Model Checking of Linearizability of Concurrent List Implementations

Pavol Černý, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri and Rajeev Alur

IST Austria (Institute of Science and Technology Austria)

Am Campus 1

A-3400 Klosterneuburg

Technical Report No. IST-2010-0001

http://pub.ist.ac.at/Pubs/TechRpts/2010/IST-2010-0001.pdf

April 19, 2010

Model Checking of Linearizability of Concurrent List Implementations

Pavol Černý[1], Arjun Radhakrishna[1], Damien Zufferey[1], Swarat Chaudhuri[2], and Rajeev Alur[3]

[1] IST Austria
[2] Pennsylvania State University
[3] University of Pennsylvania

Abstract. Concurrent data structures with fine-grained synchronization are notoriously difficult to implement correctly. The difficulty of reasoning about these implementations does not stem from the number of variables or the program size, but rather from the large number of possible interleavings. These implementations are therefore prime candidates for model checking. We introduce an algorithm for verifying linearizability of singly-linked heap-based concurrent data structures. We consider a model consisting of an unbounded heap where each node consists an element from an unbounded data domain, with a restricted set of operations for testing and updating pointers and data elements. Our main result is that linearizability is decidable for programs that invoke a fixed number of methods, possibly in parallel. This decidable fragment covers many of the common implementation techniques — fine-grained locking, lazy synchronization, and lock-free synchronization. We also show how the technique can be used to verify optimistic implementations with the help of programmer annotations. We developed a verification tool CoLT and evaluated it on a representative sample of Java implementations of the concurrent set data structure. The tool verified linearizability of a number of implementations, found a known error in a lock-free implementation and proved that the corrected version is linearizable.

1 Introduction

Concurrency libraries such as the java.util.concurrent package JSR-166 [12] or the Intel Threading Building Blocks support the development of efficient multithreaded programs by providing *concurrent data structures*, that is, concurrent implementations of familiar data abstractions such as queues, sets, and stacks. Many sophisticated algorithms that use lock-free synchronization have been proposed for this purpose (see [9] for an introduction). Such implementations are not race-free in the classic sense because they allow concurrent access to shared memory locations without using locks for mutual exclusion. This also makes them notoriously hard to implement correctly, as witnessed by several bugs found in published algorithms [4, 13]. The complexity of such algorithms is not due to the number of lines of code, but due to the multitude of interleavings that must be

examined. This suggests that such applications are prime candidates for formal verification, and in particular, that *model checking* can be a potentially effective technique for analysis.

A typical implementation of data structures such as queues and sets consists of a linked list of vertices, with each vertex containing a data value and a next pointer. Such a structure has two distinct sources of infinity: the data values in individual vertices range over an unbounded domain, and the number of vertices is unbounded. A key observation is that methods manipulating data structures typically access data values in a restricted form using only the operations of equality and order. This suggests that the contents of a list can be modeled as a *data word*: given an unbounded domain D with equality and ordering, and a finite enumerated set Σ of symbols, a data word is a finite sequence over $D \times \Sigma$. In our context, the set D can model keys used to search through a list, the ordering can be used to keep the list sorted, and Σ can be used to capture features such as marking bits or vertex-local locks used by many algorithms. However, when concurrent methods are operating on a list without acquiring global locks, vertices may become inaccessible from the head of the list. Indeed, many bugs in concurrent implementations are due to the fact that "being a list" is not an invariant, and thus, we need to explicitly model the next pointers and the shapes they induce (see Figure 1). In this paper, we propose a formal model for a class of such algorithms, identify restrictions needed for decidability of linearizability, and show that many published algorithms do satisfy these restrictions. We propose the model of *singly-linked data heaps* for representing singly-linked concurrent data structures. A singly-linked data heap consists of a set of vertices, along with a designated start vertex, where each vertex stores an element of $D \times \Sigma$ and a next field that is either null or a pointer to another vertex. Methods operating on such structures are modeled by *method automata*. A method automaton has a finite internal state and a finite number of pointer variables ranging over vertices in the heap. The automaton can test equality of pointers and equality as well as ordering of data values stored in vertices referenced by its pointer variables. It can update fields of such vertices, and update its pointer variables, for instance, by following the next fields. The model restricts the updates to pointers to ensure that the list is traversed in a monotonic manner from left to right. We show that this model is adequate to capture operations such as search, insert, and delete, implemented using a variety of synchronization mechanisms, such as fine grained vertex-local locking, lazy synchronization, and primitives such as compare-and-set.

Our main result concerns decidability of linearizability of parallel composition of two method automata. Linearizability [10] is a central correctness requirement for concurrent data structure implementations. After presenting the construction for concurrent execution of two automata, we show that it generalizes to a fixed number of method automata composed using sequential and parallel composition. Our decidability proof is based on two results.

First, we show how linearizability of two method automata A_1 and A_2 can be reduced to a reachability condition on a method automaton A. The automaton A

simulates the parallel composition for A_1 and A_2, i.e. $A_1 \parallel A_2$, and both possible linearizations, $A_1 \ ; \ A_2$ and $A_2 \ ; \ A_1$. The principal insight in the construction of $A_1 \parallel A_2$ is that the two automata can proceed through the list almost in a lock-step manner.

Second, we show that reachability for a single method automaton is decidable: given a method automaton, we want to check if there is a way to invoke the automaton so that it can reach a specified state. We show that the problem can be reduced to finite state reachability problem. The main idea is that one needs not to remember values in D, but only the equality and order information on such values.

We implemented a tool CoLT (short for Concurrency using Lockstep Tool) based on the decidability results. We evaluated the tool on a number of implementations, including one that uses hand-over-hand vertex local locking, one that uses an optimistic approach called lazy synchronization, and one that uses lock-free synchronization via compare-and-set. All of these algorithms are described in [9] and the Java source code was taken from the book's website. The tool verified that the fine-grained and lazy algorithms are linearizable, and found a known bug in the remove method of the lock-free algorithm. The experiments show that our techniques scale to real implementations of concurrent sets. The running times were under a minute for all cases of fine-grained and lazy methods (even without linearization points), and around ten minutes for lock-free methods (when the programmer specified linearization points).

Related Work Verifying correctness of concurrent data structures has received a lot of attention recently. A number machine-checked manual proofs of correctness of exist in the literature [7, 6]. Several approximate static-analysis-based approaches to the verification of concurrent data structures have been proposed [17, 16, 15, 8]. Our approach is to the best of our knowledge the first sound and complete automated analysis that captures concurrent set implementations. As for the model of the heap, closest to ours is the model of [1], but the work in [1] is on abstraction of sequential heap accessing programs. There is an emerging literature on automata and logics over data words [14, 3] and algorithmic analysis of programs accessing data words [2]. While existing literature studies acceptors and languages of data words, we want to handle destructive methods that insert and delete elements, and thus, we need a model of transducers.

2 Singly-Linked Data Heaps and Method Automata

Singly-Linked Data Heaps Let D be an unbounded set of data values equipped with equality and linear order $(D, =, <)$. Let Σ be a finite set of symbols. A *singly-linked data heap* is a tuple $(V, \mathit{next}, \mathit{flag}, \mathit{data}, h)$, where V is a finite set of vertices, *next* is a partial function from V to V, *flag* is a function from V to Σ, *data* is a function from V to D, and $h \in V$ denotes the initial vertex.

The structure L is naturally viewed as a labeled graph with edge relation *next*. L is *well-formed* if this graph has no cycle reachable from h. For each well-formed list L as above, we define a finite data word (a word over $\Sigma \times D$) that captures the part of L starting at h. Let $s = s_0 s_1 \dots s_n$ be the maximal sequence of vertices such that s_0 is h and for all i, $next(s_i) = s_{i+1}$. Figure 1 shows a singly-linked data heap with six vertices that contain values from Σ and D.

Method automata: syntax A *method automaton* is a tuple $(Q, P, DV, T, q_0, F, head, O)$, where Q is a finite set of states, P is a finite partially-ordered set of pointer variables, DV is a finite set of data variables, T is a transition relation, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, *head* is a pointer constant, and O is a set of pointer constants.

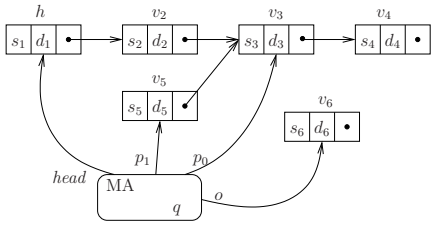

Fig. 1. Singly-linked data heap and a method automaton

A method automaton operates on a singly-linked data heap $L = (V, next, flag, data, h)$. The pointer variables range over $V \cup \{nil\}$, where *nil* is a special value, and are denoted by $p_0, p_1, p_2, \dots$. Let $\leq_P$ be the partial order on P. The partial order is required to have a minimum element, denoted by p_0. The variable p_0 is called the *current pointer*, and the other variables in P are called *lagging pointers*. The constant *head* points to the vertex h and is shared across method automata. The pointer constants in the set $O = \{o, o_0, o_1, \dots\}$ give method automata input/output capabilities and are referred to as *IO pointers*. These constants are shared with a client that invokes the method automaton. The method automaton does not change the value of pointer constants. In what follows, we use the word pointers to refer to pointer variables and constants. The set R of pointers of a method automaton is defined by $R = P \cup \{head\} \cup O$. The data variables in $DV = \{v, v_0, v_1, \dots\}$ range over the unbounded domain D.

The transition relation T is defined by a set of tuples of the form (q, G, A, q'), where $q, q' \in Q$ are states, G is a guard, and A is an action. There are no outgoing transitions from the final states.

We denote by DE a data expression defined as follows: $DE = v \mid data(p)$, where p is a pointer variable or constant. Also, let *succ* be the successor relation defined by the partial order $\leq_P$. (Note that since P is finite, $\leq_P$ defines a successor relation.)

The language of *guards* G is now defined as:

$$\begin{aligned} G ::= {} & flag(p) = s \quad (\text{for } s \in \Sigma) \mid DE = DE \mid DE < DE \mid p = p' \\ & \mid p = nil \mid p = next(p') \mid next(p) = nil \mid G \text{ and } G \mid \neg G \mid true \end{aligned}$$

where p, p' are pointer constants or variables.

An *action* of A is a term derived by the grammar

$$\begin{aligned} Act ::= \; & \mathit{flag}(p) := s \quad (\text{where } s \in \Sigma) \mid \mathit{data}(p) := DE \mid \; v := DE \mid \\ & p := p' \quad (\text{where } \mathit{succ}(p', p)) \mid \mathit{next}(p) := p' \quad (\text{where } \mathit{succ}(p', p)) \\ & \mathit{next}(p) := \mathit{nil} \mid p := \mathit{nil} \mid p_0 := \mathit{next}(p_0) \mid \mathit{Act}; \mathit{Act}. \end{aligned}$$

where p, p' are pointer variables, p_0 is the current pointer (minimum pointer variable). The restriction enforce that the list is traversed in a monotonic manner. This necessitates that pointer variables are statically ordered, and the furthest pointer can be assigned to the next of its vertex, but lagging pointers can be assigned only to a pointer further up in this ordering. Fields of vertices, including the next field, corresponding to lagging pointers can still be updated.

We require the actions to satisfy a restriction *OW*, abbreviation for "*One write before move.*" This restriction states that there is at most one action modifying $\mathit{flag}(p)$, at most one action modifying $\mathit{data}(p)$, and at most one action modifying $\mathit{next}(p)$ performed between two successive changes of the value of the pointer variable p. The restriction can be enforced syntactically — we omit the details. We note that the restriction OW holds for every implementation we have encountered and that we show that without this restriction, the linearizability problem becomes undecidable.

Figure 1 shows a method automaton in state q. Its *head* pointer points to the vertex h of the list. A client of the automaton can store values in the vertex v_6 pointed to by the IO pointer o. The variables p_1 and p_2 are pointer variables of the method automaton.

Examples We illustrate the model by showing how the model captures synchronization primitives and other core features of concurrent data structure algorithms.

- *Traversing a list.* Let us suppose we want the current pointer p_0 to traverse a list (assumed to be sorted) until it finds a data value equal or larger to the one stored at a node pointed to by an IO pointer o. A method automaton can achieve this by having a transition such as: $(q, \mathit{data}(p_0) < \mathit{data}(o), p_0 := \mathit{next}(p_0), q)$.
- *Inserting a node.* We show how a node can be inserted. We assume that the position to insert the node has been found - the new node o is to be inserted between p_1 and p_0. The transition relation can then include $(q, \mathit{true}, \mathit{next}(o) := p_0, q_1)$ and $(q_1, \mathit{true}, \mathit{next}(p_1) := o, q_2)$.
- *Locking individual vertices.* We can model locking of nodes by the Σ nodes. Let us suppose that $\Sigma = \{u, l_1, l_2, \ldots\}$, for unlocked, locked by thread 1, locked by thread 2, etc. A locking transition can then look as follows: $(q_0, \mathit{flag}(p) = u, \mathit{flag}(p) := l_1, q_1)$ for thread number 1. Unlocking can be modeled as follows: $(q_1, \mathit{flag}(p) = l_1, \mathit{flag}(p) := u, q_2)$.
- *Modeling compare-and-set.* The synchronization operation compare-and-set is supported by several contemporary architectures as well as Java Concurrency library. The operation takes two arguments, an expected value (ev) and an update value (uv). If the current value of the register (for hardware) or a reference (in Java) is equal to the expected value, then it is replaced by

the update value. The operation returns a Boolean indicating whether the value changed. The operation is modeled by the following transition tuple: $(q, \mathit{data}(p) = ev, \mathit{data}(p) := uv, q'))$.

Semantics An *automaton invocation*, denoted by $A(L, \mathit{ionodes})$, consists of a method automaton $A = (Q, P, DV, T, q_0, F, \mathit{head}, O)$, a singly-linked data heap $L = (V, \mathit{next}, \mathit{flag}, \mathit{data}, h)$, and a function *ionodes* from O to V. The pair $(L, \mathit{ionodes})$ is referred to as *method input*. A method input is *well-formed* if L is well-formed, and for all variables $o \in O$, we have that the vertex $\mathit{ionodes}(o)$ is unreachable from h and $\mathit{next}(\mathit{ionodes}(o))$ is undefined. A method automaton is thus initialized by having its *head* pointer pointing to the head of L and its input variables in O initialized by the function *ionodes*. The output of a method is also realized via the variables O, which are shared with the client.

The semantics is given by the transition system denoted by $[\![A(L, \mathit{ionodes})]\!]$. The definition formalizes the following intuition: a transition of the method automaton is chosen nondeterministically and executed atomically. Let us use a special value *nil* to model the null pointer, and let $q_{err} \notin Q$ be a special state reached on null-pointer dereference. Let $L = (V, \mathit{next}, \mathit{flag}, \mathit{data}, h)$ and $A = (Q, P, DV, T, q_0, F, \mathit{head}, O)$. A node[4] $s = (L, q, U, dv)$ of $[\![A(L, \mathit{ionodes})]\!]$ has four components: a list L, a state q in $Q_{err} = Q \cup \{q_{err}\}$, a valuation of pointers $U : R \to V \cup \{\mathit{nil}\}$ and a valuation of data variables $dv : DV \to D$. A node is *initial* if it is of the form (L, q_0, U, dv), where U sets all pointer variables to h and dv sets all the pointer variables to the value $\mathit{data}(h)$. Note that there is a unique initial node in $[\![A(L, \mathit{ionodes})]\!]$.

The transition relation of $[\![A(L, \mathit{ionodes})]\!]$ is defined as expected. For example, if $(q, \mathit{true}, p := \mathit{next}(p), q')$ is a transition of the method automaton A, then there is a transition from a node (L, q, U, dv) to a node (L, q', U', dv), where $U'(p') = U(p')$ for all $p' \in R$ such that $p' \neq p$ and $U'(p) = U(\mathit{next}(p))$.

Composition of method automata Let us consider two method automata A_1 and A_2. Their parallel composition $A_1 \parallel A_2$ will be defined using a transition system. The transition system will be denoted by $T_P(A_1, A_2, L, \mathit{ionodes})$. Intuitively, the set of nodes of the transition system is the product of sets of nodes of transition systems $[\![A_1(L, \mathit{ionodes}_1)]\!]$ and $[\![A_2(L, \mathit{ionodes}_2)]\!]$, with the singly-linked data heap L being shared between the two automata. The transition function defines interleaving semantics. We omit further details in the interest of space. The relation $(L, \mathit{ionodes}) \xrightarrow{A_1 \| A_2} (L', \mathit{ionodes}')$ is defined similarly as in the case of one automaton. Note that here, *ionodes* is a valuation of IO variables of both automata. For *sequential composition* $A_1 \,;\, A_2$, we analogously define the system $T_S(A_1, A_2, L, \mathit{ionodes})$ and $(L, \mathit{ionodes}) \xrightarrow{A_1 \| A_2} (L', \mathit{ionodes}')$.

[4] In this paper, we reserve the word state for the states of method automata. For other transitions systems, we talk of nodes.

3 Verifying Linearizability

Linearizability [10] is the standard correctness condition for concurrent data structure implementations. In this section, we study the linearizability problem for parallel composition of method automata.

Representing Abstract Data Types We start by defining an equivalence relation on singly-linked data heaps. Two singly-linked data heaps are equivalent when they represent the same value of an abstract data type ADT. We define a function adt that takes a singly-linked data heap and returns a value in ADT. As an example, we consider sets of elements of the data domain D as the abstract data type. In this case, the range of the function adt would be 2^D. For example, the value $adt(L)$ can be the set of data values from all the reachable unmarked nodes (if marked nodes are assumed to be deleted).

Let A be a method automaton. Let $(L, ionodes)$ and $(L', ionodes')$ be method inputs. The relation $(L, ionodes) \equiv (L', ionodes')$ holds $adt(L) = adt(L')$ and $ionodes(o) = ionodes(o)$, for all IO variables o of A.

Two method linearizability Given two method automata A_1 and A_2, we say that the parallel composition $A_1 \parallel A_2$ is *linearizable* if and only if the following condition holds: For all $L_P, L'_P, ionodes_P, ionodes'_P$ such that $(L_P, ionodes_P)$ is a well-formed method input, if $(L_P, ionodes_P) \xrightarrow{A_1 \parallel A_2} (L'_P, ionodes'_P)$, then

- either there exist $L_{S1}, L'_{S1}, ionodes_{S1}, ionodes'_{S1}$ such that $(L_{S1}, ionodes_{S1})$ is a well-formed method input, and $(L_{S1}, ionodes_{S1}) \xrightarrow{A_1;A_2} (L'_{S1}, ionodes'_{S1})$ and $(L_P, ionodes_P) \equiv (L_{S1}, ionodes_{S1})$ and $(L'_P, ionodes'_P) \equiv (L'_{S1}, ionodes'_{S1})$
- or there exist $L_{S2}, L'_{S2}, ionodes_{S2}, ionodes'_{S2}$ such that $(L_{S2}, ionodes_{S2})$ is a well-formed method input, and $(L_{S2}, ionodes_{S2}) \xrightarrow{A_2;A_1} (L'_{S2}, ionodes'_{S2})$ and $(L_P, ionodes_P) \equiv (L_{S2}, ionodes_{S2})$ and $(L'_P, ionodes'_P) \equiv (L'_{S2}, ionodes'_{S2})$.

The definition of two method linearizability intuitively says that for every (interleaved) execution of $A_1 \parallel A_2$, there is an equivalent (sequential) execution of either $A_1 \,;\, A_2$ or $A_2 \,;\, A_1$. Note that the standard definition of linearizability [10] requires that if a method m_1 starts executing after m_2 has finished, the effect of m_2 must be visible to m_1. This requirement does not explicitly appear in our definition, because if there are only two methods composed in parallel, and one of them starts after the other has finished, the execution is sequential.

The definition of two method linearizability captures the standard definition of linearizability [10] for the case of parallel composition of two methods. In the standard definition, the requirement that all *histories* (possibly of unbounded length) are linearizable is used to capture the requirement on the contents of the list from our definition. In other words, given two methods A_1 and A_2, two-method-linearizability not only checks that all histories of $A_1 \parallel A_2$ are linearizable, it also checks that all histories of $P_1 \,;\, P \,;\, P_2$ are linearizable, for all sequential programs P_1 and P_2. As an example, consider a set with methods `insert`, `contains`. With these two methods, the requirement from our definition that starting on the same list, the interleaved and one of the sequential executions should finish with the same list is captured by the history that (starting

with the empty list) calls insert at the beginning and contains at the end of the execution. A formal comparison of the definitions is deferred to the full version.

Decision problem We can now formulate the decision problem we consider in this paper:

> Given two method automata A_1 and A_2, the *two method linearizability problem* is to decide whether $A_1 \parallel A_2$ is linearizable.

3.1 Reachability

In order to show that the two method linearizability is decidable, we will need the following results. First, we show that the effect of two method automata running in parallel can be captured by a single method automaton, which is built using a lockstep construction. Second, we show that reachability is decidable for method automata.

Theorem 1. *Given two method automata A_1 and A_2, there exists a method automaton $LS(A_1 \parallel A_2)$ such that for all $L_P, L'_P, ionodes_P, ionodes'_P$ such that $(L_P, ionodes_P)$ is a well-formed method input, we have $(L, ionodes) \xrightarrow{A_1 \| A_2} (L', ionodes')$ iff $(L, ionodes) \xrightarrow{LS(A_1 \| A_2)} (L', ionodes')$.*

Proof. The idea behind constructing a method automaton $LS(A_1 \parallel A_2)$ is to update the current pointers of the two A_1 and A_2 automata in lockstep manner — i.e. the current pointers of the two automata traverse the list at most one step apart. If e.g. the current pointer of A_1 is one step ahead of the current pointer of A_2, then transitions of A_2 are scheduled until the current pointers point to the same position. The lockstep construction is a type of partial-order reduction. The construction is complicated by the presence of lagging pointers. The solution consists of nondeterministically guessing the interaction of the automata via lagging pointers. It is in this step that the restriction OW is needed.

We now present the proof in more detail. We start by demonstrating the main ideas in a simplified setting, and we then show how to extend the proof. We assume that neither of the automata A_1 and A_2 have lagging pointers and they do not modify the *next* field of the elements (i.e. no pointer modifications are performed). Both A_1 and A_2 have only one pointer (the current pointer). If A conforms to these restrictions, we call it a *simple MA*.

Furthermore, we simplify the question to a reachability question. We ask whether a pair (q_1, q_2) is reachable in $A_1 \parallel A_2$ when the current pointers of both A_1 and A_2 point to the same vertex of the list L. More formally, the pair (q_1, q_2) is *same-vertex-reachable in* $A_1 \parallel A_2$ if there exists a well-formed input $(L, ionodes)$ and if a vertex $(L, q_1, q_2, U_1, U_2, dv_1 dv_2)$ is reachable in $T_P(A_1, A_2, L, ionodes)$ and we have that $U_1(p_0^1) = U_2(p_0^2)$ where p_0^1 and p_0^2 denote the current pointers of the two automata.

Lemma 1. *Let A_1 and A_2 be two simple MAs, and let q_1 be a state of A_1 and q_2 be a state of A_2. There exists a method automaton $LS(A_1 \parallel A_2)$ and a state*

q of $LS(A_1 \parallel A_2)$ such that (q_1, q_2) is same-vertex-reachable in $A_1 \parallel A_2$ if and only if q is reachable in $LS(A_1 \parallel A_2)$.

In the rest of this proof, A_1 and A_2 are arbitrary but fixed. We denote $LS(A_1 \parallel A_2)$ by LS. We construct the MA LS as follows. Let LS be defined by the tuple $(Q_C, P_C, dv, T_C, q_0^C, F_C, head, O_C)$. The set Q_C is the product of the sets Q_1 and Q_2 and Q_B, where Q_B allows storing some bookkeeping information. The set of pointer variables contains two pointers - the current pointer p_0^C, and one lagging pointer p_1^C. The initial state q_0^C is (q_0^1, q_0^2, q_0^B) and the set of final states F_C is $F_1 \times F_2 \times F_B$.

The transition function of LS simulates the parallel composition of A_1 and A_2, with an important restriction: the current pointers of the two automata are never more than one step apart. Specifically, the following is an invariant: $p_0^C = p_1^C$ or $p_0^C = next(p_1^C)$. When A_1 is one step ahead, the pointer p_0^C represents the current pointer of A_1 and p_1^C pointer represents the current pointer of A_2; when A_2 is one step ahead, the situation is reversed. The automaton LS keeps track of whether A_1 or A_2 is one step ahead in the Q_B part of the state.

A *move-transition* is a transition where the action is of the form $p_0 = next(p_0)$. The automaton LS proceeds by simulating sequences of transitions of A_1 or A_2. It simulates A_1 by performing a transition of A_1 on the Q_1 component of the state. The simulation proceeds as follows:

- If $p_0^C = p_1^C$, then LS chooses nondeterministically whether to simulate A_1 (A_2). It then simulates A_1 (A_2) until A_1 (A_2) performs a move-transition. At that point, $p_0^C = next(p_1^C)$, and the simulation continues as described next.
- If $p_0^C = next(p_1^C)$ then:
  - If A_1 is one step ahead, CA simulates A_2 until A_2 performs a move-transition. At that point, $p_0^C = p_1^C$.
  - If A_2 is one step ahead, CA simulates A_1 until A_1 performs a move-transition. At that point, $p_0^C = p_1^C$.

The pair (q_1, q_2) is *same-vertex reachable in LS* if there exist a well-formed input $(L, ionodes)$, a list L' and function U'_L, dv'_L such that (L', q, U'_L, dv'_L) is reachable from initial state in $[\![LS(L, ionodes)]\!]$, $U'_L(p_0^C) = U'_L(p_1^C)$ and $q = (q_1, q_2, q_b)$, for some q_b.

We now prove that LS is the desired automaton, i.e. it is an automaton which we can analyze in order to decide whether (q_1, q_2) is same-vertex-reachable in $A_1 \parallel A_2$ — that is, we need to prove that (q_1, q_2) is same-vertex reachable in $A_1 \parallel A_2$ if and only if (q_1, q_2) is same-vertex reachable in LS.

A node in $[\![LS(L, ionodes)]\!]$ (or $T_P(A_1, A_2, L, ionodes)$) is called a *target node* if it represents a point in the execution where the current pointers of the two automata point to the same state and their states are q_1 and q_2.

Let t_C be a target node in $[\![LS(L, ionodes)]\!]$. It is easy to prove that if t_C is reachable, then there exists a target node t reachable in $T_P(A_1, A_2, L, ionodes)$, as the set of paths in $T_P(A_1, A_2, L, ionodes)$ is a superset of the set of paths in $[\![LS(L, inp_O)]\!]$.

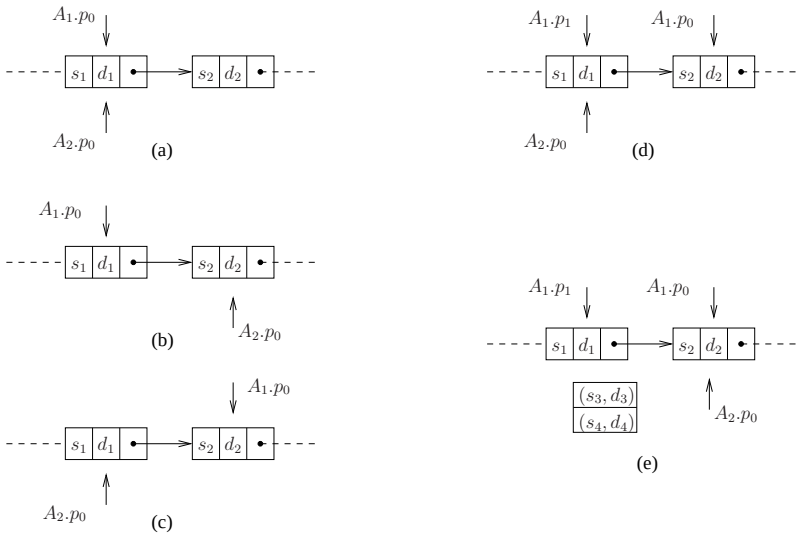

Fig. 2. Lockstep construction

The other implication is more difficult to prove. Let t be a target node in $T_P(A_1, A_2, L, ionodes)$. Let Z be a path leading to t from the initial node. Let $s = s_0 s_1 \dots s_n$ be the maximal sequence of vertices in the list L such that s_0 is h and for all i, $next(s_i) = s_{i+1}$. Let $K \subseteq \{0, 1, \dots n\}$, such that $i \in K$ if A_1 was first at position i (more formally $i \in K$ iff a node where $U(p_0^1) = s_i$ occurs in Z before a node where $U(p_0^2) = s_i$).

The only time when "scheduling" occurs in LS is when $p_0^C = p_1^C$. To construct the desired path that reaches a target node t_C, we resolve the nondeterminism using the set K obtained from the path Z in $T_P(A_1, A_2, L, ionodes)$. If $p_0^C = p_1^C = s_i$ and $i + 1 \in K$, then we continue on the path where LS simulates A_1 first (if $i + 1 \notin K$ the situation is reversed).

In this way, we have constructed a path Z' from the initial state in $[\![LS(L, ionodes)]\!]$. We can now prove that for all i, whenever in Z' a node occurs where $U(p_0^C) = U(p_1^C) = s_i$ for the first time, then the the two simulated automata A_1 and A_2 are in the same states as they are in the node in Z where $U(p_0^1) = U(p_1^2) = s_i$ for the first time. Intuitively, there are two reasons for this: first, for every position i, the same automaton reaches the position i first in Z and Z'. Second, what the second automaton sees is not influenced by how far ahead is the first automaton. This concludes the proof of Lemma 1.

Proof (of Theorem 1): Lemma 1 shows that a special case of the concurrent reachability for simple MAs can be reduced to reachability in MAs. We will therefore need to generalize the proof of the lemma — first, from concurrent reachability in MAs to concurrent reachability for general MAs and second,

from reachability to the condition on initial and final lists from the statement of this theorem.

We start by explaining briefly how the proof can be generalized from the case of simple MAs to general MAs. The difference between simple MAs and MAs is that MAs have lagging pointers and MAs can modify the *next* field of the vertices in a list.

Let us consider lagging pointers. First, we explain in detail how the assumption that there are no lagging pointers was used in the construction of the automaton LS. Let us suppose that the automaton A_1 has one lagging pointer, and the automaton A_2 has no lagging pointers. Now suppose that the current pointer of A_1, the lagging pointer of A_1 and the current pointer of A_2 point to the same vertex v in the list. Furthermore suppose that A_1 executes a move-transition and is one step ahead. The situation after A_1 executed the move-transition is depicted on Figure 2, part (d). In the figure, the pointer $A_1.p_0$ is A_1's current pointer, and the pointer $A_1.p_1$ is A_1's lagging pointer. In this situation, the lock-step construction seems not to work, as the current pointer of A_2 cannot advance, because we do not know whether A_1 will write to the vertex v. The solution is that LS will guess what A_1 will write to v (and read from v) before A_2 performs a move transition. Due to the definition of MAs, the sequence of what A_1 and A_2 write is bounded. In this way, any run of $A_1 \parallel A_2$ can be simulated; note that as the run continues, the guess needs to be checked, that is, it needs to be checked whether A_1 indeed writes what was guessed. LS will store the guessed sequence in the Q_B part of the state (the sequence of values from Σ) as well as in the data variables (the sequence of values from D). The situation is depicted in Figure 2, part(e). When LS moves the current pointer of A_2, $A_2.p_0$, it needs to keep the guessed sequence (of length 2 in the figure) logically attached to the node to which the lagging pointer of A_1, $A_1.p_1$, is pointing. The construction can be generalized to the case when both automata have multiple lagging pointers. The automaton LS guesses the whole sequence of read and writes to v that happens before the last lagging pointer leaves the vertex v.

We now present how the proof of Lemma 1 can be extended to cover MAs that perform pointer modifications in the heap and extension from same-vertex-reachability to concurrent reachability.

Let us consider MAs that perform pointer modifications of the heap, that is, they change the value of the *next* field of a node. We describe the proof on two examples: inserting and deleting nodes into the heap. The solution in this cases is again based on nondeterministic guessing. For inserting, we first note that the number of vertices that can be inserted into the list at a lagging pointer is bounded by the number of pointers (including input/output pointers) the automaton has. Thus the amount of information to be guessed is bounded, and can be stored to be verified. As for deleting, let us consider one way an automaton can delete nodes from the list. It can redirect the *next* field of the node pointed to by a lagging pointer to the current pointer. Such a run can be simulated by LS by nondeterministically guessing where the deletion starts. More precisely, let us suppose that the lagging pointer of A_1 as well as the current pointer of

A_2 points to a node v, and that the current pointer of A_1 is one step ahead, i.e. it points to a node v' such that $next(v) = v'$. If LS now guesses that A_1 will in the future course of the run delete nodes by redirecting the *next* field of its lagging pointer, and A_2 will advance its current pointer only after this deletion, LS can still advance the current pointer of A_2, but not process the nodes that are to be deleted by A_1. The automaton LS will check that the deletion has in fact occurred.

We now briefly explain how can we generalize from the proof for same-vertex reachability of a pair (q_1, q_2) (as in Lemma 1) to prove concurrent reachability (as defined by the concurrent reachability problem). The difference between the two notions is that concurrent reachability does not require the current pointers of the two automata point to the same vertex in the heap. The simulation by the CA automaton is extended as follows: the automaton proceeds as before for the part of the list that is traversed by both automata. However, it can nondeterministically decide to stop simulating one of the automata. In this way, it simulates runs where the current pointers of the two automata point to a different nodes in the heap when they A_1 reaches q_1 and A_2 reaches q_2.

In the course of this proof, we have for simplicity of discussion focussed on the reachability question. As simple inspection of the proof shows that the constructed automaton LS has the same effect on the resulting list and values of output pointers as the parallel composition of A_1 and A_2.

This completes the proof of Theorem 1.

Let $A = (Q, P, DV, T, q_0, F, head, O)$ be a method automaton. Let q be a state in Q. The *method automaton reachability problem* is to decide whether there exist a well-formed method input $(L, ionodes)$, a list L', and a valuation of pointer variables U such that in the transition system $[\![A(L, ionodes)]\!]$, the node (L', q, U) is reachable from the initial node.

Theorem 2. *The method automaton reachability problem is decidable. The complexity is polynomial in the number of states of the automaton, and exponential in the number of its pointer and data variables.*

Proof. Let A be a method automaton (MA). We say that a state q is reachable in A iff there exist a well-formed input $(L, ionodes)$, a list L' and a valuation of pointer variables U and of data variables dv, such that (L', q, U, dv) is reachable in $[\![A(L, ionodes)]\!]$ from the initial state.

For simplicity, we present the rest of the proof for the case of automata which do not have data variables. Extending to the general case is no difficult.

Note that the size (number of nodes) of the transition system $[\![A(L, ionodes)]\!]$ is potentially infinite. There are two "sources of infinity": the first is the unboundedness of the heap, and the second is the unboundedness of the data domain.

The structure of the proof is as follows. First, we construct a transition system T_l (l for local), such that q is reachable in A if and only if it is reachable in T_l. In T_l, we remove one "source of infinity", namely the unbounded heap. Second, we construct a finite transition system T_f (f for finite), such that q is reachable

in T_l if and only if it is reachable in T_f. Reachability in finite transition system is decidable, which enables us to conclude the proof.

Construction of T_l. The nodes of T_l are called local nodes. A local node is a tuple $(q, U_l, \mathit{same\text{-}value}, \mathit{next}_l)$, where q is a state in Q, U_L is a function from the set of pointers R to $\Sigma \times D$, and *same-value* and next_l are relations on R. The main difference between T_l and A is that there are no pointers in T_l.

There are two main ideas behind the construction of T_l. First, the local nodes contain enough information to evaluate the guards, and this information can be updated as a result of performing actions. Second, as we are interested in reachability, we can replace the list and input by considering nondeterministic transitions. Thus the main difference between T_l and and nodes in $[\![A(L, \mathit{ionodes})]\!]$ is that local nodes do not contain the list L.

We will use the following notation in the rest of the paper. Let f be a partial function. The expression $f[a \leftarrow b]$ denotes a function that is the same as f at all points except a, where it returns the value b.

In the interest of space, we do not give the full definition of the transition relation in T_l, we only present two illustrative examples. First, let us consider a method automaton transition given by the tuple $(q, \mathit{next}(p_4) = p_3, p_4 := p_3, q')$. This tuple induces a transition in T_l as follows: a local node $(q, U_l, \mathit{same\text{-}value}, \mathit{next}_l)$ has a transition to $(q', U_l', \mathit{same\text{-}value}', \mathit{next}_l')$ if $\mathit{next}_l(p_4) = p_3$, $U_l' = U_l[p_4 \leftarrow U_l(p_3)]$, and the relations *same-value* and next_L' get updated to reflect that p_4 is now equal to p_3.

Second, let us consider a method automaton transition given by the tuple $(q, \mathit{true}, p_0 := \mathit{next}(p_0), q')$. This tuple induces a transition in T_l as follows: a local node $(q, U_l, \mathit{same\text{-}value}, \mathit{next}_l)$ has a transition to $(q', U_l', \mathit{same\text{-}value}', \mathit{next}_l')$ if $U_l' = U_l[p_0 \leftarrow (s, d)]$, where s is in Σ, d is in D and the relations *same-value* and next_L' get updated to reflect that p_0 advanced one step. Note that s and d are unconstrained.

Note that it is here, in the construction of the transition function of T_l, that the restriction R1 is used. The fact that the pointer p_0 is the only one that can move forward through the list, and that it cannot back implies that the values encountered by p_0 do not need to be stored, except for those pointed to by lagging pointers.

A state q is reachable from T_L if there exists an initial node k_0 and a function U_l and relations *same-value* and next_l, such that the node $(q, U_l, \mathit{same\text{-}value}, \mathit{next}_l)$ is reachable from k_0. We can prove that a state q is reachable in T_l iff it is reachable in A. Both directions are proven by straightforward induction on the length of the path witnessing that q is reachable.

Construction of T_f. Note that T_l is still an infinite state system, as its nodes store values from the unbounded domain D. We construct a finite-state system T_f that abstracts away the value from D and keeps only the equality and ordering information. More precisely, the function $U_l : R \to \Sigma \times D$ is replaced by a function $U_\Sigma : R \to \Sigma$ and by an order on equivalence classes on R. The nodes in

T_f are tuples $(q, U_\Sigma, PV, \textit{same-value}, \textit{next}_l)$, where PV is an order on equivalence classes on R. A node in T_f represents a set of nodes in T_l. For example, PV might be $(p_2 < p_0 = p_1)$, which represents a set of nodes which includes a node where $U_l(p_0) = (s_0, 3)$, $U_l(p_1) = (s_1, 3)$, $U_l(p_2) = (s_2, 2)$ for some $s_0, s_1, s_2 \in \Sigma$ (in this example we assume $D = \mathbb{N}$).

Let α denote the abstraction function from nodes in T_l to nodes in T_f. The transitions between nodes in T_f are defined as follows: for two nodes of T_f, t_1 and t_2, there is a transition from t_1 to t_2 iff there exist two nodes of T_l, l_1 and l_2, such that $\alpha(l_1) = t_1$ and $\alpha(l_2) = t_2$, and there exists a transition from l_1 to l_2 in T_l.

We can prove that a state q is reachable in T_l iff it is reachable in T_f. Given the definition of the transition function, it can be easily seen (and proven by induction), that if q is reachable in T_l, then it is reachable in T_f.

For the other implication, we use a technique similar to the proof of Theorem 1 in [2]. First note that for two nodes t_1 and t_2 in T_f, it is not the case that for all nodes l_1 and l_2 of T_l, such that $\alpha(l_1) = t_1$ and $\alpha(l_2) = t_2$, there exists a transition from l_1 to l_2. For a counterexample, consider a node t_1 in T_f where PV specifies that $p_1 > p_2$. Furthermore, suppose that there is a transition tuple in the method automaton with an action $p_0 = \textit{next}(p_0)$ that induces a transition (without changing values of variables other then p_0) to a node t_2 where PV is $p_1 > p_0 > p_2$. Now consider a local node where $U_l(p_1) = (s_1, 6)$ and $U_l(p_2) = (s_2, 5)$. Note now that l_1 cannot transition to any state that would correspond to the order on pointer variables required by t_2, simply because there is no node between 5 and 6.

In a key part of the proof of this implication (reachability in T_f implies reachability in T_l) is to show that for t_1 and t_2, nodes in T_f, if t_2 is reachable from t_1, then there exist l_1 and l_2, nodes in T_l, such that $t_1 = \alpha(l_1)$ and $l_2 = \alpha(t_2)$, and such that such that l_2 is reachable from l_1. The main idea for proof by induction is that we can choose l_1 in such a way that the gaps between values are large enough. More precisely, if (1) t_1 requires that e.g. $p_1 > p_2$ for two pointer variables p_1 and p_2 and (2) p_2 is reachable from p_1 via a path of length k, then it is sufficient to choose p_1 such that $U_l(p_1) = (s_1, d_1)$, $U_l(p_2) = (s_2, d_2)$, and $d_1 - d_2 > 2^k$.

Note that we can tighten the analysis above by noting that the only "new" values come from *move transitions*, i.e. transitions where the action is $p_0 := \textit{next}(p_0)$ - thus it is sufficient to consider the number of move transitions, and not the length of the path.

Complexity The proof yields an algorithm for deciding reachability of a state q. The algorithm checks reachability of q in the finite-state transition system T_f.

The number of states in T_f depends polynomially on the number of states of the method automaton and exponentially on the number of its pointer variables (as the order on equivalence classes on pointer variables is tracked). Reachability in a finite state system can be decided in the time polynomial in the number of states of this system.

This completes the proof of Theorem 2.

3.2 Deciding linearizability

The following theorem is the main result of the paper.

Theorem 3. *The two method linearizability expression problem is decidable.*

Proof. The proof is by reduction to reachability in method automata, which in turn reduces (by Theorem 2 to reachability in a finite state system. We show how linearizability of $A_1 \parallel A_2$ can be reduced to reachability in a method automaton $\mathit{LinCheck}(A_1 \parallel A_2)$. The automaton $\mathit{LinCheck}(A_1 \parallel A_2)$ essentially simulates of the parallel composition $A_1 \parallel A_2$, and both possible linearizations, $A_1 \,;\, A_2$ and $A_2 \,;\, A_1$. $\mathit{LinCheck}(A_1 \parallel A_2)$ reaches an error state if there is an unlinearizable execution of $A_1 \parallel A_2$.

First, we use Theorem 1 to show that instead of simulating $A_1 \parallel A_2$ (resp. $A_1 \,;\, A_2$ and $A_2 \,;\, A_1$), one can simulate the method automaton $LS(A_1 \parallel A_2)$ (resp. $LS(A_1 \,;\, A_2)$ and $LS(A_2 \,;\, A_1)$).

Second, we show how $\mathit{LinCheck}(A_1 \parallel A_2)$ can simulate the three method automata $LS(A_1 \parallel A_2)$, $LS(A_1 \,;\, A_2)$, $LS(A_2 \,;\, A_1)$) on the same input heap and reach an error state if there is an unlinearizable execution of $LS(A_1 \parallel A_2)$. The key idea is once again that the current pointers of the three automata can advance in a lockstep manner. The reason is much simpler in this setting than in the proof of Theorem 1 – here the three automata do not communicate at all (the only reason we are simulating the three automata together is that they run on the same input heap). $LS(A_1 \parallel A_2)$ reaches an error state e.g. if it can conclude that a particular position in the output list for $LS(A_1 \parallel A_2)$ will be different from that position in output lists of both $LS(A_1 \,;\, A_2)$ and $LS(A_2 \,;\, A_1)$.

We have reduced linearizability to reachability in method automata. We can thus conclude by using Theorem 2.

Construction of $\mathit{LinCheck}(A_1 \parallel A_2)$ Let us consider the expression $E = A_1 \parallel A_2$. Let E_1 and E_2 be two expressions describing the two linearizations that are a priori possible, i.e. $E_1 = A_1 \,;\, A_2$ and $E_2 = A_2 \,;\, A_1$. We consider three automata: $LS(E)$, $LS(E_1)$, and $LS(E_2)$.

Recall (from the proof of Theorem 1) that the set of states of the automaton $LS(E)$ is the product of sets Q_1 and Q_2 and Q_B, where Q_i is the set of states of A_i (for $i = 1, 2$), and Q_B allows storing necessary bookkeeping information. We have that apart from the bookkeeping information, which is different in $LS(E)$, $LS(E_1)$, and $LS(E_2)$, the three automata carry the information about states of A_1 and A_2.

We will construct a method automaton $\mathit{LinCheck}(E)$ that simulates the three automata $LS(E)$, $LS(E_1)$, and $LS(E_2)$. $\mathit{LinCheck}(E)$ reaches an error state if it determines that one of the following conditions hold:

- after all three automata has finished processing the node at position t of their list, the data at the node at position t in the list processed by $LS(E)$ is different from both the data in the list processed by $LS(E_1)$ and $LS(E_2)$

– after all three automata has finished processing the list, at least one of the value in the input/output nodes of $LS(E)$ is different from both the corresponding output value $LS(E_1)$ and $LS(E_2)$

We now need to explain how the automaton $LinCheck(E)$ can simulate the three automata operating on three different lists. (Note that at the beginning, the lists are the same. However, during the computation, they can be different.) As $LinCheck(E)$ processes the list, it keeps the data in the nodes pointed to by the three simulated automata in dedicated data variables. The number of the data variables is bounded - this is because of the following two facts: First, each of the three automata has a bounded number of pointer variables. Second, the current pointers can be kept close together (the difference in positions is at most one), because the three automata do not communicate — there is no channel through which they can exchange information. (We simulate them in parallel only because they process the same list.)

The automaton $LinCheck(E)$ needs to keep bookkeeping information about the data variables representing the lists being processed by the three automata. The amount of information is finite, and can be stored in the state of the automaton. We do not describe all the details, we mention only that that for each position of the list pointed to by at least one of the three automata, the automaton keeps track which automata point to it. After the last automaton leaves a position, the automaton checks that the value left begin by $LS(E)$ is the same as either the value left behind by $LS(E_1)$ and the value left behind by $LS(E_2)$. Once the automaton sees the difference in the output list between $LS(E)$ and one of $LS(E_i), i \in \{1, 2\}$ it stops simulating $LS(E_i)$. If both $LS(E_1)$ and $LS(E_2)$ are stopped, the automaton enters an error state. The values in IO nodes are similarly checked at the end of the execution.

Extension to general composition of method automata Let us consider *method expressions* that compose a finite set of methods sequentially and in parallel. Theorem 3 extends to this setting: it is decidable whether a method expression is linearizable. Note that given a method expression E, the number of automata $LinCheck(E)$ simulates grows exponentially with the number of methods in E, as $LinCheck(E)$ simulates all possible linearizations of E.

Undecidable extensions The following theorem shows that the restriction OW is necessary for decidability.

An *UWMA* is a method automaton where the OW restriction has been lifted. (UWMA is an abbreviation for method automaton with unrestricted writes.)

Theorem 4. *The two method linearizability problem is undecidable is undecidable for UWMAs.*

Proof. The proof is based on reduction from concurrent reachability for UWMAs, which is undecidable. The main idea is that if a pair of states (q_1, q_2)) is reachable, then the two automata will erase their work, thus making the execution linearizable. The fact that reachability is undecidable follows from Lemma 2.

The proof of this theorem also implies that if we lift the restrictions on how the pointer variables are updated, the two method linearizability problem becomes undecidable as well.

Let A_1 and A_2 be two UWMAs. Let q_1 be a state of A_1 and let q_2 be a state of A_2. The *concurrent reachability problem* is to decide whether there exist a well-formed method input $(L, \mathit{ionodes})$ such that in the transition system $T_P(A_1, A_2, L, \mathit{ionodes})$, a node where the state component is (q_1, q_2) is reachable.

Lemma 2. *The concurrent reachability problem for UWMA is undecidable.*

Proof. We reduce the halting problem of an arbitrary Turing machine $\mathcal{M}$ to the concurrent reachability problem in a system $(A_1||A_2)$, where A_1 and A_2 are UWMAs. The list on which $(A_1||A_2)$ operates encodes a string in the obvious way. The machine $(A_1||A_2)$ accepts iff this string is of the form $w = b\#w_1\#w_2 \ldots \#w_n$, where $b \in \{1, 2\}$, the w_i's are successive configurations of $\mathcal{M}$; w_1 and w_n are respectively the initial and final configurations of $\mathcal{M}$; and each w_i has the same length.

On an input list, the machines operate in parallel as follows. Both A_1 and A_2 maintain lagging pointers to the first node in the list and read and write its data. This node serves as a "shared-memory communication channel" between A_1 and A_2. We refer to the fragments of the input list (viewed as strings) between successive occurrences of # as *epochs*.

Initially, if the Σ-field of the first node of the list has the value b, then A_b moves first (w.l.o.g., let this be A_1). Now A_1 iterates through the list until it reaches the second epoch—in the process it checks that the first epoch represents an initial configuration of $\mathcal{M}$. At this point, it is the turn of A_2 to move. From now on, A_1 and A_2 take turns in reading the list one symbol at a time and comparing the symbols read in corresponding turns (communication is performed through the shared channel, which also stores a bit that determines whether it is currently the turn of A_1 or A_2). It is easy to see that if A_1 and A_2 finish reading the epochs w_1 and w_2 in corresponding turns, then they can check that these are successive configurations of $\mathcal{M}$. If they are not, A_1 changes state to an error state. At any point, the control state of A_1 is q if the last epoch of $\mathcal{M}$ that it read completely represents an accepting configuration. This completes the reduction.

4 Experimental Evaluation

4.1 Examples

This section presents a range of concurrent set algorithms. They all share the basic idea that a set is implemented as a linked list. The main difference comes from the synchronization style they use. The interface the concurrent set provides consists of three methods: `contains`, `add`, and `remove`. The algorithms that we analyze represent a set as a linked list whose nodes are sorted by their keys (the sortedness property enables efficient detection of an absence of a key). Each

key occurs at most once in the set. The list contains two sentinel nodes, head and tail, which are never removed, added, or searched for, and their keys have minimum and maximum possible values.

The examples differ in the synchronization techniques they use. The techniques used include *fine-grained locking*, *optimistic synchronization*, *lazy synchronization*, and *lock-free synchronization*:

- A natural approach to synchronization in concurrent lists is *fine-grained locking*, where each vertex of the list is locked separately. For operations such as insertion or deletion of nodes into a list, locking a single vertex at a time is not sufficient. Therefore we use "hand-over-hand" locking, where during the traversal, a node is unlocked only after its successor is locked. At the point in the list where insertion or deletion is to be performed, the two successive locked nodes are kept locked.
- A problem with fine-grained locking is that threads that modify disjoint parts of the list can still block each other. *Optimistic* synchronization is an attempt to remedy the problem. Here, threads do not acquire locks as they traverse the list. Instead, when a thread locates (and locks) the part of the list which it wants to modify, the thread needs to re-traverse the list to make sure the locked nodes are still reachable from the head of the list. The re-traversal is called the *validation* phase.
- The *lazy* synchronization algorithm improves the optimistic one in two main aspects. First, the methods do not need re-traversal. Second, `contains` (commonly thought to be the most used method), do not use locks anymore. The most significant change to the synchronization is that the deleted nodes are marked.
- A method is called *lock-free* if a delay in one thread executing the method cannot delay other threads executing the method. If a method uses locks, it is not lock-free. It is possible to make all methods of the set data structure lock-free using the `compareAndSet` primitive. For example, we can write a lock-free algorithm for deletion of set elements where the only atomic primitive is one that, in one atomic step, rewires the outgoing pointer from a node and marks the deleted nodes. This primitive can be implemented by the `compareAndSet` operation in Java.

4.2 Implementation

The CoLT tool chain can be seen in Figure 3. The input to the tool is a Java file and two method names. The methods in the Java file are parsed into method automata. The second tool in the toolchain is a lockstep scheduler. It takes the two method names and the corresponding method automata, and produces a finite-state model using the (simplified) construction from the proof of Theorem 3. The finite state model is then checked by the SPIN [11] model checker. If SPIN returns false, it also returns a counterexample trace that describes the execution that is not linearizable. CoLT then returns a visual representation of the trace for inspection by the programmer.

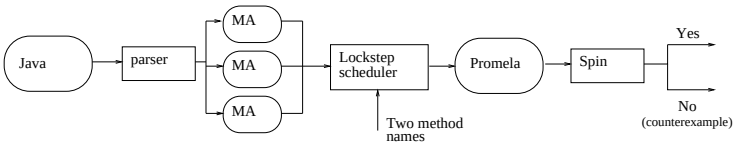

Fig. 3. CoLT Toolchain

In the rest of this subsection, we summarize the main issues in translating the Java implementations of concurrent data set algorithms to method automata.

Encoding Java Concurrency primitives We have seen in Section 2 how locks can be encoded. Here we explain in more detail about the non-blocking primitives. The Java Concurrency library includes a class called `AtomicMarkableReference` which implements `compareAndSet` and `attemptMark` operations. The class has a reference field (call it `rf`) and a mark field (call it `mf`). The method `public boolean compareAndSet(T expectedReference, T newReference, boolean expectedMark, boolean newMark)` tests whether `rf` is equal to `expectedReference` and `mf` is equal to `expectedMark`, and if it is so, sets these fields to respectively `newReference` and `newMark`. The method `public boolean attemptMark(T expectedReference, boolean newMark)` tests whether `rf` is equal to `expectedReference`, and if it is so, sets the `mf` field to `newMark`.

Note that it is the *next* pointer of a vertex that is implemented using `AtomicMarkableReference`. In our model of concurrent data lists, this is modeled using the source vertex of the pointer (recall that every vertex has at most one outgoing pointer). The method `compareAndSet` is modeled by the following transition tuple: $(q, \mathit{flag}(p) = \texttt{expectedMark}\ \mathit{and}\ p = \texttt{expectedReference}, (\mathit{flag}(p) = \texttt{newMark}; p = \texttt{newReference}, q'))$.

Implementation assumptions

In order to simplify the implementation, we made several assumptions on the programs that are handled. That is, we implement only a simplified version of the construction from the proof of Theorem 3, as the full consuction is not needed for the concurrent set implementations.

– *Phases approach* We implemented only a simplified version of the construction from the proof of Theorem 3. It relies on the fact that all the examples we considered work in two phases: in the first phase, a list is traversed without modification (or with limited modification in the case of the lock-free algorithm) and in the second phase, the list is modified "arbitrarily". This simplifies the implementation by reducing the amount of nondeterministic guessing that is necessary, but relies on annotations to identify the phases.
In more detail: we can exploit some information about the general structure of the algorithms. First, the algorithm traverse the list, looking for the part that needs to be modified (*find* phase). Then the required modifications are made (*modify* phase). We observed that during the find phase, the method automaton do not change the list structure. It might lock/unlock nodes,

but the next field is never changed. The automaton simply traverse the list. During the modify phase the automaton do not move anymore, but performs modifications on the list. This additional knowledge can be used to simplify the analysis. During the lockstep construction the algorithm has to guess what will happen. In practice it means that the model checker needs to do a case split. Knowing in which phase is the algorithm allows us to dramatically reduce the number of guesses we have to make.

Optimistic algorithms have a further validate phase between the find and modify phase, while other algorithm can also retry (i.e. restart from the beginning).

- CoLT currently assumes that at the start of execution of the two programs, the list is sorted. This is standard assumption for concurrent set implementations [9].
- CoLT currently assumes that the data value in a node is not changed. Note that this holds for concurrent set implementations — these insert and delete nodes, but do not change their values.

Validate Method automaton can only traverse the list monotonically from left to right. The optimistic algorithm described above traverses the list twice, once to find the required node and lock it and the second time to validate that the locked node is still accessible from the head of the list. We implemented a heuristic to extend the scope of our tool to cover the optimistic algorithm. For this heuristic, we require annotations in the code that mark the first and the second traversal. Given these annotations, the tool can decompose each method to two method automata, one that finds and locks the node and one for validation. A construction similar to sequential composition of these two automata is then used to model an optimistic method.

Retry The core traversal of fine-grained, lazy and lock-free algorithms satisfies the restriction of traversing the list monotonically from left to right. The only caveat is that in many cases, when an operation such as insertion or deletion fails, the method aborts and "retries" by setting all pointers to the start vertex, and our restriction rules out such a retry. We emphasize that the retry behavior is very different from the validate behavior of the optimistic algorithm. The aborted executions in the case of fine-grained, optimistic, and lazy method, have no effect on the heap. In the case of the lock-free method, the effect of the aborted execution is limited and simply defined. We implemented a simple heuristic to deal with retry behavior. The heuristic produces a method automaton that behaves as follows: whenever such a retry would occur execution is not simulated further. One can easily prove for all algorithms we have considered that if the parallel composition of method automata constructed in this way is linearizable, so are the original method. Similarly, an unlinearizable execution of parallel composition of method automata constructed in this way corresponds to an unlinearizable execution of the original methods.

Linearization points Our tool enables programmers to specify linearization points. Specifying them is not necessary, but leads to reduction of the search

```
public boolean add(T item) {
  ... //initialisation
  pred.lock();
  try {
    Node curr = pred.next;
    L1: curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        L2: curr.lock();
      }
      stage = modify; ////
      if (curr.key == node.key)
        return false;
      ... //actually add the node
      return true;
    } finally { curr.unlock(); }
  } finally { pred.unlock(); }
}
```

The method `add` linearizes when it reaches:

- `L1` iff `curr.key >= key`
- `L2` iff `curr.key >= key`

If `curr.key > key` then `item` is added between `pred` and `curr`. The locking prevents any other method to interfere with the process.
If `curr.key == key` item is already present in the set and the method returns.

Fig. 4. Linearization points of the method `add` (fine-grained locking)

space, and thus to improving memory consumption and running time of the experiments.

We take advantage of *linearization points* [10]. A linearization point for some method is an instant between its invocation and its response where the effect seems to occur atomically.

Given an complete history H, where each method call has a linearization point, an equivalent sequential history H' can be constructed using the linearization points ordering. Fortunately, there are methods which linearization point depends only on the current method's state and the list. However, there also are methods with more complicated linearization points, or where the linearization point is not even the method itself.

Let consider the `add` method in the fine-grained locking implementation of sets as linked lists [9, Chapter 9.5]. Assuming that all the other methods that traverse the lists use the same synchronization scheme, we can infer the linearization points presented in Figure 4.

Suppose we are given two methods $A1$, $A2$ with their linearization points $L1$, $L2$. We want to prove that all histories are linearizable. We divide the histories into those where $L1$ happens before $L2$ and those where $L2$ happens before $L1$. By the definition above, When $L1$ precedes $L2$ in history H, there is a sequential execution of $A1; A2$ which is equivalent to H. More formally: *for all histories h, h is linearizable* is equivalent to *for all histories h, we have that (if $L1$ precedes $L2$ in h, then h is equivalent to a history of $A1; A2$) and (if $L2$ precedes $L1$ in h, then h is equivalent to a history of $A2; A1$).*

In practice we check two smaller systems for linearizability: First, we simulate $A1 \parallel A2$ with only $A1; A2$ and we force $L1$ to happen before $L2$. Then, we simulate $A1 \parallel A2$ with $A1; A2$ and $L1$ before $L2$. If one of the two test fails. This analysis is sound, even if the linearization points are incorrect. However the

completeness rely on the correctness of the linearization points. Given incorrect points the analysis might says that correct methods are not linearizable.

The benefit of this construction is that we need to simulate only two R2MAs at the same time instead of three. The gains in term of memory consumption (and time) are substantial. With the help of this technique we have been able to prove the correctness of instances that previously ran out of memory.

To force $L2$ to happen after $L1$ we simply tell the model checker to discard the runs when $L2$ occurs before $L1$. We must be careful not to discard too many runs because of our lockstep setting. The problem arises when $A2$ cannot move without linearizing, but $A1$ needs to move ahead to linearize and the lockstep construction prevents them from being separated. The solution comes from another observation about the algorithms. All the methods we analyze stop traversing the list when they reach their linearization point. We use this fact to let $A1$ proceed without lockstep. Since $A2$ stops, we do not need to remember the node between the two automaton.

Manual processing

In the current prototype version, the following simple processing is done manually.

- Some of the methods we considered take an integer value as parameter and create a node with these data value. Since our model of method automata does not support node creation, we replaced this by passing a node as a parameter. Note that this transformation can be easily automatized.
- In some methods, a command where the reference to next field appears on both sides (e.g. `prev.next = curr.next`) was replaced by two simpler commands `aux = curr.next; prev.next = aux`. This will be corrected in a future version of CoLT .

4.3 Experiments

We evaluated the tool on the fine-grained, optimistic, lazy, and lock-free implementations of the concurrent set data structure. The Java source code was taken from the companion website to [9]. All the experiments were performed on a server with an 1.86GHz Intel Xeon processor (with 8 cores) and 32GB of RAM.

The results of the experiments are presented in Table 1. Here, the first column lists the analyzed algorithm, the second column names the analyzed parallel composition of methods, the third column contains the number of lines of code and the number of pointer variables of the first method, and the fourth column gives the same information for the second method. The fifth column indicates whether linearization points were used. The sixth column lists the maximum depth reached in the exploration of the finite state graph. The seventh and eight columns indicate the memory and time consumption. The ninth column indicates whether the method expression was linearizable.

First, to evaluate our analysis on implementations of fine-grained locking algorithms, we ran the remove method in parallel first with the contains method,

Algorithm	Methods	M_1 loc/pts	M_2 loc/pts	Lin. points	Depth	Mem (MB)	Time (s)	Res
Fine-grained	remove ‖ contains	29/2	23/2	No	157	10.2	0.85	Yes
Fine-grained	remove ‖ remove	29/2	29/2	No	141	8.3	0.46	Yes
Fine-grained	remove ‖ add	29/2	26/2	No	303	18.1	2.4	Yes
Optimistic	add ‖ remove	40/3	38/3	No	110	37.6	5.86	Yes
Optimistic	contains ‖ contains	30/3	30/3	No	150	37.6	6.9	Yes
Optimistic	remove ‖ remove	38/3	38/3	No	130	36.2	6.35	Yes
Lazy	remove ‖ remove	36/3	36/3	No	164	20.1	2.68	Yes
Lazy	remove ‖ add	36/3	34/3	No	164	26.3	3.51	Yes
Lazy	contains ‖ remove	36/3	6/1	No	136	13.2	1.28	Yes
Lazy	remove_1 ‖ add_1	36/3	34/3	No	137	24.2	3.17	No
Lazy	remove_2 ‖ remove_2	34/3	34/3	No	143	17.9	2.18	No
Lock-free	contains ‖ contains	9/2	9/2	No	98	6.4	0.25	Yes
Lock-free	remove ‖ remove	34/3	34/3	Yes	95	77.6	8.08	No
Lock-free	remCorr ‖ remCorr	34/3	34/3	Yes	268	1908.3	605	Yes
Lock-free	add ‖ remCorr	35/3	34/3	No	?	out	?	?
Lock-free	add ‖ remCorr	35/3	34/3	Yes	267	1550.3	577	Yes
Lock-free	add ‖ contains	35/3	9/2	No	400	18984.1	5700	Yes

Table 1. Experimental results

second with itself, and third with the add method. The memory consumption was under 20MB and the running time under 3s in all cases.

Second, we analyzed the optimistic implementations. The Java file was annotated to enable using the heuristic defined in the previous subsection. CoLT validates the optimistic implementations in under 40MB of memory for every case. While implementing the heuristic, some of the basic components of the tool were rewritten to be more efficient; hence, the time and memory consumption of these results are not directly comparable with the other algorithms.

Third, we analyzed lazy-synchronization implementations. The tool CoLT verified linearizability in the same three cases as in the Fine-grained Locking algorithm. We used the tool to analyze modifications of the add and remove methods suggested as exercises in [9]. One exercise suggests simplification of the validation check (methods remove_1 and add_1), the other asks not using one of the locks (method remove_2). We used the tool on parallel composition of remove_1 and add_1, and on parallel composition of remove_2 with itself. In both cases, CoLT reported these compositions to not be linearizable and produced counterexamples. The visual representations of the counterexamples are available on the tool's website [5]. Note that for both fine-grained and lazy synchronization algorithm, we did not need linearization points, as the running times and memory consumption were within reasonable bounds.

Fourth, we considered lock-free implementations of concurrent set algorithms. The tool verified linearizability of parallel composition of the contains method with itself. CoLT also found that the parallel composition of remove with itself

is not linearizable. This is a known bug, reflected in the online errata for [9]. The the counterexample is available online [5]. We then corrected the bug according to the errata (method removeCorr —short for removeCorrected). The tool showed that in this case, the parallel composition of remove with itself is linearizable. We observe that the memory usage in this example is more substantial, for example for the parallel composition of the corrected remove method with the add method, the tool needs 2GB of memory, even when the linearization are provided. The tool runs out of memory if linearization points are not provided. The reasons for increased memory consumption of the Lock-free algorithm compared to the other two algorithms are that the Lock-free algorithm has an inner loop and the input list can contain nodes marked for deletion, thus increasing the search space for analysis.

5 Conclusion

Summarizing, the main contributions of the paper are two-fold: first, we prove that linearizability is decidable for a model that captures many published concurrent list implementations, and second, we showed that the approach is practical by applying the tool to a representative sample of Java methods implementing concurrent data sets.

References

1. P. Abdulla, M. Atto, J. Cederberg, and R. Ji. Automated analysis of data-dependent programs with dynamic memory. In *ATVA*, pages 197–212, 2009.
2. R. Alur, P. Černý, and S. Weinstein. Algorithmic analysis of array-accessing programs. In *CSL*, pages 86–101, 2009.
3. M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, pages 7–16, 2006.
4. S. Burckhardt, R. Alur, and M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.
5. P. Černý, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. CoLT website. http://www.ist.ac.at/∼cernyp/colt.
6. R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, 2006.
7. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, 2004.
8. T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.
9. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier Inc., 2008.
10. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
11. G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
12. D. Lea. The java.util.concurrent synchronizer framework. In *CSJP*, 2004.

13. M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.
14. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, 2004.
15. M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, 2007.
16. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP*, pages 129–136, 2006.
17. E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.