Automatic linearizability proofs of concurrent objects with cooperating updates ⋆

Cezara Drăgoi, Ashutosh Gupta, and Thomas Henzinger

Institute of Science and Technology Austria, Klosterneuburg

Abstract. An execution containing operations performing queries or updating a concurrent object is linearizable w.r.t an abstract implementation (called specification) iff for each operation, one can associate a point in time, called linearization point, such that the execution of the operations in the order of their linearization points can be reproduced by the specification. Finding linearization points is particularly difficult when they do not belong to the operations's actions. This paper addresses this challenge by introducing a new technique for rewriting the implementation of the concurrent object and its specification such that the new implementation preserves all executions of the original one, and its linearizability (w.r.t. the new specification) implies the linearizability of the original implementation (w.r.t. the original specification). The rewriting introduces additional combined methods to obtain a library with a simpler linearizability proof, i.e., a library whose operations contain their linearization points. We have implemented this technique in a prototype, which has been successfully applied to examples beyond the reach of current techniques, e.g., Stack Elimination and Fetch&Add.

1 Introduction

Linearizability is a standard correctness criterion for concurrent objects, which demands that all concurrent executions of the object operations are equivalent to sequential executions of some abstract implementation of the same object, called specification. In this paper, we address the problem of automatically proving linearizability for concurrent objects which store values from unbounded domains (such as counters and stacks of integers) and are accessed by an unbounded number of threads.

Concurrent objects (data structures) are implemented in wide audience libraries such as `java.util.concurrent` and Intel Threading Building Blocks. The search for new algorithms for concurrent data structures that maximize the degree of parallelism between the operations is an active research area. The algorithms have become increasingly complex, which makes proving their linearizability more difficult. Automatic verification techniques are expected not only to confirm the manual correctness proofs provided with the definition of the algorithms, but also to verify the various implementations of these algorithms in different programming languages.

Proving linearizability is a difficult problem because the configurations of the concurrent object and number of threads that access it are in general unbounded. An execution is *linearizable* if there exists a permutation of its *call* and *return* actions, preserving

⋆ This work was supported in part by the Austrian Science Fund NFN RiSE (Rigorous Systems Engineering) and by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

the order between non-overlapping operations, which corresponds to a sequential execution allowed by the specification. This is equivalent to choosing for each operation an action between its *call* and *return*, called *linearization point*, such that the sequential composition of all operations, in the order in which the linearization points appear in the execution, belongs to the specification.

The techniques introduced in the literature for proving linearizability differ in the degree of automation and the class of libraries that they can handle. The works in [13, 15] present semi-automatic techniques, where human guidance is required either to interact with the theorem prover [15] or to carry out mathematical reasoning. The work presented in [5] defines a class of concurrent objects for which linearizability is decidable if the object is accessed by a bounded number of threads. More related to the present paper, [2, 3, 17] present fully automatic techniques for proving linearizability based on abstract interpretation. Usually, automatic techniques prove linearizability using an abstract analysis of the concurrent implementation simultaneously with the sequential implementation of the abstract object. The analysis choses for each concurrent operation a linearization point. When the abstract execution reaches the linearization point of an operation, its sequential implementation is executed (with the same input as the concurrent one) and later, the returned values of the two operations are compared.

The concurrent objects which have been proven automatically linearizable by previous techniques have only operations with internal linearization points, i.e., when the linearization point of each operation is one of its actions, for all executions. A notable exception is [17], which lifts this restriction for operations whose specification doesn't update the abstract object. In this paper, we extend the state of art by introducing a technique to deal with concurrent objects whose *updating operations* have external linearization points. Several classical libraries such as [8, 9, 12, 16] fall into this category.

When all operations have internal linearization points, there is a correspondence between an operation and its potential linearization points, which can be defined using assertions over variables of that operation and additional prophecy or ghost variables.

Defining such a correspondence for operations with external linearization points, i.e., the linearization point is an action of another, concurrently executing operation, is more difficult. In this case, for each action s one has to identify all possible concurrently executing operations whose linearization point could be s. Consequently, one needs an assertion language that allows relating the local variables of the operation executing s with the local variables of the threads executing the operations that have s as a linearization point, which is difficult to define and reason about.

This paper introduces a new technique for rewriting the implementation $\mathcal{L}$ of a concurrent object and its specification $\mathcal{S}$ into a new implementation $\mathcal{L}^n$ with a new specification $\mathcal{S}^n$, such that $\mathcal{L}^n$ preserves all executions of $\mathcal{L}$, and the linearizability of $\mathcal{L}^n$ w.r.t. $\mathcal{S}^n$ implies the linearizability of $\mathcal{L}$ w.r.t. $\mathcal{S}$. The aim of the rewriting is that all operations of the new implementation contain their linearization points. Let us consider two operations o and a in execution e of $\mathcal{L}$, such that a contains the linearization point of o. The execution corresponding to e, obtained by rewriting $\mathcal{L}$, does not have o and a; instead it contains an operation of a new method, denoted $o+a$, which has the combined behavior of a and o, namely, $o+a$ has all the actions of o and a, in the same order in which they occur in the original execution. The rewriting consists in (1) adding new *combined*

methods, which are interleavings of methods that have external linearization points and methods that contain these linearization points, and (2) removing code fragments from methods in the original implementation whose behaviors are captured by the new, combined methods. The second step is crucial in order to eliminate operations with external linearization points. The specification of a combined method is the non-deterministic sequential composition of the method specifications whose interleaving it represents.

We have observed that an operation with external linearization points in concurrent object implementations such as [8, 9, 12, 16] is included in a dependency cycle that witness the non-serializability of the execution in which it appears. Instead of searching for operations with external linearization points, we define an algorithm to compute an under-approximation of the set of dependency cycles and then, constructs combined methods whose executions correspond to interleavings that contain such cycles.

We reduce the problem of eliminating operations with external linearization points to the problem of eliminating non-serializable executions that contain dependency cycles. Concretely, the rewriting removes code fragments from the original implementation whose executions are included in the invocations of the combined methods. This step is reduced to checking some invariants over the executions of the original library, which are proven using a technique based on abstract interpretation [6] and counter abstraction [7]. Note that all these steps are automatic.

The linearizability of the original library is implied by a stronger version of linearizability for the new library, which restricts the possible choices for linearization points, to an interval strictly included in the time span of an operation, i.e., the interval of time between its call and return actions. Our technique rewrites operations having linearization points in other, concurrently-executing operations that form a dependency cycle identified by our algorithm. In our examples it eliminates all method invocations with external linearization points, and this allows us to use existing techniques [17, 3] to prove the linearizability of the new implementation. A theoretical limitation of our approach is that it cannot deal with unbounded sets of operations that update the concurrent object and have the same linearization point. The rewriting procedure was implemented in a prototype, which has been successfully applied to examples beyond the reach of current techniques, e.g., Stack Elimination [9] and Fetch&Add [16].

In the following, Sec. 2 presents a motivating example; Sec. 3 characterizes executions with external linearization points; Sec. 4 defines the rewriting algorithm; Sec. 5 gives the correctness theorem for the rewriting and connects the linearizability of the new library with the linearizability of the original one.

2 Motivating example

Example description: Let $\mathcal{L}_s$ be the library given in Fig. 1. It contains two methods, `push` and `pop`, which implement *Treiber's concurrent stack* extended with an elimination mechanism similar to the one used in the *Elimination stack* [9]. Each method is described by a control-flow graph (CFG, for short), whose nodes denote atomic blocks. We assume that each thread executes at most one operation, i.e., method instance.

The stack is implemented by a singly-linked list and `S` is a global pointer that points to the top of stack. In order to increase the degree of parallelism in Treiber's stack, which is limited by the sequential access to `S`, the elimination mechanism in [9] allows pairs

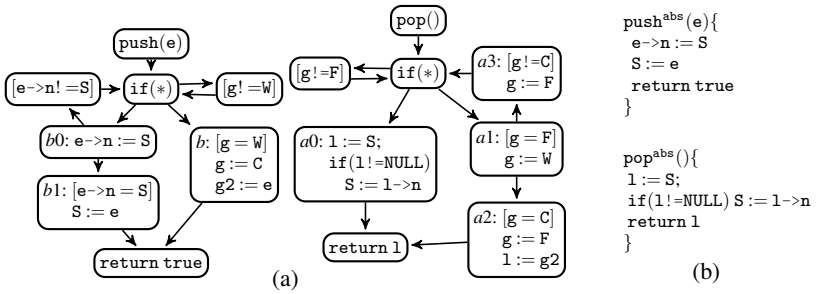


Fig. 1: (a) *Treiber's stack* with an elimination mechanism. The formulas in square brackets are assumes. Some code blocks are labelled for easy reference. (b) The specification is defined from the abstract implementations $\texttt{push}^{\texttt{abs}}$ and $\texttt{pop}^{\texttt{abs}}$ of `push` and `pop`.

of overlapping instances of `pop` and `push` to exchange their values without accessing `S`. Each method invocation non-deterministically chooses either to try to modify the stack or apply the elimination mechanism. The elimination mechanism is implemented as follows. If the value of `g` is `W`(ating), then there is an invocation of `pop` ready to exchange values and waiting for a `push`. If the value of `g` is `C`(ollided), then a pair of `push` and `pop` invocations is ready to exchange the value stored in `g2`. The exchange is complete when `pop` sets the value of `g` back to `F`(ree) so that another elimination can take place. The elimination mechanism is an example of cooperative update.

We define specifications with respect to *abstract implementations* of methods. The abstract implementations of `push` and `pop` are given in Fig. 1(b). For any method op, the abstract implementation op^{abs} has the same arguments and return values. The specification $\mathcal{S}_s$ of $\mathcal{L}_s$ is the set of all sequential executions of the abstract implementations.

Linearizability: A (concurrent) execution e of a library $\mathcal{L}$ is linearizable w.r.t a specification $\mathcal{S}$ iff there is an execution s in $\mathcal{S}$ such that (1) for each operation in e there is an operation in s with the same interface (same invocation parameters and same responses) and (2) s preserves the order between non overlapping operations. The execution s is called the *linearization* of e. The library $\mathcal{L}$ is linearizable w.r.t $\mathcal{S}$ if all its executions are linearizable w.r.t $\mathcal{S}$. Fig. 2(a) shows an execution of $\mathcal{L}_s$ consisting of one instance of `pop` and two instances of `push`. Vertical dashed lines represent the context switches. Two possible linearizations of this execution are given in Fig. 2(b1) and (b2).

Linearization points: Linearizability is often proved using the notion of linearization point [11]. An execution e of $\mathcal{L}$ is linearizable w.r.t. $\mathcal{S}$ iff for each operation op in e, one can choose an action of e located between the `call` and the `return` action of op, called linearization point, such that: the sequential composition of the abstract implementations corresponding to the operations in e, in the order defined by the linearization points, defines a linearization for e. The actions pointed to by dotted arrows in Fig. 2(a) are the linearization points that lead to the linearization in Fig. 2(b1) (resp., Fig. 2(b2)). The linearization points in Fig. 2 are called *internal* because they are actions of the operations whose linearization point they represent. A non-internal linearization point is called *external*. Proving automatically the linearizability of libraries with external linearization points is beyond the scope of all existing techniques we are aware of, except the work of Vafeiadis [17], which handles external linearization points but only for methods with specifications that *do not modify, but only read, the global data structure*.

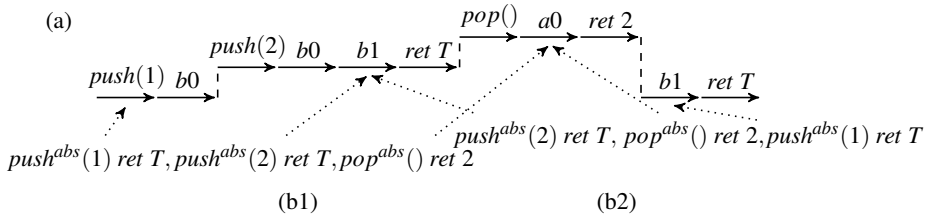


Fig. 2: A concurrent execution and two possible linearizations.

Rewriting libraries for dealing with external linearization points: Let us consider the execution in Fig. 3, which consists of three operations $pop()$, $push(1)$, and $push(4)$ (a method name is written with typewriter font and its operations are written in italics). $pop()$ and $push(1)$ execute the elimination mechanism while $push(4)$ accesses the stack. Note that the linearization point of $pop()$ is *external* and it must be an action of $push(1)$. If we choose $a1$ as a linearization point for $pop()$, then there exists no sequential execution of the abstract implementations which exposes the same interface for all operations (intuitively, $pop()$ would have to return `Empty`). The same holds for all the other actions of $pop()$. A linearization for this execution is given in Fig. 3(a). If an operation contains the linearization point of another operation, e.g. $push(1)$ contains the linearization point of $pop()$, we say that the two instances *share linearization points*.

Our approach to proving linearizability of libraries with external linearization points (e.g. $\mathcal{L}_s$) is to rewrite the original library into a new library that has similar executions but a simpler linearizability proof (if any) w.r.t. a new specification, obtained from the specification of the original library.

> If the new library is linearizable w.r.t the new specification, then the original one is also linearizable w.r.t. its specification.

In the case of $\mathcal{L}_s$, our rewriting will introduce a new method `push+pop`, that will replace the elimination mechanism in $\mathcal{L}_s$. The execution of the new library corresponding to the execution in Fig. 3(a) is given in Fig. 3(b). The operations $pop()$ and $push(1)$, that exchange values via the elimination mechanism, have been rewritten into sub-sequences of an instance of a new method $push{+}pop(1)$. The time interval of $push{+}pop(1)$ is the union of the time intervals of $pop()$ and $push(1)$ and its interface is the union of the interfaces of $push(1)$ and $pop()$. The program instructions executed by $push{+}pop(1)$ are the same, and in the same order, as the ones in $push(1)$ and $pop()$, except for `call` and `return` instructions (the `call` and the `return` of $push(1)$ have been replaced by `skip`). The abstract implementation of `push+pop` corresponds to the sequential composition of $\texttt{push}^{abs}$ and $\texttt{pop}^{abs}$. A linearization for the new execution is given in Fig. 3(b). Notice that this linearization is defined using only internal linearization points.

The rewriting (1) adds new methods, called *combined methods*, that represent exactly those interleavings between the operations with external linearization points and the operations which contain these linearization points and (2) removes fragments of code from the methods of the original library such that these interleavings are not possible anymore (they will be only instances of the combined methods). In the case of $\mathcal{L}_s$, to define the new library $\mathcal{L}_s^n$, we create combined methods, called `push+pop`, whose instances are interleavings of instances of `pop` and `push` that exchange values through the elimination mechanism. Also, we modify the code of `pop`, respectively `push`, in order

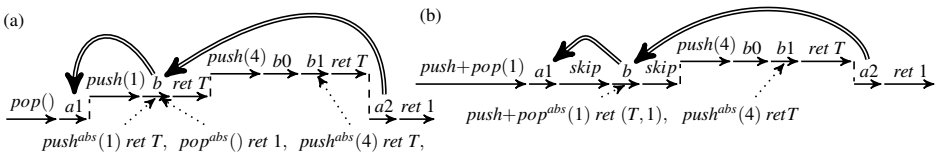


Fig. 3: (a) An execution of $\mathcal{L}_s$ where `pop` has only external linearization points. (b) An execution of $\mathcal{L}_s^n$ containing an invocation of the combined method `push+pop`.

to eliminate the instances executing the elimination mechanism. The specification of the new library is defined as follows: (1) the abstract implementation of each combined method is the non-deterministic sequential composition of the abstract implementations of the methods whose interleavings it represents and (2) the abstract implementation of the methods inherited from the original library remains the same. Fig. 5 shows one combined method from $\mathcal{L}_s^n$. Let $\mathcal{S}_s^n$ denote the new specification of $\mathcal{L}_s^n$.

In the case of $\mathcal{L}_s$, the new library has no operations with external linearization points and thus, its linearizability can be proved using the existing techniques.

Removing retry loops: The methods of a concurrent object often contain a retry loop, i.e., a loop where each iteration tries to execute the effect of the operation and if it fails it restarts forgetting any value computed in previous iterations. Given a library $\mathcal{L}$ whose methods contain retry loops, one can define a new library $\mathcal{L}'$ by replacing every retry loop `while` $(Cond)$ $\{Loop\_body\}$ with $Loop\_body$;`assume` *false* such that $\mathcal{L}$ is linearizable iff $\mathcal{L}'$ is linearizable. In principle, one can use classical data flow analyses in order to identify retry loops. In the following, we will consider concurrent objects implementations without loops.

3 Executions with external linearization points

In this section, we present a connection between the existence of operations with only external linearization points and serializability [14].

In an execution, a *conflict* is a pair of actions (program instructions) that access the same shared memory location and at least one of them modifies its value (each double arrow in Fig. 3 defines a conflict between its source and its destination). The *dependency graph* of an execution, is an oriented graph whose nodes represent operations and whose edges represent conflicts; the orientation is the order in which the actions in conflict appear in the execution. An execution is *serializable* (called also view-serializable) iff it can be reordered into an execution that has an acyclic dependency graph [1], called *serialization*, such that (1) the read actions read the same values as in the original execution and (2) both executions have the same final state. The execution in Fig. 2(a) is serializable but the execution in Fig. 3(a) is not serializable. A conflict between a read and a write action such that the read follows the write in the execution is called a *data-flow dependency*. A cycle in the dependency graph such that all of its edges define data-flow dependencies is called a *data-flow dependency cycle*.

[1] The classical definition requires that the execution be sequential (a sequential composition of operations). However, any execution with an acyclic dependency graph can be reordered into a sequential execution with the same final state and where the read actions read the same values.

We make the following empirical observation relating executions with external linearization points and non-serializable executions:

> *If an execution contains two operations o and a s.t. the linearization point of o can only be an action of a and the abstract implementation of o modifies the logical state, then the execution is not serializable. Moreover, the two operations define a data-flow dependency cycle.*

Intuitively, the dependency cycle between o and a arises because the two operations communicate through global variables in both directions.

Communication from a to o: A linearization point can be thought of as the point in time where the logical effect of the method takes place and consequently, the value returned by a method should depend on the outcome of executing its linearization point. The action of a, which is the linearization point of o, is usually a write on a shared memory location, read later by o in order to determine its return value. In the execution from Fig. 3(a), $pop()$ returns the value written by $push(1)$ in variable `g2`.

Communication from o to a: Since o has a specification that modifies the logical state, o usually needs to communicate its intended modification to a via a shared memory location. In our example, $pop()$ assigns to `g` the value `W` in block $a1$ in order to announce to $push(1)$ that a pop is ready to exchange values.

In the following, we focus on data-flow dependency cycles s.t. any reordering of their actions does not lead to an execution of the library where they are not present anymore. These cycles were sufficient for all examples we have found in the literature.

4 Rewriting algorithm

In this section, we describe the algorithm we propose for rewriting implementations of concurrent objects. The new implementations preserve all the behaviours of the original ones but, their executions contain fewer data-flow dependency cycles. To describe executions with data-flow dependency cycles we introduce a first order logic called $\mathbb{CL}$.

Let $\mathcal{L}$ be a library and $\mathcal{S}$ its specification. Let $[\![\mathcal{L}]\!]$ denote the set of executions of $\mathcal{L}$. The rewriting algorithm computes a library $\mathcal{L}^n$ and a specification $\mathcal{S}^n$ in several steps:

(1) Compute a set of formulas $\Lambda[\mathcal{L}]$ in $\mathbb{CL}$, where each formula in $\Lambda[\mathcal{L}]$ has a model in $[\![\mathcal{L}]\!]$ that has at least one data-flow dependency cycle (see Sec. 4.2).

(2) $\mathcal{L}^n$ is the union of $\mathcal{L}$ and a set of combined methods defined using the formulas in $\Lambda[\mathcal{L}]$. Any execution that contains an instance of a combined method corresponds to an execution in $[\![\mathcal{L}]\!]$, that satisfies some formula in $\Lambda[\mathcal{L}]$. The abstract implementation of each combined method is the non-deterministic sequential composition of the abstract implementations of the methods whose interleavings it represents (see Sec. 4.3).

(3) The methods of $\mathcal{L}^n$, that are copied from $\mathcal{L}$, are modified by removing CFG paths that are executed only in operations whose behaviour is captured by one of the combined methods. The goal of this step is to make the formulas in $\Lambda[\mathcal{L}]$ unsatisfiable over the executions of $\mathcal{L}^n$ (see Sec. 4.4).

Note that $\Lambda[\mathcal{L}]$ captures a *subset* of the data-flow dependency cycles in $[\![\mathcal{L}]\!]$. Indeed, if more cycles are captured by $\Lambda[\mathcal{L}]$, then our rewriting will be more effective. For simplicity, the second and the third step of the algorithm are explained only for the case when $\Lambda[\mathcal{L}]$ is a singleton. Note that the rewriting detects and removes only sets of data-flow dependency cycles of finite length.

4.1 Logical representations for data-flow dependencies

The number of data-flow dependency cycles occurring in the executions of a library is potentially infinite. Therefore, we define a symbolic representation for sets of data-flow dependencies by formulas in a logic called *Conflict-cycles logic* ($\mathbb{CL}$, for short). This logic is parametrized by a library $\mathcal{L}$.

Formulas in $\mathbb{CL}$ define a partial order between pairs of conflicting actions and relations between the local states of the operations whose actions are in conflict. The variables of $\mathbb{CL}$ are interpreted over operations of $\mathcal{L}$. A model of a formula is an execution together with an interpretation of its variables to operations in this execution.

Let v, v_1, ..., v_n denote variables of $\mathbb{CL}$ interpreted as operations. $\mathbb{CL}$ contains the following predicates and formulas:

– $v.a$ is a unary predicate, which holds iff in the considered execution, v interprets into an operation that executes the statement at control location a;

– $v_1.a < v_2.b$ is a binary predicate, which holds iff in the considered execution, the operation denoted by v_1 executes the statement at control location a before the operation denoted by v_2 executes the statement at control location b (we recall that, since we consider only loop-free methods, a location is reached only once by an operation);

– $v.a \rightarrow \mathit{Bool\_Expr}(v, v_1, \ldots, v_n, \mathtt{g})$, is true iff the state in which v reached the control location a satisfies the boolean expression $\mathit{Bool\_Expr}(v, v_1, \ldots, v_n, \mathtt{g})$, where $\mathit{Bool\_Expr}(v, v_1, \ldots, v_n, \mathtt{g})$ is build over the global variables $\mathtt{g}$ of $\mathcal{L}$ and the local variables of the threads executing $v, v_1, \ldots, v_n$.

$\mathbb{CL}$ formulas are conjunctions of the above predicates and formulas. For readability, the suffix of a variable name is a method name. Such a variable is interpreted only as an instance of that method, e.g., $v\mathtt{pop}$ is interpreted only as an instance of `pop`. The *multiset of methods* in φ, denoted by $\mathsf{Op}(\varphi)$, consists of all methods whose instances are denoted by variables of φ. An execution e *satisfies* φ iff there is a mapping from the variables of φ to operations in e such that φ holds. $\mathcal{L}$ *satisfies* φ iff there is an execution of $\mathcal{L}$ that satisfies φ. A formula φ containing the atoms $v_0.a_0 < v_1.a_1, \ldots, v_{n-1}.a_{n-1} < v_n.a_n$ describes a dependency cycle if $v_0 = v_n$ and there is $i \in 0..n$ s.t. $v_i \neq v_0$. A model of such a formula φ has a data-flow dependency cycle if for each $i \in 1..n$ the actions corresponding to $v_{i-1}.a_{i-1}$ and $v_i.a_i$ form a data flow dependency.

Example 1. The formula $\varphi_{\mathcal{L}} = v\mathtt{pop}.a1 < u\mathtt{push}.b < v\mathtt{pop}.a2$ describes a dependency cycle. The execution in Fig. 3 is a model for $\varphi_{\mathcal{L}}$ where, $v\mathtt{pop}$ and $u\mathtt{push}$ are interpreted into $pop()$ and $push(1)$ respectively. The model has a data-flow dependency cycle, since $(pop().a1, push(1).b)$ and $(push(1).b, pop().a2)$ are data flow dependencies.

4.2 Under-approximation of data-flow dependency cycles

To populate $\Lambda[\mathcal{L}]$, we search for executions containing finitely many concurrent operations that form a data-flow dependency cycle such that if one of the operations is removed from the interleaving, then the rest of them do not terminate.

We present an algorithm that given a library $\mathcal{L}$, computes an under-approximation of the data-flow dependency cycles exposed by executions of $\mathcal{L}$. It is parametrized by two *sequential* analyses: an abstract *reachability* analysis and a *may-alias* [2] analysis. To

[2] An analysis that determines if two variables may point to the same memory location.

this, we define a recursive procedure $get\_cycle(l, \mathsf{op}, \mathsf{k})$ that receives as input a control location l from some method op of $\mathcal{L}$ and an integer k and returns a $\mathbb{CL}$ formula that represents data-flow dependency cycles of length at most k containing operations of op. This procedure works as follows:

(1) If l is reachable under the sequential analysis, following any of the paths in the CFG of op starting from the initial state of the library, denoted by S_0, then *get_cycle* returns *true*.

(2) Otherwise, let a be the first control location of an `assume` statement (on a path to l), which is infeasible. Using the may-alias analysis, *get_cycle* identifies a set of assignments Wr (in op or in other methods of $\mathcal{L}$) that may modify the variables in the `assume` statement.

(3) Choose an assignment in Wr at control location b of some method op' and, if $k > 0$ then recursively call *get_cycle* on b; otherwise return $false$. Let φ' be the formula returned by the recursive call.

$get\_cycle(ret, \texttt{pop}, 2)$
/*1*/ $a2$ is unreachable
/*2*/ $Wr = \{b : g := C\}$
/*3*/ $\varphi' = get\_cycle(b, \texttt{push}, 1)$
  /*1*/ b is unreachable
  /*2*/ $Wr = \{a1 : g := W\}$
  /*3*/ $\varphi' = get\_cycle(a1, \texttt{pop}, 0)$
    /*1*/ $a1$ is reachable
    /*2*/ return $true$
  /*4*/ found interleaving: $\texttt{pop}.a1;\texttt{push}.b$
  /*5*/ return $\varphi' \wedge v\texttt{pop}.a1 < u\texttt{push}.b$
/*3*/ $Op(\varphi') = \{\texttt{push}, \texttt{pop}\}$
/*4*/ found interleaving:$\texttt{pop}.a1;\texttt{push}.b;\texttt{pop}.a2$
/*5*/ return $v\texttt{pop}.a1 < u\texttt{push}.b < v\texttt{pop}.a2$

Fig. 4: A run of *get_cycle* for computing data-flow dependency cycles in $\mathcal{L}_s$

(4) Search for an interleaving between (i) a path of op from its entry node to l, containing the failing `assume`, (ii) an interleaving of methods in $\mathsf{Op}(\varphi') \cup \{\mathsf{op}'\}$ s.t.

- the assignment of op' appears immediately before the `assume` of op,
- the interleaving of methods in $\mathsf{Op}(\varphi')$ satisfies the ordering constraints in φ',
- the `assume` of op is reachable (from S_0, under the considered reachability analysis).

If there is no such interleaving then, *get_cycle* returns *false*. If $\mathsf{op} \in \mathsf{Op}(\varphi')$ then, *get_cycle* begins by searching for an interleaving that contains only the methods in $\mathsf{Op}(\varphi')$ and satisfies all these constraints (this corresponds to the fact that an instance of op needed to make the assignment at b reachable is the same as the instance of op for which we want to prove that l is reachable). This is needed in order to complete a data-flow dependency cycle between the methods in $\mathsf{Op}(\varphi)$.

(5) If the control location l is reachable under the interleaving computed at (4) then, *get_cycle* returns $\varphi' \wedge (v\mathsf{op}'.b < v\mathsf{op}.a) \wedge (v\mathsf{op}'.b \rightarrow \psi)$, where ψ expresses the aliasing relating the variable assigned at b and the variables of the `assume` statement at a. Otherwise, it considers the next failing `assume` between a and l and goes to step 1.

Let $\Lambda[\mathcal{L}]$ be the union of the formulas generated by calling $get\_cycle(ret, \mathsf{op}, k)$ for each method op in $\mathcal{L}$, where *ret* denotes the return control location of op. To increase the precision, if $\Lambda[\mathcal{L}]$ contains two formulas $\varphi_1(v\mathsf{a}, v\mathsf{c})$ and $\varphi_2(v\mathsf{b}, v\mathsf{c})$ such that the actions of the method c appearing in both formulas lie on a common path in the CFG of c, we add $\varphi(v\mathsf{a}, v\mathsf{b}, v\mathsf{c}) ::= \varphi_1(v\mathsf{a}, v\mathsf{c}) \wedge \varphi_2(v\mathsf{b}, v\mathsf{c})$ to $\Lambda[\mathcal{L}]$ ($v\mathsf{a}, v\mathsf{b}, v\mathsf{c}$ are operations of the methods a, b, and c, respectively). This is necessary because the two formulas might describe two data-flow dependency cycles that share a method instance. Any execution satisfying a formula in $\Lambda[\mathcal{L}]$ contains data flow dependency cycles of length at most k. In Fig. 4, we present a run of *get_cycle* that returns $\varphi_{\mathcal{L}} = v\texttt{pop}.a1 < u\texttt{push}.b < v\texttt{pop}.a2$.

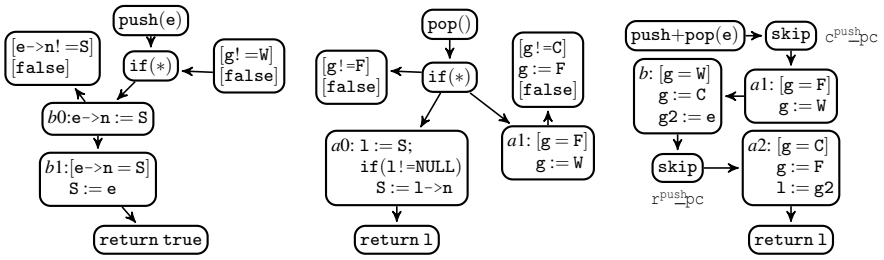


Fig. 5: Methods from the library $\mathcal{L}_s^n$ obtained by rewriting the library $\mathcal{L}_s$ in Fig. 1

4.3 Adding combined methods

Let φ be a formula in $\Lambda[\mathcal{L}]$ characterizing instances of a set of methods $\mathtt{op}_1,...,\mathtt{op}_n$. We add to the new library a set of combined methods, denoted $\mathtt{Combined}(\varphi)$, which consists of all the interleavings between $\mathtt{op}_1,...,\mathtt{op}_n$, that satisfy the ordering constraints in φ, instrumented by a set of `assume` statements that impose the relations between local variables expressed in φ. For each new method, all the `call` actions except for the first one and all the `return` actions except for the last one are replaced by `skip`. Also, all the accesses to the id of the thread that executes $\mathtt{op}_i$, $i \in 1..n$, are replaced by accesses to a new local variable $\mathtt{tid}_i$ added to the combined method, which is initialized with a unique and random integer. The number of methods in $\mathtt{Combined}(\varphi)$ equals the number of total orders over the statements in $\mathtt{op}_1,...,\mathtt{op}_n$ that access the global variables, which are consistent with the partial order defined by φ.

The input (resp., output) parameters of all methods in $\mathtt{Combined}(\varphi)$ are the union of the input (resp., output) parameters of $\mathtt{op}_1,...,\mathtt{op}_n$. The abstract implementation of each combined method is a non-deterministic choice between all orders in which one can compose the abstract implementations of $\mathtt{op}_1,...,\mathtt{op}_n$.

Example 2. In Fig. 5, we show the only method `push+pop` in $\mathtt{Combined}(\varphi_{\mathcal{L}})$, where $\varphi_{\mathcal{L}}$ is given in Ex. 1; the atomic blocks accessing global variables are totally ordered.

4.4 Removing code fragments from the original methods

The library obtained by adding combined methods still has executions with data-flow dependency cycles described by formulas in $\Lambda[\mathcal{L}]$. One needs to modify the code of the methods copied from $\mathcal{L}$ s.t. they do not generate instances satisfying formulas in $\Lambda[\mathcal{L}]$.

For simplicity, let us consider the case when $\Lambda[\mathcal{L}]$ contains only a formula φ. For any control location l in φ, $\mathit{Pref}[\varphi,l](v\mathtt{op},I)$ denotes the formula that describes the prefix of the interleavings characterized by φ, which end in l; $v\mathtt{op}$ denotes the free variable of $\mathit{Pref}[\varphi,l]$ that is interpreted as the instance reaching l and I denotes all the other free variables. $\mathit{Pref}[\varphi,l](v\mathtt{op},I)$ contains all the predicates that constrain (local states at) control locations which are less than l in the partial order defined by the ordering constraints in φ and the program order.

Example 3. Let us consider the formula $\varphi_{\mathcal{L}}$ computed in Fig. 4 and the action associated with the atomic block b. Then, $\mathit{Pref}[\varphi_{\mathcal{L}},b](v\mathtt{push},v\mathtt{pop}) = v\mathtt{pop}.a1 < v\mathtt{push}.b$, where $v\mathtt{push}$ and $v\mathtt{pop}$ denote an instance of the `push`, respectively `pop`, method. Note that

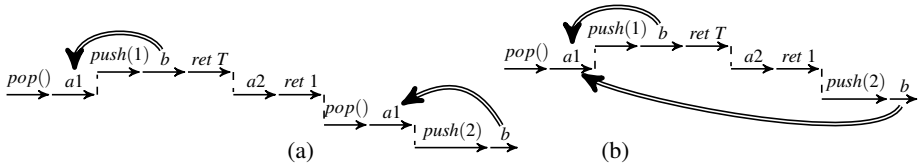


Fig. 6: (a) An execution of $\mathcal{L}_s$. (b) An execution that contains actions from `push` and `pop` but is not an execution of $\mathcal{L}_s$. Both executions are models of $\mathit{Pref}[\varphi_{\mathcal{L}}, b](v\texttt{push}, v\texttt{pop})$ (double arrows emphasize the order relation between b and $a1$).

$\mathit{Pref}[\varphi_{\mathcal{L}}, b](v\texttt{push}, v\texttt{pop})$ captures only one direction of the communication between `push` and `pop`. Also, $\mathit{Pref}[\varphi_{\mathcal{L}}, a2](v\texttt{push}, v\texttt{pop}) = \varphi_{\mathcal{L}}$.

The original methods in $\mathcal{L}$ are modified by removing statements located at control locations appearing in φ. We remove those statements that are executed only in instances which can be generated by the combined methods. A control location l is removed from the CFG of the method op if the CFG contains a unique path starting in l and all the executions of $\mathcal{L}$ satisfy two invariants $\mathsf{Inv1}(l, \varphi)$ and $\mathsf{Inv2}(l, \varphi)$, defined in the following. We begin by explaining them on our running example.

In the continuation of Ex. 2, suppose that we want to eliminate the control location b from `push`. Intuitively, the invariant $\mathsf{Inv1}(b, \varphi_{\mathcal{L}})$ states that, an instance of `push` reaches b iff there is an instance of `pop` that is ready to exchange values with the considered instance of `push`. Formally, in any execution, for any instance of `push` that reaches b there is a `pop` instance that reaches $a1$, which is expressed by the following formula:

$$\forall v\texttt{push}\ \exists v\texttt{pop}.\ v\texttt{push}.b \rightarrow v\texttt{pop}.a1 < v\texttt{push}.b \tag{1}$$

Intuitively, $\mathsf{Inv2}(b, \varphi_{\mathcal{L}})$ states that a `pop` exchanges values with at most one `push`; that is, in any execution, if there are two distinct instances of `push` that reach the atomic block b, then there are two distinct instances of `pop` such that each of them sends its job to a different `push` instance (when reaching $a1$). Formally, for any two distinct instances of `push`, denoted by $v\texttt{push}$, $v\texttt{push}'$, there are two distinct instances of `pop`, denoted by $v\texttt{pop}$, $v\texttt{pop}'$, such that $\mathit{Pref}[\varphi_{\mathcal{L}}, b](v\texttt{push}, v\texttt{pop}) \wedge \mathit{Pref}[\varphi_{\mathcal{L}}, b](v\texttt{push}', v\texttt{pop}')$ holds:

$$\forall v\texttt{push}, v\texttt{push}'\ \exists v\texttt{pop}, v\texttt{pop}'.\ \big(v\texttt{push} \neq v\texttt{push}' \wedge v\texttt{push}.b \wedge v\texttt{push}'.b\big) \rightarrow \\ \big(v\texttt{pop} \neq v\texttt{pop}' \wedge v\texttt{pop}.a1 < v\texttt{push}.b \wedge v\texttt{pop}'.a1 < v\texttt{push}'.b\big)$$

Any execution that satisfies these two properties can be rewritten such that any pair of instances of `push` and `pop` as above (reaching b and $a1$, respectively) becomes an instance of some method `push+pop`. The execution in Fig. 6(a) satisfies both invariants while the execution in Fig. 6(b) satisfies only the first one.

These two invariants, together with the ones describing the conditions under which a_2 can be eliminated, state that one instance of `push` exchanges values with exactly one instance of `pop`. The code of `push` and `pop` in the new library is given in Fig. 5. The methods have the same names, but the atomic blocks b and $a2$ have been removed. Notice that, one cannot remove $a1$ because $a1$ has two successors. The algorithm has to preserve those instances where $a3$ is reached from $a1$.

A control location l is removed from the CFG of a method op of $\mathcal{L}$ iff its CFG contains a unique path starting in l and all executions of $\mathcal{L}$ satisfy the following invariants:

Inv1(l,φ)**:** any instance of op ending in l is a sub-sequence of an instance of one of the combined methods generated using φ. Since the combined methods are defined from $\mathcal{L}$ and φ, this invariant over the executions of $\mathcal{L}$ can be expressed as follows:

$$\forall v\mathsf{op}\ \exists I.\ v\mathsf{op}.l \rightarrow \mathit{Pref}[\varphi,l](v\mathsf{op},I)$$

i.e., for every instance $v$op of op there exist other method instances, denoted by I, such that if $v$op reaches the control location l, then the interleaving between $v$op and I is a prefix of one of the interleavings defined by φ;

Inv2(l,φ)**:** in any execution, each instance of op ending in l is a sub-sequence of a different instance of a combined method generated using φ:

$$\forall v\mathsf{op}, v\mathsf{op}'\ \exists I\ \exists I'.\ \begin{pmatrix} v\mathsf{op} \neq v\mathsf{op}' \wedge \\ v\mathsf{op}.l \wedge v\mathsf{op}'.l \end{pmatrix} \rightarrow \begin{pmatrix} \bigwedge_{x\in I, y\in I'} x \neq y \wedge \\ \mathit{Pref}[\varphi,l](v\mathsf{op},I) \wedge \mathit{Pref}[\varphi,l](v\mathsf{op}',I') \end{pmatrix}$$

To understand this invariant, suppose that it is not true and there exists a method instance that reaches l which is part of two distinct models of $\mathit{Pref}[\varphi,l]$. Note that it is not possible to represent these two models by two instances of a combined method (this holds because the instance that reaches l should be included in both).

Given a formula $\varphi \in \Lambda[\mathcal{L}]$, if none of the control locations l that appear in φ were removed from the code of the original methods, then all the methods in `Combined`(φ) are removed from the new library.

4.5 Invariant checking

The invariants are *proved automatically* using a variation of the thread modular rely-guarantee analysis [10] over the concurrent system defined by the most general client of the library $\mathcal{L}$. The rely-guarantee analysis selects a statement of an arbitrary thread and executes it to generate *environment transitions*. The latter are obtained by existentially quantifying all local variables from the relation describing the effect of the selected statement. The environment transitions are subsequently applied to discover new transitions, until a fixed-point is reached.

For each invariant, our analysis needs to show that a given block of code executes only in a certain concurrent context. To compute a precise-enough over-approximation of the concurrent system, the analysis keeps a number of explicit copies of threads, called *reference threads*. All the other threads are called *environment threads*. The local states of the environment threads are described using a set of counters, each of them associated with a predicate over the local variables of an environment thread and the reference threads. These counters keep the number of environment threads that satisfy a certain predicate (over their local/global variables). The environment transitions refer to local variables of reference threads, global variables, and counters. Abstract states and environment transitions are represented by elements of a product between the abstract domains in [4] and a finite abstract domain $\{0,1,+\infty\}$ describing the counters values.

The number of reference threads depends on the invariant, i.e., it is equal to the number of universally quantified variables in the invariant. The predicates are also derived from the invariant. For each invariant *inv*, we associate (1) a predicate `PC`$=l$, for each control location l such that $v$op.l appears in *inv* and $v$op is existentially quantified, (this predicate holds iff the thread is at l) and (2) a predicate P, for each $v.a \rightarrow P$ an atomic

$e1 ::= \ \mathtt{g}' = \mathtt{g} \wedge (\mathtt{ls}(\mathtt{S},\mathtt{NULL}) * \mathtt{S}' \mapsto \mathtt{S}) \wedge c' = c$

$e2 ::= \ \mathtt{g}' = \mathtt{g} \wedge (\mathtt{ls}(\mathtt{S}',\mathtt{NULL}) * \mathtt{S} \mapsto \mathtt{S}') \wedge c' = c$

$e3 ::= \ \mathtt{g} = \mathtt{F} \wedge \mathtt{g}' = \mathtt{W} \wedge \mathtt{g2} = \mathtt{g2}' = \mathtt{NULL} \wedge \mathbf{Eq}(\mathtt{S},\mathtt{S}') \wedge c' = c+1$

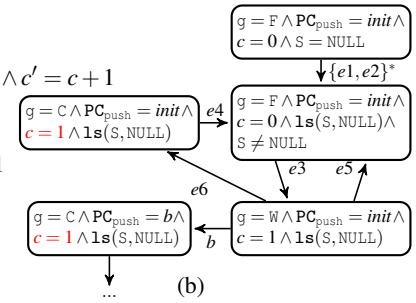
$e4 ::= \ \mathtt{g} = \mathtt{C} \wedge \mathtt{g}' = \mathtt{F} \wedge \mathtt{g2} \neq \mathtt{NULL} \wedge \mathtt{g2}' = \mathtt{NULL} \wedge$
$\mathbf{Eq}(\mathtt{S},\mathtt{S}') \wedge c > 0 \wedge c' = c-1$

$e5 ::= \ \mathtt{g} = \mathtt{W} \wedge \mathtt{g}' = \mathtt{F} \wedge \mathbf{Eq}(\mathtt{S},\mathtt{S}') \wedge c > 0 \wedge c' = c-1$

$e6 ::= \ \mathtt{g} = \mathtt{W} \wedge \mathtt{g}' = \mathtt{C} \wedge \mathtt{g2} = \mathtt{NULL} \wedge \mathtt{g2}' \neq \mathtt{NULL} \wedge$
$\mathbf{Eq}(\mathtt{S},\mathtt{S}') \wedge c' = c$

$Env = \{e1, e2, e3, e4, e5, e6\}$

(a)



(b)

Fig. 7: (a) Abstraction of the concurrent system analyzed to prove *inv*1: $e1$ and $e2$ are the actions corresponding to pushing and popping an element in a stack, respectively; e3,e4,e5,e6 are the transitions over-approximating the elimination mechanism. (b) A part of abstract reachability graph with a reference thread executing `push`.

formula in *inv* such that v is existentially quantified (the local variables of the universal instance variables are substituted by the local variables of the reference threads).

Example 4. To prove $\mathsf{Inv1}(b, \varphi_{\mathcal{L}})$ in (1), we need to count how many threads are at the control location reached after executing $a1$. Let c be a counter associated with the predicate $\mathtt{PC}{=}a1$. Fig. 7(a) presents the set of environment transitions Env computed by our rely-guarantee analysis for $\mathcal{L}_s$ with counter c and one reference thread executing `push` (they are represented by formulas in Separation Logic containing, as usual, primed and unprimed variables). The predicate $\mathtt{ls}(\mathtt{S},\mathtt{NULL})$ denotes a singly linked list starting at `S` and ending in `NULL`, $\mathtt{S} \mapsto \mathtt{S}'$ states that the field `next` of `S` points to the memory cell pointed to by $\mathtt{S}'$; the macro $\mathbf{Eq}(\mathtt{S},\mathtt{S}')$ is used to say that the memory region reached from `S` did not changed. In Fig. 7(b), we present a part of the reachability graph obtained by executing Env in parallel with the reference thread executing `push`. On this reachability graph, we check that whenever the atomic block b is executed by the reference thread, the value of the counter c is greater or equal to one. Since $c = 1$, there is exactly one thread that executed the atomic block $a1$ from `pop`, so $\mathsf{Inv1}(b, \varphi_{\mathcal{L}})$ holds.

5 Correctness of the rewriting algorithm

In this section, we show the relation between the linearizability of the original library and the linearizability of the library obtained by applying the rewriting algorithm.

First, we show that the library $\mathcal{L}^n$, obtained by applying the rewriting algorithm on the library $\mathcal{L}$, preserves the behaviors of the original library, and that the specification $\mathcal{S}^n$ contains only sequential executions that are allowed by $\mathcal{S}$. That is, (i) every execution e of $\mathcal{L}$ can be rewritten into an execution e' of $\mathcal{L}^n$ s.t. interleavings of operations from e, which define *sets* of data-flow dependency cycles, are transformed in e' into instances of combined methods and (ii) every $s' \in \mathcal{S}^n$ is the rewriting of exactly one $s \in \mathcal{S}$.

Formally, an execution $e \in [\![\mathcal{L}]\!]$ is *similar* to $e' \in [\![\mathcal{L}^n]\!]$, denoted by $e \sim e'$, iff there is a function that transforms e into e' s.t. (1) there is a total function π between the threads of e' and sets of threads of e, (2) for each state and thread t of e', the set of thread-local states in $\pi(t)$ is aggregated into one thread-local state of t (the thread id's of $\pi(t)$ become local variables in the new state) and (3) the transformation preserves all actions

of e, in the order they occur in e, except for some `call` and `return` actions, that are associated to `skip` actions. The first item states that for any execution e of the original library $\mathcal{L}$ there exists an execution e' of the new library $\mathcal{L}^n$ s.t. $e \sim e'$. This is denoted by $[\![\mathcal{L}]\!] \subseteq_{\sim} [\![\mathcal{L}^n]\!]$. The second item states that, for any $s' \in \mathcal{S}^n$, there exists exactly one $s \in \mathcal{S}$ s.t. $s' \sim^{-1} s$, where $\sim^{-1}$ is the inverse of $\sim$. This is denoted by $\mathcal{S}^n \subseteq^{!}_{\sim^{-1}} \mathcal{S}$.

Example 5. The execution e given in Fig. 3(a) is similar to the execution e' given in Fig. 3(b), i.e., $e \sim e'$. Also, the linearization s in Fig. 3(a) is similar to the linearization s' in Fig. 3(b), i.e., $s' \sim^{-1} s$.

Theorem 1. *Let $\mathcal{L}^n$ and $\mathcal{S}^n$ be the output of the rewriting algorithm for the input library $\mathcal{L}$ and its specification $\mathcal{S}$. Then,*

$$[\![\mathcal{L}]\!] \subseteq_{\sim} [\![\mathcal{L}^n]\!] \ \textit{and} \ \mathcal{S}^n \subseteq^{!}_{\sim^{-1}} \mathcal{S}.$$

Theorem 1 follows from the following lemma, which states the necessary conditions for removing statements from the CFG of a method in $\mathcal{L}$.

Lemma 1. *Let $\mathcal{L}$ be a library, φ a $\mathbb{CL}$ formula, and l a control location in a method* op *of $\mathcal{L}$ such that there is a unique path in the CFG starting from l. If $Inv_1(l,\varphi)$ and $Inv_2(l,\varphi)$ hold for all executions of the most general client of $\mathcal{L}$, then any execution e in $[\![\mathcal{L}]\!]$ is similar to an execution e' in $[\![\mathcal{L}^n]\!]$ such that all the operations from e reaching l correspond in e' to sub-sequences of instances of methods in* `Combined`(φ).

We introduce a stronger version of linearizability, called *R-linearizability*, and show that the *R*-linearizability of the new library $\mathcal{L}^n$, generated by the rewriting algorithm, w.r.t. $\mathcal{S}^n$ implies the linearizability of the original library $\mathcal{L}$ w.r.t. $\mathcal{S}$. One needs to prove the *R*-linearizability of the new library, instead of classical linearizability, because the induced rewriting on executions forgets ordering constraints between non-overlapping operations, more precisely, between pairs of operations such that at least one of them is rewritten (together with other operations) into a combined operation.

For example, let e' be the execution from Fig. 3(b) of the concurrent stack $\mathcal{L}^n_s$ given in Fig. 5. Because the instances of `push` + `pop` and `push` are overlapping, e' has two correct linearizations w.r.t. $\mathcal{S}^n_s$, i.e.,

$s'_1 = push + pop(1)ret(1)\ push(4)\ ret(T)$ and $s'_2 = push(4)\ ret(T)\ push + pop(1)ret(1)$.

The only executions similar to s'_1 and s'_2 in the specification $\mathcal{S}$ (see Th. 1) are

$s_1 = push(1)ret(T)pop()ret(1)push(4)\ ret(T)$, $s_2 = push(4)\ ret(T)push(1)ret(T)pop()ret(1)$,

respectively. We recall that the execution e in Fig. 3(a) is similar to e'. Notice that, the classical linearizability of e' is not sufficient to imply the linearizability of e, because not every linearization of e' leads to a linearization of e. For example, s_2, which is similar to s'_2 is not a correct linearization of e because the two operations of `push` in e are non-overlapping and the order between them is not respected by s_2.

To overcome this problem, the rewriting distinguishes for each combined method op two control locations: $\texttt{c}^{\underline{\texttt{op}}}\texttt{pc}$ is the `skip` statement replacing the last `call` action and $\texttt{r}^{\underline{\texttt{op}}}\texttt{pc}$ is the `skip` statement replacing the first `return` action in the interleaving represented by op (Fig. 5 distinguishes the control locations $\texttt{c}^{\underline{\texttt{op}}}\texttt{pc}$ and $\texttt{r}^{\underline{\texttt{op}}}\texttt{pc}$ for the method `push` + `pop`). For the methods inherited from the original library, $\texttt{c}^{\underline{\texttt{op}}}\texttt{pc}$ and $\texttt{r}^{\underline{\texttt{op}}}\texttt{pc}$ are the locations of their `call` and `return` actions. Then, instead of showing the linearizability of $\mathcal{L}^n$ w.r.t. $\mathcal{S}^n$, we show that for any execution e' in $[\![\mathcal{L}^n]\!]$ there exists a

sequential execution s' in $\mathcal{S}^n$ s.t. 1) for each operation in e' there is an operation in s' with the same interface and (2) s' preserves the order between non overlapping sequences of actions of the same method starting at $\mathtt{c}^{\mathtt{op}}\_\mathtt{pc}$ and ending at $\mathtt{r}^{\mathtt{op}}\_\mathtt{pc}$.

Formally, let $\mathcal{L}$ be a library and Rcp a mapping that associates to each method op of $\mathcal{L}$ two distinguished statements which are not on a loop, denoted $\mathtt{c}^{\mathtt{op}}\_\mathtt{pc}$ and $\mathtt{r}^{\mathtt{op}}\_\mathtt{pc}$. An execution e of $\mathcal{L}$ is *R-linearizable w.r.t. $\mathcal{S}$ and* Rcp iff e is linearizable w.r.t $\mathcal{S}$ and for each invocation *op* of op in e, one can choose a linearization point located between the execution of the actions associated with $\mathtt{c}^{\mathtt{op}}\_\mathtt{pc}$ and $\mathtt{r}^{\mathtt{op}}\_\mathtt{pc}$. The sequential execution in $\mathcal{S}$ corresponding to this sequence of linearization points is called the *R-linearization* of e. When Rcp contains only the call and return actions of each method, *R*-linearizability coincides with the classical definition of linearizability. The rewriting algorithm identifies a mapping Rcp for the new library such that for any execution e of $\mathcal{L}$, if the execution e' in $[\![\mathcal{L}^n]\!]$ to which it is similar, i.e., $e \sim e'$, is *R*-linearizable, i.e. there exists s' an *R*-linearization of e', then any $s \in \mathcal{S}$ such that $s \sim s'$ is a correct linearization of e.

For example, the *R*-linearization of the execution e' in Fig. 3(b) w.r.t. the mapping Rcp defined as above is s'_1. Thus, s_1 is a linearization of the execution e in Fig. 3(a).

Theorem 2. *Let $\mathcal{L}^n$ and $\mathcal{S}^n$ be the library, resp. the specification, obtained by rewriting a library $\mathcal{L}$ and its specification $\mathcal{S}$. The R-linearizability of $\mathcal{L}^n$ w.r.t. $\mathcal{S}^n$ and the mapping* Rcp *defined by the rewriting algorithm implies the linearizability of $\mathcal{L}$ w.r.t. $\mathcal{S}$.*

6 Experimental results

The crucial steps of the rewriting, generating the set of formulas $\Lambda[\mathcal{L}]$ representing dependency cycles and checking the invariants in Sec. 4.4, are implemented into a prototype tool, which is based on the abstract domain in [1]. Table 1 presents a summary of the obtained results [3]. For each of the considered libraries, $\#op_{\mathcal{L}}$, resp. $\#op_{\mathcal{L}^n}$, is the number of methods in the original library, resp. the new library, #*cycles* is the number of cycles discovered by *get_cycle*, #*pcycles* is the number of cycles, among the ones discovered by *get_cycle*, for which the abstract concurrent analysis proved the required invariants, and #*pc* is the number of control points removed from the original methods.

The implementation of Fetch&Add allows an unbounded number of operations to have the same linearization point. We have analyzed a modified version of this implementation, that introduces a bound on the number of operations whose task can be performed by the same concurrently executing operation. Similarly, the implementations based on "flat combining" [8] are tractable by our method if we bound the number of threads that make public the signature of their updates in a given time interval. This restriction is natural when implementing concurrent objects in a real programing language because otherwise, the library has runs without progress on the data structure.

All the libraries we obtained after the rewriting are *R*-linearizable. Furthermore, all their methods have *internal linearization points* (e.g., the action associated with b is the linearization point of push + pop in Fig. 5), which allows us to prove their *R*-linearizability using existing techniques, e.g., [3, 17]. The only exception is Queue Elimination [12] which is not *R*-linearizable. However, the library obtained by rewriting Queue Elimination is linearizable and this implies the linearizability of the original library because of some closure properties of the queue specification.

[3] More details are available at http://pub.ist.ac.at/~cezarad/lin2lin.html

Table 1: Experimental results on an Intel Core i3 2.4 GHz with 2GB of memory.

library $\mathcal{L}$	*size of* $\mathcal{L}$	*get_cycle*			*inv check*			size of $\mathcal{L}^n$
	#op	k	#cycles	time(sec)	#pcycles	time(sec)	#pc	#op
`Simple Stack Elim.`	2	2	1	<1	1	<1	2	3
`Stack Elim.`	1	2	2	<3	2	<3	6	8
`Modified Stack Elim.`	2	2	2	<3	2	<3	6	9
`Fetch&Add`	1	3	4	<3	4	<4	3	8

References

1. CINV. `http://www.liafa.univ-paris-diderot.fr/cinv/`.
2. Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 477–490, 2007.
3. Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 399–413, 2008.
4. Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *Proc. of PLDI*, pages 578–589, 2011.
5. Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. Model checking of linearizability of concurrent list implementations. In *Proc. of CAV*, volume 6174 of *LNCS*, pages 465–479, 2010.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL*, pages 238–252, 1977.
7. Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.
8. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. of SPAA*, pages 355–364, 2010.
9. Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proc. of SPAA*, pages 206–215, 2004.
10. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *Proc. of CAV*, pages 262–274, 2003.
11. Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
12. Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proc. of SPAA*, pages 253–262, 2005.
13. Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *Proc. of PODC*, pages 85–94, 2010.
14. Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.
15. Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to prove algorithms linearisable. In *Proc. of CAV*, volume 7358 of *LNCS*, pages 243–259, 2012.
16. Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *J. Parallel Distrib. Comput.*, 60(11), 2000.
17. Viktor Vafeiadis. Automatically proving linearizability. In *Proc. of CAV*, volume 6174 of *LNCS*, pages 450–464, 2010.