χ	0.6 χ
A	inc ⋈ inc	true	✓	1.4 ✓	1.5 ✓	4.1 ✓	1.6 ✓
A	dec ⋈ dec	true	✓	1.5 ✓	1.5 ✓	3.9 ✓	1.6 ✓
A	dec ⋈ dec	$\mathsf{s1.x} > 1$	✓	1.5 ✓	2.6 ✓	4 ✓	1.9 ✓
A	isz ⋈ isz	true	✓	1.4 ✓	4.3 ✓	4 ✓	3.4 ✓
C	dec ⋈ dec	true	χ	1.9 χ	1.5 χ	4.2 ✓✗	1.2 χ
C	dec ⋈ dec	$\mathsf{s1.x} \geq 2$	✓	1.5 ✓	13.0 ✓	4.1 ✓	5.9 ✓
C	dec ⋈ inc	true	χ	1.6 χ	0.3 χ	1.4 χ	0.3 χ
C	dec ⋈ inc	$\mathsf{s1.x} \geq 1$	✓	1.6 ✓	6.8 ✓	4.2 ✓	3.8 ✓
C	inc ⋈ isz	true	χ	1.5 χ	0.8 χ	1.2 χ	0.7 χ
C	inc ⋈ isz	$\mathsf{s1.x} > 0$	✓	1.5 ✓	5.3 ✓	4.1 ✓	2.6 ✓
C	inc ⋈ isz	$\mathsf{s1.x} > 0$	χ	1.9 **?**	**TO ?**	4.4 χ	6.9 χ
C	inc ⋈ clr	true	χ	1.3 χ	0.4 χ	2.5 χ	0.4 χ

Fig. 3. Verifying commutativity properties of simple benchmarks. For each, we report time to use Reduce^n_m vs. DAReduce^n_m. A more detailed table is in the extended version [28].

[3] https://github.com/erickoskinen/cityprover.

The objects, method pairs and commutativity conditions are shown in the first few columns of Fig. 3, along with the **Exp**ected result. We used both the Reduce^n_m (Sect. 4) and DAReduce^n_m (Sect. 6) algorithms and, in each case, compared using CPAchecker [10] and Ultimate [22] as the solver. For DAReduce^n_m, we report the total time taken for both Phase **A** and Phase **B**. A more detailed version of this table can be found in the extended version [28]. Benchmarks for which **A** succeed can all share the results of a single run of Phase **B**; meanwhile, when **A** fails, the counterexample can be found without needing **B**. These experiments confirm we can verify commutativity conditions from source. In one case, CPAchecker returned an incorrect result. While DAReduce^n_m often takes slightly more time (due to the overhead of starting up a reachability analysis twice), it does not suffer from a timeout (in the case of Counter inc / isz).

We next turned to simple data structures that store and manipulate elements. While *mn*-differencing and DAReduce^n_m support parametric/unbounded ADTs, automated reasoning about the cloned frame C typically requires quantified invariants. Automata reachability tools typically do not currently have robust support quantifiers, so we evaluate these ADTs with a fixed size. We mainly used Ultimate as we had trouble tuning CPAchecker (perhaps owing to our limited experience). In some cases (marked in blue), Ultimate failed to produce a timely response for either reduction, so we tried CPAchecker instead. Figure 4 shows the results of applying Reduce^n_m and DAReduce^n_m on these examples. In each example, we first list the running time for DAReduce^n_m's ADT-specific Phase **B**, and then list the times for Phases **A**i and **A**ii, as well as the total time.

For (SS) SimpleSet (Fig. 1), in almost all cases DAReduce^n_m outperformed Reduce^n_m, with an average speedup of **3.88**×. For

ADT	**Methods**		**Exp.**	Redu^n_m	DARe^n_m
	$m(x_1), n(y_1)$	$\varphi^{n(y_1)}_{m(x_1)}$		**Ult**	**Ult**
SS	isin ⋈ isin	true	✓	137.8 ✓	36.7 ✓
SS	isin ⋈ add	$x_1 \neq y_1$	✓	84.8 ✓	37.1 ✓
SS	isin ⋈ add	true	χ	2.4 χ	1.5 χ
SS	isin ⋈ clear	true	χ	2.6 χ	1.6 χ
SS	isin ⋈ clear	$x_1 \neq y_1$	χ	2.4 χ	1.6 χ
SS	isin ⋈ clear	$\mathsf{a} \neq x_1 \wedge \mathsf{b} \neq y_1$	χ	3.1 χ	1.4 χ
SS	isin ⋈ clear	$\mathsf{a} \neq x_1 \wedge \mathsf{b} \neq x_1$	✓	14.0 ✓	19.3 ✓
SS	isin ⋈ getsize	true	✓	41.3 ✓	24.2 ✓
AS	push ⋈ pop	$\mathsf{A}[\mathsf{top}] = x_1 \wedge \mathsf{top} > 1 \wedge \mathsf{top} < 4$	✓	**MO** –	95.5 ✓
AS	push ⋈ pop	true	χ	2.2 χ	2.0 χ
AS	push ⋈ push	true	χ	7.6 χ	17.0 χ
AS	push ⋈ push	$\mathsf{top} < 3$	χ	3.9 χ	230.7 χ
AS	push ⋈ push	$x_1 = y_1$	χ	19.8 χ	17.3 χ
AS	push ⋈ push	$x_1 = y_1 \wedge \mathsf{top} < 3$	✓	**MO** –	155.1 ✓
AS	pop ⋈ pop	$\mathsf{top} = -1$	✓	**TO** –	38.0 ✓
AS	pop ⋈ pop	true	χ	2.1 χ	1.4 χ
Q	enq ⋈ enq	true	χ	39.8 χ	35.0 χ
Q	deq ⋈ deq	true	χ	3.5 χ	3.1 χ
Q	deq ⋈ deq	$size = 0$	✓	**TO** –	174.4 ✓
Q	enq ⋈ enq	true	χ	27.8 χ	23.3 χ
Q	enq ⋈ enq	$x_1 = y_1$	χ	63.8 χ	22.6 χ
Q	emp ⋈ emp	true	✓	**TO** –	**TO** –
Q	enq ⋈ deq	$size = 1 \wedge x_1 = \mathsf{A}[front]$	✓	**TO** –	472.0 ✓
Q	enq ⋈ deq	true	χ	**MO** –	8.4 χ
Q	enq ⋈ emp	$size > 0$	✓	**MO** –	**TO** –
Q	enq ⋈ emp	true	χ	**MO** –	7.4 χ
Q	deq ⋈ emp	$size = 0$	✓	**MO** –	135.8 ✓
Q	deq ⋈ emp	true	χ	**MO** –	6.5 χ
HT	put ⋈ put	${}^1\varphi^{\mathsf{put}}_{\mathsf{put}}$	χ	262.5 χ	97.5 χ
HT	put ⋈ put	${}^2\varphi^{\mathsf{put}}_{\mathsf{put}}$	χ	202.7 χ	136.9 χ
HT	put ⋈ put	${}^3\varphi^{\mathsf{put}}_{\mathsf{put}}$	✓	**TO** –	**TO** –
HT	put ⋈ put	${}^3\varphi^{\mathsf{put}}_{\mathsf{put}}$	✓	566.5 ✓	297.5 ✓
HT	put ⋈ put	true	χ	**TO** –	102.3 χ
HT	get ⋈ get	$\mathsf{keys} = 0$	✓	**TO** –	**TO** –
HT	get ⋈ get	true	✓	**TO** –	**TO** –
HT	get ⋈ get	true	✓	50.0 ✓	56.8 ✓
HT	get ⋈ put	$x_1 \neq y_1$	✓	**TO** –	**TO** –
HT	get ⋈ put	true	χ	1.3 χ	0.9 χ

Fig. 4. Results of applying CityProver to ArrayStack, SimpleSet and Queue. A more detailed breakdown of DAReduce^n_m can be found in the extended version [28].

(AS) ArrayStack (Fig. 1), Reduce_m^n found some counterexamples quickly. However, in the other cases Reduce_m^n ran out of memory, while DAReduce_m^n was able to prove all cases. For (Q) Queue, we implemented a simple array-based queue and were able to verify all but two commutativity conditions. Finally, we implemented a rudimentary (HT) HashTable, in which hashing is done only once and insertion gives up if there is a collision. Some commutativity conditions are as follows:

$$^1\varphi_{\mathsf{put}}^{\mathsf{put}} \equiv x_1 \neq y_1, \qquad ^2\varphi_{\mathsf{put}}^{\mathsf{put}} \equiv x_1 \neq y_1 \wedge \mathsf{tb}[x_1\%\mathsf{cap}].\mathsf{k} = -1 \wedge \mathsf{tb}[y_1\%\mathsf{cap}].\mathsf{k} = -1$$
$$^3\varphi_{\mathsf{put}}^{\mathsf{put}} \equiv x_1 \neq y_1 \wedge x_1\%\mathsf{cap} \neq y_1\%\mathsf{cap} \wedge \mathsf{tb}[x_1\%\mathsf{cap}].\mathsf{k} = -1 \wedge \mathsf{tb}[y_1\%\mathsf{cap}].\mathsf{k} = -1$$

For HT, Ultimate timed out on Phase **B** and in some cases had some trouble mixing modulus with array reasoning, so we used CPAchecker. We still used Ultimate in some Phase **A** cases, because it can report a counterexample in Phase **A** even if it timed out in **B**. We also could use Ultimate for Phase **A**, given that CPAchecker already proved Phase **B**, with the same I_β. We also had to introduce a prophecy variable to assist the verifiers in knowing that array index equality distributes over modulus of equal keys.

Overall, for Reduce_m^n there were 15 cases where it reached the 15-minute timeout or out-of-memory. DAReduce_m^n performed better: it only reached the timeout in 6 cases. In 24 cases (out of 37), CityProver returned a proof or counterexample in under 2 min. In summary, these experiments confirm that DAReduce_m^n improves over Reduce_m^n: in most cases it is faster, sometimes by as much as $2\times$ or $3\times$. In **7** cases, DAReduce_m^n is able to generate an answer, while Reduce_m^n suffers from a timeout/memout. (Timeouts typically occurred during refinement loops.)

In all examples, our implementation inferred α, R_α, C and β. For those in Fig. 4, we provided I manually. For the Queue and HashTable, we used (fixed size versions of) the following:

$$I_Q \equiv \mathsf{front}_1 = \mathsf{front}_2 \wedge \mathsf{rear}_1 = \mathsf{rear}_2 \wedge \mathsf{sz}_1 = \mathsf{sz}_2 \wedge \forall i \in [\mathsf{front}_1, \mathsf{rear}_1].\mathsf{q}_1[i] = \mathsf{q}_2[i]$$
$$I_{HT} \equiv \mathsf{keys}_1 = \mathsf{keys}_2 \wedge \forall i \in [0, \mathsf{max}).\mathsf{tb}_1.k \geq 0 \Rightarrow \mathsf{tb}_1[i].k = \mathsf{tb}_2[i].k \wedge \mathsf{tb}_1[i].v = \mathsf{tb}_2[i].v$$

I_Q states that the queues have the same size, and that the values agree in the range of the queue. It is possible to weaken this relation but commutativity does not need this weakening. For the HashTable, I_{HT} states that the HashTables have the same number of keys and, in each non-empty slot, they agree on the key and value. Apart from these I relations (which someday could be inferred) our technique is otherwise completely automated: a user only provides guesses for the commutativity conditions and CityProver returns a proof or counterexample.

Working with Observational Equivalence (Obs-Eq) Relations. As compared to pre/post specifications, observational equivalence relations are simpler to work with and do not suffer from the potential to lead to unsound commutativity conclusions. There are several points to consider. *Soundness.* If a relation is an obs-eq relation then it is guaranteed to be precise enough for commutativity proofs (Thm 5.1). By contrast (see Sect. 2) pre/post specifications run the risk of being too coarse grained (and then unsound commutativity conclusions) or too fine grained (accounting for unnecessary detail). *Simplicity.* With an obs-eq relation, we only need to reason about the structure of the abstraction described by that relation. By contrast, pre/post specifications may be superfluous or unnecessarily detailed for commutativity (*e.g.* post-condition of HashTable.clear). Methods with branching or loops quickly veer toward detailed disjunctive post-conditions but, for commutativity, it only matters that an obs-eq relation holds. Even with Queue.enq, there are three cases, but these are unneeded in the obs-eq relation. *Centralized.* Unlike specs, a single obs-eq relation applies to all methods, so they are more centralized and typically less verbose. *Automation.* We feel that inferring an obs-eq relation is a more well-defined and achievable goal, akin to how numerous other verification techniques/tools prefer to synthesize loop invariants rather than synthesizing specifications. Also, many specification inference tools, to be tractable, end up with shallow specifications which, for commutativity, runs the unsoundness risk. *Usability.* We aim to make commutativity verification accessible to non-experts and, given the above mentioned unsoundness risk with imprecise specifications, asking them to write pre/post conditions is perhaps not the best strategy. Even if the non-expert succeeds in writing a correct pre/post condition, they can still lead to unsound conclusions about commutativity.

Experience. In some cases CityProver caught our mistakes/typos. We also tried to use CityProver to help us narrow down on a commutativity condition via repeated guesses. In the HashTable example the successive conditions ${}^{i}\varphi_{\mathsf{put}}^{\mathsf{put}}$ (defined in the extended version [28].) represent our repeated attempts to guess commutativity conditions. CityProver's counterexamples pointed out collisions and capacity cases. Commutativity conditions are applied in practice through the use of commutativity-based formats such as abstract locking [24], access point specifications [17] and conflict abstractions [16].

Summary. With Reduce_m^n, tools often struggle to converge on appropriate abstractions but we show that DAReduce_m^n (employing mn-differencing) leads to a more plausible algorithmic strategy: DAReduce_m^n can promptly validate commutativity conditions for 31 out of 37 examples. An important direction for future work is to further improve performance and scalability.

8 Related Work

To our knowledge, mn-differencing and reductions based on mn-differencing (*e.g.* DAReduce_m^n) have not occurred in the literature. We now survey related works on commutativity reasoning, k-safety, product programs, etc., beyond those that we have already mentioned.

Commutativity Reasoning. Bansal *et al.* [7] synthesize commutativity conditions from provided pre/post specifications, rather than implementations. They assume these specifications are precise enough to faithfully represent all effects relevant to commutativity. As discussed in Sect. 2, if specifications are coarse, Bansal *et al.* would emit unsound commutativity conditions. By contrast, our relations capture just what is needed for commutativity. Gehr *et al.* [21] describe a method based on black-box sampling, but lack a soundness guarantee. Both Aleen and Clark [4] and Tripp *et al.* [42] identify sequences of actions that commute (via random interpretation and dynamic analysis, resp.). Kulkarni et al. [32] point out that degrees of commutativity specification precision are useful. Kim and Rinard [27] verify commutativity conditions from specifications. Commutativity is also used in dynamic analysis [17]. Najafzadeh *et al.* [35] describe a tool for weak consistency, that reports commutativity checking of formulae, but not ADT implementations. Houshmand *et al.* [25] describe commutativity checking for replicated data types (CRDTs). This complementary work is geared toward CRDTs written in a high-level specification language (transitions on tuples of Sets) that can be represented in SMT with user-provided invariants.

k-safety, Product programs, Reductions. Self-composition [9,41] reduces some forms of hyper-properties [14] to properties of a single program. More recent works include product programs [8,18] and techniques for automated verification of k-safety properties. Cartesian Hoare Logic [40] is a program logic for reasoning about k-safety properties, automated via a tool called DESCARTES. Antonopoulos *et al.* [5] described an alternative automated k-safety technique based on partitioning the traces within a program. Farzan and Vandikas [19] discuss a technique and tool WEAVER for verifying hypersafety properties, based on the observation that a proof of some representative runs in a product program can be sufficient to prove that the hypersafety property holds of the original program. Others explore logical relational reasoning across multiple programs [6,20].

9 Discussion and Future Work

We have described a theory (mn-differencing), algorithm (DAREDUCE^n_m) and tool for decomposing commutativity verification of ADT implementations.

mn-differencing can be instantiated to reason about heap ADTs by using, *e.g.*, separation logic [37,38] as an assertion language. Using the separating conjunction, we can frame the mn-differencing relation apart from the cloning relation. For example, we can consider push(x)/pop on a list-based implementation of a stack containing n elements: $stk \mapsto [e_n, s_n] * \cdots * [e_1, \bot]$. We can define mn-differencing R_α to focuses on whether two list-stack states agree on the top element, and frame the rest with a relation C that specifies exact (shape and value) equivalence. It is unclear whether DAREDUCE^n_m is the right strategy for automating mn-differencing heap assertions; integrating mn-differencing into heap-based tools (*e.g.* [1,2,26]) is an interesting direction for future work.

The results of our work can be used to incorporate more commutativity conditions soundly and obtain speed ups in transactional object systems [16,24]. Further research is needed to use our commutativity proofs with parallelizing compilers. Specifically, in the years to come, parallelizing compilers could combine our proofs of commutativity with automated proofs of linearizability [12] to execute more code concurrently and safely.

Acknowledgments. We would like to thank Marco Gaboardi, Maurice Herlihy, Michael Hicks, David Naumann and the anonymous reviewers for their helpful feedback on earlier drafts of this work. This work was supported in part by NSF award #1813745 and #2008633.

References

1. Infer static analyzer. https://fbinfer.com/
2. Verifast. https://github.com/verifast/verifast
3. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**, 253–284 (1991)
4. 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) (2009), ACM, pp. 241–252 (2009)
5. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 362–375. ACM (2017)
6. Banerjee, A., Naumann, D. A., and Nikouei, M. Relational logic with framing and hypotheses. In: 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
7. Bansal, K., Koskinen, E., Tripp, O.: Automatic generation of precise and useful commutativity conditions. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 115–132. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_7
8. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
9. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSFW (2004)
10. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
11. Bolognesi, T., Smolka, S.A.: Fundamental results for the verification of observational equivalence: a survey. In: PSTV, pp. 165–179 (1987)
12. Bouajjani, A., Emmi, M., Enea, C., Hamza, J. On reducing linearizability to state reachability. In Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, 6–10 July 2015, Proceedings, Part II, pp. 95–107 (2015)
13. Chechik, M., Stavropoulou, I., Disenfeld, C., Rubin, J.: FPH: efficient non-commutativity analysis of feature-based systems. In: Russo, A., Schürr, A. (eds.) FASE 2018. LNCS, vol. 10802, pp. 319–336. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89363-1_18
14. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)

15. 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)
16. Dickerson, T.D., Koskinen, E., Gazzillo, P., Herlihy, M.: Conflict abstractions and shadow speculation for optimistic transactional objects. In: Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, 1–4 December 2019, Proceedings, pp. 313–331 (2019)
17. 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 2014) (2014)
18. Eilers, M., Müller, P., Hitz, S.: Modular product programs. ACM Trans. Programm. Lang. Syst. (TOPLAS) **42**(1), 1–37 (2019)
19. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 200–218. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_11
20. Frumin, D., Krebbers, R., Birkedal, L.: Reloc: a mechanised relational logic for fine-grained concurrency. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 442–451 (2018)
21. Gehr, T., Dimitrov, D., Vechev, M. T. Learning commutativity specifications. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, 18–24 July 2015, Proceedings, Part I, 307–323 (2015)
22. Heizmann, M., et al.: Ultimate automizer with SMTInterpol. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 641–643. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_53
23. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, 27–31 July 2002, Proceedings, pp. 526–538 (2002)
24. 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 2008) (2008)
25. Houshmand, F., Lesani, M.H.: replication coordination analysis and synthesis. Proc. ACM Program. Lang. **3**(POPL), 74:1–74:32 (2019)
26. Juhasz, U., Kassios, I.T., Müller, P., Novacek, M., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. Technical report, ETH Zurich (2014)
27. 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, ACM, pp. 528–541 (2011)
28. Koskinen, E., Bansal, K.: Reducing commutativity verification to reachability with differencing abstractions. CoRR abs/2004.08450 (2020)
29. Koskinen, E., Parkinson, M.J.: The push/pull model of transactions. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015, pp. 186–195 (2015)
30. Koskinen, E., Parkinson, M.J., Herlihy, M.: Coarse-grained transactions. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, pp. 19–30 (2010)
31. Kragl, B., Qadeer, S.: Layered concurrent programs. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 79–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_5

32. 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)
33. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), pp. 211–222. ACM (2007)
34. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
35. Najafzadeh, M., Gotsman, A., Yang, H., Ferreira, C., Shapiro, M.: The CISE tool: proving weakly-consistent applications correct. In: Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, 18 April 2016, pp. 2:1–2:3 (2016)
36. Ni, Y., et al.: 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)
37. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
38. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE (2002)
39. Rinard, M.C., Diniz, P.C.: Commutativity analysis: a new analysis technique for parallelizing compilers. ACM Trans. Program. Lang. Syst. (TOPLAS) **19**(6), 942–991 (1997)
40. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Krintz, C., Berger, E. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, 13–17 June 2016, pp. 57–69. ACM (2016)
41. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: SAS (2005)
42. Tripp, O., Manevich, R., Field, J., Sagiv, M.: JANUS: exploiting parallelism via hindsight. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, pp. 145–156 (2012)
43. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_40
44. Yang, H.: Relational separation logic. Theor. Comput. Sci. **375**(1–3), 308–334 (2007)