Universal Constructions that Ensure Disjoint-Access Parallelism and Wait-Freedom

Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, Corentin Travers

To cite this version:

Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, Corentin Travers. Universal Constructions that Ensure Disjoint-Access Parallelism and Wait-Freedom. [Research Report] 2012. <hal-00697198v2>

HAL Id: hal-00697198

https://hal.inria.fr/hal-00697198v2

Submitted on 15 May 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Universal Constructions that Ensure Disjoint-Access Parallelism and Wait-Freedom*

Faith Ellen‡
University of Toronto
faith@cs.toronto.edu

Panagiota Fatourou§
University of Crete & FORTH-ICS
faturu@csd.uoc.gr

Eleftherios Kosmas¶
University of Crete & FORTH-ICS
ekosmas@csd.uoc.gr

Alessia Milani‖**
University of Bordeaux
milani@labri.fr

Corentin Travers††
University of Bordeaux
travers@labri.fr

Abstract

Disjoint-access parallelism and wait-freedom are two desirable properties for implementations of concurrent objects. *Disjoint-access parallelism* guarantees that processes operating on different parts of an implemented object do not interfere with each other by accessing common base objects. Thus, disjoint-access parallel algorithms allow for increased parallelism. *Wait-freedom* guarantees progress for each nonfaulty process, even when other processes run at arbitrary speeds or crash.

A *universal construction* provides a general mechanism for obtaining a concurrent implementation of any object from its sequential code. We identify a natural property of universal constructions and prove that there is no universal construction (with this property) that ensures both disjoint-access parallelism and wait-freedom. This impossibility result also holds for transactional memory implementations that require a process to re-execute its transaction if it has been aborted and guarantee each transaction is aborted only a finite number of times.

Our proof is obtained by considering a dynamic object that can grow arbitrarily large during an execution. In contrast, we present a universal construction which produces concurrent implementations that are both wait-free and disjoint-access parallel, when applied to objects that have a bound on the number of data items accessed by each operation they support.

Topics: Distributed algorithms: design, analysis, and complexity; Shared and transactional memory, synchronization protocols, concurrent programming

Keywords: concurrent programming, disjoint-access parallelism, wait-freedom, universal construction, impossibility result

*An extending abstract of this work appeared in Proceedings of the 31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'12), ACM press, 2012

†Supported in part by the Natural Science and Engineering Research Council of Canada.

‡Some of the work was done while visiting Labri CNRS-UMR 5800, Bordeaux, France.

§Supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639), Hi-PEAC2 (FP7-ICT-217068), and ENCORE (FP7-ICT-248647) projects.

¶Supported by the project "IRAKLITOS II - University of Crete" of the Operational Programme for Education and Lifelong Learning 2007 - 2013 (E.P.E.D.V.M.) of the NSRF (2007 - 2013), co-funded by the European Union (European Social Fund) and National Resources.

‖Supported in part by the ANR project DISPLEXITY.

****Contact Author.** Labri, University of Bordeaux 1, 351, cours de la Liberation F-33405 Talence cedex, France. Tel:+33 (0)5 40 00 35 03. E-mail : milani@labri.fr

††Supported in part by the ANR project DISPLEXITY.

1 Introduction

Due to the recent proliferation of multicore machines, simplifying concurrent programming has become a necessity, to exploit their computational power. A *universal construction* [20] is a methodology for automatically executing pieces of sequential code in a concurrent environment, while ensuring correctness. Thus, universal constructions provide functionality similar to Transactional Memory (TM) [22]. In particular, universal constructions provide concurrent implementations of any sequential data structure: Each operation supported by the data structure is a piece of code that can be executed.

Many existing universal constructions [1, 12, 15, 16, 19, 20] restrict parallelism by executing each of the desired operations one after the other. We are interested in universal constructions that allow for increased parallelism by being disjoint-access parallel. Roughly speaking, an implementation is *disjoint-access parallel* if two processes that operate on disjoint parts of the simulated state do not interfere with each other, i.e., they do not access the same base objects. Therefore, disjoint-access parallelism allows unrelated operations to progress in parallel. We are also interested in ensuring strong progress guarantees: An implementation is *wait-free* if, in every execution, each (non-faulty) process completes its operation within a finite number of steps, even if other processes may fail (by crashing) or are very slow.

In this paper, we present both positive and negative results. We first identify a natural property of universal constructions and prove that designing universal constructions (with this property) which ensure both disjoint access parallelism and wait-freedom is not possible. We prove this impossibility result by considering a dynamic data structure that can grow arbitrarily large during an execution. Specifically, we consider a singly-linked unsorted list of integers that supports the operations $\textsc{Append}(L, x)$, which appends x to the end of the list L, and $\textsc{Search}(L, x)$, which searches the list L for x starting from the first element of the list. We show that, in any implementation resulting from the application of a universal construction to this data structure, there is an execution of $\textsc{Search}$ that never terminates.

Since the publication of the original definition of disjoint-access parallelism [24], many variants have been proposed [2, 9, 18]. These definitions are usually stated in terms of a conflict graph. A *conflict graph* is a graph whose nodes is a set of operations in an execution. An edge exists between each pair of operations that conflict. Two operations *conflict* if they access the same data item. A *data item* is a piece of the sequential data structure that is being simulated. For instance, in the linked list implementation discussed above, a data item may be a list node or a pointer to the first or last node of the list. In a variant of this definition, an edge between conflicting operations exists only if they are concurrent. Two processes *contend* on a base object, if they both access this base object and one of these accesses is a *non-trivial* operation (i.e., it may modify the state of the object). In a disjoint-access parallel implementation, two processes performing operations op and op' can contend on the same base object only if the conflict graph of the minimal execution interval that contains both op and op' satisfies a certain property. Different variants of disjoint-access parallelism use different properties to restrict access to a base object by two processes performing operations. Note that any data structure in which all operations access a common data item, for example, the root of a tree, is trivially disjoint access parallel under all these definitions.

For the proof of the impossibility result, we introduce *feeble disjoint-access parallelism*, which is weaker than all existing disjoint-access parallelism definitions. Thus, our impossibility result still holds if we replace our disjoint-access parallelism definition with any existing definition of disjoint-access parallelism.

Next, we show how this impossibility result can be circumvented, by restricting attention to data structures whose operations can each only access a bounded number of different data items. Specifically, there is a constant b such that any operation accesses at most b different data items when it is applied sequentially to the data structure, starting from any (legal) state. Stacks and queues are examples of dynamic data structures that have this property. We present a universal construction that ensures wait-freedom and disjoint-access parallelism for such data structures. The resulting concurrent implementations are linearizable [23] and satisfy a much stronger disjoint-access parallelism property than we used to prove the impossibility result.

Disjoint-access parallelism and its variants were originally formalized in the context of fixed size data structures, or when the data items that each operation accesses are known when the operation starts its execution. Dealing with these cases is much simpler than considering an arbitrary dynamic data structure where the set of data items accessed by an operation may depend on the operations that have been previously executed and on the operations that are performed concurrently.

The universal construction presented in this paper is the first that provably ensures both wait-freedom and disjoint-access parallelism for dynamic data structures in which each operation accesses a bounded number of data items. For other dynamic data structures, our universal construction still ensures linearizability and disjoint-access parallelism. Instead of wait-freedom, it ensures that progress is *non-blocking*. This guarantees that, in every execution, from every (legal) state, *some* process finishes its operation within a finite number of steps.

2 Related Work

Some impossibility results, related to ours, have been provided for transactional memory algorithms. Transactional Memory (TM) [22] is a mechanism that allows a programmer of a sequential program to identify those parts of the sequential code that require synchronization as *transactions*. Thus, a transaction includes a sequence of operations on data items. When the transaction is being executed in a concurrent environment, these data items can be accessed by several processes simultaneously. If the transaction commits, all its changes become visible to other transactions and they appear as if they all take place at one point in time during the execution of the transaction. Otherwise, the transaction can abort and none of its changes are applied to the data items.

Universal constructions and transactional memory algorithms are closely related. They both have the same goal of simplifying parallel programming by providing mechanisms to efficiently execute sequential code in a concurrent environment. A transactional memory algorithm informs the external environment when a transaction is aborted, so it can choose whether or not to re-execute the transaction. A call to a universal construction returns only when the simulated code has been successfully applied to the simulated data structure. This is the main difference between these two paradigms. However, it is common behavior of an external environment to restart an aborted transaction until it eventually commits. Moreover, meaningful progress conditions [11, 30] in transactional memory require that the number of times each transaction aborts is finite. This property is similar to the *wait-freedom* property for universal constructions. In a recent paper [11], this property is called *local progress*. Our impossibility result applies to transactional memory algorithms that satisfy this progress property. Disjoint-access parallelism is defined for transactions in the same way as for universal constructions.

Strict disjoint-access parallelism [18] requires that an edge exists between two operations (or transactions) in the conflict graph of the minimal execution interval that contains both operations (transactions) if the processes performing these operations (transactions) contend on a base object. A TM algorithm is *obstruction-free* if a transaction can be aborted only when contention is encountered during the course of its execution. In [18], Guerraoui and Kapalka proved that no obstruction-free TM can be strictly disjoint access parallel. Obstruction-freedom is a weaker progress property than wait-freedom, so their impossibility result also applies to wait-free implementations (or implementations that ensure local progress). However, it only applies to this strict variant of disjoint-access parallelism, while we consider a much weaker disjoint-access parallelism definition. It is worth-pointing out that several obstruction-free TM algorithms [17, 21, 25, 28] satisfy a weaker version of disjoint-access parallelism than this strict variant. It is unclear whether helping, which is the major technique for achieving strong progress guarantees, can be (easily) achieved assuming strict disjoint-access parallelism. For instance, consider a scenario where transaction T_1 accesses data items x and y, transaction T_2 accesses x, and T_3 accesses y. Since T_2 and T_3 access disjoint data items, strict disjoint-access parallelism says that they cannot contend on any common base objects. In particular, this limits the help that each of them can provide to T_1.

Bushkov *et al.* [11] prove that no TM algorithm (whether or not it is disjoint-access parallel) can ensure local progress. However, they prove this impossibility result under the assumption that the TM algorithm does not have access to the code of each transaction (and, as mentioned in their introduction, their impossibility result does not hold without this restriction). In their model, the TM algorithm allows the external environment to invoke actions for reading a data item, writing a data item, starting a transaction, and trying to commit or abort it. The TM algorithm is only aware of the sequence of invocations that have been performed. Thus, a transaction can be helped only after the TM algorithm knows the entire set of data items that the transaction should modify. However, there are TM algorithms that do allow threads to have access to the code of transactions. For instance, RobuSTM [30] is a TM algorithm in which the code of a transaction is made available to threads so that they can help one another to ensure strong progress guarantees.

Proving impossibility results in a model in which the TM algorithm does not have access to the code of transactions is usually done by considering certain high-level histories that contain only invocations and responses of high-level operations on data items (and not on the base objects that are used to implement these data items in a concurrent environment). Our model gives the universal construction access to the code of an invoked operation. Consequently, to prove our impossibility result we had to work with low-level histories, containing steps on base objects, which is technically more difficult.

Attiya *et al.* [9] proved that there is no disjoint-access parallel TM algorithm where read-only transactions are wait-free and *invisible* (i.e., they do not apply non-trivial operations on base objects). This impossibility result is proved for the variant of disjoint-access parallelism where processes executing two operations (transactions) concurrently contend on a base object only if there is a path between the two operations (transactions) in the conflict graph. We prove our lower bound for a weaker definition of disjoint-access parallelism and it applies even for implementations with visible reads. We remark that the impossibility result in [9] does not contradict our algorithm, since our implementation employs *visible* reads.

In [26], the concept of *MV-permissiveness* was introduced. A TM algorithm satisfies this property if a transaction aborts only when it is an update transaction that conflicts with another update transaction. An *update transaction* contains updates to data items. The paper [26] proved that no transactional memory algorithm satisfies both disjoint access parallelism (specifically, the variant of disjoint-access parallelism presented in [9]) and MV-permissiveness. However, the paper assumes that the TM algorithm does not have access to the code of transactions and is based on the requirement that the code for creating, reading, or writing data items terminates within a finite number of steps. This lower bound can be beaten if this requirement is violated. Attiya and Hillel [8] presented a strict disjoint-access parallel lock-based TM algorithm that satisfies MV-permissiveness.

More constraining versions of disjoint-access parallelism are used when designing algorithms [5, 6, 24]. Specifically, two operations are allowed to access the same base object if they are connected by a path of length at most d in the conflict graph [2, 5, 6]. This version of disjoint-access parallelism is known as the *d-local contention property* [2, 5, 6]. The first wait-free disjoint-access parallel implementations [24, 29] had $O(n)$-local contention, where n is the number of processes in the system, and assumed that each operation accesses a fixed set of data items. Afek *et al.* [2] presented a wait-free, disjoint-access parallel universal construction that has $O(k + log^*n)$-local contention, provided that each operation accesses at most k predetermined memory locations. It relies heavily on knowledge of k. This work extends the work of Attiya and Dagan [5], who considered operations on pairs of locations, i.e. where $k = 2$. Afek *et al.* [2] leave as an open question the problem of finding highly concurrent wait-free implementations of data structures that support operations with no bounds on the number of data items they access. In this paper, we prove that, in general, there are no solutions unless we relax some of these properties.

Attiya and Hillel [7] provide a k-local non-blocking implementation of k-read-modify-write objects. The algorithm assumes that double-compare-and-swap (`DCAS`) primitives are available. A `DCAS` atomically executes `CAS` on two memory words. Combining the algorithm in [7] and the non-blocking implementation of `DCAS` by Attiya and Dagan [5] results in a $O(k + log^*n)$-local non-blocking implementation of a k-read-modify-write object that only relies on single-word `CAS` primitives. Their algorithm can be adapted to work for operations whose data set is defined on the fly, but it only ensures that progress is non-blocking.

A number of wait-free universal constructions [1, 15, 16, 19, 20] work by copying the entire data structure locally, applying the active operations sequentially on their local copy, and then changing a shared pointer to point to this copy. The resulting algorithms are not disjoint access parallel, unless vacuously so.

Anderson and Moir [3] show how to implement a k-word atomic `CAS` using `LL/SC`. To ensure wait-freedom, a process may help other processes after its operation has been completed, as well as during its execution. They employ their k-word `CAS` implementation to get a universal construction that produces wait-free implementations of multi-object operations. Both the k-word `CAS` implementation and the universal construction allow operations on different data items to proceed in parallel. However, they are not disjoint-access parallel, because some operations contend on the same base objects even if there are no (direct or transitive) conflicts between them. The helping technique that is employed by our algorithm combines and extends the helping techniques presented in [3] to achieve both wait-freedom and disjoint-access parallelism.

Anderson and Moir [4] presented another universal construction that uses indirection to avoid copying the entire data structure. They store the data structure in an array which is divided into a set of consecutive data blocks. Those blocks are addressed by a set of pointers, all stored in one `LL/SC` object. An adaptive version of this algorithm is presented in [15]. An algorithm is *adaptive* if its step complexity depends on the maximum number of active processes at each point in time, rather than on the total number n of processes in the system. Neither of these universal constructions is disjoint-access parallel.

Barnes [10] presented a disjoint-access parallel universal construction, but the algorithms that result from this universal construction are only non-blocking. In Barnes' algorithm, a process p executing an operation op first simulates the execution of op locally, using a local dictionary where it stores the data items accessed during the simulation of op and their new values. Once p completes the local simulation of op, it tries to lock the data items stored in its dictionary. The data items are locked in a specific order to avoid deadlocks. Then, p applies the modifications of op to shared memory and releases the locks. A process that requires a lock which is not free, releases the locks it holds, helps the process that owns the lock to finish the operation it executes, and then re-starts its execution. To enable this helping mechanism, a process shares its dictionary immediately prior to its locking phase. The lock-free TM algorithm presented in [17] works in a similar way.

As in Barnes' algorithm, a process executing an operation op in our algorithm, first locally simulates op using a local dictionary, and then it tries to apply the changes. However, in our algorithm, a conflict between two operations can be detected during the simulation phase, so helping may occur at an earlier stage of op's execution. More advanced helping techniques are required to ensure both wait-freedom and disjoint-access parallelism.

Chuong *et al.* [12] presented a wait-free version of Barnes' algorithm that is not disjoint-access parallel and applies operations to the data structure one at a time. Their algorithm is *transaction-friendly*, i.e., it allows operations to be aborted. Helping in this algorithm is simpler than in our algorithm. Moreover, the conflict detection and resolution mechanisms employed by our algorithm are more advanced to ensure disjoint-access parallelism. The presentation of the pseudocode of our algorithm follows [12].

The first software transactional memory algorithm [27] was disjoint-access parallel, but it is only non-blocking and is restricted to transactions that access a pre-determined set of memory locations. There are other TM algorithms [14, 17, 21, 25, 28] without this restriction that are disjoint-access parallel. However, all of them satisfy weaker progress properties than wait-freedom. TL [14] ensures strict disjoint access parallelism, but it is blocking.

A hybrid approach between transactional memory and universal constructions has been presented by Crain *et al.* [13]. Their universal construction takes, as input, sequential code that has been appropriately annotated for processing by a TM algorithm. Each transaction is repeatedly invoked until it commits. They use a linked list to store all committed transactions. A process helping a transaction to complete scans the list to determine whether the transaction has already completed. Thus, their implementation is not disjoint-access parallel. It also assumes that no failures occur.

3 Preliminaries

A *data structure* is a sequential implementation of an abstract data type. In particular, it provides a representation for the objects specified by the abstract data type and the (sequential) code for each of the operations it supports. As an example, we will consider an unsorted singly-linked list of integers that supports the operations APPEND(v), which appends the element v to the end of the list (by accessing a pointer *end* that points to the last element in the list, appending a node containing v to that element, and updating the pointer to point to the newly appended node), and SEARCH(v), which searches the list for v

starting from the first element of the list.

A *data item* is a piece of the representation of an object implemented by the data structure. In our example, the data items are the nodes of the singly-linked list and the pointers $first$ and *last* that point to the first and the last element of the list, respectively. The *state* of a data structure consists of the collection of data items in the representation and a set of values, one for each of the data items. A *static* data item is a data item that exists in the initial state. In our example, the pointers $first$ and *last* are static data items. When the data structure is dynamic, the data items accessed by an instance of an operation (in a sequential execution α) may depend on the instances of operations that have been performed before it in α. For example, the set of nodes accessed by an instance of SEARCH depends on the sequence of nodes that have been previously appended to the list.

An operation of a data structure is *value oblivious* if, in every (sequential) execution, the set of data items that each instance of this operation accesses in any sequence of consecutive instances of this operation does not depend on the values of the input parameters of these instances. In our example, APPEND is a value oblivious operation, but SEARCH is not.

We consider an *asynchronous shared-memory* system with n processes $p_1, \ldots, p_n$ that communicate by accessing shared objects, such as *registers* and `LL/SC` objects. A register R stores a value from some set and supports the operations `read`(R), which returns the value of R, and `write`(R, v), which writes the value v in R. An `LL/SC` *object* R stores a value from some set and supports the operations `LL`, which returns the current value of R, and `SC`. By executing `SC`(R, v), a process p_i attempts to set the value of R to v. This update occurs only if no process has changed the value of R (by executing `SC`) since p_i last executed `LL`(R). If the update occurs, `true` is returned and we say the `SC` is successful; otherwise, the value of R does not change and `false` is returned.

A *universal construction* provides a general mechanism to automatically execute pieces of sequential code in a concurrent environment. It supports a single operation, called PERFORM, which takes as parameters a piece of sequential code and a list of input arguments for this code. The algorithm that implements PERFORM applies a sequence of operations on shared objects provided by the system. We use the term *base objects* to refer to these objects and we call the operations on them *primitives*. A primitive is *non-trivial* if it may change the value of the base object; otherwise, the primitive is called *trivial*. To avoid ambiguities and to simplify the exposition, we require that all data items in the sequential code are only accessed via the instruction CREATEDI, READDI, and WRITEDI, which create a new data item, read (any part of) the data item, and write to (any part of) the data item, respectively.

A *configuration* provides a global view of the system at some point in time. In an *initial configuration*, each process is in its initial state and each base object has its initial value. A *step* consists of a primitive applied to a base object by a process and may also contain local computation by that process. An *execution* is a (finite or infinite) sequence $C_i, \phi_i, C_{i+1}, \phi_{i+1}, \ldots, \phi_{j-1}, C_j$ of alternating configurations (C_k) and steps (ϕ_k), where the application of ϕ_k to configuration C_k results in configuration C_{k+1}, for each $i \leq k < j$. An execution α is *indistinguishable* from another execution α' for some processes, if each of these processes takes the same steps in α and α', and each of these steps has the same response in α and α'. An execution is *solo* if all its steps are taken by the same process.

From this point on, for simplicity, we use the term operation to refer to an instance of an operation. The *execution interval* of an operation starts with the first step of the corresponding call to PERFORM and terminates when that call returns. Two operations *overlap* if the call to PERFORM for one of them occurs during the execution interval of the other. If a process has invoked PERFORM for an operation that has not yet returned, we say that the operation is *active*. A process can have at most one active operation in any configuration. A configuration is *quiescent* if no operation is active in the configuration.

Let α be any execution. We assume that processes may experience *crash failures*. If a process p does not fail in α, we say that p is *correct* in α. *Linearizability* [23] ensures that, for every completed operation in α and some of the uncompleted operations, there is some point within the execution interval of the operation called its *linearization point*, such that the response returned by the operation in α is the same as the response it would return if all these operations were executed serially in the order determined by their linearization points. When this holds, we say that the responses of the operations are *consistent*. An implementation is *linearizable* if all its executions are linearizable. An implementation is *wait-free* [20] if, in every execution, each correct process completes each operation it performs within a finite number of steps.

Since we consider linearizable universal constructions, every quiescent configuration of an execution of a universal construction applied to a sequential data structure defines a state. This is the state of the data structure resulting from applying each operation linearized prior to this configuration, in order, starting from the initial state of the data structure.

Two operations *contend* on a base object b if they both apply a primitive to b and at least one of these primitives is non-trivial. We are now ready to present the definition of disjoint-access parallelism that we use to prove our impossibility result. It is weaker than all the variants discussed in Section 2.

Definition 1. (**Feeble Disjoint-Access Parallelism**). *An implementation resulting from a universal construction applied to a (sequential) data structure is* feebly disjoint-access parallel *if, for every solo execution α_1 of an operation op_1 and every solo execution α_2 of an operation op_2, both starting from the same quiescent configuration C, if the sequential code of op_1 and op_2 access disjoint sets of data items when each is executed starting from the state of the data structure represented by configuration C, then α_1 and α_2 contend on no base objects. A universal construction is* feebly disjoint-access parallel *if all implementations resulting from it are feebly disjoint-access parallel.*

We continue with definitions that are needed to define the version of disjoint-access parallelism ensured by our algorithm. Fix any execution $\alpha = C_0, \phi_0, C_1, \phi_1, \ldots,$ produced by a linearizable universal construction U. Then there is some linearization of the completed operations in α and a subset of the uncompleted operations in α such that the responses of all these operations are consistent. Let op be any one of these operations, let I_{op} be its execution interval, let C_i denote the first configuration of I_{op}, and let C_j be the first configuration at which op has been linearized. Since each process has at most one uncompleted operation in α and each operation is linearized within its execution interval, the set of operations linearized before C_i is finite. For $i \leq k < j$, let S_k denote the state of the data structure which results from applying each operation linearized in α prior to configuration C_k, in order, starting from the initial state of the data structure. Define $DS(op, \alpha)$, the data set of op in α, to be the set of all data items accessed by op when executed by itself starting from S_k, for $i \leq k < j$.

The *conflict graph* of an execution interval I of α is an undirected graph, where vertices represent operations whose execution intervals overlap with I and an edge connects two operations op and op' if and only if $DS(op, \alpha) \cap DS(op', \alpha) \neq \emptyset$. The following variant of disjoint-access parallelism is ensured by our algorithm.

Definition 2. (**Disjoint-Access Parallelism**). *An implementation resulting from a universal construction applied to a (sequential) data structure is* disjoint-access parallel *if, for every execution containing a process executing* Perform*(op_1) and a process executing* Perform*(op_2) that contend on some base object, there is a path between op_1 and op_2 in the conflict graph of the minimal execution interval containing op_1 and op_2.*

The original definition of disjoint-access parallelism in [24] differs from Definition 2 in that it does not allow two operations op_1 and op_2 to read the same base object even if there is no path between op_1 and op_2 in the conflict graph of the minimal execution interval that contains them. T Also, that definition imposes a bound on the step complexity of disjoint-access parallel algorithms. Our definition is a slightly stronger version of the disjoint-access parallel variant defined in [9] in the context of transactional memory. This definition allows two operations to contend, (but not *concurrently* contend) on the same base object if there is no path connecting them in the conflict graph. This definition makes the lower bound proved there stronger, whereas our definition makes the design of an algorithm (which is our goal) more difficult. Our definition is obviously weaker than strict disjoint-access parallelism [18], since our definition allows two processes to contend even if the data sets of the operations they are executing are disjoint.

4 Impossibility Result

To prove the impossibility of a wait-free universal construction with feeble disjoint-access parallelism, we consider an implementation resulting from the application of an arbitrary feebly disjoint-access parallel universal construction to the singly-linked list discussed in Section 3. We show that there is an execution in which an instance of SEARCH does not terminate. The idea is that, as the process p performing this instance proceeds through the list, another process, q, is continually appending new elements with different values. If q performs each instance of APPEND before p gets too close to the end of the list, disjoint-access parallelism prevents q from helping p. This is because q's knowledge is consistent with the possibility that p's instance of SEARCH could terminate successfully before it accesses a data item accessed by q's current instance of APPEND. Also, process p cannot determine which nodes were appended by process q after it started the SEARCH. The proof relies on the following natural assumption about universal constructions. Roughly speaking, it formalizes that the operations of the concurrent implementation resulting from applying a universal construction to a sequential data structure should simulate the behavior of the operations of the sequential data structure.

Assumption 3 (Value-Obliviousness Assumption)**.** *If an operation of a data structure is value oblivious, then, in any implementation resulting from the application of a universal construction to this data structure, the sets of base objects read from and written to during any solo execution of a sequence of consecutive instances of this operation starting from a quiescent configuration do not depend on the values of the input parameters.*

We consider executions of the implementation of a singly-linked list L in which process p performs a single instance of SEARCH$(L, 0)$ and process q performs instances of APPEND(L, i), for $i \geq 1$, and possibly one instance of APPEND$(L, 0)$. The sequential of the singly-linked list code is given in Appendix A. We may assume the implementation is deterministic: If it is randomized, we fix a sequence of coin tosses for each process and only consider executions using these coin tosses.

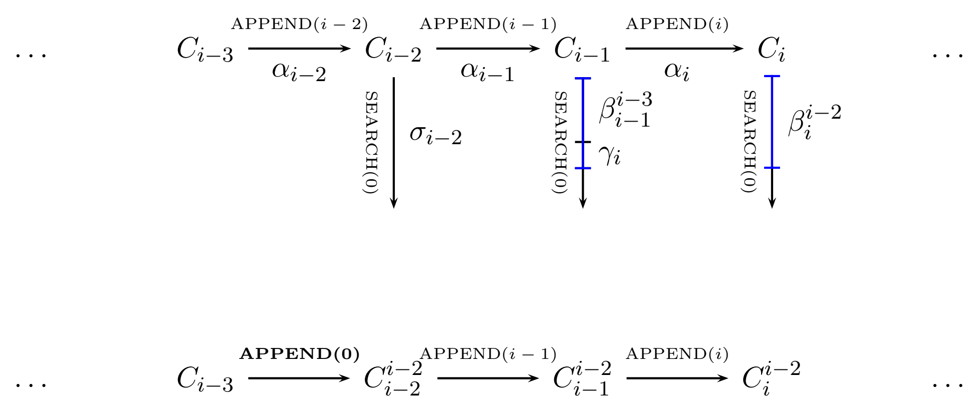

Figure 1: Configurations and Sequences of Steps used in the Proof

Let C_0 be the initial configuration in which L is empty. Let α denote the infinite solo execution by q starting from C_0 in which q performs APPEND(L, i) for all positive integers i, in increasing order. For $i \geq 1$, let C_i be the configuration obtained when process q performs APPEND(L, i) starting from configuration C_{i-1}. Let α_i denote the sequence of steps performed in this execution. Let $B(i)$ denote the set of base objects written to by the steps in α_i and let $A(i)$ denote the set of base objects these steps read from but do not write to. Notice that the sets $A(i)$ and $B(i)$ partition the set of base objects accessed in α_i. In configuration C_i, the list L consists of i nodes, with values $1, \ldots, i$ in increasing order.

For $1 < j \leq i$, let C_i^j be the configuration obtained from configuration C_0 when process q performs the first i operations of execution α, except that the j'th operation, APPEND(L, j), is replaced by APPEND$(L, 0)$;

namely, when q performs $\textsc{Append}(L,1)$, $\ldots$, $\textsc{Append}(L,j-1)$, $\textsc{Append}(L,0)$, $\textsc{Append}(L,j+1)$, $\ldots$, $\textsc{Append}(L,i)$. Since $\textsc{Append}$ is value oblivious, the same set of base objects are written to during the executions leading to configurations C_i and C_i^j. Only base objects in $\cup\{B(k) \mid j \leq k \leq i\}$ can have different values in C_i and C_i^j.

For $i \geq 3$, let σ_i be the steps of the solo execution of $\textsc{Search}(L,0)$ by p starting from configuration C_i. For $1 < j \leq i$, let β_i^j be the longest prefix of σ_i in which p does not access any base object in $\cup\{B(k) \mid k \geq j\}$ and does not write to any base object in $\cup\{A(k) \mid k \geq j\}$

Lemma 4. *For $i \geq 3$ and $1 < j \leq i$, $\beta_i^j = \beta_{i+1}^j$ and β_{i+1}^{i-1} is a prefix of β_{i+2}^i.*

Proof. Only base objects in $B(i+1)$ have different values in configurations C_i and C_{i+1}. Since β_i^j and β_{i+1}^j do not access any base objects in $B(i+1)$, it follows from their definitions that $\beta_i^j = \beta_{i+1}^j$. In particular, $\beta_{i+2}^i = \beta_{i+1}^i$, which, by definition contains β_{i+1}^{i-1} as a prefix. $\square$

For $i \geq 3$, let γ_{i+2} be the (possibly empty) suffix of β_{i+2}^i such that $\beta_{i+1}^{i-1}\gamma_{i+2} = \beta_{i+2}^i$. Figure 1 illustrates these definitions.

Let $\alpha' = \alpha_1\alpha_2\alpha_3\alpha_4\beta_4^2\alpha_5\gamma_5\alpha_6\gamma_6\cdots$. We show that this infinite sequence of steps gives rise to an infinite valid execution starting from C_0 in which there is an instance of $\textsc{Search}(L,0)$ that never terminates. The first steps of this execution are illustrated in Figure 2.

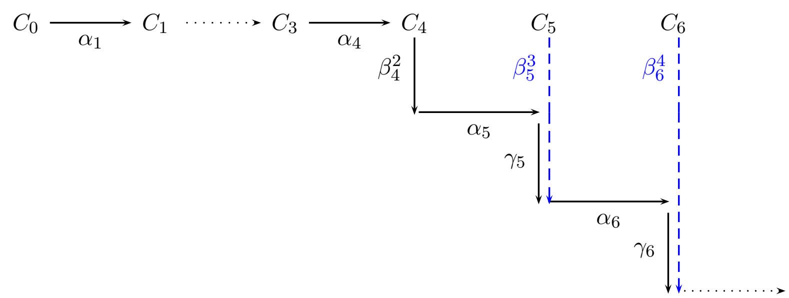

Figure 2: An Infinite Execution with a Non-terminating $\textsc{Search}$ Operation

Since β_4^2 does not write to any base objects accessed in $\alpha_2\alpha_3\cdots$ and, for $i \geq 4$, $\beta_{i+1}^{i-1} = \beta_i^{i-2}\gamma_{i+1}$ does not write to any base object accessed in $\alpha_{i-1}\alpha_i\cdots$, the executions arising from α and α' starting from C_0 are indistinguishable to process q. Furthermore, since β_{i+1}^{i-1} and, hence, γ_{i+1} does not access any base object written to by $\alpha_{i-1}\alpha_i\cdots$, it follows that $\alpha_1\alpha_2\alpha_3\alpha_4\beta_4^2\alpha_5\gamma_5\cdots\alpha_j\gamma_j$ and $\alpha_1\alpha_2\alpha_3\alpha_4\cdots\alpha_j\beta_j^{j-2}$ are indistinguishable to process p for all $j \geq 4$. Thus α' is a valid execution.

Next, for each $i \geq 4$, we prove that there exists $j > i$ such that γ_j is nonempty. By the value obliviousness assumption, only base objects in $B(i-2) \cup B(i-1) \cup B(i)$ can have different values in C_i and C_i^{i-2}. Since β_i^{i-2} does not access any of these base objects, β_i^{i-2} is also a prefix of $\textsc{Search}(L,0)$ starting from C_i^{i-2}. Since $\textsc{Search}(L,0)$ starting from C_i^{i-2} is successful, but starting from C_i is unsuccessful, $\textsc{Search}(L,0)$ is not completed after β_i^{i-2}. Therefore β_i^{i-2} is a proper prefix of σ_i. Let b be the base object accessed in the first step following β_i^{i-2} in σ_i. For $j \geq i+1$, only base objects in $\cup\{B(k) \mid i+1 \leq k \leq j\}$ can have different values in C_i and C_j. Therefore the first step following β_i^{i-2} in σ_j is the same as the first step following β_i^{i-2} in σ_i.

To obtain a contradiction, suppose that $\beta_i^{i-2} = \beta_{i+3}^{i+1}$. Then b is the base object accessed in the first step following β_{i+3}^{i+1} in σ_{i+3}. By definition of β_{i+3}^{i+1}, there is some $\ell \geq i+1$ such that the first step following β_{i+3}^{i+1} in σ_{i+3} is either an access to $b \in B(\ell)$ or a write to $b \in A(\ell)$.

Let S denote the state of the data structure in configuration $C_{\ell-1}^{\ell-3}$. In state S, the list has $\ell - 1$ nodes and the third last node has value 0. Thus, the set of data items accessed by $\textsc{Search}(L,0)$ starting from state S consists of $L.first$ and the first $\ell - 3$ nodes of the list. This is disjoint from the set of data items accessed by $\textsc{Append}(L, \ell)$ starting from state S, which consists of $L.last$, the last node of the list, and the newly appended node. Hence, by feeble disjoint access parallelism, the solo executions of $\textsc{Append}(L, \ell)$ and $\textsc{Search}(L, 0)$ starting from $C_{\ell-1}^{\ell-3}$ contend on no base objects.

By the value obliviousness assumption, $B(\ell)$ is the set of base objects written to in the solo execution of $\textsc{Append}(L, \ell)$ starting from $C_{\ell-1}^{\ell-3}$ and $A(\ell)$ is the set of base objects read from, but not written to in that execution.

By the value obliviousness assumption, only base objects in $B(\ell-3) \cup B(\ell-2) \cup B(\ell-1)$ can have different values in $C_{\ell-1}$ and $C_{\ell-1}^{\ell-3}$. Since β_i^{i-2} does not access any of these base objects, β_i^{i-2} is also a prefix of $\textsc{Search}(L, 0)$ starting from $C_{\ell-1}^{\ell-3}$ and the first step following β_i^{i-2} in this execution is the same as the first step following β_i^{i-2} in σ_i. Recall that this is either an access to $b \in B(\ell)$ or a write to $b \in A(\ell)$. Thus, the solo executions of $\textsc{Append}(L, \ell)$ and $\textsc{Search}(L, 0)$ starting from $C_{\ell-1}^{\ell-3}$ contend on b. This is a contradiction. Hence, $\beta_i^{i-2} \neq \beta_{i+3}^{i+1}$ and it follows that at least one of γ_{i+1}, γ_{i+2}, and γ_{i+3} is nonempty.

Therefore γ_j is nonempty for infinitely many numbers j and, in the infinite execution α', process p never completes its operation $\textsc{Search}(L, 0)$, despite taking an infinite number of steps. Hence, the implementation is not wait-free and we have proved the following result:

Theorem 5. *No feebly disjoint-access parallel universal construction is wait-free.*

5 The DAP-UC Algorithm

To execute an operation *op*, a process p locally simulates the execution of *op*'s instructions without modifying the shared representation of the simulated state. This part of the execution is the simulation phase of *op*. Specifically, each time p accesses a data item while simulating *op*, it stores a copy in a local dictionary. All subsequent accesses by p to this data item (during the same simulation phase of *op*) are performed on this local copy. Once all instructions of *op* have been locally simulated, *op* enters its modifying phase. At that time, one of the local dictionaries of the helpers of *op* becomes shared. All helpers of *op* then use this dictionary and apply the modifications listed in it. In this way, all helpers of *op* apply the same updates for *op*, and consistency is guaranteed.

```
type varrec
        value val
        ptr to oprec A[1..n]
type statrec
        {⟨simulating⟩,
         ⟨restart, ptr to oprec restartedby⟩,
         ⟨modifying, ptr to dictionary of dictrec changes,
                                               value output⟩
         ⟨done⟩
        } status
type oprec
        code  program
        process id  owner
        value input
        value output
        ptr to statrec status
        ptr to oprec tohelp[1..n]
type dictrec
        ptr to varrec key
        value newval
```

Figure 3: Type definitions

The algorithm maintains a record for each data item x. The first time *op* accesses x, it makes an announcement by writing appropriate information in x's record. It also detects conflicts with other operations that are accessing x by reading this record. So, conflicts are detected without violating disjoint access parallelism. The algorithm uses a simple priority scheme, based on the process identifiers of the owners of the operations, to resolve conflicts among processes. When an operation *op* determines a conflict with an operation op' of higher priority, *op* helps op' to complete before it continues its execution. Otherwise, *op* causes op' to restart and the owner of *op* will help op' to complete once it finishes with the execution of *op*, before it starts the execution of a new operation. The algorithm also ensures that before op' restarts its simulation phase, it will help *op* to complete. These actions guarantee that processes never starve.

We continue with the details of the algorithm. The algorithm maintains a record of type `oprec` (lines 11-17) that stores information for each initiated operation. When a process p wants to execute an operation *op*, it starts by creating a new `oprec` for *op* and initializing it appropriately (line 22). In particular, this record provides a pointer to the code of *op*, its input parameters, its output, the status of *op*, and an array indicating whether *op* should help other operations after its completion and before it returns. We call p the *owner* of *op*. To execute *op*, p calls Help (line 23). To ensure wait-freedom, before *op* returns, it helps all other operations listed in the *tohelp* array of its *oprec* record (lines 24-25). These are operations with which *op* had a conflict during the course of its execution, so disjoint-access parallelism is not violated. The algorithm also maintains a record of type `varrec` (lines 1-3) for each data item x, This record contains a *val* field, which is an `LL/SC` object that stores the value of x, and an array A of n `LL/SC` objects, indexed by process identifiers, which stores `oprec` records of operations that are accessing x. This array is used by operations to announce that they access x and to determine conflicts with other operations that are also accessing x.

The execution of *op* is done in a sequence of one or more *simulation phases* (lines 34-53) followed by a *modification phase* (lines 54-62). In a simulation phase, the instructions of *op* are read (lines 36, 37, and 50) and the execution of each one of them is simulated locally. The first time each process q helping *op* (including its owner) needs to access a data item (lines 38, 43), it creates a local copy of it in its (local) dictionary (lines 42, 46). All subsequent accesses by q to this data item (during the current simulation phase of *op*) are performed on this local copy (line 48). During the modification phase, q makes the updates of *op* visible by applying them to the shared memory (lines 56-62).

The *status* field of *op* determines the execution phase of *op*. It contains a pointer to a record of type `statrec` (lines 4-10) where the status of *op* is recorded. The status of *op* can be either *simulating*, indicating that *op* is in its simulation phase, $modifying$, if *op* is in its modifying phase, *done*, if the execution of *op* has been completed but *op* has not yet returned, or *restart*, if *op* has experienced a conflict and should re-execute its simulation phase from the beginning. Depending on which of these values *status* contains, it may additionally store another pointer or a value.

To ensure consistency, each time a data item x is accessed for the first time, q checks, before reading the value of x, whether *op* conflicts with other operations accessing x. This is done as follows: q announces *op* to x by storing a pointer *opr* to *op*'s `oprec` in $A[opr \rightarrow owner]$. This is performed by calling Announce (line 39). Announce first performs an `LL` on $var_x \rightarrow A[p]$ (line 68), where var_x is the `varrec` for x and $p = opr \rightarrow owner$. Then, it checks if the status of *op* (line 69) remains *simulating* and, if this is so, it performs an `SC` to store *op* in $var_x \rightarrow A[p]$ (line 70). These instructions are then executed one more time. This is needed because an obsolete helper of an operation, initiated by p before *op*, may successfully execute an `SC` on $var_x \rightarrow A[p]$ that stores a pointer to this operation's `oprec`. However, we prove in Section 6 that this can happen only once, so executing the instructions on lines 68-70 twice is enough to ensure consistency.

After announcing *op* to var_x, q calls Conflicts (line 40) to detect conflicts with other operations that access x. In Conflicts, q reads the rest of the elements of $var_x \rightarrow A$ (lines 76-77). Whenever a conflict is detected (i.e., the condition of the `if` statement of line 78 evaluates to `true`) between *op* and some other operation op', Conflicts first checks if op' is in its modifying phase (line 82) and, if so, it helps op' to complete. In this way, it is ensured that, once an operation enters its modification phase, it will complete its operation successfully. Therefore, once the status of an operation becomes $modifying$, it will next become *done*, and then, henceforth, never change. If the status of op' is *simulating*, q determines which of *op* or op' has the higher priority (line 84). If op' has higher priority (line 89), then *op* helps op' by calling Help(op'). Otherwise, q first adds a pointer opr' to the `oprec` of op' into $opr \rightarrow tohelp$ (line 85), so that the owner of *op* will help op' to complete after *op* has completed. Then q attempts to notify op' to restart, using `SC` (line 87) to change the status of op' to *restart*. A pointer *opr* is also stored in the status field of op'. When op'

```
value Perform(prog, input) by process p:
    opptr := pointer to a new oprec record
    opptr → program := prog, opptr → input := input, opptr → output := ⊥
    opptr → owner := p, opptr → status := simulating, opptr → tophelp[1..n] := [nil, ..., nil]
    Help(opptr)                                                      /* p helps its own operation */
    for p' := 1 to n excluding p do          /* p helps operations that have been restarted by its operation op */
        if (opptr → tohelp[p'] ≠ nil) then Help(opptr → tohelp[p'])
    return(opptr → output)

Help(opptr) by process p:
    opstatus := LL(opptr → status)
    while (opstatus ≠ done)
        if opstatus = ⟨restart, opptr'⟩ then                          /* op' has restarted op */
            Help(opptr')                                              /* first help op' */
            SC(opptr → status, ⟨simulating⟩)               /* try to change the status of op back to simulating */
            opstatus := LL(opptr → status)
        if opstatus = ⟨simulating⟩ then                               /* start a new simulation phase */
            dict := pointer to a new empty dictionary of dictrec records
                                                                      /* to store the values of the data items */
            ins := the first instruction in opptr → program
            while ins ≠ return(v)                                     /* simulate instruction ins of op */
                if ins is (WriteDI(x, v) or ReadDI(x)) and (there is no dictrec with key x in dict)
                then                                                  /* first access of x by this attempt of op */
                    Announce(opptr, x)                                /* announce that op is accessing x */
                    Conflicts(opptr, x)                  /* possibly, help or restart other operations accessing x */
                    if ins = ReadDI(x) then val_x := x → val  else val_x := v     /* ins is a write to x of v */
                    add new dictrec ⟨x, val_x⟩ to dict                /* create a local copy of x */
                else if ins is CreateDI() then
                    x := pointer to a new varrec record
                    x → A[1..n] := [nil, ..., nil]
                    add new dictrec ⟨x, nil⟩ to dict
                else                       /* ins is WriteDI(x, v) or ReadDI(x) and there is a dictrec with key x in dict */
                                           /* or ins is not a WriteDI(), ReadDI() or CreateDI() instruction */
                    execute ins, using/changing the value in the appropriate entry of dict if necessary
                if ¬VL(opptr → status) then break                     /* end of the simulation of ins */
                ins := next instruction of opptr → program
            /* end while */
            if ins is return(v) then                                  /* v may be empty */
                SC(opptr → status, ⟨modifying, dict, v⟩)              /* try to change status of op to modifying */
                          /* successful iff simulation is over and status of op not changed since beginning of simulation */
            opstatus := LL(opptr → status)
        if opstatus = ⟨modifying, changes, out⟩ then
            opptr → outputs := out
            for each dictrec ⟨x, v⟩ in the dictionary pointed to by changes do
                LL(x → val)                                           /* try to make writes visible */
                if ¬VL(opptr → status) then return                    /* opptr → status = done */
                SC(x → val, v)
                LL(x → val)
                if ¬VL(opptr → status) then return                    /* opptr → status = done */
                SC(x → val, v)
            /* end for */
            SC(opptr → status, done)
            opstatus := LL(opptr → status)
    /* end while */
return

Announce(opptr, x) by process p:
    q := opptr → owner
    LL(x → A[q])
    if ¬ VL(opptr → status) then return
    SC(x → A[q], opptr)
    LL(x → A[q])
    if ¬VL(opptr → status) then return
    SC(x → A[q], opptr)
    return

Conflicts(opptr, x) by process p:
    for p' := 1 to n excluding opptr → owner do
        opptr' := LL(x → A[p'])
        if (opptr' ≠ nil) then                       /* possible conflict between op and op' */
            opstatus' := LL(oppptr' → status)
            if ¬VL(opptr → status) then return
            if (opstatus' = ⟨modifying, changes, output⟩)
                then Help(opptr')
            else if (opstatus' = ⟨simulating⟩) then
                if (opptr → owner < p') then
                                                     /* op has higher priority than op', restart op' */
                    opptr → tohelp[p'] := opptr'
                    if ¬VL(opptr → status) then return
                    SC(opptr' → status, ⟨restart, opptr⟩)
                    if (LL(oppptr' → status) = ⟨modifying, changes, output⟩) then
                        Help(opptr')
            else Help(opptr')                        /* opptr → owner > p' */
    return
```

Figure 4: The code of Perform, Help, Announce, and Conflicts.

restarts its simulation phase, it will help *op* to complete (lines 30-33), if *op* is still in its simulation phase, before it continues with the re-execution of the simulation phase of op'. This guarantees that *op* will not cause op' to restart again.

Recall that each helper q of *op* maintains a local dictionary. This dictionary contains an element of type `dictrec` (lines 18-20) for each data item that q accesses (while simulating *op*). A dictionary element corresponding to data item x consists of two fields, key, which is a pointer to var_x, and $newval$, which stores the value that *op* currently knows for x. Notice that only one helper of *op* will succeed in executing the `SC` on line 52, which changes the status of *op* to $modifying$. This helper records a pointer to the dictionary it maintains for *op*, as well as its output value, in *op*'s *status*, to make them public. During the modification phase, each helper q of *op* traverses this dictionary, which is recorded in the status of *op* (lines 54, 56). For each element in the dictionary, it tries to write the new value into the `varrec` of the corresponding data item (lines 57-59). This is performed twice to avoid problems with obsolete helpers in a similar way as in Announce.

Theorem 6. *The* DAP-UC *universal construction (Figures 3 and 4) produces disjoint-access parallel, wait-free, concurrent implementations when applied to objects that have a bound on the number of data items accessed by each operation they support.*

6 Proof of the DAP-UC Algorithm

6.1 Preliminaries

The proof is divided in three parts, namely consistency (Section 6.2), wait-freedom (Section 6.3) and disjoint-access parallelism (Section 6.4). The proof considers an execution α of the universal construction applied to some sequential data structure. The configurations referred to in the proof are implicitly defined in the context of this execution. We first introduce a few definitions and establish some basic properties that follow from inspection of the code.

Observe that an `oprec` is created only when a process begins Perform (on line 22). Thus, we will not distinguish between an operation and its `oprec`.

Observation 7. *The status of each* `oprec` *is initially simulating (line 22). It can only change from simulating to* modifying *(lines 34,52), from* modifying *to done (lines 54,63), from simulating to restart (lines 83,87), and from restart to simulating (lines 30,32).*

Thus, once the status of an `oprec` becomes *modifying*, it can only change to *done*.

Observation 8. *Let op be any operation and let opptr be the pointer to its* `oprec`*. When a process returns from* Help(*opptr*) *(on line 58, 61 or 65), opptr* $\rightarrow$ *status = done.*

This follows from the exit condition of the **while** loop (line 29) and the fact that, once the status of an `oprec` becomes *modifying*, it can only change to *done*.

Observation 9. *In every configuration, there is at most one* `oprec` *owned by each process whose status is not done.*

This follows from the fact that, when a process returns from Perform (on line 26), has also returned from a call to Help (on line 23), so the status of the `oprec` it created (on line 22) has status *done*, and the fact that a process does not call Perform recursively, either directly or indirectly.

Observation 10. *For every* `varrec`*, $A[i]$, $1 \leq i \leq n$, is initially nil and is only changed to point to* `oprecs` *with owner i.*

This follows from the fact that $A[i]$, $1 \leq i \leq n$, is initialized to *nil* when the `varrec` is created (on line 44) and is updated only on lines 70 or 73.

6.2 Consistency

An *attempt* is an endeavour by a process to simulate an operation. Formally, let *op* be any operation initiated by process q in α and let *opptr* be the pointer to its `oprec`, i.e., $opptr \rightarrow owner = q$.

Definition 11. *An* attempt *of op by a process p is the longest execution interval that begins when p performs a* `LL` *on* $opptr \rightarrow status$ *on line 28, 33, or 53 that returns simulating and during which* $opptr \rightarrow status$ *does not change.*

The first step after the beginning of an attempt is to create an empty dictionary of `dictrec`s (line 35). So, each dictionary is uniquely associated with an attempt. We say that an attempt is *active* at each configuration C contained in the execution interval that defines the attempt.

Let p be a process executing an attempt *att* of *op*. If immediately after the completion of *att*, p successfully changes $opptr \rightarrow status$ to $\langle modifying, chgs, val\rangle$ (by performing a `SC` on $opptr \rightarrow status$ on line 52), then *att* is *successful*. Notice that, in this case, *chgs* is a pointer to the dictionary associated with *att*.

By Observation 7, only one process executing an attempt of *op* can succeed in executing the `SC` that changes the status of *op* to $\langle modifying, \_, \_\rangle$ (on line 52). Next observation then follows from the definition of a successful attempt:

Observation 12. *For each operation, there is at most one successful attempt.*

In *att*, p *simulates* instructions on behalf of *op* (lines 34 - 52). The simulation of an instruction *ins* *starts* when *ins* is fetched from *op*'s *program* (on lines 36 or 50) and *ends* either just before the next instruction starts simulated, or just after the execution of the `SC` on line 52 if *ins* is the last instruction of $opptr \rightarrow program$.

When p simulates a CreateDI() instruction, it allocates a new `varrec` record x in its own stripe of shared memory (line 44) and adds a pointer to it in the dictionary associated with *att* (line 46); in this case, we also say that p *simulates the creation* of, or *creates* x. Notice that x is initially *private*, as it is known only by p; it may later become *public* if *att* is successful. Next definition captures precisely the notion of public `varrec`.

We say that a `varrec` x is *referenced* by operation *op* in some configuration C, if $opptr \rightarrow status = \langle modifying, chgs, \_\rangle$, where *chgs* is a pointer to a dictionary that contains a `dictrec` record whose first component, *key*, is a pointer to x.

Definition 13. *A* `varrec` x *is* public *in configuration* C *if and only if it is static or there exists an operation that references* x *in* C *or in some configuration that precedes it.*

We say that p *simulates an access* of (or *access*) some `varrec` x by (for) *op*, if it either simulates an $ins \in \{\text{ReadDI}(x), \text{WriteDI}(x, \_)\}$, or creates x. Observe that if x is public in configuration C, it is also public in every configuration that follows. Also, before it is made public, x cannot be accessed by a process that has not created it.

Observation 14. *If, in att, p starts the simulation of an instruction* $ins \in \{\text{WriteDI}(x, \_), \text{ReadDI}(x)\}$ *at some configuration* C*, then either* x *is created by p in att before* C*, or there exists a configuration preceding the simulation of ins in which* x *is public.*

Notice that each time p accesses for the first time a `varrec` x during *att*, a new `dictrec` record is added for x to the dictionary associated with *att* (on lines 42 or 46). From this and by inspecting the code lines 38, 42, 43 and 46 follows the observation bellow.

Observation 15. *If a* `varrec` x *is accessed by p during att for op, then the first time that it accesses* x*, the following hold:*

1. *p executes either lines 38 to 42 or lines 43 to 46 exactly once for* x*,*
2. *p inserts a* `dictrec` *record for* x *in the dictionary associated with att exactly once, i.e., this record is unique.*

We say that p *announces* op on a `varrec` x during *att*, if it successfully executes an `SC` of line 70 or line 73 on $x.A[q]$ (recall that $opptr \rightarrow owner = q$) with value *opptr*, during a call of $\text{Announce}(opptr, x)$ (on line 39). Distinct processes may perform attempts of the same operation *op*. However, once an operation has been announced to a `varrec`, it can only be replaced by a more recent operation owned by the same process (i.e., one initiated by q after *op*'s response), as shown by the next lemma.

Lemma 16. *Assume that p calls* $\text{Announce}(opptr, x)$ *in att. Suppose that in the configuration C_A immediately after p returns from that call, att is active. Then, in configuration C_A and every configuration that follows in which $oppptr \rightarrow status \neq done$, $(x.A[opptr \rightarrow owner]) = opptr$.*

Proof. Since *att* is active when p returns from $\text{Announce}(opptr, x)$, the tests performed on lines 69 and 72 are successful. So, p performed $\texttt{LL}(x \rightarrow A[q], opptr)$ on lines 68 and 71 respectively. Let C_{LL1} and C_{LL2} be the configurations immediately after p performed line 68 and 71, respectively.

Let C be a configuration after p has returned from the call of $\text{Announce}(opptr, x)$ in which $opptr \rightarrow status \neq done$. Assume, by contradiction, that $(x.A[q]) = opptr'$ in C, where $opptr'$ is a pointer to an operation $op' \neq op$. Let p' be the last process that changes the value of $x.A[q]$ to $opptr'$ before C. Therefore p' performed a successful $\texttt{SC}(x.A[q], opptr')$ on line 70 or line 73. This `SC` is preceded by a $\texttt{VL}(opptr' \rightarrow status)$ (on line 69 or line 72), which is itself preceded by a $\texttt{LL}(x \rightarrow A[q])$ (on line 68 or line 71). Denote by C'_{SC}, C'_{VL} and C'_{LL}, respectively, the configurations that immediately follow each of these steps. Since the `VL` applied by p' on $(opptr' \rightarrow status)$ is successful, $opptr' \rightarrow status = simulating$ in configuration C'_{VL}.

By Observation 10, $opptr' \rightarrow owner = q$. By Observation 9, in every configuration, there is only one operation owned by q whose status is not *done*. Since *op* has status *simulating* when p started its attempt and the status of *op* is not equal to *done* in C, it then follows from Observation 7 that the status of op' is *done* when the attempt *att* of *op* by p started. Therefore, configuration C'_{VL}, in which the status of op' is simulating, must precede the first configuration in which *att* is active. In particular, C'_{VL} precedes C_{LL1} and thus C'_{LL} precedes C_{LL1}.

We consider two cases according to the order in which C_{LL2} and C'_{SC} occur:

- C'_{SC} occurs before C_{LL2}. In that case, no process performs a successful $\texttt{SC}(x \rightarrow A[q], opptr'')$, where $opptr''$ is a pointer to an operation $op'' \neq op$, after C'_{SC} and before C; this follows from the definition of p'. Notice that the second $\texttt{SC}(x \rightarrow A[q], opptr)$ performed by p on line 73 is executed after C'_{SC}, so it cannot be successful. However, this `SC` is unsuccessful only if a process $\neq p$ performs a successful `SC` on $x \rightarrow A[q]$ after C_{LL2} and before it, thus between C'_{SC} and C, which is a contradiction.

- $C_{SC'}$ occurs after C_{LL2}. Notice that C'_{LL} precedes C_{LL1} and p performs a `SC`(x.A[q],opptr) (on line 70) between C_{LL1} and C_{LL2}. If this `SC` is successful, then the `SC`($x.A[q]$]),opptr') performed by p' immediately before $C_{SC'}$ cannot be successful, which is a contradiction. Otherwise, another process performs a successful `SC` on $x.A[q]$ after C_{LL1} and before p performs the $\texttt{SC}(x.A[q], opptr)$ on line 70, which also prevents the `SC` performed by p' from being successful, which is a contradiction. ☐

Attempts of distinct operations may access the same `varrec`s. When an attempt *att* of *op* accesses a `varrec` x for the first time by simulating $\text{ReadDI}(x)$ or $\text{WriteDI}(x, \_)$, the operation is first announced to x (on line 39) and then $\text{Conflicts}(opptr, x)$ is called (on line 40, *opptr* is a pointer to *op*) to check whether another attempt att' of a distinct operation op' is concurrently accessing x. If this is the case (line 78), op' is either restarted (on line 87) or helped (on lines 82, 88 or 89). Since when $\text{Help}(op')$ returns, the status of op' is done (Observation 8), in both cases attempt att' is no longer active when the call to $\text{Conflicts}(opptr, x)$ returns. This is precisely what next Lemma establishes.

Lemma 17. *Let att, att' be two attempts by two processes denoted p and p', respectively, of two operations op, op' owned by q, q', where $q \neq q'$, respectively. Let x be a* `varrec`. *Denote by opptr and $opptr'$ two pointers to op and op' respectively. Suppose that:*

- *in att, p calls* Announce*$(opptr, x)$ and returns from that call,*

- *in* att'*,* p' *calls* CONFLICTS*(*$opptr', x$*) (on line 40) and returns from that call; denote by* C'_D *the configuration that follows the termination of* CONFLICTS*(*$opptr', x$*) by* p'*.*

- p' *returns from* ANNOUNCE*(*$opptr', x$*) after* p *returns from* ANNOUNCE*(*$opptr, x$*).*

Then, if att' *is active in* C'_D*, the following hold:*

1. *att is not active in* C'_D*;*

2. *if att is successful,* $opptr \rightarrow status = done$ *in* C'_D*.*

Proof. Let C_A denote the configuration immediately after p returns from ANNOUNCE($opptr, x$). Similarly, denote by C'_A the configuration immediately after p' returns from ANNOUNCE($opptr', x$). We have that C'_A occurs after C_A, and C'_A occurs before p' calls CONFLICTS($opptr', x$).

The proof is by contradiction. Let us assume that att' is active in C'_D and either *att* is active in C'_D or *att* is successful and $opptr \rightarrow status \neq done$ in C'_D. Consider the execution by p' of the call CONFLICTS($opptr', x$), which ends at configuration C'_D. In particular, as $q' = op' \rightarrow owner \neq op \rightarrow owner = q$, process p' checks whether an operation owned by q has been announced to the `varrec` pointed to by x (on line 76). We derive a contradiction by examining the steps taken by process p' in the iteration of the **for** loop in which $x \rightarrow A[q]$ is examined.

Let C be a configuration that follows C_A and precedes C'_D or is equal to C'_D. We show that $x \rightarrow A[q] = opptr$ in C. On one hand, *att* is active in configuration C_A and thus $opptr \rightarrow status = simulating$ in this configuration. On the other hand, either *att* is still active in C'_D, or *att* is successful, but $opptr \rightarrow status \neq done$ in C'_D. Therefore, by Observation 7, the status of *op* does not change between C_A and C'_D or is changed to $\langle modifying, \_, \_\rangle$. Hence, $opptr \rightarrow status \in \{simulating, \langle modifying, \_, \_\rangle\}$ in C.

In particular the configuration C'_{RA} that immediately precedes the read of $x.A[q]$ by p' (`LL` on line 77) occurs after C_A and before C'_D. C'_{RA} thus occurs after the call of ANNOUNCE($opptr, x$) by p returns, and the status of *op* is not done in this configuration. Therefore, by applying Lemma 16, we have that $A[q] = opptr$ in C'_{RA}.

As attempt att' is active in C'_D, it is active when p' performs CONFLICTS($opptr', x$). In particular, each `VL` on $opptr' \rightarrow status$ performed by p' (on line 80 or 86) in the execution of CONFLICTS($opptr', x$) returns `true`. Therefore, p' reads the status of the operation pointed to by *oppptr* (`LL`($opptr \rightarrow status$) on line 79). In the configuration to which this `LL` is applied, which occurs between C_A and C'_D, the status of *op* is either *simulating* or $\langle modifying, \_, \_\rangle$ for what above stated.

We consider two cases, according to the value read from $opptr \rightarrow status$ by p':

- The read of $opptr \rightarrow status$ by p' returns $\langle modifying, \_, \_\rangle$. In that case, p' calls HELP($opptr$) (line 82). In the configuration C in which p' returns from this call, $opptr \rightarrow status = done$ (Observation 8). As C is C'_D or occurs prior to C'_D, but after C_A, and the status of *op* is never changed to *done* between C_A and C'_D, this is a contradiction.

- The read of $opptr \rightarrow status$ by p' returns *simulating* (line 83). We distinguish two sub-cases according to the relative priorities of *op* and op':

  - $q' < q$, i.e., op' has higher priority than *op*. In this case, p' tries to change the status of *op* to $\langle restart, \_\rangle$ by performing a `SC` on $opptr \rightarrow status$ with parameter $\langle restart, opptr'\rangle$ (line 87). The `SC` is performed in a configuration that follows C_A and that precedes C'_D. The `SC` cannot succeed. Otherwise there is a configuration between C_A and C'_D where $opptr \rightarrow status$ is $\langle restart, opptr'\rangle$. This contradicts the fact that the status of *op* is *simulating* or $\langle modifying, \_, \_\rangle$ in every configuration between C_A and C'_D. Therefore, $opptr \rightarrow status$ has been changed to $\langle modifying, \_, \_\rangle$ before the `SC` is performed by p'. Thus, p' calls HELP($opptr$) (on line 88) after performing the unsuccessful `SC`. When this call returns, $opptr \rightarrow status = done$ (Observation 8) which is a contradiction.

– $q < q'$. In that case, p' calls HELP(*opptr*). As in the previous case, a contradiction can be obtained, since when p' returns from this call, $opptr \rightarrow status = done$ (Observation 8), and p' returns from the call to HELP(*opptr*) before C'_D. □

In an attempt of *op*, a new `varrec` is created each time a CREATEDI() instruction is simulated on line 44. For such a `varrec` to be later accessed in another attempt, a pointer to it must be either written to the *val* field of another `varrec`, or passed as an input parameter to an operation. Moreover, when the `varrec` is accessed, the status of the operation *op* is done.

Lemma 18. *Suppose that in att, p creates a* `varrec` *x. If an instruction* READDI*(x) or* WRITEDI*(x, _) is simulated in an attempt att′ of an operation op′ ≠ op, then op → status = done in the configuration preceding the beginning of the simulation of this instruction.*

Proof. Recall that x is allocated to a new shared memory slot (on line 44) and then a `dictrec` with key a pointer to x is added to the dictionary associated with *att* (on line 46). While *att* is active, the dictionary associated with it is private. Hence, in order for a WRITEDI() or READDI() with parameter x to be simulated in *att′*, the dictionary associated with *att* has to be made public, which can occur only if *att* is successful. Moreover, there is a `varrec` x' created by *att* such that x' is written to a `varrec` that is not created by *att*, or it is returned by *op*. This is so, since otherwise, no `varrecs` created in *att* can be accessed in any attemp other than *att*, which contradicts the fact that x is accessed by *att′*. In the second case, the code (lines 23 and 26) and Observation 8 imply that $opptr \rightarrow status = done$ before a pointer to x is passed as a parameter to *op′*, that is before *att′* simulates an access on x; so, the claim holds. We continue with the first case. Denote by W the set of `varrecs` that are written by *att* but have not been created by it.

In *att′*, an instruction WRITEDI(x, _) or READDI(x) is simulated. Since x is a dynamic `varrec`, this instruction is preceded by a simulation of a READDI() instruction on some data item not created by *att′* that returns a pointer to x. Assume that the first such instruction R has parameter y. We argue that R is the first access of y by *att′*. This is so since a copy of y is inserted into the dictionary of *att′* the first time it is accessed by *att′* and any subsequent access of y by *att′* returns the value written in the dictionary.

1. $y \in W$. Note that y is neither created in *att* nor in *att′* but accessed in both attempts. Therefore, Observation 15 implies that the first time it is accessed in *att*, ANNOUNCE(*opptr*, y) and CONFLICTS(*opptr*, y) are called (lines 39–40). Both calls terminate, as *att* is successful. Denote by C_A and C_D the configurations that follow the termination of ANNOUNCE(*opptr*, y) and CONFLICTS(*opptr*, y), respectively. Notice that *att* is active in C_D. This is due to the fact that *att* remains active until the `SC` on line 52 that changes the status of *op* to $\langle modifying, \_, \_\rangle$ is applied.

   Similarly, Observation 15 implies that ANNOUNCE(*opptr′*, y) and CONFLICTS(*opptr′*, y) are called when *att′* simulates READDI(y). Both calls terminate, since the simulation of READDI(y) by *att′* returns a value. Denote by C'_A and C'_D the configurations that follow the termination of ANNOUNCE(*opptr′*, y) and CONFLICTS(*oppptr′*, y), respectively. Note that *att′* is active in C'_D since another instruction, namely, READDI(x) or WRITEDI(x, _), is simulated later, and the status of *op′* is validated before a new instruction is simulated (line 49).

   If C_A occurs before C'_A, it follows from Lemma 17 that $opptr \rightarrow status = done$ in C'_D. Therefore, by Observation 7, the status of *op* is *done* when the simulation of READDI(x) or WRITEDI(x, _) starts in *att′*. Otherwise, C'_A occurs before C_A. In that case, it follows from Lemma 17 that *att′* is not active in C_D. Since the `SC` on line 52 by *att* is executed after C_D and x becomes visible to other attempts only after this `SC`, it is not possible for *att′* to access x, which is a contradiction.

2. $y \notin W$. In this case, a pointer ptr_x to x is written to $y.val$ before $y.val$ is read in *att′*. This means that in an attempt $att'' \notin \{att, att'\}$, an instruction WRITEDI(y, ptr_x) is simulated. Moreover, as in *att′*, this instruction is preceded by the simulation of a READDI() instruction that returns x. We apply inductively the same reasoning to *att″* to prove the Lemma. In each induction step, the number of configurations between the creation of x (in *att*) and the first time a READDI() that returns x is simulated in the attempt considered strictly decreases. This ensures the termination of the induction process. □

Next lemma establishes that in every configuration, no two operations that are in their modifying phase reference the same varrec. This lemma plays a central role in the definition of the state of the data structure at the end of a prefix of the (concurrent) execution.

Lemma 19. *Let op, op' denote two distinct operations, and let C be a configuration. Suppose that in C, $op \rightarrow status = \langle modifying, chgs, \_\rangle$ and $op' \rightarrow status = \langle modifying, chgs', \_\rangle$, where $chgs$ and $chgs'$ are pointers to dictionaries d and d' respectively. Then there is no* dictrec *with the same key in both d and d'.*

Proof. Assume, by contradiction, that dictionaries d and d' have a dictrec whose *key* field points to the same varrec x in configuration C. Since every process owns at most one operation with $status \neq done$ in every configuration (Observation 9), $op \rightarrow owner \neq op' \rightarrow owner$.

Consider a process that changes the status of op to $\langle modifying, chgs, \_\rangle$. This occurs when this process performs a SC on $op \rightarrow status$ (on line 52). Since once the status of an operation is $\langle modifying, \_, \_\rangle$, it can only change to *done* (Observation 7), and for this SC to be successful, the status of op must be *simulating* in the configuration in which it is applied, there is a unique such process. Denote by p this process. Before changing the status of op to $\langle modifying, chgs, \_\rangle$, p performs a (successful) attempt of op (lines 36 - 50). Denote att this attempt. Note that the dictionary associated with att is d. Hence, a dictrec $\langle x, \_\rangle$ is added to d during att. Define similarly attempt att' by process p', the successful attempt of op' that ends with the SC that changes the status of op' to $\langle modifying, chgs', \_\rangle$. As in att, a dictrec $\langle x, \_\rangle$ is added to d' in att'.

We consider two cases, according to the instructions simulated when a dictrec with a pointer ptr_x to x is added in att or att'.

- In both att and att', some dictrec with key x is added to d when a $\text{READDI}(x)$ or $\text{WRITEDI}(x, \_)$ is simulated. By the code, p calls in att $\text{ANNOUNCE}(opptr, x)$ and $\text{CONFLICTS}(opptr, x)$ (on lines 39 and 40, respectively) before adding a dictrec $\langle ptr_x, \_\rangle$, to its dictionary (on line 42), where $opptr$ is pointing to op. Similarly, p' calls in att' $\text{ANNOUNCE}(opptr', x)$ and $\text{CONFLICTS}(opptr', x)$, where $opptr'$ is a pointer to op', and p' returns from both calls. Assume without loss of generality that p' returns from $\text{ANNOUNCE}(opptr', x)$ after p returns from $\text{ANNOUNCE}(opptr, x)$ by p. Denote by C'_D the configuration immediately after p' returns from $\text{CONFLICTS}(opptr', x)$. As att' is a successful attempt, whose end occurs when p' changes the status of op' to $\langle modifying, \_, \_\rangle$, att' is active in C'_D.

  Therefore, by Lemma 17, att is not active in C'_D and, since att is a successful attempt, the status of op is *done* in this configuration. This contradicts the fact that the status op and op' is $\langle modifying, \_, \_\rangle$ at C that folows C'_D.

- A dictrec with key x is added to d or d' when a $\text{CREATEDI}()$ is simulated. Whenever a new varrec is created (on line 44), a distinct shared memory slot is allocated to this varrec. A dictrec record $\langle ptr_x, \_\rangle$ cannot thus be added in both d and d' at line 46 when a $\text{CREATEDI}()$ instruction is simulated.

  Suppose without loss of generality that, in att, $\langle x, \_\rangle$ is added to d on line 46, as a result of the simulation of a $\text{CREATEDI}()$ instruction. ptr_x is thus added to d' the first time a $\text{READDI}(x)$ or $\text{WRITEDI}(x, \_)$ instruction for op' is simulated by p' in att'. By Lemma 18, op status is *done* in the configuration immediately before the simulation of this instruction begins. Therefore there is no configuration in which the status of op and op' is $\langle modifying, \_, \_\rangle$: a contradiction. ☐

Suppose that att is a successful attempt of op. Hence, the status of op is changed just after att to $\langle modifying, chgs, \_\rangle$. The changes resulting from the instructions simulated in att are stored in the dictionary pointed to by $chgs$. While the status of op is $\langle modifying, chgs, \_\rangle$, some processes try to apply these changes by modifying the value of the varrecs referenced by op (on lines 54–64). Next lemma establishes that the changes described by the dictionary pointed to by $chgs$ are successfully applied by the time that the status of op is changed to *done*.

Lemma 20. *Suppose that C_M is the last configuration in which the status of op is $\langle modifying, chgs, \_\rangle$, where $chgs$ is a pointer to a dictionary d of* dictrec*s. Let C be a configuration that follows C_M. For every* dictrec *$\langle ptr_x, v\rangle$ in d, where ptr_x is a pointer to a* varrec *x, $ptr_x \rightarrow val = v$ in C or there exists a configuration C' following C_M and preceding C and an operation op' such that op' is referencing x in C'.*

Proof. Let p be the process that successfully performs $\texttt{SC}(op \rightarrow status, done)$ on line 63 just after C_M. Suppose that in every configuration C' following C_M and preceding C, no operation references x. Assume, by contradiction, that $ptr_x \rightarrow val = v' \neq v$ in C.

Consider the steps performed by p in the execution of the iteration of the **for** loop (lines 57 - 62) that corresponds to the `dictrec` $\langle ptr_x, v\rangle$. Notice that these steps precede C_M. In this iteration, p tries to change the *val* of x to v. Since p is the process that changes the status of *op* to *done*, it follows that p does not return on lines 58 and 61. Thus, p executes two `SC` instructions $\texttt{SC}_1$ and $\texttt{SC}_2$ on lines 59 and 59, respectively; let $\texttt{LL}_1$ and $\texttt{LL}_2$ be the matching `LL` instructions to these `SC`. Notice that, for each $i \in \{1, 2\}$, there is a successful `SC` between $\texttt{LL}_i$ and $\texttt{SC}_i$. Let $\texttt{SC}'_i$ be this successful `SC` (notice that $\texttt{SC}'_i$ may be $\texttt{SC}_i$ if $\texttt{SC}_i$ is successful).

Since $ptr_x \rightarrow val = v' \neq v$ in configuration C, some process changes $ptr_x \rightarrow val$ to v'. Let p' be the last process that changes $ptr_x \rightarrow val$ to v' prior to C. By the code, p' performs successfully $\texttt{SC}(ptr_x \rightarrow val, v')$ on line 59 or 62; denote by $\texttt{SC}'$ this `SC` and let $\texttt{LL}'$ and $\texttt{VL}'$ be its mathing `LL` and `VL` (which are executed on lines 57 and 58 or 60 and 61), respectively. Since $ptr_x \rightarrow val = v' \neq v$ in C, either $\texttt{SC}' = \texttt{SC}'_2$ or $\texttt{SC}'$ occurs after SC'_2.

The status of op' when VL' is executed is $\langle modifying, chgs', \_\rangle$, where $chgs'$ is a pointer to a dictionary that includes a `dictrec` $\langle x, v'\rangle$, thus op' references x when VL' is executed. Since we have assumed that no operation references x in any configuration between C_M and C, VL' precedes C_M. By Lemma 19, x cannot be referenced by two operations at the same time. Hence, VL' occurs before the status of *op* is changed to $\langle modifying, chgs, \_\rangle$. In particular, VL', and therefore also $\texttt{LL}'$ precedes $\texttt{LL}_1$. Since SC' is realized at SC'_2 or after it, $\texttt{SC}'_1$ occurs between $\texttt{LL}'$ and $\texttt{SC}'$. Thus, $\texttt{SC}'$ is not successful. This is a contradiction. ☐

Recall that the state of a sequential data structure is a collection of pairs (x, v) where x is a data item and v is a value for that data item. The state of the data structure we consider does not depend on where its data items are stored, so by the value of a pointer we mean which object it points to and not the location of that object in shared memory. The initial state of a sequential data structure consists of its static data items and their initial values.

Initially, there is one `varrec` for each static data item of the data structure. Each `varrec` that is created (on line 44) becomes a public dynamic data item if the attempt that creates it is successful. The *current value* of a `varrec` in a configuration is the value of its *val* field, unless the `varrec` is referenced by an operation *op*, in which case it is the *newval* field in `dictrec`, the dictionary contained in *op*'s *status*, whose *key* points to this `varrec`. Note that, by Lemma 19, in each configuration, each `varrec` is referenced by at most one operation.

Recall that a `varrec` is public in configuration C if it corresponds to a `varrec` of a static data item or there exists a configuration C' equal to C or preceding it in which it is referenced by an operation. For every configuration C in α, denote by D_C the set of pairs (x, v), where x is a public `varrec` and v is its current value in C. Notice that $D_0 = S_0$, where S_0 is the initial state of the data structure. We establish in Theorem 24 that, after having assign linearization points to operations, D_C is the state of the data structure that results if the operations linearized before C are applied sequentially, in order, starting from the initial state, i.e., that $D_C = S_C$.

If an attempt by p of an operation *op* is active in configuration C, we define the *local state* of the data structure in C for the operation and the process that performs the attempt as follows.

Definition 21. *For every configuration C and every operation op, if an attempt att by p of op is active in C, the* local state $LS(C, p, op)$ *of the data structure in configuration C for att is the set of pairs (x, v) such that, in configuration C:*

- *the dictionary associated with att contains a* `dictrec` $\langle x, v\rangle$ *or,*
- *the dictionary associated with att does not contain any* `dictrec` *with key x and $(x, v) \in D_C$.*

The goal is to capture the state of the data structure after the instructions simulated so far in *att* are applied sequentially to D_C. We will indeed establish in Theorem 24 that $LS(C, p, op)$ is the state of the data structure, resulting from the sequential application of the instructions of att simulated thus far by p to S_C. Operations are linearized as follows:

Definition 22. *Each operation is linearized at the first configuration in the execution at which its status is* $\langle modifying, \_, \_\rangle$.

By the code and the way the linearization points are assigned, it follows that:

Lemma 23. *The linearization point of each operation is within its execution interval.*

We continue with our main theorem which proves consistency.

Theorem 24 (Linearizability)**.** *Let C be any configuration in execution α. Then, the following hold:*

1. $D_C = S_C$.
2. *Let att be an attempt of an operation op by a process p that is active in C and let τ be the sequence of instructions of op that have been simulated by p until C. Denote by ρ the sequence of the first $|\tau|$ instructions in a sequential execution of op starting from state S_C. Then, $\rho = \tau$ and $LS(C, p, op) = S_C\tau$, where $S_C\tau$ is the state of the data structure if the instructions in τ are applied sequentially starting from S_C.*

The proof of Theorem 24 relies on the following lemma.

Lemma 25. *Let att denote an attempt by p of some operation op. Suppose that in att, $x \to val$ is read by p while an instruction* ReadDI*(x) is simulated (line 41), let r be this read of $x \to val$, let v be the value returned by r, and denote by C_r the configuration immediately before this read. Then, in every configuration C such that C is C_r or some configuration that follows C_r and att is active at C, v is the value of x in D_C.*

Proof. Assume, by contradiction, that in some configuration C_b between C_r and C, the value of x in S_{C_b} is not v. Denote by C' the first such configuration, and let v' be the value of x in $S_{C'}$. Note that C' may be configuration C_r.

By definition of $S_{C'}$, v' is the current value of x in $S_{C'}$ if either there exists an operation op' whose status is $\langle modifying, chgs', \_\rangle$ where $chgs'$ is pointing to a dictionary that contains a `dictrec` with key x or no such operation exists and $v' = x \to val$.

In configuration C_r, which is equal to C' or precedes C', $x \to val = v \neq v'$. Since in every configuration C'' between C_r and C' (if any), the value of x is v in $S_{C''}$, there exists an operation op' whose status is $\langle modifying, chgs', \_\rangle$ where $chgs'$ is pointing to a dictionary that contains a `dictrec` with key x. By Lemma 19, op' is unique.

Let p' be the process that changes the status of op' from *simulating* to $\langle modifying, chgs', \_\rangle$. Notice that this occurs before C'. By the code, it follows that p' calls Announce($opptr', x$) and Conflicts($opptr', x$) where $opptr'$ is pointing to op'. Denote by C'_A and C'_D the configurations in which p' returns from Announce($opptr', x$) and Conflicts($opptr', x$), respectively. Notice that C'_A and C'_D precede C'.

By the code it follows that before reading $x \to val$, p calls Announce($opptr, x$) and Conflicts($opptr, x$) where $opptr$ is pointing to op. Denote by C_A and C_D the configurations in which p returns from Announce($opptr, x$) and Conflicts($opptr, x$), respectively. Notice that C_A and C_D precede C_R and therefore also C'.

We consider two cases based on the order in which C_A and C'_A occur.

- C'_A occurs after C_A. By Lemma 17, att is not active in C'_D. This is a contradiction, since att is active in configurations C_A and C, and C'_D occurs between C'_A (which, by assumption, follows C_A) and C.
- C_A occurs after C'_A. The attempt of op' by p' in which it calls Announce($opptr', x$) and Conflicts($opptr', x$) is successful, since p' is the process that changes the status of op' to $\langle modifying, \_, \_\rangle$. Thus, it follows from Lemma 17 that the status of op' in C_D is *done*, contradicting the fact that op' status is $\langle modifying, \_, \_\rangle$ at C' that occurs later. ∎

We finally prove Theorem 24.

Proof. The proof is by induction on the sequence of configurations in α. The claims are trivially true for the initial configuration C_0. Suppose that the claims is true for configuration C and every configuration that precedes it. Let C' be the configuration that immediately follows C in α.

We first prove claim 1. If no operation has its status changed to $\langle modifying, \_, \_\rangle$ between C and C', then $D_{C'} = D_C = S_C$. This follows from the definition of D_C, Lemma 20, and the induction hypothesis (claim 1). Otherwise, denote by *op* the operation whose status is changed to $\langle modifying, chgs, \_\rangle$ in C'. The status of *op* is changed by a `SC` performed by some process p on line 52. This `SC` ends a (successful) attempt *att* of *op* by p. Then, in configuration C', the dictionary pointed to by *chgs* is the dictionary associated with *att*. Hence, by definition of $D_{C'}$ and $LS(C, p, op)$, $D_{C'} = LS(C, p, op)$. By the inductive hypothesis (claim 2), $LS(C, p, op) = S_C\tau$, where τ is the sequence of instructions simulated by *att* until C. Notice that the last instruction of τ is the last instruction of *op* and *op* is the only operation that is linearized at C'. Thus, by definition of $S_{C'}$, it follows that $S_C\tau = S_{C'}$. Since $LS(C, p, op) = S_C\tau$, and $D_{C'} = LS(C, p, op)$, it follows that $D_{C'} = S_{C'}$, as needed by claim 1.

Since by claim 1, $D_{C'} = S_{C'}$, it follows that for each data item in $S_{C'}$ there is a unique `varrec` in $D_{C'}$ that corresponds to this data item and vice versa. So, in the rest of proof, we sometimes abuse notation and use x to refer either to a `varrec` in $D_{C'}$ or to a data item in $S_{C'}$.

We now prove claim 2. Let *att* be an attempt by p of some operation *op*. If *att* is not active in C but is active in C', the step preceding C' is a `LL` that reads the status of *op* (on lines 28, 33, 53 or 64). In that case, no step of *op* has been simulated until C', so ρ and τ are empty and by definition, $LS(C', p, op) = S_{C'}$. So, claim 2 holds trivially in this case.

In the remaining of the proof, we assume that *att* is active in both C and C'. Denote by τ and τ' the sequences of instructions of *op* simulated in *att* until C and C', respectively. Let d_C and $d_{C'}$ be the values of the dictionary d that is associated with attempt *att*, in configurations C and C', respectively.

We argue below that two properties, called P1 and P2 below, which are important ingredients of the proof, are true:

P1 Let C_i be either C or a configuration that precedes C in which *att* is active. Let τ_i be the sequence of instructions that have been simulated in *att* until C_i. If x is a `varrec` such that ReadDI(x) is the first access of x in τ_i then the value of x is the same in states S_{C_i} and $S_{C'}$.

To prove P1, denote by v the value returned by the simulation of the first ReadDI(x) in τ_i. Notice that this is also the value read on line 41 when ReadDI(x) is simulated in *att*. Also, since ReadDI(x) has been simulated by C_i, it follows that this read precedes C_i. Since *att* is active in configurations C_i and C', Lemma 25 implies that v is the value of x in both states S_{C_i} and $S_{C'}$.

P2 Let C_i be either C or a configuration that precedes C in which *att* is active. Denote by d_{C_i} the value of d in C_i and by τ_i the sequence of instructions that have been simulated in *att* until C_i. A `dictrec` $\langle x, v\rangle$ is contained in d_{C_i} if and only if x has been accessed in τ_i and v is the value of x in $S_{C_i}\tau_i$.

To prove P2, notice that by the code, a `dictrec` with key x is added to d if and only if an instruction accessing x is simulated (on lines 42 or 46). By the induction hypothesis for C_i (claim 2), $S_{C_i}\tau_i$ is well defined and $LC(C_i, p, op) = S_{C_i}\tau$. Thus, by the definition of $LC(C_i, p, op)$, $\langle x, v\rangle$ is contained in d_{C_i} if and only if x has been accessed in τ_i and v is the value of x in $S_{C_i}\tau_i$.

Fix any x that *att* has accessed for the first time by performing ReadDI(x). Property P1 implies that x has the same value in S_C and $S_{C'}$. Since we have assumed that operations are deterministic and the state of the data structure does not depend on where its data items are stored, it follows that the first $|\tau|$ instructions of *op* are the same and return the same values, independently of whether they are applied in a sequential execution starting from S_C or from $S_{C'}$. Since, by the induction hypothesis (claim 2), τ is the same sequence as that containing the first $|\tau|$ instructions of *op* executed sequentially starting from state S_C, τ is also the same as the sequence of first $|\tau|$ instructions of *op* executed sequentially starting from state $S_{C'}$. Thus, if $\tau = \tau'$, claim 2 follows.

Assume now that τ and τ' differ, i.e., $\tau' = \tau \cdot ins$. Let C'' be the configuration immediately before the simulation of ins starts. If the simulation of ins starts on line 36, that is, τ is the empty sequence and thus $\tau' = ins$ and ins is the first instruction of *op* executed. Thus, ins is the first instruction of *op* when executed sequentially starting from state S'_C. Otherwise, the simulation of ins starts on line 50. In C'', the sequence of instructions of *op* that have been simulated is τ. The fact that it is instruction ins that is simulated next depends on the input of *op*, the value $d_{C''}$ of the dictionary d in configuration C'' and *op*'s program. On the other hand, in a sequential execution, the instruction of *op* that follows τ depends only on the input of *op*, the value of each data item accessed in τ after τ has been applied, and *op*'s program. By property $P2$ applied to C'', d contains in C'' a `dictrec` $\langle x, v\rangle$ if and only if x is accessed in τ and v is the value of x in $S_{C''}\tau$. Therefore ins is the instruction of *op* that follows τ in any sequential execution in which *op* is applied to $S_{C''}$.

Moreover, in a sequential execution of *op* starting from state $S_{C'}$, τ is also the sequence of the first instructions of *op*. Hence, the same data items are accessed by the first $|\tau|$ instructions of *op*, regardless of whether *op* is applied to $S_{C''}$ or $S_{C'}$. Moreover, by property $P1$ applied to C'' and the fact that program of *op* is deterministic, each of these data items have the same value in $S_{C''}\tau$ and $S_{C'}\tau$. Therefore, ins is also the next instruction of *op* following τ in any sequential execution in which *op* is applied to $S_{C'}$. We thus conclude that the first $|\tau'|$ instructions of *op* when executed starting from state $S_{C'}$ in a sequential execution is τ'.

By the code, a `dictrec` with key x is added to d if and only if an instruction accessing x is simulated (on lines 42 or 46). Hence, in configuration C', there is a `dictrec` with key x in d if and only if x is accessed in τ' when *op* is applied to $S_{C'}$ in a sequential execution. Therefore, the set of `varrecs` in $LC(C', p, op)$ is the same as the set of data items in the state $S_{C'}\tau'$. Consider two pairs $(x, v) \in LC(C', p, op)$ and $(x, u) \in S_{C'}\tau'$. To complete the proof that $LC(C', p, op) = S_{C'}\tau'$, we show that $u = v$:

- There is no `dictrec` with key x in d in configuration C', or equivalently, x is not accessed by any instruction of τ' when *op* is applied to $S_{C'}$ in a sequential execution. Then the value of x in $LC(C', p, op)$ is the value of x in $S_{C'}$ which is the value of x in $S_{C'}\tau'$.

- $\tau' = \tau$ or $\tau' = \tau \cdot ins$ but x is not accessed by ins. In that case, the value v of x in $LC(C', p, op)$ is also the value of x in $LC(C', p, op)$. By the induction hypothesis, v is also the value of x in $S_C\tau$. Since $\tau = \tau'$ or ins is not accessing x, v is also the value of x in $S_{C'}\tau'$.

- $\tau' = \tau \cdot ins$ and x is accessed by ins. If ins is $\textsc{ReadDI}(x)$ and x is not accessed in τ, it follows from Lemma 25 and the fact that *att* is active in C' that v is the value of x in $S_{C'}$. Thus v is also the value of x in $S_{C'}\tau'$. If ins is $\textsc{ReadDI}(x)$ but x is accessed in τ, x has the same value in $LS(C, p, op)$ and in $LS(C', p, op)$. Since x has also the same value in $S_{C'}\tau'$ and $S_C\tau$, it follows by the induction hypothesis that x has the same value in $LS(C', p, op)$ and $S_{C'}\tau'$.

  Finally, if $ins = \textsc{WriteDI}(x, v)$ or ins is a $\textsc{CreateDI}()$ that creates x, x has the same value (v or *nil* if $ins = \textsc{CreateDI}()$) in both $LC(p, C', op)$ and $S_{C'}\tau'$.

□

6.3 Wait Freedom

Consider any sequential data structure and suppose there is a constant M such that every sequential execution of an operation applied to the data structure starting from any (legal) state accesses at most M data items. Then we will prove that, in any (concurrent) execution α of our universal construction, DAP-UC, applied to the data structure, every call of PERFORM by a nonfaulty process eventually returns.

Observation 26. *For every* `oprec`, *tohelp*$[p']$ *is initially nil and is only changed to point to* `oprecs` *with owner* p'.

This follows from the fact that *tohelp*$[p']$ is initialized to *nil* when the `oprec` is created (on line 22) and when it is updated (on line 85), *opptr*$'$ points to an `oprec` whose owner is p', by Observation 10 (line 77).

We say that *op restarts* op' in an execution if some process calls CONFLICTS($opptr, x$), where *opptr* points to *op* and x points to a `varrec`, and successfully performs $\texttt{SC}(opptr' \to status, \langle restart, opptr\rangle)$ (on line 87), where $opptr'$ points to op'. Note that, by line 84, this can only happen if the owner of *op* has higher priority (i.e. smaller identifier) than the owner of op'. Thus, an operation cannot restart another operation that has the same owner. Next, we show that an operation cannot restart more than one operation owned by each other process.

Lemma 27. *For any operation op and any process p other than its owner, there is at most one time that op restarts an operation owned by p.*

Proof. Suppose operation *op* has restarted operation op' owned by process p. Before any process can change the status of op' from $\langle restart, opptr\rangle$ back to *simulating* (on line 32), where *opptr* is a pointer to *op*, it performs HELP(*opptr*) on line 31. When this returns, the status of *op* is *done*, by Observation 8.

Consider any process q performing HELP(*opptr*) with *opptr* pointing to *op*, after the status of *op* has been set to *done*. If, when it performs `LL` on line 79, q sees that op' has status *simulating*, it will see that the status of *op* is done, when it performs line 86. Hence, q will not restart op' on line 87. ☐

Conversely, we show that an operation cannot be restarted more than twice by operations owned by a single process.

Lemma 28. *For any operation op' and for any process p other than its owner, at most two operations owned by p can restart op'.*

Proof. Let S be the set containing those operations initiated by p that restart op', which is owned by process $p' \neq p$. Let $opptr'$ be a pointer to the `oprec` record of op'. Let $|S| = k$ and assume, by the way of contradiction, that $k > 2$. Let $op_i \in S$, $1 \leq i \leq k$, be the i-th operation that restarts op' when a process q_i executing an attempt of op_i successfully executes the `SC` on line 87 for op'; let $opptr_i$ be a pointer to the `oprec` record of op_i. Before doing so, q_i set $opptr_i \to tohelp[p'] = opptr'$ (on line 85) and then checked that the status of op_i was still *simulating* (on line 86); thus, $opptr_i \to tohelp[p']$ is written before the completion of op_i.

Lemma 27 implies that op_i will not restart any other operation owned by process p'. Recall that p does not call PERFORM recursively, either directly or indirectly; so, before op_{i+1} is initiated by p, p's call of PERFORM($opptr_i$) should respond (on line 26). Before this response, p reads $opptr_i \to tohelp[p']$ on line 25. Since, the call of HELP($opptr_i$) by p (on line 23) has responded before this read, Observation 8 implies that this read is performed after the *status* of op_i changed to *done*; thus, it is performed after q_i set $opptr_i \to tohelp[p'] = opptr'$.

If in the meantime the value of $opptr_i \to tohelp[p']$ has not changed, then p calls HELP($opptr'$). By Observation 8, the status of op' is *done* when this call responds. Thus, any subsequent operation owned by p will see the status of op' is *done* and will not *restart* it. So, it should be that in the meantime some process q'_i set $opptr_i \to tohelp[p'] = opptr'_i$, where $opptr'_i \neq opptr'$, while executing an attempt of $opptr_i$. Observation 26 implies that $opptr'_i$ points to the `oprec` record of some operation op'_i initiated by p'; op'_i should be initiated by p' before op', since otherwise Observation 9 implies that the status of op' has changed to *done* (so, any subsequent operation owned by p will see the status of op' is *done* and will not *restart* it). Observation 26 implies that the status of any operation initiated by p' before $opptr'$ (including $opptr'_i$), changed to *done* before the initiation of $opptr'$, that is before q_i sets $opptr_i \to tohelp[p'] = opptr'$, that is before p reads $opptr_i \to tohelp[p']$ (on line 25), that is before p initiates $opptr_{i+1}$.

Now consider any j, $1 < j \leq k$. Notice that q'_j reads $opptr'_j$ on line 77 and before it executes line 85, which sets $opptr_j \to tohelp[p'] = opptr'_j$, it reads the *status* of $opptr'_j$ (on line 79) and checks whether it is still *simulating* (on line 83). Since, this read is performed after the initiation of $opptr_j$, it follows that before it the *status* of $opptr'_j$ has changed to *done*. So, the check fails and line 85 is not executed; that is a contradiction. ☐

From Lemmas 27 and 28, we get the following result.

Corollary 29. *An operation can be restarted at most* $2 * (n - 1)$*.*

Next, we bound the depth of recursion that can occur.

Lemma 30. *Suppose that, while executing* HELP($opptr_i$)*, a process calls* HELP($opptr_{i+1}$)*, for* $1 \leq i < k$*. Then* $k \leq n$*.*

Proof. Process p may perform recursive calls to HELP($opptr'$) on lines 31, 82, 88, and 89. If p calls HELP($opptr'$) recursively on line 82 or 88, then, by Observation 7, $opptr' \rightarrow status$ is either *modifying* or *done*, so, this recursive call will eventually return without itself making recursive calls to HELP.

By line 77 and Observation 10, when line 84 is performed, $opptr' \rightarrow owner = p'$. From line 84, if p calls HELP($opptr'$) recursively on line 89, then $opptr \rightarrow owner > opttr' \rightarrow owner$.

If $opptr' \rightarrow status = \langle restart, opptr \rangle$, then, from lines 87 and 84, $opptr \rightarrow owner < opttr' \rightarrow owner$. Hence, if p calls HELP($opptr'$) recursively on line 31, $opptr \rightarrow status = \langle restart, opptr' \rangle$, so, again, $opptr \rightarrow owner > opttr' \rightarrow owner$.

Thus, in any recursively nested sequence of calls to HELP, the process identifiers of the owners of the operations with which HELP is called is strictly decreasing, except for possibly the last call. Therefore $k \leq n$. ☐

Lemma 31. *Every call of* HELP*(opptr) by a nonfaulty process eventually returns.*

Proof. Consider any call of HELP(*opptr*) by a nonfaulty process p where *opptr* points to *op*. Immediately prior to every iteration of the **while** loop on lines 29–63 during HELP(*opptr*), process p performs `LL`($opptr \rightarrow status$) on line 28, 33, 53, or 64.

If *op* has status *done* at the beginning of an iteration, HELP(*opptr*) returns immediately. If *opptr* has status *modifying*, no recursive calls to HELP are performed during the iteration. Then, Observation 15 and Theorem 24 (item 1) imply that the `dictrecs` in a dictionary have different keys (i.e. point to different `varrecs`) and correspond to different data items accessed by a sequential execution of *op* applied to the data structure (lines 38, 42, and 46). Thus, the total number of `dictrecs` in a dictionary is bounded above by M and, so, at most M iterations of the **for** loop on lines 56–62 are performed. Hence HELP(*opptr*) eventually returns.

If *opptr* has status *restart*, then, during an iteration of the **while** loop, p performs one recursive call to HELP (on line 31) and, excluding this, performs a constant numbers of steps.

Finally, suppose that *opptr* has status *simulating* at the beginning of an iteration. Theorem 24 (item 2) implies that p simulates a finite number of instructions while it is executing an active attempt of *op*. After this attempt becomes inactive, the test on line 49 evaluates to true during this iteration, so p may simulate at most one more instruction during this iteration; so, the number of instructions is finite. For each instruction in its program, p performs one iteration of the **while** loop on lines 37–50, in which it takes a constant number of steps, excluding calls to CONFLICTS. Observation 15, Theorem 24 (item 2), and the definition of M, imply that CONFLICTS can be called at most M times during an active attempt of *op*. Then, Theorem 24 (item 2) imply that process p performs a constant number of steps and at most one recursive call to HELP (on line 82, 88, or 89) each time it calls CONFLICTS. Thus, excluding the recursive calls to HELP, this iteration of the **while** loop on lines 29–63 eventually completes.

If p does not return on line 65 after exiting from the **while** loop or on line 58 or 61, it tries to change $opptr \rightarrow status$ via an `SC` on line 32, 52, or 63. Therefore, each time p performs an iteration of the **while** loop on lines 29–63, $opptr \rightarrow status$ changes. It follows from Observation 7 and Corollary 29 that p performs at most $2n$ complete iterations of this **while** loop during HELP(*opptr*).

By Lemma 30, the depth of recursion of calls to HELP is bounded. Therefore, the call of HELP(*opptr*) by p eventually returns. ☐

Finally, we prove wait freedom:

Theorem 32. *Every call of* PERFORM *by a nonfaulty process eventually returns.*

Proof. Consider any call of PERFORM by a nonfaulty process. In PERFORM, the process calls HELP at most n times (excluding recursive calls), each time for an `oprec` owned by a different process It follows from Lemma 31 that all these instances of HELP eventually return. Thus, this call of PERFORM eventually returns. ☐

6.4 Disjoint access parallelism

As in the other part of the proof, we consider an execution α of our universal construction applied to some data structure. Recall that the execution interval I_{op} of an operation *op* starts with the first step of the corresponding call to PERFORM() and terminates when this call returns. In the following to simplify the presentation we denote PERFORM(*op*) the call to PERFORM corresponding to operation *op*.

Let C_{op} be the configuration immediately after p performs line 22, that is, immediately after an `oprec` has been initialized for *op*, and let C'_{op} be the first configuration at which the status of *op* is $\langle modifying, \_, \_\rangle$. Note that $C_{i'}$ is the configuration at which *op* is linearized, see Definition 22.

Let $\mathcal{S} = \{S_C \mid C$ is between C_{op} and $C'_{op}\}$. Then, for the data set $DS(op)$ of *op*, it holds that $DS(op) = \cup_{S_C \in \mathcal{S}}\{$set of data items accessed by *op* when executed sequentially starting from $S_C\}$.

We recall also the definition of the conflict graph of an execution interval I. The conflict graph is an undirected graph, where vertices represent operations whose execution interval overlaps I and an edge connects two operations whose data sets intersect. Given two operations *op* and *op*$'$, we denote by $CG(op, op')$ the conflict graph of the minimal execution interval that contains I_{op} and $I_{op'}$. Finally, recall that we say that two processes contend on a base object b if they both apply a primitive on b, and at least one of these primitives is non-trivial.

Recall that an *attempt* of an operation *op* by a process p is a longest execution interval that begins when p performs `LL` on $op \rightarrow status$ on line 28, 33, 53 or 64 that returns *simulating* and during which $op \rightarrow status$ does not change.

Lemma 33. *When* ANNOUNCE$(opptr, x)$ *is called, the data item x is in the data set of the operation to which opptr points.*

Proof. Let C be the configuration before p calls ANNOUNCE$(opptr, x)$ at which p last performs an `LL` or a successful `VL` on $opptr \rightarrow status$ (on lines 28, 33, or 49). By the code, such a configuration C exists, and if p performs an `LL` at C, this `LL` returns *simulating*. Hence, an attempt *att* of *op* by p, the operation pointed to by *opptr*, is active in configuration C. It thus follows from Theorem 24(2) that the sequence of instructions τ of *op* that have been simulated before C is the same as in a sequential execution of *op* applied to S_C. Hence, as in the concurrent execution, ANNOUNCE$(opptr, x)$ is called in a simulation of a write to or of a read from x following τ, x is also accessed in the sequential execution of the first instructions τ of *op* applied to S_C. Therefore, $x \in DS(op)$. ☐

Inspecting the code of ANNOUNCE, we then obtain:

Corollary 34. *If $x \rightarrow A[p] \neq nil$, then the data item x is in the data set of the operation to which $x \rightarrow A[p]$ points.*

Observation 35. *If a process executes a successful* `VL`*(opptr $\rightarrow$ status) while performing* ANNOUNCE*(opptr, x) or* CONFLICTS*(opptr, x), then the* `oprec` *to which opptr is pointing has status simulating.*

This is because a process only calls ANNOUNCE$(opptr, x)$ (on line 39) and CONFLICTS$(opptr, x)$ (on line 40) if $opptr \rightarrow status$ was *simulating* (line 34) when p last executed `LL`$(opptr \rightarrow status)$ (on line 28, 33, or 53).

When helping an operation *op*, process p may starts helping another operation *op*$'$. This occurs for example when a conflict between the two operations is discovered by p, that is, when the two operations access the same `varrec`. Next Lemma shows that indeed, when p calls HELP(op') while executing HELP(op), the datasets of *op* and *op*$'$ share a common element.

Suppose that p calls $\textsc{Help}(opptr)$ and $\textsc{Help}(opptr')$, where $opptr$ and $opptr'$ are pointers to operations op and op', respectively. Denote by I the execution interval of $\textsc{Help}(opptr)$. We say that $\textsc{Help}(opptr')$ is *directly called by p after* $\textsc{Help}(opptr)$ if p calls $\textsc{Help}(opptr')$ in I and every other call to $\textsc{Help}$ previously made in by p in I has returned when $\textsc{Help}(opptr')$ is called by p .

Lemma 36. *If* $\textsc{Help}(opptr')$ *with* $opptr'$ *pointing to* op' *is called directly by* p *after calling* $\textsc{Help}(opptr)$ *with* $opptr$ *pointing to* op*, then* $DS(op) \cap DS(op') \neq \emptyset$*.*

Proof. In an instance of $\textsc{Help}(opptr)$ by p, where $opptr$ is pointing to op, $\textsc{Help}(opptr')$ with $opptr'$ pointing to op' may be called on line 31, when p discovers that op has been restarted, or in the resolution of the conflicts for some `varrec` x, when p executes $\textsc{Conflicts}(opptr, x)$ (lines 82, 88 or 89). We consider these two cases separately:

- $\textsc{Help}(opptr')$ is called in the execution of $\textsc{Conflicts}(opptr, x)$. Before calling $\textsc{Conflicts}(opptr, x)$, p calls $\textsc{Announce}(opptr, x)$ (line 39). Therefore, it follows from Lemma 33 that $x \in DS(op)$. For $\textsc{Help}(opptr')$ to be called in $\textsc{Conflicts}(opptr, x)$, $opptr'$ is read from $x \to A[q']$, where q' is the owner of op' (`LL` on line 77). Hence, op' has been previously announced to x, from which we conclude by corollary 34 that $x \in DS(op')$.

- $\textsc{Help}(opptr')$ is called on line 31. This means that some process p' has changed the status of op to $\langle restart, opptr' \rangle$ (`SC` on line 87). p' thus calls $\textsc{Conflicts}(opptr', x)$ for some `varrec` x in which it applies a successful `SC`$(opptr, \langle restart, opptr' \rangle)$. By the code of $\textsc{Conflicts}$, this implies that $opptr$ is read from $x \to A[q]$, where q is the owner of op (`LL` on line 77). Thus, op has been announced to x, from which we have by Corollary 34 that $x \in DS(op)$. Moreover, p' calls $\textsc{Conflicts}(opptr', x)$ after returning from a call to $\textsc{Announce}(opptr', x)$. Hence, by Lemma 33, $x \in DS(op')$. □

When a process p is performing an operation op, i.e., p has called $\textsc{Perform}(op)$ but has not yet returned from that call, it may access `oprec`s of operations $op' \neq op$. We show that if p applies a non-trivial primitive to an `oprec` $op' \neq op$ then the execution interval $I_{op'}$ of that operation overlaps the execution interval I_{op} of op.

Lemma 37. *If* p *applies a non-trivial primitive to an* `oprec` op' *in* I_{op}*,* $I_{op'} \cap I_{op} \neq \emptyset$*.*

Proof. A non-trivial primitive may be applied to `oprec` op' on line 32, 52, 55, 63 in the code of $\textsc{Help}$ or on lines 85 or 87 in the code of $\textsc{Conflicts}$. The non-trivial primitive applied by p on line 32, 52 or 63 is a `SC` that aims at changing the status of op' to *simulating*, $\langle modifying, \_, \_ \rangle$ or *done* respectively. On line 55, the *output* of op' is changed. Any of these steps, if applied by p, is preceded by an `LL`$(opptr' \to status)$ by p (on lines 28, 33, 53 or 64), where $opptr'$ is pointing to op'. The value returns by this `LL` is $\neq done$. Therefore, in the configuration at which this `LL` is applied, the call of $\textsc{Perform}(op')$ has not yet returned. Hence, $I_{op} \cap I_{op'} \neq \emptyset$.

In the remaining case, p writes $opptr'$ to $opptr \to tohelp[p']$ on line 85 or applies `SC`$(opptr' \to, \langle restart, \_ \rangle)$ on line 87. Here also, before these steps, an `LL`$(opptr' \to status)$ by p occurs (on line 79) and this `LL` returns a value $\neq done$. As above, we then conclude that $I_{op} \cap I_{op'} \neq \emptyset$. □

Lemma 38. *If* p *applies a primitive to a* `varrec` x *in* I_{op}*, there exists an operation* op' *such that* $x \in DS(op')$*,* $I_{op'} \cap I_{op} \neq \emptyset$ *and* p *calls* $\textsc{Help}(opptr')$ *where* $opptr'$ *is pointing to* op'*.*

Proof. Let x denote a `varrec` accessed by p. By the code, x is accessed in one of the following cases:

- The step in which p accesses x occurs in a call to $\textsc{Announce}(opptr', x)$ (lines 68, 70, 71, or 73), in a call to $\textsc{Conflicts}(opptr', x)$ (line 77) where $opptr'$ is pointing to some operation op', or in the simulation of $\textsc{ReadDI}(x)$ on behalf of op' (line 41). Each of these accesses to x occurs after p has called $\textsc{Announce}(opptr', x)$. Therefore, by Lemma 33, $x \in DS(op')$. Moreover, before applying any of these steps, p has verified that the status op' is $\neq done$ (by applying a `LL` on $opptr' \to status$ on line 28, 33 or 53). More precisely, consider the last configuration C at which p applies `LL`$(opptr' \to status)$

before accessing x. Such a step occurs since the first step following a call to Help($opptr'$) is a `LL` on $opptr' \to status$ (line 28). This last `LL` must returns *simulating* since p has to pass the test on line 34 before applying any step considered in the present case. Therefore, in C, the call to Perform(op') has not returned, from which we have $I_{op} \cap I_{op'} \neq \emptyset$.

- The step in which p accesses x is a `LL`, `VL` or `SC` on the *val* field of x (lines 57, 58, 59, 60, 61 or 62). Before applying any of these steps, p performs a `LL`($opptr' \to status$) (on lines 28, 33 or 53), where $opptr'$ is pointing to op', which returns $\langle modifying, chgs', \_\rangle$ since the test on line 54 is passed. In the configuration in which this `LL` is applied, the calls to Perform(op) and Perform(op') have not returned, hence $I_{op} \cap I_{op'} \neq \emptyset$.

  Consider the dictionary d' pointed to by $chgs'$. Note that x is the key of a `dictrec` in d'. Hence, in a successful attempt of op' by some process p', a `dictrec` with key x is added to the dictionary associated with that attempt (on line 42 or 46) when an instruction of op' simulated. Therefore, it follows from Theorem 24 that $x \in DS(op')$. ☐

Lemma 39. *If p calls* Help*($opptr'$) in I_{op}, where $opptr'$ is pointing to op', then $I_{op} \cap I_{op'} \neq \emptyset$.*

Proof. Process p can only call Help($opptr'$) on line 23, line 25, line 31, line 82, line 88 or line 89. If p calls Help($opptr'$) on line 23, $op' = op$ and the Lemma holds.

If p calls Help($opptr'$) on line 25, a conflict with op' has been detected by some process q and q has tried to restart op'. More precisely, there exists some process q, and a `varrec` x such that q calls Conflicts($opptr, x$) and, before returning from that call, writes $opptr'$ to $opptr \to tohelp[p]$ (line 85), where $opptr$ is pointing to op. By the code, before calling Conflicts($opptr, x$), q verifies that the status of op is *simulating* by applying a `LL` on $opptr \to status$. Denote by C_{LL} the last configuration that precedes the call to Conflicts($opptr, x$) at which a `LL`($opptr \to status$) is applied by q. $opptr \to status = simulating$ at C. Moreover, it follows from the code of Conflicts that before writing to $opptr \to tohelp[p]$, q performs a successful `VL`($opptr \to status$) on line 80. Let C_{VL} denote the configuration at which this step is applied. By observation 35, $opptr \to status = simulating$ in C_{VL} and has not changed since C_{LL}. In its previous step, q reads $opptr' \to status$ (line 79), and the value it gets back is *simulating*, since the test on line 83 is later passed. Therefore, there exists a configuration between C_{LL} and C_{VL} in which $opptr' \to status = simulating$, from which we conclude that $I_{op} \cap I_{op'} \neq \emptyset$.

Help($opptr'$) is called on line 31. As in the previous case, a process q' performs the successful `SC` that changes $opptr \to status$ to $\langle restart, opptr'\rangle$ (on line 87). This occurs when q' is executing Conflicts($opptr', x$) for some `varrec` x. The same reasoning as in the previous case (inverting $opptr$ and $opptr'$) can be used to establish the existence of a configuration in which $opptr \to status = opptr' \to status = simulating$, from which it follows that $I_{op} \cap I_{op'} \neq \emptyset$.

Otherwise, process p calls Help($opptr'$) on line 82, 88 or 89. Before calling Help($opptr'$) on any of these lines, p has read the status of op' (`LL`($opptr' \to status$) on line 79), and this `LL` returns a value $\neq$ *done* (By the tests on line 82 or line 83, $opptr' \to status$ has to be *simulating* or $\langle modifying, \_, \_\rangle$ in order for p to call Help($opptr'$) on line 82, 88 or 89). As this occurs before p returns from the call of Perform(op), $I_{op} \cap I_{op'} \neq \emptyset$. ☐

Lemma 40. *Suppose that p applies a primitive operation to an* `oprec` *op' after calling* Help*(op) and before returning from that call. Denote by C and C' the configuration at which* Help*(op) is called and the primitive is applied respectively. If every call by p to* Help*() that occurs between C and C' returns before C' then $op = op'$ or $DS(op) \cap DS(op') \neq \emptyset$.*

Proof. Suppose that $op \neq op'$. By the code, p accesses op while executing Conflicts($oppptr, x$) where x is a `varrec` and $opptr$ is pointing to op. Since every call to Conflicts($oppptr, x$) is preceded by a call to Announce($opptr, x$) (lines 39 and 40), it follows from Lemma 33 that $x \in DS(op)$. op' is accessed by p via the announce array $x \to A$. Hence op' has been announced to x and thus by corollary 34, $x \in DS(op')$. ☐

Theorem 41. *Let b be a base object and let op, op' be two operations. Suppose that p and p' apply a primitive on b in I_{op} and $I_{op'}$ respectively. Then, if at least one of the primitives is non-trivial, there is a path between op and op' in $CG(op, op')$.*

Proof. Base object b is a field of either an `oprec` or a `varrec`, a `dictrec` or a `statrec`. A `statrec` can only be accessed through the unique `oprec` that points to it. A `dictrec` can only be accessed through the unique `statrec` that points to the unique dictionary that contains it. Thus to access b, p and p' have to access the same `oprec` or the same `varrec`. We consider these two cases separately:

- p and p' access the same `oprec` op^*. Suppose that op^* is accessed by p and p' while in some instances of $\textsc{Help}()$. That is, there exists an operation op_1 such p calls $\textsc{Help}(opptr_1)$, where $opptr_1$ is pointing to op_1, and has not returned from that call when op^* is accessed. Moreover, when it accesses op^*, p has returned from each of its calls to $\textsc{Help}$ that are initiated after the call to $\textsc{Help}(opptr_1)$ and before the access of op^*.

  This also holds for p' for some operation op'_1. Thus, there exists two chains of operations $\langle op = op_k, \ldots, op_1\rangle$ and $\langle op = op'_{k'}, \ldots, op'_1\rangle$ such that:

  - $\forall i, 1 \leq i \leq k, \forall i', 1 \leq i' \leq k'$: p calls $\textsc{Help}(opptr_i)$ and p' calls $\textsc{Help}(opptr'_{i'})$ where $opptr_i$ and $opptr'_{i'}$ are pointing to op_i and $op'_{i'}$ respectively;
  - $\forall i, 2 \leq i \leq k, \forall i', 2 \leq i' \leq k'$: after calling $\textsc{Help}(opptr_i)$, and before returning from this call, p calls directly $\textsc{Help}(opptr_{i-1})$. Similarly, after calling $\textsc{Help}(opptr'_{i'})$, and before returning from this call, p' calls directly $\textsc{Help}(opptr'_{i'-1})$.

  It thus follows from the second property that for each $i, 2 \leq i \leq k$, $\textsc{Help}(opptr_{i-1})$ is called directly in an attempt of op_i, from which we derive by Lemma 36 that $DS(op_i) \cap DS(op_{i-1}) \neq \emptyset$. Moreover, it follows from Lemma 39 that $I_{op} \cap I_{op_i} \neq \emptyset$, for each $i, 1 \leq i \leq k$. Therefore, operations $op = op_k, \ldots, op_1$ are vertexes of the graph $CG(op, op')$ and there is path from $op = op_k$ to op_1. Similarly, $op = op'_{k'}, \ldots, op'_1$ are vertexes of the graph $CG(op, op')$ and there is path from $op' = op'_{k'}$ to op'_1.

  op^* is also a vertex of $GC(op, op')$ because, as p or p' applies a non-trivial primitive to op^*, $I_{op} \cap I_{op^*} \neq \emptyset$ or $I_{op'} \cap I_{op^*} \neq \emptyset$ by Lemma 37. p applies a primitive to op^* after calling $\textsc{Help}(opptr_1)$ and before returning from this call. Moreover, when this step is applied, every call to $\textsc{Help}()$ by p that follows the call of $\textsc{Help}(opptr_1)$ has returned. Hence by Lemma 40, $op_1 = op^*$ or $DS(op_1) \cap DS(op^*) \neq \emptyset$. Similarly, $op'_1 = op^*$ or $DS(op'_1) \cap DS(op^*) \neq \emptyset$. We conclude that there is a path between op and op' in $GC(op, op')$.

  If $op^* = op$ or $op^* = op'$, one chain consists in a single operation, namely op^*. The reasoning above is still valid.

  Finally, op^* may be accessed by p or p' on line 25, when p or p' helps an operation that may have been restarted by some process helping op or op' respectively. Without loss of generality, assume that op^* is accessed in this way, that is p' accesses op^* by reading $tohelp[p^*]$, where p^* is the owner of op^*. As p next calls $\textsc{Help}(opptr^*)$, where $opptr^*$ is pointing to op^*, it follows from Lemma 39 that $I_{op} \cap I_{op^*} \neq \emptyset$. Therefore, op^* is a vertex of the graph $CG(op, op')$. Consider the step in which $opptr^*$ is written to $opptr \rightarrow tohelp[p^*]$ (line 85). This occurs while $\textsc{Conflicts}(opptr, x)$ is executed, for some `varrec` x. By Lemma 33 and the fact that the call $\textsc{Conflicts}(opptr, x)$ is preceded by a call to $\textsc{Announce}(opptr, x)$, $x \in DS(op)$. Moreover, by the code of $\textsc{Conflicts}()$, op^* has been announced in to x, and thus by corollary 34, $x \in DS(op^*)$. Hence op and op^* are connected in $CG(op, op')$. Depending on how op^* is accessed by p', the same reasoning or the reasoning above can be used to show that there is a path between op^* and op' in $CG(op, op')$. Therefore, there is a path between op and op' in $CG(op, op')$.

- p and p' access the same `varrec` x^*. By Lemma 38, there exists op_1, op'_1 such that (1) p calls $\textsc{Help}(op_1)$ and p' calls $\textsc{Help}(op'_1)$, (2) $x^* \in DS(op_1) \cap DS(op'_1)$ and (3) $I_{op} \cap I_{op_1} \neq \emptyset$ and $I_{op} \cap I_{op'_1} \neq \emptyset$.

If $op'_1 = op_1 = op^*$, p and p' access the same `oprec` op^*. In the proof of the previous item, we use the fact that p or p' applies a non-trivial primitive to op^* only to show that $I_{op^*} \cap I_{op} \neq \emptyset$ or $I_{op^*} \cap I_{op'} \neq \emptyset$. Here, we already known that this holds. Therefore, by the same argument as in the first case, we conclude that there is a path between op and op' in $CG(op, op')$.

If $op'_1 \neq op_1$, we consider the two chains of operations chains of operations $\langle op = op_k, \ldots, op_1 \rangle$ and $\langle op = op'_{k'}, \ldots, op'_1 \rangle$ defined as in the first case. By the same reasoning as in the first case, each of these operations is a vertex and (op_i, op_{i-1}), $(op'_{i'}, op_{i'-1})$ are edges of $CG(op, op')$, for each $i, i' : 2 \leq i \leq k, 2 \leq i' \leq k'$. Since $DS(op_1) \cap DS(op'_1) \neq \emptyset$, we conclude that op and op' are connected by a path in $CG(op, op')$. $\square$

References

[1] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 538–547, New York, NY, USA, 1995. ACM.

[2] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations (extended abstract). In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, PODC '97, pages 111–120, New York, NY, USA, 1997. ACM.

[3] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 184–193, New York, NY, USA, 1995. ACM.

[4] J. H. Anderson and M. Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, dec 1999.

[5] H. Attiya and E. Dagan. Universal operations: unary versus binary. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 223–232, New York, NY, USA, 1996. ACM.

[6] H. Attiya and E. Hillel. Built-in coloring for highly-concurrent doubly-linked lists. In S. Dolev, editor, *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, volume 4167 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2006.

[7] H. Attiya and E. Hillel. Highly-concurrent multi-word synchronization. In *Proceedings of the 9th international conference on Distributed computing and networking*, ICDCN'08, pages 112–123, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] H. Attiya and E. Hillel. Single-version stms can be multi-version permissive. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 83–94, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 69–78, New York, NY, USA, 2009. ACM.

[10] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[11] V. Bushkov, R. Guerraoui, and M. Kapalka. On the liveness of transactional memory. In *Proceedings of the 31st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '12, New York, NY, USA, 2012. ACM.

[12] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 335–344, New York, NY, USA, 2010. ACM.

[13] T. Crain, D. Imbs, and M. Raynal. Towards a universal construction for transaction-based multiprocess programs. In *ICDCN*, pages 61–75, 2012.

[14] D. Dice and N. Shavit. What Really Makes Transactions Faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006.

[15] P. Fatourou and N. D. Kallimanis. The redblue adaptive universal constructions. In *Proceedings of the 23rd international conference on Distributed computing*, DISC'09, pages 127–141, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325 – 334, 2011.

[17] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[18] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 304–313, New York, NY, USA, 2008. ACM.

[19] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming*, PPOPP '90, pages 197–206, New York, NY, USA, 1990. ACM.

[20] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[21] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23:146–196, May 2005.

[22] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[23] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, Jul 1990.

[24] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, PODC '94, pages 151–160, New York, NY, USA, 1994. ACM.

[25] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

[26] D. Pelerman, R. Fan, and I. Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 16–25, New York, NY, USA, 2010. ACM.

[27] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[28] F. Tabba, M. Moir, J. R. Goodman, A. Hay, and C. Wang. NZTM: Nonblocking zero-indirection transactional memory. In *SPAA '09: Proc. 21st Symposium on Parallelism in Algorithms and Architectures*, aug 2009.

[29] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '92, pages 212–222, New York, NY, USA, 1992. ACM.

[30] J.-T. Wamhoff, T. Riegel, C. Fetzer, and P. Felber. Robustm: A robust software transactional system. In S. D. *et al.*, editor, *Stabilization, Safety, and Security in Distributed Systems, Proceedings*, volume 6366 of *Lecture Notes in Computer Science*. Springer, 2010.

A Sequential code for singly-linked list

Figure 5 present the sequential implementation of APPEND and SEARCH. Figure 6 presents the enhanced code of APPEND and SEARCH where CREATEDI READDI and WRITEDI have been incorporated in the code of Figure 5.

According to the enhanced sequential code, we have three types of operations: INITIALIZELIST, APPEND, and SEARCH. INITIALIZELIST ($\mathcal{L}$) initializes two previously declared pointers, *L.start* and *L.end*, to *nil*. APPEND($\mathcal{L}$, *num*) appends the element *num* to the end of the list $\mathcal{L}$ by appending a node containing *num* as the next element of that pointed to by *end*, and updating *end* to point to the newly appended node. SEARCH($\mathcal{L}$, *num*) searches $\mathcal{L}$ for the first occurrence of *num*, starting from the element pointed to by *start*. SEARCH returns `true` if *num* is in $\mathcal{L}$ and `false` otherwise. Throughout this code, if T is a type with some field f, then `ptr to` T t declares that t is a pointer to an object of type T and $t \rightarrow f$ denotes the f field of that object. CREATEDI(T) creates a new data item of type T and returns a pointer to it. READDI() and WRITEDI() are used when accessing a data item or a field of a data item.

```
struct {
      int key: initially 0;
      ptr to Node next: initially nil;
} Node;

struct {
      ptr to Node start: initially nil;
      ptr to Node end: initially nil;
} List;

List L;
APPEND(List L, int value) {
      ptr to Node new := allocate new Node;        /* create a new Node, return a pointer to it */
      ptr to Node e := L.end;
      new → key := value;
      new → next := nil;
      if (e ≠ nil) then e → next := new
      else L.start := new;
      L.end := new;
}

Boolean SEARCH(List L, int value) {
      ptr to Node s := L.start;
      if (s = nil) then return false;
      while (s → key ≠ value AND s → next ≠ nil)
            s := s → next);
      if (s → key = value) then return true;
      else return false;
}
```

Figure 5: Sequential implementation of a singly-linked list data structure that supports APPEND and SEARCH.

```
struct {
        int key: initially 0;
        ptr to Node next: initially nil;
} Node;

struct {
        ptr to Node start: initially nil;
        ptr to Node end: initially nil;
} List;

/* Initialization of the access points of the data structure as static data items */
List L;
L.start := CreateDI(ptr to Node): initially nil
L.end := CreateDI(ptr to Node): initially nil;

/* Programs for the operations passed to the universal construction */
Append(List L, int value) {
        ptr to Node new := CreateDI(Node); initially ⟨value, nil⟩;
        ptr to Node e := ReadDI(L.end);
        if (e ≠ nil) then WriteDI(e → next, new)
        else WriteDI(L.start, new);
        WriteDI(L.end, new);
        return
}

Boolean Search(List L, int value) {
        int k;
        ptr to Node s := ReadDI(L.start);
        if (s = nil) then return false;
        ⟨k, s⟩ := ReadDI(s);
        while(k ≠ value and s ≠ nil)
                ⟨k, s⟩ := ReadDI(s);
        if (k = value) then return true;
        else return false;
}
```

Figure 6: Enhanced version of the pseudocode of Figure 5 that includes calls to CreateDI, ReadDI, and WriteDI.