Verification of semantic commutativity conditions and inverse operations on linked data structures

Verification of Semantic Commutativity Conditions and Inverse Operations on Linked Data Structures

Deokhwan Kim Martin C. Rinard

Massachusetts Institute of Technology
{dkim,rinard}@csail.mit.edu

Abstract

We present a new technique for verifying *commutativity conditions*, which are logical formulas that characterize when operations commute. Because our technique reasons with the abstract state of verified linked data structure implementations, it can verify commuting operations that produce semantically equivalent (but not necessarily identical) data structure states in different execution orders. We have used this technique to verify sound and complete commutativity conditions for all pairs of operations on a collection of linked data structure implementations, including data structures that export a set interface (ListSet and HashSet) as well as data structures that export a map interface (AssociationList, HashTable, and ArrayList). This effort involved the specification and verification of 765 commutativity conditions.

Many speculative parallel systems need to undo the effects of speculatively executed operations. *Inverse operations*, which undo these effects, are often more efficient than alternate approaches (such as saving and restoring data structure state). We present a new technique for verifying such inverse operations. We have specified and verified, for all of our linked data structure implementations, an inverse operation for every operation that changes the data structure state.

Together, the commutativity conditions and inverse operations provide a key resource that language designers, developers of program analysis systems, and implementors of software systems can draw on to build languages, program analyses, and systems with strong correctness guarantees.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Commutativity Condition, Data Structure, Inverse Operation, Verification

1. Introduction

Commuting operations on shared data structures (operations that produce the same result regardless of the order in which they execute) play a central role in many parallel computing systems:

- **Parallelizing Compilers:** If a compiler can statically detect that all operations in a given computation commute, it can generate parallel code for that computation [41].
- **Deterministic Parallel Languages:** Including support for commuting operations in deterministic parallel languages increases the expressive power of the language while preserving guaranteed deterministic parallel execution [5, 42].
- **Transaction Monitors:** If a transaction monitor can detect that operations within parallel transactions commute, it can use efficient locking algorithms that allow commuting operations from different transactions to interleave [17, 49]. Because such locking algorithms place fewer constraints on the execution order, they increase the amount of exploitable parallelism.
- **Irregular Parallel Computations:** Exploiting commuting operations has been shown to be critical for obtaining good parallel performance in irregular parallel computations that manipulate linked data structures [28–30]. The reason is essentially the same as for efficient transaction monitors — it enables the use of efficient synchronization algorithms for atomic transactions that execute multiple (potentially commuting) operations on shared objects. For similar reasons, exploiting commuting operations has also been shown to be essential for obtaining good parallel performance for the SPEC benchmarks [7].

Despite the importance of commuting operations, there has been relatively little research in automatically analyzing or verifying the conditions under which operations commute. Indeed, the deterministic parallel language, transaction monitor, and irregular parallel computation systems cited above all rely on the developer to identify commuting operations, with no way to determine whether the operations do, in fact, commute or not. A mistake in identifying commuting operations invalidates both the principles upon which the systems operate and the correctness guarantees that they claim to provide.

1.1 Previous Research

Commutativity analysis [41] uses static program analysis to find operations that produce identical concrete object states in all execution orders. But this approach is inadequate for linked data structures — consider, for example, a linked list that implements a set interface. Operations that insert elements into this data structure commute at the semantic level — all insertion orders produce the same abstract set of elements. But they do not commute at the concrete implementation level — different insertion orders (even though they produce the same set) produce different linked lists. Any commutativity analysis that reasons at the concrete implementation level (as opposed to the abstract semantic level) would therefore conservatively conclude that such operations do not commute.

Another approach uses *random interpretation* [22] to detect commuting operations [1]. This technique explores control flow paths from different execution orders, with affine join operations combining states from different control flow paths to avoid exponential blowup in the number of analyzed states. Instead of directly comparing states from different execution orders, the technique reasons about the return values of the functions that the program uses to observe the different states. Because effective affine join operations do not currently exist for linked data structures, this approach does not detect commuting operations on linked data structures.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

Both of these approaches are designed only to find operations that commute in all possible object states and for all possible parameter values. But some operations commute only under certain conditions. Consider, for example, an operation that removes a key, value pair from a hash table and an operation that looks up a given key in the hash table. These operations commute only if the two keys are different. Recognizing and exploiting such *commutativity conditions* is essential to obtaining good parallel performance for many irregular computations [28–30] — these computations use the commutativity conditions to dynamically recognize and exploit commuting operations whose commutativity properties they cannot statically resolve.

1.2 Semantic Commutativity Analysis

This paper presents a new approach for verifying the specific conditions under which operations on linked data structures semantically commute. Instead of reasoning directly about the concrete data structure state, this approach builds on the availability of fully verified linked data structure implementations [51, 52] to reason at the higher semantic level of the (verified) abstract data structure state. The approach is therefore able to detect operations that commute at the semantic level even though they may produce different concrete data structure states. We have used this approach to verify both *soundness* (conceptually, if the conditions hold, the operations produce the same abstract data structure state regardless of the order in which they execute, see Section 4) and *completeness* (conceptually, if the conditions do not hold, then different execution orders produce different abstract data structure states, see Section 4) of commutativity conditions.

We have specified and verified sound and complete commutativity conditions for all pairs of operations from a variety of linked data structure implementations:

- **Sets:** ListSet and HashSet both implement a set interface. ListSet uses a singly-linked list; HashSet uses a hash table.
- **Maps:** AssociationList, HashTable, and ArrayList all implement a map interface. AssociationList uses a singly-linked list of key, value pairs; HashTable implements a separately-chained hash table — an array contains linked lists of key, value pairs with a hash function mapping keys to linked lists via the array. ArrayList maps integers to objects and is optimized for storing maps from a dense subset of the integers starting at 0.
- **Accumulator:** Accumulator maintains a value that clients can increase and read.

Altogether, we specified and verified 765 commutativity conditions (216 from ListSet and HashSet, 294 from AssociationList and HashTable, 243 from ArrayList, and 12 from Accumulator).

Because the implementations of all of these data structures have been verified to correctly implement their specifications, the semantic commutativity conditions and inverses are guaranteed to be valid for the concrete data structure implementations that execute when the program runs. We emphasize, however, that our technique works with data structure specifications, not data structure implementations. In particular, it is capable of verifying semantic commutativity conditions and inverses even in the absence of any implementation at all (in this case, of course, the commutativity conditions and inverses are sound only to the extent that the actual data structure implementation, when provided, correctly implements its specification).

We note that the well-known difficulty of reasoning about semantic properties of linked data structures [43] has limited the range of available results in this area. These verified commutativity conditions therefore provide a solid foundation for the use of these linked data structures in a range of parallel programs and systems.

1.3 Inverse Operations

In one of our usage scenarios, the system uses the commutativity conditions to dynamically detect speculatively executed operations that do not commute with previously executed operations [28–30]. In this case, the system must roll the data structure back to the abstract semantic state before the operations executed, then continue from this restored state.

Executing *inverse operations* that undo the effect of executed operations can be substantially more efficient than alternate approaches (such as pessimistically saving the data structure state before operations execute, then restoring the state to roll back the effect of the operations). Note that even though the restored abstract semantic state is the same, the underlying concrete states may differ. For example, the inverse of an operation that removes an element from a set implemented as a linked list inserts the removed element back into the list. Even though the reinserted element may appear in a different position in the list, the restored abstract set is the same as the original set.

We have developed an approach that is capable of verifying semantic inverse operations. We have specified inverses for all of the operations on our set of data structures that update the data structure state. We have used our approach to verify that all of these inverses undo the effect of the corresponding operation to correctly restore the initial abstract state of the linked data structure.

We note that the need to undo the effects of executed operations occurs pervasively throughout computer systems, from classical database transaction processing systems [21] to systems that recover from security breaches [20, 27, 38, 45]. In addition to the specific motivating use described above, the verified inverse operations may therefore find broader applicability in a variety of contexts in which it is desirable to efficiently undo data structure state changes.

1.4 Jahob

We use the Jahob program specification and verification system to specify and verify the data structure implementations, commutativity conditions, and inverse operations. Jahob enables developers to write higher-order logic specifications for Java programs [51, 52]. It also enables developers to guide proofs of complex program properties by using the Jahob integrated proof language to resolve key choice points in these proofs [52]. Once these choice points have been resolved, Jahob uses *integrated reasoning* to invoke a variety of powerful reasoning systems (such as first-order provers [44, 48], SMT provers [10, 19], MONA [24], and the BAPA [31, 34] decision procedure) to discharge the resulting automatically generated verification conditions [51, 52].

In our approach, a data structure implementor specifies commutativity conditions for pairs of data structure operations and/or inverses for individual operations that update the data structure state. Our commutativity condition and inverse operation verification system generates stylized Jahob methods whose verification establishes the validity of the corresponding commutativity conditions and inverse operations. In our experience, Jahob is often able to verify these methods without further intervention from the data structure implementor. Specifically, for our set of linked data structure implementations, all but 57 of the 1530 automatically generated commutativity testing methods and all of the eight automatically generated inverse testing methods verify as generated. If a method does not verify, the data structure implementor uses the Jahob proof language [52] to appropriately guide the verification of the method. For our set of linked data structure implementations, Jahob was able to verify the remaining 57 commutativity testing methods after we augmented the methods with a total of 201 Jahob proof language commands (see Section 5).

1.5 Contributions

This paper makes the following contributions:

- **Semantic Commutativity Analysis:** It presents a new commutativity analysis technique that verifies the soundness and completeness of semantic commutativity conditions for linked data structures. Because this analysis reasons about the abstract semantic state of the data structure (as opposed to the concrete implementation state), it can verify semantic commutativity conditions that are inherently beyond the reach of previously proposed approaches.

  To the best of our knowledge, this analysis is the first to verify semantic commutativity conditions for linked data structures.

- **Semantic Commutativity Conditions:** It presents verified sound and complete commutativity conditions for a variety of linked data structures. In this paper all of these commutativity conditions are provided by the developer and verified by our implemented system.

  To the best of our knowledge, these are the first fully verified semantic commutativity conditions for linked data structures.

- **Semantic Inverse Analysis:** It presents a new analysis for verifying inverse operations that undo the effect of previously executed operations on linked data structures. Because the analysis reasons about the abstract data structure state, it can verify semantic inverses that correctly restore the abstract data structure state even though they may produce different concrete states.

  To the best of our knowledge, this analysis is the first to verify semantic inverse operations for linked data structures.

- **Semantic Inverse Operations:** It presents verified inverses for operations that update the data structure state. Systems can use these operations to efficiently roll back speculatively executed data structure operations.

  To the best of our knowledge, these are the first fully verified semantic inverse operations for linked data structures.

- **Experience:** It discusses our experience using the Jahob [6, 51, 52] program specification and verification system to specify and verify commutativity conditions and inverse operations for a group of data structures including a ListSet and HashSet that implement a set interface, an AssociationList, HashTable, and ArrayList that implement a map interface, and an Accumulator.

We emphasize that in this paper, we focus on the specification and verification of the commutativity conditions and inverse operations. We assume that any parallel system that uses these conditions will implement some synchronization mechanism that ensures that the operations execute atomically. Such mechanisms are already implemented and available in many of the systems which we expect to be of interest [5, 7, 17, 28, 29, 41, 42, 49].

2. Example

Figure 1 presents the Jahob interface for the HashSet class, which uses a separately-chained hash table [9] to implement a set interface. The HashSet class, like all of our data structures, is written in Java augmented with specifications written in the Jahob higher-order logic specification language. The interface exports a collection of specified operations. Each operation specification consists of a precondition (the `requires` clause), a postcondition (the `ensures` clause), and a `modifies` clause. These specifications completely capture the desired behavior of the data structure (with the exception of properties involving execution time and/or memory consumption) [51, 52].

2.1 Abstract State

The interface uses the *abstract state* of the HashSet to specify the behavior of HashSet operations. This state consists of the set `contents` of objects in the HashSet, the `size` of this set, and the flag `init`, which is true if the HashSet has been initialized (see lines 2, 3, and 4 of Figure 1). The specification for the `add(v)` operation, for example, uses this abstract state to specify that, if the HashSet is initialized and the parameter `v` is not null, it adds `v` to the set of objects in the HashSet (see lines 11-14 of Figure 1).

2.2 Concrete State and Abstraction Function

When the program runs, the HashSet operations manipulate the *concrete state* of the HashSet. The concrete state consists of the array `table`, which contains pointers to linked lists of elements in the HashSet, and the `int _size`, which stores the size of the hash table (see lines 5 and 6 of Figure 1).

An *abstraction function* in the form of Jahob invariants (not shown, but see the complete data structure specifications and implementations available in the technical report version of the paper [26]) specifies the relationship between the concrete and abstract states [51, 52]. Like all of the data structures in this paper, we have used the Jahob system to verify that the HashSet correctly implements its interface [51, 52]. This verification, of course, includes the verification of the abstraction function.

2.3 Commuting Operations

Consider the `add(v1)` and `contains(v2)` operations on a HashSet `s`. These operations commute if and only if `v1` does not equal `v2` or `v1` is already in `s`. Figure 2 presents the two methods that our system automatically generates to verify the soundness and completeness of this commutativity condition. The first method (`contains_add_between_s_40`, line 1 of Figure 2) verifies soundness. The second method (`contains_add_between_c_40`, line 14 of Figure 2) verifies completeness. The methods are written in a subset of Java with Jahob annotations [52].

2.4 Verifying Commutativity Condition Soundness

The generated soundness testing method executes the `add(v1)` and `contains(v2)` operations in both execution orders on equivalent HashSets (HashSets with the same abstract state). The method specification checks that, if the commutativity condition is true, then both execution orders produce the same return values and final abstract HashSet states. If Jahob verifies the method (which it does in this case without developer assistance), it has verified that if the commutativity condition holds, then the operations commute.

The `requires` clause (lines 2 and 3 of Figure 2) ensures that the method starts with two HashSets (`sa` and `sb`) that have identical abstract states. The method first applies the `sa.contains(v1)` and `sa.add(v2)` operations to one of the HashSets (`sa`). A Jahob `assume` command (line 8 of Figure 2) instructs Jahob to assume the commutativity condition (`v1 ~= v2 | r1a`). The method next executes the two operations in the reverse order on the second HashSet `sb` (lines 10 and 11 of Figure 2). The `assert` command at the end of the method (line 12 of Figure 2) checks that the return values are the same in both execution orders and that the two HashSets have the same abstract states at the end of the method.

In this example the commutativity condition works with the *between state* that is available after the first operation executes but before the second operation executes. A system would use such a *between condition* just before executing the `add(v2)` operation to dynamically check if this operation commutes with a previously executed `contains(v1)` operation. We also verify *before conditions* (which may be used to determine if two operations that have yet to execute will commute when they execute) and *after conditions* (which may be used to trigger rollbacks when already executed operations do not commute [28, 29]).

```
public class HashSet {
    /*: public ghost specvar init :: "bool" = "False"; */

    /*: public ghost specvar contents :: "obj set" = "{}"; */
    /*: public specvar size :: "int"; */

    private Node[] table;
    private int _size;

    public HashSet()
    /*: modifies "init", "contents", "size"
        ensures "init & contents = {} & size = 0" */ { ... }

    public boolean add(Object v)
    /*: requires "init & v ~= null"
        modifies "contents", "size"
        ensures "(v ~: old contents --> contents = old contents Un {v} & size = old size + 1 & result) &
                 (v : old contents --> contents = old contents & size = old size & ~result)" */ { ... }

    public boolean contains(Object v)
    /*: requires "init & v ~= null"
        ensures "result = (v : contents)" */ { ... }

    public boolean remove(Object v)
    /*: requires "init & v ~= null"
        modifies "contents", "size"
        ensures "(v : old contents --> contents = old contents - {v} & size = old size - 1 & result) &
                 (v ~: old contents --> contents = old contents & size = old size & ~result)" */ { ... }

    public int size()
    /*: requires "init"
        ensures "result = size" */ { ... }
}
```

Figure 1. The Jahob HashSet Specification

```
static void contains_add_between_s_40(HashSet sa, HashSet sb, Object v1, Object v2)
/*: requires "sa ~= null & sb ~= null & sa ~= sb & sa..init & sb..init & v1 ~= null & v2 ~= null &
              sa..contents = sb..contents & sa..size = sb..size"
    modifies "sa..contents", "sb..contents", "sa..size", "sb..size"
    ensures "True" */
{
    boolean r1a = sa.contains(v1);
    /*: assume "v1 ~= v2 | r1a" */
    sa.add(v2);

    sb.add(v2);
    boolean r1b = sb.contains(v1);

    /*: assert "r1a = r1b & sa..contents = sb..contents & sa..size = sb..size" */
}

static void contains_add_between_c_40(HashSet sa, HashSet sb, Object v1, Object v2)
/*: requires "sa ~= null & sb ~= null & sa ~= sb & sa..init & sb..init & v1 ~= null & v2 ~= null &
              sa..contents = sb..contents & sa..size = sb..size"
    modifies "sa..contents", "sb..contents", "sa..size", "sb..size"
    ensures "True" */
{
    boolean r1a = sa.contains(v1);
    /*: assume "~(v1 ~= v2 | r1a)" */
    sa.add(v2);

    sb.add(v2);
    boolean r1b = sb.contains(v1);

    /*: assert "~(r1a = r1b & sa..contents = sb..contents & sa..size = sb..size)" */
}
```

Figure 2. HashSet Commutativity Testing Methods for Between Commutativity Condition for `contains(v1)` and `add(v2)`

```
1 static void add_0(HashSet s, Object v)
2 /*: requires "s ~= null & s..init & v ~= null"
3     modifies "s..contents ", "s..size"
4     ensures "True" */
5 {
6     boolean r = s.add(v);
7     if (r) { s.remove(v); }

8     /*: assert "s..contents = s..(old contents) & s..size = s..(old size)" */
9 }
```

Figure 3. HashSet Inverse Operation Testing Method for `add(v)`

```
1 static void put_0(HashTable s, Object k, Object v)
2 /*: requires "s ~= null & s..init & k ~= null & v ~= null"
3     modifies "s..contents ", "s..size"
4     ensures "True" */
5 {
6     Object r = s.put(k, v);
7     if (r != null) { s.put(k, r); } else { s.remove(k); }

8     /*: assert "s..contents = s..(old contents) & s..size = s..(old size)" */
9 }
```

Figure 4. HashTable Inverse Operation Testing Method for `put(k, v)`

2.5 Verifying Commutativity Condition Completeness

The `contains_add_between_c_40` method, which checks completeness, uses a similar pattern except it negates both the commutativity condition and the assertion at the end of the generated method. If Jahob verifies the method (which it does in this case without developer assistance), it has verified that if the commutativity condition does not hold, then the operations produce different return values or different abstract data structure states when they execute in different orders.

2.6 Verifying Inverse Operations

Figure 3 presents the generated inverse testing method for the HashSet `add(v)` operation. This method first executes the `add(v)` operation, then the inverse `if (r) { s.remove(v); }`. The inverse must consider two cases: when `v` was in the set before the execution of `add(v)` (in which case the inverse must not remove `v`) and when `v` was not in the set before the execution of `add(v)` (in which case the inverse must remove `v`). Note that the inverse uses the return value `r` to distinguish these two cases. The final `assert` command forces Jahob to prove that the final abstract state is the same as the initial abstract state. Note that there is no requirement that the final concrete state must be the same as the initial concrete state.

Figure 4 presents the inverse testing method for a more complex operation, the HashTable `put(k, v)` operation. If the initial state of the HashTable mapped `k` to a value, the inverse reinserts the mapping (the value to which the HashTable mapped `k` is available as the return value `r` from the `put(k, v)` operation). If the initial state did not map `k` to a value, the inverse removes the mapping that the `add(k,v)` inserted, leaving `k` unmapped as in the initial state.

Both of the inverses in our example use the return value from the operation to carry information from the initial state that the inverse can then use to undo the effect of the operation. This approach works for all of our linked data structures. We note that it is, in general, possible for operations to destroy information from the initial state that the inverse needs to restore this initial state. In this case the operation needs to save information from the initial state so that the inverse can later use this saved information to restore the initial state.

3. Commutativity and Inverse Testing Methods

The commutativity testing method generator takes as input the data structure interface and, for each pair of data structure operations, developer-specified before, between, and after commutativity conditions. It produces as output the commutativity testing methods. It then presents each method to the Jahob program verification system [51, 52]. If the method verifies, the system has verified the corresponding commutativity condition. If it does not verify, either the commutativity condition is not sound or complete or Jahob is not capable of verifying the soundness and completeness without additional developer assistance. The developer then, as appropriate, either modifies the commutativity condition or augments the generated commutativity testing methods with additional proof commands written in the Jahob proof language [52].

3.1 Completeness Commutativity Testing Template

Figure 5 presents the template that the generator uses to produce the completeness commutativity testing method. The generation process simply iterates over all commutativity testing conditions (and corresponding pairs of operations in the data structure interface), filling in the template parameters as appropriate. In Figure 5 all template parameters appear in *italic font*.

The name of the commutativity testing method contains the names of the two operations, a field that specifies whether the method tests a before, between, or after commutativity condition, the tag `c` (which identifies the method as a completeness testing method), and a numerical identifier *id*. The method takes as parameters two data structures (*sa* and *sb*) and the parameters of the two data structure operations. The `requires` clause ensures the data structures are distinct but have identical abstract states.

The generated method uses Jahob `assume` commands to instruct Jahob to assume that the preconditions of the operations hold in the first execution order. If the preconditions do not involve the state of the data structure (as in our example in Figure 2), the generator moves the preconditions up into the `requires` clause.

The generator also uses an `assume` command to insert the negation of the commutativity condition (recall that the template is a completeness template and therefore includes the negation of the condition) in the appropriate place in the generated method. The

template identifies the insertion points for all three kinds of commutativity conditions (before, between, and after). A generated method, of course, contains a commutativity condition at only one of these points.

The method next contains the operations in the reverse order, with `assume` commands instructing Jahob to assume that the preconditions of the operations hold. Once again, if the preconditions do not depend on the data structure state, the generator places them in the `requires` clause of the method, not before the operation invocations as in the template.

The method ends with the final assertion (which Jahob must prove) that either one of the corresponding return values or the final abstract states are different in the two different execution orders.

As is appropriate for a completeness testing method, this structure forces Jahob to prove that if the operation preconditions and the negation of the commutativity condition holds in the first execution order, then either one of the operation preconditions is violated in the reverse execution order or the final assertion holds.

3.2 Soundness Commutativity Testing Template

The soundness commutativity testing template has the same basic structure as the completeness template, with the exception that 1) it inserts the commutativity testing condition (not its negation), 2) it omits the `assume` command for the operation preconditions in the second execution order, and 3) the final assertion forces Jahob to prove that the return values and final abstract states are the same in both execution orders.

As is appropriate for a soundness testing method, this structure forces Jahob to prove that if the operation preconditions and commutativity condition holds in the first execution order, then the operation preconditions hold in the reverse execution order and the return values and final abstract states are the same.

3.3 Inverse Testing Methods

The inverse testing method generator takes as input the data structure interface and a developer-specified set of inverse operation pairs. It produces as output the inverse testing methods and feeds each method to the Jahob program verification system [52]. As for the commutativity testing methods, the developer may, if necessary, augment the inverse testing methods with additional Jahob proof commands.

Figure 6 presents the template that the generator uses to produce the inverse testing methods. The generation process simply iterates over all of the specified inverses, filling in the template parameters (in *italic font*) as appropriate. The final Jahob `assert` command requires Jahob to prove that the final abstract state (after the application of the inverse operation) is the same as the initial abstract state from the start of the method. Jahob must also prove the precondition of the inverse operation.

```
1  static void method1_method2_(before | between | after)_c_id
2                (sa_decl, sb_decl, argv1_decls, argv2_decls)
3  /*: requires "sa ~= null & sb ~= null & sa ~= sb &
4                 sa_abstract_state = sb_abstract_state"
5      modifies "sa_frame_condition", "sb_frame_condition"
6      ensures "True" */
7  {
8      [/*: assume "~(before_commutativity_condition)" */]
9      /*: assume "method1_precondition" */
10     r1a_type r1a = sa.method1(argv1);
11     [/*: assume "~(between_commutativity_condition)" */]
12     /*: assume "method2_precondition" */
13     r2a_type r2a = sa.method2(argv2);
14     [/*: assume "~(after_commutativity_condition)" */]

15     /*: assume "method2_precondition" */
16     r2b_type r2b = sb.method2(argv2);
17     /*: assume "method1_precondition" */
18     r1b_type r1b = sb.method1(argv1);

19     /*: assert "~(r1a = r1b & r2a = r2b &
20                    sa_abstract_state = sb_abstract_state)" */
21 }
```

Figure 5. Template for Completeness Commutativity Testing Methods

```
1  static void method_id(s_decl, argv_decls)
2  /*: requires "s ~= null & method_precondition"
3      modifies "s_frame_condition"
4      ensures "True" */
5  {
6      r_type r = s.method(argv);
7      execute_inverse_operation();

8      /*: assert "s_abstract_state = s_initial_abstract_state" */
9  }
```

Figure 6. Template for Inverse Testing Methods

4. Formal Treatment

We assume a set $s \in S$ of concrete states and a corresponding set $\widehat{s} \in \widehat{S}$ of abstract states. We also assume that the data structure defines an abstraction function $\alpha : S \to \widehat{S}$.

The commutativity and inverse testing methods work with logical formulas written in the higher-order logic Jahob specification language [52]. For our data structures, the specifications, commutativity conditions, commutativity testing methods, and inverse testing methods require only first-order logic.

Given an operation $m(v)$ on a given data structure, $\text{pre}(m(v))$ denotes the precondition of the method m from the data structure specification. The precondition is a logical formula written in the Jahob specification language [52]. It is expressed in the name space of the caller (i.e., with the formal parameter from the definition of m replaced by the actual parameter v from the caller). We write $\alpha(s) \models \text{pre}(m(v))$ if the precondition is true in the abstract state $\alpha(s)$.

We write $\langle s', r \rangle = s.m(v)$ if executing the operation $m(v)$ in state s produces return value r and new state s'. Given a starting state s and two operations $m_1(v_1)$ and $m_2(v_2)$, we are interested in the following states and return values (see Figure 7):

- $\langle s_{①;②}, r_{①;②} \rangle = s.m_1(v_1)$: the intermediate state $s_{①;②}$ and return value $r_{①;②}$ that results from executing $m_1(v_1)$ in the original state s.
- $\langle s_{1;②}, r_{1;②} \rangle = s_{①;②}.m_2(v_2)$: the final state $s_{1;②}$ and return value $r_{1;②}$ that results from executing $m_2(v_2)$ in the intermediate state $s_{①;②}$.
- $\langle s_{②;1}, r_{②;1} \rangle = s.m_2(v_2)$: the intermediate state $s_{②;1}$ and return value $r_{②;1}$ that results from executing $m_2(v_2)$ in the original state s.
- $\langle s_{2;①}, r_{2;①} \rangle = s_{②;1}.m_1(v_1)$: the final state $s_{2;①}$ and return value $r_{2;①}$ that results from executing $m_1(v_1)$ in the intermediate state $s_{②;1}$.

We are interested in states and return values for the two operations $m_1(v_1)$ and $m_2(v_2)$ executing in both execution orders ($m_1(v_1)$ followed by $m_2(v_2)$ and $m_2(v_2)$ followed by $m_1(v_1)$). The subscripts in our notation are designed to identify both the execution order and (with a circle) the most recent operation that has executed. So, for example, $s_{①;2}$ denotes the state after the opera-

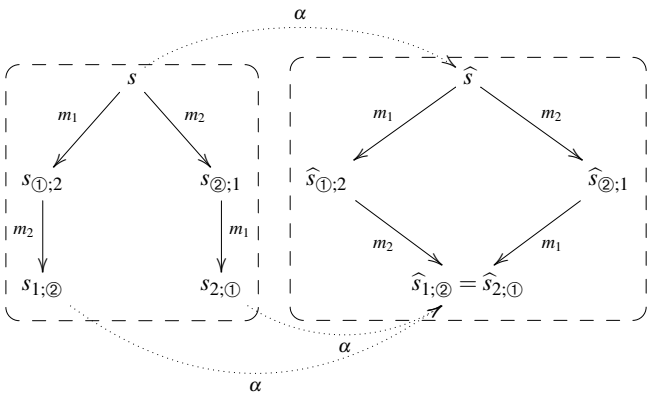

Figure 7. Execution on Concrete States and Abstract States

tion $m_1(v_1)$ executes when the operation $m_1(v_1)$ executes first and then $m_2(v_2)$ executes. Similarly, $r_{2;①}$ denotes the value that the operation $m_1(v_1)$ returns when the operation $m_2(v_2)$ executes first and then the operation $m_1(v_1)$ executes and returns $r_{2;①}$.

4.1 Commutativity Conditions

A commutativity condition ϕ is a logical formula written in the Jahob specification language [52]. In general, the free variables of ϕ can include the arguments v_1 and v_2, the return values $r_{①;2}$ and $r_{1;②}$, and abstract specification variables that denote various elements of the three abstract states $\alpha(s)$, $\alpha(s_{①;2})$, and $\alpha(s_{1;②})$. We write $(\langle s_{①;2}, r_{①;2}\rangle = s.m_1(v_1);\ \langle s_{1;②}, r_{1;②}\rangle = s_{①;2}.m_2(v_2)) \models \phi$ if the commutativity condition ϕ is satisfied when the operations execute in the order $m_1(v_1)$; $m_2(v_2)$ (first $m_1(v_1)$, then $m_2(v_2)$).

We anticipate that static analyses will work with commutativity conditions that involve the abstract state. Systems that dynamically evaluate commutativity conditions, of course, must work with the concrete data structure state, not the abstract data structure state. We therefore use the abstraction function to translate commutativity conditions over the abstract states into commutativity conditions over the concrete states. Section 5 presents examples that illustrate this translation.

4.1.1 Soundness and Completeness

Conceptually, a commutativity condition is *sound* if, whenever the commutativity condition and the preconditions of the two operations are satisfied in the first execution order, then 1) the preconditions of the operations are satisfied in the second execution order, 2) the operations return the same return values in both execution orders, and 3) the abstract final states are the same in both execution orders. We formalize this concept as follows.

Given a commutativity condition ϕ for two operations $m_1(v_1)$ and $m_2(v_2)$, the verification of the soundness commutativity testing method for these two operations establishes (by the construction of this method) the following property:

Property 1 (soundness)**.**
If $\alpha(s) \models \text{pre}(m_1(v_1))$ *and* $\alpha(s_{①;2}) \models \text{pre}(m_2(v_2))$ *and*
$(\langle s_{①;2}, r_{①;2}\rangle = s.m_1(v_1);\ \langle s_{1;②}, r_{1;②}\rangle = s_{①;2}.m_2(v_2)) \models \phi$
then $\alpha(s) \models \text{pre}(m_2(v_2))$ *and* $\alpha(s_{②;1}) \models \text{pre}(m_1(v_1))$ *and*
$r_{①;2} = r_{2;①}$ *and* $r_{1;②} = r_{②;1}$ *and* $\alpha(s_{1;②}) = \alpha(s_{2;①})$.

Conceptually, a commutativity condition is *complete* if, whenever the preconditions of the two operations are satisfied in the first execution order but the commutativity condition is not satisfied, then either 1) the preconditions of the operations are not satisfied in the second execution order, 2) the return values are different in the second execution order, or 3) the abstract final states are different in the two execution orders. We formalize this concept as follows.

Given a commutativity condition ϕ for two operations $m_1(v_1)$ and $m_2(v_2)$, the verification of the completeness commutativity testing method for these two operations establishes (by the construction of this method) the following property:

Property 2 (completeness)**.**
If $\alpha(s) \models \text{pre}(m_1(v_1))$ *and* $\alpha(s_{①;2}) \models \text{pre}(m_2(v_2))$ *and*
$(\langle s_{①;2}, r_{①;2}\rangle = s.m_1(v_1);\ \langle s_{1;②}, r_{1;②}\rangle = s_{①;2}.m_2(v_2)) \models \tilde{}\,\phi$
then $\alpha(s) \models \tilde{}\,\text{pre}(m_2(v_2))$ *or* $\alpha(s_{②;1}) \models \tilde{}\,\text{pre}(m_1(v_1))$ *or*
$r_{①;2} \neq r_{2;①}$ *or* $r_{1;②} \neq r_{②;1}$ *or* $\alpha(s_{1;②}) \neq \alpha(s_{2;①})$.

4.1.2 Kinds of Commutativity Conditions

In general, a system or analysis may wish to test if a commutativity condition is satisfied either 1) before either operation executes, 2) after the first operation executes but before the second operation executes, and/or 3) after both operations execute. We therefore identify the following kinds of commutativity conditions:

- **Before Condition:** A commutativity condition ϕ is a *before condition* if its free variables include at most the arguments v_1 and v_2 and elements of the initial abstract state $\alpha(s)$. Because a before condition does not involve the return values $r_{①;2}$ and $r_{1;②}$, the intermediate state $\alpha(s_{①;2})$, or the final state $\alpha(s_{1;②})$, it is possible to evaluate the condition before either of the operations executes.
- **Between Condition:** A commutativity condition ϕ is a *between condition* if its free variables include at most the arguments v_1 and v_2, the initial abstract state $\alpha(s)$, the first return value $r_{①;2}$, and elements of the intermediate abstract state $\alpha(s_{①;2})$. Because the between condition does not involve the second return value $r_{1;②}$ or the final abstract state $\alpha(s_{1;②})$, it is possible to evaluate the condition after the first operation $m_1(v_1)$ executes but before the second operation $m_2(v_2)$ executes. Note that if the between condition references elements of the initial abstract state $\alpha(s)$, the system may need to save corresponding values from this state before the first operation $m_1(v_1)$ executes so that it can subsequently use the saved values to evaluate the between condition after the first operation executes.
- **After Condition:** All commutativity conditions are *after conditions*. Note that if an after condition references the first return value $r_{①;2}$ or elements of the initial or intermediate abstract states $\alpha(s)$ or $\alpha(s_{①;2})$, the system may need to save referenced elements of these states so that it can evaluate the after condition after both operations execute.

In practice we expect developers to minimize references to elements of previously computed initial or intermediate abstract states when they specify between and after conditions. One strategy replaces clauses in commutativity conditions that reference elements of the initial or intermediate abstract states with equivalent clauses that reference return values from executed operations. Before conditions for our set-based data structures, for example, often test whether a parameter v of one of the operations is an element of the set in the initial state s. The return value of the operation often indicates whether the element was, in fact, an element of this initial set. The corresponding between and after conditions can then replace the clause that tests membership in the initial set with an equivalent clause that tests the return value. See Section 5 for specific occurrences of this pattern in the commutativity conditions for our set of data structures.

For completeness, some of the between conditions cannot help querying the initial state (the state before the first operation executes). For the same reason, some of the after conditions query the initial and/or between states. In practice, there are two ways to

dynamically check such commutativity conditions: 1) perform the query before the operation executes and record the result for the commutativity condition to check after the operation executes, or 2) drop the clause containing the query from the commutativity condition and use the resulting simpler, conservative, but not complete commutativity condition that does not reference the initial and/or between states.

Because the commutativity conditions for our set of data structures are both sound and complete, the before, between, and after conditions are equivalent even if they reference different return values or elements of different abstract states.

4.2 Inverse Operations

Conceptually, an operation is an *inverse* of an initial operation if whenever the precondition of the initial operation is satisfied, then 1) the precondition of the inverse is satisfied after the initial operation executes and 2) executing the inverse after the initial operation executes restores the abstract state (but not necessarily the concrete state) back to what it was before the initial operation executed. We formalize this concept as follows.

Given an operation $m_1(v_1)$ with inverse operation $m_2(v_2)$, the verification of the inverse testing method for these two operations establishes (by the construction of this method) the following property:

Property 3 (inverse)**.**
If $\alpha(s) \models \text{pre}(m_1(v_1))$
then $\alpha(s_{①;2}) \models \text{pre}(m_2(v_2))$ *and* $\alpha(s) = \alpha(s_{1;②})$.

5. Experimental Results

We next discuss the commutativity conditions, inverse operations, and verification process for our set of data structures. We first discuss the operations that each data structure exports. The source code for all of the data structures (including both specification and implementation) as well as the commutativity and inverse testing methods (which contain all of the commutativity conditions and Jahob proof constructs required to enable Jahob to verify the methods) is available in the technical report version of the paper [26].

The Accumulator implements a counter with two operations:

- `increase(v)`: Adds the number `v` to the counter.
- `read()`: Returns the value in the counter.

HashSet and ListSet implement a set of elements with the following operations. Because they implement the same specification, they have the same commutativity conditions.

- `add(v)`: Adds the element `v` to the set of elements in the data structure. Returns false if the element was already present and true otherwise.
- `contains(v)`: Returns true if the element `v` is in the set and false otherwise.
- `remove(v)`: Removes the element `v` from the set. Returns true if `v` was included in the set and false otherwise.
- `size()`: Returns the number of elements in the set.

HashTable and AssociationList implement a map from keys to values with the following operations. Because they implement the same specification, they have the same commutativity conditions.

- `containsKey(k)`: Returns true if there exists a value `v` for the key `k` in the map.
- `get(k)`: Returns the value `v` for the key `k`, or `null` if `k` is not mapped.
- `put(k, v)`: Maps the key `k` to the value `v`. Returns the previous value for the key `k`, or `null` if `k` was not mapped.
- `remove(k)`: Removes the mapping for the key `k`. Returns the value that the key `k` was mapped to, or `null` if the the data structure did not have a mapping for the key `k`.
- `size()`: Returns the number of key, value pairs in the data structure.

ArrayList implements a map from the integers to objects with the following operations:

- `add_at(i, v)`: Pushes all objects with indices greater than or equal to `i` up one position to create an empty position at index `i`, then inserts the object `v` into that position.
- `get(i)`: Returns the object at index `i`.
- `indexOf(v)`: Returns the index of the first occurrence of the object `v` or -1 if the object `v` is not in the map.
- `lastIndexOf(v)`: Returns the index of the last occurrence of the object `v` or -1 if the object `v` is not in the map.
- `remove_at(i)`: Removes the element at the specified index `i`, then slides all objects above `i` down one position to fill the newly empty position at index `i`.
- `set(i, v)`: Replaces the object at the index `i` with the object `v`. Returns the replaced object previously at index `i`.
- `size()`: Returns the number of elements in the map.

5.1 Commutativity Conditions

Tables 1 through 7 present the commutativity conditions for selected illustrative pairs of operations from our set of linked data structures. For space reasons, we do not present all 765 commutativity conditions (the complete set of conditions is available in the technical report version of the paper [26]).

The first and second columns in each table identify the pair of operations. The third column presents the commutativity conditions in terms of the arguments, return values, and abstract data structure states. These commutativity conditions are suitable for static analyses that reason about the commutativity conditions at the level of the abstract states. The fourth column translates any abstract state queries (typically set membership operations) into operations that can be invoked on the concrete data structure. These commutativity conditions are suitable for dynamically checking the commutativity conditions when the program runs.

The commutativity conditions assume the operations operate on the same data structure (operations on different data structures trivially commute). s_1 denotes the data structure state before the first operation executes; s_2 denotes the state of the same data structure after the first operation executes but before the second operation executes. Each commutativity condition in the table corresponds to the execution order in which the operation in the first column executes first followed by the operation in the second column.

The before condition tables are symmetric (for a given pair of operations, the commutativity conditions are the same for both execution orders). The between condition tables may be asymmetric if the commutativity condition references either the return value from the first operation (which is not available in the other execution order) or depends on the intermediate data structure state (which may be different in the other execution order). Similarly, the after tables may be asymmetric if the commutativity condition depends on the intermediate or final data structure states.

In general, the commutativity conditions take the form of a disjunction of clauses. Dropping clauses produces conservative sound commutativity conditions that may be easier (for static analyses) or more efficient (for dynamic checkers) to work with. Such com-

Methods		Before / Between / After Commutativity Condition
$s_1.\texttt{increase}(v_1)$	$s_2.\texttt{increase}(v_2)$	*true*
	$r_2 = s_2.\texttt{read}()$	$v_1 = 0$
$r_1 = s_1.\texttt{read}()$	$s_2.\texttt{increase}(v_2)$	$v_2 = 0$
	$r_2 = s_2.\texttt{read}()$	*true*

Table 1. Before / Between / After Commutativity Conditions on Accumulator

Methods		Before Commutativity Condition	
$s_1.\texttt{add}(v_1)$	$s_2.\texttt{add}(v_2)$	*true*	*true*
	$r_2 = s_2.\texttt{contains}(v_2)$	$v_1 \neq v_2 \vee v_1 \in s_1$	$v_1 \neq v_2 \vee s_1.\texttt{contains}(v_1) = \mathit{true}$
	$s_2.\texttt{remove}(v_2)$	$v_1 \neq v_2$	$v_1 \neq v_2$
$r_1 = s_1.\texttt{contains}(v_1)$	$s_2.\texttt{add}(v_2)$	$v_1 \neq v_2 \vee v_1 \in s_1$	$v_1 \neq v_2 \vee s_1.\texttt{contains}(v_1) = \mathit{true}$
	$r_2 = s_2.\texttt{contains}(v_2)$	*true*	*true*
	$s_2.\texttt{remove}(v_2)$	$v_1 \neq v_2 \vee v_1 \notin s_1$	$v_1 \neq v_2 \vee s_1.\texttt{contains}(v_1) = \mathit{false}$
$s_1.\texttt{remove}(v_1)$	$s_2.\texttt{add}(v_2)$	$v_1 \neq v_2$	$v_1 \neq v_2$
	$r_2 = s_2.\texttt{contains}(v_2)$	$v_1 \neq v_2 \vee v_1 \notin s_1$	$v_1 \neq v_2 \vee s_1.\texttt{contains}(v_1) = \mathit{false}$
	$s_2.\texttt{remove}(v_2)$	*true*	*true*

Table 2. Before Commutativity Conditions on ListSet and HashSet

Methods		Between Commutativity Condition	
$s_1.\texttt{add}(v_1)$	$s_2.\texttt{add}(v_2)$	*true*	*true*
	$r_2 = s_2.\texttt{contains}(v_2)$	$v_1 \neq v_2 \vee v_1 \in s_1$	$v_1 \neq v_2 \vee s_1.\texttt{contains}(v_1) = \mathit{true}$
	$s_2.\texttt{remove}(v_2)$	$v_1 \neq v_2$	$v_1 \neq v_2$
$r_1 = s_1.\texttt{contains}(v_1)$	$s_2.\texttt{add}(v_2)$	$v_1 \neq v_2 \vee r_1 = \mathit{true}$	$v_1 \neq v_2 \vee r_1 = \mathit{true}$
	$r_2 = s_2.\texttt{contains}(v_2)$	*true*	*true*
	$s_2.\texttt{remove}(v_2)$	$v_1 \neq v_2 \vee r_1 = \mathit{false}$	$v_1 \neq v_2 \vee r_1 = \mathit{false}$
$s_1.\texttt{remove}(v_1)$	$s_2.\texttt{add}(v_2)$	$v_1 \neq v_2$	$v_1 \neq v_2$
	$r_2 = s_2.\texttt{contains}(v_2)$	$v_1 \neq v_2 \vee v_1 \notin s_1$	$v_1 \neq v_2 \vee s_1.\texttt{contains}(v_1) = \mathit{false}$
	$s_2.\texttt{remove}(v_2)$	*true*	*true*

Table 3. Between Commutativity Conditions on ListSet and HashSet

Methods		Before Commutativity Condition	
$r_1 = s_1.\texttt{get}(k_1)$	$r_2 = s_2.\texttt{get}(k_2)$	*true*	*true*
	$s_2.\texttt{put}(k_2, v_2)$	$k_1 \neq k_2 \vee \langle k_1, v_2 \rangle \in s_1$	$k_1 \neq k_2 \vee s_1.\texttt{get}(k_1) = v_2$
	$s_2.\texttt{remove}(k_2)$	$k_1 \neq k_2 \vee \langle k_1, \_ \rangle \notin s_1$	$k_1 \neq k_2 \vee s_1.\texttt{containsKey}(k_1) = \mathit{false}$
$s_1.\texttt{put}(k_1, v_1)$	$r_2 = s_2.\texttt{get}(k_2)$	$k_1 \neq k_2 \vee \langle k_1, v_1 \rangle \in s_1$	$k_1 \neq k_2 \vee s_1.\texttt{get}(k_1) = v_1$
	$s_2.\texttt{put}(k_2, v_2)$	$k_1 \neq k_2 \vee v_1 = v_2$	$k_1 \neq k_2 \vee v_1 = v_2$
	$s_2.\texttt{remove}(k_2)$	$k_1 \neq k_2$	$k_1 \neq k_2$
$s_1.\texttt{remove}(k_1)$	$r_2 = s_2.\texttt{get}(k_2)$	$k_1 \neq k_2 \vee \langle k_1, \_ \rangle \notin s_1$	$k_1 \neq k_2 \vee s_1.\texttt{containsKey}(k_1) = \mathit{false}$
	$s_2.\texttt{put}(k_2, v_2)$	$k_1 \neq k_2$	$k_1 \neq k_2$
	$s_2.\texttt{remove}(k_2)$	*true*	*true*

Table 4. Before Commutativity Conditions on AssociationList and HashTable

Methods		After Commutativity Condition	
$r_1 = s_1.\texttt{get}(k_1)$	$r_2 = s_2.\texttt{get}(k_2)$	*true*	*true*
	$s_2.\texttt{put}(k_2, v_2)$	$k_1 \neq k_2 \vee r_1 = v_2$	$k_1 \neq k_2 \vee r_1 = v_2$
	$s_2.\texttt{remove}(k_2)$	$k_1 \neq k_2 \vee r_1 = \mathit{null}$	$k_1 \neq k_2 \vee r_1 = \mathit{null}$
$s_1.\texttt{put}(k_1, v_1)$	$r_2 = s_2.\texttt{get}(k_2)$	$k_1 \neq k_2 \vee \langle k_1, v_1 \rangle \in s_1$	$k_1 \neq k_2 \vee s_1.\texttt{get}(k_1) = v_1$
	$s_2.\texttt{put}(k_2, v_2)$	$k_1 \neq k_2 \vee v_1 = v_2$	$k_1 \neq k_2 \vee v_1 = v_2$
	$s_2.\texttt{remove}(k_2)$	$k_1 \neq k_2$	$k_1 \neq k_2$
$s_1.\texttt{remove}(k_1)$	$r_2 = s_2.\texttt{get}(k_2)$	$k_1 \neq k_2 \vee \langle k_1, \_ \rangle \notin s_1$	$k_1 \neq k_2 \vee s_1.\texttt{containsKey}(k_1) = \mathit{false}$
	$s_2.\texttt{put}(k_2, v_2)$	$k_1 \neq k_2$	$k_1 \neq k_2$
	$s_2.\texttt{remove}(k_2)$	*true*	*true*

Table 5. After Commutativity Conditions on AssociationList and HashTable

Methods		Between Commutativity Condition	
s_1.add_at(i_1, v_1)	s_2.add_at(i_2, v_2)	$(i_1 < i_2 \leq \|s_2\| - 1 \wedge s_2[i_2] = v_2) \vee$ $(i_1 = i_2 \wedge v_1 = v_2) \vee$ $(i_1 > i_2 \wedge s_2[i_1 - 1] = v_1)$	$(i_1 < i_2 \leq s_2.\texttt{size}() - 1 \wedge s_2.\texttt{get}(i_2) = v_2) \vee$ $(i_1 = i_2 \wedge v_1 = v_2) \vee$ $(i_1 > i_2 \wedge s_2.\texttt{get}(i_1 - 1) = v_1)$
	$r_2 = s_2$.indexOf(v_2)	$\neg(\exists i : s_2[i] = v_2) \vee$ $(\exists i < i_1 : s_2[i] = v_2) \vee$ $(\neg(\exists i < i_1 : s_2[i] = v_2) \wedge s_2[i_1] = v_2 \wedge$ $s_2[i_1 + 1] = v_2)$	$s_2.\texttt{indexOf}(v_2) < 0 \vee$ $0 \leq s_2.\texttt{indexOf}(v_2) < i_1 \vee$ $(s_2.\texttt{indexOf}(v_2) = i_1 \wedge$ $s_2.\texttt{get}(i_1 + 1) = v_2)$
	s_2.remove_at(i_2)	$(i_1 < i_2 < \|s_2\| - 1 \wedge$ $s_2[i_2] = s_2[i_2 + 1]) \vee$ $(\|s_2\| - 2 \geq i_1 = i_2 \wedge s_2[i_1 + 1] = v_1) \vee$ $(\|s_2\| - 2 \geq i_1 > i_2 \wedge s_2[i_1 + 1] = v_1)$	$(i_1 < i_2 < s_2.\texttt{size}() - 1 \wedge$ $s_2.\texttt{get}(i_2) = s_2.\texttt{get}(i_2 + 1)) \vee$ $(s_2.\texttt{size}() - 2 \geq i_1 = i_2 \wedge s_2.\texttt{get}(i_1 + 1) = v_1) \vee$ $(s_2.\texttt{size}() - 2 \geq i_1 > i_2 \wedge s_2.\texttt{get}(i_1 + 1) = v_1)$
$r_1 = s_1$.indexOf(v_1)	s_2.add_at(i_2, v_2)	$(r_1 < 0 \wedge v_1 \neq v_2) \vee$ $0 \leq r_1 < i_2 \vee$ $(r_1 = i_2 \wedge v_1 = v_2)$	$(r_1 < 0 \wedge v_1 \neq v_2) \vee$ $0 \leq r_1 < i_2 \vee$ $(r_1 = i_2 \wedge v_1 = v_2)$
	$r_2 = s_2$.indexOf(v_2)	*true*	*true*
	s_2.remove_at(i_2)	$r_1 < 0 \vee$ $0 \leq r_1 < i_2 \vee$ $(r_1 = i_2 \wedge i_2 < \|s_2\| - 1 \wedge s_2[i_2 + 1] = v_1)$	$r_1 < 0 \vee$ $0 \leq r_1 < i_2 \vee$ $(r_1 = i_2 \wedge i_2 < s_2.\texttt{size}() - 1 \wedge s_2.\texttt{get}(i_2 + 1) = v_1)$
s_1.remove_at(i_1)	s_2.add_at(i_2, v_2)	$(i_1 < i_2 \wedge s_2[i_2 - 1] = v_2) \vee$ $(i_1 = i_2 \wedge s_1[i_1] = v_2) \vee$ $(i_1 > i_2 \wedge s_2[i_1 - 1] = s_1[i_1])$	$(i_1 < i_2 \wedge s_2.\texttt{get}(i_2 - 1) = v_2) \vee$ $(i_1 = i_2 \wedge s_1.\texttt{get}(i_1) = v_2) \vee$ $(i_1 > i_2 \wedge s_2.\texttt{get}(i_1 - 1) = s_1.\texttt{get}(i_1))$
	$r_2 = s_2$.indexOf(v_2)	$(\neg(\exists i : s_2[i] = v_2) \wedge s_1[i_1] \neq v_2) \vee$ $(\exists i < i_1 : s_2[i] = v_2) \vee$ $(\neg(\exists i < i_1 : s_2[i] = v_2) \wedge s_2[i_1] = v_2 \wedge$ $s_1[i_1] = v_2 \wedge i_1 < \|s_2\|)$	$(s_2.\texttt{indexOf}(v_2) < 0 \wedge s_1.\texttt{get}(i_1) \neq v_2) \vee$ $0 \leq s_2.\texttt{indexOf}(v_2) < i_1 \vee$ $(s_2.\texttt{indexOf}(v_2) = i_1 \wedge$ $s_1.\texttt{get}(i_1) = v_2 \wedge i_1 < s_2.\texttt{size}())$
	s_2.remove_at(i_2)	$(i_1 < i_2 \wedge s_2[i_2 - 1] = s_2[i_2]) \vee$ $i_1 = i_2 \vee$ $(\|s_2\| > i_1 > i_2 \wedge s_1[i_1] = s_2[i_1])$	$(i_1 < i_2 \wedge s_2.\texttt{get}(i_2 - 1) = s_2.\texttt{get}(i_2)) \vee$ $i_1 = i_2 \vee$ $(s_2.\texttt{size}() > i_1 > i_2 \wedge s_1.\texttt{get}(i_1) = s_2.\texttt{get}(i_1))$

Table 6. Between Commutativity Conditions on ArrayList

Methods		After Commutativity Condition	
s_1.add_at(i_1, v_1)	s_2.add_at(i_2, v_2)	$(i_1 < i_2 \leq \|s_3\| - 2 \wedge s_3[i_2 + 1] = v_2) \vee$ $(i_1 = i_2 \wedge v_1 = v_2) \vee$ $(i_1 > i_2 \wedge s_3[i_1] = v_1)$	$(i_1 < i_2 \leq s_3.\texttt{size}() - 2 \wedge s_3.\texttt{get}(i_2 + 1) = v_2) \vee$ $(i_1 = i_2 \wedge v_1 = v_2) \vee$ $(i_1 > i_2 \wedge s_3.\texttt{get}(i_1) = v_1)$
	$r_2 = s_2$.indexOf(v_2)	$r_2 < 0 \vee$ $0 \leq r_2 < i_1 \vee$ $(r_2 = i_1 \wedge s_3[i + 1] = v_2)$	$r_2 < 0 \vee$ $0 \leq r_2 < i_1 \vee$ $(r_2 = i_1 \wedge s_3.\texttt{get}(i_1 + 1) = v_2)$
	s_2.remove_at(i_2)	$(i_1 < i_2 < \|s_3\| \wedge s_2[i_2] = s_3[i_2]) \vee$ $(\|s_3\| - 1 \geq i_1 = i_2 \wedge s_3[i_1] = v_1) \vee$ $(\|s_3\| - 1 \geq i_1 > i_2 \wedge s_3[i_1] = v_1)$	$(i_1 < i_2 < s_3.\texttt{size}() \wedge s_2.\texttt{get}(i_2) = s_3.\texttt{get}(i_2)) \vee$ $(s_3.\texttt{size}() - 1 \geq i_1 = i_2 \wedge s_3.\texttt{get}(i_1) = v_1) \vee$ $(s_3.\texttt{size}() - 1 \geq i_1 > i_2 \wedge s_3.\texttt{get}(i_1) = v_1)$
$r_1 = s_1$.indexOf(v_1)	s_2.add_at(i_2, v_2)	$(r_1 < 0 \wedge v_1 \neq v_2) \vee$ $0 \leq r_1 < i_2 \vee$ $(r_1 = i_2 \wedge v_1 = v_2)$	$(r_1 < 0 \wedge v_1 \neq v_2) \vee$ $0 \leq r_1 < i_2 \vee$ $(r_1 = i_2 \wedge v_1 = v_2)$
	$r_2 = s_2$.indexOf(v_2)	*true*	*true*
	s_2.remove_at(i_2)	$r_1 < 0 \vee$ $0 \leq r_1 < i_2 \vee$ $(r_1 = i_2 \wedge i_2 < \|s_3\| \wedge s_3[i_2] = v_1)$	$r_1 < 0 \vee$ $0 \leq r_1 < i_2 \vee$ $(r_1 = i_2 \wedge i_2 < s_3.\texttt{size}() \wedge s_3.\texttt{get}(i_2) = v_1)$
s_1.remove_at(i_1)	s_2.add_at(i_2, v_2)	$(i_1 < i_2 \wedge s_3[i_2 - 1] = v_2) \vee$ $(i_1 = i_2 \wedge s_1[i_1] = v_2) \vee$ $(i_1 > i_2 \wedge s_3[i_1] = s_1[i_1])$	$(i_1 < i_2 \wedge s_3.\texttt{get}(i_2 - 1) = v_2) \vee$ $(i_1 = i_2 \wedge s_1.\texttt{get}(i_1) = v_2) \vee$ $(i_1 > i_2 \wedge s_3.\texttt{get}(i_1) = s_1.\texttt{get}(i_1))$
	$r_2 = s_2$.indexOf(v_2)	$(r_2 < 0 \wedge s_1[i_1] \neq v_2) \vee$ $0 \leq r_2 < i_1 \vee$ $(r_2 = i_1 \wedge s_1[i_1] = v_2 \wedge i_1 < \|s_3\|)$	$(r_2 < 0 \wedge s_1.\texttt{get}(i_1) \neq v_2) \vee$ $0 \leq r_2 < i_1 \vee$ $(r_2 = i_1 \wedge s_1.\texttt{get}(i_1) = v_2 \wedge i_1 < s_3.\texttt{size}())$
	s_2.remove_at(i_2)	$(i_1 < i_2 \wedge s_3[i_2 - 1] = s_2[i_2]) \vee$ $i_1 = i_2 \vee$ $(\|s_3\| + 1 > i_1 > i_2 \wedge s_1[i_1] = s_3[i_1 - 1])$	$(i_1 < i_2 \wedge s_3.\texttt{get}(i_2 - 1) = s_2.\texttt{get}(i_2)) \vee$ $i_1 = i_2 \vee$ $(s_3.\texttt{size}() + 1 > i_1 > i_2 \wedge s_1.\texttt{get}(i_1) = s_3.\texttt{get}(i_1 - 1))$

Table 7. After Commutativity Conditions on ArrayList

mutativity conditions are, of course, no longer complete. If the dropped clauses usually have no effect on the value of the commutativity condition, the gain in ease of reasoning or efficiency may be worth the the loss of completeness.

One particularly useful special case is when the commutativity condition is `true` — i.e., the operations commute regardless of the data structure state. For example, `add` operations typically commute with other `add` operations, `contains` operations typically commute with other `contains` operations, and `remove` operations typically commute with other `remove` operations. Such commutativity conditions are particularly easy to reason about at compile time since the compiler does not need to reason about the parameter values or state to find commuting operations.

In general, our data structures implement the update operations that return values (`add(v)`, `remove(v)`, `put(k, v)`, `remove(k)`, `remove_at(i)`, and `set(i, v)`). For example, the `add(v)` operation from the ListSet and HashSet data structures returns true if the elememt `v` was not already present in the abstract set, while the `remove(v)` operation returns true if the element `v` was in the abstract set.

We have verified commutativity conditions for two variants of these operations — one in which the client records the return value (typically by assigning the return value to a variable) and another in which the client discards the return value. The tables in this paper present the commutativity conditions only for the variants that discard the return value; the complete tables available in the technical report version of the paper [26] present commutativity conditions for both variants. Because clients that record the return values observe more information about the data structure, the commutativity conditions for these variants can be more complex. For example, the between commutativity condition for the `r1a = sa.add(v1)`, `r2a = sa.add(v2)` pair is `(v1 ~= v2 | ~r1a)` (i.e., either `v1` and `v2` are different or `v1` was already in the set before the first operation executed), while the commutativity condition for the `s.add(v1)`, `s.add(v2)` pair is simply `true`.

For a data structure with n operations, there are $3n^2$ commutativity conditions — a before, between, and after condition for each pair of operations. For our data structures we consider two versions of operations that update the data structure — one with a return value and one that discards the return value. So there are 2 operations for Accumulator, 6 for HashSet and ListSet, 7 for HashTable and AssociationList, and 9 for ArrayList, for a total of $(3*2^2)+2*(3*6^2)+2*(3*7^2)+(3*9^2)=765$ commutativity conditions and 1530 generated commutativity testing methods — a soundness testing method and a completeness testing method for each commutativity condition.

5.2 Verification of the Commutativity Conditions

For HashSet, ListSet, AssociationList, HashTable, and Accumulator, all of the automatically generated commutativity testing methods verify as generated. Table 8 presents the time required to verify all of these automatically generated methods. The verification times are all quite reasonable — less than four minutes for all data structures except ArrayList.

For ArrayList, 429 of the 486 methods verify as generated. The entry for ArrayList in Table 8 indicates that Jahob spent 12m 18s attempting to verify all 486 automatically generated methods (with the majority of this time spent waiting for the Jahob integrated reasoning systems to time out as they try, but fail, to verify the 57 methods that require additional proof commands), 3m 04s verifying the 429 methods that verify as generated, and 27s verifying the remaining 57 methods after the addition of the required Jahob proof commands.

In general, the commutativity conditions for ArrayList are substantially more complicated than for other the data structures. We attribute this complexity in part to the use of integer indexing and in part to the presence of operations (such as `add_at` and `remove_at`) that shift the indexing relationships across large regions of the data structure.

Data Structure	Verification Time
Accumulator	0.8s
AssociationList	1m 35s
HashSet	44s
HashTable	3m 20s
ListSet	40s
ArrayList	12m 18s[†] (3m 04s, 27s)

Table 8. Commutativity Testing Method Verification Times

Proof Language Command	Count
`note`	128
`assuming`	51
`pickWitness`	22
Total	201

Table 9. Additional Jahob Proof Language Commands for Remaining 57 ArrayList Commutativity Testing Methods

The verification of the remaining 57 ArrayList commutativity testing methods required the addition of 128 `note` commands, 51 `assuming` commands, and 22 `pickWitness` commands (see Table 9).

In general, the `note` command allows the developer to specify an intermediate formula for Jahob to prove. Jahob can then use this formula in subsequent proofs. In this way, the developer can identify a lemma structure that helps Jahob find the proof.

The `assuming` command allows the developer to prove formulas of the form $A \implies B$ (by assuming A, then using A to prove B). We typically use the `assuming` command when Jahob is unable to prove a goal B in one case A of the cases of the commutativity condition (or, when proving completeness, the negation of the commutativity condition). Providing a proof of $A \implies B$ enables Jahob to verify the commutativity condition.

The `pickWitness` command allows the developer to start with an existentially quantified formula, name an element for which the formula holds, then remove the quantifier and use the resulting formula in a subsequent proof. We typically use this command when the commutativity condition (or its negation) contains an existential quantifier and we need to use the commutativity condition to prove a goal.

5.3 Verifying the Remaining 57 Methods

The 57 remaining methods fall naturally into four categories. Each requires the proof language commands to manipulate either an existentially quantified formula or the negation of such a formula.

12 of the 57 methods are a soundness testing method for a combination of either `add_at(i,v1)` or `remove_at(i)` with either `indexOf(v2)` or `lastIndexOf(v2)`. The commutativity condition is either a between or after condition. We discuss the between condition for `add_at(i,v1)` with `indexOf(v2)`. The other combinations are similar. One of the cases of the commutativity condition states that `v2` is not present in the intermediate state of the map (so that `indexOf(v2)` returns -1). In this case Jahob must prove that the element is also not present in the initial state before `add_at(i,v1)` executes (so that the return value of `indexOf(v2)` is -1 in both execution orders). In effect, Jahob must prove that if the element is not present in the intermediate state, it is also not present in the initial state — in other words, Jahob must prove that the

[†] The Z3 [10] and CVC3 [19] decision procedures were each given a 20-second timeout.

negation of one existentially quantified formula implies the negation of another existentially quantified formula. To enable Jahob to prove this fact, we use an `assuming` command, a `pickWitness` command, and several `note` commands to prove the contraposition (i.e., that if the element is present in the initial state, then it is also present in the intermediate state). A key step in the proof of the contraposition involves the identification of the new position of `v2` in the array after the `add_at(i,v1)` shifts it over (Jahob can automatically prove the cases when it is not shifted).

8 of the 57 methods are a soundness testing method for combinations of `remove_at(i)` with `indexOf(v)`. In these methods Jahob must prove that the return value of `indexOf(v)` is the same in both execution orders. The proof involves a case analysis of the initial state of the ArrayList. In one of the cases, the initial state contains two adjacent copies of `v`: one at location `i` and the other at location `i+1`. In this case, `remove_at(i)` removes the first occurrence of `v`, leaving the second occurrence of `v` in location `i`. In both execution orders `indexOf(v)` returns `i` (but `i` references conceptually different versions of `v` in the two execution orders). Jahob is unable to prove this fact without help. The addition of a `note` command that identifies the case and the new position of the second `v` after the `remove_at(v)` operation executes enables Jahob to complete the proof. The formula that identifies the case is the negation of a complex existentially quantified formula.

20 of the 57 methods are a completeness testing method for various combinations of `add_at(i,v)`, `remove_at(i,v)`, and `set(i,v)`. In these methods Jahob must prove that the two final abstract states are different. In general, Jahob accomplishes such a proof by finding an element that is present in one abstract state but not the other. In some cases, however, Jahob is unable to find such an element. The addition of an `assuming` command (which identifies the case) and `note` commands that identify the element and help Jahob prove which abstract state contains the element and which does not enables Jahob to complete the case analysis. The relevant formula identifying the case is existentially quantified.

17 of the 57 methods are a completeness testing method for combinations of either `add_at(i,v1)` or `remove_at(i)` with either `indexOf(v2)` or `lastIndexOf(v2)`. Recall that the operation `add_at(i,v1)` shifts the region of the map above `i` up to make space for `v1` at index `i`. Similarly, `remove_at(i)` shifts the region of the map above `i` down to fill the hole left by the removed element. The verification of the completeness testing method involves a case analysis of the relative positions of the inserted or removed element and the element `v2` (whose index is returned by `indexOf(v2)` or `lastIndexOf(v2)`). In one of the cases Jahob is unable to reason successfully about these relative positions. The addition of an `assuming` command (which identifies the case) and a `note` command that identifies the precise position of `v2` (this `note` command follows from the formula which identifies the case) enables Jahob to complete the case analysis. Once again, the formula identifying the relevant case is existentially quantified.

5.4 Inverse Operations

Table 10 presents, for every operation that changes the data structure's abstract state, the corresponding inverse operation that rolls back the effect of the first operation to restore the original abstract state. s_1 and s_2 denote the data structure states before and after the first operation executes, respectively. Note that some of the inverse operations use the return value from the first operation. Any system that applies such inverse operations must therefore store the return value from the first operation so that it can provide the return value to the corresponding inverse operation. All of the eight inverse testing methods verified as generated without the need for additional Jahob proof commands.

Operation		Inverse Operation
Accumulator	$s_1.\texttt{increase}(v)$	$s_2.\texttt{increase}(-v)$
ListSet HashSet	$r = s_1.\texttt{add}(v)$	if $r = \mathit{true}$ then $s_2.\texttt{remove}(v)$
	$r = s_1.\texttt{remove}(v)$	if $r = \mathit{true}$ then $s_2.\texttt{add}(v)$
AssociationList HashTable	$r = s_1.\texttt{put}(k,v)$	if $r \neq \mathit{null}$ then $s_2.\texttt{put}(k,r)$ else $s_2.\texttt{remove}(k)$
	$r = s_1.\texttt{remove}(k)$	if $r \neq \mathit{null}$ then $s_2.\texttt{put}(k,r)$
ArrayList	$s_1.\texttt{add\_at}(i,v)$	$s_2.\texttt{remove\_at}(i)$
	$r = s_1.\texttt{remove\_at}(i)$	$s_2.\texttt{add\_at}(i,r)$
	$r = s_1.\texttt{set}(i,v)$	$s_2.\texttt{set}(i,r)$

Table 10. Inverse Operations

6. Related Work

A general theme in this research is decoupling the verification of data structure implementations from the analysis of data structure clients. The immediate goal of the research presented in this paper is to verify commutativity conditions and inverses for sophisticated linked data structures. Other systems can then build on the availability of these verified conditions and inverses to more effectively analyze and transform clients that use the data structures.

This approach is designed to encapsulate the complex reasoning required to verify sophisticated data structure properties (such as commutativity and inverses) within a specialized analysis and verification framework. This encapsulation then enables the development of simpler but more scalable client analyses that can work at the higher level of the verified properties rather than attempting to directly analyze the data structure implementations along with the client together in the same analysis framework.

Decoupling data structure and client analyses is appropriate because data structure implementations and clients have different analysis/verification needs. With current technology, the verification of linked data structure implementations requires the use of sophisticated but unscalable techniques. Such techniques are appropriate in this context because of the tractable size of data structure implementations, the abstraction boundary between data structure implementations and clients, and because the cost of the resulting ambitious data structure verification efforts can be effectively amortized across many uses of the verified properties. Client analyses, in contrast, face significant scalability requirements. Working with verified data structure properties (instead of directly with data structure implementations) can enable the development of simpler but still precise client analyses that can acceptably scale to analyze large client code bases.

In previous research we have found that aspect-oriented techniques can help developers productively factor and modularize the desired client correctness properties, enabling client analyses to work within an appropriately focused analysis scope [35]. In particular, this approach supports the expression and verification of client correctness properties that involve interactions among multiple data structures and invariants over them. Other examples of projects that are designed to exploit or enable decoupled data structure implementation and client analyses include the Hob project [33, 35–37, 50], the Jahob project [6, 51, 52], research on efficient implementations of multiple relations over objects [23], and research on verifying properties of programs that use containers [13]. Typestate analyses are designed to scalably verify simpler properties involving abstract object state changes [3, 11, 15, 16, 18, 32, 37, 47].

We recently became aware of a project that reduces the verification of commutativity conditions and inverses to solving automatically generated SAT problems [40]. The specification language is an abstract imperative language (as opposed to logic specifications as in our research). To enable the reduction to SAT, the specification language does not include loops and recursion. It instead adds additional constructs to provide an acceptably expressive specification language.

We have already surveyed related work in detecting and exploiting commuting operations (see Section 1). To use the commutativity conditions in practice, systems must typically deploy some synchronization mechanism to ensure that operations execute atomically. The specific synchronization mechanisms are orthogonal to the issues we address in this paper (our goal is to verify the correctness of commutativity conditions and inverses, not to design mechanisms that ensure that operations execute atomically).

One synchronization approach is to simply use standard pessimistic (such as mutual exclusion locks [2, 4, 21] or their extension to views of objects [12]) or optimistic (such as optimistic locks [21] or software transactional memory [46]) concurrency control mechanisms. It is also possible to deploy customized nonblocking synchronization algorithms that are tailored for the specific data structure at hand [25, 39].

It is often possible to exploit the structure of the commutativity conditions to develop optimized synchronization mechanisms. For example, *abstract locks*, *forward gatekeepers*, and *general gatekeepers* are successively more general synchronization mechanisms, each of which is appropriate for a successively larger class of commutativity conditions [30]. Our sound and complete commutativity conditions typically take the form of a disjunction of clauses. Dropping clauses produces sound, simpler, but in general incomplete commutativity conditions. Simpler commutativity conditions are typically more efficient to check but expose less concurrency. It is possible to start with a sound and complete commutativity condition and generate a lattice of sound commutativity conditions by dropping clauses (here the least upper bound is disjunction [30]). Which commutativity condition is most appropriate for a given context depends on the interaction between the commutativity condition checking overhead and the amount of concurrency that the commutativity condition exposes in that context.

Commuting operations can also be used to simplify correctness proofs of parallel programs [14]. The basic idea is to use commutativity information to enable *reduction* — obtaining larger-grained atomic blocks by showing that finer-grain statements adjacent in one thread commute with statements in other threads. The specific method uses a form of computation abstraction (replacing statements with statements that have more behaviors) to enhance their ability to obtain statements that commute with other statements. While this may increase the possible behaviors of the program, the idea is to prove assertions at the end of the program. If these assertions are valid under the extended set of behaviors of the abstracted program, they are also valid for the original program.

The research presented in this paper uses a different form of abstraction (data abstraction as opposed to computation abstraction) for a different purpose (reasoning about the semantic equivalence of commuting and inverse operations on linked data structures). Our results may, however, enhance the effectiveness of techniques that reason about explicitly parallel programs — they provide such reasoning techniques with useful commutativity and inverse information about operations that manipulate linked data structures.

Bridge predicates enable developers to specify equivalent states in parallel programs with operations that are intended to execute atomically but whose execution is, in practice, interleaved [8]. These predicates can then be used to recognize and discard false positives in the interleaved execution. In the absence of bridge predicates, such false positives occur when the interleaved execution produces a concrete state that is unrealizable in the atomic execution but nevertheless semantically equivalent to some state that an atomic execution produces. Bridge predicates enable the testing system to recognize the state equivalence and therefore the acceptability of the interleaved execution.

7. Conclusion

Commuting operations, commutativity conditions, and inverse operations play an important role in a broad range of current and envisioned static reasoning systems and parallel programs, languages, and systems. We have presented new techniques for verifying semantic commutativity conditions and inverse operations for linked data structures. Our results show that these techniques can effectively verify inverse operations and sound and complete commutativity conditions for a collection of challenging linked data structures. Our results therefore provide a useful foundation that others can build on as they develop static reasoning systems and parallel programs, languages, and systems.

In the longer term we envision the integration of the commutativity condition and inverse verification techniques presented in this paper into mature software development kits. Deploying these techniques in this way would promote their wider use by developers within a familiar environment as well as their productive integration with other software development tools.

Acknowledgments

We would like to thank Karen Zee for explaining various aspects of Jahob in detail. We also acknowledge an earlier technical report version of this paper [26].

This work was supported in part by the National Science Foundation (Grants CCF-0811397, CCF-0905244, CCF-1036241 and IIS-0835652), the United States Department of Energy (Grant DE-SC0005288), and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology / National Research Foundation of Korea (Grant 2010-0001717).

References

[1] F. Aleen and N. Clark. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[2] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parall. Distrib. Syst.*, 1(1), 1990.

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proc. of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011.

[4] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[5] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2009.

[6] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. C. Rinard. Using first-order theorem provers in the Jahob data structure verification system. In *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2007.

[7] M. J. Bridges, N. Vachharajani, Y. Zhang, T. B. Jablin, and D. I. August. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, 28(1), 2008.

[8] J. Burnim, G. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

[10] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *Proc. of the International Conference on Automated Deduction (CADE)*, 2007.

[11] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[12] B. Demsky and P. Lam. Views: Object-inspired concurrency control. In *Proc. of the ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[13] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *Proc. of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011.

[14] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proc. of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2009.

[15] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[16] M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *Proc. of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2003.

[17] A. Fekete, N. A. Lynch, M. Merritt, and W. E. Weihl. Commutativity-based locking for nested transactions. In *Proc. of the International Workshop on Persistent Object Systems (POS)*, 1989.

[18] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Proc. of the International Static Analysis Symposium (SAS)*, 2003.

[19] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *Proc. of the International Conference on Automated Deduction (CADE)*, 2007.

[20] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The Taser intrusion recovery system. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[21] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[22] S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. In *Proc. of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2005.

[23] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *Proc. of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2011.

[24] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1995.

[25] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queue as an example. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[26] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. Technical Report MIT-CSAIL-TR-2010-056, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Dec. 2010.

[27] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[28] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proc. of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2007.

[29] M. Kulkarni, D. Prountzos, D. Nguyen, and K. Pingali. Defining and implementing commutativity conditions for parallel execution. Technical Report TR-ECE-09-11, School of Electrical and Computer Engineering, Purdue University, Aug. 2009.

[30] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[31] V. Kuncak and M. Rinard. Towards efficient satisfiability checking for Boolean algebra with Presburger arithmetic. In *Proc. of the International Conference on Automated Deduction (CADE)*, 2007.

[32] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2002.

[33] V. Kuncak, P. Lam, K. Zee, and M. C. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Trans. Softw. Eng.*, 32 (12), 2006.

[34] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean algebra with Presburger arithmetic. *Journal of Automated Reasoning*, 36(3), 2006.

[35] P. Lam, V. Kuncak, and M. Rinard. Crosscutting techniques in program specification and analysis. In *Proc. of the International Conference on Aspect-Oriented Software Development (AOSD)*, 2005.

[36] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *Proc. of the International Conference on Compiler Construction (CC)*, 2005.

[37] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2005.

[38] P. Mahajan, R. Kotla, C. C. Marshall, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2009.

[39] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1996.

[40] E. Moss. Personal communication, 2011.

[41] M. C. Rinard and P. C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Trans. Prog. Lang. Syst.*, 19(6), 1997.

[42] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Prog. Lang. Syst.*, 20(3), 1998.

[43] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst.*, 24(3), 2002.

[44] S. Schulz. E — a brainiac theorem prover. *AI Commun.*, 15(2–3), 2002.

[45] F. Shafique, K. Po, and A. Goel. Correlating multi-session attacks via replay. In *Proc. of the Workshop on Hot Topics in System Dependability (HotDep)*, 2006.

[46] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2), 1997.

[47] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1), 1986.

[48] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 27, pages 1965–2013. The MIT Press, 2001.

[49] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12), 1988.

[50] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2006.

[51] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[52] K. Zee, V. Kuncak, and M. C. Rinard. An integrated proof language for imperative programs. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.