Commutativity Condition Refinement

Kshitij Bansal
New York University

Eric Koskinen
IBM TJ Watson Research Center

Omer Tripp
IBM TJ Watson Research Center

***Abstract*—We present a technique for automatically generating commutativity conditions from (abstract-level) data-structure specifications. We observe that one can pose the commutativity question in a way that does not introduce additional quantifiers, via a mechanized lifting of a (potentially partial) specification to an equivalent total specification. We then describe an algorithm that iteratively refines an under-approximation of the commutativity (and non-commutativity) condition for two data-structure methods m and n into an increasingly precise version. Our algorithm terminates if/when the entire state space has been considered, and can be aborted at any time to obtain a partial, sound commutativity condition. We prove soundness, relative completeness, describe how to obtain input predicates, and discuss heuristics to improve qualitative aspects of our algorithm's output.**

We have implemented our technique in a tool called SERVOIS, which uses an SMT solver. We show that we can automatically generate symbolic commutativity conditions for a range of data structures including Set, HashTable, Accumulator, Counter, and Stack. This work has the potential to impact a variety of contexts, in particular, multicore software where it has been realized that commutativity is at the heart of increased concurrency.

I. INTRODUCTION

Recent decades have seen the development of a variety of paradigms to exploit the opportunity for concurrency in multicore architectures, including parallelizing compilers [22], [26], speculative execution (*e.g.* transactional memory [11]), futures, etc. It has been shown, across all of these domains, that understanding the commutativity of concurrent data-structure operations provides a key avenue to improved performance [6] as well as ease of verification [15], [14].

Intuitively, linearizable data-structure operations that commute can be executed concurrently because their effects don't interfere with each other in a harmful way. When using a (linearizable) HashTable, for example, knowledge that `put(x,'a')` commutes with `get(y)` provided that $\texttt{x} \neq \texttt{y}$ enables significant parallelization opportunities as both can be performed concurrently.

Commutativity conditions are an important part of the concurrent programming toolkit, but they are tedious to specify manually and require nontrivial and error-prone reasoning. Recent advances have been made on verification of commutativity conditions [13], as well as attempts at synthesis based on random interpretation [3] or dynamic profiling [24]. Thus far, however, generating commutativity conditions automatically has been largely overlooked.

Koskinen was previously supported in part by NSF CCF Award #1421126 at New York University.

We present the first known technique for automatic refinement of commutativity conditions. We build on a vast body of existing research, extending over the last five decades, on specification and representation of abstract data types (ADTs) in terms of logical $(Pre_m, Post_m)$ specifications [12], [9], [10], [4], [20], [17] of methods $m, n, \ldots$.

Our technique generates a logical commutativity condition φ_m^n for each pair m, n that specifies when method m commutes with method n. We first observe that one can pose the commutativity question in a way that does not introduce additional quantifiers via a mechanized lifting of a (potentially partial) method specification to an equivalent total specification.

Next, the predicate vocabulary for expressing the condition φ_m^n is populated automatically by deriving atoms, used to describe the pre/post footprint of operations, from the transition system's specification. Intuitively, since these atoms suffice to capture the effect of an operation, they may also capture the conditions under which the effects of two operations do, or do not, conflict.

Based on the predicate vocabulary, our algorithm iteratively relaxes an under-approximation of the commutativity (and non-commutativity) condition, starting from false, into an increasingly precise version. At each step, we conjunctively subdivide the state space into regions, searching for areas where m and n commute and where they don't. Throughout this recursive process, we accumulate the commutativity condition as a growing disjunction of these regions.

We have proved that the algorithm is sound, and can also be aborted at any time to obtain a partial commutativity condition. This is often useful in practice, as partial commutativity conditions typically outperform alternate implementations that don't use commutativity. We also show that if the algorithm terminates without being aborted, it results in a complete specification and provide useful conditions under which termination is guaranteed.

We have implemented our approach as the SERVOIS* tool, which is able to generate commutativity conditions for various popular data structures, including Set, Counter, HashTable and Stack. The conditions typically combine multiple theories, such as sets, integers, arrays, etc. As such, we have implemented our technique atop the SMT solver CVC4 [5].

This paper makes the following principal contributions:

- Techniques to lift partial transition specifications and iteratively refine commutativity conditions (Sec. IV and V).

*http://cs.nyu.edu/~kshitij/projects/servois/

- Proof of soundness and relative completeness (Sec. VI).
- Automated extraction of base formula terms (Sec. VII).
- SMT-based implementation (Sec. VII).
- Demonstrated efficacy for several key data structures including Set, HashTable, Accumulator, Counter, and Stack (Sec. VII).

Other related work. Both Aleen and Clark [3] and Tripp *et al.* [25] identify sequences of actions that commute (via random interpretation and dynamic analysis, respectively) but neither technique yields an explicit commutativity condition. Kulkarni et al. [16] point out that varying degrees of commutativity specification precision are useful. Kim and Rinard [13] use Jahob to verify manually specified commutativity conditions of several different linked data structures. Data-structure commutativity specifications are also found in dynamic analysis techniques [8].

More distantly related is the work on synthesizing program implementations, such as CEGIS [23], and synchronization synthesis [28], [27]. Aderhold [2] describes a method to synthesize, or extract, induction axioms from programs with indirect recursive calls (*e.g.* algorithms on data structures). Leino [18] explains the process by which verification conditions are generated for the object-oriented language Spec#. As with our encoding scheme, Leino's conditions target SMT solvers.

II. Overview

Motivation. Specifying commutativity conditions is nontrivial. Not only is this task burdensome since it has to be done pairwise for all methods (*i.e.* quadratic), but even if there are few operations, commutativity conditions are often subtle.

As an illustration, consider the Set ADT, whose state consists of a single variable, S, that stores an unordered collection of unique elements. We focus on two operations:

- `contains(`x`)/bool`, which performs a side-effect-free check whether the element x is in S; and
- `add(`y`)/bool`, which adds y to S if it is not already in there and returns true, or otherwise returns false.

`add` and `contains` clearly commute if they refer to different elements in the set. There is, however, another case that is less obvious: `add` and `contains` commute if they refer to the same element e, as long as in the prestate $e \in S$. In this case, in both orders of execution `add` and `contains` leave the set unmodified and return false and true, respectively.

Capturing precise conditions such as these by hand, and doing so for many pairs of operations, is tedious, error prone, and benefits from automation.

Iterative Refinement Algorithm. The algorithm we describe in this paper automatically produces a precise logical formula φ that captures this commutativity condition, *i.e.* the disjunction of the two cases above: $\varphi \equiv x \neq y \vee (x = y \wedge x \in S)$. The algorithm also generates the conditions under which the methods *do not* commute: $\tilde{\varphi} \equiv x = y \wedge x \notin S$. This is precise since φ is the negation of $\tilde{\varphi}$.

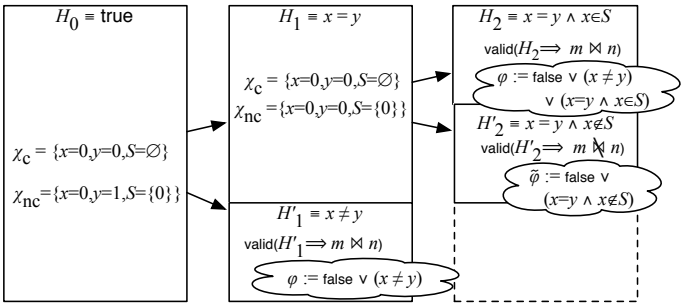


Figure 1. An example of how our technique generates commutativity conditions for methods `add` and `contains` operating on a Set. Each subsequent panel depicts a partitioning of the state space. The counterexamples χ_c, χ_{nc} give values for the arguments x, y and the current state of the set S.

The main thrust of the algorithm is to recursively subdivide the state space via predicates until, at the base case, regions are found that are either entirely commutative or else entirely non-commutative. As in the example above, the conditions we incrementally generate are denoted φ and $\tilde{\varphi}$, respectively.

We illustrate how our algorithm proceeds on the running example in Figure 1. We denote by H the logical formula that describes the current state space at a given recursive call. As expected, we begin with $H_0 =$ true, $\varphi =$ false, and $\tilde{\varphi} =$ false. There are essentially three cases for a given H: (i) H describes a precondition for m and n in which m and n *always* commute; (ii) H describes a precondition for m and n in which m and n *never* commute; or (iii) neither of the above. The latter case drives the recursive algorithm to subdivide the region by choosing a new predicate.

In Section VI, we state the formal guarantees of the algorithm. We have proved that it is sound, *i.e.* it produces sound commutativity conditions, even if aborted. Soundness is guaranteed even if the ADT description involves undecidable theories. We further show that termination implies completeness, and specify broad conditions that imply termination (*i.e.* relative completeness).

Challenges. While the algorithm, as outlined so far, executes a relatively standard refinement loop, there are interesting challenges that are implicit in its description.

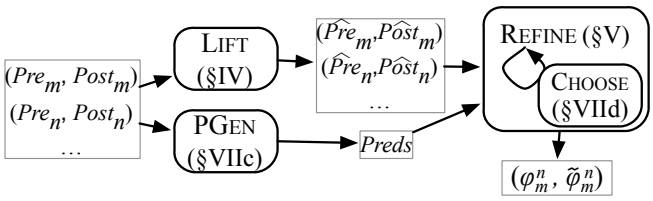
The first challenge pertains to the complexity and decidability of the validity queries discharged to the supporting SMT solver. In particular, if the query contains alternating quantification (specifically the $\forall\exists$ pattern), then decidability is lost. In our technique, we *avoid introducing additional quantification by construction*. Hence, if the underlying theories are decidable, then the queries we pose to the SMT solver are guaranteed to also be decidable. We achieve this mechanically by symbolically completing the (potentially) partial transition system into a total system through the addition of a new Err state. Err becomes the image of states without successors. This modified encoding, formalized in Section IV, ensures that universal quantification suffices without the need to introduce additional quantification.

Next, there is the critical question of which predicates

to range over during the iterative refinement process. If the predicate vocabulary is not sufficiently expressive, then the algorithm would not be able to converge on precise commutativity and non-commutativity conditions. In Section VII, we provide a mechanized solution to this problem, whereby the predicate vocabulary is populated with the atoms that occur in the transition relations' *Pre* and *Post* formulas. As we demonstrate in Section VII, this strategy works extremely well in practice. An intuitive explanation is that the *Pre* and *Post* formulas suffice to express the footprint of an operation, and so the atoms comprising them are an effective vocabulary to express when operations do, or do not, interfere.

A final challenge, having fixed the predicate vocabulary, is to prioritize the predicates. This choice essentially drives the iterative refinement loop, and so it controls not only the algorithm's performance, but also the quality (or conciseness) of the resulting conditions. Our choice of next predicate p is governed by two requirements. First, for progress, $p/\neg p$ must eliminate the counterexamples to commutativity/non-commutativity due to the last iteration (where previously selected predicates ensure the same for their respective counterexamples). This may still leave multiple choices, and we propose two heuristics with different trade-offs to break ties. We discuss this in more detail in Section VII.

In summary, the following diagram illustrates the overall flow of our automated algorithm, including the components discussed above to generate useful predicates (PGEN), complete the transition relation (LIFT), and choose the next predicate along the refinement process (CHOOSE box inside REFINE).



In the diagram, we denote the total version of the $Pre_m/Post_m$ specifications as $\widehat{Pre_m}/\widehat{Post_m}$, the generated commutativity condition as φ_m^n and the non-commutativity condition $\tilde{\varphi}_m^n$.

III. PRELIMINARIES

States and actions. We will work with a state space denoted Σ, with decidable equality and a set of *actions* A. For each $\alpha \in A$, we have a transition function $(\!|\alpha|\!) : \Sigma \rightharpoonup \Sigma$. We denote a single transition as $\sigma \xrightarrow{\alpha} \sigma'$ and we assume that each such action arc completes in finite time. Let $\mathfrak{T} \equiv (\Sigma, A, (\!|\bullet|\!))$.

Definition III.1 (Action commutativity [8])**.** *We say that two actions α_1 and α_2 commute, denoted $\alpha_1 \bowtie \alpha_2$, provided that $(\!|\alpha_1|\!) \circ (\!|\alpha_2|\!) = (\!|\alpha_2|\!) \circ (\!|\alpha_1|\!)$.*

Note that $\bowtie$ is with respect to $\mathfrak{T} = (\Sigma, A, (\!|\bullet|\!))$.
Our formalism, implementation, and evaluation all extend to a more fine-grained notion of commutativity: an asymmetric version called left-movers and right-movers [19] where a method commutes in one direction and not the other. For ease of presentation, the formal detail in the body of this paper discusses only commutativity, but a discussion of how our technique generalizes can be found in Appendix A. Also, in our evaluation (Section VII) we show left-/right-mover conditions that were generated by our implementation.

Methods. An action $\alpha \in A$ is of the form $m(\bar{x})/\bar{r}$ where m, $\bar{x}$ and $\bar{r}$ are called a *method*, *arguments* and *return values* respectively. For actions corresponding to a method n, we use $\bar{y}$ for arguments and $\bar{s}$ for return values. In our context, the set of methods will be finite, inducing a finite partitioning of A. We refer to an action, say $m(\bar{x})/\bar{r}$, as *corresponding* to method m. The set of actions corresponding to a specific method m, denoted A_m, might be infinite as the arguments and return values may be infinite.

Definition III.2 (Method commutativity)**.** *For m and n,*

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

Above we have quantified over all actions corresponding to m and n. That is, the quantification $\forall \bar{x}\bar{r}$ means $\forall m(\bar{x})/\bar{r} \in A_m$.

Abstract specifications. We describe the actions of a (potentially partial) method m symbolically, as a pre-condition Pre_m and post-condition $Post_m$. Pre-conditions are logical formulae over method arguments and the initial state, and post-conditions are over method arguments, and return values, initial state and final state:

$$\begin{aligned} [\![Pre_m]\!] &: \bar{x} \to \Sigma \to \mathsf{Prop} \\ [\![Post_m]\!] &: \bar{x} \to \bar{r} \to \Sigma \to \Sigma \to \mathsf{Prop} \end{aligned}$$

If we have pre-/post-conditions $(Pre_m, Post_m)$ for every method m, then we define a transition system $\mathfrak{T} = (\Sigma, A, (\!|\bullet|\!))$ such that $\sigma \xrightarrow{m(\bar{x})/\bar{r}} \sigma'$ *iff* $[\![Pre_m]\!]\ \bar{x}\ \sigma$ and $[\![Post_m]\!]\ \bar{x}\ \bar{r}\ \sigma\ \sigma'$.

Since our approach works on deterministic transition systems, we have implemented a check (discussed in Section VII) that ensures the input transition system is deterministic. Deterministic specifications were sufficient to model the Set, HashTable, Accumulator, Counter, and Stack, which is unsurprising given the inherent difficulty of creating efficient concurrent implementations of nondeterministic operations, whose effects are hard to characterize. We believe it may be possible to reduce nondeterministic data-structure methods to deterministic ones through symbolic partial determinization [1], [7], but we leave this as future work.

Logical commutativity formulae. We will work with, and generate, a commutativity condition for methods m and n as logical formulae over initial states and the arguments/return values of the methods. We denote a logical commutativity formula as φ and assume a decidable interpretation of formulae: $[\![\varphi]\!] : (\sigma, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \to \mathbb{B}$. (We tuple the arguments for brevity.) The first argument is the initial state. Commutativity *post-* and *mid*-conditions can also be written [13] but here, for simplicity, we focus only *pre*-conditions. Throughout this paper, we may write $[\![\varphi]\!]$ as, simply, φ when it is clear from context that φ is meant to be interpreted.

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

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

Trouble with existential quantification. Notice that proving commutativity of methods m and n via Definition III.2 requires showing equivalence between different compositions of potentially non-surjective functions. That is, $(\!|\alpha_1|\!) \circ (\!|\alpha_2|\!) = (\!|\alpha_2|\!) \circ (\!|\alpha_1|\!)$ if and only if:

$$\forall \sigma_0\ \sigma_1\ \sigma_{12}.\ (\!|\alpha_1|\!)\sigma_0 = \sigma_1 \wedge (\!|\alpha_2|\!)\sigma_1 = \sigma_{12}$$
$$\Rightarrow\ \exists\ \sigma_3.\ (\!|\alpha_2|\!)\sigma_0 = \sigma_3 \wedge (\!|\alpha_1|\!)\sigma_3 = \sigma_{12}.$$
(*and a similar case for the other direction*)

Even when the transition relation can be expressed in a decidable theory, because of $\forall\exists$ quantifier alternation (which is undecidable in general), any procedure requiring such a check would be incomplete. SMT solvers are particularly poor at handling such constraints. In the next section, we describe a mechanized transformation of the transition system that allows us to encode commutativity as a universal formula with no quantifier alternation.

IV. Lifting for Commutativity

In this section, we describe a simple transformation on transition systems to a lifted domain, and a definition of commutativity in the lifted domain $m\ \hat{\bowtie}\ n$ that is equivalent to Definition III.2. This new definition requires only *universal* quantification, and as such, is better suited to automation in SMT-based algorithms (*e.g.* Section V).

Definition IV.1 (Lifted transition function)**.** *For a given* $\mathfrak{T} = (\Sigma, A, (\!|\bullet|\!))$*, we* Lift $\mathfrak{T}$ *to transition system* $\hat{\mathfrak{T}} = (\hat{\Sigma}, A, [\![\bullet]\!])$ *where* $\hat{\Sigma} = \Sigma \cup \{Err\}$*,* $Err \notin \Sigma$*, and* $[\![\alpha]\!] : \hat{\Sigma} \to \hat{\Sigma}$ *is:*

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

Intuitively, $[\![\alpha]\!]$ wraps $(\!|\alpha|\!)$ so that Err loops back to Err, and the (potentially partial) $(\!|\alpha|\!)$ is made to be total, by mapping elements to Err when they are undefined in $(\!|\alpha|\!)$. Note that it is not necessary to lift the actions (or, indeed, the methods), but only the states and transition function. Once lifted, for a given state $\hat{\sigma}_0$, the question of *some* successor state becomes equivalent to *all* successor states because there is exactly one successor state.

Abstraction. Our prior pre-/post-conditions $(\mathit{Pre}_m, \mathit{Post}_m)$ were suitable for specifications of potentially partial transition systems. We now describe a translation of to a new $(\widehat{\mathit{Pre}}_m, \widehat{\mathit{Post}}_m)$ that induces a corresponding lifted transition system that (i) is surjective and (ii) remains deterministic. These lifted specifications have types over lifted state spaces:

$$[\![\widehat{\mathit{Pre}}_m]\!] :\ \bar{x} \to \hat{\Sigma} \to \mathsf{Prop}$$
$$[\![\widehat{\mathit{Post}}_m]\!] :\ \bar{x} \to \bar{r} \to \hat{\Sigma} \to \hat{\Sigma} \to \mathsf{Prop}$$

Our tool performs lifting automatically via translation from a $(\mathit{Pre}_m, \mathit{Post}_m)$ specification to:

$$\widehat{\mathit{Pre}}_m(\bar{x}, \hat{\sigma}) \equiv \mathsf{true}$$
$$\widehat{\mathit{Post}}_m(\bar{x}, \bar{r}, \hat{\sigma}, \hat{\sigma}') \equiv$$
$$\bigvee \begin{cases} \hat{\sigma} = Err \wedge \hat{\sigma}' = Err \\ \hat{\sigma} \neq Err \wedge \mathit{Pre}_m(\bar{x}, \hat{\sigma}) \wedge \hat{\sigma}' \neq Err \wedge \mathit{Post}_m(\bar{x}, \bar{r}, \hat{\sigma}, \hat{\sigma}') \\ \hat{\sigma} \neq Err \wedge \neg \mathit{Pre}_m(\bar{x}, \hat{\sigma}) \wedge \hat{\sigma}' = Err \end{cases}$$

(We abuse notation, giving $\hat{\sigma}$ as an argument to Pre_m, etc.) It is easy to see that the lifted transition system induced by this translation $(\hat{\Sigma}, [\![\bullet]\!])$ is of the form given in Definition IV.1. In the Appendix, we show how our tool transforms a counter specification (Apx. B) into an equivalent lifted version (Apx. C). Notice that this specification is now a total transition system: $\widehat{\mathit{Pre}}_m = \mathsf{true}$.

A. Posing commutativity with only universal qualifiers

With the above lifting, we can pose commutativity as a universal question (modulo quantifiers in the underlying data-structure theory). We will use the notation $\hat{\bowtie}$ to mean $\bowtie$ but over the lifted transition system $\hat{\mathfrak{T}}$. Notice, now that since $\hat{\bowtie}$ is over *total* and *determinsitic* transition functions, $\alpha_1\ \hat{\bowtie}\ \alpha_2$ it is equivalent to:

$$\forall \hat{\sigma}_0.\ \hat{\sigma}_0 \neq Err\ \Rightarrow\ [\![\alpha_2]\!]\ [\![\alpha_1]\!]\ \hat{\sigma}_0 = [\![\alpha_1]\!]\ [\![\alpha_2]\!]\ \hat{\sigma}_0$$

The equivalence above is state equality. Importantly, this is a universally quantified formula which translates to a ground satisfiability check in an SMT solver (modulo the theories used to model the data-structure).

Theorem IV.1 (Lifted commutativity equivalence)**.** *For methods* m *and* n*,* $m\ \bowtie\ n$ *if and only if* $m\ \hat{\bowtie}\ n$*.*

Proof. Both directions are proved with classical reasoning, functional extensionality and case analysis on whether, for a given state σ, $(\!|\alpha|\!)\sigma$ is defined or undefined. □

B. Checking whether $(H_m^n \Rightarrow m\ \bowtie\ n)$ or $(H_m^n \Rightarrow m\ \hat{\not\bowtie}\ n)$.

For a logical formula $H_m^n(\hat{\sigma}, \bar{x}, \bar{y}, \bar{r}, \bar{s})$, given the lifting described above, we can check whether H_m^n specifies conditions under which $m\ \bowtie\ n$ via an SMT query that does not introduce quantifier alternation. For brevity, we define the following syntactic sugar:

$$\mathsf{valid}(H_m^n\ \Rightarrow\ m\ \hat{\bowtie}\ n) \equiv \mathsf{valid}\left(\begin{array}{l} \forall \hat{\sigma}_0\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}. \\ \quad H_m^n(\hat{\sigma}_0, \bar{x}, \bar{y}, \bar{r}, \bar{s})\ \Rightarrow \\ \quad \hat{\sigma}_0 \neq Err\ \Rightarrow \\ \quad m(\bar{x})/\bar{r}\ \hat{\bowtie}\ n(\bar{y})/\bar{s} \end{array}\right)$$

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

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

$$\begin{aligned} \alpha_1\ \hat{\not\bowtie}\ \alpha_2 &\equiv \forall \hat{\sigma}_0.\ \hat{\sigma}_0 \neq Err\ \Rightarrow \\ &\quad\ [\![\alpha_2]\!]\ [\![\alpha_1]\!]\ \hat{\sigma}_0 \neq [\![\alpha_1]\!]\ [\![\alpha_2]\!]\ \hat{\sigma}_0 \\ m\ \hat{\not\bowtie}\ n &\equiv \forall \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\ m(\bar{x})/\bar{r}\ \hat{\not\bowtie}\ n(\bar{y})/\bar{s} \end{aligned}$$

```
1  REFINE^m_n(H, P) {
2      if valid(H ⇒ m ⋈̂ n) then
3        φ := φ ∨ H;
4      else if valid(H ⇒ m ⋈̸̂ n) then
5        φ̃ := φ̃ ∨ H;
6      else
7        let χc, χnc = counterex. to ⋈̂ and ⋈̸̂ (resp.) in
8        let p = CHOOSE(H, P, χc, χnc) in
9          REFINE^m_n(H ∧ p, P \ {p});
10         REFINE^m_n(H ∧ ¬p, P \ {p});
11 }
12 main {
13     φ := false;  φ̃ := false;
14     try { REFINE^m_n(true, P); }
15     catch (InterruptedException e) { skip; }
16     return(φ, φ̃);
17 }
```

Figure 2. The refinement algorithm for generating a commutativity condition φ and non-commutativity condition $\tilde{\varphi}$ for two methods m and n.

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

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

V. Commutativity Condition Refinement

We now present an iterative refinement strategy that, when given a lifted abstract transition system, generates commutativity and non-commutativity conditions. In Section VI we discuss soundness and relative completeness.

The refinement algorithm symbolically searches the state space for regions where the operations commute (or do not commute) in a conjunctive manner, adding on one predicate at a time. We add each subregion H (described conjunctively) in which commutativity always holds to a growing disjunctive description of the commutativity condition φ, and each subregion H in which commutativity never holds to a growing disjunctive description of the non-commutativity condition $\tilde{\varphi}$.

The algorithm in Figure 2 begins by setting $\varphi = \mathsf{false}$ and $\tilde{\varphi} = \mathsf{false}$. REFINE begins a symbolic binary search through the state space H, beginning with the entire state: $H = \mathsf{true}$. It also may use a collection of predicates $\mathcal{P}$ (discussed later).

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

We now need to subdivide H into two regions. This is accomplished by selecting a new predicate p via the CHOOSE method. For now, let the method CHOOSE and the choice of predicate vocabulary $\mathcal{P}$ be parametric. REFINE is sound regardless of the behavior of CHOOSE, in Section VI we give the conditions on CHOOSE that ensure relative completeness, and in Section VII, we discuss our particular strategy. Regardless of what p is returned by CHOOSE, two recursive calls are made to REFINE, one with argument $H \wedge p$, and the other with argument $H \wedge \neg p$.

The refinement algorithm generates commutativity conditions that are in disjunctive normal form: $\varphi ::= \varphi \vee \varphi \mid (Y)$ where $Y ::= Y \wedge Y \mid p$ and p is from a language of predicates. Hence, any (finite) logical formula can be represented. This logical language is more expressive than previous commutativity logics that, because they were designed for run-time purposes, were restricted to conjunctions of inequalities [16] and boolean combinations of predicates over finite domains [8].

VI. Soundness and Relative Completeness

The following theorem shows that φ is a sound approximation of when $m \hat{\bowtie} n$ always holds (and similar for $\tilde{\varphi}$).

Theorem VI.1 (Soundness)**.** *At each iteration of* REFINE^m_n, $\varphi \Rightarrow m \hat{\bowtie} n$, *and* $\tilde{\varphi} \Rightarrow m \not\hat{\bowtie} n$.

Proof. By induction. Initially, false is a suitable condition for when commutativity holds, and false is a suitable condition under which commutativity does not hold. At each iteration, φ or $\tilde{\varphi}$ may be updated (not both, but for soundness this does not matter). Consider φ. It must also be the case that $(\varphi \vee H) \Rightarrow m \hat{\bowtie} n$ because we know that $\varphi \Rightarrow m \hat{\bowtie} n$ (from the previous iteration) and that $H \Rightarrow m \hat{\bowtie} n$ (from the valid check on Line 2). Similar reasoning for $\tilde{\varphi}$. □

Soundness holds regardless of what CHOOSE returns (not surprising since updates to φ and $\tilde{\varphi}$ are guarded by validity checks) and even when the theories used to model the underlying data-structure are incomplete. Next we show that termination implies completeness (Lemma VI.2) and give some conditions under which termination, and thus completeness, is ensured (Theorem VI.3).

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

Proof. The recursive calls of the REFINE algorithm induces a *binary* tree T, where nodes are labeled by the conjunction of predicates. If REFINE terminates, then T is finite, and each node is labeled with a finite conjunction $p_0 \wedge ... \wedge p_n$.

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

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

Theorem VI.3 (Sufficient Conditions for Termination)**.** *Provided that*

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

then REFINE_n^m *terminates.*

Proof. By contradiction. As in the proof for Lemma VI.2, we represent the algorithm's execution as a binary tree T, induced by the recursive REFINE calls, whose nodes are labeled by the conjunction of predicates (lines 9 and 10 in Algorithm 2). Assume there exists an infinite path along T, and let their respective labels be $\pi = p_0, p_0 \wedge p_1, p_0 \wedge p_1 \wedge p_2, ...$

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

By the above claim, at least one of the predicates ψ_i is not contained in any finite prefix of π. This contradicts the (fairness) assumption, whereby any predicate $p \in \mathcal{P}$ is chosen after finitely many CHOOSE invocations (provided the algorithm hasn't terminated). □

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

VII. IMPLEMENTATION & EVALUATION

We have implemented the algorithm in Section V, along with the other parts of the system (illustrated in Section II) in our tool SERVOIS[†]. We use CVC4 [5] as the backend solver. We evaluated the algorithm on various abstract data structures, and discuss some challenges, choices and optimizations in the following.

a) Encoding the transition system: We use an input specification language building on YAML (which has parsers and printers in all common programming languages) with SMTLIB as the logical language. It is human-readable as well as can be easily auto-generated allowing to easily fit in other toolchains [12], [9], [10], [4], [20], [17]. See Appendix B for the Counter ADT specification which was derived from the *Pre* and *Post* conditions used in earlier work [13]. The novelty of our work is that we automatically generate commutativity conditions for any implementation that respects these contracts.

The states (state) of a transition system describing an ADT are encoded as list of variables (each as a name, type pair), and each method (methods) specification requires a list of argument types (args), return type (return), and *Pre* (requires) and *Post* (ensures) conditions. The full specifications for Counter, Accumulator, Set, HashTable, and Stack we used can be found in the Appendix. We used the quantifier-free integer theory in SMTLIB to encode the abstract state and contracts for the counter and accumulator ADTs. For Set, we used the theory of finite sets along with integers to track size; for HashTable we used sets to track the keys, and arrays for the HashMap itself. In order to encode the Stack example, we utilized the observation that for the purpose of pairwise commutativity it is sufficient to track the behavior of boundedly many top elements. In specific, since two operations can *at most* either pop the top two elements or push two elements, tracking four elements is sufficient.

b) LIFT*:* As mentioned in previous sections, our implementation uses an SMT solver to check that the transition system encoding given as input is deterministic and we have implemented the LIFT transformation described in Section IV, which gives a new specification to make sure it is total by construction (see Appendix C for the auto-lifted Counter).

c) PGEN *(predicate generation):* One of the questions that arises is how to obtain a relevant set of the predicates. As mentioned in Section II, our intuition was that the commutativity condition would need to involve terms and predicates that are used to describe the methods. Using this intuition, the procedure we use to generate a set of predicates for input to our REFINE is as follows: (*i*) take all terms that appear in the input specification of the *Pre* and *Post* conditions, grouped by the sort, (*ii*) take all predicate symbols that appear in the specification, and generate all possible atoms that are well-typed using terms extracted. For example, if `size, 1, (size+1)` are terms of sort $\mathbb{Z}$ that appear in the formula along with the predicates = and $\geq$, we generate $(\texttt{size} = 1), (\texttt{size} \geq 1)$, etc. for a total of 18 predicates. We filter out those that are trivial.

By this process, depending on the pair of methods, the number of predicates generated by our implementation of PGEN were (in parenthesis, after filtering): Counter: 25-25 (12-12), Accumulator: 1-20 (0-20), Set: 17-55 (17-34), HashTable: 18-36 (6-36), Stack: 41-61 (41-42).

d) CHOOSE*:* Even though the number of predicates obtained is relatively small, our algorithm makes two recursive calls at each step. It is thus important to be able to identify *relevant* predicates for the algorithm to be practical.

To this end, in addition to filtering trivial predicates, inspired by CEGAR techniques we prioritize predicates based on the two counterexamples generated from the validity checks in REFINE. Predicates that distinguish between the given counter examples are tried first (call these *distinguishing* predicates). The property of these predicates is that they ensure both counterexamples can be valid on recursing and thus guarantee progress. More formally, CHOOSE must return a predicate such that $\chi_c \Rightarrow H \wedge p$ and $\chi_{nc} \Rightarrow H \wedge \neg p$. In our implementation, we provided the SMT solver all the predicates upfront, which returns the evaluation on the counterexample without any additional queries. This still left us with several predicates, and we discuss the heuristics we tried to break ties.

[†]http://cs.nyu.edu/~kshitij/projects/servois/

Meth. $m(\bar{x})$		Meth.$n(\bar{y})$	Simple	Poke	φ_n^m generated by Poke heuristic
Counter			Qs (time)	Qs (time)	
decrement	$\bowtie$	decrement	3 (0.11)	3 (0.11)	true
increment	$\triangleright$	decrement	10 (0.36)	34 (0.91)	$\neg$(0 = c)
decrement	$\triangleright$	increment	3 (0.11)	3 (0.12)	true
decrement	$\bowtie$	reset	2 (0.10)	2 (0.10)	false
decrement	$\bowtie$	zero	6 (0.19)	26 (0.66)	$\neg$(1 = c)
increment	$\bowtie$	increment	3 (0.12)	3 (0.11)	true
increment	$\bowtie$	reset	2 (0.09)	2 (0.10)	false
increment	$\bowtie$	zero	10 (0.30)	34 (0.86)	$\neg$(0 = c)
reset	$\bowtie$	reset	3 (0.11)	3 (0.11)	true
reset	$\bowtie$	zero	9 (0.24)	30 (0.69)	0 = c
zero	$\bowtie$	zero	3 (0.11)	3 (0.11)	true
Accumulator					
increase	$\bowtie$	increase	3 (0.11)	3 (0.11)	true
increase	$\bowtie$	read	13 (0.31)	28 (0.63)	c + x1 = c
read	$\bowtie$	read	3 (0.09)	3 (0.09)	true
Set					
add	$\bowtie$	add	10 (0.40)	140 (4.47)	[y1 = x1 $\wedge$ y1 $\in$ S] $\vee$ [$\neg$(y1 = x1)]
add	$\bowtie$	contains	10 (0.42)	122 (3.63)	[x1 $\in$ S] $\vee$ [$\neg$(x1 $\in$ S) $\wedge$ $\neg$(y1 = x1)]
add	$\bowtie$	getsize	6 (0.21)	31 (0.93)	x1 $\in$ S
add	$\bowtie$	remove	6 (0.28)	66 (2.28)	$\neg$(y1 = x1)
contains	$\bowtie$	contains	3 (0.18)	3 (0.16)	true
contains	$\bowtie$	getsize	3 (0.13)	3 (0.13)	true
contains	$\bowtie$	remove	17 (0.57)	160 (4.81)	[S\{x1} = {y1}] $\vee$ [$\neg$(S\{x1} = {y1}) $\wedge$ y1 $\in$ {x1} $\wedge$ ...] $\vee$ [...]
getsize	$\bowtie$	getsize	3 (0.12)	3 (0.13)	true
getsize	$\bowtie$	remove	13 (0.39)	37 (1.03)	$\neg$(y1 $\in$ S)
remove	$\bowtie$	remove	21 (0.75)	192 (6.47)	[S\{y1} = {x1}] $\vee$ [$\neg$(S\{y1} = {x1}) $\wedge$ y1 $\in$ {x1} $\wedge$ ...] $\vee$ [...]
HashTable					
get	$\bowtie$	get	3 (0.17)	3 (0.15)	true
get	$\bowtie$	haskey	3 (0.14)	3 (0.14)	true
put	$\triangleright$	get	13 (0.47)	74 (2.37)	[H[x1=...] = H $\wedge$ y1 $\in$ keys] $\vee$ [$\neg$(H[x1=...] = H) $\wedge$ $\neg$(y1 = x1)]
get	$\triangleright$	put	10 (0.37)	48 (1.54)	[H[y1] = y2] $\vee$ [$\neg$(H[y1] = y2) $\wedge$ $\neg$(y1 = x1)]
remove	$\triangleright$	get	3 (0.17)	3 (0.16)	true
get	$\triangleright$	remove	13 (0.45)	40 (1.23)	$\neg$(y1 = x1)
get	$\bowtie$	size	3 (0.14)	3 (0.14)	true
haskey	$\bowtie$	haskey	3 (0.14)	3 (0.14)	true
haskey	$\bowtie$	put	10 (0.37)	52 (1.63)	[y1 $\in$ keys] $\vee$ [$\neg$(y1 $\in$ keys) $\wedge$ $\neg$(y1 = x1)]
haskey	$\bowtie$	remove	17 (0.59)	44 (1.36)	[x1 $\in$ keys $\wedge$ $\neg$(y1 = x1)] $\vee$ [$\neg$(x1 $\in$ keys)]
haskey	$\bowtie$	size	3 (0.14)	3 (0.14)	true
put	$\bowtie$	put	24 (0.97)	357 (13.50)	[H[y1] = y2 $\wedge$ x2 = H[x1] $\wedge$ ...] $\vee$ [H[y1] = y2 $\wedge$ x2 = H[x1] $\wedge$ ...] $\vee$ [...]
put	$\bowtie$	remove	6 (0.30)	33 (1.26)	$\neg$(y1 = x1)
put	$\bowtie$	size	6 (0.29)	23 (0.82)	x1 $\in$ keys
remove	$\bowtie$	remove	21 (0.89)	192 (6.95)	[keys\{x1} = {y1}] $\vee$ [$\neg$(keys\{x1} = {y1}) $\wedge$ y1 $\in$ {x1} $\wedge$ ...] $\vee$ [...]
remove	$\bowtie$	size	13 (0.45)	37 (1.13)	$\neg$(x1 $\in$ keys)
size	$\bowtie$	size	3 (0.14)	3 (0.14)	true
Stack					
clear	$\bowtie$	clear	3 (0.13)	3 (0.13)	true
clear	$\bowtie$	pop	2 (0.10)	2 (0.11)	false
clear	$\bowtie$	push	2 (0.12)	2 (0.11)	false
pop	$\bowtie$	pop	6 (0.23)	20 (0.62)	nextToTop = top
push	$\triangleright$	pop	72 (2.14)	115 (3.53)	$\neg$(0 = size) $\wedge$ top = x1
pop	$\triangleright$	push	34 (0.99)	76 (2.21)	y1 = top
push	$\bowtie$	push	13 (0.58)	20 (0.72)	y1 = x1

Figure 3. Automatically generated commutativity conditions. The φ_n^m column shows the generated commutativity condition. When right moverness ($\triangleright$) conditions are same for a pair of methods, we show them together in one row ($\bowtie$). **Qs** has the number of SMT queries made, and running time in seconds in parentheses. The experiments were run on a 2.53 GHz Intel Core 2 Duo machine with 8 GB RAM.

Simple Heuristic. One relatively simple heuristic we tried was to start by picking the predicates with the least number of terms. The intuition was that conditions would at least involve some simple atoms, and would consequently lead to simple conditions. This worked very well, on all our examples this heuristic terminated with precise commutativity conditions. In Figure 3, we give the number of queries posed to the solver and total time (in parentheses) consumed by this heuristic.

Poke Heuristic. Though the simple heuristic produces precise conditions, we now focus on the *qualitative* aspect of our synthesis algorithm. We found that in some cases the simple CHOOSE heuristic would pick predicates to split on that could have been technically avoided in the commutativity condition. Not an issue from correctness point of view, nevertheless, we tried a heuristic which tries more aggressively to find *concise* conditions in addition to being precise.

We call this *poke* heuristic, which in order to decide which predicate to pick, recurses one level deep on each predicate

and computes number of distinguishing predicates would the two calls have. The sum of values returned by the two calls becomes the weight of the predicate. We then pick the predicate with lowest weight (fewest remaining distinguishing predicates). This heuristic was found to converge much faster to the more relevant predicates. This requires more calls to the SMT solver, but since the queries were relatively simple for CVC4, it was not overall an issue. The conditions in the Figure 3 are those generated by the Poke heuristic. Please see the appendix for a comparison with those generated by Simple heuristic.

On the theoretical side, our CHOOSE implementation is fair (it satisfies condition 2 of Theorem VI.3 as in lines 9-10 of algorithm remove the predicate being tried from $\mathcal{P}$). Also, from our experiments we can conclude that our choice of predicates satisfies condition 1 in VI.3.

e) Validation: Although our algorithm is sound, we manually validated the implementation of SERVOIS by examining its output and comparing the generated commutativity conditions with those manually written in prior works. In the case of the Accumulator and Counter, our commutativity conditions were identical to those given in [13]. For the Set data-structure, the work of [13] used a less precise Set abstraction, so we instead validated against the conditions of [16]. For the HashTable, we validated that our conditions matched those given by Dimitrov *et al.* [8].

VIII. CONCLUSION

Our work shows that it possible to automatically generate commutativity conditions, something that was done manually so far. The conditions are correct by construction and ensure special cases aren't missed. These conditions can be derived statically, and used in a variety of contexts including transactional boosting [11], open nested transactions [21], and other non-transactional concurrency paradigms such as race detection [8], automatic parallelization [22], etc.

REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, May 1991.

[2] M. Aderhold. Automated synthesis of induction axioms for programs with second-order recursion. In *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR 2010)*, pages 263–277, 2010.

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

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, 2005.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806, pages 171–177. Springer, July 2011.

[6] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4):10, 2015.

[7] B. Cook and E. Koskinen. Making prophecies with decision predicates. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 399–410, 2011.

[8] D. Dimitrov, V. Raychev, M. T. Vechev, and E. Koskinen. Commutativity race detection. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 33. ACM, 2014.

[9] G. W. Ernst and W. F. Ogden. Specification of abstract data types in modula. *ACM Trans. Program. Lang. Syst.*, 2(4):522–543, Oct. 1980.

[10] L. Flon and J. Misra. A unified approach to the specification and verification of abstract data types. In *Proc. Specifications of Reliable Software Conf., IEEE Computer Society*, 1979.

[11] 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.

[12] C. A. R. Hoare. Software pioneers. chapter Proof of Correctness of Data Representations, pages 385–396. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[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*, pages 528–541. ACM, 2011.

[14] E. Koskinen and M. J. Parkinson. The push/pull model of transactions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, Portland, OR, USA, June, 2015*, 2015.

[15] E. Koskinen, M. J. Parkinson, and M. Herlihy. Coarse-grained transactions. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 19–30. ACM, 2010.

[16] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 542–555. ACM, 2011.

[17] K. R. M. Leino. Specifying and verifying programs in spec#. In *Proceedings of the 6th International Perspectives of Systems Informatics, Andrei Ershov Memorial Conference, PSI 2006*, page 20, 2006.

[18] K. R. M. Leino. Designing verification conditions for software. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, page 345, 2007.

[19] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[20] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[21] Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007*, pages 68–78. ACM, 2007.

[22] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, November 1997.

[23] A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation PLDI 2008*, pages 136–148, 2008.

[24] O. Tripp, R. Manevich, J. Field, and M. Sagiv. Janus: Exploiting parallelism via hindsight. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 145–156, New York, NY, USA, 2012. ACM.

[25] 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*, pages 145–156, 2012.

[26] O. Tripp, G. Yorsh, J. Field, and M. Sagiv. Hawkeye: Effective discovery of dataflow impediments to parallelization. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 207–224, New York, NY, USA, 2011. ACM.

[27] M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 125–135, 2008.

[28] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 327–338, 2010.

APPENDIX

A. Right-/Left-movers

Definition A.1 (Action right-mover [19])**.** *We say that an action* α_1 moves to the right of *action* α_2 *commute, denoted* $\alpha_1 \triangleright \alpha_2$*, provided that* $(\!|\alpha_2|\!) \circ (\!|\alpha_1|\!) \subseteq (\!|\alpha_1|\!) \circ (\!|\alpha_2|\!)$.

Note that left-movers can be defined as right-movers, but with arguments swapped.

Definition A.2 (Method right-mover)**.** *For* m *and* n*,*

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

A *logical right-mover condition* denoted $\vec{\Psi}^n_m$ has the same type as a commutativity condition and, again $[\![\vec{\Psi}^n_m]\!]$ denotes interpretations of $\vec{\Psi}^n_m$. Moreover, we say that $\vec{\Psi}^n_m$ is a right-mover condition for m and n provided that $\forall \sigma_0\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\ [\![\vec{\Psi}^n_m]\!]\ \sigma_0\ (m(\bar{x})/\bar{r})\ (n(\bar{y})/\bar{s}) = \mathsf{true} \Rightarrow m \triangleright n$ and similar for a *non*-right-mover condition.

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

$$\mathsf{valid}\left(\begin{array}{l}\forall \hat{\sigma}_0\ \bar{x}\ \bar{y}\ \bar{r}\ \bar{s}.\\ \quad \varphi^n_m(\hat{\sigma}_0, \bar{x}, \bar{y}, \bar{r}, \bar{s}) \Rightarrow\\ \quad \hat{\sigma}_0 \neq Err \Rightarrow\\ \quad (\!|n(\bar{y})/\bar{s}|\!)\ (\!|m(\bar{x})/\bar{r}|\!)\ \hat{\sigma}_0 \neq Err \Rightarrow\\ \quad (\!|n(\bar{y})/\bar{s}|\!)\ (\!|m(\bar{x})/\bar{r}|\!)\ \hat{\sigma}_0 = (\!|m(\bar{x})/\bar{r}|\!)\ (\!|n(\bar{y})/\bar{s}|\!)\ \hat{\sigma}_0.\end{array}\right)$$

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

B. Counter

```
# Counter data structure's abstract definition

name: counter

state:
  - name: contents
    type: Int

states_equal:
  definition: (= contents_1 contents_2)

methods:
  - name: increment
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 0)
    ensures: |
      (and (= contents_new (+ contents 1))
           (= result true))
    terms:
      Int: [contents, 1, (+ contents 1)]
  - name: decrement
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 1)
    ensures: |
      (and (= contents_new (- contents 1))
           (= result true))
    terms:
      Int: [contents, 1, (- contents 1), 0]
  - name: reset
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 0)
    ensures: |
      (and (= contents_new 0)
           (= result true))
    terms:
      Int: [contents, 0]
  - name: zero
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= contents 0)
    ensures: |
      (and (= contents_new contents)
           (= result (= contents 0)))
    terms:
      Int: [contents, 0]

predicates:
  - name: "="
    type: [Int, Int]
```

- decrement ⋈ decrement
  Simple:
  true
  Poke:
  true
- increment ▷ decrement
  Simple:
  [1 = contents]
  ∨ [¬(1 = contents) ∧ ¬(0 = contents)]
  Poke:
  ¬(0 = contents)
- decrement ▷ increment
  Simple:
  true
  Poke:
  true
- decrement ⋈ reset
  Simple:
  false
  Poke:
  false
- decrement ⋈ zero
  Simple:
  ¬(1 = contents)
  Poke:
  ¬(1 = contents)
- increment ⋈ increment
  Simple:
  true
  Poke:
  true
- increment ⋈ reset
  Simple:
  false
  Poke:
  false
- increment ⋈ zero
  Simple:
  [1 = contents]
  ∨ [¬(1 = contents) ∧ ¬(0 = contents)]
  Poke:
  ¬(0 = contents)
- reset ⋈ reset
  Simple:
  true
  Poke:
  true
- reset ⋈ zero
  Simple:
  ¬(1 = contents) ∧ 0 = contents
  Poke:
  0 = contents
- zero ⋈ zero

Simple:
true
Poke:
true

C. Counter (lifted, auto-generated)

```
methods:
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not
err_new) (>= contents 0)\n\
    \ (and (= contents_new (+ contents 1))\n     (= result
true))\n)\n    (and (not\
    \ err) err_new (not (>= contents 0)\n)))"
  name: increment
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 1
    - (+ contents 1)
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not
err_new) (>= contents 1)\n\
    \ (and (= contents_new (- contents 1))\n     (= result
true))\n)\n    (and (not\
    \ err) err_new (not (>= contents 1)\n)))"
  name: decrement
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 1
    - (- contents 1)
    - 0
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not
err_new) (>= contents 0)\n\
    \ (and (= contents_new 0)\n     (= result true))\n)\n
(and (not err) err_new\
    \ (not (>= contents 0)\n)))"
  name: reset
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 0
- args: []
  ensures: "(or (and err err_new)\n    (and (not err) (not
err_new) (>= contents 0)\n\
    \ (and (= contents_new contents)\n     (= result (=
contents 0)))\n)\n    (and\
    \ (not err) err_new (not (>= contents 0)\n)))"
  name: zero
  requires: 'true'
  return:
  - name: result
    type: Bool
  terms:
    Int:
    - contents
    - 0
name: counter
predicates:
- name: '='
  type:
  - Int
  - Int
state:
- name: contents
  type: Int
- name: err
  type: Bool
states_equal:
  definition: '(or (and err_1 err_2) (and (not err_1) (not
err_2)

    (= contents_1 contents_2)

    ))'
```

D. Accumulator

```
# Accumulator abstract definition

name: accumulator

state:
  - name: contents
    type: Int

options:


states_equal:
  definition: (= contents_1 contents_2)

methods:
  - name: increase
    args:
      - name: n
        type: Int
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (and (= contents_new (+ contents n))
           (= result true))
    terms:
      Int: [$1, contents, (+ contents $1)]
  - name: read
    args: []
    return:
      - name: result
        type: Int
    requires: |
      true
    ensures: |
      (and (= contents_new contents)
           (= result contents))
    terms:
      Int: [contents]

predicates:
  - name: "="
    type: [Int, Int]
```

- increase ⋈ increase
  Simple:
  true
  Poke:
  true
- increase ⋈ read
  Simple:
  [x1 = contents ∧ contents + x1 = contents]
  ∨ [¬(x1 = contents) ∧ contents + x1 = contents]
  Poke:
  contents + x1 = contents
- read ⋈ read
  Simple:
  true
  Poke:
  true

E. Set

```
name: set

preamble: |
  (declare-sort E 0)

state:
  - name: S
    type: (Set E)
```

```
  - name: size
    type: Int

states_equal:
  definition: (and (= S_1 S_2) (= size_1 size_2))

methods:
  - name: add
    args:
      - name: v
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (ite (member v S)
           (and (= S_new S)
                (= size_new size)
                (not result))
           (and (= S_new (union S (singleton v)))
                (= size_new (+ size 1))
                result))
    terms:
      E: [$1]
      Int: [size, 1, (+ size 1)]
      (Set E): [S, (singleton $1), (union S (singleton $1))]
  - name: remove
    args:
      - name: v
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (ite (member v S)
           (and (= S_new (setminus S (singleton v)))
                (= size_new (- size 1))
                result)
           (and (= S_new S)
                (= size_new size)
                (not result)))
    terms:
      E: [$1]
      Int: [size, 1, (- size 1)]
      (Set E): [S, (singleton $1), (setminus S (singleton
$1))]
  - name: contains
    args:
      - name: v
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (and (= S_new S)
           (= size_new size)
           (= (member v S) result))
    terms:
      E: [$1]
      Int: [size]
      (Set E): [S, (singleton $1), (setminus S (singleton
$1))]
  - name: getsize
    args: []
    return:
      - name: result
        type: Int
    requires: |
      true
    ensures: |
      (and (= S_new S)
           (= size_new size)
           (= size result))
    terms:
      Int: [size]

predicates:
  - name: "="
    type: [Int, Int]
  - name: "="
    type: [E, E]
  - name: "="
    type: [(Set E), (Set E)]
  - name: "member"
    type: [E, (Set E)]
```

- add ⋈ add
  Simple:
  [y1 = x1 ∧ y1 ∈ S]
  ∨ [¬(y1 = x1)]
  Poke:
  [y1 = x1 ∧ y1 ∈ S]
  ∨ [¬(y1 = x1)]
- add ⋈ contains
  Simple:
  [y1 = x1 ∧ y1 ∈ S]
  ∨ [¬(y1 = x1)]
  Poke:
  [x1 ∈ S]
  ∨ [¬(x1 ∈ S) ∧ ¬(y1 = x1)]
- add ⋈ getsize
  Simple:
  x1 ∈ S
  Poke:
  x1 ∈ S
- add ⋈ remove
  Simple:
  ¬(y1 = x1)
  Poke:
  ¬(y1 = x1)
- contains ⋈ contains
  Simple:
  true
  Poke:
  true
- contains ⋈ getsize
  Simple:
  true
  Poke:
  true
- contains ⋈ remove
  Simple:
  [y1 = x1 ∧ 1 = size ∧ ¬(y1 ∈ S)]
  ∨ [y1 = x1 ∧ ¬(1 = size) ∧ ¬(y1 ∈ S)]
  ∨ [¬(y1 = x1)]
  Poke:
  [S\{x1} = {y1}]
  ∨ [¬(S\{x1} = {y1}) ∧ y1 ∈ {x1} ∧ ¬(y1 ∈ S)]
  ∨ [¬(S\{x1} = {y1}) ∧ ¬(y1 ∈ {x1})]
- getsize ⋈ getsize
  Simple:
  true
  Poke:
  true
- getsize ⋈ remove
  Simple:
  [1 = size ∧ ¬(y1 ∈ S)]
  ∨ [¬(1 = size) ∧ ¬(y1 ∈ S)]
  Poke:
  ¬(y1 ∈ S)
- remove ⋈ remove
  Simple:
  [1 = size ∧ y1 = x1 ∧ ¬(y1 ∈ S)]
  ∨ [1 = size ∧ ¬(y1 = x1)]
  ∨ [¬(1 = size) ∧ y1 = x1 ∧ ¬(y1 ∈ S)]
  ∨ [¬(1 = size) ∧ ¬(y1 = x1)]
  Poke:
  [S\{y1} = {x1}]
  ∨ [¬(S\{y1} = {x1}) ∧ y1 ∈ {x1} ∧ ¬(y1 ∈ S)]
  ∨ [¬(S\{y1} = {x1}) ∧ ¬(y1 ∈ {x1})]

F. HashTable

```
# Hash table data structure's abstract definition

name: HashTable

preamble: |
  (declare-sort E 0)
  (declare-sort F 0)
```

```
state:
  - name: keys
    type: (Set E)
  - name: H
    type: (Array E F)
  - name: size
    type: Int

states_equal:
  definition: |
    (and (= keys_1 keys_2)
         (= H_1 H_2)
         (= size_1 size_2))

methods:
  - name: haskey
    args:
      - name: k0
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (and (= keys_new keys)
           (= H_new H)
           (= size_new size)
           (= (member k0 keys) result)
       )
    terms:
      Int: [size]
      E: [$1]
      (Set E): [keys]
      (Array E F): [H]
  - name: remove
    args:
      - name: v
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (ite (member v keys)
           (and (= keys_new (setminus keys (singleton v)))
                (= size_new (- size 1))
                (= H_new H)
                result)
           (and (= keys_new keys)
                (= size_new size)
                (= H_new H)
                (not result)))
    terms:
      Int: [size, 1, (- size 1)]
      E: [$1]
      (Set E): [keys, (singleton $1), (setminus keys
(singleton $1))]
      (Array E F): [H]
  - name: put
    args:
      - name: k0
        type: E
      - name: v0
        type: F
    return:
      - name: result
        type: Bool
    requires: |
      true
    ensures: |
      (ite (member k0 keys)
           (and (= keys_new keys)
                (= size_new size)
                (ite (= v0 (select H k0))
                     (and (not result)
                          (= H_new H))
                     (and result
                          (= H_new (store H k0 v0)))))
           (and (= keys_new (insert k0 keys))
                (= size_new (+ size 1))
                result
                (= H_new (store H k0 v0))))
    terms:
      Int: [size, 1, (+ size 1)]
      E: [$1]
      F: [$2, (select H $1), ]
      (Set E): [keys, (insert $1 keys)]
      (Array E F): [H, (store H $1 $2)]
  - name: get
    args:
      - name: k0
        type: E
    return:
      - name: result
        type: F
    requires: |
      (member k0 keys)
    ensures: |
      (and (= keys_new keys)
           (= H_new H)
           (= size_new size)
           (= (select H k0) result)
           )
    terms:
      Int: [size]
      E: [$1]
      F: [(select H $1)]
      (Set E): [keys]
      (Array E F): [H]
  - name: size
    args: []
    return:
      - name: result
        type: Int
    requires: |
      true
    ensures: |
      (and (= keys_new keys)
           (= H_new H)
           (= size_new size)
           (= size result))
    terms:
      Int: [size]
      (Set E): [keys]
      (Array E F): [H]

predicates:
  - name: "="
    type: [Int, Int]
  - name: "="
    type: [E, E]
  - name: "="
    type: [F, F]
  - name: "="
    type: [(Set E), (Set E)]
  - name: "="
    type: [(Array E F), (Array E F)]
  - name: "member"
    type: [E, (Set E)]
```

- get ⋈ get
  Simple:
  true
  Poke:
  true
- get ⋈ haskey
  Simple:
  true
  Poke:
  true
- put ▷ get
  Simple:
  [x2 = H[y1] ∧ y1 ∈ keys]
  ∨ [¬(x2 = H[y1]) ∧ ¬(y1 = x1)]
  Poke:
  [H[x1=x2] = H ∧ y1 ∈ keys]
  ∨ [¬(H[x1=x2] = H) ∧ ¬(y1 = x1)]
- get ▷ put
  Simple:
  [H[y1] = y2]
  ∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)]
  Poke:
  [H[y1] = y2]

∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)]

- remove ▷ get
  Simple:
  true
  Poke:
  true
- get ▷ remove
  Simple:
  [1 = size ∧ ¬(y1 = x1)]
  ∨ [¬(1 = size) ∧ ¬(y1 = x1)]
  Poke:
  ¬(y1 = x1)
- get ⋈ size
  Simple:
  true
  Poke:
  true
- haskey ⋈ haskey
  Simple:
  true
  Poke:
  true
- haskey ⋈ put
  Simple:
  [y1 = x1 ∧ y1 ∈ keys]
  ∨ [¬(y1 = x1)]
  Poke:
  [y1 ∈ keys]
  ∨ [¬(y1 ∈ keys) ∧ ¬(y1 = x1)]
- haskey ⋈ remove
  Simple:
  [y1 = x1 ∧ 1 = size ∧ ¬(y1 ∈ keys)]
  ∨ [y1 = x1 ∧ ¬(1 = size) ∧ ¬(y1 ∈ keys)]
  ∨ [¬(y1 = x1)]
  Poke:
  [x1 ∈ keys ∧ ¬(y1 = x1)]
  ∨ [¬(x1 ∈ keys)]
- haskey ⋈ size
  Simple:
  true
  Poke:
  true
- put ⋈ put
  Simple:
  [x2 = y2 ∧ x2 = H[y1] ∧ y1 ∈ keys]
  ∨ [x2 = y2 ∧ x2 = H[y1] ∧ ¬(y1 ∈ keys) ∧ ¬(y1 = x1)]
  ∨ [x2 = y2 ∧ ¬(x2 = H[y1]) ∧ ¬(y1 = x1)]
  ∨ [¬(x2 = y2) ∧ ¬(y1 = x1)]
  Poke:
  [H[y1] = y2 ∧ x2 = H[x1] ∧ size + 1 = 1 ∧ y1 ∈ keys]
  ∨ [H[y1] = y2 ∧ x2 = H[x1] ∧ size + 1 = 1 ∧ ¬(y1 ∈ keys) ∧ ¬(y1 = x1)]
  ∨ [H[y1] = y2 ∧ x2 = H[x1] ∧ ¬(size + 1 = 1) ∧ x1 ∈ keys]
  ∨ [H[y1] = y2 ∧ x2 = H[x1] ∧ ¬(size + 1 = 1) ∧ ¬(x1 ∈ keys) ∧ ¬(y1 = x1)]
  ∨ [H[y1] = y2 ∧ ¬(x2 = H[x1]) ∧ ¬(y1 = x1)]
  ∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)]
- put ⋈ remove
  Simple:
  ¬(y1 = x1)
  Poke:
  ¬(y1 = x1)
- put ⋈ size
  Simple:
  x1 ∈ keys
  Poke:
  x1 ∈ keys
- remove ⋈ remove
  Simple:
  [1 = size ∧ y1 = x1 ∧ ¬(y1 ∈ keys)]
  ∨ [1 = size ∧ ¬(y1 = x1)]
  ∨ [¬(1 = size) ∧ y1 = x1 ∧ ¬(y1 ∈ keys)]
  ∨ [¬(1 = size) ∧ ¬(y1 = x1)]
  Poke:
  [keys\{x1} = {y1}]
  ∨ [¬(keys\{x1} = {y1}) ∧ y1 ∈ {x1} ∧ ¬(y1 ∈ keys)]
  ∨ [¬(keys\{x1} = {y1}) ∧ ¬(y1 ∈ {x1})]
- remove ⋈ size
  Simple:
  [1 = size ∧ ¬(x1 ∈ keys)]
  ∨ [¬(1 = size) ∧ ¬(x1 ∈ keys)]
  Poke:
  ¬(x1 ∈ keys)
- size ⋈ size
  Simple:
  true
  Poke:
  true

G. Stack

```
# Stack definition

name: stack

preamble: |
  (declare-sort E 0)

state:
  - name: size
    type: Int
  - name: top
    type: E
  - name: nextToTop
    type: E
  - name: secondToTop
    type: E
  - name: thirdToTop
    type: E

states_equal:
  definition:
    (and (= size_1 size_2)
         (or (= size_1 0)
             (and (= size_1 1) (= top_1 top_2))
             (and (= top_1 top_2) (= nextToTop_1
nextToTop_2))))

methods:
  - name: push
    args:
      - name: v
        type: E
    return:
      - name: result
        type: Bool
    requires: |
      (>= size 0)
    ensures: |
      (and (= size_new (+ size 1))
           (= top_new v)
           (= nextToTop_new top)
           (= secondToTop_new nextToTop)
           (= thirdToTop_new secondToTop)
           (= result true))
    terms:
      Int: [size, 1, (+ size 1)]
      E: [top, nextToTop, secondToTop, thirdToTop, $1]
  - name: pop
    args: []
    return:
      - name: result
        type: E
    requires: |
      (>= size 1)
    ensures: |
      (and (= size_new (- size 1))
           (= result top)
           (= top_new nextToTop)
           (= nextToTop_new secondToTop)
           (= secondToTop_new thirdToTop))
    terms:
      Int: [size, 1, (- size 1), 0]
      E: [top, nextToTop, secondToTop, thirdToTop]
  - name: clear
    args: []
    return:
      - name: result
        type: Bool
    requires: |
      (>= size 0)
    ensures: |
      (and (= size_new 0)
           (= result true))
    terms:
      Int: [size, 0]
```

```
    E: [top, nextToTop, secondToTop, thirdToTop]

predicates:
  - name: "="
    type: [Int, Int]
  - name: "="
    type: [E, E]
```

- clear ⋈ clear
  Simple:
  true
  Poke:
  true
- clear ⋈ pop
  Simple:
  false
  Poke:
  false
- clear ⋈ push
  Simple:
  false
  Poke:
  false
- pop ⋈ pop
  Simple:
  nextToTop = top
  Poke:
  nextToTop = top
- push ▷ pop
  Simple:
  [1 = size ∧ nextToTop = top ∧ nextToTop = thirdToTop ∧ nextToTop = x1]
  ∨ [1 = size ∧ nextToTop = top ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = x1]
  ∨ [1 = size ∧ ¬(nextToTop = top) ∧ nextToTop = thirdToTop ∧ nextToTop = secondToTop ∧ top = x1]
  ∨ [1 = size ∧ ¬(nextToTop = top) ∧ nextToTop = thirdToTop ∧ ¬(nextToTop = secondToTop) ∧ top = x1]
  ∨ [1 = size ∧ ¬(nextToTop = top) ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = secondToTop ∧ top = x1]
  ∨ [1 = size ∧ ¬(nextToTop = top) ∧ ¬(nextToTop = thirdToTop) ∧ ¬(nextToTop = secondToTop) ∧ top = x1]
  ∨ [¬(1 = size) ∧ ¬(0 = size) ∧ nextToTop = thirdToTop ∧ nextToTop = secondToTop ∧ top = x1]
  ∨ [¬(1 = size) ∧ ¬(0 = size) ∧ nextToTop = thirdToTop ∧ ¬(nextToTop = secondToTop) ∧ top = x1]
  ∨ [¬(1 = size) ∧ ¬(0 = size) ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = secondToTop ∧ top = x1]
  ∨ [¬(1 = size) ∧ ¬(0 = size) ∧ ¬(nextToTop = thirdToTop) ∧ ¬(nextToTop = secondToTop) ∧ top = x1]
  Poke:
  ¬(0 = size) ∧ top = x1
- pop ▷ push
  Simple:
  [nextToTop = y1 ∧ nextToTop = top]
  ∨ [¬(nextToTop = y1) ∧ nextToTop = thirdToTop ∧ nextToTop = secondToTop ∧ y1 = top]
  ∨ [¬(nextToTop = y1) ∧ nextToTop = thirdToTop ∧ ¬(nextToTop = secondToTop) ∧ y1 = top]
  ∨ [¬(nextToTop = y1) ∧ ¬(nextToTop = thirdToTop) ∧ nextToTop = secondToTop ∧ y1 = top]
  ∨ [¬(nextToTop = y1) ∧ ¬(nextToTop = thirdToTop) ∧ ¬(nextToTop = secondToTop) ∧ y1 = top]
  Poke:
  y1 = top
- push ⋈ push
  Simple:
  [thirdToTop = y1 ∧ thirdToTop = x1]
  ∨ [¬(thirdToTop = y1) ∧ y1 = x1]
  Poke:
  y1 = x1