Automating the Proof that Data Structure Implementations Commute with mn-Differencing

Eric Koskinen
Stevens Institute of Technology

Kshitij Bansal
Google, Inc.

***Abstract*—Commutativity of data structure methods is of ongoing interest, with roots in the database community. In recent years there has been renewed interest, with results showing that commutativity is 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 condition can be verified automatically from its implementation. Existing strategies based on ADT specifications struggle to find the right assertion granularity; and commutativity cannot be reduced to 2-safety in a straightforward way.**

We describe techniques to automatically prove the correctness of method commutativity conditions from data structure implementations. The key enabling insight is to characterize the precision necessary for commutativity reasoning using, what we call, mn-differencing relations. We then describe a reduction to reachability that decomposes the problem using mn-differencing relations and observational equivalence relations. Finally, we describe a proof-of-concept implementation and encouraging preliminary experiments, verifying commutativity of simple data structures such as a memory cell, counter, two-place Set, array-based stack, queue and a rudimentary hash table.

Source code is available (perhaps with help from the PC chair) at http://www.erickoskinen.com/cityprover/.

I. Introduction

For an object o, with state σ let $o.m(\bar{x})/\bar{r}$ denote a method signature, including a vector of corresponding return values $\bar{r}$. *Commutativity* of two methods, denoted $o.m(\bar{x})/\bar{r} \bowtie o.n(\bar{y})/\bar{s}$, are circumstances where 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 whether the two operations will always commute from σ.

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 [1], transactional memory [2], [3], [4], optimistic parallelism [5], speculative execution, features [6], layer concurrent programs [7], and software scalability [8]. If two code fragments commute then, when combined with linearizability (for which proof techniques exist, *e.g.*,[9], [10]) they can be executed concurrently. It is thus 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 [11], [12] or else they rely on data structure specifications as intermediaries [13], [14], which has pitfalls (see Sec. II).

Intuitively, commutativity is a multi-trace property: relating the behaviors in one circumstance with those in another. It is therefore tempting to pose the problem as a k-safety problem and attempt to leverage existing techniques for k-safety [15], [16], [17], [18], [19]. As we detail in Section II, however, the reduction is not immediate: the post-condition for commutativity necessitates a weaker notion than concrete equivalence and approaches that attempt to use specifications [13], [14] struggle to determine what granularity is appropriate for commutativity. Weak specifications lead to unsound conclusions, while strong specifications are unnecessarily burdensome to derive.

Contributions. We describe the first methods for verifying a given commutativity condition of a data structure, directly from its source code.

(1) We begin with a reduction $\textsc{Reduce}_m^n$ to automaton reachability that is designed to strengthen the pre-condition by only considering reachable data-structure states and weaken the notion of data-structure equivalence in the post-condition to observational equivalence. (Sec. IV) Although $\textsc{Reduce}_m^n$ is sound, reachability solvers struggle to verify the resulting encoding: their abstraction strategies lack the ability to decompose the problem in a manner suitable to commutativity.

(2) To resolve this issue, we return to the question of finding the appropriate abstraction granularity for commutativity. 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, while abstracting away state mutations that would be the same, regardless of the order in which they are applied. R_α-related post-states may not yet be equivalent. We show the pieces fit together by using an observational equivalence relation I_β. Proving I_β is an observational equivalence relation can be done using a separate abstraction β that is more appropriate for that concern. Theorem V.1 shows that a proof using this decomposition entails that φ_m^n is a valid commutativity condition.

(3) We introduce a second reduction $\text{DA}\textsc{Reduce}_m^n$, which exploits mn-differencing and observational equivalence relations. $\text{DA}\textsc{Reduce}_m^n$emits two reachability tasks: automata $\mathcal{A}_{\mathbf{A}}(\varphi_m^n, I)$ and $\mathcal{A}_{\mathbf{B}}(I)$. Notably, $\mathcal{A}_{\mathbf{B}}(I)$ is independent of m, n

```
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);}
  bool 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 non-zero identifiers (using fields a and b) and tracking the size sz of the Set. On the right, a simple array-based stack, ArrayStack.

and φ_m^n, so it can be proved safe once and then reused for every subsequent φ_m^n query. DAREDUCE$_m^n$ allows reachability analyses to synthesize separate abstractions α and β for $\mathcal{A}_{\mathbf{A}}(\varphi_m^n, I)$ and $\mathcal{A}_{\mathbf{B}}(I)$, respectively.

(4) We implement REDUCE$_m^n$ and DAREDUCE$_m^n$ in a new tool called CITYPROVER[1]. (Sec. VII) It takes as input simple data structures in C (with integers, arrays, and some pointers) and a candidate formula φ_m^n. CITYPROVER employs Ultimate's [20] or CPAchecker's [21] reachability analyses to prove safety of the automata tasks (or generate counterexamples).

(5) We report encouraging preliminary results that CITYPROVER can verify commutativity properties of some simple numeric data structures such as a memory cell, counter, two-place Set, array stack, array queue and rudimentary hash table. (Sec. VIII) We also show that DAREDUCE$_m^n$ scales better than REDUCE$_m^n$. Commutativity conditions can be fairly compact so, with CITYPROVER, a user can guess commutativity conditions and rely on CITYPROVER to prove them or report counterexamples.

Limitations. We focused on numeric programs but mn-differencing abstractions and our reductions generalize to heap programs, left for future work. Our reductions highlight some limitations of existing reachability solvers, namely, the need for improved disjunctive reasoning about permutations, a subject we leave to future work. We plan to release our benchmarks to SVCOMP so that can be used in the future.

II. MOTIVATING EXAMPLES

Consider the SimpleSet data structure 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 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.

Methods isin(x) and isin(y) always commute because neither modifies the ADT, so we say $\varphi_{\mathsf{isin}(x)}^{\mathsf{isin}(y)} \equiv \mathsf{true}$. Commutativity of 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, 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 commutativity condition $\varphi_{\mathsf{push}(x)}^{\mathsf{pop}} \equiv \mathsf{top} > -1 \wedge \mathsf{A}[\mathsf{top}] = x \wedge \mathsf{top} < \mathsf{MAX}$ 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.

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 because many parallelization strategies [2], [3], [5], [6], [7], [8] crucially depend on them being correct and, if they aren't, then parallelization becomes unsafe. Despite some attempts [12], [13], [11], [14], to our knowledge, there are no *sound* techniques for verifying commutativity conditions from source code. We will now discuss what's lacking in the state of the art, answering (i) why these existing works are insufficient and (ii) why the problem cannot easily be reduced to 2-safety.

What's hard about this problem? At first, commutativity seems like it could be easily reduced to a 2-safety problem. To prove that φ_m^n is a commutativity condition for $m(\bar{x}) \bowtie n(\bar{y})$, one could attempt to pose the problem as a 2-safety verification, perhaps using the following Hoare quadruple:

$$\begin{array}{l} \{\varphi_m^n \wedge \sigma_1 = \sigma_2\} \\ \begin{array}{l|l} r_m^1 := m(\bar{a}); & r_n^2 := n(\bar{b}); \\ r_n^1 := n(\bar{b}); & r_m^2 := m(\bar{a}); \end{array} \\ \{r_m^1 = r_m^2 \wedge r_n^1 = r_n^2 \wedge \sigma_1' = \sigma_2'\} \end{array}$$

This would be convenient because it would allow us to use existing 2-safety tools such as Descartes [18] or Weaver [19]. If we try the ArrayStack $\varphi_{\mathsf{push}(x)}^{\mathsf{pop}()} \equiv A[top] = x \wedge top > 1 \wedge top < MAX$ example, running an existing tool (*e.g.* using a product program [17] and Ultimate [20]) yields a counterexample, with the starting state: $A = [z, y, x, \alpha] \wedge top = 2$. The counterexample shows that in this case the post states

[1]Released soon. Available at https://file.io/V0GSL5Qe.

are different. Depending on the order methods are applied, one reaches either $A = [z, y, x, \alpha] \wedge top = 2$ or else $A[z, y, x, x] \wedge top = 2$. Our knowledge of the semantics of a stack tell us that these are the same state (because the garbage 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 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 in Kim and Rinard [13] and Bansal *et al.* [14]. Then we ask whether $Post_m(Post_n(\sigma_1)) = Post_n(Post_m(\sigma_1))$. One limitation with this strategy is that specifications are not always available, especially for *ad hoc* data structures and inferring such specifications is difficult. However, there is a bigger issue: it is unclear what precision is appropriate for commutativity. Consider a coarse specification such as $\{true\}\mathsf{push}(x)\{true\}$. Using this specification in our Hoare quad may lead to a post-relation $\{true\}$, which seems to indicate that all post-states are related and we would be inclined to incorrectly conclude that any φ^n_m is a valid commutativity condition. When specifications are too coarse like this one, Bansal *et al.* [14] would incorrectly synthesize commutativity condition $\varphi^{\mathsf{push}(y)}_{\mathsf{push}(x)} \equiv true$. The problem is that abstraction does not capture effects of $\mathsf{push}(x)$ that are relevant to commutativity.

Alternatively, we might try a more-fine grained specification using, *e.g.*, a sequential S to represent the stack and carefully relating every post-state to the pre-state, with disjunction to account for the various cases, etc. Such fine-grained specifications are particularly hard to come by for programmers' home-grown data structures, especially at this granularity that is nearly full-functional verification. Moreover, it isn't 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.

Challenges & Contributions. To begin, in Sec. IV we introduce a "one shot" reduction $\textsc{Reduce}^n_m$ from verifying commutativity conditions of an ADT to a (single) automaton reachability problem. $\textsc{Reduce}^n_m$ accounts for a few key factors. First, we observe that the reduction can be done pairwise, focusing the problem on the method pair m,n of concern. Second, $\textsc{Reduce}^n_m$ ensures that we only concern ourselves with commutativity from *reachable* states of the object. Third, in the post-relation, we exploit the automaton-based treatment to weaken the notion of equivalence to *observational* equivalence. We prove $\textsc{Reduce}^n_m$ to be sound but demonstrate that it does not lead to scalable tools. Reachability solvers struggle to effectively decompose the problem.

We thus return to the question, *What is the right abstraction granularity for commutativity?* which foiled prior works. We first observe that the necessary precision depends on methods under consideration. For example, with SimpleSet and commutativity of $\mathsf{isin}(y)$/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.*, a predicate 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 ArrayStack $\mathsf{push}(x)$/$\mathsf{pop}()$, we only need to consider the top value and we can abstract away deeper parts of the stack. While, on the other hand, for $\mathsf{pop}() \bowtie \mathsf{pop}()$, the second-from-top matters.

In Sec. V we present mn-differencing abstraction, which formalizes this intuition. 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. For SimpleSet , we can define α based on the above predicates, and then use the relation:
$R_\alpha(\sigma_1, \sigma_2) \equiv [(\mathsf{a} = x)_1 \vee (\mathsf{b} = x)_1] \Leftrightarrow [(\mathsf{a} = x)_2 \vee (\mathsf{b} = x)_2]$, *i.e.* the relation that tracks if σ_1 and σ_2 agree on those predicates. R_α is a relation on abstract states whose purpose is to "summarize" the possible pairs of post-states that will have agreed on return values.

States that are related by R_α may not necessarily be observationally equivalent and, thus far, we don't have a way of summarizing observational equivalence. We next show that the pieces fit together by working with observational equivalence *relations*. For reasoning about this equivalence, we use a separate abstraction β and a relation I_β in that abstract domain and describe the (standard) conditions under which I_β implies observational equivalence. For the ArrayStack and SimpleSet examples, we can use:

$$I_{AS}(\sigma_1, \sigma_2) \equiv \mathsf{top}_1 = \mathsf{top}_2 \wedge (\forall i \in [0, \mathsf{top}_1].\ \mathsf{a}_1[i] = \mathsf{a}_2[i])$$
$$I_{SS}(\sigma_1, \sigma_2) \equiv ((\mathsf{a}_1 = \mathsf{a}_2 \wedge \mathsf{b}_1 = \mathsf{b}_2) \vee (\mathsf{a}_1 = \mathsf{b}_2 \wedge \mathsf{b}_1 = \mathsf{a}_2)) \wedge\ (\mathsf{sz}_1 = \mathsf{sz}_2)$$

I_{AS} says the two states agree on the (ordered) values in the Stack. (top_1 means the value of top in σ_1.) 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 relations are simpler than full ADT specifications. Putting it all together, Theorem V.1 shows that if there is an R_α and I_β such that φ^n_m "implies" R_α and R_α "implies" I_β, then φ^n_m is a valid commutativity condition.

An outcome of this decomposition is that reasoning about I_β (which pertains to all methods of the ADT) can be separated from reasoning about R_α (which pertains to a given triple m, n, φ^n_m). Consequently, the first part can be done once, and then the second part can be done for each new commutativity validity query. In Sec. VI we describe a more modular reduction $\text{DA}\textsc{Reduce}^n_m$, which employs mn-differencing and observational equivalence relations. $\text{DA}\textsc{Reduce}^n_m$ emits a *pair* of automata $\mathcal{A}_{\mathbf{A}}(\varphi^n_m, I)$ and $\mathcal{A}_{\mathbf{B}}(I)$, such that the safety of the former entails that $\varphi^n_m \Rightarrow R_\alpha$ and that $R_\alpha \Rightarrow I_\beta$ while the safety of the latter entails that I_β captures observational equivalence.

Finally, in Sec. VII we describe a preliminary implementation CITYPROVER and in Sec. VIII we show itcan verify commutativity properties of some simple data structures. We also show that DAREDUCE^n_m is more tractable than REDUCE^n_m.

III. PRELIMINARIES

Language of ADT implementations. We work with a simple model of a (sequential) object-oriented language. Objects can have member fields $o.a$ and, for the purposes of this paper, we assume them to be integers, structs or integer arrays. We use $\bar{a}$ to denote a vector of argument *values*, $\bar{u}$ to denote a vector of return *values* and $m(\bar{a})/\bar{u}$ 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) [22], discussed in the next Section. Edges are labeled with straight-line code denoted s. For simplicity, we assume one object method cannot call another, and all object methods terminate. Non-terminating object methods are typically not useful and their termination can be confirmed using existing termination tools (*e.g.* [23], [24], [25], [26], [20]).

We fix a single object o, denote that object's concrete state space Σ. We denote $\sigma \xrightarrow{m(\bar{a})/\bar{u}} \sigma'$ for the big-step semantics in which the arguments are provided, and the entire method is reduced. We omit the small-step semantics $[\![s]\!]$. 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 [14] and prophecy variables [27], resp.

Definition III.1 (Observational equivalence for commutativity (*e.g.* [4]))**.** *We define relation $\simeq \subseteq \Sigma \times \Sigma$ as the following gfp:*

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

The above definition expresses that two states σ_1 and σ_2 of an object are observationally equivalent $\simeq$ provided that, when any given action $m(\bar{a})$ is applied to both σ_1 and σ_2, then the respective return values agree. Moreover, the resulting post-states maintain the $\simeq$ relation. An observational equivalence *relation* I is a relation such that $I \Rightarrow \simeq$.

We next use observational equivalence to define commutativity ([28], [14]) first at the layer of an action, which are particular *values*, and second at the layer of a method.

Definition III.2 (Commutativity)**.** *For methods m and n, and values $\bar{a}, \bar{b}, \bar{u}, \bar{v}$,* **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 for all σ, if $\sigma \xrightarrow{m(\bar{a})/\bar{u}} \sigma_m \xrightarrow{n(\bar{b})/\bar{v}} \sigma_{mn}$ and $\sigma \xrightarrow{n(\bar{b})/\bar{v}} \sigma_n \xrightarrow{m(\bar{a})/\bar{u}} \sigma_{nm}$ then $\sigma_{mn} \simeq \sigma_{nm}$. (Action commutativity requires return value agreement.)* **Methods** *$o.m$ and $o.n$ commute denoted $o.m \bowtie o.n$ provided that $\forall \bar{a}\ \bar{b}\ \bar{u}\ \bar{v}.\ o.m(\bar{a})/\bar{u} \bowtie o.n(\bar{b})/\bar{v}$.*

Quantification $\forall \bar{a}\ \bar{b}\ \bar{u}\ \bar{v}$ above means vectors of all possible argument and return values. Our work extends to a more fine-grained notion of left-movers and right-movers [29].

We denote a commutativity condition as φ^n_m and assume a decidable interpretation of formulae: $[\![\varphi^n_m]\!] : (\sigma, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \rightarrow \mathbb{B}$. The first argument is the initial state. Commutativity *post*- and *mid*-conditions can also be written [13] but for simplicity we focus on *pre*-conditions.

Definition III.3 (Commutativity Condition)**.** *We say that a formula φ^n_m is a commutativity condition for m and n provided that $\forall \sigma\ \bar{a}\ \bar{b}\ \bar{u}\ \bar{v}.\ [\![\varphi^n_m]\!]\ \sigma\ \bar{a}\ \bar{b}\ \bar{u}\ \bar{v} \Rightarrow m(\bar{a})/\bar{u} \bowtie n(\bar{b})/\bar{v}$.*

IV. ONE-SHOT REDUCTION TO REACHABILITY

We now describe an algorithm for reducing the task of verifying commutativity condition φ^n_m to reachability, which incorporates only reachable object states in the precondition, and employs observational equivalence for reasoning about data structure equality. The algorithm is a transformation REDUCE^n_m from an input object CFA to an output automaton $\mathcal{A}(\varphi^n_m)$ with an error state q_{er}. We prove if q_{er} is unreachable in $\mathcal{A}(\varphi^n_m)$, then φ^n_m is a valid commutativity condition.

Object Implementations. Our formalism needs a representation of object implementations, and the output encoding. We build on the well-established notion of control-flow automata:

Definition IV.1 ([22])**.** *A (deterministic)* control flow automaton *$\mathcal{A} = \langle Q, q_0, X, s, \longrightarrow\!\!\!\!\!\rightarrow \rangle$ is a finite set Q of control locations, initial location q_0, set X of typed variables, loop/branch-free statement language s and finite set of labeled edges $\longrightarrow\!\!\!\!\!\rightarrow \subseteq Q \times s \times Q$.*

We summarize other standard definitions (more detail in [30].). We define a *valuation* of variables $\theta : X \rightarrow Val$ and $\theta' \in [\![s]\!]\theta$. A *run* $r = q_0, \theta_0, q_1, \theta_1, q_2, \ldots$. We say $\mathcal{A}$ can *reach* automaton state q provided there exists a run to q. We use q_{er} as a special error state. We next conservatively extend CFAs to represent data structure implementations:

Definition IV.2 (Object implementation CFAs)**.** *An* object impl. CFA *for object o with methods $M = \{m_1, \ldots m_k\}$, is:*

$$\mathcal{A}_o = \langle Q_o, [q_0^{\textit{init}}, q_0^{\textit{clone}}, q_0^{m_1}, \ldots, q_0^{m_k}], X_o, s, \longrightarrow\!\!\!\!\!\rightarrow \rangle$$
$$X_o = X^{st} \cup \{\textit{this}_o\} \bigcup_{f \in M \cup \{\textit{init,close}\}} (X^f \cup X^f_{\bar{x}_f} \cup X^f_{\bar{r}_f})$$

(Detailed explanation in [30].)

Above, we will call each $q_0^{m_i}$ 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* $q_{ex}^{m_i}$. We require that the edges that lead to $q_{ex}^{m_i}$ contain $\mathrm{return}(\bar{v})$ statements. Subsets of variables X_o are reserved for object fields, this, method-local variables, parameters and return variables. For incorporating data structure implementations, we use inlining sugar: $q \dashrightarrow\! \mathsf{inl}(m_i, o, \bar{x}, \bar{r}) \longrightarrow\!\!\!\!\!\rightarrow_{\mathcal{E}} q' \equiv \{q \xrightarrow{\bar{x}_i := \bar{x};}\!\!\!\!\rightarrow_{\mathcal{E}} q_0^{m_i},\ q_{ex}^{m_i} \xrightarrow{\bar{r} := \bar{r}_i;}\!\!\!\!\rightarrow_{\mathcal{E}} q'\}$ This definition emulates calls to a method m_i, starting from CFA node q. Values $\bar{x}$ are provided as arguments, and arcs are created to the entry node $q_0^{m_i}$ for method m_i. Return values are saved into $\bar{r}$ and there is an arc from the exit node $q_{ex}^{m_i}$ to q'.

Edges are required to be deterministic. This is not without loss of generality: nondeterminism can be supported through

$\mathcal{A}(\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)}) \equiv$

```
1 SimpleSet s1 = new SimpleSet();
2 while(*) { int t = *; assume (t>0); switch(*) {
3   case 1: s1.add(t); case 2: s1.isin(t);
4   case 3: s1.size(); case 4: s1.clear(); }}
5 int x = *; int y = *;
6 assume( φ_{add(x)}^{isin(y)}(s1,x,y) );
7 SimpleSet s2 = s1.clone();
```
(A)

// Precondition: $\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)}$ and any reachable object state, duplicated

```
5  r^1_m = s1.add(x);  |  r^2_n = s1.isin(y);
   r^1_n = s2.isin(y); |  r^2_m = s2.add(x);
6 if(r^1_m ≠ r^2_m || r^1_n ≠ r^2_n) ERR;
```
(B)

// Postcondition: Now must show observational equivalence $\mathsf{s1} \simeq \mathsf{s2}$

```
11 while(true) { int t = *; assume (t>0); switch(*) {
12    case 1: if(s1.add(t) ≠ s2.add(t)) ERR;
13    case 2: if(s1.isin(t) ≠ s2.isin(t)) ERR;
14    case 3: if(s1.clear() ≠ s2.clear()) ERR;
15    case 4: if(s1.size() ≠ s2.size()) ERR; } }
```
(C)

Fig. 2. An example of REDUCE_m^n, when applied to the source of $\mathsf{add}(x)$ and $\mathsf{isin}\,(y)$. Formally, the result is an automaton but here it is depicted as a program. When a candidate $\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)}$ is supplied to this encoding, a proof of safety entails that $\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)}$ is a commutativity condition.

prophecy variables (see [31]). The semantics $(q, \theta) \xrightarrow{s} (q', \theta')$ of $\mathcal{A}_o$ induce a labeled transition system, with state space $\Sigma_{\mathcal{A}_o} = Q_o \times \Theta$. Naturally, commutativity of an object CFA is defined in terms of this induced transition system.

Example: SimpleSet. We first illustrate REDUCE_m^n by demonstrating the result of applying it to the SimpleSet example from Section II. Fig. 2 shows the output of $\text{REDUCE}_m^n(\mathcal{A}_{SS}, \mathsf{add}(x), \mathsf{isin}\,(y))$, which is an automaton denoted $\mathcal{A}(\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)})$ represented in pseudocode. Note ERR is q_{er}. $\mathcal{A}(\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)})$ should never be executed. Rather, when a program analysis tool for reachability is applied, the tool is tricked into finding abstractions to prove commutativity. There are three main portions to $\mathcal{A}(\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)})$:

(A) *Establishing the pre-condition.* For any reachable abstract state σ of the SimpleSet object, there will be a run of $\mathcal{A}(\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)})$ such that the SimpleSet on Line 7 will be in state σ. 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 $\mathsf{s1}$. From $\mathsf{s1}$, $\mathcal{A}(\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)})$ assumes that provided commutativity condition φ_m^n on Line 6 and runs will clone $\mathsf{s1}$.

(B) *Product program.* We next employ a standard product-program construction [17], using a trivial alignment. This portion causes 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 6.

(C) *Post-condition with observational equivalence.* Lines 11-15 consider any sequence of method calls $m'(\bar{a}'), m''(\bar{a}''), \ldots$ that could be applied to both $\mathsf{s1}$ and $\mathsf{s2}$. If observational equivalence does not hold, then there will be a run of $\mathcal{A}(\varphi_{\mathsf{add}(x)}^{\mathsf{isin}\,(y)})$ that applies that sequence to $\mathsf{s1}$ and $\mathsf{s2}$, eventually finding a discrepancy in return values and going to q_{er}.

Transformation REDUCE_m^n. The above example provides intuition and, for lack of space, the details of the following definition are in Appendix A.

Definition IV.3 (REDUCE_m^n). *For an input CFA $\mathcal{A}_o = \langle Q_o, [q_0^c, q_0^{\mathit{init}}, q_0^{m_1}, \ldots, q_0^{m_k}], X_o, s, \longrightarrow\!\!\!\!\rightarrow \rangle$, the output of $\text{REDUCE}_m^n(\mathcal{A}_o, m(\bar{x}), n(\bar{y}))$ is automaton $\mathcal{A}(\varphi_m^n)$ given in Appendix A.*

We now show $\mathcal{A}(\varphi_m^n)$'s safety entails commutativity.

Theorem IV.1. *For object implementation $\mathcal{A}_o$ and resulting encoding $\mathcal{A}(\varphi_m^n)$, if every run of $\mathcal{A}(\varphi_m^n)$ avoids q_{er}, then φ_m^n is a commutativity condition for $m(\bar{x})$ and $n(\bar{y})$.*

Proof Sketch. By case analysis on the runs of $\mathcal{A}(\varphi_m^n)$, correlating the variables in the valuation θ with the object state. $\square$

While REDUCE_m^n is sound, we show in Sec. VIII that tools don't scale well at proving the safety of REDUCE_m^n's output. In the next Sec. V we describe an abstraction targeted at proving commutativity to better enable automated reasoning.

V. mn-DIFFERENCING ABSTRACTION

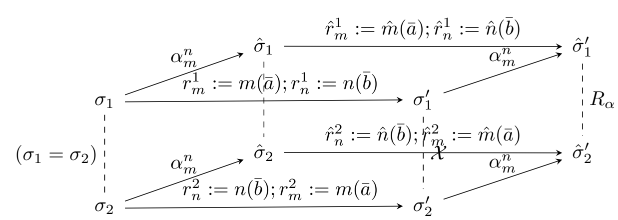
As seen in Sec. VIII, REDUCE_m^n generates an output verification task for which existing tools do not perform well. We now describe how to decompose the reduction into pieces. The challenge is thus: what kind of abstraction is coarse enough to be tractable, yet fine enough to reason about commutativity? We now address this with mn-differencing abstractions and later incorporate them into DAREDUCE_m^n. We first define posts to be the set of all pairs of post-states originating from σ after the methods are applied in the two alternate orders:

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

Return value agreement $\mathsf{rvsagree}$ is a predicate indicating that all such post-states agree on return values:

$$\mathsf{rvsagree}(\sigma, m, \bar{a}, n, \bar{b}) \equiv \text{for } \bar{r}_m^1, \bar{r}_n^1, \bar{r}_n^2, \bar{r}_m^2 \text{ then } \sigma \xrightarrow{m(\bar{a})/\bar{r}_m^1} \sigma_1 \xrightarrow{n(\bar{b})/\bar{r}_n^2} \sigma_1' \ \wedge \ \sigma \xrightarrow{n(\bar{b})/\bar{r}_n^2} \sigma_2 \xrightarrow{m(\bar{a})/\bar{r}_m^2} \sigma_2', \Rightarrow \bar{r}_m^1 = \bar{r}_m^2 \text{ and } \bar{r}_n^2 = \bar{r}_n^1$$

The idea of mn-differencing can be visualized as follows:



On the left, we start with states σ_1 and σ_2 that are exactly equal. The product program leads to σ_1' and σ_2'. For these post states, we require return value agreement: $\mathcal{X} \equiv r_m^1 = r_m^2 \wedge r_n^1 = r_n^2$. Next, we have an abstraction α_m^n, specific to this m/n pair, and a product program in this abstract domain.

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 $\mathsf{isin}(x)$/clear, let α be a predicate abstraction, 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
$R_\alpha(\sigma_1, \sigma_2) \equiv (\mathsf{a} = x)_1 \vee (\mathsf{b} = x)_1 \Leftrightarrow (\mathsf{a} = x)_2 \vee (\mathsf{b} = x)_2$, *i.e.* the relation that tracks if σ_1 and σ_2 agree on those predicates. Formally, mn-differencing is defined as:

Definition V.1. *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. Let $R_\alpha : \Sigma^\alpha \to \Sigma^\alpha \to \mathbb{B}$ be a relation on abstract states. We say that (α, R_α) is* an mn-differencing abstraction *if*

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

A relation R_α may not hold for every initial state σ, hence the need for a commutativity condition. So we need to ask whether R_α holds, under the assumption that φ_m^n holds in the pre-condition, defined as follows:

Definition V.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 φ_m^n* **implies** *(α, R_α) if $\forall \sigma\ \bar{a}\ \bar{b}\ \bar{r}\ \bar{s}.\ \varphi_m^n(\sigma, \bar{a}, \bar{b}, \bar{r}, \bar{s}) \Rightarrow R_\alpha(\alpha(\sigma_1), \alpha(\sigma_2))$ where $(\sigma_1, \sigma_2) = \textsf{posts}(\sigma, m, \bar{a}, n, \bar{b})$.*

If we let $\varphi_{\mathsf{isin}(x)}^{\mathsf{clear}} \equiv \mathsf{a} \neq x \wedge \mathsf{b} \neq x$, this will imply R_α in the posts: the post states will agree on whether x is in the set.

Post-state equivalence. States that are R_α-related are not necessarily equivalent. We identify the next stage of reasoning with observational equivalence *relations* and separate abstractions there for. This is a fairly standard definition of observational equivalence relations [32]. Importantly, we can use an abstraction β here that is separate from α. Formally,

Definition V.3. *Let $\beta : \Sigma \to \Sigma^\beta$ be an abstraction, with concretiz. $\delta : \Sigma^\beta \to \mathcal{P}(\Sigma)$, and let $I_\beta : \Sigma^\beta \times \Sigma^\beta \to \mathbb{B}$. I_β is an* observational equivalence relation *iff: $\forall \sigma_1^\beta\ \sigma_2^\beta \in \Sigma^\beta.\ I_\beta(\sigma_1^\beta, \sigma_2^\beta) \Rightarrow \forall \sigma_1 \in \delta(\sigma_1^\beta), \sigma_2 \in \delta(\sigma_2^\beta).\ \sigma_1 \simeq \sigma_2$*

I_{SS} and I_{AS}, defined earlier, are such relations. We connect R_α with I_β as follows:

Definition V.4. *We say (α, R_α)* **implies** *(β, I_β) iff:*
$\forall \sigma_1, \sigma_2 \in \Sigma . R_\alpha(\alpha(\sigma_1), \alpha(\sigma_2)) \Rightarrow I_\beta(\beta(\sigma_1), \beta(\sigma_2))$

To satisfy this implication, R_α may need to be more precise than simply witnessing return value agreement. In the case of SimpleSet, R_α must be refined so that it also implies that $\mathsf{sz}_1 = \mathsf{sz}_2$ and $(\mathsf{a}_1 = \mathsf{a}_2 \wedge \mathsf{b}_1 = \mathsf{b}_2) \vee (\mathsf{a}_1 = \mathsf{b}_2 \wedge \mathsf{a}_2 = \mathsf{b}_1)$.

Finally, sufficient conditions for a commutativity condition for the two methods, with respect to these abstractions are:

Theorem V.1. *Let φ_m^n be a logical formula on Σ and actions of m and n. If there exists (α_m^n, R_m^n) and (β, I_β), that φ_m^n implies (α_m^n, R_m^n) and (α_m^n, R_m^n) implies (β, I_β) then φ_m^n is a commutativity condition. (Proof sketch in Apx. B.)*

VI. Reachability and Differencing Abstractions

We now employ mn-differencing abstractions to introduce $\text{DAR}\textsc{educe}_m^n$ that decomposes reasoning into two phases: (**A**) finding a sufficient R_α that implies I_β and then (**B**) proving that I_β is an observational equivalence relation. In short, commutativity proving is reduced to the question:

$$\exists I.\ \exists R.\ \ \mathcal{A}_{\mathbf{A}}(\varphi_m^n, R, I) \text{ is safe} \quad \text{and} \quad \mathcal{A}_{\mathbf{B}}(I) \text{ is safe}$$

This allows tools separately synthesize abstractions β and α, each targeted to the reachability needs of the phase. Moreover, when operating on the automata from $\text{DAR}\textsc{educe}_m^n$, tools can sometimes synthesize I and, in all cases we evaluated, tools synthesized R. Finally, $\mathcal{A}_{\mathbf{B}}(I)$ turns out to be independent of method pair and can be proved once for the ADT and then commutativity conditions only need to be plugged into $\mathcal{A}_{\mathbf{A}}$.

Definition VI.1 ($\text{DAR}\textsc{educe}_m^n$)**.** *For an input object CFA $\mathcal{A}_o$, $\text{DAR}\textsc{educe}$ when applied to methods $m(\bar{x}), n(\bar{y})$, yields two automata $\mathcal{A}(\varphi_m^n, I)_{\mathbf{A}}$ and $\mathcal{A}(I)_{\mathbf{B}}$ defined as follows:*
Phase A: Proving that $\varphi_m^n \Rightarrow I$ via R_α:

$$\mathcal{A}(\varphi_m^n, I)_{\mathbf{A}} = \langle Q_{\mathbf{A}}, q_{\mathbf{A}}^0, X_{\mathbf{A}}, s_{\mathbf{A}}, \twoheadrightarrow_{\mathbf{A}} \rangle \textit{ where } \twoheadrightarrow_{\mathbf{A}} \textit{ is:}$$
$$\bigcup \left\{ \begin{array}{ll} \twoheadrightarrow & \textit{(o's source)} \\ q_{\mathbf{A}}^0 \text{—} \textsf{inl}(\textsf{init}, nil, [], [o_1]) \text{—}_{\mathbf{A}}\, q_1 & \textit{(Construct.)} \\ \cup_{m_i} \left\{ \begin{array}{l} q_1 \xrightarrow{\bar{x}:=\bar{*};}\!\!\twoheadrightarrow_{\mathbf{A}} q_{1i}, \\ q_{1i} \text{—} \textsf{inl}(m, o_1, \bar{x}, nil) \text{—}_{\mathbf{A}}\, q_1 \end{array} \right. & \textit{(Reachbl. } o_1\textit{)} \\ q_1 \xrightarrow{\bar{a}:=\bar{*};\ \bar{b}:=\bar{*};\ assume(\varphi_m^n(o_1, \bar{a}, \bar{b}))}\!\!\twoheadrightarrow_{\mathbf{A}} q_{11} & \textit{(Assume } \varphi_m^n\textit{)} \\ q_{11} \text{—} \textsf{inl}(\textsf{clone}, o_1, [], [o_2]) \text{—}_{\mathbf{A}}\, q_2 & \textit{(Clone } o_1\textit{))} \\ q_2 \text{—} \textsf{inl}(m, o_1, \bar{a}, \bar{r}_m^1) \text{—}_{\mathbf{A}}\, q_{21} \text{—} \textsf{inl}(n, o_1, \bar{b}, \bar{r}_n^1) \text{—}_{\mathbf{A}}\, q_{22} & \\ q_{22} \text{—} \textsf{inl}(n, o_2, \bar{b}, \bar{r}_n^2) \text{—}_{\mathbf{A}}\, q_{23} \text{—} \textsf{inl}(m, o_2, \bar{a}, \bar{r}_m^2) \text{—}_{\mathbf{A}}\, q_3 & \\ q_3 \text{—} \textsf{inl}(asm(\bar{r}_m^1 \neq \bar{r}_m^2 \vee \bar{r}_n^2 \neq \bar{r}_n^1)) \text{—}_{\mathbf{A}}\, q_{er}^{\mathbf{A}} & \textit{(Rv's agree)} \\ q_3 \text{—} \textsf{inl}(asm(\neg I)) \text{—}_{\mathbf{A}}\, q_{er}^{\mathbf{A}} & \textit{(Ensure } I\textit{)} \end{array} \right.$$

Phase B: Proving that I is an obs. eq. relation via β:

$$\mathcal{A}(I)_{\mathbf{B}} = \langle Q_{\mathbf{B}}, q_{\mathbf{B}}^0, X_{\mathbf{B}}, s_{\mathbf{B}}, \twoheadrightarrow_{\mathbf{B}} \rangle \textit{ where } \twoheadrightarrow_{\mathbf{B}} \textit{ is:}$$
$$\bigcup \left\{ \begin{array}{ll} \twoheadrightarrow, \{ q_{\mathbf{B}}^0 \xrightarrow{assume(I)}\!\!\twoheadrightarrow_{\mathbf{B}} q_2 \} & \textit{(o's source)} \\ \cup_{m_i} \left\{ \begin{array}{l} q_2 \xrightarrow{\bar{a}:=\bar{*}}\!\!\twoheadrightarrow_{\mathbf{B}} q_{2i} \\ q_{2i} \text{—} \textsf{inl}(m_i, o_1, \bar{a}, \bar{r}) \text{—}_{\mathbf{B}}\, q_{2i'} \\ q_{2i'} \text{—} \textsf{inl}(m_i, o_2, \bar{a}, \bar{s}) \text{—}_{\mathbf{B}}\, q_{2i''} \\ q_{2i''} \xrightarrow{assume(\bar{r} \neq \bar{s})}\!\!\twoheadrightarrow_{\mathbf{B}} q_{er}^{\mathbf{B}} \\ q_{2i''} \xrightarrow{assume(\neg I)}\!\!\twoheadrightarrow_{\mathbf{B}} q_{er}^{\mathbf{B}} \end{array} \right. & \textit{(Any } m_i\textit{)} \end{array} \right.$$

The definitions for $Q_{\mathbf{A}}$, initial state $q_{\mathbf{A}}^0$, variables $X_{\mathbf{A}}$, statements $s_{\mathbf{A}}$ are straight-forward. Same for $\mathcal{A}(I)_{\mathbf{B}}$.

The $\text{DAR}\textsc{educe}_m^n$ output encoding $\mathcal{A}(\varphi_m^n, I)_{\mathbf{A}}$ (**Phase A**), like $\textsc{Reduce}_m^n$, begins with a region that ensures that, for any reachable state of the ADT, there will be a run of $\mathcal{A}(\varphi_m^n, I)_{\mathbf{A}}$ to location q_1 where o is in that reachable state. It also sets up the preconditions that φ_m^n hold and that $o_1 = o_2$ (concretely) and constructs the product program between the two orders of method application. Unlike $\textsc{Reduce}_m^n$, $\mathcal{A}(\varphi_m^n, I)_{\mathbf{A}}$ ends with a possible transition to $q_{er}^{\mathbf{A}}$ whenever $\neg I$ holds.

To prove that $\neg I$ can never hold, an analysis on $\mathcal{A}(\varphi_m^n, I)_\mathbf{A}$ must establish an invariant at q_3 that is (1) strong enough to capture return value agreement and (2) strong enough to imply I. This invariant will indeed be an mn-differencing relation due to (1), and it will imply I due to (2).

The DAReduce$_m^n$ output $\mathcal{A}(I)_\mathbf{B}$ (**Phase B**) is designed so that a safety proof on $\mathcal{A}(I)_\mathbf{B}$ entails that I is an observational equivalence relation. There is a pre-condition edge that assumes I, and edges to the entry node of each possible method m_i, nondeterministically selecting arguments. To prove that I is an observational equivalence relation, a reachability solver will synthesized an appropriate abstraction β.

Theorem VI.1. *For object CFA $\mathcal{A}_o$, and automata $\mathcal{A}(\varphi_m^n, I)_\mathbf{A}$ and $\mathcal{A}(I)_\mathbf{B}$ resulting from* DAReduce$_m^n$*, if there exists an I such that for every run of $\mathcal{A}(\varphi_m^n, I)_\mathbf{A}$ avoids $q_{er}^\mathbf{A}$, and every run of $\mathcal{A}(I)_\mathbf{B}$ avoids $q_{er}^\mathbf{B}$, then φ_m^n is a commutativity condition.*

Proof. Similar to the proof of Theorem IV.1 but employing Theorem V.1 to combine phases. □

DAReduce$_m^n$ improves over Reduce$_m^n$ by decomposing the verification problem, making it more tractable to automation (see Sec. VIII). One can employ a mn-differencing abstraction α for finding R_α and a separate abstraction β for observational equivalence. Phase **B** does not depend on the method pair under consideration. Consequently, a proof of safety of $\mathcal{A}(I)_\mathbf{B}$ can be done once for the entire ADT.

VII. Implementation

We developed a simple prototype implementation called CityProver[2]. CityProver is written in Perl and OCaml and takes, as input, C source code Examples input ADTs can be found in Appendix E. We have written them as C macros so that our experiments focus on commutativity rather than testing existing tools' inter-procedural reasoning. Also provided as input is a commutativity condition φ_m^n. CityProver then implements implements Reduce$_m^n$ and DAReduce$_m^n$ via a program transformation.

VIII. Experiments

There are no other existing tools for verifying ADT commutativity directly from source code. Our goals were to evaluate whether our reductions enable existing reachability solvers to verify commutativity conditions and whether DAReduce$_m^n$ performs better than Reduce$_m^n$. To this end, we created some small examples (with integers, simple pointers, 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 virtual host.

We began with simple ADTs including: a Memory cell; an Accumulator with increment, decrement, and a check whether the value is 0; and 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	$\varphi_{m(x_1)}^{n(y_1)}$	Exp.	**Oneshot** Reduce$_m^n$ CPA	**Oneshot** Reduce$_m^n$ Ult	DAReduce$_m^n$ CPA	DAReduce$_m^n$ Ult
Memory	rd ⋈ wr	$s1.x = y_1$	✓	1.4 ✓	0.7 ✓	3.9 ✓	1 ✓
Memory	rd ⋈ wr	true	χ	1.4 χ	0.2 χ	1.3 χ	0.2 χ
Memory	wr ⋈ wr	$y_1 = x_1$	✓	1.4 ✓	0.5 ✓	3.9 ✓	0.8 ✓
Memory	wr ⋈ wr	true	χ	1.3 χ	0.3 χ	2.4 χ	0.4 χ
Memory	rd ⋈ rd	true	✓	1.4 ✓	0.6 ✓	3.9 ✓	1 ✓
Accum.	dec ⋈ isz	$s1.x > 1$	✓	1.5 ✓	2.2 ✓	4 ✓	2.6 ✓
Accum.	dec ⋈ isz	true	χ	1.5 χ	0.7 χ	1.2 χ	0.6 χ
Accum.	dec ⋈ inc	$s1.x > 1$	✓	1.5 ✓	1.3 ✓	4.1 ✓	1.7 ✓
Accum.	dec ⋈ inc	true	✓	1.5 ✓	1.2 ✓	4 ✓	1.5 ✓
Accum.	inc ⋈ isz	$s1.x > 1$	✓	1.5 ✓	3.3 ✓	4.1 ✓	2.9 ✓
Accum.	inc ⋈ isz	true	χ	1.6 χ	0.7 χ	1.2 χ	0.6 χ
Accum.	inc ⋈ inc	true	✓	1.4 ✓	1.5 ✓	4.1 ✓	1.6 ✓
Accum.	dec ⋈ dec	true	✓	1.5 ✓	1.5 ✓	3.9 ✓	1.6 ✓
Accum.	dec ⋈ dec	$s1.x > 1$	✓	1.5 ✓	2.6 ✓	4 ✓	1.9 ✓
Accum.	isz ⋈ isz	true	✓	1.4 ✓	4.3 ✓	4 ✓	3.4 ✓
Counter	dec ⋈ dec	true	χ	1.9 χ	1.5 χ	4.2 ✓✗	1.2 χ
Counter	dec ⋈ dec	$s1.x \geq 2$	✓	1.5 ✓	13.0 ✓	4.1 ✓	5.9 ✓
Counter	dec ⋈ inc	true	χ	1.6 χ	0.3 χ	1.4 χ	0.3 χ
Counter	dec ⋈ inc	$s1.x \geq 1$	✓	1.6 ✓	6.8 ✓	4.2 ✓	3.8 ✓
Counter	inc ⋈ isz	true	χ	1.5 χ	0.8 χ	1.2 χ	0.7 χ
Counter	inc ⋈ isz	$s1.x > 0$	✓	1.5 ✓	5.3 ✓	4.1 ✓	2.6 ✓
Counter	inc ⋈ isz	$s1.x > 0$	χ	1.9 **?**	**TO** **?**	4.4 χ	6.9 χ
Counter	inc ⋈ clr	true	χ	1.3 χ	0.4 χ	2.5 χ	0.4 χ

Fig. 3. Results of applying CityProver to the simple benchmarks. For each benchmark, we report the time to use Reduce$_m^n$ versus DAReduce$_m^n$. A more detailed breakdown of DAReduce$_m^n$ can be found in Appendix C.

The objects, method pairs, and commutativity conditions are shown in the first few columns of Fig. 3, along with the **Exp**ected result. We ran CityProver using both the Reduce$_m^n$ and DAReduce$_m^n$ algorithms and, in each case, compared using CPAchecker and Ultimate as the underlying solver. For DAReduce$_m^n$, we report the total time taken for both Phase **A** and Phase **B**. A more detailed version of this table can be found in Appendix C. 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$_m^n$ 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 data structures that store and manipulate elements. For these ADTs, 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. The results of these benchmarks are given in Fig. 4 and an extended version is in Appendix D.. For SimpleSet most cases where straightforward. In almost all cases DAReduce$_m^n$ outperformed Reduce$_m^n$, with an average speedup of **3.88**×. *ArrayStack*. (Fig. 1) push(x)/pop condition ${}^1\varphi_{\text{push}}^{\text{pop}}$ is defined in Appendix D. 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 *Queue*, CityProver was able to prove all but two commutativity conditions. We finally applied

[2] To be released on GitHub. See https://file.io/V0GSL5Qe.

ADT	Methods $m(x_1), n(y_1)$	$\varphi^{n(y_1)}_{m(x_1)}$	Exp.	REDUCE^n_m Ult	DARE^n_m Ult
SSet	isin $\bowtie$ isin	true	✓	137.8 ✓	36.7 ✓
SSet	isin $\bowtie$ add	$x_1 \neq y_1$	✓	84.8 ✓	37.1 ✓
SSet	isin $\bowtie$ add	true	χ	2.4 χ	1.5 χ
SSet	isin $\bowtie$ clear	true	χ	2.6 χ	1.6 χ
SSet	isin $\bowtie$ clear	$x_1 \neq y_1$	χ	2.4 χ	1.6 χ
SSet	isin $\bowtie$ clear	$a_1 \neq x_1 \wedge b_1 \neq y_1$	χ	3.1 χ	1.4 χ
SSet	isin $\bowtie$ clear	$a_1 \neq x_1 \wedge b_1 \neq x_1$	✓	14.0 ✓	19.3 ✓
SSet	isin $\bowtie$ getsize	true	✓	41.3 ✓	24.2 ✓
AStack	push $\bowtie$ pop	$a_1[top] = x_1 \wedge top > 1 \wedge top < 5 - 1$	✓	**MO** –	95.5 ✓
AStack	push $\bowtie$ pop	true	χ	2.2 χ	2.0 χ
AStack	push $\bowtie$ push	true	χ	7.6 χ	17.0 χ
AStack	push $\bowtie$ push	$top < 3$	χ	3.9 χ	230.7 χ
AStack	push $\bowtie$ push	$x_1 = y_1$	χ	19.8 χ	17.3 χ
AStack	push $\bowtie$ push	$x_1 = y_1 \wedge top < 3$	✓	**MO** –	155.1 ✓
AStack	pop $\bowtie$ pop	$top = -1$	✓	**TO** –	38.0 ✓
AStack	pop $\bowtie$ pop	true	χ	2.1 χ	1.4 χ
Queue	enq $\bowtie$ enq	true	χ	39.8 χ	35.0 χ
Queue	deq $\bowtie$ deq	true	χ	3.5 χ	3.1 χ
Queue	deq $\bowtie$ deq	$size = 0$	✓	**TO** –	174.4 ✓
Queue	enq $\bowtie$ enq	true	χ	27.8 χ	23.3 χ
Queue	enq $\bowtie$ enq	$x_1 = y_1$	χ	63.8 χ	22.6 χ
Queue	emp $\bowtie$ emp	true	✓	**TO** –	**TO** –
Queue	enq $\bowtie$ deq	$size = 1 \wedge x_1 = a_1[front]$	✓	**TO** –	472.0 ✓
Queue	enq $\bowtie$ deq	true	χ	**MO** –	8.4 χ
Queue	enq $\bowtie$ emp	$size > 0$	✓	**MO** –	**TO** –
Queue	enq $\bowtie$ emp	$true$	χ	**MO** –	7.4 χ
Queue	deq $\bowtie$ emp	$size = 0$	✓	**MO** –	135.8 ✓
Queue	deq $\bowtie$ emp	$true$	χ	**MO** –	6.5 χ
HashTable	put $\bowtie$ put	${}^1\varphi^{\text{put}}_{\text{put}}$	χ	262.5 χ	97.5 χ
HashTable	put $\bowtie$ put	${}^2\varphi^{\text{put}}_{\text{put}}$	χ	202.7 χ	136.9 χ
HashTable	put $\bowtie$ put	${}^3\varphi^{\text{put}}_{\text{put}}$	✓	**TO** –	**TO** –
Hashtable	put $\bowtie$ put	${}^3\varphi^{\text{put}}_{\text{put}}$	✓	566.5 ✓	297.5 ✓
HashTable	put $\bowtie$ put	true	χ	**TO** –	102.3 χ
HashTable	get $\bowtie$ get	$o_1.keys = 0$	✓	**TO** –	**TO** –
HashTable	get $\bowtie$ get	true	✓	**TO** –	**TO** –
HashTable	get $\bowtie$ get	true	✓	50.0 ✓	56.8 ✓
HashTable	get $\bowtie$ put	$x_1 \neq y_1$	✓	**TO** –	**TO** –
HashTable	get $\bowtie$ 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 Appendix D.

CITYPROVER to a simple *HashTable*, where hashing is done only once and insertion gives up if there is a collision.

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 ${}^i\varphi^{\text{put}}_{\text{put}}$ (defined in Appendix D.) represent our repeated attempts to guess commutativity conditions. CITYPROVER's counterexamples pointed out collisions and capacity cases. For Hashtable Phase **B**, Ultimate had some trouble mixing modulus with array reasoning, so we used CPAchecker. We also had to introduce a prophecy variable to assist the verifiers in knowing that array index equality distributes over modulus of equal keys.

DAREDUCE^n_m can be amended so that it coërces tools to infer I_β (as well as R_α). For small examples, we could get Ultimate to infer I_β. For bigger examples, we manually provided I, but Ultimate discovered an appropriate abstraction β. For all examples, Ultimate inferred an R_α.

CITYPROVER, especially with DAREDUCE^n_m can promptly validate commutativity conditions for these 38 examples. For DAREDUCE^n_m, in 7 cases it took more than 2 minutes, and in 6 cases it reached the 15 minute timeout.

IX. Related work

a) Commutativity reasoning: Bansal *et al.* [14] 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 Section II, if specifications are coarse, Bansal *et al.* would emit unsound commutativity conditions. On the other hand, precise specifications are harder to come by (by hand or by static analysis) because the precision needed may be tantamount to full functional correctness specifications. We instead capture just what is needed for commutativity.

Gehr *et al.* [11] describe a method based on black-box sampling. Both Aleen and Clark [12] and Tripp *et al.* [33] identify sequences of actions that commute (via random interpretation and dynamic analysis, resp.). Kim and Rinard [13] verify commutativity conditions from specifications. Commutativity is also used in dynamic analysis [28]. Najafzadeh *et al.* [34] describe a tool for weak consistency, that reports commutativity checking of formulae, but not ADT implementations.

b) k-safety, product programs, reductions.: Reductions to reachability have been used in security. Self-composition [15], [16]—reduces (some forms of) hyperproperties [35] to properties of a single program. More recent works include product programs [17], [36] and a number of techniques for automated verification of k-safety properties. Cartesian Hoare Logic [18] is a program logic for reasoning about k-safety properties, automated via a tool called DESCARTES. Antonopoulos *et al.* [37] described an alternative automated k-safety technique based on partitioning the traces. 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 approaches to relational reasoning across multiple programs [38], [39].

X. Conclusion

We have described a theory, algorithm and tool for automatically verify commutativity conditions of data structure implementations. The key insight is that mn-differencing relations can be used to target commutativity reasoning and this can be employed in reduction DAREDUCE^n_m to decompose the problem to make it more amenable to general-purpose reachability solvers. Our proofs can enable commutativity conditions to be more safely integrated into various concurrency settings. In the future work, we hope to explore techniques for more complex data-structures, especially those with layout permutations.

Acknowledgments. We thank the anonymous reviewers for their helpful feedback on earlier drafts of this work.

References

[1] M. C. Rinard and P. C. Diniz, "Commutativity analysis: A new analysis technique for parallelizing compilers," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 6, pp. 942–991, November 1997.

[2] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly concurrent transactional objects," in *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'08)*, 2008.

[3] Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman, "Open nesting in software transactional memory," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007*. ACM, 2007, pp. 68–78. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229442

[4] E. Koskinen and M. J. Parkinson, "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*, 2015, pp. 186–195.

[5] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 211–222.

[6] M. Chechik, I. Stavropoulou, C. Disenfeld, and J. Rubin, "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.*, 2018, pp. 319–336.

[7] B. Kragl and S. Qadeer, "Layered concurrent programs," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 79–102. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_5

[8] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, p. 10, 2015. [Online]. Available: http://doi.acm.org/10.1145/2699681

[9] V. Vafeiadis, "Automatically proving linearizability," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 450–464.

[10] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza, "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*, 2015, pp. 95–107.

[11] T. Gehr, D. Dimitrov, and M. T. Vechev, "Learning commutativity specifications," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, 2015, pp. 307–323. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21690-4_18

[12] F. Aleen and N. Clark, "Commutativity analysis for software parallelization: letting program transformations see the big picture," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*. ACM, 2009, pp. 241–252.

[13] D. Kim and M. C. Rinard, "Verification of semantic commutativity conditions and inverse operations on linked data structures," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*. ACM, 2011, pp. 528–541.

[14] K. Bansal, E. Koskinen, and O. Tripp, "Automatic generation of precise and useful commutativity conditions," in *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Greece, April 2018, Proceedings, Part II*, 2017.

[15] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *CSFW*, 2004.

[16] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *SAS*, 2005.

[17] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *International Symposium on Formal Methods*. Springer, 2011, pp. 200–214.

[18] M. Sousa and I. Dillig, "Cartesian hoare logic for verifying k-safety properties," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. Berger, Eds. ACM, 2016, pp. 57–69. [Online]. Available: https://doi.org/10.1145/2908080.2908092

[19] A. Farzan and A. Vandikas, "Automated hypersafety verification," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds., vol. 11561. Springer, 2019, pp. 200–218. [Online]. Available: https://doi.org/10.1007/978-3-030-25540-4_11

[20] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski, "Ultimate automizer with smtinterpol," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 641–643.

[21] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 184–190.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer, "Temporal-safety proofs for systems code," in *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, 2002, pp. 526–538.

[23] M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman, "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*, 2016, pp. 387–393.

[24] "AProVE: Automated program verification environment," http://aprove.informatik.rwth-aachen.de/.

[25] "FuncTion," https://www.di.ens.fr/∼urban/FuncTion.html.

[26] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 184–190.

[27] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, pp. 253–284, May 1991. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(91)90224-P

[28] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen, "Commutativity race detection," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[29] R. J. Lipton, "Reduction: a method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717–721, 1975.

[30] E. Koskinen and K. Bansal, "Reducing commutativity verification to reachability with differencing abstractions," 2020, extended arXiv version. https://arxiv.org/abs/2004.08450.

[31] B. Cook and E. Koskinen, "Making prophecies with decision predicates," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, 2011, pp. 399–410. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926431

[32] T. Bolognesi and S. A. Smolka, "Fundamental results for the verification of observational equivalence: A survey." in *PSTV*, 1987, pp. 165–179.

[33] O. Tripp, R. Manevich, J. Field, and M. Sagiv, "JANUS: exploiting parallelism via hindsight," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, 2012, pp. 145–156. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254083

[34] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro, "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*, 2016, pp. 2:1–2:3.

[35] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[36] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 42, no. 1, pp. 1–37, 2019.

[37] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, A. Cohen and M. T. Vechev, Eds. ACM, 2017, pp. 362–375. [Online]. Available: https://doi.org/10.1145/3062341.3062378

[38] A. Banerjee, D. A. Naumann, and M. Nikouei, "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.

[39] D. Frumin, R. Krebbers, and L. Birkedal, "Reloc: A mechanised relational logic for fine-grained concurrency," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, 2018, pp. 442–451.

$$
\mathcal{A}(\varphi_m^n) = \langle Q_{\mathcal{E}}, q_0^{\mathcal{E}}, X_{\mathcal{E}}, s_{\mathcal{E}}, \twoheadrightarrow_{\mathcal{E}} \rangle \text{ where } \twoheadrightarrow_{\mathcal{E}} \equiv
$$

$$
\cup \left\{ \begin{array}{ll}
\twoheadrightarrow & (o\text{'s source}) \\
q_0^{\mathcal{E}} \;\dashv \mathsf{inl}(\,\mathsf{init}\,, nil, [], [o_1]) \twoheadrightarrow_{\mathcal{E}} q_1 & (\text{Construct.}) \\
\cup_{m_i} \left\{ \begin{array}{l} q_1 \overset{\bar{x}:=\bar{*};}{\twoheadrightarrow}_{\mathcal{E}} q_{1i}, \\ q_{1i} \;\dashv \mathsf{inl}(m, o_1, \bar{x}, nil) \twoheadrightarrow_{\mathcal{E}} q_1 \end{array} \right. & (\text{Reachbl. } o_1) \\
q_1 \xrightarrow{\bar{a}:=\bar{*};\ \bar{b}:=\bar{*};\ assume(\varphi_m^n(o_1,\bar{a},\bar{b}))}\!\!\!\twoheadrightarrow_{\mathcal{E}} q_{11} & (\text{Assume } \varphi_m^n) \\
q_{11} \;\dashv \mathsf{inl}(\mathsf{clone}, o_1, [], [o_2]) \twoheadrightarrow_{\mathcal{E}} q_2 & (\text{Clone } o_1)) \\
q_2 \;\dashv \mathsf{inl}(m, o_1, \bar{a}, \bar{r}_m^1) \twoheadrightarrow_{\mathcal{E}} q_{21} & (m; n \text{ for } o_1) \\
q_{21} \;\dashv \mathsf{inl}(n, o_1, \bar{b}, \bar{r}_n^1) \twoheadrightarrow_{\mathcal{E}} q_{22} & \\
q_{22} \;\dashv \mathsf{inl}(n, o_2, \bar{b}, \bar{r}_n^2) \twoheadrightarrow_{\mathcal{E}} q_{23} & (n; m \text{ for } o_2) \\
q_{23} \;\dashv \mathsf{inl}(m, o_2, \bar{a}, \bar{r}_m^2) \twoheadrightarrow_{\mathcal{E}} q_3 & \\
q_3 \xrightarrow{assume(\bar{r}_m^1 \neq \bar{r}_m^2 \vee \bar{r}_n^2 \neq \bar{r}_n^1)}\!\!\!\twoheadrightarrow_{\mathcal{E}} q_{er} & (\text{Rv's agree}) \\
q_3 \twoheadrightarrow_{\mathcal{E}} q_4 & (\text{Loop}) \\
\cup_{m_i} \left\{ \begin{array}{l} q_4 \overset{\bar{a}:=\bar{*}}{\twoheadrightarrow}_{\mathcal{E}} q_{4i} \\ q_{4i} \;\dashv \mathsf{inl}(m_i, o_1, \bar{a}, \bar{r}) \twoheadrightarrow_{\mathcal{E}} q_{4i'} \\ q_{4i'} \;\dashv \mathsf{inl}(m_i, o_2, \bar{a}, \bar{s}) \twoheadrightarrow_{\mathcal{E}} q_{4i''} \\ q_{4i''} \xrightarrow{assume(\bar{r} \neq \bar{s})}\!\!\!\twoheadrightarrow_{\mathcal{E}} q_{er} \\ q_{4i''} \twoheadrightarrow_{\mathcal{E}} q_4 \end{array} \right. & (\text{Any } m_i)
\end{array} \right.
$$

and $Q_{\mathcal{E}}$ is the union of Q_o and all CFA nodes above, $s_{\mathcal{E}}$ is the union of s and all additional statements above, and $X_{\mathcal{E}} = X_o \cup \{o_1, o_2, \bar{a}, \bar{b}, \bar{r}_m^1, \bar{r}_n^2, \bar{r}_n^1, \bar{r}_m^2, nil, \bar{r}, \bar{s}\}$.

Fig. 5. One-shot reduction $\textsc{Reduce}_m^n$ emits automaton $\mathcal{A}(\varphi_m^n)$ defined above.

Appendix

A. *Transformation* $\textsc{Reduce}_m^n$

We now define $\textsc{Reduce}_m^n$. Below we let $assume(\bar{x} \neq \bar{y})$ mean the disjunction of inequality between corresponding vector elements.

Definition A.1 (Transformation). *For an input object CFA* $\mathcal{A}_o = \langle Q_o, [q_0^c, q_0^{\mathsf{init}}, q_0^{m_1}, \ldots, q_0^{m_k}], X_o, s, \twoheadrightarrow \rangle$*, the output of* $\textsc{Reduce}_m^n(\mathcal{A}_o, m(\bar{x}), n(\bar{y}))$ *is automaton* $\mathcal{A}(\varphi_m^n)$ *given in Figure 5.*

In Figure 5 node $q_0^{\mathcal{E}}$ is the initial node of the automaton. The transformation employs the implementation *source code* of the data structure CFA, given by $\twoheadrightarrow$. The key theorem below says that non-reachability of q_{er} entails that φ_m^n is a commutativity condition. The key features of $\mathcal{A}(\varphi_m^n)$ are:

1) *Reachable ADT states.* For any reachable object state σ, there will be some run of $\mathcal{A}(\varphi_m^n)$ that witnesses that state in a valuation θ at q_1. This is accomplished by edges from q_1 into the entry node of each possible m_i, first letting $Q_{\mathcal{E}}$ nondeterminsitically set arguments $\bar{x}$.
2) *Commutativity pre-condition.* From q_1, nondeterministic choices are made for the method arguments $m(\bar{a})$ and $n(\bar{b})$, and then candidate condition φ_m^n is assumed.
3) *Product program.* The edge to q_2 causes a run to clone o_1 to o_2, invoke $m(\bar{a}); n(\bar{b})$ on o_1 and invoke $n(\bar{b}); m(\bar{a})$ on o_2.
4) *Return values.* From q_3, there is an edge to q_{er} which is feasible if the return values disagree.
5) *Obs. Equivalence.* From q_4, for any possible method m_i, there is an edge with a statement $\bar{a} := \bar{*}$ to chose nondeterministic values, and then invoke $m_i(\bar{a})$ 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_{er}.

B. *Proof of Theorem V.1*

From Definition V.2, we have that R holds for the α-abstraction of post states, and then from Definition V.1 it follows that the return values agree. On the other hand, from Definition V.4 it follows that I_β holds for the β-abstraction of post states as well, and from Definition V.3 it follows that the (concrete) post states are observationally equivalent.

C. *Detailed version of Figure 3*

ADT	**Methods** $m(x_1), n(y_1)$	$\varphi^{n(y_1)}_{m(x_1)}$	**Exp.**	REDUCE^n_m **CPA**	REDUCE^n_m **Ult**	DAREDUCE^n_m mn**-differencing** **CPA**	DAREDUCE^n_m mn**-differencing** **Ult**
Memory	read $\bowtie$ write	$s1.x = y_1$	✓	1.4 ✓	0.7 ✓	1.3 (✓) + 1.3 (✓) + oe = 3.9 ✓	0.3 (✓) + 0.3 (✓) + oe = 1 ✓
Memory	read $\bowtie$ write	true	χ	1.4 χ	0.2 χ	1.3 (χ) + n/a(n/a) + oe = 1.3 χ	0.2 (χ) + n/a(n/a) + oe = 0.2 χ
Memory	write $\bowtie$ write	$y_1 = x_1$	✓	1.4 ✓	0.5 ✓	1.3 (✓) + 1.3 (✓) + oe = 3.9 ✓	0.2 (✓) + 0.3 (✓) + oe = 0.8 ✓
Memory	write $\bowtie$ write	true	χ	1.3 χ	0.3 χ	1.2 (✓) + 1.2 (χ) + oe = 2.4 χ	0.2 (✓) + 0.2 (χ) + oe = 0.4 χ
Memory	read $\bowtie$ read	true	✓	1.4 ✓	0.6 ✓	1.3 (✓) + 1.3 (✓) + oe = 3.9 ✓	0.3 (✓) + 0.3 (✓) + oe = 1 ✓
Accum.	decr $\bowtie$ isz	$s1.x > 1$	✓	1.5 ✓	2.2 ✓	1.3 (✓) + 1.4 (✓) + oe = 4 ✓	0.7 (✓) + 1.0 (✓) + oe = 2.6 ✓
Accum.	decr $\bowtie$ isz	true	χ	1.5 χ	0.7 χ	1.2 (χ) + n/a(n/a) + oe = 1.2 χ	0.6 (χ) + n/a(n/a) + oe = 0.6 χ
Accum.	decr $\bowtie$ incr	$s1.x > 1$	✓	1.5 ✓	1.3 ✓	1.3 (✓) + 1.5 (✓) + oe = 4.1 ✓	0.3 (✓) + 0.4 (✓) + oe = 1.7 ✓
Accum.	decr $\bowtie$ incr	true	✓	1.5 ✓	1.2 ✓	1.3 (✓) + 1.4 (✓) + oe = 4 ✓	0.3 (✓) + 0.3 (✓) + oe = 1.5 ✓
Accum.	incr $\bowtie$ isz	$s1.x > 1$	✓	1.5 ✓	3.3 ✓	1.4 (✓) + 1.4 (✓) + oe = 4.1 ✓	0.6 (✓) + 1.3 (✓) + oe = 2.9 ✓
Accum.	incr $\bowtie$ isz	true	χ	1.6 χ	0.7 χ	1.2 (χ) + n/a(n/a) + oe = 1.2 χ	0.6 (χ) + n/a(n/a) + oe = 0.6 χ
Accum.	incr $\bowtie$ incr	true	✓	1.4 ✓	1.5 ✓	1.4 (✓) + 1.4 (✓) + oe = 4.1 ✓	0.3 (✓) + 0.3 (✓) + oe = 1.6 ✓
Accum.	decr $\bowtie$ decr	true	✓	1.5 ✓	1.5 ✓	1.3 (✓) + 1.3 (✓) + oe = 3.9 ✓	0.3 (✓) + 0.4 (✓) + oe = 1.6 ✓
Accum.	decr $\bowtie$ decr	$s1.x > 1$	✓	1.5 ✓	2.6 ✓	1.3 (✓) + 1.4 (✓) + oe = 4 ✓	0.3 (✓) + 0.6 (✓) + oe = 1.9 ✓
Accum.	isz $\bowtie$ isz	true	✓	1.4 ✓	4.3 ✓	1.4 (✓) + 1.3 (✓) + oe = 4 ✓	1.8 (✓) + 0.7 (✓) + oe = 3.4 ✓
Counter	decr $\bowtie$ decr	true	χ	1.9 χ	1.5 χ	1.4 (✓) + 1.4 (✓) + oe = 4.2 ✓✗	1.2 (χ) + n/a(n/a) + oe = 1.2 χ
Counter	decr $\bowtie$ decr	$s1.x >= 2$	✓	1.5 ✓	13.0 ✓	1.3 (✓) + 1.4 (✓) + oe = 4.1 ✓	2.0 (✓) + 2.4 (✓) + oe = 5.9 ✓
Counter	decr $\bowtie$ incr	true	χ	1.6 χ	0.3 χ	1.4 (χ) + n/a(n/a) + oe = 1.4 χ	0.3 (χ) + n/a(n/a) + oe = 0.3 χ
Counter	decr $\bowtie$ incr	$s1.x >= 1$	✓	1.6 ✓	6.8 ✓	1.4 (✓) + 1.4 (✓) + oe = 4.2 ✓	1.5 (✓) + 1.0 (✓) + oe = 3.8 ✓
Counter	incr $\bowtie$ isz	true	χ	1.5 χ	0.8 χ	1.2 (χ) + n/a(n/a) + oe = 1.2 χ	0.7 (χ) + n/a(n/a) + oe = 0.7 χ
Counter	incr $\bowtie$ isz	$s1.x > 0$	✓	1.5 ✓	5.3 ✓	1.3 (✓) + 1.4 (✓) + oe = 4.1 ✓	0.6 (✓) + 0.5 (✓) + oe = 2.6 ✓
Counter	incr $\bowtie$ isz	$s1.x > 0$	χ	1.5 **?**	**TO ?**	1.4 (✓) + 1.5 (**?**) + oe = 4.4 χ	1.1 (✓) + 2.3 (✓) + oe = 6.9 χ
Counter	incr $\bowtie$ clear	true	χ	1.3 χ	0.4 χ	1.3 (✓) + 1.2 (χ) + oe = 2.5 χ	0.2 (✓) + 0.2 (χ) + oe = 0.4 χ

Extended version of Fig. 3. Results of applying CITYPROVER to the simple benchmarks. For each benchmark, we report the time to use both REDUCE^n_m and DAREDUCE^n_m, using either CPAchecker [26] or Ultimate [20] to solve the reachability tasks. Note that $s_1.x$ is the object field, and x_1, y_1 are m, n parameters, resp.

D. *Detailed version of Figure 4*

ADT	Methods $m(x_1), n(y_1)$	$\varphi^{n(y_1)}_{m(x_1)}$	Exp.	REDUCE^n_m Ult	DAREDUCE^n_m mn-**differencing** Ult
SimpleSet	n/a	n/a	✓	n/a	$\mathcal{A}_{\mathbf{B}}(I_\beta)$: ult - 16.9, cpa - 1.2 ✓
SimpleSet	isin ⋈ isin	true	✓	137.8 ✓	13.6 (✓) + 6.3 (✓) + oe = 36.7 ✓
SimpleSet	isin ⋈ add	$x_1 \neq y_1$	✓	84.8 ✓	11.4 (✓) + 8.9 (✓) + oe = 37.1 ✓
SimpleSet	isin ⋈ add	true	χ	2.4 χ	1.5 (χ) + n/a(n/a) + oe = 1.5 χ
SimpleSet	isin ⋈ clear	true	χ	2.6 χ	1.6 (χ) + n/a(n/a) + oe = 1.6 χ
SimpleSet	isin ⋈ clear	$x_1 \neq y_1$	χ	2.4 χ	1.6 (χ) + n/a(n/a) + oe = 1.6 χ
SimpleSet	isin ⋈ clear	$a_1 \neq x_1 \wedge s1.b \neq y_1$	χ	3.1 χ	1.4 (χ) + n/a(n/a) + oe = 1.4 χ
SimpleSet	isin ⋈ clear	$a_1 \neq x_1 \wedge s1.b \neq x_1$	✓	14.0 ✓	0.9 (✓) + 1.5 (✓) + oe = 19.3 ✓
SimpleSet	isin ⋈ getsize	true	✓	41.3 ✓	3.4 (✓) + 4.1 (✓) + oe = 24.2 ✓
ArrayStack	n/a	n/a	✓	n/a	$\mathcal{A}_{\mathbf{B}}(I_\beta)$: ult - 35.4, cpa - 66.6 ✓
ArrayStack	push ⋈ pop	$a_1[top_1] = x_1 \wedge top_1 > 1 \wedge top_1 < 5 - 1$	✓	**MO** –	34.6 (✓) + 25.5 (✓) + oe = 95.5 ✓
ArrayStack	push ⋈ pop	true	χ	2.2 χ	2.0 (χ) + n/a(n/a) + oe = 2.0 χ
ArrayStack	push ⋈ push	true	χ	7.6 χ	17.0 (χ) + n/a(n/a) + oe = 17.0 χ
ArrayStack	push ⋈ push	$top_1 < 3$	χ	3.9 χ	229.3 (✓) + 1.4 (χ) + oe = 230.7 χ
ArrayStack	push ⋈ push	$x_1 = y_1$	χ	19.8 χ	17.3 (χ) + n/a(n/a) + oe = 17.3 χ
ArrayStack	push ⋈ push	$x_1 = y_1 \wedge top_1 < 3$	✓	**MO** –	3.6 (✓) + 116.3 (✓) + oe = 155.1 ✓
ArrayStack	pop ⋈ pop	$top_1 = -1$	✓	**TO** –	0.7 (✓) + 2.0 (✓) + oe = 38.0 ✓
ArrayStack	pop ⋈ pop	true	χ	2.1 χ	1.4 (χ) + n/a(n/a) + oe = 1.4 χ
Queue	n/a	n/a	✓	n/a	$\mathcal{A}_{\mathbf{B}}(I_\beta)$: ult - 132.9, cpa - 109.1 ✓
Queue	enq ⋈ enq	true	χ	39.8 χ	35.0 (χ) + n/a(n/a) + oe = 35.0 χ
Queue	deq ⋈ deq	true	χ	3.5 χ	3.1 (χ) + n/a(n/a) + oe = 3.1 χ
Queue	deq ⋈ deq	$size_1 = 0$	✓	**TO** –	2.2 (✓) + 63.1 (✓) + oe = 174.4 ✓
Queue	enq ⋈ enq	true	χ	27.8 χ	23.3 (χ) + n/a(n/a) + oe = 23.3 χ
Queue	enq ⋈ enq	$x_1 = y_1$	χ	63.8 χ	22.6 (χ) + n/a(n/a) + oe = 22.6 χ
Queue	isempty ⋈ isempty	true	✓	**TO** –	6.6 (✓) + **TO** (**?**) + oe = **TO** –
Queue	enq ⋈ deq	$size_1 = 1 \wedge x_1 = a_1[front_1]$	✓	**TO** –	2.3 (✓) + 360.6 (✓) + oe = 472.0 ✓
Queue	enq ⋈ deq	true	χ	**MO** –	8.4 (χ) + n/a(n/a) + oe = 8.4 χ
Queue	enq ⋈ isempty	$size_1 > 0$	✓	**MO** –	6.1 (✓) + **TO** (**?**) + oe = **TO** –
Queue	enq ⋈ isempty	$true$	χ	**MO** –	7.4 (χ) + n/a(n/a) + oe = 7.4 χ
Queue	deq ⋈ isempty	$size_1 = 0$	✓	**MO** –	2.1 (✓) + 0.8 (✓) + oe = 135.8 ✓
HashTable	n/a	n/a	✓	n/a	$\mathcal{A}_{\mathbf{B}}(I_\beta)$: ult - **TO**, cpa - 7.1 ✓
HashTable	put ⋈ put	${}^1\varphi^{put}_{put}$	χ	262.5 χ	97.5 (χ) + n/a(n/a) + oe = 97.5 χ
HashTable	put ⋈ put	${}^2\varphi^{put}_{put}$	χ	202.7 χ	136.9 (χ) + n/a(n/a) + oe = 136.9 χ
HashTable	put ⋈ put	${}^3\varphi^{put}_{put}$	✓	**TO** –	**TO** (**?**) + n/a(n/a) + oe = **TO** –
Hashtable	put ⋈ put	${}^3\varphi^{put}_{put}$	✓	566.5 ✓	1.5 (✓) + 288.9 (✓) + oe = 297.5 ✓
HashTable	put ⋈ put	true	χ	**TO** –	102.3 (χ) + n/a(n/a) + oe = 102.3 χ
HashTable	get ⋈ get	$o_1.keys = 0$	✓	**TO** –	**TO** (**?**) + n/a(n/a) + oe = **TO** –
HashTable	get ⋈ get	true	✓	**TO** –	**TO** (**?**) + n/a(n/a) + oe = **TO** –
HashTable	get ⋈ get	true	✓	50.0 ✓	2.0 (✓) + 47.7 (✓) + oe = 56.8 ✓
HashTable	get ⋈ put	$x_1 \neq y_1$	✓	**TO** –	**TO** (**?**) + n/a(n/a) + oe = **TO** –
HashTable	get ⋈ put	true	χ	1.3 χ	0.9 (χ) + n/a(n/a) + oe = 0.9 χ

$${}^1\varphi^{pop}_{push} \equiv \quad o_1.a[o_1.top] = x_1 \wedge o_1.top > 1 \wedge o_1.top < MAX \qquad {}^1\varphi^{put}_{put} \equiv x_1 \neq y_1$$
$${}^2\varphi^{put}_{put} \equiv \quad x_1 \neq y_1 \wedge o_1.table[x_1\%CAP].key = -1 \wedge o_1.table[y_1\%CAP].key = -1$$
$${}^3\varphi^{put}_{put} \equiv \quad x_1 \neq y_1 \wedge x_1\%CAP \neq y_1\%CAP \wedge o_1.table[x_1\%CAP].key = -1 \wedge o_1.table[y_1\%CAP].key = -1$$

Extended version of Fig. 4. Some commutativity conditions are listed below the table. Our implementation actually subdivides Phase **A** into two parts: first one in which we prove return value agreement (*i.e.* R_α) and then one in which we prove that I_β is implied. Hence, in the DAREDUCE^n_m column, we report the some of these two sub-phases plus the time for Phase **B**. For each benchmark, we re-ran the Phase **B**, even though all could have shared one run. For each ADT above, we report an example of the time to prove $\mathcal{A}_{\mathbf{B}}(I_\beta)$ with both Ultimate and CPAchecker. We mostly used Ultimate but, in some of the Queue or Hashtable cases, it did not perform well and we instead tried CPAchecker. Those cases are denoted in blue.

Results. We show that CITYPROVER is tractable at proving commutativity conditions. Moreover, DAREDUCE^n_m improves over REDUCE^n_m: in most cases it is faster, sometimes by as much as 2× or 3×. In **7** cases, DAREDUCE^n_m is able to generate an answer, while REDUCE^n_m suffers from a timeout/memout. For Hashtable, Ultimate timed out on Phase **B**. 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.

E. Benchmark Sources

Memory

```
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; }
```

Counter

```
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); }
```

Accumulator

```
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); }
```

Array Queue

```
#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; \
  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)
```

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)
```

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)->table[slot].key == -1) {                 \
      (st)->table[slot].key = k;                              \
      (st)->table[slot].value = v;                            \
```

```
      (st)->keys++;                              \
      rv= 1; }                                   \
    else if ((st)->table[slot].key == k) {       \
      (st)->table[slot].value = v;               \
      rv = 1; }                                  \
    else if ((st)->table[slot].key != k) {       \
      rv = -1; }                                 \
    else { rv = -1; } }

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

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

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

Simple Set

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

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

#define o_new(st) { \
  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; }
```