Reducing Commutativity Verification to Reachability with Differencing Abstractions

Eric Koskinen
Stevens Institute of Technology

Kshitij Bansal
Google

Abstract
Commutativity of data structure methods is of ongoing interest, with roots in the database community. In recent years commutativity has been shown to be a key ingredient to enabling multicore concurrency in contexts such as parallelizing compilers, transactional memory, speculative execution and, more broadly, software scalability. Despite this interest, it remains an open question as to how a data structure’s commutativity specification can be verified automatically from its implementation.

In this paper, we describe techniques to automatically prove the correctness of method commutativity conditions from data structure implementations. We introduce a new kind of abstraction that characterizes the ways in which the effects of two methods differ depending on the order in which they are applied, and abstracts away effects of methods that would be the same regardless of the order. We then describe a novel algorithm that reduces the problem to reachability, so that off-the-shelf program analysis tools can perform the reasoning necessary for proving commutativity. Finally, we describe a proof-of-concept implementation and experimental results, showing that our tool can verify commutativity of data structures such as a memory cell, counter, two-place Set, array-based stack, queue, and a rudimentary hash table. We conclude with a discussion of what makes a data structure’s commutativity provable with today’s tools and what needs to be done to prove more in the future.

1. Introduction
For an object \( o \) with state \( \sigma \) and methods \( m, n \), etc., let \( \bar{x} \) and \( \bar{y} \) denote argument vectors and \( o.m(\bar{x})/\bar{r} \) denote a method signature, including a vector of corresponding return values \( \bar{r} \). Commutativity of two methods, denoted \( o.m(\bar{x})/\bar{r} \triangleright o.n(\bar{y})/\bar{s} \), 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 \( \varphi^m_n(\sigma, \bar{x}, \bar{r}, \bar{y}, \bar{s}) \) indicating, for a given state \( \sigma \), whether the two operations will always commute, as a function of parameters and return values.

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 [33], transactional memory [23, 25, 32], optimistic parallelism [28], speculative execution, features [10], etc. More broadly, a recent paper from the systems community [13] found that, when software fragments are implemented so that they commute, better scalability is achieved. Intuitively, commutativity captures independence and, if two code fragments commute then, when combined with linearizability (for which proof techniques exist, e.g., [8, 36]) they can be executed concurrently. To employ this approach to concurrency 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 [2, 19] or else they rely on data structure specifications as intermediaries [3, 24] (See Section 7).

In this paper, we describe the first automatic method for verifying a given commutativity condition directly from the data structure’s source code. At a first look, this problem seems difficult because data structures can be implemented in many ways and commutativity may seem to necessitate reasoning about full functional correctness. Our first enabling insight is that, unlike elaborate invariants & abstractions needed for verifying full functional correctness, we can achieve commutativity reasoning with specialized abstractions that are more targeted. Given two methods, we introduce an abstraction of states with respect to these methods that allows us to reason safely within the limited range of exploring the differences between the behavior of pairs of operations when applied in either order and abstracting away the state mutations that would be the same, regardless of the order in which they are applied. We call this an \( mn \)-differencing abstraction.

Commutativity reasoning is challenging also because we need to know whether two different concrete post states—one arising from \( m(\bar{x}); n(\bar{y}) \) and one from \( n(\bar{y}); m(\bar{x}) \)—represent the same object state. Toward this challenge, we employ a notion of observational equivalence in our definition of commutativity, enabling us to reason automatically about equivalent post-states, even in the absence of a data structure specification. Crucially, in our observational equivalence relations, we exploit the convenience of using both abstract equality for some parts of the data structure (those that are order dependent such as the top few elements of a stack), as well as direct equality for other parts of the data structure (other effects on, e.g., the remaining region of the stack, that are not order dependent).

Next, we introduce a novel automata-theoretic transformation for automating commutativity reasoning. Our transformation \( E(s_m; s_n) \) takes, as input, data structure implementations \( s_m \) and \( s_n \), the transformation generates a symbolic automaton \( A(\varphi^m_n) \), using a form of product construction to relate \( s_m; s_n \) with \( s_m; s_m \), but designed in a particular way so that, when it is provided a candidate commutativity condition \( \varphi^m_n \), a safety proof on \( A(\varphi^m_n) \) entails that \( \varphi^m_n \) is a commutativity condition. Our encoding coheres a reachability analysis to perform the reasoning necessary for verifying commutativity: finding \( mn \)-differencing abstractions and proving observational equivalence. For example, when applied to a Stack’s \( \text{push}(v) \) and \( \text{pop} \) operations, a predicate analysis would discover predicates including whether the top value of the stack is equal to \( v \) (as well as negations), and track where in the Stack implementation behaviors diverge on the basis of this abstraction.

We implement our strategy in a new tool called CityProver. It takes as input data structures written in a C-like language (with integers, arrays, and some pointers), implements the above described
This condition specifies three situations (disjuncts) in which the involved:

\[ \text{commutativity condition of methods add} \]
\[ \phi \]
\[ \sigma \]

describes the conditions under which two methods and, if so, stores
variable. Method SimpleSet

Consider the

benchmarks for future work to improve those tools.

\[ \text{Limitations.} \]

some further research, they could be combined with linearizability
with existing transactional object systems [17]. Moreover, with

This data structure is a simplification of a Set, and is capable of

of storing up to two natural numbers using private integers

This encoding and then employs Ultimate’s [20] or CPAchecker’s [6]
reachability analyses to prove commutativity. We show our approach
to work on simple numeric data structures such as Counter,
Memory, SimpleSet, ArrayStack, ArrayQueue and a hash table.

We consider the usability benefits of our strategy. First, it can be applied to ad hoc data structures for which the specification is not readily available. Also, unlike data structure specifications, commutativity conditions can be fairly compact, written as smaller
categorical formulae. Consequently, with CityProVer, a user can
guess commutativity conditions and rely on our machinery to either
prove them correct or find a counterexample; we discuss our use of
this methodology.

Our experiments confirm that our technique permits out-of
the-box reachability tools to verify some ADTs. They also reveal
the current limitations of those tools to support ADTs in which
equivalence is up to permutations. This leads to a blowup in the

Disjunctive reasoning that would be necessary, and an important
question for future work.

Contributions. We present the first automated technique for verifying given commutativity conditions directly from data structure
implementations.

• We formalize \( mn \)-differencing abstractions and observational
equivalence relations for commutativity.

• We present a symbolic technique for reducing commutativity
verification to reachability.

• We build a prototype tool CityProVer, which generates a
proof (or finds a counterexample) that \( \varphi_{mn}^{\text{in}} \) is a commutativity condition.

• We demonstrate that CityProVer can prove commutativity of
simple data structures including a memory cell, counter, two
place Set, array stack, array queue and hash table.

Our verified commutativity conditions can immediately be used
with existing transactional object systems [17]. Moreover, with
some further research, they could be combined with linearizability
proofs and used inside parallelizing compilers.

Limitations. In this paper we have focused on numeric programs.
However, \( mn \)-differencing abstractions and our reduction strategy
appear to generalize to heap programs, left for future work. Also,
while our reduction appears to be general, we were limited by the
reasoning power of existing reachability tools, specifically, the
need for permutation invariants. Our work therefore establishes
benchmarks for future work to improve those tools.

1.1 Motivating Examples

Consider the SimpleSet data structure shown at the top of Fig. 1.
This data structure is a simplification of a Set, and is 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 \( \text{add}(x) \) checks to see if there is space available
and, if so, stores \( x \) where space is available. Methods \( \text{isin}(y), \text{getsize}() \) and \( \text{clear}() \) are straightforward.

A commutativity condition, written as a logical formula \( \varphi_{mn}^{\text{in}} \),
describes the conditions under which two methods \( n(x) \) and \( n(y) \)
commute, in terms of the argument values and the state of the
data structure \( \sigma \). Two methods \( \text{isin}(x) \) and \( \text{isin}(y) \) always commute
because neither modifies the ADT, so we say \( \varphi_{\text{isin}}^{\text{in}}(y) \equiv \text{true} \). The
commutativity condition of methods \( \text{add}(x) \) and \( \text{isin}(y) \) is more involved:

\[ \varphi_{\text{isin}}^{\text{in}}(y) \equiv x \neq y \lor (x = y \land a = x) \lor (x = y \land b = x) \]

This condition specifies three situations (disjuncts) in which the
two operations commute. In the first case, the methods are oper-
ating on different values. Method \( \text{isin}(y) \) 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, \( \text{add}(x) \) will either
succeed or not (depending on whether space is available) and this
operation will not be affected by \( \text{isin}(y) \). 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. Other commutativity conditions include:

\[ \varphi_{\text{isin}}^{\text{clear}}(y) \equiv (a \neq y \land b \neq y), \varphi_{\text{isin}}^{\text{getsize}}(y) \equiv \text{true}, \varphi_{\text{add}}^{\text{getsize}}(x) \equiv \text{false}, \]
\[ \varphi_{\text{isin}}^{\text{clear}}(y) \equiv \text{isin}(y) \equiv \text{true}, \varphi_{\text{add}}^{\text{getsize}}(x) \equiv \text{false}, \]
\[ \varphi_{\text{isin}}^{\text{clear}}(y) \equiv \text{isin}(y) \equiv \text{true}, \varphi_{\text{add}}^{\text{getsize}}(x) \equiv \text{false}, \]

\[ \varphi_{\text{pop}}^{\text{pop}}(y) \equiv \text{pop}(y) \equiv \text{true} \land \text{pop}(y) \equiv \text{true} \]

While this example is somewhat artificial and small, we picked
it to highlight couple of aspects. One, commutativity reasoning can
quickly become onerous to do manually. Two, there can be multiple
concrete ways of representing the same semantic data structure
state: \( a = 5 \land b = 3 \) is the same as \( a = 3 \land b = 5 \). This is typical
of most data structure implementations, such as list representations
of unordered sets, hash tables, binary search trees, etc. Our goal of
commutativity reasoning from source code means that we do not
have the specification provided to us when two concrete states are
actually the same state semantically. Below we will discuss how we
overcome this challenge.

As a second running example, let us consider an array based
implementation of Stack, given at the bottom 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 commutativity condition
\( \varphi_{\text{push}}^{\text{pop}}(x) \equiv \text{top} > -1 \land A[\text{top}] = x \land \text{top} < \text{MAX} \) captures

```java
class SimpleSet {
    private int a, b, sz;
    SimpleSet() { a=b=-1; sz=0; }
    void add(uint x) {
        if (a==1 & & b==1) { a=x; sz++; }
        if (a==1 & & b==1) { b=x; sz++; }
        if (a==1 & & b==1) { return; }
    }
    bool isin (uint y) { return (a==y||b==y); }
    bool getsize () { return 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) return false;
        A[top+1] = x; return true;
    }
    int pop() {
        if (top == -1) return -1;
        else return A[top--];
    }
    bool isempty() { return (top==-1); }
}
```

Figure 1. (a) On the top, a SimpleSet data structure, capable of
storing up to two non-zero identifiers (using private memory \( a \) and \( b \) ) and tracking the size \( sz \) of the Set. (b) On the bottom, an
ArrayStack data structure that implements a simple stack using an
array and top index.
that they commuted 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.

1.2 Enabling abstractions

As these examples show, specialized abstractions are needed for commutativity reasoning. When considering the commutativity of $\text{isin}(y)$ and clear, we can use an abstraction that ignores sz and instead just reasons about a and b, such as the predicates $a = y$ and $b = y$, along with their negations. This abstraction not only ignores sz, but it also ignores other possible values for a and b. The only relevant aspect of the state is whether or not y is in the set. For ArrayStack, when considering push(x) and pop, we can similarly abstract away deeper parts of the stack: for determining return values, our abstraction only needs to consider the top value. In Section 3, we will formalize this concept as an mn-differencing abstraction denoted $(a^m_m, R_a)$ and, in Section 4, describe a strategy that discovers these abstractions automatically. This concept is illustrated on the left of the following diagram:

Intuitively, $R_a$ is a relation on abstract states whose purpose is to "summarize" the possible pairs of post-states that (i) originate from a state $\sigma_o$ satisfying the commutativity condition $\varphi^m_m$ and (ii) will have agreed on intermediate return values along the way. To this end, the abstraction $\alpha^m_m$ must be fine-grained enough to reason about agreement on these intermediate return values $(r_m \equiv r_m, r_n \equiv r_m)$ but can abstract away other details. In the $\text{isin}(y)$ example, the proposed abstraction that simply uses predicates $a = y$ and $b = y$ is an mn-differencing abstraction because it is all that’s needed to show that, in either order, the return values agree.

Observational Equivalence. While this mn-differencing abstraction ensures that return values agree, how can we ensure that either order leads to equivalent post-states? First, for some start state $\sigma$, let us denote by $\sigma_{mn}$ the state arising by applying $mn(\bar{a}); n(\bar{b})$. We similarly define $\sigma_{nm}$. We define observational equivalence in a standard way, by requiring that any sequence of method invocations (or actions) $m_1(\bar{a_1})/r_1; m_1(\bar{a_1})/r_2; \ldots$ when separately applied to $\sigma_{mn}$ and $\sigma_{nm}$, returns sequences of values that agree. This can be seen on the right in the above diagram: relation $I_3$ is maintained.

Observational equivalence lets us balance abstraction (for reasoning about $m, n$-specific interactions) with equality (for reasoning about parts of the state that are unaffected by the order of $m$ and $n$). That is, observational equivalence is here used in the context of two states ($\sigma_{mn}$ and $\sigma_{nm}$) that originate from precisely the same state $\sigma$, and have likely not stayed too far from each other. In this way, the infinite extension often collapses to comparing two states that are exactly equal. For example, consider the relations:

$$I_{AS}(\sigma_{mn}, \sigma_{nm}) \iff \top_{mn} = \top_{nm} \land (\forall i. 0 \leq i \leq \top_{mn} \Rightarrow a_{mn}[i] = a_{nm}[i])$$

$$I_{SS}(\sigma_{mn}, \sigma_{nm}) \iff (a_{mn} = b_{mn} \land b_{nm} = b_{mn}) \lor (a_{mn} = b_{nm} \land b_{nm} = a_{nm}) \land (sz_{mn} = sz_{nm})$$

Note that it is sufficient to start from some concrete state, rather than two observationally equivalent start states in the reasoning, as only one of the order of the methods execution will occur at run-time.

For the ArrayStack, $I_{AS}$ says that the two states agree on the (ordered) values in the Stack. $(\top_{mn}, \top_{nm})$ 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.

If we can find an $I$ and a valid mn-differencing relation $R_a$ such that $R_a \Rightarrow I$, then we have proved commutativity. These abstractions work well together when used in tools (discussed below). When considering, for example, push and pop, only the top element of the stack is changed differently depending on the order, and our abstraction lets us treat the other aspects of the state (that are unchanged or changed in the same way regardless of $m - n$ order) as being exactly the same. This observation let us employ verification analysis in a way that is focused, and avoids the need for showing full functional correctness.

1.3 Challenges

Above we have sketched some of the enabling insights behind our work, but many questions remain. In the rest of this paper, we answer questions such as:

- Is there a formal equivalence between proving commutativity and finding sufficient mn-differencing abstractions and observational equivalence? (Section 3)
- Can we employ these concepts to reduce commutativity verification to reachability, so that mn-differencing and observational equivalence abstractions can be correlated with non-reachability proofs? (Section 4)
- How can we build a tool around these concepts? (Section 5)
- How can we leverage existing reachability solvers to prove commutativity of simple ADTs? (Section 6)
- How usable is our technique? (Section 6.1)

2. Preliminaries

Language. 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 $o.m(\bar{x})$, $o.n(\bar{y})$, ... where $\bar{x}$ is a vector of the arguments. We often will omit the o when it is unneeded; we use the notation $m(\bar{x})/\bar{v}$ to refer to the return variables $\bar{v}$. We use $\bar{a}$ to denote a vector of argument values, $\bar{u}$ to denote a vector of return values and $(\bar{a})/\bar{u}$ to denote a corresponding invocation of a method which we call an action. For a method $m$, our convention will be the signature $n(\bar{y})/s$ and values $n(\bar{b})/e$. Methods source code is parsed from C into control-flow automata (CFA) [22], discussed in Section 4.1, using assume to represent branching and loops. Edges are labeled with straight-line ASTs:

$$e ::= o.a \mid c \mid x \mid e \lor e, \quad b ::= b \lor b \mid \neg b \mid \text{true}, \quad s ::= \text{assume}(b) \mid s_1 ; s_2 \mid x ::= e$$

Above $\lor$ represents typical arithmetic operations on integers and $\lor$ represents typical boolean relations. We use $s_{mn}$ to informally refer to the source code of object method $m$ (which is, formally, represented as an object CFA, discussed in Section 4.1). For simplicity, we assume that one object method cannot call another, and that all object methods terminate. Non-terminating object methods are typically not useful and their termination can be confirmed using existing termination tools (e.g. [9]).

We fix a single object o, denote that object’s concrete state space $\Sigma$, and assume decidable equality. We denote $\sigma \overset{m(\bar{a})}{\rightarrow} \sigma'$ for the big-step semantics in which the arguments are provided, the entire method is reduced. For lack of space, we omit the small-step
semantics [8] 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 [3] and prophecy techniques [1], resp.

**Definition 2.1** (Observational Equivalence (e.g. [25])). We define relation $≃⊆ \Sigma \times \Sigma$ as the following greatest fixpoint:

$$∀m(\bar{a}) \in M. σ_1 \xrightarrow{m(\bar{a})/\bar{v}} σ'_1 \land σ_2 \xrightarrow{m(\bar{a})/\bar{v}} σ'_2 \implies \bar{r} = \bar{s} \land σ'_1 ≃ σ'_2$$

The above co-inductive definition expresses that two states $σ_1$ and $σ_2$ of an object are observationally equivalent $≃$ 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 $≃$ relation. Notice that a counterexample to observational equivalence (i.e., something that is not in the relation) is a finite sequence of method invocations $m_1(\bar{a}_1), ..., m_k(\bar{a}_k)$ applied to both $σ_1$ and $σ_2$ such that for $m_k(\bar{a}_k)$, $r_k \neq s_k$.

We next use observational equivalence to define commutativity. As is typical [3, 18] 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 values.

**Definition 2.2** (Commutativity). For values $\bar{a}, \bar{b}$, return values $\bar{u}, \bar{v}$, and methods $m$ and $n$, “actions $o.m(\bar{a})/\bar{u}$ and $o.n(\bar{b})/\bar{v}$ commute,” denoted $o.m(\bar{a})/\bar{u} \bowtie o.n(\bar{b})/\bar{v}$, if

$$∀σ, \sigma \xrightarrow{m(\bar{a})/\bar{u}} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}} σ_n \implies σ_m ≤ σ_n$$

(Notice that these are values, so action commutativity requires return value agreement.) We further define “methods $o.m$ and $o.n$ commute,” denoted $o.m \bowtie o.n$ provided that

$$∀\bar{a}, \bar{b}, v.o.m(\bar{a})/\bar{u} \bowtie o.n(\bar{b})/\bar{v}$$

The quantification $∀\bar{a}, \bar{b}, \bar{u}, \bar{v}$ above means vectors of all possible argument and return values. Note that commutativity begins with a single state $σ$, but implies a relation on states. (We will come back to this later.) Our work extends to a more fine-grained notion of commutativity: an asymmetric version called left-movers and right-movers [29], 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/return values of the methods. We denote a logical commutativity formula as $ϕ_m^n$ and assume a decidable interpretation of formulae: $[ϕ_m^n] : (σ, x, y, r, s) → \mathbb{B}$. (We tuple the arguments for brevity.) The first argument is the initial state. Commutativity post- and mid-conditions can also be written [24] but, here for simplicity, we focus on commutativity pre-conditions. We may write $[ϕ_m^n]$ as $ϕ_m^n$ when it is clear from context that $ϕ_m^n$ is meant to be interpreted.

**Definition 2.3** (Commutativity Condition). We say that logical formula $ϕ_m^n$ is a commutativity condition for $m$ and $n$ provided that

$$∀σ \bar{a} \bar{b} \bar{u} \bar{v} : [ϕ_m^n] σ \bar{a} \bar{b} \bar{u} \bar{v} ⇒ m(\bar{a})/\bar{u} \bowtie n(\bar{b})/\bar{v}$$

3. Abstraction

We now formalize abstractions for two interconnected aspects of commutativity reasoning: $mn$-differencing and observational equivalence relations. These treatments provide a route to automation and re-use of abstraction synthesis techniques, discussed in Section 4.

For convenience, we define post-states posts and return value agreement $\text{rvagree}$ as follows:

$$\text{posts}(σ, m, \bar{a}, n, \bar{b}) \equiv (σ_m, σ_n)$$

such that

$$σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n \land σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n \land σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n \land σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n \land σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n \land σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n \land σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n \land σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n \land σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n \land σ \xrightarrow{m(\bar{a})/\bar{v}_m} σ_m \land σ \xrightarrow{n(\bar{b})/\bar{v}_n} σ_n$$

We now formalize $mn$-differencing abstraction:

**Definition 3.1** ($mn$-differencing Abstraction $(α, R_α)$). Let $o$ be an object with state space $\Sigma$, and consider two methods $m$ and $n$. Let $α : Σ ↠ Σ^α$ be an abstraction of the states, and $γ : Σ^α ↠ Π(Σ)$ the corresponding concretization. Let $R_α : Σ^α ↠ Σ^α$ be a relation on abstract states. We say that $(α, R_α)$ is an $mn$-differencing abstraction if

$$∀σ_1', σ_2' \in [R_α][σ_1, σ_2] ⇒ ∀σ_1 σ_2, \text{posts}(σ, m, \bar{a}, n, \bar{b}) \in [γ(σ_1')] \times [γ(σ_2')] ⇒ \text{rvagree}(σ, m, \bar{a}, n, \bar{b})$$

Here $α$ provides an abstraction and $R_α$ is a relation in that abstract domain. Intuitively, $R_α$ relates pairs of post-states that (i) originate from same state $σ$ and (ii) agree on intermediate return values. $α$ must be a precise enough abstraction so that $R_α$ can discriminate between pairs of post-states where return values will have agreed versus disagreed. For the $isin(x)$ clear example, we can call the abstraction that tracks whether $a = x$ and whether $b = x$. Then $R_α(σ_1, σ_2) \equiv (a = x), (b = x) \implies (b = x), i.e. the relation that tracks if $σ_1$ and $σ_2$ agree on whether $x$ is in the set.

**Definition 3.2**. Let $(α, R_α)$ be an $mn$-differencing abstraction and $ϕ_m^n$ a logical formula on concrete states and actions of $m$ and $n$. We say that $ϕ_m^n$ implies $(α, R_α)$ if

$$∀σ \bar{a} \bar{b} \bar{r} \bar{s} : [ϕ_m^n] σ \bar{a} \bar{b} \bar{r} \bar{s} ⇒ R_α(α(σ_m), α(σ_n))$$

where $(σ_m, σ_n) = \text{posts}(σ, m, \bar{a}, n, \bar{b})$.

The above definition treats commutativity condition $ϕ_m^n$ as a precondition, specifically one that characterizes certain start states $σ_0$ for which the $mn$-differencing abstraction will hold. If we let $ϕ_{isin}(x) \equiv a \neq x$ or $b \neq x$, this will imply $R_α$ in the posts: the post states will agree on whether $x$ is in the set, thus capturing our intuitive understanding of commutativity.

Next, we introduce another (possibly different) abstraction which helps to reason about observational equivalence of the poststates reached.

**Definition 3.3**. Let $β : Σ ↠ Σ^β$ be an abstraction function, with corresponding concretization function $δ : Σ^β ↠ Π(Σ)$, and let $I_β$ be a relation on these abstract states, $[I_β] : Σ^β × Σ^β ↠ B$. Then $I_β$ is an observational equivalence relation iff:

$$∀σ_1', σ_2' \in [β][σ_1, σ_2] ⇒ ∀σ_1 ∈ δ(σ_1'), σ_2 ∈ δ(σ_2'), σ_1 ≃ σ_2$$

$I_{SS}$, defined earlier, is such a relation.

The next definition is a natural notion stating when an abstraction is coarser than the other; we use it later.

**Definition 3.4**. For all $(α, R_α)$ and $(β, I_β)$ we say that “$(α, R_α)$ implies $(β, I_β)$” iff:

$$∀σ_1, σ_2 ∈ [R_α][σ_1, σ_2] ⇒ [I_β](β(σ_1), β(σ_2))$$
The following theorem provides sufficient conditions for a commutativity condition for the two methods, with respect to these abstractions.

**Theorem 3.1.** Let \( o \) be an object with state space \( \Sigma_o \) and methods \( m(\vec{x}) \) and \( n(\vec{y}) \). Let \( \varphi_{m}^o \) be a logical formula on \( \Sigma_o \) and actions of \( m \) and \( n \). Let \( (\alpha_m, R_{m}^o) \) be an \( m/n \)-differencing abstraction and \((\beta, I_{\beta})\) such that \( I_{\beta} \) is an abstract observational equivalence relation. If \( \varphi_{m}^o \) implies \((\alpha_m, R_{m}^o)\) and \( (\alpha_n, R_{n}^o) \) implies \((\beta, I_{\beta})\) then \( \varphi_{m}^o \) is a commutativity condition for \( m \) and \( n \).

**Proof sketch.** Fix \( \sigma \in \Sigma_o \) and \( m(\vec{a})/\vec{r} \) and \( n(\vec{b})/\vec{s} \) actions for \( m \) and \( n \) respectively. We need to show that \( m(\vec{a})/\vec{r} \equiv n(\vec{b})/\vec{s} \). In particular, we need to show that the return values agree, and the post states reached with methods commuted are observationally equivalent. From Definition 3.2, we have that \( R \) holds for the abstraction of post states, and then from Definition 3.1 it follows that the return values agree. On the other hand, from Definition 3.4 it follows that \( I_{\beta} \) holds for the \( \beta \)-abstraction of post states as well, and from Definition 3.3 it follows that the (concrete) post states are observationally equivalent.

The idea is that for a \( m/n \) pair, find an \((\alpha, R_o)\) that witnesses the way in which the order causes divergence but strong enough to imply some abstract equivalence relation \((\beta, I_{\beta})\). Given that \( \varphi_{\alpha}^o \) implies the above \( R_o \) and that it is easy to see that \( R_o \) implies \( I_{SS} \), we can conclude that \( \varphi_{\alpha}^o \) is a valid commutativity condition.

### 4. Reduction to Reachability

We now describe how we reduce the task of verifying commutativity condition \( \varphi_{m}^o \) to a reachability problem. To this end, we need a representation of object implementations, as well as the output encoding. As noted, we build on the well-established notion of control-flow automata (CFA) [22], extending them slightly to represent objects. We then describe our transformation from an input object CFA to an output encoding CFA \( A(\varphi_{m}^o) \) with an error state \( q_{err} \). Finally, we prove that, if \( q_{err} \) is unreachable in \( A(\varphi_{m}^o) \), then \( \varphi_{m}^o \) is a valid commutativity condition.

#### 4.1 Object Implementations

**Definition 4.1 (Control-flow automaton [22]).** A (deterministic) control-flow automaton is a tuple \( A = (Q, q_0, X, s, \rightarrow) \) where \( Q \) is a finite set of control locations and \( q_0 \) is the initial control location, \( X \) is a finite set of typed variables, \( s \) is the loop/branch-free statement language (as defined earlier) and \( s \rightarrow x \subseteq Q \times s \times Q \) is a finite set of labeled edges.

**Valuations and semantics.** We define a valuation of variables \( \theta : X \rightarrow \text{Val} \) to be a mapping from variable names to values. Let \( \Theta \) be the set of all valuations. The notation \( \theta \in [x]_{\Theta} \) means that executing statement \( s \), using the values given in \( \theta \), leads to a new valuation \( \theta' \), mapping variables \( X \) to new values. Notations \( [\theta]_{\Theta} \) and \( [\theta]_{\Theta} \) represent side-effect free numeric and boolean valuations, respectively. We assume that for every \( \theta, \theta' \), that \( \theta[s] \) is computed in finite time. The automaton edges, along with \( [\theta] \), give rise to possible transitions, denoted \( (q, \theta) \rightarrow (q', \theta') \) but omit these rules for lack of space.

A run of a CFA is an alternation of automaton states and valuations denoted \( r = q_0, \theta_0, q_1, \theta_1, q_2, \ldots \) such that \( \forall i \geq 0, (q_i, \theta_i) \rightarrow (q_{i+1}, \theta_{i+1}) \). We say \( A \) can reach automaton state \( q \) (safety) provided there exists a run \( r = q_0, \theta_0, q_1, \theta_1, \ldots \) such that there is some \( i \geq 0 \) such that \( q_i = q \). We next conservatively extend CFAs to represent data structure implementations:

**Definition 4.2 (Object CFAs).** An object control flow automaton for object \( o \) with methods \( M = \{m_1, \ldots, m_k\} \), is:

\[
A_o = (Q_o, \{q_0^{init}\}, X_{o, s}, \rightarrow) \\
X_o = X^{st} \cup \{ \text{th} \} \cup X^{init} \cup \bigcup_{i \in [k]} (X^{m_i} \cup X^{m_i^o} \cup X^{m_i^o})
\]

where \( Q_o \) is a finite set of control locations, \( q_0^{init} \) is the initial control location of the initialization routine, and \( q_0^{init} \) is the initial control location for method \( m_i \). Component \( X_o \) is a union of sets of typed variables; \( X^{st} \) representing the object statefields, this variable, and \( X^{init} \) representing the initialization routine’s local variables. For each \( m_i, X^{m_i} \) represents method \( m_i \)'s local variables, \( X^{m_i^o} \) represents parameters, and \( X^{m_i^o} \) represents return variables. Finally, \( s \) is the statement language and \( s \rightarrow x \subseteq Q_o \times s \times Q_o \) is a finite set of labeled edges.

For simplicity, the above definition avoids inter-procedural reasoning. However, we need to be able to invoke methods. Above, we will call each \( q_0^{init} \) node the entry node for the implementation of method \( m_i \) and we additionally require that, for every method, there is a special exit node \( q_0^{exit} \). We require that the edges that lead to \( q_0^{exit} \) contain return statements. Arcs are required to be deterministic. This is not without loss of generality. We support nondeterminism in our examples by symbolically determining the input CFA: whenever there is a nondeterministic operation \( m(\vec{x}) \), we can augment \( \vec{x} \) with a fresh prophecy variable \( p \), and replace the appropriate statements with a version that consults \( p \) to resolve nondeterminism (see [14]). The semantics \( (q, \theta) \rightarrow (q', \theta') \) of \( A_o \) induce a labeled transition system, with state space \( \Sigma_{A_o} = Q_o \times \Theta \). Naturally, commutativity of an object CFA is defined in terms of this induced transition system.

#### 4.2 Transformation

We now define the transformation. First, syntactic sugar:

\[
q \rightarrow \text{init}(m_1, o, \vec{x}, \vec{r}) \quad \rightarrow \quad q \equiv \{ q \rightarrow \text{init}(m_0, q_0, \vec{x}, \vec{r}) \}
\]

This definition allow us to emulate function calls to a method \( m_i \), starting from CFA node \( q \). Values \( \vec{x} \) are provided as arguments, and arcs are created to the entry node \( q_0^{init} \) for method \( m_i \). Furthermore, return values are saved into \( \vec{r} \) and an arc is created from the exit node \( q_0^{exit} \) to \( q' \). Also, we will let assume\((\vec{x} \neq \vec{y})\) mean the disjunction of inequality between corresponding vector elements, i.e. if there are \( N \) elements, then the notation means \( x_0 \neq y_0 \lor \cdots \lor x_N \neq y_N \).

**Definition 4.3 (Transformation).** For an input object CFA \( A_o = (Q_o, \{q_0^{init}, q_0^{init}, \ldots, q_0^{init}\}, X_o, s, \rightarrow) \), the result of the transformation when applied to methods \( m(\vec{x}), n(\vec{y}) \), is output CFA \( A(\varphi_{m}^o) = (Q_e, q_0^e, X_e, s_e, \rightarrow) \), where \( \rightarrow \equiv \) is the union of

\[
\begin{cases}
q_0^e \rightarrow \text{init}(\text{init}, \text{nil}, [], [o]) \rightarrow q_1 \\
\cup_{\text{init} \rightarrow \text{init}} \{ q \rightarrow \text{init}(m_1, o, \vec{x}, \vec{r}) \rightarrow q_{11} \\
\text{Reachbl. o}_1 \rightarrow \{ q \rightarrow \text{clone}(\text{clone}, o_1, [], [o_2]) \rightarrow q_2 \}
\end{cases}
\]

where \( o \) is the source

\( q_0^e \rightarrow \text{init}(\text{init}, \text{nil}, [], [o]) \rightarrow q_1 \) (Assume \( \varphi_{m}^o \))
and $Q_E$ is the union of $Q_s$ and all CFA nodes above, $s_E$ is the union of $s$ and all additional statements above, and $X_{E} = X_{o} \cup \{o_1, o_2, \bar{a}, \bar{b}, \bar{r}_m, \bar{r}_n, \bar{f}_m, \bar{f}_n, nnil, \bar{f}, \bar{s}\}$.

We now describe the above transformation intuitively. Node $q^E_0$ is the initial node of the automaton. The transformation employs the implementation source code of the data structure CFA, given by $->$. The key theorem below says that non-reachability of $q_{cr}$ entails that $\phi^m_{cr}$ is a commutativity condition. We now discuss how the components of $A(\phi^m_{cr})$ exploit this strategy:

1. The first arcs of $Q_E$ are designed so that, for any reachable state $\sigma$ of object $o$, there will be some run of $A(\phi^m_{cr})$ that witnesses that state. This is accomplished by arcs from $q_1$ into the entry node of each possible $m_1$, first letting $Q_E$ nondeterministically set arguments $\bar{x}$.

2. From $q_1$, nondeterministic choices are made for the method arguments $m(\bar{a})$ and $n(\bar{b})$, and then candidate condition $\phi^m_{cr}$ is assumed. And arc then causes a run to clone $o_1$ to $o_2$, and then invoke $m(\bar{a})$; $n(\bar{b})$ on $o_1$ and $n(\bar{b})$; $m(\bar{a})$ on $o_2$.

3. From $q_3$, there is an arc to $q_{cr}$ which is feasible if the return values disagree. There is also an arc to $q_4$.

4. $q_4$ is the start of a loop. From $q_4$, for any possible method $m_1$, there is an arc with a statement $\bar{a} \leftarrow \bar{x}$ to choose nondeterministic values, and then invoke $m(\bar{a})$ on both $o_1$ and $o_2$. If it is possible for the resulting return values to disagree, then a run could proceed to $q_{cr}$.

**Proving that $q_{cr}$ is unreachable.** From the structure of $Q_E$, there are two ways in which $q_{cr}$ could be reached and a proof of non-reachability involves establishing invariants at two locations:

- **mn-differencing relation at $q_3$:** At this location a program analysis tool must infer a condition $R_n$ that is strong enough to avoid $q_{cr}$. However, it may leverage the fact that $o_1$ and $o_2$ are similar.

- **Observational equivalence at $q_4$:** At this location, $A(\phi^m_{cr})$ considers any sequence of method calls $m', m'', \ldots$ that could be applied to both $o_1$ and $o_2$. If observational equivalence does not hold, then there will be a run of $A(\phi^m_{cr})$ that applies that some sequence of method calls to $o_1$ and $o_2$, eventually finding a discrepancy in return values and making a transition to $q_{cr}$. To show this is impossible, a program analysis tool can use an observational equivalence relation $I(n_1, n_2)$.

- **Implication:** Finally, if it can be shown that $R_n$ implies $I$, then $\phi^m_{cr}$ is a valid commutativity condition.

The utility of our encoding can be seen when considering what happens when a reachability analysis tool is applied to $A(\phi^m_{cr})$. Specifically, we can leverage decades of ingenuity that went into the development of effective automatic abstraction techniques (such as trace abstraction [21], interpolation [30], CEGAR [11], k-Induction, block encoding [5]) for reachability verification tools. These tools are unaware of the concept of “commutativity,” nor do they concern with properties such as memory safety, full-functionality, correctness, etc. These are targeted tools, tuned toward proving non-reachability of $q_{cr}$. In this way, when these abstraction strategies are applied to $A(\phi^m_{cr})$, they focus on only what is needed to show non-reachability, and avoid the need to reason about large aspects of the program that are irrelevant to reachability (and thus irrelevant to commutativity).

**Theorem 4.1 (Soundness).** For object implementation $A_o$ and resulting encoding $A(\phi^m_{cr})$, if every run of $A(\phi^m_{cr})$ avoids $q_{cr}$, then $\phi^m_{cr}$ is a commutativity condition for $m(\bar{x})$ and $n(\bar{y})$.

**Proof Sketch.** We prove the theorem by considering a program analysis’s proof that $q_{cr}$ cannot be reached which takes the form of abstractions and invariants. Since $q_{cr}$ is not reachable, after reaching $q_3$ it must be the case that the return values agree. This forces $R_n$ to be such that it is an $m, n$-differencing abstraction. Since $q_{cr}$ is not reachable, after reaching $q_4$, the automaton loops in $q_4$ where $I$ holds followed by non-deterministic call to $m'$ with nondeterministic action of $m'$. Since the observational equivalence relation is the greatest fixpoint, we can conclude that $I$ satisfies the condition for being an abstract observational equivalence relation. Thus, these invariants $R_n$ and $I$ serve as the $m$, $n$-differencing abstraction and abstract observational equivalence relation, respectively. Moreover, the encoding also ensures that $\phi^m_{cr}$ implies $(\alpha, R_n)$ and that $(\alpha, I)$ implies $(\alpha, I)$. Finally, we employ Theorem 3.1.

The above captures the property of the transformation, which is carefully done so that the abstractions satisfy the hypotheses in Theorem 3.1.

**4.3 Example: SimpleSet**

Fig. 2 illustrates the output of applying our encoding to the methods $\text{add}(x)$ and $\text{isin}(y)$ of the SimpleSet (from Fig. 1) denoted...
\(A(\phi_{\text{add}}(v))\). It is important to remember that this resulting encoding
\(A(\phi_{\text{add}}(v))\) should never be executed. Rather, it is designed
so that, when a program analysis tool for reachability is applied, the
tool is tricked into performing the reasoning necessary proving
commutativity. We now discuss the key aspects of the encoding,
reffering to the corresponding portion of the pseudocode in Fig. 2.

(A) Any initial state. The encoding is designed so that, for any
reachable abstract state \(\sigma\) of the SimpleSet object, there will be
a run of \(A(\phi_{\text{add}}(v))\) such that the SimpleSet on Line 7 will be in
state \(\sigma\).

(B) Assume \(\phi_1(n)\) and consider \(m,n\) as well as \(n,m\). A program
analysis will consider all runs that eventually exit the first loop
(we don’t care about those that never exit), and the corresponding
reachable state \(s_1\). From \(s_1\), the encoding assumes that provided
commutativity condition on Line 6 and then a clone of \(s_1\) will be
materialized. Our encoding then will cause a program analysis
to consider the effects of the methods applied in each order, and
whether or not the return values will match on Line 10.

(C) Observational equivalence. Lines 11-16 consider any sequence
of method \(m'(a'), m''(a'')\ldots\) that could be applied to both
\(s_1\) and \(s_2\). If observational equivalence does not hold, then there
will be a run of \(A(\phi_{\text{add}}(v))\) that applies to sequence \(s_1\) and \(s_2\),
eventually finding a discrepancy in return values and going to \(q_{er}\).
The encoding contains two boxed assertions: \(R_{\text{add}}^j(s)\) on Line 11
must be strong enough to guard against the possibility of a run
reaching \(q_{er}\) due to return values. Similarly, \(I_{SS}\) on Line 11 must
be strong enough to guard against the possibility of a run reaching
\(q_{er}\) owing to an observable difference between \(s_1\) and \(s_2\). Thus,
to prove that \(ERROR\) is unreachable, a verification tool needs to have
or discover strong enough such invariants.

5. Implementation

We have developed CityProver, capable of automatically verifying
commutativity conditions of object methods directly from their
implementations. CityProver takes, as input, C source code for
the object (represented as a struct state\(\cdot\) and series of methods
each denoted \(\text{m\text{-}}(\text{struct state}\cdot\) this,\ldots\)). This includes the
description of the object (the header file), but no specification.
Examples of this input can be found in Appendix A. We have written
them as C macros so that our experiments focus on commutativity
rather than testing existing tools’ inter-procedural reasoning power,
which is a separate matter. Also provided as input to CityProver
is a commutativity condition \(\phi_m\) and the method names \(m\) and \(n\).
CityProver then implements the encoding \(A(\phi_m)\), via a program
transformation.

Abstracting twice. The reduction given in Section 4.2 describes
an encoding for which a single abstraction can be used. However,
we found that existing tools struggled to find a single abstraction
for reasoning about the multiple aspects of a commutativity proof.
To resolve this issue, we developed an alternative transformation,
employed in our implementation, which takes advantage of the two
abstractions described in Section 3: \(\alpha\) for \(mn\)-differencing reasoning
and \(\beta\) for observational equivalence.

The implementation of our transformation actually generates
different encoding “pieces” whose total safety is equivalent
to the safety of the single-step reductive in Section 4.2. In this way,
different abstractions can be found for each piece. An illustration
is given in Fig. 3. Piece 1 captures the first part of the general
encoding, except the generated CFA ends just after the instances
of \(m; n; n; m\), with a single assume arc to \(q_{er}\), with the condition
that \(r_m \neq r_m \lor r_n \neq r_m\). This piece ensures that \(\phi_m\) at least
entails that return values agree. When a program analysis is applied
in this piece, it discovers a \((\alpha, R_\alpha)\). Piece 2 repeats some of piece
and the overall time in seconds. These experiments confirm that both CPAchecker and Ultimate can prove commutativity of these simple benchmarks, within a few seconds. In one case, CPAchecker returned an incorrect result.

**Larger benchmarks.** We next turned to data structures that store and manipulate elements, and considered the ADTs listed below with various candidate commutativity conditions. For these ADTs, we had difficulty tuning CPAchecker (perhaps owing to our limited experience with CPAchecker) so we only report our experience using Ultimate as a reachability solver. The results of these benchmarks are given in Fig. 5. We discuss each ADT in turn.

- **ArrayStack.** (Fig. 1) Note that push/pop commutativity condition \( \varphi_{\text{push}}^{\text{pop}} \) is defined below the table in Fig. 5. For this ADT, some method pairs and commutativity conditions took a little longer than others.
- **SimpleSet.** (Fig. 1) Most cases where straightforward; not surprisingly, \( x/\text{add}(y) \) took more time.
- A simple HashTable, where hashing was done only once and insertion gives up if there is a collision. Here again, note the definitions of some commutativity conditions below the table.
- **Queue.** implemented with an array. CITYPROVER was able to prove all but two commutativity conditions for the Queue. The enq/deq case ran out of memory.
- **List Set.** Some commutativity conditions could be quickly proved or refuted. For more challenging examples, e.g. add/rm, Ultimate ran out of memory. In these cases reachability involves reasoning between lists with potentially different element orders. We believe that the necessary disjunctive reasoning overwhelmed Ultimate.

In summary, CITYPROVER allowed us to prove a variety of commutativity conditions using unmodified Ultimate. The power of CITYPROVER greatly depends on the power of said underlying verification tool.

For this prototype, we provided observational equivalence relations manually. As tools improve their ability to synthesize loop invariants, this input may become unnecessary.

### 6.1 Usability

We now discuss the usability of CITYPROVER, including lessons learnt from our experience.

Does the approach avoid the need for full specifications? Yes. \( mn \)-differencing focuses on the aspects of data structure implementations that pertain to commutativity and need not capture full functional correctness. Since \( mn \)-differencing is based on method orderings from the same start state, many details of a specification become trivial. Notice that the commutativity conditions in Fig. 5 are far simpler than what the full specifications of those data structures would be. The ability to proceed without specifications is also convenient because, for many data structures that occur in the wild, specifications are not readily available.

Are commutativity formula easier to come up with than specifications? Intuitively, yes, because commutativity conditions can often be very compact (e.g. \( x \neq y \)). By contrast, specifications relate every post-state with its corresponding pre-state and typically involve multiple lines of logical formulae, not easily written by users. Since commutativity conditions are smaller, users can make a few guesses until they find one that holds.

We followed this methodology in some of the benchmarks: (i) SimpleSet. For \( \text{isin} (x) \rightarrow \text{clear} \), we originally thought that \( x \neq y \) would be a valid commutativity condition, expressing that neither a nor b had the value. The tool caught us, and reported a counterexample, showing the type \( y \) should be \( x_1 \). (ii) Queue. For \( \text{enq}/\text{enq} \), we guessed “true” as a first condition. CITYPROVER then produced a counterexample, showing they do not commute if the arguments are different. We strengthened the condition, but that still was not enough: the next counterexample showed they do not commute if there’s not enough space. (iii) HashTable. The successful conditions \( \varphi_{\text{put}}^{\text{put}} \) represent our repeated attempts to guess commutativity conditions for \( \text{put}(x_1) \rightarrow \text{put}(y_1) \). We first tried a simple condition \( \varphi_{\text{put}}^{\text{put}} \) expressing that the keys are distinct. CITYPROVER returned a counterexample, showing that commu-

---

| ADT          | Meths. \( m(x_1), n(y_1) \) \( m(y_1) \) | Exp. | CPAchecker (by Piece) | \( \Sigma \) Out | Ultimate (by Piece) | \( \Sigma \) Out |
|--------------|------------------------------------------|------|-----------------------|----------------|---------------------|----------------|
| Memory       | read  deq  write \( a_1, x = y_1 \)     | ✓    | 1.238 + 1.162 + 1.513 | (✓,✓,✓)       | 3.553               | ✓              |
| Memory       | read  deq  write \( y_1 = x_1 \)         | ✓    | 1.190 + 1.160 + 1.181 | (✓,✓,✓)       | 3.531               | ✓              |
| Memory       | write  deq  write \( y_1 = x_1 \)        | ✓    | 1.250 + 1.178 + 1.315 | (✓,✓,✓)       | 3.86               | ✓              |
| Memory       | read  deq  read \( y_1 = x_1 \)          | ✓    | 1.202 + 1.176 + 1.202 | (✓,✓,✓)       | 3.58               | ✓              |
| Accum. deq  isz \( a_1, x > 1 \)        | ✓    | 1.227 + n/a + n/a  | (y,n/a,n/a)   | 1.227           | 1.38               | ✓              |
| Accum. deq  incr \( a_1, x > 1 \)        | ✓    | 1.257 + 1.437 + 1.346 | (✓,✓,✓)       | 3.798           | 1.5               |
| Accum. incr  isz \( a_1, x > 1 \)        | ✓    | 1.298 + 1.220 + 1.238 | (✓,✓,✓)       | 3.756           | 3.1               |
| Accum. incr  incr \( a_1, x > 2 \)        | ✓    | 1.284 + 1.277 + 1.251 | (✓,✓,✓)       | 3.812           | 1.9               |
| Counter deq  incr \( a_1, x > 1 \)       | ✓    | 1.15 + n/a + n/a  | (x,n/a,n/a)   | 1.214           | 0.6               |
| Counter incr  isz \( a_1, x > 2 \)        | ✓    | 1.255 + 1.422 + 1.460 | (✓,✓,✓)       | 4.179           | 1.5               |
| Counter incr  incr \( a_1, x > 1 \)       | ✓    | 1.257 + 1.437 + 1.499 | (✓,✓,✓)       | 4.179           | 1.5               |
| Counter incr  incr \( a_1, x > 2 \)       | ✓    | 1.257 + 1.437 + 1.499 | (✓,✓,✓)       | 4.179           | 1.5               |

**Figure 4.** Results of applying CityProver to the small benchmarks. We report results when using CPAchecker as well as when using Ultimate.
7. Related work

Bansal et al. [3] describe how to synthesize commutativity conditions from provided pre/post specifications, rather than from implementations. They assume that these specifications are precise enough to faithfully represent all effects that are relevant to the ADT’s commutativity. By contrast, our work does not require the data structure’s specification to be provided which is convenient because specifications are typically not readily available for ad hoc data structures.

Moreover, it is not straightforward to infer the specification (e.g., via static analysis) because the appropriate precision is not known apriori. On one hand, a sound but imprecise specification can lead to incorrect commutativity conditions: for a Stack, specification \{true\} push(v) \{true\} is sound but does not capture effects of push that are relevant to commutativity with, say, pop. With such an imprecise specification, Bansal et al. would emit incorrect commutativity conditions. On the other hand, a sound but overly precise specification can be burdensome and unnecessary for commutativity reasoning: specification \{A[n]\} push(v) \{A'[n+1] = v ∧ ∀i ≤ n, A'[i] = A[i], n + 1\} captures too much information about the stack (deep values in the stack) that is irrelevant to commutativity. By contrast, in this paper we capture just what is needed for commutativity, e.g., only the top and second-to-top elements of the stack.

![Figure 5](image-url) Results of applying CityProver to the more involved data structures, using Ultimate as a reachability solver. Longer commutativity conditions are defined below the table. Note that \(a, x\) is the object field, and \(x, y, 1, n, \) \(\) are \(m, n\) parameters, respectively.
Gehr et al. [19] describe a method 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 is no soundness or completeness guarantee. Both Aleen and Clark [2] and Tripp et al. [35] identify sequences of actions that commute (via random interpretation and dynamic analysis, respectively). However, neither technique yields an explicit commutativity condition. Kulkarni et al. [27] point out that varying degrees of commutativity specification precision are useful. Kim and Rinard [24] use Jahob to verify manually specified commutativity conditions of several different linked data structures. Commutativity specifications are also found in dynamic analysis techniques [18]. Najazfazadeh et al. [31] describe a tool for weak consistency, which reports some commutativity checking of formulae. It does not appear to apply to data structure implementations.

Reductions to reachability are common in verification. They have been used in security—so called self-composition [4, 34]—reduces (some forms of) hyper-properties [12] to properties of a single program. Others have reduced crash recoverability [26], temporal verification [15], and linearizability [8] to reachability.

8. Conclusion

We have described a theory, algorithm and tool that, together, allow one to automatically verify commutativity conditions of data structure implementations. We describe an abstraction called mnr-differencing, that focuses the verification question on the differences in the effects of mns and m that depend on the order in which they are applied, abstracting away effects that would have been the same regardless of the order. We further showed that verification of commutativity conditions can be reduced to reachability. Finally, we described CityProver, the first tool capable of verifying commutativity conditions of data structure implementations.

The results of our work can be used to incorporate more commutativity conditions soundly and obtain speed ups in transactional object systems [17, 23]. 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 [8] to execute more code concurrently and safely.

References

[1] Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82 (May 1991), 253–284. Issue 2. http://dx.doi.org/10.1016/0304-3975(91)90224-P

[2] Farhana Aleen and Nathan Clark. 2009. 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)*. ACM, 241–252.

[3] Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2017. Automatic Generation of Precise and Useful Commutativity Conditions. In *Tools and Algorithms for the Construction and Analysis of Systems - 44th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Greece, April 18-20, Proceedings, Part II*.

[4] Gilles Barthe, Pedro R D’Argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In CSFW.

[5] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M Erkan Keremoglu, Roberto Sebastiani, et al. 2009. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*. IEEE, 25–32.

[6] Dirk Beyer and M. Erkan Keremoglu. 2011. CPachecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 184–190.

[7] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV’11)*. Springer-Verlag, Berlin, Heidelberg, 184-190.

[8] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jada Hamza. 2015. On Reducing Linearizability to State Reachability. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, 95–107.

[9] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 387–393.

[10] Marsha Chechik, Ioanna Stavropoulou, Cynthia Disenfeld, and Julia Rubin. 2018. 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*, 319–336.

[11] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Computer aided verification*. Springer, 154–169.

[12] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[13] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. 2015. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Trans. Comput. Syst.* 32, 4 (2015), 10. http://doi.org/10.1145/2690618

[14] Byron Cook and Eric Koskinen. 2011. Making prophecies with deci- sion predicates. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, Austin, TX, USA, January 26-28, 2011, 399–410. http://doi.org/10.1145/1926385.1926431

[15] Byron Cook, Eric Koskinen, and Moshe Y. Vardi. 2011. Temporal Property Verification as a Program Analysis Task. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011, Proceedings*, 333–348.

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008, Proceedings*, 337–340.

[17] Thomas D. Dickerson, Eric Koskinen, Paul Gazziello, and Maurice Herlihy. 2019. Conflict Abstractions and Shadow Speculation for Optimistic Transactional Objects. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, 313–331.

[18] Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. 2014. Commutativity Race Detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’14)*.

[19] Timon Gehr, Dimitar Dimitrov, and Martin T. Vechev. 2015. Learning Commutativity Specifications. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, 307–323. http://doi.org/10.1007/978-3-319-21090-4_18

[20] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Scilling, and Andreas Podelski. 2013. Ultimate automizer with SMT-Interpol. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 641–643.

[21] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *Com-
[22] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. 2002. Temporal-Safety Proofs for Systems Code. In Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002. Proceedings. 526–538.

[23] Maurice Herlihy and Eric Koskinen. 2008. 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).

[24] Deokhwan Kim and Martin C. Rinard. 2011. 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, 528–541.

[25] Eric Koskinen and Matthew J. Parkinson. 2013. The Push/Pull model of transactions. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 16-17, 2013. 186–195.

[26] Eric Koskinen and Junfeng Yang. 2015. Reducing crash recoverability to reachability. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. 97–108.

[27] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. 2011. Exploiting the commutativity lattice. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011. ACM, 542–555.

[28] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramnarayan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI’07), Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 211–222.

[29] Richard J. Lipton. 1975. Reduction: a method of proving properties of parallel programs. Commun. ACM 18, 12 (1975), 717–721. https://doi.org/10.1145/361227.361234

[30] Kenneth L McMillan. 2003. Interpolation and SAT-based model checking. In International Conference on Computer Aided Verification. Springer, 1–13.

[31] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. 2016. 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, April 18, 2016. 2:1–2:3.

[32] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open nesting in software transactional memory. In Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2007. ACM, 68–78. https://doi.org/10.1145/1229428.1229442

[33] Martin C. Rinard and Pedro C. Diniz. 1997. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. ACM Transactions on Programming Languages and Systems (TOPLAS) 19, 6 (November 1997), 942–991.

[34] Tachio Terauchi and Alex Aiken. 2005. Secure information flow as a safety problem. In SAT.

[35] Omer Tripp, Roman Manevich, John Field, and Mooly Sagiv. 2012. JANUS: exploiting parallelism via hindsight. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12. 145–156. https://doi.org/10.1145/2254064.2254083

[36] Viktor Vafeiadis. 2010. Automatically proving linearizability. In International Conference on Computer Aided Verification. Springer, 450–464.
A. Benchmark Sources

A.1 Memory

```c
struct state_t { int x; };

#include <stdlib.h>

#define o_new(res) res = (struct state_t *)malloc(sizeof(struct state_t))
#define o_clone(src) { 
    struct state_t *tmp;
    o_new(tmp);
    tmp->x = (src)->x;
    dst = tmp;
 }
#define o_read(st, rv) rv = (st)->x
#define o_write(st, rv, v) { (st)->x = v; rv = 0; }
```

A.2 Counter

```c
struct state_t { int x; };

#include <stdlib.h>

#define o_new(res) res = (struct state_t *)malloc(sizeof(struct state_t))
#define o_close(src) { 
    struct state_t *tmp;
    o_new(tmp);
    tmp->x = src->x;
    dst = tmp;
 }
#define o_incr(st, rv) { (st)->x += 1; rv = 0; }
#define o_clear(st, rv) { (st)->x = 0; rv = 0; }
#define o_decr(st, rv) { if ((st)->x == 0) rv = -1; else { (st)->x = 1; rv = 0; } }
#define o_isz(st, rv) { rv = ((st)->x == 0 ? 1 : 0); }
```

A.3 Accumulator

```c
struct state_t { int x; };

#include <stdlib.h>

#define o_new(res) res = (struct state_t *)malloc(sizeof(struct state_t))
#define o_close(src) { 
    struct state_t *tmp;
    o_new(tmp);
    tmp->x = src->x;
    dst = tmp;
 }
#define o_incr(st, rv) { (st)->x += 1; rv = 0; }
#define o_decr(st, rv) { (st)->x -= 1; rv = 0; }
#define o_isz(st, rv) { rv = ((st)->x == 0 ? 1 : 0); }
```

A.4 Array Queue

```c
#define MAXQUEUE 5

struct state_t { int a[MAXQUEUE]; int front; int rear; int size; };

#include <stdlib.h>

#define o_new(res) { 
    res = malloc(sizeof(struct state_t)); 
    res->front = 0; 
    res->rear = MAXQUEUE-1; 
    
```
```c
res->size = 0;
}

#define o_enq(st,rv,v) {
    if((st)->size == MAXQUEUE) rv = 0;
    else {
        (st)->size++; (st)->rear = ((st)->rear + 1) % MAXQUEUE; (st)->a[(st)->rear] = v; rv = 1;
    }
}

#define o_deq(st,rv) {
    if((st)->size==0) rv = -1;
    else {
        int r = (st)->a[(st)->front];
        (st)->front = ((st)->front + 1) % MAXQUEUE;
        (st)->size--; rv = r;
    }
}

#define o_isempty(st,rv) rv = ((st)->size==0 ? 1 : 0)

A.5 Array Stack

#define MAXSTACK 5

struct state_t {
    int a[MAXSTACK];
    int top;
}

#define o_new(res) res = (struct state_t *)malloc(sizeof(struct state_t)); res->top = -1;

#define o_push(st,rv,v) {
    if ((st)->top == (MAXSTACK-1)) {rv = 0;}
    else {
        (st)->top++; (st)->a[(st)->top] = v; rv = 1;
    }
}

#define o_pop(st,rv) {
    if ((st)->top == -1) rv = -1;
    else rv = (st)->a[(st)->top--];
}

#define o_isempty(st,rv) rv = ((st)->top==-1 ? 1 : 0)

A.6 Hash Table

#define HTCAPACITY 11

struct entry_t { int key; int value; };

struct state_t { struct entry_t table[HTCAPACITY]; int keys; };

#include <stdlib.h>

#define o_new(st) { s1 = malloc(sizeof(struct state_t));
    for (int i=0;i<HTCAPACITY;i++) { s1->table[i].key = -1; } 
    s1->keys = 0; }

#define o_put(st,rv,k,v) { int slot = k % HTCAPACITY;
    if (((st)->a[slot].key == -1) {
        (st)->a[slot].key = k;
        (st)->a[slot].value = v;
        (st)->keys++; rv=1;
    } else if (((st)->a[slot].key == k) {
        (st)->a[slot].value = v;
        rv = 1;
    } else if (((st)->a[slot].key != k) {
        rv = -1;
    } else {
        rv = -1; }
}

#define o_get(st,rv,k) { int slot = k % HTCAPACITY;
    if (((st)->a[slot].key == k) rv = -1;
    else rv = (st)->a[slot].value; }

#define o_rm(st,rv,k) { int slot = k % HTCAPACITY;
    if (((st)->a[slot].key == k) {
        (st)->a[slot].value = rv;
    } else {
        rv = -1; }
```
A.7 Simple Set

```c
#define o isempty(st, rv) { if ((st)->keys == 0) rv = 1; else rv = 0; }

struct state_t { int a; int b; int sz; }

#include <assert.h>
#include <stdlib.h>

#define o new(st,rv) { st = malloc(sizeof(struct state_t)); st->a = -1; st->b = -1; st->sz = 0; }

#define o add(st,rv,v) { if ((st)->a == -1 && (st)->b == -1) { (st)->a = v; (st)->b = (st)->b + 0; (st)->sz++; rv = 0; } else if ((st)->a != -1 && (st)->b == -1) { (st)->b = v; (st)->a = (st)->a + 0; (st)->sz++; rv = 0; } else if ((st)->a == -1 && (st)->b != -1) { (st)->a = v; (st)->b = (st)->b + 0; (st)->sz++; rv = 0; } else { rv = 0; }}

#define o isin(st, rv, v) { rv = 0; if ((st)->a == v) rv = 1; if ((st)->b == v) rv = 1; }

#define o getsize(st, rv) { rv = (st)->sz; }

#define o clear(st, rv) { (st)->a = -1; (st)->b = -1; (st)->sz = 0; rv = 0; }

#define o norm(st, rv) { if ((st)->a > (st)->b) { int t = (st)->b; (st)->b = (st)->a; (st)->a = t; } } rv = 0; }

A.8 List

#define LISTCAP 5

struct state_t { int list [LISTCAP]; int end; }

/*struct state_t * o_clone(struct state_t * s1);
int o_add(struct state_t * s, int v);
int o_rm(struct state_t * s, int v);
int o_contains(struct state_t * s, int v);
int o isempty(struct state_t * s);
int o print (struct state_t * s);*/

#include <stdio.h>
#include <stdlib.h>

#define o new(st) { st = malloc(sizeof(struct state_t)); st->end = -1; }

#define o add(st,rv,v) { rv = 1; for (int i=0; i<(st)->end; i++) { if ((st)->list[i] == v) { rv = 0; break; } }
if (rv == 1 && (st)->end == LISTCAP) { rv = 0; }
else { (st)->list[(st)->end+1] = v; (st)->end++; }
```
```c
#define o_rm(st, rv, v) {
    rv = 0;
    for (int i = 0; i <= (st)->end; i++) {
        if ((st)->list[i] == v) {
            (st)->list[i] = (st)->list[(st)->end]; (st)->end--; rv = 1; break;
        }
    }
}
#define o_contains(st, rv, v) {
    rv = 0;
    for (int i = 0; i <= (st)->end; i++) {
        if ((st)->list[i] == v) rv = 1; break;
    }
}
#define o_isempty(st, rv) rv = ((st)->end - (st)->begin) > 0 ? 0 : 1
```