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 vertex stores 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 [13] or the Intel Threading Building Blocks support the development of efficient multi-threaded 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 [10] 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

* This research was partially supported by NSF grants CCF 0905464 and CAREER Award 0953507 and by the Gigascale Systems Research Center.

published algorithms [5, 16]. 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 introduce 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 is the decidability of linearizability of method expressions. A method expression allows to combine a fixed number of method automata using sequential and parallel composition. Linearizability [11] is a central correctness requirement for concurrent data structure implementations. Our algorithm takes as input a precondition I in addition to a method expression E and checks that all executions of E starting from a heap that satisfies I are linearizable. For example, given two methods to insert and delete elements of a list, our decision procedure can check whether every execution of the parallel composition of the two methods that starts from a sorted list is linearizable. Our decidability proof is developed in two steps.

First, we show how linearizability of a method expression E can be reduced to a reachability condition on a method automaton A. The automaton A simulates E and all of its possible linearizations. For instance, if E is the $A_1 \parallel A_2$ then the possible linearizations are $A_1; A_2$ and $A_2; A_1$. The principal insight in the construction of A is that the automata in E can proceed through the list almost in a lock-step manner. This result assumes that the methods we analyze are deterministic when run sequentially. Note that the assumption is satisfied by all the implementations we analyzed.

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 need 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. The tool implements only the case of the parallel composition of two method automata. 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 [10] 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 tool allows the user to provide linearization points, which reduces search space significantly. 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 of machine-checked manual proofs of correctness exists in the literature [7, 19]. We know of two model checking approaches ([14, 20]). Both these works consider only a bounded heap, whereas our approach does not impose any bound on the size of the heap. Static analysis methods based on shape analysis [3, 18] require user-specified linearization points. In contrast, our approach does not need linearization points. The user has an option to provide linearization points to improve performance. The experiments show that this is not necessary at all for e.g. the fine-grained locking, and lazy list implementations. Furthermore, shape analysis approaches are sound, but not complete techniques, whereas our algorithm is both sound and complete for a bounded number of threads. 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 [17, 4] 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.

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 $(D, =, <)$ and 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.

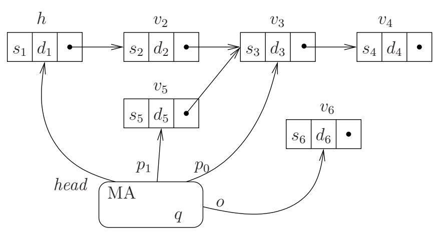

Fig. 1. Singly-linked data heap and a method automaton

The structure L can be naturally viewed as a labeled graph with edge relation *next*. L is *well-formed* if this graph has no cycles reachable from h. For each well-formed heap L as above, we define a finite data word (over $\Sigma \times D$) represented by the list starting at h. Figure 1 shows a singly-linked data heap with six vertices that contain values from Σ and D which define the data word $(s_1, d_1)(s_2, d_2)(s_3, d_3)(s_4, d_4)$.

Method automata: Syntax A *method automaton* is a tuple $(Q, P, DV, T, q_0, F, \mathit{head}, O)$, 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 set of transitions (as explained below), $q_0 \in Q$ 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.

A method automaton operates on a singly-linked data heap $L = (V, \mathit{next}, \mathit{flag}, \mathit{data}, h)$. The pointer variables range over $V \cup \{\mathit{nil}\}$, where *nil* is a special value, and are denoted by e.g. p, p_0, p_1. 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 (denoted by e.g.o, o_0, o_1) give method automata input/output capabilities and are referred to as *IO pointers*. The set R of pointers (i.e. pointer variables and pointer constants) of a method automaton is defined by $R = P \cup \{\mathit{head}\} \cup O$. The data variables in DV range over the unbounded domain D.

The set of transitions T is 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.

Let $succ_P$ be the successor relation defined by the partial order $\leq_P$. The syntax of *guards* G and *actions* A are now defined as:

$$
\begin{aligned}
DE &::= v \mid data(p) \\
G &::= flag(p) = s \quad (\text{where } s \in \Sigma) \mid DE = DE \mid DE < DE \mid p = p' \\
&\quad \mid p = nil \mid p = next(p') \mid next(p) = nil \mid G \text{ and } G \mid \neg G \mid true \\
Act &::= flag(p) := s \quad (\text{where } s \in \Sigma) \mid data(p) := DE \\
&\quad \mid next(p) := nil \ \mid next(p) := p' \quad (\text{where } succ_P(p', p)) \\
&\quad \mid values(p) := (s, DE, p') \quad (\text{where } succ_P(p', p)) \\
&\quad \mid v := DE \mid \ p := p' \quad (\text{where } succ_P(p', p)) \\
&\quad \mid p := nil \ \mid p_0 := next(p_0).
\end{aligned}
$$

where p, p' are pointer variables, p_0 is the current pointer (minimum pointer variable), and v is a data variable.

The guards include symbol, data and pointer comparison and their boolean combinations. The restriction $succ_P(p', p)$ placed on some actions enforce that the heap 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 be updated. Also, the three fields of vertices (Σ value, data value, and the next pointer) can be updated together atomically (this is needed for encoding some of the Java concurrency primitives).

We require the actions of a method automaton to satisfy a restriction OW, abbreviation for "*One write before move.*" This restriction states that there is at most one action modifying $flag(p)$, at most one action modifying $data(p)$, and at most one action modifying $next(p)$ 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.

A method automaton is *deterministic* iff given a state and a valuation of variables, at most one guarded action is enabled.

Figure 1 shows a method automaton in state q. Its *head* pointer points to the vertex h of the heap. A client of the automaton can store values in the vertex v_6 pointed to by the IO pointer o. The variables p_0 and p_1 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 vertex pointed to by an IO pointer o. A method automaton can achieve this by having a transition such as: $(q, data(p_0) < data(o), p_0 := next(p_0), q)$.
- *Inserting a vertex.* Assume that the position to insert the vertex has been found - the new vertex o is to be inserted between p_1 and p_0. The transition

relation can then include $(q, true, next(o) := p_0, q_1)$ and $(q_1, true, next(p_1) := o, q_2)$.

- *Locking individual vertices.* We can model locking of vertices using the Σ value. Let us suppose that $\Sigma = \{u, l_1, l_2, \ldots\}$, for unlocked, locked by thread 1, locked by thread 2, etc. Locking is captured by the transition: $(q_0, flag(p) = u, flag(p) := l_1, q_1)$ for thread number 1. Unlocking can be modeled as follows: $(q_1, flag(p) = l_1, flag(p) := u, q_2)$.
- *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 tuples: $(q, data(p) = ev, data(p) := uv, q')$ and $(q, data(p) \neq ev, -, q'')$.

Semantics An *automaton invocation* $A(L, io)$ consists of a method automaton $A = (Q, P, DV, T, q_0, F, head, O)$, a singly-linked data heap $L = (V, next, flag, data, h)$, and a function $io : O \to V$. The pair (L, io) is the *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 $io(o)$ is unreachable from h and $next(io(o))$ is undefined. A method automaton is initialized by having its *head* pointer pointing to h and its input variables in O initialized by the function io. 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 $[\![A(L, io)]\!]$ for a well-formed input (L, io). 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 $L = (V, next, flag, data, h)$ and $A = (Q, P, DV, T, q_0, F, head, O)$. A configuration $s = (L, q, U, dv)$ of $[\![A(L, io)]\!]$ has four components: a heap L, a state q in $Q_{err} = Q \cup \{q_{err}\}$, a valuation of pointers $U : R \to V \cup \{nil\}$ and a valuation of data variables $dv : DV \to D$. A configuration 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 data variables to the value $data(h)$. Note that there is a unique initial configuration in $[\![A(L, io)]\!]$.

The transition relation of $[\![A(L, io)]\!]$ is defined as expected. For example, if $(q, true, p := next(p), q')$ is a transition of the method automaton A, then there is a transition from a configuration (L, q, U, dv) to (L, q', U', dv), where $U'(p') = U(p')$ for all $p' \in R$ such that $p' \neq p$ and $U'(p) = U(next(p))$. The relation $(L, io) \xrightarrow{A} (L', io')$ is defined to hold if there exists a path from the initial configuration of $[\![A(L, io)]\!]$ to a configuration (L', q, U, dv), where q is a final state and io' is a restriction of U to IO pointers.

Composition of method automata Consider two method automata A_1 and A_2. We define the *parallel composition* $A_1 \parallel A_2$ informally by describing the semantics. The state space of the parallel composition of A_1 and A_2 with IO pointers $io_1 \cup io_2$ is the product of the state space of $[\![A_1(L, io_1)]\!]$ and $[\![A_2(L, io_2)]\!]$, 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. We analogously define *sequential composition* $A_1 ; A_2$. *Method expressions* compose a finite set of method automata sequentially and in parallel, they are defined by the following grammar rules: $E ::= E_S \mid (E_S \parallel E)$ and $E_S ::= A \mid (A ; E_S)$, where A is a method automaton. The semantics is given by the transition system $T(E, L, io)$ and the relation $(L, io) \xrightarrow{E} (L', io')$ is defined as in the case of single automata. Given a method expression E let $Aut(E)$ be the set of method automata that are components of E.

3 Verifying Linearizability

Linearizability [11] is the standard correctness condition for concurrent data structure implementations. In this section, we study the linearizability problem for method expressions. The proofs omitted here are available in [6].

A *history* is a sequence of method invocations and method returns (a pair of method invocation and corresponding return is called a *method call*). We say that a history h is a history of a method expression E, if h corresponds to an execution of E. A *sequential history* is such that a method invocation is immediately followed by the corresponding method return. A history is *complete* if each method invocation is followed (not necessarily immediately) by a method return. Intuitively, a sequential history h_s is a *linearization* of a complete history h, if for all threads, the projection of h to a thread is the same as the projection h_s to the same thread, and the following condition holds: if a method call m_0 precedes method call m_1 in h, then the same is true in h_s. We omit further details for lack of space, and we refer the reader to [11, 10] for a formal definition, as well as for a definition of linearizations of histories that are not complete.

A method expression is *sequential*, if it does not contain any parallel composition. Note that given a sequential method expression E_s, there is a unique complete history of E_s, denoted by $hist(E_s)$, which calls all the automata in $Aut(E_S)$. Given a method expression E and a history h of E, let $Seq(E, h)$ be the set of sequential method expressions E_s such that $hist(E_s)$ is a linearization of h. For example consider the method expression $E = E_1 \parallel E_2 \parallel E_3$ and an execution h of E where E_3 starts only after E_2 has finished and the execution of E_1 overlaps with both E_2 and E_3. The set $Seq(E, h)$ is $\{(E_1 ; E_2 ; E_3), (E_2 ; E_1 ; E_3), (E_2 ; E_3 ; E_1)\}$. Let $Seq(E)$ denote the set of sequential method expressions E_s such that $Aut(E) = Aut(E_S)$. Note that we always have $Seq(E, h) \subseteq Seq(E)$.

For a method expression E, a history h of E, a well-formed input (L, io), a heap L' and a function io', we write $(L, io) \xrightarrow{E,h} (L', io')$ if a node corresponding to (L', io') is reached in $T(E, L, io)$ using an execution whose history is h.

We have now defined the notions we need to state the definition of linearizability. However, it is often useful to specify a condition under which we are interested in checking linearizability. Such preconditions can be defined using acceptors — method automata that do not modify the heap. An example precondition is that the data values in the list starting in the initial node are sorted.

A method automaton I is called an *acceptor* if it does not use the commands that modify the heap (the first five actions defined by the grammar in Section 2): Given an acceptor I, and a well-formed input (L, io), I *accepts* (L, io) (denoted by $I \models (L, io)$) if there exists (L', io') such that $(L, io) \xrightarrow{I} (L', io')$.

We now define 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. As an example, we consider sets of elements of the data domain D as the abstract data type. A list can represent a set containing data values from unmarked nodes (marking is represented by Σ-values). Two heaps are then equivalent if the unmarked elements they contain are the same.

A method automaton is an *adt-checker* if it is a deterministic method automaton with no IO pointers. Given an *adt*-checker C, two heaps L_1 and L_2 are equivalent ($L_1 \equiv_C L_2$), if there exists a heap L' such that $L_1 \xrightarrow{C} L'$ and $L_2 \xrightarrow{C} L'$. The relation $\equiv_C$ is extended to pairs (L, io) as follows: $(L_1, io_1) \equiv_{C,b} (L_2, io_2)$ iff $L_1 \equiv_C L_2$, b is a bijection between the domains of io_1 and io_2 and we have $io_1(o) = io_2(b(o))$. We omit b if it is clear from the context, for instance when comparing different compositions of the same automata.

We are now ready to state the central definition of this paper:

> Given an acceptor I and an *adt*-checker C, a method expression E is (I, C)*-linearizable* if and only if the following condition holds: for all L, io, L_P, io_P, h such that (L, io) is a well-formed input, $I \models (L, io)$, we have that if $(L, io) \xrightarrow{E,h} (L_P, io_P)$, then there exists a sequential method expression E_s in $Seq(E, h)$ and L_S, io_S such that $(L, io) \xrightarrow{E_s} (L_S, io_S)$ and $(L_P, io_P) \equiv_C (L_S, io_S)$.

The definition of method expression linearizability captures the standard definition of linearizability [11] for the case of composition of a bounded number of methods. In the standard definition, we have the requirement that all histories (possibly of unbounded length) are linearizable. Method expression linearizability not only checks that all bounded histories of E are linearizable, it also checks that starting on the same list, every interleaved execution should finish with the same list as at least one sequential execution whose history is a linearization of the history of the interleaved execution. Put yet another way, method expression linearizability checks not only that all histories of E are linearizable, but also checks that all histories of P_1 ; E ; P_2 are linearizable, for all sequential programs P_1 and P_2. As an example, consider a set data structure with methods `insert` and `contains`. With these two methods, the requirement 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 now formulate the decision problem considered in this paper:

Given a method expression E, an acceptor I and an *adt*-checker C the *method expression linearizability problem* is to decide whether E is (I, C)-linearizable.

In the remainder of this paper, we assume that the method expressions E are composed of deterministic method automata. This assumption means that given an expression E, all sequential method expressions in $Seq(E)$ are deterministic.

3.1 Reachability

In order to show that method expression linearizability is decidable, we will need the following results. First, we show that the effect of the method expression 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 a method expression E, there exists a method automaton $LS(E)$ such that for all L_P, L'_P, io_P, io'_P such that (L_P, io_P) is a well-formed method input, we have $(L, io) \xrightarrow{E} (L', io')$ iff $(L, io) \xrightarrow{LS(E)} (L', io')$.*

Proof. The idea behind constructing method automaton $LS(E)$ is to update the current pointers of all the method automata in $Aut(E)$ in lockstep manner — i.e. that the current pointers of the automata traverse the list at most one step apart. For example, if the current pointer of A_1 is one step ahead of the current pointers of the other automata, then transitions of the other automata are scheduled until the current pointers point to the same position. At that point, a transition of any automaton can be chosen. The lockstep construction is a 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. In this step the restriction OW is needed. □

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

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.*

The main insight in the construction is that, as the automaton traverses the heap monotonically from left to right, the information stored in vertices pointed to by lagging pointers that is needed for evaluating guards of the transitions can be encoded in a finite manner. More concretely, one need not to remember values in D, but only the equality and order information on these values.

3.2 Deciding linearizability

The following theorem is the main result of the paper.

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

Proof. The proof is by reduction to reachability in method automata, which in turn reduces (by Theorem 2) to reachability in finite state systems. We show how method expression linearizability can be reduced to reachability in a method automaton. Given an acceptor I, a method expression E and an *adt*-checker C, the method automaton $\mathit{LinCheck}(I, E, C)$ simulates I followed by E followed by C, and compares the results to simulation of I followed by E_s followed by C for all $E_s \in \mathit{Seq}(E)$. $\mathit{LinCheck}(I, E, C)$ reaches an error state if there is an unlinearizable execution of E starting from a heap accepted by I. Given a method expression E, the number of automata $\mathit{LinCheck}(I, E, C)$ simulates grows exponentially with the number of methods in E.

First, we use Theorem 1 to show that instead of simulating $(I\ ;\ E\ ;\ C)$ (resp. $(I\ ;\ E_s\ ;\ C)$), one can simulate the method automaton $LS(I\ ;\ E\ ;\ C)$ (resp. $LS(I\ ;\ E_s\ ;\ C)$). Second, we show how $\mathit{LinCheck}(I, E, C)$ can simulate the automaton $LS(I\ ;\ E\ ;\ C)$ and all the automata $LS(I\ ;\ E_s\ ;\ C)$ for $E_S \in \mathit{Seq}(E)$. on the same input heap and reach an error state if there is an unlinearizable execution of $LS(E)$. The key idea is once again that the current pointers of all the automata can advance in a lockstep manner. The reason is much simpler in this setting than in the proof of Theorem 1 — here the automata do not communicate at all (the only reason we are simulating the the automata together is that they run on the same input heap). $\mathit{LinCheck}(I, E, C)$ reaches an error state if none of the expressions in $\mathit{Seq}(E)$ can simulate $LS(E)$. This is the case when for example a particular position in the output list for $LS(I\ ;\ E\ ;\ C)$ is different from that position in output lists of $LS(I\ ;\ E_S\ ;\ C)$ for all $E_S \in \mathit{Seq}(E)$. Such condition is checkable by a method automaton, so $\mathit{LinCheck}(I, E, C)$ can take a transition to a particular state u if it occurs. The state u is then reachable iff E is not (I, C)-linearizable. We have reduced linearizability to reachability in method automata. We can thus conclude by using Theorem 2. □

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

Theorem 4. *For method automata without the OW restriction, the method expression linearizability problem is undecidable.*

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

4 Experimental Evaluation

4.1 Examples

This section presents a range of concurrent set algorithms where the set is implemented as a linked list whose vertices are sorted by their keys. Each key

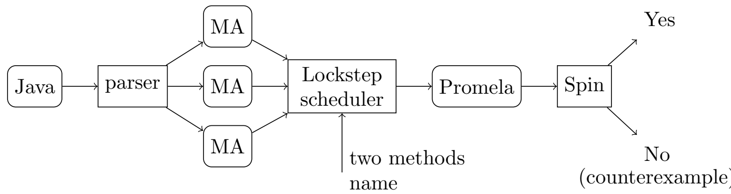

Fig. 2. CoLT Toolchain

occurs at most once in the set. The concurrent set provides an interface consisting of three methods: `contains`, `add`, and `remove`. The main difference in the algorithms comes from the synchronization style they use. The synchronization techniques we consider in our experiments are *fine-grained locking*, *optimistic synchronization*, *lazy synchronization*, and *lock-free synchronization*:

– In the *fine-grained locking* approach each vertex is locked separately. During the traversal we use "hand-over-hand" locking, where a vertex is unlocked only after its successor is locked. When an insertion or a deletion is performed, two successive vertices are kept locked.
– A problem with fine-grained locking is that modifications in disjoint parts of the list can still block each other. In *Optimistic* synchronization [10] a thread does not acquire locks as it traverses the list, but only when it finds the part it is interested in. Then the thread needs to re-traverse the list to make sure the locked vertices are still reachable (*validation* phase).
– The *lazy* synchronization algorithm [9] 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 is that the deleted vertices are marked.
– A method is called *lock-free* if delay in one thread executing the method cannot delay other threads executing the method. The lock-free algorithms [8, 15] we analyze use the Java `compareAndSet` operation to overwrite values.

4.2 Implementation

The CoLT tool chain can be seen in Figure 2. The input to the tool is a Java file and two method names. The Java methods are parsed into method automata. Then the lockstep scheduler selects the method automata corresponding to the given method names, 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 [12] model checker. If SPIN cannot validate the model, it returns a counterexample trace that describes an unlinearizable execution. CoLT then gives the programmer a visual representation of the trace.

In the rest of this subsection, we summarize the main issues in translating the Java implementations of concurrent data set algorithms to method automata. We refer the reader to [6] for further details on implementation.

Acceptors and ***adt*****-checkers** We use an acceptor to assert that the input list is sorted. In the case of the optimistic algorithm we also need an *adt*-checker to handle the marked vertices, i.e. vertices removed logically but not physically. For the other algorithms the *adt*-checker is the identity function.

Phases approach We implemented 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.

Validate The optimistic algorithm violates the monotonic traversal restriction as it traverses the list twice, once to find the required vertex and lock it and again to validate that the locked vertex 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 into two method automata, one that finds and locks the vertex 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 is monotonic. The only caveat is that when an operation such as insertion or deletion fails the method might abort and "retry" by setting all pointers to the head, which our restrictions disallows. We emphasize that retry behavior is very different from the validate behavior of the optimistic algorithm. The aborted executions in the fine-grained, optimistic, and lazy methods have no effect on the heap. In the lock-free method, the effect is limited and simply defined. We implemented a simple heuristic to deal with retry behavior. The heuristic produces a method automaton that stops simulating an execution if a retry occurs. One can easily prove for all algorithms we have considered that if the parallel composition of method automata constructed in this way is linearizable iff the original parallel composition is linearizable.

Linearization points Our tool enables programmers to specify linearization points. Specifying them is not needed, but leads to reduction of the search space, and thus to improving memory consumption and running time of experiments.

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 [10]. All the experiments were performed on a server with an 1.86GHz Intel Xeon processor and 32GB of RAM.

The results of the experiments are presented in Table 1. the third (fourth) column contains the number of lines of code and the number of pointer variables of the first (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 last column indicates whether the method expression was linearizable.

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

First, to evaluate our analysis on implementations of fine-grained locking algorithms, we ran the remove method in parallel with itself, the contains method, and 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 use the heuristic described in the previous subsection. CoLT validates the optimistic implementations in under 40MB of memory for every case. The heuristic influence heavily some of the tool's components; hence, the resources consumption of these results are not directly comparable with the others.

Third, we analyzed lazy-synchronization implementations. The tool CoLT verified linearizability in the same cases as the fine-grained locking algorithm. We used the tool to analyze modifications of the add and remove methods suggested as exercises in [10]. One exercise suggests simplification of the validation check (methods remove_1 and add_1), the other asks using only one lock (method remove_2). We used the tool on $\text{remove}_1 \parallel \text{add}_1$, and on $\text{remove}_2 \parallel \text{remove}_2$. In both cases, CoLT reported these compositions not to be linearizable.

Fourth, we considered lock-free implementations. CoLT found that the parallel composition of remove with itself is not linearizable. This is a known bug, reflected in the online errata for [10]. When we corrected the bug according to the errata (method removeCorr —short for removeCorrected), the tool showed that the parallel composition of remove with itself is linearizable. We observe that the memory usage is larger, for example for the parallel composition of the corrected remove method with the add method, even when the linearization are provided. The tool runs out of memory without the linearization points. The reason is that, compared to the other algorithms, the input list can contain vertices marked for deletion, thus increasing the number of inputs to consider.

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. D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, pages 477–490, 2007.
4. M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, pages 7–16, 2006.
5. S. Burckhardt, R. Alur, and M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.
6. P. Černý, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent data structure implementations. Technical Report IST-2010-0001, IST Austria, April 2010.
7. R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, 2006.
8. T. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.
9. S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.
10. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., 2008.
11. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
12. G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.
13. D. Lea. The java.util.concurrent synchronizer framework. In *CSJP*, 2004.
14. Y. Liu, W. Chen, Y. Liu, and J. Sun. Model checking linearizability via refinement. In *FM*, pages 321–337, 2009.
15. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.
16. M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.
17. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, 2004.
18. M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Abstract transformers for thread correlation analysis. In *APLAS*, pages 30–46, 2009.
19. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP*, pages 129–136, 2006.
20. M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *SPIN*, pages 261–278, 2009.