UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE

Techniques for Enhancing Parallelism in Mechanisms that Automatically Execute Sequential Code in Concurrent Environments

Dissertation Submitted by

Eleftherios Kosmas

in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

Heraklion, February 2015

UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE

Techniques for Enhancing Parallelism in Mechanisms that Automatically Execute Sequential Code in Concurrent Environments

Dissertation submitted by

Eleftherios Kosmas

in partial fulfillment of the requirements for the
Doctor of Philosophy degree in Computer Science

Author: [illegible]
Eleftherios Kosmas

Examination Committee:

Panagiota Fatourou, Assistant Professor, University of Crete

[illegible]
Angelos Bilas, Professor, University of Crete

[illegible]
Dimitrios S. Nikolopoulos, Professor, Queen's University of Belfast

[illegible]
Polyvios Pratikakis, Researcher, Foundation for Research and Technology

[illegible]
Vassilios V. Dimakopoulos, Associate Professor, University of Ioannina

Faith Ellen
Faith Ellen, Professor, University of Toronto

[illegible]
Paolo Romano, Assistant Professor, University of Lisbon

Departmental approval:
Antonis A. Argyros, Professor, University of Crete

Heraklion, February 2015

European Union
European Social Fund

OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS, CULTURE & SPORTS
MANAGING AUTHORITY

NSRF
2007-2013
programme for development
EUROPEAN SOCIAL FUND

Co-financed by Greece and the European Union

Acknowledgements

This work is a result of the continuous support provided by my supervisor Panagiota Fatourou.

I would like to thank the co-authors of my results, Faith Ellen, Alessia Milani, Corentin Travers, Shlomi Dolev, Hillel Avni, Mykhailo Iaremko, Eleni Kanellou, and M. Forhad Rabbi. I am grateful for meeting and working with them, not only because our discussions inspired most of the work in this thesis, but most importantly because they significantly helped me enjoy this long journey.

Also, I would like to thank the members of my advisory committee Angelo Bila and Dimitrio S. Nikolopoulo for their helpful suggestions, and the rest members of my examination committee Polyvio Pratikaki, Vassilio D. Dimakopoulo, Faith Ellen, and Paolo Romano for carefully reading my thesis and providing useful comments.

This work was 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), which is co-funded by the European Union (European Social Fund) and National Resources. Moreover, I would like to acknowledge the support of the Institute of Computer Science of the Foundation of Research and Technology (FORTH-ICS) and especially the DCS and CARV research laboratories.

Abstract

Two well-known mechanisms for automatically executing sequential code segments in a concurrent environment are universal constructions and software transactional memory. They both have the same goal of simplifying the task of parallel programming. A *universal construction* is a mechanism which takes as input the sequential code and executes it in a concurrent environment. *Software transactional memory* (STM) employs *transactions* to avoid conflicting accesses to common data (known as *data items* or *transactional variables*). A transaction may either *commit*, in which case it appears as if it has been executed at a single point in time, or *abort*, in which case it appears as if it is never executed. Notice that if a transaction commits, then its updates to data items become visible, otherwise, if it aborts, all its changes are discarded.

In this thesis, we study how to achieve increased concurrency while designing such mechanisms, without sacrificing correctness and progress. One well-studied technique for enhancing concurrency is ensuring a property called disjoint-access parallelism. Roughly speaking, *disjoint-access parallelism* guarantees that processes operating on different parts of an implemented object do not interfere with each other. Thus, disjoint-access parallel implementations allow for increased parallelism.

Wait-freedom is a well-known progress property which ensures that each process completes its execution, even when other processes run at arbitrary speeds or crash. Wait-freedom is highly desirable because implementations ensuring this property are highly fault-tolerant and usually ensure bounds on the number of steps executed before an implemented operation responds.

In this thesis, we prove that it is not possible for a universal construction to achieve both disjoint-access parallelism and wait-freedom; this impossibility result holds for STM as well. Specifically, 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. Our impossibility proof is obtained by considering a dynamic data structure that can grow arbitrarily large during an execution. This impossibility result can be beaten if we focus on data structures that have a bound on the number of pieces of data accessed by each operation they support. For this setting, we present a universal construction that ensures both wait-freedom and disjoint-access parallelism.

We further introduce and study weaker versions of disjoint-access parallelism, which still however allow for increased parallelism. Motivated be the way current STM algorithm work, we introduce *timestamp-ignoring disjoint-access parallelism*, which allows operations operating on different parts of an implemented data structure to proceed in parallel, except for accesses to a timestamp object. A *timestamp object* allows a process to know the "time" at which it accesses the object, relative to accesses by other processes (on the same timestamp object); specifically, a process is able to determine whether it accessed the object before or after some other process accessed it. We present a universal construction that ensures wait-freedom and timestamp-ignoring disjoint-access parallelism, for certain classes of data structures.

We next concentrate on important issues in achieving enhanced parallelism in STM computing. Most STM algorithms employ an *optimistic* approach, where transactions are executed speculatively, as if they will never read inconsistent data. When a conflict occurs, STM algorithms usually abort one of the transactions to ensure correctness; two concurrent transactions *conflict* if they both access the same data item and at least one of them attempts to modify it. The work performed by a transaction that aborts is discarded and it is later re-executed as a new transaction; this incurs a performance penalty. Moreover, for read-intensive workloads, it is really important to ensure that transactions that never update a data item (which are called *read-only* transactions) never abort and are *wait-free*; i.e. they always commit within a finite number of steps. For these reasons, the literature also contains *pessimistic* STM algorithms, which never abort transactions and support wait-freedom for read-only transactions. However, most of them achieve this by "pessimistically" requiring transactions that update data items (called *update* transactions) to be executed sequentially. This significantly restricts parallelism in many cases and therefore it also leads to performance degradation.

As a first step towards achieving enhanced parallelism, we introduce WFR-TM, an STM algorithm which attempts to combine some of the advantages of pessimistic and optimistic STM. In WFR-TM, as in pessimistic STMs, read-only transactions never abort and are wait-free. WFR-TM additionally ensures that read-only transactions never execute expensive synchronization instructions. In contrast to pessimistic STMs, these properties are achieved without sacrificing all parallelism between update transactions. More specifically, update transactions use a pessimistic approach to synchronize with concurrently executed read-only transactions: they wait for read-only transactions to complete. However, they use an optimistic approach to synchronize with each other: they are executed concurrently in a speculative way, and they commit if they have not encountered any conflict with other update transactions during their execution. Thus, WFR-TM achieves more parallelism than pessimistic STMs.

Finally, we introduce SemanticTM, an STM algorithm in which parallelism is achieved at the level of transactional instructions; i.e. not only the transactions themselves but also the instructions of each transaction may be executed concurrently. With compiler support, SemanticTM guarantees that simple transactions are wait-free, by ensuring that no transactions conflict. We remark that STM algorithms that never abort transactions are highly desirable since they additionally support transactions that perform irrevocable operations, e.g. I/O operations.

keywords: Asynchronous System, Shared-Memory, Concurrent Programming, Universal Construction, Software Transactional Memory (STM), Optimistic STM, Pessimistic STM, Disjoint-Access Parallelism, Wait-Freedom, Impossibility, Timestamps, Read-only Transactions, Abort-Free Transactions, Fine-Grained Parallelism

Περίληψη

Μέχρι πρόσφατα, η αύξηση της συχνότητας του επεξεργαστή αποτελούσε την κυρίαρχη τεχνική για τη βελτίωση της απόδοσής του. Ωστόσο, τα τελευταία χρόνια, φυσικοί περιορισμοί μικροηλεκτρονικής φύσης δεν επιτρέπουν την περαιτέρω αύξηση της συχνότητας του επεξεργαστή. Η συνεχόμενη ανάγκη των σύγχρονων εφαρμογών για αυξημένη επεξεργαστική ισχύ οδήγησε τους μεγαλύτερους κατασκευαστές επεξεργαστών στη σχεδίαση *πολυπύρηνων αρχιτεκτονικών*, όπου πολλαπλές επεξεργαστικές μονάδες ή *πυρήνες* εμπεριέχονται στον ίδιο επεξεργαστή.

Σε επίπεδο λογισμικού, χρησιμοποιούμε τον όρο *διεργασία* για να αναφερθούμε στη συνάρτηση που εκτελείται σε κάποιο πυρήνα και αναλαμβάνει να εκτελέσει την ακολουθία εντολών που καταφθάνουν σε αυτόν τον πυρήνα. Οι διεργασίες αυτές μοιράζονται μία *κοινή* ή *διαμοιραζόμενη μνήμη*, στην οποία διατηρούνται τα δεδομένα της εκάστοτε εφαρμογής. Επίσης, οι διεργασίες ενδέχεται να *σφάλλουν*, δηλαδή να σταματήσουν να λειτουργούν οποιαδήποτε χρονική στιγμή.

Οι εφαρμογές μπορούν να ευεργετηθούν από τις πολυπύρηνες αρχιτεκτονικές εάν ο κώδικας τους γραφτεί με τέτοιο τρόπο ώστε τμήματά τους να εκτελούνται *παράλληλα* από διαφορετικές διεργασίες. Ιδανικά η αύξηση της απόδοσης της εφαρμογής θα ήταν γραμμική ως προς το πλήθος των πυρήνων. Ωστόσο, αυτή η αύξηση επιτυγχάνεται μόνο σε εξαιρετικές περιπτώσεις εφαρμογών. Συνήθως τα διαφορετικά τμήματα μιας εφαρμογής αλληλοεξαρτώνται, επιβάλλοντας έτσι την ανάγκη επίτευξης συγχρονισμού μεταξύ των διεργασιών κατά την προσπέλαση διαμοιραζόμενων δεδομένων. Όμως είναι κοινά αποδεκτό ότι η ανάπτυξη *παράλληλων προγραμμάτων* επιτυγχάνοντας ταυτόχρονα συγχρονισμό και υψηλή απόδοση, είναι μια δύσκολη διαδικασία, η οποία γίνεται κυρίως από έμπειρους προγραμματιστές.

Έχοντας ως κύριο στόχο την απλοποίηση της διαδικασίας παράλληλου προγραμματισμού, στην παρούσα διατριβή μελετούμε δύο γενικούς μηχανισμούς που αυτοματοποιο-

ύν τη διαδικασία ανάπτυξης παράλληλων προγραμμάτων, υποστηρίζοντας την εκτέλεση οποιουδήποτε σειριακού κώδικα σε ένα πολυπύρηνο επεξεργαστή. Συγκεκριμένα μελετούμε τα *καθολικά αντικείμενα* (*universal constructions* ή *UC*) και την τεχνική *συγχρονισμού διεργασιών μέσω δοσοληψιών* (*software transactional memory* ή *STM*).

Ένα UC σύστημα αναλαμβάνει να εφαρμόσει ατομικά έναν σειριακό κώδικα στα δεδομένα της εφαρμογής που διατηρούνται στη διαμοιραζόμενη μνήμη. Από την άλλη, ένα STM σύστημα επιχειρεί να εκτελέσει ατομικά έναν σειριακό κώδικα ως μία *δοσοληψία*, η οποία μπορεί να καταλήξει είτε ως *επιτυχής*, καθιστώντας όλες τις αλλαγές που εφάρμοσε στη διαμοιραζόμενη μνήμη ορατές στις υπόλοιπες δοσοληψίες, είτε ως *μη-επιτυχής*, οπότε οι αλλαγές της αγνοούνται. Αυτή είναι και η βασική διαφορά των δύο αυτών τεχνικών.

Μια εξαιρετικά επιθυμητή ιδιότητα για τη βελτίωση του παραλληλισμού είναι η *παραλληλία αποσπασματικής προσπέλασης* (*disjoint access parallelism*, ή *DAP*). Διαισθητικά, η ιδιότητα DAP απαιτεί από διαφορετικές διεργασίες που προσπελάζουν διαφορετικά τμήματα της διαμοιραζόμενης μνήμης να μην προκαλούν παρεμβολές η μία στην άλλη, έτσι ώστε να μπορούν να εκτελεστούν ταυτόχρονα. Μία ακόμη εξαιρετικά επιθυμητή ιδιότητα είναι η ιδιότητα προόδου *ελευθερία-αναμονής*, καθώς παρέχει πεπερασμένη χρονική πολυπλοκότητα και μέγιστη ανοχή σφαλμάτων. Συγκεκριμένα κάθε διεργασία εφαρμόζει τον σειριακό κώδικα που εκτελεί σε πεπερασμένο χρονικό διάστημα, ανεξάρτητα από τις αποτυχίες ή την ταχύτητα των υπόλοιπων διεργασιών.

Ωστόσο, αποδεικνύουμε με αυστηρό τρόπο ότι είναι αδύνατο να σχεδιασθεί ένας UC ή STM αλγόριθμος που ικανοποιεί ταυτόχρονα τις ιδιότητες ελευθερία-αναμονής και DAP για οποιαδήποτε εφαρμογή. Η απόδειξή μας βασίζεται σε μία δομή δεδομένων το πλήθος των στοιχείων της οποίας μπορεί να μεγαλώσει αυθαίρετα κατά τη διάρκεια μιας εκτέλεσης. Σε αντίθεση, παρουσιάζουμε έναν UC αλγόριθμο που ικανοποιεί τις ιδιότητες ελευθερία-αναμονής και DAP για εφαρμογές που έχουν κάποιο άνω όριο στο πλήθος των δεδομένων που κάθε σειριακός κώδικάς τους μπορεί να προσπελάσει, σε οποιαδήποτε εκτέλεση.

Για να ξεπεράσουμε αυτό το αρνητικό αποτέλεσμα, προτείνουμε μία ασθενέστερη έκδοση της ιδιότητας DAP, η οποία επιτρέπει σε δύο διεργασίες να προσπελάζουν το ίδιο *αντικείμενο απόδοσης χρονοσφραγίδων*, ακόμη και εάν προσπελάζουν διαφορετικά τμήματα της διαμοιραζόμενης μνήμης. Επομένως, ενώ καταστρατηγεί μερικώς την ιδιότητα DAP, και η έκδοση αυτή βελτιώνει τον παραλληλισμό. Στη συνέχεια, παρουσιάζουμε έναν UC αλγόριθμο που ικανοποιεί αυτή την έκδοση της ιδιότητας DAP και την ιδιότητα ελευθερία-αναμονής.

Ακόμη, για την επίτευξη προόδου και τη βελτίωση του παραλληλισμού οι περισσότεροι αλγόριθμοι STM ακολουθούν μία *αισιόδοξη* τεχνική, όπου οι δοσοληψίες εκτελούνται χρησιμοποιώντας τιμές διαμοιραζόμενων δεδομένων που δεν είναι απαραίτητα συνεπής. Σε περιπτώσεις συνωστισμού κατά την προσπέλαση συγκεκριμένων τμημάτων της διαμοιραζόμενης μνήμης, ενδέχεται κάποιες ή και όλες οι δοσοληψίςε να αποτυγχάνουν, μειώνοντας έτσι την απόδοση της εφαρμογής. Από την άλλη, σε έναν *απαισιόδοξο* STM αλγόριθμο όλες οι δοσοληψίες εκτελούνται επιτυχώς. Ωστόσο, οι δοσοληψίες που τροποποιούν δεδομένα της διαμοιραζόμενης μνήμης, ή αλλιώς *δοσοληψίες ενημέρωσης*, εκτελούνται σειριακά η μία μετά την άλλη. Αυτό μειώνει σημαντικά τον παραλληλισμό σε πολλές περιπτώσεις και οδηγεί σε μείωση της απόδοσης.

Ως πρώτο βήμα για την επίτευξη τόσο προόδου όσο και παραλληλισμού στην τεχνική STM, προτείνουμε τον STM αλγόριθμο WFR-TM ο οποίος συνδυάζει πλεονεκτήματα τόσο από τους *αισιόδοξους* όσο και από τους *απαισιόδοξους* STM αλγορίθμους. Ο αλγόριθμος WFR-TM εγγυάται την ιδιότητα προόδου ελευθερίας-αναμονής για όσες δοσοληψίες δεν τροποποιούν δεδομένα της διαμοιραζόμενης μνήμης, ή αλλιώς *δοσοληψίες ανάγνωσης*, αποφεύγοντας τη σειριακή εκτέλεση των δοσοληψιών ενημέρωσης. Οι δοσοληψίες ενημέρωσης χρησιμοποιούν μία απαισιόδοξη τεχνική για να συγχρονιστούν με τις δοσοληψίες ανάγνωσης. Συγκεκριμένα, περιμένουν τις δοσοληψίες ανάγνωσης να ολοκληρωθούν. Επίσης, οι δοσοληψίες ενημέρωσης χρησιμοποιούν μία αισιόδοξη τεχνική για να συγχρονιστούν μεταξύ τους. Συγκεκριμένα, εκτελούνται ταυτόχρονα και μία δοσοληψία ενημέρωσης ολοκληρώνεται επιτυχώς μόνο εάν καμία άλλη δοσοληψία ενημέρωσης δεν προσπελάζει ταυτόχρονα το τμήμα της διαμοιραζόμενης μνήμης στο οποίο εργάζεται. Αξίζει να σημειωθεί ότι ο αλγόριθμος WFR-TM εγγυάται ότι κάθε δοσοληψία ενημέρωσης περιμένει ένα πεπερασμένο πλήθος από δοσοληψίες ανάγνωσης. Επίσης, με δεδομένο πως καμία διεργασία δε σφάλει, ο αλγόριθμος WFR-TM εγγυάται ότι σε κάθε σημείο της εκτέλεσης υπάρχει μία δοσοληψία ενημέρωσης που μπορεί να εκτελεστεί επιτυχώς σε πεπερασμένο χρονικό διάστημα.

Τέλος, προτείνουμε τον STM αλγόριθμο Semantic-TM ο οποίος εγγυάται πως όλες οι δοσοληψίες υποστηρίζουν την ιδιότητα προόδου ελευθερία-αναμονής. Ο αλγόριθμος Semantic-TM επιτυγχάνει ‘λεπτόκοκκο’ παραλληλισμό στο επίπεδο των εντολών των δοσοληψιών, το οποίο επιτρέπει τόσο την ταυτόχρονη εκτέλεση δοσοληψιών όσο και την

ταυτόχρονη εκτέλεση ανεξάρτητων μεταξύ τους εντολών της ίδιας δοσοληψίας.

Λέξεις κλειδιά: Ασύγχρονο Σύστημα, Διαμοιραζόμενη Μνήμη, Παράλληλος Προγραμματισμός, Καθολικά Αντικείμενα, Συγχρονισμός Διεργασιών Μέσω Δοσοληψιών, Παραλληλία Αποσπασματικής Προσπέλασης, Ελευθερία-Αναμονής, Αρνητικό Αποτέλεσμα, Αντικείμενο Χρονο-Σφραγίδων, Αισιόδοξος, Απαισιόδοξος Συγχρονισμός Διεργασιών Μέσω Δοσοληψιών, Δοσοληψίες-Ανάγνωσης, Δοσοληψίες Ελεύθερες Αποτυχιών, Λεπτόκοκκος Παραλληλισμός

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Up until recently, increasing CPU speed was the dominant approach for improving computer performance. However, the last few years, due to physical constraints the size of electronic circuits cannot become smaller; so CPU speed is no longer rising. The continuous demand for more efficient computing has led hardware designers to move towards *multi-core architectures*.

To take advantage of the increased computational power of multi-core machines, *concurrency* should be employed. Introducing as much concurrency as possible into software applications has become urgent. Remarkably, most of the current applications are sequential. To execute such an application concurrently its code segments that can be executed in parallel should first be identified. Then, concurrent algorithms should be provided to allow each processes to execute any of these segments concurrently with (and ideally, independently of) the other processes.

Ideally, the speed-up from parallelization would be linear in the number of cores. However, this speed-up is rarely achieved due to dependencies that exist and synchronization that is needed between the different code segments. Because of these complications, it is commonly accepted that writing concurrent programs is extremely hard work, currently undertaken only by experts. The difficulty is inherent in achieving communication and synchronization between processes that run concurrently. Despite the difficulty of concurrent programming, all modern applications must employ concurrency in order to achieve high performance.

To simplify concurrent programming we need mechanisms to automatically execute sequential code segments in a concurrent environment. Two well-established such mechanisms are universal constructions and software transactional memory. They both have the same goal of simplifying concurrent programming by providing mechanisms to efficiently execute sequential code in a concurrent environment.

A *universal construction* [1, 2] provides a concurrent implementation of any sequential data structure. Pieces of sequential code can be thought of as operations of a data structure. Thus, universal constructions can be used to execute any piece of sequential code (which may require synchronization) in a concurrent environment. *Software transactional memory* (STM) [3, 4] is a mechanism that allows a programmer of a sequential program to identify as *transactions* those pieces of the sequential code that require synchronization. Thus, a transaction includes a sequence of instructions on pieces of the simulated state, known as *data items*. When the transaction is being executed in a concurrent environment, data items can be accessed by several processes simultaneously. Thus, synchronization is needed when accessing data items. If the transaction *commits*, all its changes take effect as if they all are applied sequentially at a given point in time during the execution of the transaction. Otherwise, the transaction *aborts* and none of its changes take effect.

In STM, when a transaction is aborted, the STM algorithm 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.

A universal construction or STM system is implemented by an expert programmer who addresses all problems encountered when concurrency is employed. The resulting algorithms are guaranteed to be correct, and have specific progress and performance properties. The naive programmer simply provides sequential code and it is the universal construction or the STM system that undertakes the task to execute it concurrently. Thus, if universal constructions or STMs achieve enhanced parallelism, and therefore gain performance, they do so at no cost to the naive programmer.

1.2 Focus

In this thesis, we study universal constructions and STMs from the perspective of achieving increased concurrency, without sacrificing correctness and progress. Specifically, the correctness property we consider for universal constructions is *linearizability* [5] and for STM systems it is *opacity* [6]. Roughly speaking, linearizability says that every completed operation (and some of the non-completed ones) appears as if it has been executed sequentially at some point within its execution interval. In addition to guaranteeing this condition for committed (and some *commit-pending*) transactions, opacity ensures that even transactions that do not commit see “consistent” simulated states.

To ensure fault-tolerance, we study implementations that ensure strong progress guarantees, namely wait-freedom and local progress. A universal construction 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. We remark that it is common behavior of an STM algorithm to restart an aborted transaction until it eventually commits. A meaningful progress condition [7, 8] in STM requires that the number of times each transaction aborts is finite. This property, known as *local progress* [7], is similar to wait-freedom.

We focus on established techniques to enhance parallelism; i.e. techniques that have received attention in previous research. One such technique is to ensure that an algorithm satisfies a property known as disjoint-access parallelism. Roughly speaking, *disjoint-access parallelism* guarantees that if two processes operate on disjoint parts of the simulated state, they do not access any common shared objects, so they do not interfere with one another. Therefore, disjoint-access parallelism allows unrelated operations to progress in parallel. This property has been extensively studied in the literature in the context of both universal constructions [9, 10] and STM [7, 11, 12, 13, 14, 15, 16, 17, 18].

Speculation is the main technique for achieving enhanced concurrency in the STM context. Specifically, processes *optimistically* execute instructions, using values that are not guaranteed to be consistent and taking corrective actions whenever they discover that these values are inconsistent. In cases where such corrective actions are rarely needed, e.g. under low contention, such a strategy may offer significant performance gains. The potential disadvantage of having processes perform unnecessary tasks is usually greatly outweighed by the enhanced concurrency it allows.

Most STM algorithms are optimistic: they execute transactions speculatively and they may proactively abort transactions if they “suspect" that their execution may jeopardize correctness. Specifically, when a conflict between two transactions occurs, STM algorithms usually abort one of the transactions to ensure consistency. Two concurrent transactions *conflict* if they both access the same data item and at least one of them attempts to modify it. Unfortunately, this proactive behavior often leads to a big number of *spurious* aborts, i.e. transactions are aborted even in cases where they could commit without violating consistency. Research on STM has given special attention to this issue [12, 19, 20, 21, 22, 23], as it degrades performance. The work performed by a transaction that aborts is discarded and it is later re-executed as a new transaction; this incurs a performance penalty. So, the nature of STM is optimistic: if transactions never abort then no work is ever discarded.

It is highly desirable that all transactions eventually commit. However, this property is not ensured by the currently available STM systems, which in their majority lead to many transaction aborts. STM algorithms that never abort transactions are highly desirable since they additionally support transactions that perform irrevocable operations, e.g. I/O operations. Ideally, we would like to have STM systems in which all transactions terminate successfully within a finite number of steps, i.e. ensure local progress. However, Bushkov *et al.* [7] proved that no opaque STM algorithm can achieve this property.

In terms of achieving good performance, the system should additionally guarantee that parallelism is achieved. So, transactions should not be executed sequentially. A pessimistic STM algorithm [24, 25] never aborts any transaction. However, the way most existing pessimistic STM algorithms achieve this is by "pessimistically" imposing a sequential order in the execution of all *update transactions*, i.e. all those transactions that update data items. This significantly restricts parallelism in many cases and therefore it leads to performance degradation.

1.3 Contribution

In this thesis, we prove a collection of results about disjoint-access parallel universal constructions, including two new definitions for disjoint-access parallelism, an impossibility result, and two algorithms. We also present two new STM algorithms, namely WFR-TM and SemanticTM, with the goal of improving concurrency, while not sacrificing correctness and progress.

1.3.1 Disjoint-Access Parallelism in Shared-Memory Computing

We prove a collection of positive and negative results. On the negative side, we prove that linearizable universal constructions which ensure both disjoint access parallelism and wait-freedom are not possible; this impossibility result applies to STM algorithms that satisfy local progress. We prove this impossibility result by considering a dynamic data structure that can grow arbitrarily large during an execution. The proof considers a singly-linked unsorted list of integers which supports an operation that appends an element to the end of the list and an operation that searches the list starting from its beginning. It shows that, in any disjoint-access parallel implementation resulting from the application of a universal construction to this data structure, there is an execution of a search that never terminates. For the proof of the impossibility result, we introduce *feeble disjoint-access parallelism*, which is weaker than all existing disjoint-access parallelism definitions. Thus, the impossibility result still holds if we replace the disjoint-access parallelism definition with any existing definition of disjoint-access parallelism. This result relies on a natural assumption about universal construction, which roughly says that the operations of a concurrent implementation resulting from applying a universal construction to a sequential data structure should simulate its operations.

For data structures of bounded size we present a universal construction, called DAP-UC, that achieves both disjoint-access parallelism and wait-freedom. Specifically, the universal construction 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, the universal construction still ensures linearizability and disjoint-access parallelism, but a weaker progress property, know as *lock-freedom*. Lock-freedom guarantees that, in an infinite execution, some (non-faulty) process completes infinitely many operations, even if other processes may fail (by crashing) or are very slow.

Disjoint-access parallelism [27] and the variants of it [11, 9, 26] presented in the literature 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.

In this thesis, we also study reasonable relaxations of the definition of disjoint-access parallelism. Specifically, we define a variant of disjoint-access parallelism, called *timestamp-ignoring disjoint-access parallelism*, which is similar to classical disjoint-access parallelism [11] but allows multiple operations to access a timestamp object, even though they operate on disjoint parts of the simulated state. A wait-free timestamp object can be easily implemented with a $fetch\&increment$ object or a shared global clock. If the `getTimestamp`() operation never attempts to modify the timestamp object, for example, when it is implemented from a shared global clock that increments automatically, then timestamp-ignoring disjoint-access parallelism is (in many cases, depending on the formal definition of disjoint-access parallelism) the same as disjoint-access parallelism.

Several examples of algorithms that ensure timestamp-ignoring disjoint-access parallelism can be found in the literature. For instance, several well-known STM algorithms [29, 30, 31] assign timestamps to transactions. Each transaction may then use its timestamp (as well as the timestamps of other transactions) to resolve conflicts and/or determine whether the data items it has read are consistent. If the access to the global timestamp object is not taken into consideration, some of these algorithms are disjoint access parallel (e.g. [29], [30] and [32]). However, none of these algorithms are wait-free. The definition of timestamp-ignoring disjoint-access parallelism can be motivated by the existence of these algorithms. This definition allows operations operating on different parts of the simulated data structure to proceed in parallel without any interference, except for accesses to the timestamp object.

Finally, we present a new universal construction, called TI-DAP-UC, that ensures wait-freedom and timestamp-ignoring disjoint-access parallelism when applied to any sequential data structure that has a bounded number of entry points; an *entry point* to a data structure is any data item passed as input to an operation on the data structure. For instance, in the linked-list example, the entry points are the pointer to the (first) node from which search starts and the pointer to the (last) node on which append is applied.

1.3.2 New Software Transactional Memory Algorithms

1.3.2.1 WFR-TM

For read-intensive workloads it is really important to ensure that transactions that never update a data item (which are called *read-only* transactions) never abort and are wait-free; i.e. they always commit within a finite number of steps. As a first step towards achieving enhanced parallelism in the STM context, we introduce WFR-TM, an STM algorithm which aims at combining some of the advantages from both optimistic and pessimistic STM, while trying to avoid their drawbacks. Specifically, in WFR-TM read-only transactions are wait-free, and they perform only two writes to shared memory and these writes are to single-writer registers. Additionally, WFR-TM allows multiple *update* transactions (that are not read-only) to execute in parallel. Specifically, update transactions use a pessimistic approach to synchronize with concurrently executed read-only transactions: they wait for such transactions to complete. However, they use an optimistic approach to synchronize with each other: they are executed concurrently in a speculative way, and they commit if they have not encountered any conflict with other update transactions during their execution. Thus, WFR-TM, in contrast to pessimistic STM algorithms, imposes less restrictions on parallelism.

Briefly, in WFR-TM, a read-only transaction T_r starts by announcing itself. An update transaction T_w that wants to update a data item x after T_r is announced (and thus probably after T_r has read x), does so only after T_r has committed. So, before T_w completes, it waits for all read-only transactions that have been initiated and not yet completed at some point of T_w's execution (before T_w began its waiting phase), to commit. Notice that T_w may wait a read-only transaction with which it does not conflict. Also, notice that an update transaction waits only for a finite number of read-only transactions. We remark that it is not necessary to know in advance whether a transaction is read-only; any transaction is read-only when it begins and becomes an update transaction the first time it accesses a data item for write. Update transactions in WFR-TM employ fine-grained locking for accessing data items, so that those that do not conflict can commit in parallel. We remark that, in WFR-TM, read-only transactions are able to read a consistent value even for a locked data item, without having to wait for its owner to unlock it. Thus, wait-freedom of read-only transactions is not violated in case an update transaction fails while holding a lock on some data item.

1.3.2.2 SemanticTM

Finally, we introduce SemanticTM, an STM algorithm which achieves fine-grain parallelism at the transactional instruction level. This means that in addition to instructions of different transactions, instructions of the same transaction that do not depend on each other can be executed concurrently. Additionally, for simple transactions and assuming compiler support, SemanticTM ensures that all transactions (both read-only and update) complete within a finite number of steps and never abort, by ensuring that no transactions conflict. Since SemanticTM ensures local-progress, it naturally supports irrevocable operations.

Briefly, SemanticTM employs a list for each data item. The instructions of each transaction are placed in the appropriate lists in FIFO order; specifically, since each instruction is executed on a single data item, it is placed in the list of the data item that it accesses. A set of worker processes execute instructions from the lists, in order. The algorithm is highly fault-tolerant; even if some worker processes fail by crashing, all transactions whose instructions have been placed in the lists will be executed. In this thesis we focus on relatively simple transactions that access a known set of data items, and their codes contain `read` and `write` instructions on them, conditionals (i.e. `if`, `else if`, and `else`), loops (i.e. `for`, `while`, etc.), and function calls. For such transactions, the work of placing the instructions of the transaction together with their dependencies in lists can be done at compile time (so there is no need to employ a scheduling component for doing so). Despite this fact, for simplicity, we refer to a scheduling process (sometimes called scheduler) which undertakes this task. We briefly discuss, in Section 7.2, how to extend SemanticTM to cope with more complicated transactional codes.

We remark that several dependencies may exist among the instructions of a single transaction; specifically, a single instruction may have several dependencies. SemanticTM requires these dependencies to be predicted statically. By using compiler support, these dependencies become known before the beginning of the execution of the transactions. SemanticTM stores information about them together with the corresponding instruction in the appropriate list. In Section 6.2.1, we describe the dependencies expected by SemanticTM in order to guarantee the correct execution of the corresponding transactions. It is worth mentioning that in this work, we do not focus on how these dependencies are extracted. SemanticTM can make use of any existing or future work on dataflow analysis. After its placement in the appropriate list, each transactional instruction is executed as soon as its data are available. Thus, SemanticTM can be thought of as a dataflow algorithm in the sense that it mimics, in software, a dataflow architecture.

In Section 6.5, we present some experimental results where a simplified version of SemanticTM executes simple static transactions testing different conflict patterns among them. In the experiments, SemanticTM exhibits good performance; specifically, in all these experiments, SemanticTM performs better than GccSTM [33] which is an industry software transactional memory standard.

The current version of SemanticTM does not support dynamic transactions. A discussion on how this limitation could be overcomed is provided in Section 7.2. Since SemanticTM ensures that all transactions will commit, it does not provide any support for explicitly aborting transactions.

1.4 Roadmap

In Chapter 2, we present the model of computation, provide several useful definitions for universal constructions and STM algorithms, describe correctness and progress properties, and some variants of disjoint-access parallelism, including the two new definitions introduced. In Chapter 3, we provide a discussion of the related work.

In Chapter 4, we present the results on disjoint-access parallelism. Specifically, the impossibility result is presented in Section 4.2. Then, in Section 4.3, the new universal construction called DAP-UC is presented and in Section 4.4, we formally prove that (for certain classes of data structures) the concurrent data structures that DAP-UC produces are linearizable, wait-free, and disjoint-access parallel. Finally, in Section 4.5, we present an extension of DAP-UC which (for certain classes of data structures) produces linearizable, wait-free, and timestamp-ignoring disjoint-access parallel concurrent data structures.

In Chapter 5, we present WFR-TM and prove that it ensures opacity, wait-freedom for read-only transactions, and deadlock-freedom for update transactions. In Chapter 6, we present SemanticTM. Finally, in Chapter 7, we describe possible extensions of the work performed in this thesis and open research problems.

Chapter 2

Model

2.1 Abstract Data Types

An *abstract data type* specifies a set of objects and a set of functions that can be computed on these objects. As an example, consider an abstract data type whose objects are integers and finite sets of integers. Then, answering the question "$v \in S$" where v is an integer and S is a finite set of integers is one of the operations of the abstract data type. Another example is add, which adds the element v to the set S.

We remark that several abstract data types can be combined into a single one that supports all their functions. Each function has a set of parameters that are determined each time the function is *invoked*, and a set of possible *responses*. Then, an *instance* of the function is a pair consisting of an invocation and a response. The *sequential specification* of an abstract data type is a set of sequences of function instances. The sequences that belong to the sequential specification are called *legal*.

2.2 Sequential Data Structures

A *sequential data structure* is a sequential implementation of an abstract data type. In particular, it provides a representation for each of the data objects specified by the abstract data type and sequential code for each of the functions it supports. This sequential code constitutes the *operations* of the sequential data structure. An *instance* of an operation of a sequential data structure begins with the call of the code, which is called its *invocation*, and ends with the return of this code, which is called its *response*. A sequence of operation instances of a sequential data structure is *legal* if it belongs to the sequential specification of the corresponding abstract data type. A sequential data structure is *correct* if each of the sequences of operation instances it produces is legal. Throughout this thesis, we consider only correct sequential data structures. We remark that given a finite set of pieces of sequential code, we treat each of them as a single operation of the same sequential data structure.

A *data item* is a piece of the representation of an object implemented by the sequential data structure. For clarity, the data items in the sequential code are accessed via the *instructions* CREATEDI, READDI, and WRITEDI, which create a new data item and return a pointer to it, read the data item and return its value, and write to the data item and return acknowledgement, respectively.

As an example of a sequential implementation of the abstract data type presented in

Section 2.1, we will consider an unsorted singly-linked list of integers that supports two operations: APPEND(L, v), which appends the element v to the end of the list L, and SEARCH(L, v), which searches the list L for v starting from the first element of the list. The data items are the nodes of the singly-linked list and the pointers $L.start$ and $L.end$ that point to the first and the last element of the list, respectively. Notice that both APPEND and SEARCH take as input the two data items $L.start$ and $L.end$, as well as a *value parameter* v. Figure 2.1 presents the pseudo-code of this data structure, where the first parameter of READDI and WRITEDI is a pointer to the data item read or written, respectively.

```
type node
    int $key$
    ptr to node $next$

type list
    ptr to node $start$
    ptr to node $end$

APPEND(list $L$, int $v$)
    $new$ := CREATEDI(node)
    WRITEDI($new$, $\langle v, \text{null}\rangle$)

    $e$ := READDI(&$L.end$)
    if ($e \neq$ null) then
        $\langle k, -\rangle$ := READDI($e$)
        WRITEDI($e$, $\langle k, new\rangle$)
    else WRITEDI(&$L.start$, $new$)

    WRITEDI(&$L.end$, $new$)

boolean SEARCH(list $L$, int $v$)
    $s$ := READDI(&$L.start$)
    if ($s$ = null) then return false

    $\langle k, s\rangle$ := READDI($s$)
    while ($k \neq v$ and $s \neq$ null)
        $\langle k, s\rangle$ := READDI($s$)
    if ($k = v$) then return true
    else return false
```

Figure 2.1: Sequential Implementation of a Singly-Linked List Supporting APPEND and SEARCH

The *state* of a sequential data structure consists of the collection of data items 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 $L.start$ and $L.end$ are static data items. When the sequential data structure is dynamic, the data items accessed by (an instance of) an operation (in a sequential execution) may depend on the operations that have been performed before it. In our 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 *entry point* to a sequential data structure is any data item passed as input to an instance of an operation of this data structure. In our example, the entry points of SEARCH and APPEND are $L.start$ and $L.end$. In general, different instances of the same operation can have different entry points. For example, an operation of some sequential data structure may return one of the data items it accesses (or creates). Then, this data item can be passed as an input parameter to subsequent operation instances.

An operation of a data structure is *value oblivious* if the set of data items accessed by any instance of that operation in any (sequential) execution, does not depend on the values of its value input parameters. In our example, the data items accessed by APPEND do not depend on the value of the parameter v. Thus, APPEND is a value oblivious operation. Notice that this is not the case for SEARCH.

2.3 Model of Computation

We consider an *asynchronous shared-memory* system with n processes $p_1, \ldots, p_n$ that communicate by accessing shared *base objects*; these are simple objects usually provided by the hardware. In an asynchronous system there are no restrictions on the computation speed of processes. Each of the base objects we consider in this thesis has a *value* and supports a set of atomic *primitives* to access and modify its value.

The simplest base object is a *read-write* (`R/W`) *object* O that stores a value from some set and supports primitives `read` and `write`; `read`(O) returns the value of O, and `write`(O, v) sets the value of O to v and returns an acknowledgement. We also consider `CAS` and `LL/SC` base objects. A `CAS` *object* O stores a value from some set and supports additionally to `read`, the primitive `CAS`. When applying `CAS`(O, u, v) a process checks whether the value of O is u and, if so, it sets the value of O to v. If the update occurs, `true` is returned and we say that the `CAS` is successful; otherwise, the value of O does not change and `false` is returned. An `LL/SC` *object* O stores a value from some set and supports the primitives `LL`, `VL`, and `SC`. `LL`(O) returns the current value of O. By applying `SC`(O, v), a process p attempts to set the value of O to v. This update occurs only if no process has set the value of O since p last applied `LL`(O). If the update occurs, `true` is returned and we say that the `SC` is successful; otherwise, the value of O does not change and `false` is returned. By applying VL(O), a process p checks whether any process has set the value of O since p last applied LL(O). If so, false is returned and we say that the VL is unsuccessful. If not, true is returned and we say that the VL is successful. Note that LL, VL, and SC can be implemented from CAS, read, and write, so that each primitive has $O(1)$ worst case step complexity [34].

A primitive is *non-trivial* if it may change the value of a base object; otherwise, the primitive is *trivial*. For instance, the primitive write of a read-write object is non-trivial, whereas its read primitive is trivial.

A *(static) timestamp object* supports one primitive: getTimestamp(), which returns a timestamp from a universe U with a binary relation $<$ such that $t < t'$ if an instance of getTimestamp() that returned t was executed before an instance of getTimestamp() that returned t'.

2.4 Concurrent Data Structures

A *concurrent data structure* is an implementation of an abstract data type in an asynchronous shared memory system. In particular, it provides a representation from base objects for each of the objects specified by the abstract data type and an algorithm, for each process, to perform each of the functions supported by the abstract data type.

A *configuration* provides a global view of the system at some point in time. In an *initial configuration*, each process is in an initial state and each base object has an 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 interval* 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 and steps, where the application of step ϕ_k to configuration C_k results in configuration C_{k+1}, for each $i \leq k < j$. An *execution* is an execution interval starting from an initial configuration. An execution (interval) α is *indistinguishable* from another execution (interval) 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 (interval) is *solo* if all the steps in it are taken by the same process. Processes may *crash* during an execution, in which case they take no more steps.

In case some configuration C occurs before some other configuration C' (or step ϕ) in α, we say that C *precedes* C' (or ϕ) in α and we denote this by $C < C'$ (or $C < \phi$). We define what it means for a step ϕ to *precede* a configuration C or a step ϕ' (denoted by $\phi < C$ or $\phi < \phi'$) in a similar way. We also define similarly what it means for a configuration or a step to *follow* another configuration or step, denoted by $>$ (instead of $<$). We say that an execution interval a *precedes* an execution interval a' and we denote this by $a < a'$, if the last configuration of a either precedes or it is the same with the first configuration of a'; also we say that a' *follows* a and we denote this by $a' > a$.

2.5 Transforming a Sequential Data Structure to a Concurrent Data Structure

We consider below two general mechanisms to produce a concurrent data structure based on its sequential implementation. Recall that this gives a mechanism to automatically execute pieces of sequential code in an asynchronous shared-memory system.

2.5.1 Universal Constructions

A *universal construction* provides a single method, called PERFORM, which takes as parameters an operation and a list of input arguments for it, executes an instance of this operation, and responds with a list of return values. The list of return values of PERFORM is considered to be the response of the corresponding operation instance. The algorithm that implements PERFORM applies a sequence of primitives on shared base objects. Given an execution α, the *execution interval* of an operation instance that responds in α starts with the configuration preceding the first step of the corresponding call to PERFORM and terminates with the configuration following its last step. If an operation instance does not respond in α, then its execution interval is the suffix of α that starts with the configuration preceding the first step of the corresponding call to PERFORM. An operation instance is *completed* if the corresponding call of PERFORM returns; otherwise, it is *active*.

2.5.2 Software Transactional Memory

A *software transactional memory* (STM) algorithm supports the execution of *transactions*. A transaction is an *attempt* of the STM to execute an operation instance in an asynchronous shared-memory system; this attempt may fail due to synchronization problems with other transactions that are concurrently executed. To execute the instructions (i.e. accesses on data items) of some operation instance, STM supports methods CREATEDI, READDI, and WRITEDI. Additionally, it provides methods BEGINTX and COMMITTX, used to identify the beginning of a transaction and request its termination as committed, respectively. In this thesis, we call these routines *t-methods*. If the call to COMMITTX for some transaction responds with `true`, then all the changes to data items performed by the transaction take effect, and we say that the transaction *commits*, or this attempt *succeeds*. Otherwise, if any call to READDI, WRITEDI, or COMMITTX for some transaction responds with some special value indicating that the method was not completed successfully, then none of its intended updates are realized and we say that the transaction *aborts*, or this attempt *fails*.

An STM system operates as an interpreter of the code of each operation by invoking the t-methods included in the code and receiving their responses.

A transaction is a single attempt to apply some operation instance. This attempt may fail. On the other hand, in a universal construction a call to PERFORM returns only after the operation instance is successfully applied. This is the main difference between the two paradigms. An aborted transaction is usually restarted until it eventually commits. Throughout this thesis, we assume that this is the case. So, in any execution of an STM, an operation instance is related to at most one committed transaction and possibly with several aborted transactions.

To implement each t-method, the algorithm applies a sequence of primitives on shared base objects. We assume that the implementation of each t-method does not contain invocations of other t-methods. Consider an execution α. A transaction that either commits or aborts in α is *completed*; otherwise, it is *live* in α. A live transaction that called COMMITTX is called *commit-pending*. The *execution interval* of a completed transaction in α starts with the configuration preceding the first step of the corresponding call to BEGINTX and it terminates with the configuration following the last step of the last t-method executed for this transaction. If a transaction does not complete in α, then its execution interval is the suffix of α that starts with the configuration preceding the first step of the corresponding call to BEGINTX. If the execution interval of some transaction T' ends before the execution interval of some transaction T starts, then T' *precedes* T. The *read-set* (*write-set*) of a transaction contains the data items accessed through READDI (WRITEDI) during its execution interval. A transaction with an empty write-set is called *read-only*; otherwise, it is an *update* transaction. When the read-set and the write-set of a transaction are known a priori before its initiation, we say that the transaction is *static*; otherwise, the transaction is *dynamic*. The *execution interval* of an operation instance that completed in α starts with the configuration preceding the first step of the call to BEGINTX from the first transaction associated with this operation instance and ends when the execution interval of the last transaction associated with this operation instance ends. If an operation instance does not complete in α, then its execution interval is the suffix of α that starts with the configuration preceding the first step of the call to BEGINTX from the first transaction associated with this operation instance. An operation instance is *completed* in α if the last transaction associated with this operation is committed in α; otherwise, it is *active*. If an operation instance *op* completes, then the return values for *op* is considered to be the response of *op*.

2.5.3 Common Definitions

Consider a concurrent data structure which results from either a universal construction or an STM applied on some sequential data structure and let α be any execution of this concurrent data structure. From this point on, for simplicity, we use the term operation to refer to an instance of an operation executed by some process in α. A process is *executing* an operation *op* in α while it is executing PERFORM for *op* (in case of a universal construction) or after it has invoked the first t-method in the code of *op* and until the last invoked t-method in this code returns (in case of an STM), in α. Consider an operation *op* that is executed by some process p in α; we say that *op* is *contained* in α. When p applies a primitive to some base object while executing *op* in α, we sometimes say that *op applies* the primitive to this base object in α.

Two execution intervals in α *overlap* if there is a configuration that is contained in both intervals. Two operations (or transactions) *overlap* or they are *concurrent*, if their execution intervals overlap. An operation *op* is executed solo in its execution interval is solo. A process can have at most one active operation. A configuration is *quiescent* if no operation is active in that configuration.

Two concurrent operations *conflict* if they both access the same data item and at least one of them writes it.

2.6 Correctness

Definition 1. (**Linearizability** *[5]*). *An execution α of a concurrent data structure is* linearizable *if its completed operations and possibly some of its uncompleted operations* *each has a* linearization point *within its execution interval such that the response returned by each of these operations is the same as the response it would return if all these operations were executed sequentially in the order of their linearization points. The sequence of these linearization points is called a* linearization *of α.*

A concurrent data structure is *linearizable* if all its executions are linearizable. A universal construction or an STM is *linearizable* if all concurrent data structures resulting from it are linearizable.

Notice that linearizability (Definition 1) can also be applied to STM. A transaction is not necessarily an attempt to execute a single operation but it may be an attempt to execute a *composite operation* containing a sequence of several operations. Then, linearizability for STM is stated as follows.

Definition 2. (**Linearizability for STM**). *An execution α of an* STM *is* linearizable *if its completed composite operations and possibly some of its uncompleted composite operations each has a* linearization point *within its execution interval such that, for any data structure $\mathcal{D}$, the response returned by each of the operations of $\mathcal{D}$ executed during these composite operations, is the same as the response it would return if all these operations were executed sequentially in the order of the linearization points of the corresponding composite operations (containing each of them).*

We remark that, if transactions are restricted to execute only one operation, Definition 2 is the same as Definition 1. Moreover, if we concentrate on the transactions executed during a composite operation, then linearizability, which is then called *strict serializability*, is stated as follows.

Definition 3. (**Strict Serializability**). *An execution α of an* STM *is* strictly serializable *if its committed transactions and possibly some of its commit-pending transactions each has a* serialization point *within its execution interval such that, for any data structure $\mathcal{D}$, the response returned by each of the operations of $\mathcal{D}$ executed during these transactions, is the same as the response it would return if all these operations were executed sequentially in the order of the serialization points of the corresponding transactions (containing each of them).*

We remark that *strict serializability* introduced for executions of database transactions [35], is similar to Definition 3, but it restricts only to committed transactions. Additionally, in strict serializability [35], transactions may apply operations from multiple instances of a single sequential data structure, which provides functionality similar with a read-write object.

Notice that Definitions 3 and strict serializability do not impose any restrictions on live transactions. We remark that a live transaction may cause an exception or enter into an infinite loop after reading inconsistent values of different data items. In order to avoid such undesirable situations, a well known correctness property for STM, called *opacity*, has been proposed in [6].

Definition 4. (**Opacity**). *An execution α of an* STM *is* opaque *if it is strictly serializable and the following holds. For each live or aborted transaction T, let T' be a transaction such that among all transactions that precede T in α and exist in the serialization order T' is the one that comes last in the serialization order. Consider all the serialized transactions up to T'. Then, the responses returned by the operations of $\mathcal{D}$ executed during these transactions, followed by the responses of operations of $\mathcal{D}$ applied by T, are the same as the responses they would return if all these operations were executed sequentially in the order of the serialization points of the corresponding transactions (containing each of them).*

It is worth pointing out that, considering an STM system that ensures either Definition 2 or strict-serializability, exceptions (and similar errors) can be avoided if we assume that an exception is a response of the corresponding composite operation. Then, such an STM system will never produce exceptions. Moreover, undesired situations where a transaction enters an infinite loop will not appear in STM systems that ensure standard progress properties like obstruction-freedom or deadlock-freedom, as defined in Section 2.7.

2.7 Progress

A concurrent data structure is called *blocking* if it produces executions in which processes may *block*. A process blocks if in order to make progress it requires that other processes take steps. If in an execution of a blocking data structure some process crashes, we remark that it is possible all other (non-crashed) processes to block in this execution. To enhance fault tolerance, ensuring a *non-blocking* property [2, 36] is desirable. The most well known such properties are wait-freedom [2], lock-freedom [2], and obstruction-freedom [36].

A concurrent data structure is *wait-free* if, in every execution, each process that does not crash completes each operation it performs within a finite number of its own steps. It is *lock-free* if, in every execution α, starting from any configuration C of α some process that does not crash completes the operation it is executing at C (or a newly initiated operation after C, if it is executing no operation at C) within a finite number of steps. Notice that in every infinite execution of a lock-free concurrent data structure, some process completes infinitely many operations. A concurrent data structure is *obstruction-free* if, in every execution α, each process that does not crash and executes solo from any configuration, completes its operation (or a newly initiated operation) within a finite number of steps. We remark that a wait-free concurrent data structure ensures the strongest progress property. More specifically, a wait-free concurrent data structure is also lock-free and a lock-free concurrent data structure is also obstruction-free. Consider an execution α of some concurrent data structure. If this concurrent data structure is not wait-free, some process may not complete its operation within a finite number of steps, i.e. it may experience *starvation*, in α. If a concurrent data structure is neither wait-free nor lock-free, all processes may experience starvation, i.e. they may experience a *livelock*, in α.

Consider now an execution α of a blocking concurrent data structure. We say that a set of processes experiences *deadlock* in α, if there is a configuration C in α in which each of these processes has an active operation in α, yet all of these processes are blocked in α. A blocking concurrent data structure satisfies *deadlock-freedom* if, in every execution in which no process crashes, not all processes experience deadlock. Notice that if a concurrent data structure is deadlock-free then in the absence of crashes it is lock-free. Moreover, a concurrent data structure satisfies *starvation-freedom*, if in the absence of crashes, it satisfies wait-freedom.

A universal construction or an STM is blocking, if at least one concurrent data structure resulting from it is blocking. A universal construction or an STM satisfies a progress property if all concurrent data structures resulting from it satisfy this property.

Meaningful progress conditions [7, 8] in STM require that the number of attempts before each operation completes, is finite. This property, which is called *local progress* [7], is similar to wait-freedom. In [7], a property similar to lock-freedom, called *global progress*, is defined for STM. Global progress requires that in an infinite execution infinitely many attempts succeed. Also, a property similar to the obstruction-freedom, called *solo progress*, is defined for STM in [7]. Solo progress requires that any operation that is executed solo[1]

[1]In [7] an STM model similar to the one of [6] is used and a process is considered to run solo when it is not executed concurrently with commit-pending transactions; notice that in [7], a process runs solo not only when it is executed concurrently with no other processes, but also when it is executed concurrently with live but not

completes within a finite number of attempts.

2.8 Data Set of an Operation

For any operation instance *op* and any state S of the sequential data structure, let $DS(op, S)$, the *data set* of *op* starting from state S, be the set of all data items accessed during the sequential execution of *op* starting from S, i.e. to which CREATEDI, READDI, or WRITEDI is applied.

Consider any linearizable execution α of a concurrent data structure and fix some linearization $op_1, op_2, \ldots$ of it. Let S_0 denote the initial state of the sequential data structure and, for $i \geq 1$, let $S_i = S_0 op_1 \cdots op_i$ denote the state of the sequential data structure that results from applying the first i operations linearized in α sequentially, in order, starting from S_0. For any configuration C of α, let S_C denote the state of the sequential data structure that results from applying each operation linearized in α prior to C sequentially, in order, starting from S_0. In other words, if there are i operations linearized before C, then $S_C = S_i$.

If op_i is concurrent with no other operations, then the *data set* of op_i in α is $DS(op_i, \alpha) = DS(op_i, S_{i-1})$. However, if op_i is concurrent with other operations, it may have had some failed attempts and the definition of $DS(op_i, \alpha)$ should also include data items accessed in those attempts. So, let j be the largest index of any operation that finished in α before op_i began, or 0, if no operation finished in α before op_i began. Define $DS(op_i, \alpha) = \bigcup_{k=j}^{i-1} DS(op_i, S_k)$. Then $DS(op_i, \alpha)$ is the union of the sets of all data items accessed during the sequential executions of op_i starting from S_k, for $j \leq k < i$.

2.9 Disjoint-Access Parallelism

Informally, a concurrent data structure is disjoint-access parallel if operations on different parts of the data structure do not interfere with one another. A universal construction or an STM is disjoint-access parallel if all concurrent data structures resulting from it are disjoint-access parallel. There are many different ways to make the definition of disjoint-access parallelism more precise, but they depend on the following concepts.

commit-pending transactions.

Consider any linearizable execution α of a concurrent data structure and fix some linearization $\mathcal{L}$ of it. The *shared-access graph*[2] of an execution interval I of α for $\mathcal{L}$ 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 $DS(op, \alpha) \cap DS(op', \alpha) \neq \emptyset$.

Two operations *contend* on a base object b in α if they both apply a primitive to b and at least one of these primitives is non-trivial. They *concurrently contend* if there is some configuration in which the next steps of both operations access the same base object and at least one of these steps applies a non-trivial primitive to this base object.

We present three versions of disjoint-access parallelism. The first, *feeble disjoint-access parallelism*, is extremely weak. In fact, it is satisfied by any concurrent data structure that satisfies any of the existing definitions of disjoint-access parallelism [27, 11, 26, 37]. Thus, our impossibility result also holds for those definitions. We employ this weak version of disjoint-access parallelism to prove our impossibility result. This makes the impossibility result stronger.

Definition 5. (**Feeble Disjoint-Access Parallelism**). *A concurrent data structure is* feebly disjoint-access parallel *if, for every reachable quiescent configuration C and every two solo execution intervals, α_1 and α_2, of operations, op_1 and op_2, starting from C that contend on some base object, there is a data item accessed by both op_1 starting from S_C and op_2 starting from S_C, i.e. $DS(op_1, S_C) \cap DS(op_2, S_C)) \neq \phi$.*

We continue to present a much stronger version of disjoint-access parallelism which is satisfied by our universal construction presented in Section 4.3.

Definition 6. (**Disjoint-Access Parallelism**). *A concurrent data structure is* disjoint-access parallel *if, for every execution α containing two operations op_1 and op_2 that contend on some base object, there exists a linearization of α such that there is a path between op_1 and op_2 in the shared-access graph of the minimal execution interval containing op_1 and op_2 for this linearization.*

Finally, in order to overcome our impossibility result, we introduce *timestamp-ignoring* disjoint-access parallelism. It is similar to disjoint-access parallelism (Definition 6), but allows multiple operations to access a static timestamp object, even though the sets of data items that the operations access do not intersect.

[2]In the literature this graph is also called *conflict graph*. To avoid confusion with the definition of conflicting transactions we use the term shared-access graph, instead.

Definition 7. (**Timestamp-Ignoring Disjoint-Access Parallelism**). *A concurrent data structure is* timestamp-ignoring disjoint-access parallel *if, for every execution α containing two operations op_1 and op_2 that contend on some base object,* other than the timestamp object*, there exists a linearization of α such that there is a path between op_1 and op_2 in the shared-access graph of the minimal execution interval containing op_1 and op_2 for this linearization.*

We remark that a concurrent data structure that satisfies Definition 6 also satisfies timestamp-ignoring disjoint-access parallelism. Notice also that if the `getTimestamp`() operation never attempts to modify the timestamp object, for example, when it is implemented from a shared global clock that increments automatically, then a concurrent data structure that satisfies timestamp-ignoring disjoint-access parallelism also satisfies Definition 6.

Chapter 3

Related Work

3.1 Disjoint-Access Parallelism Definitions

In this section, we present definitions of disjoint-access parallelism that have already appeared in the literature.

Since the publication of the original definition of disjoint-access parallelism [27], many variants of disjoint-access parallelism have been proposed [11, 9, 26]. The original definition of disjoint-access parallelism in [27] requires every two operations that both *access* some base object to have a path between them in the shared-access graph of the minimal execution interval that contains them. Weak disjoint-access parallelism, defined in [11], requires two operations that *concurrently contend* on a base object to have a path between them in the shared-access graph of the minimal execution interval that contains them. Thus, a concurrent data structure that satisfies the definition of disjoint-access parallelism in [27], it also satisfies Definition 6 and, if a concurrent data structure satisfies Definition 6, it also satisfies the definition in [11].

Strict disjoint-access parallelism [26] requires every two operations that contend on a base object to have data sets that intersect. In other words, such operations have an *edge* between them in the shared-access graph of the minimal execution interval that contains them (or the entire execution). Note that a concurrent data structure that satisfies this definition of disjoint-access parallelism also satisfies Definition 6.

In d*-local contention* [9, 37, 38], two operations can *access* the same base object, provided they are connected by a path of length at most d in the shared-access graph of *the entire execution*. For $d > 1$, a concurrent data structure with $(d-1)$-local contention also has d-local contention.

A concurrent data structure that satisfies strict disjoint-access parallelism also satisfies 1-local contention. Moreover, a concurrent data structure with 1-local contention also satisfies the definition of disjoint-access parallelism in [27]. However, d-local contention, for $d \geq 1$, is incomparable to the definitions of disjoint-access parallelism in [11, 27] and Definition 6.

Definition	base objects		execution
Definition 6	contend	path	interval
original [27]	access	path	interval
weak [11]	concurrently contend	path	interval
strict [26]	contend	edge	interval or entire
d-local contention	access	path of length $\leq d$	entire

Figure 3.1: A Comparison of Different Definitions of Disjoint Access Parallelism

3.2 Impossibilities for Disjoint-Access Parallelism in Shared-Memory Computing

In this section, we present impossibility results concerning disjoint-access parallelism that appear in the literature.

In [26], Guerraoui and Kapalka proved that no obstruction-free STM can be strictly disjoint access parallel; specifically, in [26] an STM algorithm is obstruction-free if a transaction can be aborted only when its execution interval is not solo. 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, whereas feeble disjoint-access parallelism (Definition 5), which we consider in our impossibility result, is much weaker. It is worth-pointing out that several obstruction-free STM algorithms [15, 16, 17, 18] 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 shared objects. In particular, this limits the help that each of them can provide to T_1.

Bushkov *et al.* [7] prove that no STM algorithm (whether or not it is disjoint-access parallel) can ensure both local progress and opacity. However, they prove this impossibility result under the assumption that the STM algorithm does not have access to the code of each transaction. As mentioned in their concluding remarks, their impossibility result does not apply to universal constructions, in which the code is provided for each operation to be simulated. In their model, the STM algorithm allows the "external environment" (i.e. the user) to invoke actions for reading a data item, writing a data item, starting a transaction, and trying to commit or abort it. The STM algorithm is only aware of the sequence of invocations of the actions that have been performed and their responses. Thus, a transaction can be helped only after the STM algorithm knows the entire set of data items that the transaction should modify.

Proving impossibility results in a model in which the STM algorithm does not have access to the code of transactions is usually [7, 11, 26] 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.* [11] proved that there is no disjoint-access parallel STM algorithm where read-only transactions are wait-free and *invisible*. A read-only transaction is invisible, if it does not apply non-trivial primitives to shared objects. This impossibility result is proved for the variant of disjoint-access parallelism where processes executing two operations (transactions) concurrently contend on a shared object only if there is a path between the two operations (transactions) in the shared-access graph. We prove our impossibility result for a weaker definition of disjoint-access parallelism and it applies even for implementations with visible reads. We remark that the impossibility result in [11] does not contradict our algorithms, since our implementations employ *visible* reads.

In [23], the concept of *MV-permissiveness* is introduced. An STM algorithm satisfies MV-permissiveness if a transaction aborts only when it is an update transaction that conflicts with another update transaction. The paper [23] proved that no transactional memory algorithm satisfies both disjoint access parallelism (specifically, the variant of disjoint-access parallelism presented in [11]) and MV-permissiveness. However, the paper assumes that the STM 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 impossibility result can be beaten if this requirement is violated. Attiya and Hillel [39] presented a strict disjoint-access parallel lock-based STM algorithm that satisfies MV-permissiveness.

3.3 Disjoint-Access Parallel or Wait-Free Implementations

We continue by presenting universal constructions, STM algorithms, and other implementations that satisfy either disjoint-access parallelism or wait-freedom, from previous work.

More constrained versions of disjoint-access parallelism are used when designing universal constructions or concurrent data structures [37, 38, 27]. Recall that in implementations that ensure d-local contention two operations are allowed to access the same shared object if they are connected by a path of length at most d in the shared-access graph [9, 37, 38]. Afek *et al.* [37, 9] 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 data items. It relies heavily on knowledge of k. This work extends the work of Attiya and Dagan [37], who considered operations on pairs of locations, i.e. where $k = 2$. Afek *et al.* [9] 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 thesis, we prove that, in general, there are no solutions unless we relax some of these properties.

Attiya and Hillel [40] provide a k-local lock-free 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 [40] and the lock-free implementation of `DCAS` by Attiya and Dagan [37] results in an $O(k + \log^* n)$-local lock-free 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 sets of accessed data items are defined on the fly, but it only ensures that progress is lock-free.

A number of wait-free universal constructions [41, 42, 43, 1, 2] 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. Since all operations try to change the shared pointer, the resulting algorithms are not disjoint-access parallel.

Anderson and Moir [44] 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 shared objects even if the sets of data items they access do not (directly or transitively) intersect. The helping technique that is employed by our algorithms combines and extends the helping technique presented in [44] to achieve both wait-freedom, and disjoint-access parallelism (or timestamp-ignoring disjoint-access parallelism).

Anderson and Moir presented in [45] a 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 [42]. 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 lock-free. 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 [15] works in a similar way.

As in Barnes' algorithm, a process executing an operation *op* in our algorithms, locally simulates *op* using a local dictionary, and then tries to apply the changes. However, in our algorithms (DAP-UC and TI-DAP-UC), we detect operations' accesses on the same data item during the simulation phase, so helping occurs at an earlier stage of *op*'s execution. So, more advanced helping techniques are required to ensure wait-freedom and disjoint-access parallelism.

Chuong *et al.* [46] 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 algorithms. Moreover, the concurrent accesses detection and resolution mechanisms employed by our algorithms are more advanced to ensure disjoint-access parallelism.

The first software transactional memory algorithm [3] was disjoint-access parallel, but it is only lock-free and is restricted to transactions that access a pre-determined set of data items. There are other STM algorithms [13, 14, 15, 16, 17, 18] without this restriction that are disjoint-access parallel. However, all of them satisfy weaker progress properties than wait-freedom. TL [13] ensures strict disjoint access parallelism, but it is blocking.

3.4 Wait-Free or Never Aborting Read-Only Transactions

In this section, we present proposed STM algorithms in the literature, that either never abort read-only transaction or guarantee wait-freedom for read-only transactions.

Pessimistic STM algorithms that never abort transactions have been presented in [24, 25]. They use ideas from [48] where an STM system is presented which supports the execution of transactions that contain irrevocable instructions. In the algorithms of [24, 25], read-only transactions are wait-free. However, these algorithms restrict parallelism since the updaters use a single coarse-grain lock for accessing data items; so, update transactions are executed sequentially. We remark that a read-only transaction can read the value of a locked data item without having to wait until it is unlocked; thus, wait-freedom is not violated. Recall that update transactions in WFR-TM employ fine-grained locking for accessing data items, so that those of them that do not conflict can commit in parallel. Popular lock-based STM implementations, which, like WFR-TM use fine-grained locking on each data item that they update, include [29, 32, 49, 3]. There, however, read-only transactions are not wait-free since they may be aborted spuriously.

In [50], a multi-version STM algorithm is introduced which supports wait-free read-only transactions by keeping a list for each data item, where each value that it has had is recorded; read-only transactions can find values for the data items that they read that are mutually consistent. Recall that in [23], MV-permissiveness is introduced which guarantees that read-only transactions never abort. Multi-version MV-permissive STM algorithms are also presented in [51, 23] enhanced with efficient garbage collection for obsolete versions of data items. WFR-TM ensures MV-permissiveness while being single-version, i.e. it does not maintain multiple versions of data items. Thus, WFR-TM is more space efficient in comparison to multi-version algorithms. We remark that in WFR-TM read-only transactions not only never abort, but additionally, they always complete by committing.

Attiya and Hillel present in [12] PermiSTM, an STM algorithm that ensures MV-permissiveness without actually maintaining multiple versions of data items. Instead, transactions that read a data item announce their presence by incrementing a dedicated read-counter linked to this data item; this is done by repeatedly executing CAS until it succeeds. So, a read-only transaction that executes concurrently with update transactions that read the same data item may repeatedly fail to increment the read-counter of the data item. Since all read-only transactions may experience the same problem, it follows that read-only transactions in [12] are obstruction-free; recall that obstruction-freedom does not ensure that a transaction completes unless the thread executing it runs solo for a sufficient number of steps after some point during the transaction's execution. PermiSTM pays this cost in order to ensure disjoint-access parallelism. It has been proved in [11] that in disjoint-access parallel STM implementation with wait-free read-only transactions, a read-only transaction that reads m data items has to perform non-trivial operations on at least $m-1$ shared objects. In WFR-TM, read-only transactions perform only two non-trivial primitives on shared base objects and these base objects are R/W objects. So, WFR-TM does not perform any expensive synchronization primitives at all. However, WFR-TM is not disjoint-access parallel.

Similarly to WFR-TM, PermiSTM supports parallelism among update transactions; update transactions are executed speculatively and they may abort. In PermiSTM, a write-transaction does not update the data items until all read-only transactions that are accessing it are committed (after decrementing the read counter of the data item). Thus, update transactions writing to a data item may face a read-counter whose value is never equal to zero, leading them to run forever. This behavior is avoided in WFR-TM by having update transactions waiting for the completion of only a limited number of read-only transactions.

3.5 Contention Managers, Scheduling, Dependence-Aware Systems

To enhance progress, a lot of research has been performed on designing efficient contention managers and transactional schedulers. A contention manager [52, 53] is a component aiming at ensuring progress by providing efficient conflict resolution policies. When two transactions conflict, the contention manager is employed to decide whether simple techniques, like back-off, would be sufficient, or which of the transactions should abort or be paused to allow the other transaction to complete. SemanticTM prevents conflicts from occurring thus making the use of a contention manager unnecessary.

Somewhat closer to SemanticTM, a *transactional scheduler* [54, 55, 56, 57, 58, 59, 60] is a more elaborated STM component which places transactions in a set of queues, usually one for each thread; a set of working threads then execute transactions from these queues. In addition to deciding which transaction to delay or abort when a conflict occurs, and when to restart a delayed or aborted transaction, a scheduler also decides in which scheduling queue the transaction will be placed once its execution will be resumed or restarted. Some of the schedulers always abort one of the two transactions and place it in an appropriate queue to guarantee that the transaction will be restarted only after the conflicting transaction has finished its execution, i.e. they serialize the execution of the two transactions. CarSTM [58], Adaptive Transaction Scheduling [60], and Steal-on-Abort [54] work in this way. In [56], a scheduler was presented which alternates between reading epochs (where priority is given to the execution of read-only transactions) and writing epochs (where the opposite occurs). This technique behaves better for read-dominated [61] and bimodal [56] workloads, for which schedulers like those presented in [54, 58, 60] may serialize more than necessary. However, the working threads in the algorithm of [56] use locks; additionally, aborts are not avoided. To evaluate a transactional scheduler, competitive analysis is often employed [55, 56, 62, 52] where the total time needed to complete a set of transactions (known as *makespan*) is compared to the makespan of a clairvoyant scheduler [59].

In [63], scheduling is done based on future prediction of the transactions' sets of accessed data items on the basis of a short history of past transactions and the accesses that they performed. If a transaction is predicted to conflict with a live transaction, it is serialized. To avoid serializing more than necessary in cases of low contention, a heuristic is used where prediction and serialization occur only if the completion rate of transactions falls below a certain threshold.

In [64], a lock-based dependence-aware STM system is presented which dynamically detects and resolves conflicts. Its implementation extends ideas from TL II [29] with support of dependence detection and data forwarding. The algorithm serializes transactions that conflict; in case of aborts, cascading aborts may occur. The current version of SemanticTM copes only with transactions that their set of accessed data items are known. However, SemanticTM ensures that all transactions will always commit within a bounded number of steps.

In [65], a database transaction processing system similar to SemanticTM is proposed. From the STM perspective, a database transaction can be thought of as a transaction whose set of accessed data items is known. In this system consecutive transactional instructions of a transaction are separated into groups, called *actions*, according to the set of data items they access. Each worker thread is responsible to execute instructions for a disjoint group of these sets, and each action is scheduled to the appropriate thread. Data dependencies between actions are maintained using extra metadata. Specifically, a shared object (additional to database's tables), called *rendezvous point*, is maintained for the dependencies of each action of some transaction; a single action may have several data dependencies and each of those dependencies will be resolved by the corresponding thread. Using these rendezvous points the execution of a transaction is separated into phases, with each phase containing independent actions. A thread initiating the execution of a transaction T, schedules the independent actions (of the first phase) to the appropriate worker threads. When a worker thread resolves the last dependency of some rendezvous point, it initiates the next phase of T's execution by scheduling the next independent actions of this transaction. However, due to its execution scheme a transaction executed in this system may have to abort, since its actions may conflict with other concurrently executing actions of different transactions. Recall that in SemanticTM transactions never abort.

3.6 Speculation

A way of achieving speculative parallelism is through *thread-level data speculation* (TLDS) [66], [67]. There, code segments are executed in parallel in an optimistic way. The execution of such a code segment may roll back and restart in case inconsistencies are discovered. TLDS can be implemented in software. However, dedicated hardware can facilitate the detection of inconsistencies between different processes.

With goals similar to STM and closely related to SemanticTM, Thread Level Speculation (TLS) [68, 69, 70] uses compiler support to split a program into several tasks which are speculatively executed and each of them finishes by trying to commit. Whenever a consistency violation is detected the conflicting tasks are appropriately aborted, like in STM. In [71], an algorithm that incorporates TLS support on an STM algorithm has been proposed, where each transaction of the STM program is split into several tasks. In this case, consistency violations may arise as a result of either an intra-transaction conflict (i.e. a conflict between the instruction of the same transaction) or an inter-transaction conflict (i.e. a conflict between instructions of different transactions). In both cases, an appropriate tasks' abort policy ensures that no consistency violation occurs. However, in SemanticTM instead of executing tasks, threads execute sets of instructions, each performed on a specific data item (this set may contain instructions of several transactions); so, no conflict ever occurs.

Chapter 4

Disjoint-Access Parallelism in Shared-Memory Computing

4.1 General

In this chapter, we prove a collection of positive and negative results for disjoint-access parallelism. We start by proving in Section 4.2 that linearizable universal constructions which ensure both disjoint access parallelism and wait-freedom are not possible. Then, in Section 4.3, we present a universal construction, called DAP-UC, that achieves both disjoint-access parallelism and wait-freedom, for dynamic data structures in which each operation accesses a bounded number of data items. Also, in Section 4.4 we prove the correctness, progress, and disjoint-access parallelism properties ensured by DAP-UC. Finally, in Section 4.5, we present a new universal construction, called TI-DAP-UC, that ensures wait-freedom and timestamp-ignoring disjoint-access parallelism when applied to any sequential data structure that has a bounded number of entry points.

4.2 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 2.2. 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 says 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 8 (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 set of base objects accessed by trivial primitives and the set of base objects accessed by non-trivial primitives during any solo execution of a*

$$\begin{tikzcd}
C_0 \arrow[r, "\text{APPEND}(L,1)", "\alpha_1"'] & \cdots\cdots \arrow[r] & C_{i-3} \arrow[r, "\text{APPEND}(L,i-2)", "\alpha_{i-2}"'] & C_{i-2} \arrow[r, "\text{APPEND}(L,i-1)", "\alpha_{i-1}"'] \arrow[d, "\text{SEARCH}(L,0)"', "\sigma_{i-2}"] & C_{i-1} \arrow[r, "\text{APPEND}(L,i)", "\alpha_i"'] \arrow[d, "\text{SEARCH}(L,0)"', "\beta_{i-1}\ \gamma_i"] & C_i \arrow[r, "\text{APPEND}(L,i+1)", "\alpha_{i+1}"'] \arrow[d, "\text{SEARCH}(L,0)"', "\beta_i"] & \cdots \\
 & & & {} & {} & {} &
\end{tikzcd}$$

Figure 4.1: The Execution α with Solo Executions of SEARCH$(L, 0)$ Starting from Various Configurations

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 code of the singly-linked list is given in Figure 2.1. 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.

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 quiescent 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 accessed by non-trivial primitives during α_i and let $A(i)$ denote the set of base objects not in $B(i)$ accessed during α_i. Note that base objects in $A(i)$ are only accessed by trivial primitives during α_i. In configuration C_i, the list L consists of i nodes, with values $1, \ldots, i$ in increasing order.

In our proof, we build an infinite execution α' which is indistinguishable from α to process q and which contains an infinite number of steps of a single instance of SEARCH$(L, 0)$ by p. The steps taken by process p in α' are chosen from the solo executions of SEARCH$(L, 0)$ by p starting from C_i, for $i \geq 4$. This is illustrated in Figure 4.1.

For any $i \geq 4$, let $\alpha''_i = \alpha_i\alpha_{i+1}\cdots$ denote the suffix of α starting from C_{i-1}. The set $\bigcup\{B(k) \mid k \geq i\}$ consists of all base objects to which q applies a non-trivial primitive in α''_i

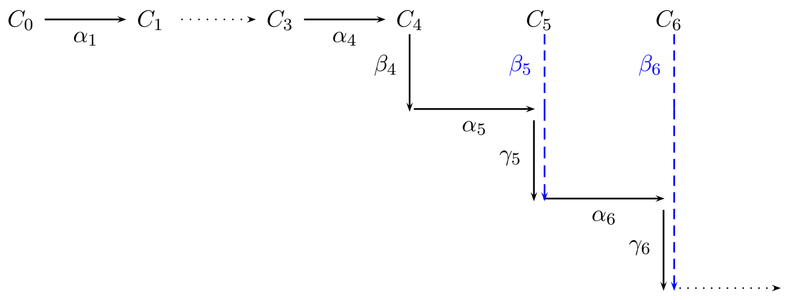


Figure 4.2: An Infinite Execution α' with a Non-terminating SEARCH Operation

and $\bigcup\{A(k) \mid k \geq i\} \cup \bigcup\{B(k) \mid k \geq i\}$ is the set of all base objects accessed by q in α''_i. Let σ_i be the steps of the solo execution of SEARCH$(L, 0)$ by p starting from configuration C_i. Let β_i be the longest prefix of σ_i that does not contend with α''_i, i.e. in which p does not access any base object in $\bigcup\{B(k) \mid k \geq i\}$ and does not apply non-trivial primitives to any base object in $\bigcup\{A(k) \mid k \geq i\}$.

Lemma 9. *For $4 \leq i \leq j$, β_i is a prefix of β_j.*

Proof. Only base objects in $\bigcup\{B(k) \mid i < k \leq j\}$ can have different values in configurations C_i and C_j. Since β_i does not access any base objects in $\bigcup\{B(k) \mid k \geq i\}$, it follows that β_i is also a prefix of σ_j. Since β_i does not contend with α''_i and α''_j is a suffix of α''_i, β_i does not contend with α''_j. By definition of β_j, it follows that β_i is a prefix of β_j. □

For $i \geq 5$, let γ_i be the (possibly empty) suffix of β_{i-1} such that $\beta_{i-1}\gamma_i = \beta_i$, as illustrated in Figure 4.1. We show that $\alpha' = \alpha_1\alpha_2\alpha_3\alpha_4\beta_4\alpha_5\gamma_5\alpha_6\gamma_6\cdots$ is an infinite legal execution starting from C_0. The beginning of this execution appears in Figure 4.2.

Lemma 10. *α' is a legal execution starting from C_0.*

Proof. By definition, β_4 does not apply non-trivial primitives to any base objects accessed in α''_4, and, for $i \geq 5$, $\beta_i = \beta_{i-1}\gamma_i$ (and, hence, γ_i) does not apply non-trivial primitives to any base object accessed in α''_i. Therefore the executions arising from α and α' starting from C_0 are indistinguishable to process q.

We prove by induction that, for all $i \geq 5$, $\alpha_1 \cdots \alpha_4\beta_4\alpha_5\gamma_5 \cdots \alpha_i\gamma_i$ and $\alpha_1 \cdots \alpha_i\beta_i$ are indistinguishable to process p. First consider $i = 5$. Since β_4 does access any base object to which α_5 applies a nontrivial primitive, $\alpha_1 \cdots \alpha_4 \beta_4 \alpha_5 \gamma_5$ and $\alpha_1 \cdots \alpha_4 \alpha_5 \beta_4 \gamma_5 = \alpha_1 \cdots \alpha_4 \alpha_5 \beta_5$ are indistinguishable to process p.

Let $i > 5$ and assume the claim is true for $i - 1$. Then $\alpha_1 \cdots \alpha_4 \beta_4 \alpha_5 \gamma_5 \cdots \alpha_{i-1} \gamma_{i-1}$ and $\alpha_1 \cdots \alpha_{i-1} \beta_{i-1}$ are indistinguishable to process p. Hence, $\alpha_1 \cdots \alpha_4 \beta_4 \alpha_5 \gamma_5 \cdots \alpha_{i-1} \gamma_{i-1} \alpha_i \gamma_i$ and $\alpha_1 \cdots \alpha_{i-1} \beta_{i-1} \alpha_i \gamma_i$ are also indistinguishable to p. Since β_{i-1} does not access any base object to which α_i applies a non-trivial primitive, $\alpha_1 \cdots \alpha_{i-1} \beta_{i-1} \alpha_i \gamma_i$ and $\alpha_1 \cdots \alpha_{i-1} \alpha_i \beta_{i-1} \gamma_i = \alpha_1 \cdots \alpha_i \beta_i$ are indistinguishable to p. Therefore $\alpha_1 \cdots \alpha_4 \beta_4 \alpha_5 \gamma_5 \cdots \alpha_{i-1} \gamma_{i-1} \alpha_i \gamma_i$ and $\alpha_1 \cdots \alpha_i \beta_i$ are indistinguishable to p.

It follows that α' is a legal execution. □

$$\begin{tikzcd}
C_0 \arrow[r, "\text{APPEND}(L,1)", "\alpha_1"'] & \cdots\cdots \arrow[r] & C_{i-1} \arrow[r, "\text{APPEND}(L,0)", "\alpha_i^i"'] & C_i^i \arrow[r, "\text{APPEND}(L,i+1)", "\alpha_{i+1}^i"'] & C_{i+1}^i \arrow[r, "\text{APPEND}(L,i+2)", "\alpha_{i+2}^i"'] & C_{i+2}^i \arrow[r] & \cdots
\end{tikzcd}$$

Figure 4.3: The Execution Obtained from α by Replacing APPEND(L, i) by APPEND$(L, 0)$

For $2 \leq i \leq j$, let C_j^i be the quiescent configuration obtained from configuration C_0 when process q performs the first j operations of execution α, except that the i'th operation, APPEND(L, i), is replaced by APPEND$(L, 0)$; namely, when q performs APPEND$(L, 1)$, …, APPEND$(L, i - 1)$, APPEND$(L, 0)$, APPEND$(L, i + 1)$, …, APPEND(L, j). Let α_i^i denote the solo execution of APPEND$(L, 0)$ by process q starting from configuration C_{i-1} and, for $j > i$, let α_j^i denote the solo execution of APPEND(L, j) by process q starting from configuration C_{j-1}^i This is illustrated in Figure 4.3. Since APPEND is value oblivious, non-trivial primitives are applied to the same set of base objects during the executions leading to configurations C_j and C_j^i. Thus, only base objects in $\cup\{B(k) \mid i \leq k \leq j\}$ can have different values in C_j and C_j^i. Let σ_j^i be the solo execution by p of a SEARCH$(L, 0)$ starting from C_j^i.

Lemma 11. *For $i \geq 4$, β_i is a proper prefix of σ_i.*

Proof. By definition, β_i is a prefix of σ_i. Since β_i does not access any base object in $B(i)$ and these are the only objects that can have different values in C_i and C_i^i, it follows that β_i is a prefix of σ_i^i. Linearizability implies that SEARCH$(L, 0)$ starting from C_i^i is successful, but starting from C_i is unsuccessful. Thus, SEARCH$(L, 0)$ is not completed after β_i. Therefore β_i is a proper prefix of σ_i. □

Lemma 12. *For $i \geq 5$, σ_{i-1}^{i-3} and α_i^{i-3} do not contend.*

Proof. Let S denote the state of the data structure in the quiescent configuration C_{i-1}^{i-3}. In state S, the list has $i - 1 \geq 4$ nodes and the third last node has value 0. Thus, the set of data items accessed by SEARCH$(L, 0)$ starting from state S consists of $L.first$ and the first $i - 3$ nodes of the list. This is disjoint from the set of data items accessed by APPEND(L, i) 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, σ_{i-1}^{i-3} and α_i^{i-3} do not contend. □

Next, for each $i \geq 4$, we prove that there exists $j > i$ such that γ_j is nonempty.

Lemma 13. *For $i \geq 4$, $\beta_i \neq \beta_{i+3}$.*

Proof. To obtain a contradiction, suppose that $\beta_i = \beta_{i+3}$. By Lemma 11, β_i is a proper prefix of σ_i. Let b be the base object accessed in the first step following β_i in σ_i. Then b is also the base object accessed in the first step following β_{i+3} in σ_{i+3}. By definition of β_{i+3}, there is some $\ell \geq i + 3$ such that this step is either an access to $b \in B(\ell)$ or the application of a non-trivial primitive to $b \in A(\ell)$.

By the value obliviousness assumption, the set of base objects access by non-trivial primitives during $\alpha_{\ell-3}\alpha_{\ell-2}\alpha_{\ell-1}$ and $\alpha_{\ell-3}^{\ell-3}\alpha_{\ell-2}^{\ell-3}\alpha_{\ell-1}^{\ell-3}$ are the same, so 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 $\ell - 3 \geq i$, β_i does not access any of these base objects, so β_i is also a prefix of $\sigma_{\ell-1}^{\ell-3}$. Furthermore, the first step following β_i in this execution is the same as the first step following β_i in σ_i, i.e. it is either an access to $b \in B(\ell)$ or an application of a non-trivial primitive to $b \in A(\ell)$. By the value obliviousness assumption, $B(\ell)$ is the set of base objects accessed by non-trivial primitives during $\alpha_\ell^{\ell-3}$ and $A(\ell)$ is the set of base objects not in $B(\ell)$ accessed during this execution. Thus, $\sigma_{\ell-1}^{\ell-3}$ and $\alpha_\ell^{\ell-3}$ contend on b. This contradicts Lemma 12. Hence, $\beta_i \neq \beta_{i+3}$. □

It follows that, for $i \geq 4$, at least one of γ_{i+1}, γ_{i+2}, and γ_{i+3} is nonempty. Hence γ_j is nonempty for infinitely many integers $j \geq 5$. Therefore, in the infinite execution α', process p never completes its operation 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 14. *No feebly disjoint-access parallel linearizable universal construction that satisfies the value obliviousness assumption is wait-free.*

4.3 The DAP-UC Universal Construction

In this section, we present a universal construction that is linearizable, wait-free and disjoint-access parallel (Definition 6) provided each operation of the sequential data structure to which it is applied never access more than M data items, M is a constant.

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 direc
    value $val$
    ptr to oprec $A[1..n]$

type statrec
    $\{\langle st : simulating \rangle,$
     $\langle st : restart,$ ptr to oprec $restartedby \rangle,$
     $\langle st : modifying,$ ptr to dictionary of dictrec $changes,$ value $output \rangle$
     $\langle st : done \rangle\}$

type oprec
    code $program$
    process id $owner$
    value $input$
    value $output$
    statrec $status$
    ptr to oprec $tohelp[1..n]$

type dictrec
    ptr to direc $key$
    value $newval$
```

Figure 4.4: Type Definitions of DAP-UC

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 other operations that are concurrently accessing x by reading this record. So, concurrent accesses on the same data item are detected without violating disjoint access parallelism.

```
19  value PERFORM($prog, input$) by process $p$:
20      $opptr$ := pointer to a new oprec record
        $opptr \rightarrow program := prog$, $opptr \rightarrow input := input$, $opptr \rightarrow output := \perp$
        $opptr \rightarrow owner := p$, $opptr \rightarrow status := \langle simulating \rangle$,
        $opptr \rightarrow tophelp[1..n] := [nil, \ldots, nil]$
21      HELP($opptr$)                       /* $p$ helps its own operation */
22      for $q := 1$ to $n$ excluding $p$ do       /* $p$ helps operations that have been restarted by its operation $op$ */
23          if ($opptr \rightarrow tohelp[q] \neq nil$) then HELP($opptr \rightarrow tohelp[q]$)
24      return ($opptr \rightarrow output$)
```

Figure 4.5: The Code of PERFORM of DAP-UC

The algorithm uses a simple priority scheme, based on the identifiers of the processes that invoke the operations, to resolve situations where processes are concurrently accessing the same data item. When an operation *op* determines that it concurrently accesses the same data item with an operation *op*$'$ of higher priority, *op* helps *op*$'$ to complete before it continues its execution. On the other hand, if *op* has higher priority than *op*$'$, *op* causes *op*$'$ to restart. In this case, 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 9-15) 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 20). 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 p should help other operations before starting a new operation. We call p the *owner* of *op*. To execute *op*, p calls HELP (line 21). To ensure wait-freedom, before *op* returns, the owner of *op* helps all other operations (with lower priority) listed in the $tohelp$ array of the $oprec$ record of *op* (lines 22-23). These are operations that concurrently accessed the same data item with *op* during the course of its execution, so disjoint-access parallelism is not violated. The algorithm also maintains a record of type `direc` (lines 1-3) for each data item x. In the code, we also denote by x a pointer to the `direc` corresponding to that data item. 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 detect operations that are concurrently accessing x.

```
25  HELP($opptr$) by process $p$:
26      $opstatus$ := LL($opptr \rightarrow status$)
27      while ($opstatus \neq \langle done \rangle$)

28          if ($opstatus = \langle restart, opptr' \rangle$) then          /* $op'$ has restarted $op$ */
29              HELP($opptr'$)                                          /* first help $op'$ */
30              SC($opptr \rightarrow status$, $\langle simulating \rangle$)       /* try to change the status of $op$ back to $\langle simulating \rangle$ */
31              $opstatus$ := LL($opptr \rightarrow status$)

32          if ($opstatus = \langle simulating \rangle$) then               /* start a new simulation phase */
33              $dict$ := pointer to a new empty dictionary of dictrec records
34              $ins$ := first instruction in $opptr \rightarrow program$
35              while $ins$ is not a return do                          /* simulate instruction $ins$ of $op$ */
36                  if (($ins$ is WRITEDI($x, v$) or READDI($x$)) and             /* first access of $x$ by
                        (there is no dictrec with key $x$ in $dict$)) then     this attempt of $op$ */
37                      ANNOUNCE($opptr, x$)                            /* announce that $op$ is accessing $x$ */
38                      CONCURRENTACCESSES($opptr, x$)  /* possibly, help or restart other operations accessing $x$ */
39                      if ($ins$ = READDI($x$)) then $v$ := $x \rightarrow val$
40                      add new dictrec $\langle x, v \rangle$ to $dict$               /* create a local copy of $x$ */
41                  else if ($ins$ is CREATEDI()) then
42                      $x$ := pointer to a new direc record
43                      $x \rightarrow A[1..n] := [nil, \dots, nil]$
44                      $x \rightarrow A[opptr \rightarrow owner] := opptr$
45                      add new dictrec $\langle x, nil \rangle$ to $dict$
46                  else   /* either $ins$ 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

47                  if ($\neg$VL($opptr \rightarrow status$) then break                /* end of the simulation of $ins$ */
48                  $ins$ := next instruction of $opptr \rightarrow program$
              /* end while */

49              if ($ins$ is return ($v$)) then                                 /* $v$ may be empty */
                          /* try to change status of $op$ to modifying; it is successful iff simulation is over and status of $op$ unchanged
50                  SC($opptr \rightarrow status$,$\langle modifying, dict, v \rangle$)               since beginning of simulation */

51              $opstatus$ := LL($opptr \rightarrow status$)

52          if ($opstatus = \langle modifying, changes, out \rangle$) then
53              $opptr \rightarrow outputs := out$
54              for each dictrec $\langle x, v \rangle$ in the dictionary pointed to by $changes$ do

55                  LL($x \rightarrow val$)  /* try to make writes visible */
56                  if ($\neg$VL($opptr \rightarrow status$)) then return                 /* $op$ is completed */
57                  SC($x \rightarrow val, v$)

58                  LL($x \rightarrow val$)
59                  if ($\neg$VL($opptr \rightarrow status$)) then return                 /* $op$ is completed */
60                  SC($x \rightarrow val, v$)

61              SC($opptr \rightarrow status$, $\langle done \rangle$)
62              $opstatus$ := LL($opptr \rightarrow status$)
      /* end while */
63      return
```

Figure 4.6: The Code of HELP of DAP-UC

```
64  ANNOUNCE($opptr, x$) by process $p$:
65      $q := opptr \rightarrow owner$
66      LL($x \rightarrow A[q]$)
67      if ($\neg$ VL($opptr \rightarrow status$)) then return
68      SC($x \rightarrow A[q], opptr$)
69      LL($x \rightarrow A[q]$)
70      if ($\neg$VL($opptr \rightarrow status$)) then return
71      SC($x \rightarrow A[q], opptr$)
72      return

73  CONCURRENTACCESSES($opptr, x$) by process $p$:
74      for $q' := 1$ to $n$ excluding $opptr \rightarrow owner$ do
75          $opptr' :=$ LL($x \rightarrow A[q']$)
76          if ($opptr' \neq nil$) then                                  /* $op$ may concurrently access $x$ with $op'$ */
77              $opstatus' :=$ LL($oppptr' \rightarrow status$)
78              if ($\neg$VL($opptr \rightarrow status$)) then return
79              if ($opstatus' = \langle modifying, -, - \rangle$) then HELP($opptr'$)
80              else if ($opstatus' = \langle simulating \rangle$) then
81                  if ($opptr \rightarrow owner < q'$) then            /* $op$ has higher priority than $op'$, restart $op'$ */
82                      $opptr \rightarrow tohelp[q'] := opptr'$
83                      if ($\neg$VL($opptr \rightarrow status$)) then return
84                      SC($opptr' \rightarrow status, \langle restart, opptr \rangle$)
85                      if (LL($oppptr' \rightarrow status$) = $\langle modifying, -, - \rangle$) then HELP($opptr'$)
86                  else HELP($opptr'$)                                  /* $op$ has lower priority that $op'$, help $op'$ */
87      return
```

Figure 4.7: The Code of ANNOUNCE and CONCURRENTACCESSES of DAP-UC

The execution of *op* is done in a sequence of one or more *simulation phases* (lines 32-51) followed by a *modification phase* (lines 52-60). In a simulation phase, the instructions of *op* are read (lines 34, 35, and 48) and the execution of each one of them is simulated locally. During each simulation phase, the first time a process q helping *op* (including its owner) needs to access a data item (lines 36, 41), it creates a local copy of it in its (local) dictionary (lines 40, 45). All subsequent accesses by q to this data item (during the current simulation phase of *op*) are performed on this local copy (line 46). During the modification phase, q makes the updates of *op* visible by applying them to the shared memory (lines 54-60).

The $status$ field of *op* determines the execution phase of *op*. It contains a pointer to a record of type `statrec` (lines 4-8) where the status of *op* is recorded. The status of *op* can be either $\langle simulating \rangle$, indicating that *op* is in its simulation phase, $\langle modifying, -, - \rangle$, if *op* is in its modifying phase, $\langle done \rangle$, if the execution of *op* has been completed (although *op* may not have yet returned), or $\langle restart, -\rangle$, if *op* has concurrently accessed some data item with another operation (of higher priority) 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.

Whenever process p accesses a data item x for the first time during a simulation phase, p checks, before reading the value of x, whether *op* is concurrently accessing x with other operations. This is done as follows: p announces *op* to x by storing a pointer *opptr* to *op*'s `oprec` in $A[q]$, where $q = opptr \rightarrow owner$. This is performed by calling ANNOUNCE (line 37). ANNOUNCE first applies an `LL` on $x \rightarrow A[q]$ (line 66), where x is the `direc` for x. Then, it checks if the status of *op* (line 67) remains $\langle simulating\rangle$ and, if this is so, it applies a `SC` to store *opptr* in $x \rightarrow A[q]$ (line 68). These three 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 $x \rightarrow A[q]$ that stores a pointer to this operation's `oprec` causing the `SC` by q (on line 68) to fail. However, we prove in Section 4.4 that this can happen only once, so executing the instructions on lines 66-68 twice is enough.

After announcing *op* to x, p calls CONCURRENTACCESSES (line 38) to detect other operations that are concurrently accessing x. In CONCURRENTACCESSES, p reads all the elements of $x \rightarrow A$ except $A[q]$ (lines 74-75). Whenever is detected that *op* concurrently accesses some data item with some other operatiom *op*$'$ (i.e. the condition of the `if` statement of line 76 evaluates to `true`), p first checks if *op*$'$ is in its modifying phase (line 79) 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 $\langle modifying, -, -\rangle$, it will next become $\langle done\rangle$, and then, henceforth, never changes. If the status of *op*$'$ is $\langle simulating\rangle$, p determines which of *op* or *op*$'$ has the higher priority (line 81). If *op*$'$ has higher priority (line 86), then p helps *op*$'$ by calling HELP(*op*$'$). Otherwise, p first adds a pointer *opptr*$'$ to the `oprec` of *op*$'$ into $opptr \rightarrow tohelp$ (line 82), so that q, the owner of *op*, will help *op*$'$ to complete after *op* has completed. Then p attempts to restart *op*$'$, using `SC` (line 84) to change the status of *op*$'$ to $\langle restart, opptr\rangle$, where *opptr* is a pointer to the `oprec` of *op*. When *op*$'$ restarts its simulation phase, it will help *op* to complete (lines 28-31), if *op* is still in its simulation phase, before it continues with the re-execution of the simulation phase of *op*$'$. This guarantees that $opttr' \rightarrow status$ will not be set to $\langle restart, opptr\rangle$ again.

Recall that each helper p of *op* maintains a local dictionary. This dictionary contains an element of type `dictrec` (lines 16-18) for each data item that p accesses (while simulating *op*). A dictionary element corresponding to data item x consists of two fields, key, which is a pointer to the `direc` corresponding to 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 50, which changes the status of *op* to $\langle modifying, -, -\rangle$. 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 52, 54). For each element in the dictionary, it tries to write the new value into the `direc` of the corresponding data item (lines 55-57). This is performed twice to avoid problems with obsolete helpers in a similar way as in ANNOUNCE.

Theorem 15. *The* DAP-UC *universal construction (Figures 4.4, 4.5, 4.6, and 4.7) produces disjoint-access parallel, wait-free, concurrent data structures when applied to sequential data structures whose operations access a bounded number of data items in any sequential execution.*

4.4 Proof of the DAP-UC Algorithm

4.4.1 Preliminaries

The proof is divided in three parts, namely consistency (Section 4.4.2), wait-freedom (Section 4.4.3) and disjoint-access parallelism (Section 4.4.4). 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 20). Thus, we will not distinguish between an operation and its `oprec`.

Observation 16. *The status of each* `oprec` *is initially $simulating$ (line 20). It can only change from $simulating$ to* modifying *(lines 32, 50), from* modifying *to $done$ (lines 52, 61), from $simulating$ to $restart$ (lines 80, 84), and from $restart$ to $simulating$ (lines 28, 30).*

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

Observation 17. *Let op be any operation and let opptr be the pointer to its* `oprec`. *When a process returns from* Help($opptr$) *(on line 56, 59 or 63), $opptr \rightarrow status = done$.*

This follows from the exit condition of the **while** loop (line 27) and the fact that, once the status of an `oprec` becomes *modifying*, it can only change to $done$.

Observation 18. *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 24), has also returned from a call to HELP (on line 21), so the status of the `oprec` it created (on line 20) has status $done$, and the fact that a process does not call PERFORM recursively, either directly or indirectly.

Observation 19. *For every* `direc`*,* $A[i]$*,* $1 \leq i \leq n$*, 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 `direc` is created (on line 42) and is updated only on lines 68 or 71.

4.4.2 Linearizability

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 20. *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 26, 31, or 51 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 `dictrecs` (line 33). 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 50), then att is *successful.* Notice that, in this case, $chgs$ is a pointer to the dictionary associated with att.

By Observation 16, 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 50). Next observation then follows from the definition of a successful attempt:

Observation 21. *For each operation, there is at most one successful attempt.*

In att, p *simulates* instructions on behalf of *op* (lines 32 - 50). The simulation of an instruction ins *starts* when ins is fetched from *op*'s $program$ (on lines 34 or 48) and *ends* either just before the next instruction starts simulated, or just after the execution of the `SC` on line 50 if ins is the last instruction of $opptr \rightarrow program$.

When p simulates a CREATEDI() instruction, it allocates a new `direc` record x in its own stripe of shared memory (line 42) and adds a pointer to it in the dictionary associated with att (line 45); 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 `direc`.

We say that a `direc` 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 22. *A* `direc` 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 `direc` x by (for) *op*, if it either simulates an $ins \in \{$READDI(x), 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 23. *If, in att,* p *starts the simulation of an instruction* $ins \in \{$WriteDI$(x, _)$, 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 `direc` x during att, a new `dictrec` record is added for x to the dictionary associated with att (on lines 40 or 45). From this and by inspecting the code lines 36, 40, 41 and 45 follows the observation below.

Observation 24. *If a* `direc` 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 36 to 40 or lines 41 to 45 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 `direc` x during *att*, if it successfully executes an `SC` of line 68 or line 71 on $x.A[q]$ (recall that $\mathit{opptr} \to \mathit{owner} = q$) with value *opptr*, during a call of ANNOUNCE(opptr, x) (on line 37). Distinct processes may perform attempts of the same operation *op*. However, once an operation has been announced to a `direc`, 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 25. *Assume that p calls* 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 $\to$ status $\neq$ done,* $(x.A[\mathit{opptr} \to \mathit{owner}]) = \mathit{opptr}$.

Proof. Since *att* is active when p returns from ANNOUNCE(opptr, x), the tests performed on lines 67 and 70 are successful. So, p performed `LL`$(x \to A[q], \mathit{opptr})$ on lines 66 and 69 respectively. Let C_{LL1} and C_{LL2} be the configurations immediately after p performed line 66 and 69, respectively.

Let C be a configuration after p has returned from the call of ANNOUNCE(opptr, x) in which $\mathit{opptr} \to \mathit{status} \neq \mathit{done}$. Assume, by contradiction, that $(x.A[q]) = \mathit{opptr}'$ in C, where opptr' is a pointer to an operation $\mathit{op}' \neq \mathit{op}$. Let p' be the last process that changes the value of $x.A[q]$ to opptr' before C. Therefore p' performed a successful `SC`$(x.A[q], \mathit{opptr}')$ on line 68 or line 71. This `SC` is preceded by a `VL`$(\mathit{opptr}' \to \mathit{status})$ (on line 67 or line 70), which is itself preceded by a `LL`$(x \to A[q])$ (on line 66 or line 69). 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 $(\mathit{opptr}' \to \mathit{status})$ is successful, $\mathit{opptr}' \to \mathit{status} = \mathit{simulating}$ in configuration C'_{VL}.

By Observation 19, $\mathit{opptr}' \to \mathit{owner} = q$. By Observation 18, in every configuration, there is only one operation owned by q whose status is not *done*. Since *op* has status $\mathit{simulating}$ when p started its attempt and the status of *op* is not equal to *done* in C, it then follows from Observation 16 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 `SC`$(x \to$

$A[q], \mathit{opptr}'')$, 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 SC($x \to A[q], \mathit{opptr}$) performed by p on line 71 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 $x \to A[q]$ 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 68) 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 SC$(x.A[q], \mathit{opptr})$ on line 68, which also prevents the SC performed by p' from being successful, which is a contradiction. □

Attempts of distinct operations may access the same direcs. When an attempt *att* of *op* accesses a direc x for the first time by simulating READDI(x) or WRITEDI(x, _), the operation is first announced to x (on line 37) and then CONCURRENTACCESSES(opptr, x) is called (on line 38, *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 76), op' is either restarted (on line 84) or helped (on lines 79, 85 or 86). Since when HELP(op') returns, the status of op' is done (Observation 17), in both cases attempt att' is no longer active when the call to CONCURRENTACCESSES(opptr, x) returns. This is precisely what next Lemma establishes.

Lemma 26. *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* direc. *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* ConcurrentAccesses*(opptr', x) (on line 38) and returns from that call; denote by C'_D the configuration that follows the termination of* ConcurrentAccesses*(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, $\mathit{opptr} \to \mathit{status} = \mathit{done}$ 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 CONCURRENTACCESSES(*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 CONCURRENTACCESSES(*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 `direc` pointed to by x (on line 74). 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 16, 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 75) 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 25, we have that $A[q] = opptr$ in C'_{RA}.

As attempt att' is active in C'_D, it is active when p' performs CONCURRENTACCESSES(*opptr*$'$, x). In particular, each `VL` on $opptr' \rightarrow status$ performed by p' (on line 78 or 83) in the execution of CONCURRENTACCESSES(*opptr*$'$, x) returns `true`. Therefore, p' reads the status of the operation pointed to by *oppptr* (`LL`($opptr \rightarrow status$) on line 77). 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 79). In the configuration C in which p' returns from this call, $opptr \rightarrow status = done$ (Observation 17). 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 80). 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 84). 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 85) after performing the unsuccessful `SC`. When this call returns, $opptr \rightarrow status = done$ (Observation 17) 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 17), and p' returns from the call to HELP(*opptr*) before C'_D. □

In an attempt of *op*, a new `direc` is created each time a CREATEDI() instruction is simulated on line 42. For such a `direc` to be later accessed in another attempt, a pointer to it must be either written to the val field of another `direc`, or passed as an input parameter to an operation. Moreover, when the `direc` is accessed, the status of the operation *op* is done.

Lemma 27. *Suppose that in att, p creates a* `direc` x*. If an instruction* ReadDI*(*x*) or* WriteDI*(*$x, \_$*) is simulated in an attempt att*′ *of an operation op*′ $\neq$ *op, then op* $\rightarrow$ $status = done$ *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 42) and then a `dictrec` with key a pointer to x is added to the dictionary associated with *att* (on line 45). 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 `direc` x' created by *att* such that x' is written to a `direc` that is not created by *att*, or it is returned by *op*. This is so, since otherwise, no `direcs` 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 21 and 24) and Observation 17 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 `direc`s 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 `direc`, 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 24 implies that the first time it is accessed in att, ANNOUNCE(*opptr*, y) and CONCURRENTACCESSES(*opptr*, y) are called (lines 37–38). 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 CONCURRENTACCESSES(*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 50 that changes the status of *op* to $\langle modifying, \_, \_\rangle$ is applied.
Similarly, Observation 24 implies that ANNOUNCE(*opptr*$'$, y) and CONCURRENTACCESSES(*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 CONCURRENTACCESSES(*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 47).
If C_A occurs before C'_A, it follows from Lemma 26 that $opptr \rightarrow status = done$ in C'_D. Therefore, by Observation 16, 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 26 that att' is not active in C_D. Since the SC on line 50 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 `direc`. 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 28. *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 `direc` x in configuration C. Since every process owns at most one operation with $status \neq done$ in every configuration (Observation 18), $op \rightarrow owner \neq op' \rightarrow owner$.

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 50). Since once the status of an operation is $\langle modifying, \_, \_\rangle$, it can only change to $done$ (Observation 16), 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 34 - 48). 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 READDI(x) or WRITEDI($x, \_$) is simulated. By the code, p calls in att ANNOUNCE($opptr, x$) and CONCURRENTACCESSES($opptr, x$) (on lines 37 and 38, respectively) before adding a `dictrec` $\langle ptr_x, \_\rangle$, to its dictionary (on line 40), where *opptr* is pointing to *op*. Similarly, p' calls in *att'* ANNOUNCE(*opptr'*, x) and CONCURRENTACCESSES(*opptr'*, x), where *opptr'* is a pointer to *op'*, and p' returns from both calls. Assume without loss of generality that p' returns from ANNOUNCE(*opptr'*, x) after p returns from ANNOUNCE(*opptr*, x) by p. Denote by C'_D the configuration immediately after p' returns from CONCURRENTACCESSES(*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 26, *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 CREATEDI() is simulated. Whenever a new `direc` is created (on line 42), a distinct shared memory slot is allocated to this `direc`. A `dictrec` record $\langle ptr_x, \_\rangle$ cannot thus be added in both d and d' at line 45 when a CREATEDI() instruction is simulated.

  Suppose without loss of generality that, in *att*, $\langle x, \_\rangle$ is added to d on line 45, as a result of the simulation of a CREATEDI() instruction. ptr_x is thus added to d' the first time a READDI(x) or WRITEDI($x, \_$) instruction for *op'* is simulated by p' in *att'*. By Lemma 27, *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 `direcs` referenced by *op* (on lines 52–62). 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 29. *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* `dictrecs`*. 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* `direc` *x, $ptr_x \to val = v$ 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 SC($op \rightarrow status, done$) on line 61 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 55 - 60) 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 56 and 59. Thus, p executes two SC instructions $\texttt{SC}_1$ and $\texttt{SC}_2$ on lines 57 and 57, 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 SC($ptr_x \rightarrow val, v'$) on line 57 or 60; denote by $\texttt{SC}'$ this SC and let $\texttt{LL}'$ and $\texttt{VL}'$ be its mathing LL and VL (which are executed on lines 55 and 56 or 58 and 59), 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 28, 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 direc for each static data item of the data structure. Each direc that is created (on line 42) becomes a public dynamic data item if the attempt that creates it is successful. The *current value* of a direc in a configuration is the value of its val field, unless the direc 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 `direc`. Note that, by Lemma 28, in each configuration, each `direc` is referenced by at most one operation.

Recall that a `direc` is public in configuration C if it corresponds to a `direc` 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 `direc` 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 33 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 30. *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 33 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 31. *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 32. *The linearization point of each operation is within its execution interval.*

We continue with our main theorem which proves consistency.

Theorem 33 (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 33 relies on the following lemma.

Lemma 34. *Let att denote an attempt by p of some operation op. Suppose that in att, $x \rightarrow val$ is read by p while an instruction* ReadDI*(x) is simulated (line 39), let r be this read of $x \rightarrow val$, 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 $v' = x \rightarrow val$.

In configuration C_r, which is equal to C' or precedes C', $x \rightarrow val = v \neq v'$. 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 28, *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 CONCURRENTACCESSES(*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 CONCURRENTACCESSES(*opptr*$'$, x), respectively. Notice that C'_A and C'_D precede C'.

By the code it follows that before reading $x \rightarrow val$, p calls ANNOUNCE($opptr, x$) and CONCURRENTACCESSES($opptr, x$) where *opptr* is pointing to *op*. Denote by C_A and C_D the configurations in which p returns from ANNOUNCE($opptr, x$) and CONCURRENTACCESSES($opptr, x$), 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 26, 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 CONCURRENTACCESSES($opptr', x$) is successful, since p' is the process that changes the status of *op*$'$ to $\langle modifying, \_, \_\rangle$. Thus, it follows from Lemma 26 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 33.

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 29, 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 50. 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 `direc` 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 `direc` 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 26, 31, 51 or 62). 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 `direc` 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 39 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 34 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 40 or 45). 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 $\text{READDI}(x)$. 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 34, 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 48. 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 40 or 45). 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 `direcs` 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 READDI(x) and x is not accessed in τ, it follows from Lemma 34 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 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 $ins =$ WRITEDI(x, v) or ins is a CREATEDI() that creates x, x has the same value (v or *nil* if $ins =$ CREATEDI()) in both $LC(p, C', op)$ and $S_{C'}\tau'$.

□

4.4.3 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 35. *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 20) and when it is updated (on line 82), *opptr*$'$ points to an `oprec` whose owner is p', by Observation 19 (line 75).

We say that *op restarts op*$'$ in an execution if some process calls CONCURRENTACCESSES(*opptr*$, x$), where *opptr* points to *op* and x points to a `direc`, and successfully performs `SC`(*opptr*$' \rightarrow status, \langle restart, opptr \rangle$) (on line 84), where *opptr*$'$ points to *op*$'$. Note that, by line 81, 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 36. *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 30), where *opptr* is a pointer to *op*, it performs HELP(*opptr*) on line 29. When this returns, the status of *op* is $done$, by Observation 17.

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 77, q sees that *op*$'$ has status $simulating$, it will see that the status of *op* is done, when it performs line 83. Hence, q will not restart *op*$'$ on line 84. □

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

Lemma 37. *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 84 for *op*$'$; let $opptr_i$ be a pointer to the `oprec` record of op_i. Before doing so, q_i set $opptr_i \rightarrow tohelp[p'] = opptr'$ (on line 82) and then checked that the status of op_i was still $simulating$ (on line 83); thus, $opptr_i \rightarrow tohelp[p']$ is written before the completion of op_i.

Lemma 36 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 24). Before this response, p reads $opptr_i \rightarrow tohelp[p']$ on line 23. Since, the call of HELP($opptr_i$) by p (on line 21) has responded before this read, Observation 17 implies that this read is performed after the $status$ of op_i changed to $done$; thus, it is performed after q_i set $opptr_i \rightarrow tohelp[p'] = opptr'$.

If in the meantime the value of $opptr_i \rightarrow tohelp[p']$ has not changed, then p calls HELP($opptr'$). By Observation 17, 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 $opptr_i \rightarrow tohelp[p'] = opptr'_i$, where $opptr'_i \neq opptr'$, while executing an attempt of $opptr_i$. Observation 35 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 18 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 35 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 $opptr_i \rightarrow tohelp[p'] = opptr'$, that is before p reads $opptr_i \rightarrow tohelp[p']$ (on line 23), 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 75 and before it executes line 82, which sets $opptr_j \rightarrow tohelp[p'] = opptr'_j$, it reads the $status$ of $opptr'_j$ (on line 77) and checks whether it is still $simulating$ (on line 80). 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 82 is not executed; that is a contradiction. □

From Lemmas 36 and 37, we get the following result.

Corollary 38. *An operation can be restarted at most* $2 * (n - 1)$.

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

Lemma 39. *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 29, 0, 85, and 86. If p calls HELP($opptr'$) recursively on line 0 or 85, then, by Observation 16, $opptr' \rightarrow status$ is either *modifying* or $done$, so, this recursive call will eventually return without itself making recursive calls to HELP.

By line 75 and Observation 19, when line 81 is performed, $opptr' \rightarrow owner = p'$. From line 81, if p calls HELP(*opptr'*) recursively on line 86, then $opptr \rightarrow owner > opttr' \rightarrow owner$.

If $opptr' \rightarrow status = \langle restart, opptr \rangle$, then, from lines 84 and 81, $opptr \rightarrow owner < opttr' \rightarrow owner$. Hence, if p calls HELP(*opptr'*) recursively on line 29, $opptr \rightarrow status = \langle restart, opptr' \rangle$, 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 40. *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 27–61 during HELP(*opptr*), process p performs LL$(opptr \rightarrow status)$ on line 26, 31, 51, or 62.

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 24 and Theorem 33 (item 1) imply that the `dictrecs` in a dictionary have different keys (i.e. point to different `direcs`) and correspond to different data items accessed by a sequential execution of *op* applied to the data structure (lines 36, 40, and 45). 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 54–60 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 29) and, excluding this, performs a constant numbers of steps.

Finally, suppose that *opptr* has status $simulating$ at the beginning of an iteration. Theorem 33 (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 47 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 35–48, in which it takes a constant number of steps, excluding calls to CONCURRENTACCESSES. Observation 24, Theorem 33 (item 2), and the definition of M, imply that CONCURRENTACCESSES can be called at most M times during an active attempt of *op*. Then, Theorem 33 (item 2) imply that process p performs a constant number of steps and at most one recursive call to HELP (on line 0, 85, or 86) each time it calls CONCURRENTACCESSES. Thus, excluding the recursive calls to HELP, this iteration of the **while** loop on lines 27–61 eventually completes.

If p does not return on line 63 after exiting from the **while** loop or on line 56 or 59, it tries to change $opptr \rightarrow status$ via an SC on line 30, 50, or 61. Therefore, each time p performs an iteration of the **while** loop on lines 27–61, $opptr \rightarrow status$ changes. It follows from Observation 16 and Corollary 38 that p performs at most $2n$ complete iterations of this **while** loop during HELP($opptr$).

By Lemma 39, 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 41. *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 40 that all these instances of HELP eventually return. Thus, this call of PERFORM eventually returns. □

4.4.4 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 20, 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 31.

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 shared-access graph of an execution interval I. The shared-access 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 $SAG(op, op')$ the shared-access 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 26, 31, 51 or 62 that returns $simulating$ and during which $op \rightarrow status$ does not change.

Lemma 42. *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 26, 31, or 47). 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 33(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 43. *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 44. *If a process executes a successful* `VL`*(opptr $\rightarrow$ status) while performing* Announce*(opptr, x) or* ConcurrentAccesses*(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 37) and CONCURRENTACCESSES$(opptr, x)$ (on line 38) if $opptr \rightarrow status$ was $simulating$ (line 32) when p last executed `LL`$(opptr \rightarrow status)$ (on line 26, 31, or 51).

When helping an operation *op*, process p may starts helping another operation *op*$'$. This occurs for example when two operations concurrently accessing the same data item are discovered by p, that is, when the two operations access the same `direc`. 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 HELP(*opptr*) and HELP(*opptr*$'$), where *opptr* and *opptr*$'$ are pointers to operations *op* and *op*$'$, respectively. Denote by I the execution interval of HELP(*opptr*). We say that HELP(*opptr*$'$) is *directly called by p after* HELP(*opptr*) if p calls HELP(*opptr*$'$) in I and every other call to HELP previously made in by p in I has returned when HELP(*opptr*$'$) is called by p .

Lemma 45. *If* Help*(opptr$'$) with opptr$'$ pointing to op$'$ is called directly by p after calling* Help*(opptr) with opptr pointing to op, then $DS(op) \cap DS(op') \neq \emptyset$.*

Proof. In an instance of HELP(*opptr*) by p, where *opptr* is pointing to *op*, HELP(*opptr*$'$) with *opptr*$'$ pointing to *op*$'$ may be called on line 29, when p discovers that *op* has been restarted, or in the resolution of the concurrent accesses on some `direc` x, when p executes CONCURRENTACCESSES(*opptr*, x) (lines 79, 85 or 86). We consider these two cases separately:

- HELP(*opptr*$'$) is called in the execution of CONCURRENTACCESSES(*opptr*, x). Before calling CONCURRENTACCESSES(*opptr*, x), p calls ANNOUNCE(*opptr*, x) (line 37). Therefore, it follows from Lemma 42 that $x \in DS(op)$. For HELP(*opptr*$'$) to be called in CONCURRENTACCESSES(*opptr*, x), *opptr*$'$ is read from $x \rightarrow A[q']$, where q' is the owner of *op*$'$ (`LL` on line 75). Hence, *op*$'$ has been previously announced to x, from which we conclude by corollary 43 that $x \in DS(op')$.
- HELP(*opptr*$'$) is called on line 29. This means that some process p' has changed the status of *op* to $\langle restart, opptr' \rangle$ (`SC` on line 84). p' thus calls CONCURRENTACCESSES(*opptr*$'$, x) for some `direc` x in which it applies a successful `SC`(*opptr*, $\langle restart, opptr' \rangle$). By the code of CONCURRENTACCESSES, this implies that *opptr* is read from $x \rightarrow A[q]$, where q is the owner of *op* (`LL` on line 75). Thus, *op* has been announced to x, from which we have by Corollary 43 that $x \in DS(op)$. Moreover, p' calls CONCURRENTACCESSES(*opptr*$'$, x) after returning from a call to ANNOUNCE(*opptr*$'$, x). Hence, by Lemma 42, $x \in DS(op')$. □

When a process p is performing an operation *op*, i.e. p has called 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 46. *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 30, 50, 53, 61 in the code of HELP or on lines 82 or 84 in the code of CONCURRENTACCESSES. The non-trivial primitive applied by p on line 30, 50 or 61 is a `SC` that aims at changing the status of op' to $simulating$, $\langle modifying, \_, \_\rangle$ or $done$ respectively. On line 53, the $output$ of op' is changed. Any of these steps, if applied by p, is preceded by an `LL`($opptr' \rightarrow status$) by p (on lines 26, 31, 51 or 62), 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 PERFORM(op') has not yet returned. Hence, $I_{op} \cap I_{op'} \neq \emptyset$.

In the remaining case, p writes $opptr'$ to $opptr \rightarrow tohelp[p']$ on line 82 or applies `SC`($opptr' \rightarrow, \langle restart, \_\rangle$) on line 84. Here also, before these steps, an `LL`($opptr' \rightarrow status$) by p occurs (on line 77) and this `LL` returns a value $\neq done$. As above, we then conclude that $I_{op} \cap I_{op'} \neq \emptyset$. □

Lemma 47. *If p applies a primitive to a* `direc` *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* Help*($opptr'$) where $opptr'$ is pointing to op'.*

Proof. Let x denote a `direc` 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 ANNOUNCE($opptr', x$) (lines 66, 68, 69, or 71), in a call to CONCURRENTACCESSES($opptr', x$) (line 75) where $opptr'$ is pointing to some operation op', or in the simulation of READDI(x) on behalf of op' (line 39). Each of these accesses to x occurs after p has called ANNOUNCE($opptr', x$). Therefore, by Lemma 42, $x \in DS(op')$. Moreover, before applying any of these steps, p has verified that the status op' is $\neq done$ (by applying a `LL` on $opptr' \rightarrow status$ on line 26, 31 or 51). More precisely, consider the last configuration C at which p applies `LL`($opptr' \rightarrow status$) before accessing x. Such a step occurs since the first step following a call to HELP($opptr'$) is a `LL` on $opptr' \rightarrow status$ (line 26). This last `LL` must returns $simulating$ since p has to pass the test on line 32 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 55, 56, 57, 58, 59 or 60). Before applying any of these steps, p performs a LL(*opptr*$' \rightarrow status$) (on lines 26, 31 or 51), where *opptr*$'$ is pointing to *op*$'$, which returns $\langle modifying, chgs', \_\rangle$ since the test on line 52 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 40 or 45) when an instruction of *op*$'$ simulated. Therefore, it follows from Theorem 33 that $x \in DS(op')$. □

Lemma 48. *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 21, line 23, line 29, line 79, line 85 or line 86. If p calls HELP(*opptr*$'$) on line 21, $op' = op$ and the Lemma holds.

If p calls HELP(*opptr*$'$) on line 23, a concurrent accesses 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 `direc` x such that q calls CONCURRENTACCESSES(*opptr*, x) and, before returning from that call, writes *opptr*$'$ to $opptr \rightarrow tohelp[p]$ (line 82), where *opptr* is pointing to *op*. By the code, before calling CONCURRENTACCESSES(*opptr*, x), q verifies that the status of *op* is $simulating$ by applying a LL on $opptr \rightarrow status$. Denote by C_{LL} the last configuration that precedes the call to CONCURRENTACCESSES(*opptr*, x) at which a LL($opptr \rightarrow status$) is applied by q. $opptr \rightarrow status = simulating$ at C. Moreover, it follows from the code of CONCURRENTACCESSES that before writing to $opptr \rightarrow tohelp[p]$, q performs a successful VL($opptr \rightarrow status$) on line 78. Let C_{VL} denote the configuration at which this step is applied. By observation 44, $opptr \rightarrow status = simulating$ in C_{VL} and has not changed since C_{LL}. In its previous step, q reads $opptr' \rightarrow status$ (line 77), and the value it gets back is $simulating$, since the test on line 80 is later passed. Therefore, there exists a configuration between C_{LL} and C_{VL} in which $opptr' \rightarrow status = simulating$, from which we conclude that $I_{op} \cap I_{op'} \neq \emptyset$.

HELP(*opptr*$'$) is called on line 29. As in the previous case, a process q' performs the successful SC that changes $opptr \rightarrow status$ to $\langle restart, opptr'\rangle$ (on line 84). This occurs when q' is executing CONCURRENTACCESSES(*opptr*$'$, x) for some `direc` 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 $opptr \rightarrow status = opptr' \rightarrow status = simulating$, from which it follows that $I_{op} \cap I_{op'} \neq \emptyset$.

Otherwise, process p calls HELP(*opptr*$'$) on line 79, 85 or 86. Before calling HELP(*opptr*$'$) on any of these lines, p has read the status of *op*$'$ (`LL`(*opptr*$' \rightarrow status$) on line 77), and this `LL` returns a value $\neq$ *done* (By the tests on line 79 or line 80, $opptr' \rightarrow status$ has to be $simulating$ or $\langle modifying, \_, \_\rangle$ in order for p to call HELP(*opptr*$'$) on line 79, 85 or 86). As this occurs before p returns from the call of PERFORM(*op*), $I_{op} \cap I_{op'} \neq \emptyset$. □

Lemma 49. *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 CONCURRENTACCESSES(*oppptr*$, x$) where x is a `direc` and *opptr* is pointing to *op*. Since every call to CONCURRENTACCESSES(*oppptr*$, x$) is preceded by a call to ANNOUNCE(*opptr*$, x$) (lines 37 and 38), it follows from Lemma 42 that $x \in DS(op)$. *op*$'$ is accessed by p via the announce array $x \rightarrow A$. Hence *op*$'$ has been announced to x and thus by corollary 43, $x \in DS(op')$. □

Theorem 50. *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 `direc`, 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 `direc`. 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 HELP(). That is, there exists an operation op_1 such p calls 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 HELP that are initiated after the call to 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 HELP($opptr_i$) and p' calls 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 HELP($opptr_i$), and before returning from this call, p calls directly HELP($opptr_{i-1}$). Similarly, after calling HELP($opptr'_{i'}$), and before returning from this call, p' calls directly HELP($opptr'_{i'-1}$).

It thus follows from the second property that for each $i, 2 \leq i \leq k$, HELP($opptr_{i-1}$) is called directly in an attempt of op_i, from which we derive by Lemma 45 that $DS(op_i) \cap DS(op_{i-1}) \neq \emptyset$. Moreover, it follows from Lemma 48 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 46. p applies a primitive to op^* after calling HELP($opptr_1$) and before returning from this call. Moreover, when this step is applied, every call to HELP() by p that follows the call of HELP($opptr_1$) has returned. Hence by Lemma 49, $op_1 = op^*$ 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 23, 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 HELP($opptr^*$), where $opptr^*$ is pointing to op^*, it follows from Lemma 48 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 82). This occurs while

CONCURRENTACCESSES(*opptr*, x) is executed, for some `direc` x. By Lemma 42 and the fact that the call CONCURRENTACCESSES(*opptr*, x) is preceded by a call to ANNOUNCE(*opptr*, x), $x \in DS(op)$. Moreover, by the code of CONCURRENTACCESSES(), op^* has been announced in to x, and thus by corollary 43, $x \in DS(op^*)$. 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 `direc` x^*. By Lemma 47, there exists op_1, op'_1 such that (1) p calls HELP(op_1) and p' calls 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 $CG(op, op')$. □

4.5 The TI-DAP-UC Universal Construction

It is an extension of the universal construction DAP-UC presented in Section 4.3. The additions to DAP-UC are highlighted in the code (Figures 4.8, 4.9, and 4.10).

Recall that DAP-UC does not ensure wait-freedom when it is applied to data structures on which each operation can access an unbounded number of different data items. To overcome this limitation, TI-DAP-UC enhances DAP-UC in the following ways. When p invokes an operation *op*, it acquires a new timestamp by calling `getTimestamp`. The timestamp and all entry points of *op* are stored in the data record v_x of each data item x created by *op*. Static data items have timestamp 0 and entry point `null`. The first time *op* accesses a data item x, it announces itself in v_x and then checks whether the timestamp of x

is larger than the timestamp of *op*. If so, the execution interval of *op* overlaps with the execution interval of the operation op' that created x, and *op* announces itself in the data record of each entry point to the data structure used by op'. Any successive operation that uses any one of these entry points will detect a concurrent access with *op* and help it to complete, in accordance with the priority scheme used in DAP-UC. We assume an upper bound on the number of entry points to the data structure. Therefore, each operation is initiated with a finite number of entry points. Moreover, since the state of the data structure is finite when an operation *op* is initiated, *op* accesses a finite number of dynamic data items before it is announced in the data record of each entry point used by each operation that creates after the initiation of *op*, a data item that *op* accesses. Each operation invoked after this point will either not contend with *op* or will help *op*, if it is not yet completed.

In the singly-linked list, suppose a SEARCH accesses a data item that was created by an APPEND operation op', which was invoked after the SEARCH. Then the SEARCH is announced in the data record, v_{last}, for the pointer to the last element in the list. Hence, the next APPEND invoked by each process q will help the SEARCH to complete, if the SEARCH is still in progress.

Finally, we prove that our algorithm does not violate timestamp-ignoring disjoint-access parallelism. The difficult case is when *op* is an operation that accesses a data item x created by an operation op' with a larger timestamp. Then *op* announces itself in the data records of the entry points used by op'. Let op'' be any operation that accesses one of these entry points y. Because op' is concurrent with *op*, its execution interval overlaps the minimum execution interval containing the execution intervals of *op* and op''. Thus, op' belongs to CG. Since op' accesses both y and x, there is an edge between op' and *op* and an edge between op' and op'' in CG. Thus, there is a path between *op* and op'' in CG.

Theorem 51. *The* TI-DAP-UC *universal construction (Figures 4.8, 4.9, and 4.10) produces timestamp-ignoring disjoint-access parallel, wait-free, concurrent data structures when applied to sequential data structures with a bounded number of entry points.*

```
1   type direc
        value $val$
        tmval $tm$
3       set of ptr to direc
        $pvar$
4       ptr to oprec $A[1..n]$

5   type statrec
6       $\{\langle st : simulating \rangle$,
7        $\langle st : restart$, ptr to oprec $restartedby\rangle$,
8        $\langle st : modifying$, ptr to dictionary of dictrec $changes$, value $output\rangle$
9        $\langle st : done\rangle\}$

10  type oprec
11      code  $program$
12      process id  $owner$
13      value $input$
14      value $output$
        tmval $tm$
15      set of ptr to direc
        $pentry$
16      statrec $status$
17      ptr to oprec $tohelp[1..n]$

18  type dictrec
19      ptr to direc $key$
20      value $newval$


21  value PERFORM($prog, input$) by process $p$:
22      $opptr$ := pointer to a new oprec record
        $opptr \rightarrow program := prog$, $opptr \rightarrow input := input$, $opptr \rightarrow output := \bot$
        $\boxed{opptr \rightarrow tm := \texttt{getTimestamp}(),\ opptr \rightarrow pentry := input.entry}$
        $opptr \rightarrow owner := p$, $opptr \rightarrow status := \langle simulating \rangle$,
        $opptr \rightarrow tophelp[1..n] := [nil, \ldots, nil]$
23      HELP($opptr$)                              /* $p$ helps its own operation */
24      for $q := 1$ to $n$ excluding $p$ do       /* $p$ helps operations that have been restarted by its operation $op$ */
25          if ($opptr \rightarrow tohelp[q] \neq nil$) then HELP($opptr \rightarrow tohelp[q]$)
26      return ($opptr \rightarrow output$)
```

Figure 4.8: Type Definitions and the Code of PERFORM of TI-DAP-UC

```
HELP($opptr$) by process $p$:
    $opstatus := \texttt{LL}(opptr \rightarrow status)$
    while ($opstatus \neq \langle done \rangle$)
        if ($opstatus = \langle restart, opptr' \rangle$) then            /* $op'$ has restarted $op$ */
            HELP($opptr'$)                                               /* first help $op'$ */
            $\texttt{SC}(opptr \rightarrow status, \langle simulating \rangle)$   /* try to change the status of $op$ back to $\langle simulating \rangle$ */
            $opstatus := \texttt{LL}(opptr \rightarrow status)$
        if ($opstatus = \langle simulating \rangle$) then                 /* start a new simulation phase */
            $dict$ := pointer to a new empty dictionary of dictrec records
            $ins$ := first instruction in $opptr \rightarrow program$
            while $ins$ is not a return do                               /* simulate instruction $ins$ of $op$ */
                if (($ins$ is WRITEDI($x, v$) or READDI($x$)) and         /* first access of $x$ by
                  (there is no dictrec with $key$ $x$ in $dict$)) then      this attempt of $op$ */
                    | if ($opptr \rightarrow tm < x \rightarrow tm$) then                         |
                    |     for each $y$ in $x \rightarrow pvar$ do ANNOUNCE($opptr, y$)            |
                    ANNOUNCE($opptr, x$)                                 /* announce that $op$ is accessing $x$ */
                    CONCURRENTACCESSES($opptr, x$)                       /* possibly, help or restart other operations accessing $x$ */
                    if ($ins$ = READDI($x$)) then $v := x \rightarrow val$
                    add new dictrec $\langle x, v \rangle$ to $dict$      /* create a local copy of $x$ */
                else if ($ins$ is CREATEDI()) then
                    $x$ := pointer to a new direc record
                    $\boxed{x \rightarrow tm := opptr \rightarrow tm,\ x \rightarrow pvar := opptr \rightarrow pentry}$
                    $x \rightarrow A[1..n] := [nil, \ldots, nil]$
                    $x \rightarrow A[opptr \rightarrow owner] := opptr$
                    add new dictrec $\langle x, nil \rangle$ to $dict$
                else /* either $ins$ 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 ($\neg\texttt{VL}(opptr \rightarrow status)$ then break  /* end of the simulation of $ins$ */
                $ins$ := next instruction of $opptr \rightarrow program$
            if ($ins$ is return ($v$)) then                              /* $v$ may be empty */
                       /* try to change status of $op$ to modifying; it is successful iff simulation is over and status of $op$ unchanged
                $\texttt{SC}(opptr \rightarrow status, \langle modifying, dict, v \rangle)$       since beginning of simulation */
            $opstatus := \texttt{LL}(opptr \rightarrow status)$
        if ($opstatus = \langle modifying, changes, out \rangle$) then
            $opptr \rightarrow outputs := out$
            for each dictrec $\langle x, v \rangle$ in the dictionary pointed to by $changes$ do
                $\texttt{LL}(x \rightarrow val)$ /* try to make writes visible */
                if ($\neg\texttt{VL}(opptr \rightarrow status)$) then return    /* $op$ is completed */
                $\texttt{SC}(x \rightarrow val, v)$

                $\texttt{LL}(x \rightarrow val)$
                if ($\neg\texttt{VL}(opptr \rightarrow status)$) then return    /* $op$ is completed */
                $\texttt{SC}(x \rightarrow val, v)$

            $\texttt{SC}(opptr \rightarrow status, \langle done \rangle)$
            $opstatus := \texttt{LL}(opptr \rightarrow status)$
    return
```

Figure 4.9: The Code of HELP of TI-DAP-UC

```
ANNOUNCE($opptr, x$) by process $p$:
    $q := opptr \rightarrow owner$

    LL($x \rightarrow A[q]$)
    if ($\neg$ VL($opptr \rightarrow status$)) then return
    SC($x \rightarrow A[q], opptr$)

    LL($x \rightarrow A[q]$)
    if ($\neg$VL($opptr \rightarrow status$)) then return
    SC($x \rightarrow A[q], opptr$)

    return

CONCURRENTACCESSES($opptr, x$) by process $p$:
    for $q' := 1$ to $n$ excluding $opptr \rightarrow owner$ do
        $opptr' :=$ LL($x \rightarrow A[q']$)
        if ($opptr' \neq nil$) then                    /* $op$ may concurrently access $x$ with $op'$ */
            $opstatus' :=$ LL($oppptr' \rightarrow status$)

            if ($\neg$VL($opptr \rightarrow status$)) then return

            if ($opstatus' = \langle modifying, -, - \rangle$) then HELP($opptr'$)

            else if ($opstatus' = \langle simulating \rangle$) then
                if ($opptr \rightarrow owner < q'$) then        /* $op$ has higher priority than $op'$, restart $op'$ */
                    $opptr \rightarrow tohelp[q'] := opptr'$

                    if ($\neg$VL($opptr \rightarrow status$)) then return
                    SC($opptr' \rightarrow status, \langle restart, opptr \rangle$)

                    if (LL($oppptr' \rightarrow status$) = $\langle modifying, -, - \rangle$) then HELP($opptr'$)

                else HELP($opptr'$)                             /* $op$ has lower priority that $op'$, help $op'$ */
    return
```

Figure 4.10: The Code of ANNOUNCE and CONCURRENTACCESSES of TI-DAP-UC

Chapter 5

The WFR-TM Software Transactional Memory Algorithm

5.1 General

In this chapter, we introduce an STM algorithm that guarantees wait-freedom for read-only transactions, called *Wait-Free Read-only Transactional Memory* or WFR-TM. For read-intensive workloads it is important to ensure that read-only transactions never abort and are wait-free. In Section 5.2, the main ideas of WFR-TM are presented. Also, in Sections 5.3 and 5.4, the pseudocode of WFR-TM is described. Finally, in Section 5.5, the proof of correctness and progress properties ensured by WFR-TM are presented.

5.2 Main Ideas

Each transaction T starts by announcing itself into an appropriate element of an announce array; this array has size n, with one entry for each process, used by the corresponding process to announce its transactions. Then, during its *simulation phase*, it speculatively executes its code while it keeps track of the data items it accesses by maintaining two sets implementing its read-set and its write-set. For each data item read by T, its (implementation of) read-set contains the value read; similarly, for each data item written by T, its (implementation of) write-set contains the value that T wants to write to it. At commit time, an update transaction also executes an *updating phase*, where it actually updates the data items in its write-set, and a *waiting phase*, where it waits for live announced read-only transactions to commit; the latter phase is needed to ensure wait-freedom for read-only transactions.

Update transactions employ fine-grained locking to ensure consistency when updating data items. Specifically, at commit time and before entering its updating phase, an update transaction T_w attempts to obtain the locks that are associated with each data item in its read-set and its write-set. In order to avoid deadlocks, the locks are acquired in ascending order based on the address of the data item. Consider any data item x in the write-set of T_w and let e be the entry for x in T_w's write-set. After acquiring the lock of x, T_w adds in e the value that x has at the time locked by T_w. Once T_w acquires all the required locks, it first enters its updating phase, then it enters its waiting phase, and finally it releases all the acquired locks and commits.

If an update transaction enters its updating phase, WFR-TM guarantees that it will commit in a finite number of steps. Also, WFR-TM guarantees that any read-only transaction that does not crash commits in a finite number of steps; i.e. it guarantees wait-freedom for read-only transactions.

For each transaction T, WFR-TM maintains a record that contains: i) the status of T, a variable that represents the current state of T and can take the values *simulating*, *updating*, *waiting*, *committed* or *aborted*, ii) the read-set and write-set of T, and iii) a set called $beforeMe$ of live transactions that will be linearized before T, which is used in order to ensure consistency of reads, as described below.

For each data item x, WFR-TM maintains a record that contains: i) the current value of x, ii) its version which is a strictly increasing sequential number, updated whenever the value of x is updated, and iii) a pointer $owner$ to some transaction's record which indicates whether x is locked. An update transaction T_w *acquires* the lock of x each time it successfully executes a `CAS` to identify itself as the owner of x; x is considered to be *unlocked* if either the owner field of its record is `null` or the status of the transaction that it points to is aborted or committed. T_w *releases* all the locks it has acquired in a single step, by updating its status to either committed or aborted (using a `write` primitive).

In order to provide wait-freedom for read-only transactions, WFR-TM ensures that each read-only transaction T_r reads consistent values independently of whether the data items that it accesses are locked or not, as follows. When a data item x is unlocked, T_r reads its value directly from the record maintained for x. Suppose now that x is locked by some update transaction T_w at some point. We define an old value and a new value for x at that point. The *old value* of x is the value stored in x's record at the moment that it was locked by T_w, whereas the *new value* of x is the value that T_w wants to write to x. Notice that the old value of x is contained it its record until T_w actually writes the new value for it, during its updating phase; afterwards, the old value is recorded in the write-set of T_w.

During its initialization, each transaction T takes a *snapshot* of the announce array; this snapshot is a consistent view of the announced transactions together with their statuses. Using this snapshot, T decides whether it must read or ignore the new values written by live update transactions. Specifically, T adds into the $beforeMe$ set all those announced transactions whose status is waiting or committed and ignores any new values written by transactions not contained it this set, as described below. Before committing, each update transaction reads all entries of the announce array and *waits* for the completion of each announced read-only transaction that it encounters. By incorporating this *waiting mechanism*, WFR-TM ensures that if a read-only transaction T_r ignores the value written to a data item by an update transaction T_w, then T_w does not commit before T_r has committed. This is necessary to argue that at the time that T_r commits, it will not have read an inconsistent set of values. This ensures consistency of read-only transactions. We also remark that this waiting mechanism provides the wait-free property to read-only transactions.

When some transaction T accesses a data item locked by some update transaction T_w, it checks if T_w is in T's $beforeMe$ set. If $T_w \in beforeMe$, then T does not ignore the new value of x written by T_w and reads it directly from the record of x; this value is the new value written by T_w. Otherwise, if $T_w \notin beforeMe$, T ignores the new value written from T_w and decides from where to read the old value of x based on the phase of T_w. If T_w is in its simulating phase, then T reads the old value from x's record, since T_w has not yet started updating its data items. If T_w is in its updating phase, then T reads the old value of x from T_w's write-set. Notice that this is necessary since in this case, T_w is in the process of updating the data items contained in its write-set, so some of them may contain the new values and some of them may still contain the old values; so, if for instance the read-set of T contains two data items x and y updated by T_w, and T reads both of them from their records, it may read the old value for x and the new value for y, which would be inconsistent. The same action is taken by T when T_w is in its waiting phase, since similar consistency problems could appear if T has read other data items written by T_w while T_w was in its updating phase.

For each data item x, there is a version that is associated to it whose value is unique for each value stored in x. An update transaction T_w performs its reads by executing the same actions described above for read-only transactions. Additionally, since the waiting mechanism is not employed between update transactions, in order to ensure opacity, T_w must validate its read-set whenever it reads a data item for the first time, as well as a final time before it starts its updating phase. Specifically, T_w validates the read-set by comparing the current version of each data item contained there in, against the version that T_w last read for this data item (which is contained in its read-set). T_w aborts if a mismatch is found for some data item. We remark that, T_w performs a final (indirect) validation by acquiring the lock of each data item contained in its read-set. If a version mismatch is found, the `CAS` used to acquire the lock of the corresponding data item, fails, and T_w aborts.

5.3 Type Definitions

Figure 5.1 presents the data structures of WFR-TM.

For each transaction T, WFR-TM stores a record of type `txrec` that contains: 1) the identifier pid of the process that initiated T, 2) a three-bit variable $status$, storing the

```
typedef statval {SIMULATING, UPDATING, WAITING, COMMITTED, ABORTED}

type txrec
    unsigned int $pid$
    statval $status$
    set of wnode elements $wset$
    set of pointers to
        txrec elements $beforeMe$

type direc
    value $val$
    unsigned int $ver$
    txrec *$owner$

type rnode
    direc *$tvar$
    value $val$
    unsigned int $ver$

type wnode
    direc *$tvar$
    value $oldval$
    unsigned int $oldver$
    value $newval$

/* Shared variable */
shared txrec *$A[1..n]$

/* Persistent local variable for process $p$ */
set of rnode elements $rset_p$
```

Figure 5.1: Data structures of WFR-TM.

status of T, 3) a set $wset$ of `wnode` elements, implementing the write-set of T, 4) a set $beforeMe$ of pointers to elements of type `txrec`. Also, each process maintains a local set $rset$ of `rnode` elements, implementing the read-set of the transactions it initiates.

For each data item x, WFR-TM stores a `CAS` object of type `direc`, containing: i) the value val of x, ii) the version ver of x, an unsigned integer, and iii) a pointer $owner$ to a `txrec` record. To implement WFR-TM with single-word `CAS` objects, indirection can be used (as in [14, 18]).

We remark that an element of type `rnode` contains: i) a pointer $tvar$ to the `direc` record of x, ii) the value val of x read by T, and iii) an unsigned integer value ver representing the version of x read by T. Moreover, an element of type `wnode` contains: i) a pointer $tvar$ to the `direc` record of x, ii) the (old) value $oldval$ of x; this field is initialized to $\perp$ and it is set to the value of x at the time x is locked by T, iii) an unsigned integer $oldver$ representing the (old) version of x; this field is also initialized to $\perp$ and it is set to the version of x at the time x is locked by T, and iv) the value $newval$ that T will store into x.

Finally, A is the announce array maintained by WFR-TM. Initially, each entry of A points to a dummy `txrec` record whose $status$ field is equal to `COMMITTED` and its $wset$ set is empty. Also, for each data item x, the fields of the `direc` record of x have the following values: i) val contains an initial value, ii) ver is equal to 0, and iii) $owner$ points to a dummy `txrec` record whose *status* field is equal to `COMMITTED`.

5.4 The Code of the WFR-TM Algorithm

The pseudocode of WFR-TM is provided in Figures 5.2 and 5.3[1].

BEGINTX. When called by process p for transaction T, it creates (line 24) and initializes (lines 25 - 30) the `txrec` record of T, and then announces T in $A[p]$. Finally, it calls CHECKIFPERFORMED to appropriately initialize the *beforeMe* set of T (line 31). Each iteration of the while loop of CHECKIFPERFORMED, reads all elements of A (lines 35 - 36) and adds to T's *beforeMe* (line 38) new update transactions (i.e. those that are not already in *beforeMe*) whose status is either waiting or committed (line 37). A new iteration will start if some transaction is added to *beforeMe* in the current iteration. This procedure guarantees that at the beginning of the last execution of the `for` of line 35, i.e. the `for` executed during the last execution of the `do while` of lines 34 to 39 *beforeMe* contains a consistent snapshot of the announced transactions that have entered their waiting phase. We now explain why CHECKIFPERFORMED terminates within a finite number of steps. Any transaction T' that is announced after the announcement of T cannot commit before CHECKIFPERFORMED completes, given that even if T' reaches its commit phase, T' will consider T as a read-only transaction (since T has an empty write-set as long as it executes CHECKIFPERFORMED), so T' will wait for T to either terminate or become an update transaction. This ensures that only a limited number of new transactions can appear while CHECKIFPERFORMED is executed, which in turn ensures that CHECKIFPERFORMED returns in a finite number of steps.

CREATEDI. When called by process p for transaction T, it creates, initializes (line 41) and returns (line 42) a new `direc` record for the newly allocated data item.

READDI. When called by T to read the value of some data item x, it first checks if there is an entry for x in the write-set or in the read-set of T. If this is the case, READDI returns the value from there (to ensure consistency). Otherwise, the value of x is determined on lines 48-51.

[1]Notice that in the algorithm pseudocode of this section, the abort response is modelled by boolean value `false`, while the commit response is modelled by boolean value `true`.

```
txrec *BeginTx() by process $p$:
    txrec *$newTx$ := new txrec
    $newTx \rightarrow pid := p$
    $newTx \rightarrow status =$ SIMULATING
    $newTx \rightarrow wset :=$ empty set of wnode elements
    $newTx \rightarrow beforeMe :=$ empty set of pointers to txrec elements
    $rset_p :=$ empty set of rnode elements

    $A[p] := newTx$                          /* $T$ announces itself */
    CheckIfPerformed($newTx$)                /* $T$ initializes its $beforeMe$ set */
    return ($newTx$)

CheckIfPerformed(txrec *$newTx$) by process $p$:
    do
        for $i = 1$ up to $n$, excluding $p$, do
            $tran := A[i]$
                /* check if $tran$ is an update transaction not in $newTx$'s $beforeMe$ set that has entered its waiting phase,
            if ($tran \notin newTx \rightarrow beforeMe$ AND $tran \rightarrow wset \neq \emptyset$ AND
                $tran \rightarrow status \in \{$WAITING, COMMITTED$\}$) then
                add $tran$ in $newTx \rightarrow beforeMe$          then add it once to $beforeMe$ */
    while a new element is added in $newTx \rightarrow beforeMe$

direc *CreateDI(txrec *$tx$) by process $p$:
    direc $newTvar$ := new direc $\langle \perp, 0, tx \rangle$
    return ($newTvar$)

$\langle$boolean, value$\rangle$ *ReadDI(txrec *$tx$, direc *$tvar$) by process $p$:
    if an element $el$ with $el.tvar = tvar$ exists in $tx \rightarrow wset$ then
        return $\langle$true, $el.newval\rangle$
    if an element $el$ with $el.tvar = tvar$ exists in $rset_p$ then
        return $\langle$true, $el.val\rangle$

    $\langle val, ver, owner\rangle := *tvar$
    $status := owner \rightarrow status$
                /* if $tvar$ is locked by a transaction $T_w$ that is not to be linearized before $tx$ and $T_w$
                is in its updating or waiting phase, then read the old value of $tvar$ from $T_w$'s write-set */
    if (an element $el$ with $el.tvar = tvar \in owner \rightarrow wset$ AND
        $owner \notin tx \rightarrow beforeMe$ AND $status \neq$ SIMULATING) then
        $\langle val, ver\rangle := \langle el.oldval, el.oldver\rangle$

    add $\langle tvar, val, ver\rangle$ in $rset_p$

    if ($tx \rightarrow wset \neq \emptyset$ AND Validate($tx$) = false) then   /* call Validate to ensure opacity */
        $tx \rightarrow status =$ ABORTED
        return $\langle$false, $\perp\rangle$

    return $\langle$true, $val\rangle$

boolean Validate(txrec *$tx$) by process $p$:
    for each element $el$ in $rset_p$
        $\langle val, ver, owner\rangle := *el.tvar$
        if ($ver \neq el.ver$) then return false
    return true
```

Figure 5.2: Pseudocode for BeginTx, CheckIfPerformed, CreateDI, ReadDI, and Validate of WFR-TM

```
boolean **WRITEDI** (txrec *$tx$, direc *$tvar$, value $value$) by process $p$:
    if an element $el$ with $el.tvar = tvar$ exists in $tx \rightarrow wset$ then
        update $el.newval$ with $value$
    else add $\langle tvar, \perp, \perp, value \rangle$ in $tx \rightarrow wset$
    return true

boolean **COMMITTX**(txrec *$tx$)by process $p$:
    if ($tx \rightarrow wset =$ null) then          /* if $tx$ is read-only, commit */
        $tx \rightarrow status :=$ COMMITTED
        return true

    if (LOCKDATASET($tx$) $=$ false) then    /* if locking of some data item fails, abort */
        $tx \rightarrow status :=$ ABORTED
        return false

    $tx \rightarrow status :=$ UPDATING            /* $tx$ enters updating phase */
    for each element $el$ in $tx \rightarrow wset$ do
        CAS(*$el.tvar$,*$el.tvar$, $\langle el.newval, el.tvar \rightarrow ver + 1, tx \rangle$)
                                   /* write here would also do; we use CAS to be coherent with our model */

    $tx \rightarrow status :=$ WAITING             /* $tx$ enters waiting phase */
    WAITREADERS($tx$)                              /* $tx$ waits announced read-only transactions */

    $tx \rightarrow status :=$ COMMITTED           /* $tx$ commits */
    return true

boolean **LOCKDATASET** (txrec *$tx$) by process $p$:
    for each element $el$ in $tx \rightarrow wset \cup rset_p$, in ascending order (based on $tvar$ field)
        if $\exists$ an element $el' \in rset_p$ with $el'.tvar = el.tvar$ then
                      /* if $tx$ has read the $tvar$ before, use this old value for consistency */
            $\langle val, ver, owner \rangle := \langle el'.val, el'.ver, el'.tvar \rightarrow owner \rangle$
                      /* otherwise, if the $tvar$ was not read before, use the current value as old value */
        else $\langle val, ver, owner \rangle :=$ *$(el.tvar)$

        if ($owner \rightarrow status \notin \{$COMMITTED, ABORTED$\}$)     /* $el.tvar$ is locked */
            if $\exists$ an element $el'' \in owner \rightarrow wset$ with $el''.tvar = el.tvar$ then
                return false   /* if it is in the write-set of $owner$, locking fails; otherwise, wait until it is unlocked1 */
            else wait until $owner \rightarrow status \in \{$COMMITTED, ABORTED$\}$
                                                                  /* **l-cas**: try to lock $el.tvar$ */
        if (CAS(*$el.tvar$, $\langle val, ver, owner \rangle$, $\langle val, ver, tx \rangle$) = false) then
            return false
                      /* if $el$ is written by $tx$, then maintain the old value of $el.tvar$ */
        if ($el \in tx \rightarrow wset$) then update $\langle el.oldval, el.oldver \rangle$ with $\langle val, ver \rangle$

    return true

**WAITREADERS**(txrec *$tx$) by process $p$:
    for $i = 1$ up to $n$, excluding $p$, do
        $tran := A[i]$
        if ($tran \neq$ null AND $tran \rightarrow wset =$ null) then
            wait until ($tran \rightarrow status =$ COMMITTED OR $tran \rightarrow wset \neq$ null)
```

Figure 5.3: Pseudocode for WRITEDI, COMMITTX, LOCKDATASET, and WAITREADERS of WFR-TM

Initially, the value $\langle val, ver, owner \rangle$ of x's `direc` record (line 48) and the status of x's *owner* (line 49) are read. Assume first that x is not locked. Then, the value for x that T returns is val, as read on line 48. Assume now that x is locked by a transaction T_w. If the status of T_w is simulating, then again the value for x that T returns is val. Otherwise, the status of T_w is either updating or waiting or committed, and the first and third condition of line 50 evaluate to `true`. Recall that we consider that x has an *old value* and a *new value*, which are stored in T_w's write-set entry for x (specifically, in fields $oldval$ and $newval$ of this entry, respectively). If T_w is contained in T's *beforeMe* set, i.e. the second condition of line 50 evaluates to `false`, then T_w's update on x has already been performed before the beginning of T. Therefore, again the value for x that T should read is val. However, if T_w is not contained in T's *beforeMe* set, then T should not read T_w's update on x, i.e. the new value of x, and should instead read the old value of x; this value is read on line 51.

After T determines the value to read for x, it adds it together with its corresponding version in its read-set (line 52). In case T is an update transaction, then its read-set is validated by calling VALIDATE (line 53); VALIDATE (lines 58-61) returns `true` when no version of the elements in T's read-set has changed; it returns `false` otherwise. We remark that this validation mechanism can also be implemented using a timestamping mechanism as that presented in TLII [29] or LSA [30], to boost performance.

WRITEDI. When called by T_w to update some data item x with value val, T_w first checks whether it has previously invoked WRITEDI to modify x. If this is so, then there is already an element for x in T_w's write-set (line 63) and WRITEDI updates to val the $newval$ field of this element (line 64). Otherwise, a new `wnode` element for x is added in T_w's write-set (line 65).

Recall that when T_w enters its updating phase, the $oldval$ and $oldver$ fields of x's `wnode` must contain the value and version, respectively, written by the transaction for which it holds that it had x in its write-set and was the last to commit before T_w's acquisition of the lock of x (or the initial values if such a transaction does not exist). WFR-TM allows another transaction T' to snoop into T_w's write-set (line 51) in order to read the old value of some data item contained there. Therefore, T_w's write-set must offer a way to T' to read values that are mutually consistent. To achieve this, WRITEDI sets the $oldval$ and $oldver$ fields of new `wnode` elements that are added in a write-set to be equal to $\perp$ (line 65). This is necessary for avoiding bad scenarios such as the following: Apart from x, assume that T_w wants to write also another data item y and let C be a configuration at which T_w has written

x but not yet y. Thus, T has created a write set entry for x, but there is no such entry in T's write-set for y. T_w has also read (before C) the contents of x's `direc` to store in the *oldval* and *oldver* fields of x's `wnode`. Now, let another transaction T'' lock and update both x and y, and commit. Then, T_w continues by writing y. So it places an entry in its write-set for y and reads the contents of y's `direc` to store in the *oldval* and *oldver* fields of this entry. Then, T_w acquires the locks of both x and y. So, if T' snoops both x and y from T_w's write-set, it reads inconsistent values.

COMMITTX. If T is a read-only transaction (its write-set is empty), COMMITTX changes T's status to committed and returns `true` (lines 68-70). If T is an update transaction, it attempts to acquire the required locks by calling LOCKDATASET (line 71), which is described in the next paragraphs. If it fails to acquire some lock (LOCKDATASET returns `false`), T is aborted (lines 71-73). Otherwise, all the required locks have been acquired (LOCKDATASET returns `true`). Then, T enters its updating phase (lines 74 - 76) and updates the data items in its write-set (line 76). Notice that it also increments the version of each data item by one. Afterwards, T enters its waiting phase (line 77) and waits until all announced read-only transactions commit. This is done by calling WAITREADERS (line 78). WAITREADERS goes through the announce array A, and waits until each active read-only transaction (line 97) either commits or turns out to be an update transaction (line 98).

LOCKDATASET is called by T to lock each data item in its read-set and write-set. Deadlocks are avoided by acquiring the locks in (ascending) order (based on the *tvar* pointer contained in each `rnode` or `wnode` element). Initially, LOCKDATASET determines the value and version of each data item x that it wants to lock, as follows: If x exists in T's read-set, these values are taken from the corresponding read-set entry (line 83). Otherwise, they are read from x's `direc` record (line 85).

LOCKDATASET tries to lock x using a `CAS` operation which stores a pointer to T's `txrec` record into the *owner* field of x's `direc` record (line 91). Notice that this `CAS` also serves as a final validation of the value of x read by T (in case x is in T's read-set). LOCKDATASET returns `true` only if it successfully locks all the data items in T's read-set and write-set (line 93). If x is already locked by some transaction T' (lines 86 to 88), LOCKDATASET by T returns `false`. If x is locked by some transaction that does not intend to update it, LOCKDATASET waits until this transaction completes (line 89). Finally, recall that when LOCKDATASET is invoked, the contents of the *oldval* and *oldver* fields of x's element in T's write-set are $\perp$. In case x is locked, these fields are updated with the determined current values for x (line 92), so that if T enters its updating phase these fields are appropriately set in each element of T's write-set.

5.5 Proof of the WFR-TM Algorithm

In this section, we prove that WFR-TM is opaque. We also study the progress properties of WFR-TM. In Section 5.5.1, we provide some preliminaries including useful notation, in Section 5.5.2, we present the full proof of correctness of WFR-TM, and in Section 5.5.3, we study its progress properties.

5.5.1 Preliminaries

The *execution interval* of some transaction T is denoted by α_T. The process p that initiates T is its *initiator*. Whenever p applies a primitive while executing T, we also say that T applies this primitive. We denote by CE_T the last configuration of α_T. We say that T *announces* itself when it executes the write to $A[p]$ of line 30.

By inspection of the code of WRITEDI (lines 63 - 65), T adds in its write-set a unique record for each data item that it writes. Moreover, by inspection of the code of READDI (lines 44 - 56), for each data item x read by T, T executes lines 48 - 56 during the first instance of READDI for x executed by T; we denote by $RT_{x,T}$ this instance. We remark that each subsequent instance of READDI for x executed by T returns either on line 45 or on line 47. So, by inspection of the code (line 52), T maintains in its read-set a unique record for each data item read by it.

Observation 1. *Consider any transaction T and let C be any configuration. Then,*

1. *if T has executed at least one instance of* WriteDI *for some data item x at C, there is a unique record for x in T's write set at C;*
2. *if T has executed $RT_{x,T}$ for some data item x at C, there is a unique record for x in T's write set at C; any instance of* ReadDI *for x by T following $RT_{x,T}$ does not execute lines 48 - 56.*

Each time T *successfully* executes the CAS instruction of line 90 for some data item x, we say that T *becomes the owner* of x or *acquires the lock* for x. We call the CAS instruction of line 90 *l-cas*. Since, LOCKDATASET is executed at most once (line 71) by

T, by inspection of the code (lines 82 - 91), at most one l-cas is executed for each data item in the data-set of T. Assume that T acquires the lock for x. We denote by $CL_{x,T}$ the configuration after the successful execution of the l-cas for x by T. Each time T executes the write instruction of line 76 for some data item x with values $\langle v, d \rangle$, we say that T *updates* the value and the version of x with v and d, respectively, or *writes* the value v and version d for x.

By inspection of the code (line 24), each transaction T is associated with a unique `txrec` record. Recall that the status of T is the value of the field $status$ in this record. Throughout this proof we abuse notation and we use the same notation to refer both to the name of some transaction and to its `txrec` record.

Fix any execution α of WFR-TM. Consider any transaction T in α. By inspection of the code (line 26), $T.status$ is initially `SIMULATING`. Notice that no other transaction can update the status of T. If T is read-only, by inspection of the code (lines 53, 56, and 68 - 70), its status can only change from `SIMULATING` to `COMMITTED` (line 69).

If T is an update transaction, then by inspection of the code (lines 54 - 55 and 72 - 73), its status may change from `SIMULATING` to `ABORTED`. Also, by inspection of the code (lines 74, 77 and 79), its status may change from `SIMULATING` to `UPDATING`, from `UPDATING` to `WAITING`, and from `WAITING` to `COMMITTED`. As long as its status is `SIMULATING`, `UPDATING`, or `WAITING`, we say that T is in its *simulating*, *updating*, or *waiting phase*, respectively.

Observation 2. *The following hold for each transaction T:*

1. *if T's status is* `SIMULATING`*, it can change either to* `ABORTED` *or to* `UPDATING`*;*
2. *if T's status is* `UPDATING`*, it can only change to* `WAITING`*;*
3. *if T's status is* `WAITING`*, it can only change to* `COMMITTED`.

If the status of T becomes `COMMITTED` or `ABORTED`, then it never changes again. If its status becomes `COMMITTED` or `ABORTED`, we say that T *completes* (*commits* or *aborts*, respectively). Notice that if a committed (aborted) transaction returns, it returns `true` (`false`). If T commits in α, we denote by CM_T the configuration after the execution of line 79 which changes T's status to `COMMITTED`. If T aborts in α, we denote by CA_T the configuration after the execution of the write instruction (line 54 or line 72) that changes the status of T to `ABORTED`. Notice that if T completes, then either $CE_T = CM_T$ or $CE_T = CA_T$, depending on whether T commits or aborts, respectively.

Table 5.1 briefly summarizes the notation introduced thus far, as well as some notation that will be introduced later. Note that notation that refers to some configuration starts with the letter C.

α_T	the execution interval of T
$RT_{x,T}$	the (first and) unique instance of READDI for x by T during which T executes lines 48 - 56 for x; x is globally read by T
CE_T	the last configuration of α_T
$CL_{x,T}$	the configuration after the successful execution of the l-cas for x by T (line 90)
CU_T	the configuration after the execution of line 74 that changes the status of T to `UPDATING`
CW_T	the configuration after the execution of line 77 that changes the status of T to `WAITING`
CM_T	the configuration after the execution of line 79, that changes the status of T to `COMMITTED`
CA_T	the configuration after the execution of the write instruction (lines 54 or 72) that changes the status of T to `ABORTED`.
$\alpha_{x,T}$	the execution interval of α_T during which T maintains the lock for x
CR_T	the configuration at the beginning of the last execution of the `for` of line 35 in CHECKIFPERFORMED by T
$RS_T(C)$	the set containing each triple $\langle x, v, d\rangle$ added to the set $rset$ (of the process executing T) until configuration C
RS_T	$RS_T(CE_T)$
C_T^w	the configuration preceding the first execution of line 65 by T, i.e. the point at which T adds its first element in its write set, thus indicating that it is an update transaction.
ℓ_C	the sequence of transactions of α that have been serialized before or at C

Table 5.1: Useful Notation for the Proof of Correctness

Consider any update transaction T_w. If T_w enters its waiting phase in α, we denote by CU_{T_w} and CW_{T_w} the configurations after the execution of lines 74 and 77, respectively, which change T_w's status to `UPDATING` and `WAITING`, respectively. By inspection of the code (lines 52, 68, 71, and 74), T_w calls LOCKDATASET before CU_{T_w} and this call returns `true` (i.e. it was successful). Thus, by inspection of the code (lines 52, 65, 82, 90, and 93), T_w has acquired the locks for all data items accessed by T_w before CU_{T_w}.

If T_w acquires the lock for some data item x, by inspecting the code it follows that at CL_{x,T_w} the status of T_w is equal to `SIMULATING`. We say that T_w *maintains* the lock for x, or x is *locked* by T_w, in each configuration following CL_{x,T_w} (including it) in which the status of T_w is neither `COMMITTED` nor `ABORTED`. The change of the status of T_w to `COMMITTED` or `ABORTED`, indicates that T_w *releases* all locks it has acquired. We denote by α_{x,T_w} the execution interval of α_{T_w} during which T_w maintains the lock for x. We remark that α_{x,T_w} starts with CL_{x,T_w} and, in case T_w completes in α, it ends with the configuration preceding CM_{T_w} or CA_{T_w} (depending on whether T_w commits or aborts respectively). If T_w does not complete in α, α_{x,T_w} is the suffix of α, starting at CL_{x,T_w}.

The inspection of the code (lines 71 - 77) and the definition of α_{x,T_w} imply the following.

Observation 3. *Consider any update transaction T_w that enters its waiting phase in α. For each data item x accessed by T_w, CU_{T_w} and CW_{T_w} occur in α_{x,T_w}.*

Lemma 4. *Consider any update transaction T_w that acquires the lock for some data item x. During α_{x,T_w}, the owner field of the* `direc` *record of x contains a pointer to the* `txrec` *record of T_w.*

Proof. By inspection of the code (line 90) and by the definition of CL_{x,T_w}, at CL_{x,T_w} the claim holds. Assume, by the way of contradiction, that there is some configuration in α_{x,T_w} in which the *owner* field of the `direc` record of x contains a pointer to the `txrec` record of a transaction $T'_w \neq T_w$. Let C be the first such configuration. By inspection of the code (line 90), T'_w has acquired the lock for x. Let l_{CAS} be the successful l-cas that T'_w executed in order to acquire the lock for x. Before executing $lcas$, T'_w reads the value $\langle -, -, owner \rangle$ from the `direc` record of x either on line 84 or on line 85; let r_x be this read. Notice that r_x is executed before the end of α_{x,T_w}.

To derive a contradiction, we consider the following cases. Assume first that r_x is performed after CL_{x,T_w}. In this case, the *owner* field of the `direc` record of x contains a pointer to the `txrec` record of T_w. By definition of α_{x,T_w}, $T_w.status \notin \{\texttt{COMMITTED}, \texttt{ABORTED}\}$ during α_{x,T_w}. So, by inspection of the code (lines 87 and 88), the instance of LOCKDATASET executed by T'_w returns `false`. Then, by inspection of the code (lines 71 - 73), T'_w aborts, so it does not attempt to lock x. This contradicts the assumption that T'_w has acquired the lock for x at C.

Assume now that r_x is performed before CL_{x,T_w}. Let r_x return $owner = T''_w$, where $T''_w \neq T_w$. Notice that $lcas$ can only succeed if the *owner* field of the `direc` record of x

contains a pointer to the `txrec` record of T''_w. However, since $lcas$ is the first successful `CAS` for x executed after CL_{x,T_w}, the $owner$ field of the `direc` of x contains a pointer to the `txrec` of T_w when $lcas$ is executed (and not to T''_w). It follows that $lcas$ does not succeed. This contradicts the definition of $lcas$. □

By Observation 3 and by inspection of the code (lines 74 - 77), it follows that any update transaction T_w updates each data item x in its write-set during α_{x,T_w}. Moreover, by inspection of the code (lines 71, 77, 82, 90, and 93), each update transaction that enters its waiting phase acquires the lock for each data item in its read-set and each data item in its write-set. Observation 3 and Lemma 4 imply the following.

Corollary 5. *Consider any update transaction T_w that enters its waiting phase in α. Consider any data item x in the read-set or the write-set of T_w. Then, during α_{x,T_w}, 1) T_w updates x during α_{x,T_w} and 2) for each update transaction $T'_w \neq T_w$ that updates x, α_{x,T_w} and α_{x,T'_w} do not overlap.*

The version of each data item changes only by executing the `CAS` of line 76, i.e. when an update transaction executes its updating phase. Corollary 5 and the inspection of the code (lines 41 and 76) imply the following claim.

Observation 6. *The version of each data item is strictly increasing.*

We continue to prove the following stronger claim.

Lemma 7. *Let $S = T_1T_2, \ldots$ be the sequence of update transactions that acquire the lock for some data item x in α. Then, for each integer d, $d > 0$, (1) T_d has a* `wnode` *element for x in its write set with value $d-1$ stored in its $oldver$ field, (2) T_d writes d as the new version for x, and 3) all but the last transaction in S enter their waiting phase.*

Proof. The proof is by induction on the value of d. Fix any $d > 0$ and assume that the claim holds for $d-1$. We prove that the claim holds also for d. Recall that T_d acquires the lock for x by successfully executing an l-cas for x (line 90). In case $d = 1$, then the initial value of the version of x is 0. In case $d > 1$, then by the induction hypothesis (claim 2), it follows that T_{d-1} writes the value $d-1$ as the new version of x. Since T_d is the first transaction to execute a successful l-cas for x after T_{d-1}, by inspection of the code (lines 83, 85, and 92) and Corollary 5, it follows that T_d uses $\langle -, d-1, T_w \rangle$ as the old value for its l-cas. by inspection of the code (lines 83, 84, 89, 90), it follows that $d - 1$ is the value that T_d stores as the $oldver$ field in the `wnode` for x in its write set. So, claim (1) holds.

Moreover, by inspection of the code (lines 76 and 79), T_d updates the version of x during α_{x,T_d}. By Lemma 4, it follows that T_d updates the version of x to d. So, claim (2) holds. Also, if T_d is not the last transaction of S, then Lemma 4 implies claim (3). $\square$

Corollary 8. *Let $S = T_1T_2, \ldots$ be the sequence of update transactions that acquire the lock for some data item x and enter their waiting phase in α. Then, for each $d > 0$, $CW_{T_d} < CW_{T_{d+1}}$, i.e. the serialization point of T_d precedes that of $T_d + 1$ in α.*

Consider any update transaction T_w that enters its waiting phase in α. Then, by inspection of the code (lines 77 and 78), if T_w calls WAITREADERS, it does so after CW_{T_w}. Consider also any read-only transaction T_r. By inspection of the code (lines 30, 77 - 78, and 95 - 98), if T_r performs its announcement before CW_{T_w}, T_w will wait (line 98) for T_r to commit. Therefore, in this case, T_r commits before the completion of T_w.

Lemma 9. *Consider any update transaction T_w that enters its waiting phase in α and any read-only transaction T_r that commits in α. If T_r performs its announcement before CW_{T_w}, then T_r commits before the completion of the waiting phase of T_w in α.*

In WFR-TM, we assign serialization points to read-only transactions that commit in α and to update transactions that enter their waiting phase in α. Consider any such transaction T in α. Let CR_T be the configuration at the beginning of the last execution of the `for` of line 35 in CHECKIFPERFORMED by T. Notice that CR_T is the configuration where the first iteration of the for of line 35 starts executing during the last execution of the `do while` of lines 34 to 39. If T is a read-only transaction, we place its serialization point at CR_T. If T is an update transaction that enters its waiting phase, we place its serialization point at CW_T. By the way serialization points are assigned, the serialization point of each transaction is placed in its execution interval. Moreover, at each configuration C, there is a sequence of transactions of α that have been serialized before or at C. Let ℓ_C denote this sequence.

Consider any transaction T in α and let C be any configuration. Let $RS_T(C)$ be the set containing each triple $\langle x, v, d \rangle$ that has been added to the set indicated by the $rset$ set of the process executing T until C. If T completes, let $RS_T = RS_T(CE_T)$. Consider any triple $\langle x, -, d \rangle \in RS_T(C)$. We say that d is *consistent* at C, if it is the version written by the last transaction in ℓ_C that updates x. $RS_T(C)$ is consistent at C, if for each triple $\langle x, -, d \rangle \in RS_T(C)$ the version d of x is consistent at C.

Consider a transaction T that adds a triple with version d for some data item x in its read-set during $RT_{x,T}$. Let T_d and T_{d+1} be the update transactions that write versions d and $d+1$, respectively, for x. The next lemma proves that during $RT_{x,T}$, T reads, as the owner for x, on line 48 either T_d or T_{d+1} and if it reads T_d then T has included T_d in its $beforeMe$ set.

Lemma 10. *Let T be any transaction and let C be a configuration at which a triple triple $\langle x, -, d\rangle \in RS_T(C)$. T adds $\langle x, -, d\rangle$ in its read-set (line 52) during $RT_{x,T}$. Let r and r' be the reads of line 48 and line 49, respectively, executed by T in $RT_{x,T}$ and let T_w be the value returned by r for $x \rightarrow owner$. If T_d and T_{d+1} are the update transactions that write d and $d+1$, respectively, for the version of x (line 76). then, either $T_w = T_d$ and $T_d \in T \rightarrow beforeMe$, or $T_w = T_{d+1}$.*

Proof. We start by providing an intuitive explanation of the proof. We proce that if T reads T_d on line 48, then line 51 is not executed. On the other hand, if T reads T_{d+1}, we argue that $T_{d+1} \notin T \rightarrow beforeMe$ set. If T_{d+1}'s status is `SIMULATING`, then T_{d+1} has not yet updated x when T executes line 49. So, on line 48, T has read d for x and line 51 is not executed. If T reads a value other than `SIMULATING` for the status of T_{d+1} (on line 49), then T executes line 51. So, T reads the version d for x from the write-set of T_{d+1}.

We now provide the details of the proof. To obtain a contradiction, suppose that either $T_w \notin \{T_d, T_{d+1}\}$, or $T_w = T_d$ and $T_d \notin T \rightarrow beforeMe$. We first argue that if T reads d on line 51, then $T_w = T_{d+1}$. If T reads d on line 51, then by inspection of the code (lines 48 and 50), T reads x's version in the $oldver$ field of some element e for x in the write-set of T_w. Then, Lemma 7 implies that $T_w = T_{d+1}$.

Assume first that $T_w \notin \{T_d, T_{d+1}\}$. Since, $T_w \neq T_{d+1}$, it follows that T does not read d on line 51. Lemma 7 implies that if T reads d on line 48, then $T_w = t_d$ which is a contradiction. Assume now that $T_w = T_d$ and $T_d \notin T \rightarrow beforeMe$. Since $T_w \neq T_{d+1}$, it follows that T does not read d on line 51. Thus, T reads d on line 48. We consider the following cases for the status of T_d returned by r'. Notice that this value is other than `ABORTED` since T_d enters its updating phase.

If r' returns `SIMULATING` then $r' < CU_{x,T_d}$. Since (1) r' is performed during α_{x,T_d}, (2) $r' < CU_{x,T_d}$, and (3) T_d changes the version of x from $d-1$ to d after CU_{T_d} (lines 74 and 76), Corollary 5 implies that r returns $d-1$. This is a contradiction to the assumption that r returns d.

Assume now that r' returns a value other than `SIMULATING`. By inspection of the code (line 82 and 90) and since T_d updates x and acquires the lock for x, T_d has added an element for x in its write-set. So, since r' is performed after CL_{x,T_d}, an element e with $e.tvar = x$ exists in the write set of T_d. Lemma 7 implies that $e.oldver = d-1$. Also, since by assumption $T_d \notin T_r \rightarrow beforeMe$, by inspection of the code (lines 50 - 51), T reads $e.oldver$ on line 51 and it is this value that T adds to its read-set (line 52); this contradicts our assumption that T reads d on line 48. □

Lemma 11. *Consider any transaction T and let C be a configuration at which a triple $\langle x, -, d\rangle \in RS_T(C)$. Let T_d be the update transaction that writes the version d for x (line 76) and let p' be the process executing T_d. Then, T_d enters its waiting phase and $CW_{T_d} < C$.*

Proof. During the execution of CHECKIFPERFORMED by T, T repeatedly reads (line 36), the transaction that is announced in $A[p']$ and the status of this transaction (line 37). Let r_1 and r_2 be these two reads executed by T in the last iteration of the `for` loop of line 35, during the last iteration of the `do while` loop of lines 34 - 39. Moreover, during the execution of $RT_{x,T}$, T reads the `direc` for x (line 48) and the status (line 49) of the transaction that it read as the owner of x on line 48 (during the execution of $RT_{x,T}$). Let r_3 and r_4 be these reads.

To obtain a contradiction, suppose that either T_d does not enter its waiting phase or $CW_{T_w} > C$; let C' be either the configuration following the last step taken by T in α, or CW_{T_w}, respectively. We first argue that $T_d \notin T \rightarrow beforeMe$. T reads d for x by executing either line 48 or line 51 during $RT_{x,T}$. If T executes line 51, let r_5 be this read. Notice that by inspection of the code, $r_1 < r_2 < r_3 < r_4 < r_5 < C$, and by assumption, $C < C'$. Thus, the definitions of r_1 and r_2 imply that in the instance of its CHECKIFPERFORMED, either T does not read T_w in $A[p']$ whenever it executes line 36 or if it reads T_w in $A[p']$, is some of these reads, it does not read a value equal to `WAITING` or `COMMITTED` for the status of T_w on line 37. Therefore, by inspection of the code (lines 36 to 38), $T_w \notin T \rightarrow beforeMe$.

Since $\langle x, -, d\rangle \in RS_T(C)$, Lemma 10 implies that either r_3 returns T_{d+1} or r_3 returns T_d and $T_d \in T \rightarrow beforeMe$. Since $T_d \notin T \rightarrow beforeMe$, it follows that r_3 returns T_{d+1}. Lemma 7 implies that there is a sequence $S = T_1, T_2, \ldots$ of update transactions in α that acquire the lock for x and T_d precedes T_{d+1} in S. Corollary 5 implies that $\alpha_{x,T_1} < \alpha_{x,T_2} < \ldots$. Observation 3 implies that if CW_{T_d} occurs, then it occurs in α_{x,T_d}. Since $r_3 < C'$, it follows that r_3 cannot return T_{d+1}. This is a contradiction. □

5.5.2 Correctness

5.5.2.1 Correctness of read-only transactions

Throughout this section, we consider a read-only transaction T_r that commits in α. The next lemma proves that if T_r reads d for a data item x, then the serialization point of the update transaction T_w which writes the version d for x is placed before the serialization point of T_r.

Lemma 12. *Consider any triple $\langle x, -, d\rangle \in RS_{T_r}$. Let T_d be the update transaction that writes the value d for the version of x. Then, $CW_{T_d} < CR_{T_r}$.*

Proof. To obtain a contradiction, suppose that $CW_{T_d} > CR_{T_r}$. Let r and r' be the reads on line 48 and line 49, respectively, executed during $RT_{x,T}$. Let T_w be the transaction returned by r as the owner of x. Then, $CL_{x,T_w} < r$. If T_{d+1} is the update transaction that writes the version $d + 1$ for x, then Lemma 10 implies that either $T_w = T_{d+1}$, or $T_w = T_d$ and $T_d \in T_r \rightarrow beforeMe$.

Assume first that $T_w = T_d$ and that $T_d \in T_r \rightarrow beforeMe$. By inspection of the code (lines 37 and 38), T_d can be added in the $beforeMe$ set of T_r only after CW_{T_d}. Since $CR_{T_r < CW_{T_d}}$, this addition occurs after CR_{T_r}. By inspection of the code (line 39), it follows that an iteration of the `do-while` loop of lines 36 to 38 is initiated after CR_{T_r}. This is a contradiction to the definition of CR_{T_r}.

We next assume that $T_w = T_d + 1$. Since T_r reads d for x and T_d writes d for x, Observation 3 implies that $r > CL_{x,T_w}$. Since we have assumed that $CW_{T_d} > CR_{T_r}$, Lemma 9 implies that T_r commits before T_d completes its waiting phase. So, r occurs in α_{x,T_d}. Lemma 7 implies that there is a sequence $S = T_1, T_2, \ldots$ of update transactions in α that acquire the lock for x and T_d precedes T_{d+1} in S. Corollary 5 implies that $\alpha_{x,T_1} < \alpha_{x,T_2} < \ldots$. Observation 3 implies that CW_{T_d} occurs in α_{x,T_d}. Since r occurs in α_{x,T_d}, it follows that r cannot return T_{d+1}. This is a contradiction. □

Lemma 13. *The set RS_{T_r} is consistent at CR_{T_r}.*

Proof. For each triple $\langle x, -, d\rangle \in RS_{T_r}$, we prove that d is written by the last committed transaction that updates x and is serialized before CR_{T_r}. By Observation 6, there is a unique update transaction T_d that writes d in x (`CAS` of line 76). Let C_d be the configuration following this `CAS` (line 76). By inspection of the pseudocode, $C_d < CW_{T_d}$. By Lemma 12 $CW_{T_d} < CR_{T_r}$.

Assume, by the way of contradiction, that the last committed transaction (let it be T_w) that updates x and is serialized before CR_{T_r}, writes the value $d' \neq d$ for x. Let C_w be the configuration following this CAS (line 76) and let p be the process that executes T_w. Since $CW_{T_d} < CR_{T_r}$, and T_w is the last committed transaction that updates x and is serialized before CR_{T_r}, it follows that $CW_{T_d} < CW_{T_w}$ By Observation 3, CW_{T_d} is in α_{x,T_d} and CW_{T_w} is in α_{x,T_w}. Then, Corollary 5 implies that $C_d < C_w$. So, by Observation 6 it follows that $d' > d$. Thus, $C_d < CW_{T_d} < C_w < CW_{T_w} < CR_{T_r}$.

Notice that after CR_{T_r}, T_r reads on line 36, the transaction that is announced in $A[p]$ and then T_r reads the status of this transaction on line 37. Let r_1 and r_2 be these two reads. Moreover, T_r reads the `direc` for x on line 48, in the first instance of READDI that it initiates for x, and on line 49 the status of the transaction that it read as the owner of x on line 48. Let r_3 and r_4 be these reads.

We argue that r_3 does not return d for the version x. Thus, T_r must read d in the $oldver$ field of some transaction by executing line 51. Let r_5 be this read. We also argue that r_5 reads from the write-set of T_w. We argue that the read of line 51 occurs only if $T_w \notin T_r \rightarrow beforeMe$. We also argue that r_1 read T_w in $A[p]$ and r_2 reads `WAITING` for the status of T_w. Then, by inspection of the code, T_r adds T_w in $T_r \rightarrow beforeMe$, which is a contradiction. We start by proving that r_3 does not return d for the version of x.

By inspection of the code (lines 76 and 78), T_w has updated the version of x to d' before CW_{T_w}. Since $r_1 > CW_{T_w}$, Observation 6 implies that r_1 returns either d', or a value larger than d' for the version of x. Thus, d is not read by T_r on line 48. So, by inspection of the code , d must be read by T_r on line 51, through the $oldver$ field of the element maintained for x in the write-set of the owner of x at that point in time.

Moreover, since $CR_{T_r} > CW_{T_w}$, r_3 returns as the owner of x a transaction T'_w, which is either T_w or some other transaction that acquired the lock for x after T_w. We argue that this owner is T_w. We next argue that $T_w \notin T_r \rightarrow beforeMe$. Since T_w writes $d' > d$, Observation 6 implies that any transaction that acquires the lock after T_w writes a value larger than d'. Lemma 7 implies that all these transactions other than T_w have a value larger than d stored in the $oldver$ field of the `wnode` for$x$ in their write-sets. Lemma 7 also implies that it must be T_w that has the value d in the $oldver$ field of the `wnode` for x in its write-set, and that T_w writes $d + 1$. It follows that the read of line 48 returns T_w as the owner for x. by inspection of the code, $T_w \notin T_r \rightarrow beforeMe$ and r_4 returns either `WAITING` or `UPDATING` for the status of T_w. Since r_4 occurs after CR_{T_r} and therefore, after CW_{T_w}, it follows that r_4 returns `WAITING` for the status of T_w.

Since, T_w is announced before CW_{T_w} (lines 30 and 77), $CW_{T_w} < CR_{T_r} < r_1 < r_3 < r_4$, and r_4 returns `WAITING` for the status of T_w, it follows that r_1 returns T_w as the owner of x and r_2 returns `WAITING` for the status of T_w. So, by inspection of the code (lines 37 - 38), $T_w \in T_r \rightarrow beforeMe$. This is a contradiction.

□

5.5.2.2 Correctness of update transactions

Throughout this section, we consider an update transaction T_w that enters its waiting phase. By inspection of the code (lines 27 and 65), T_w is initiated as read-only and it becomes an update transaction after the first execution of line 65 by it. Let $C^w_{T_w}$ be the configuration before the first execution of line 65 by T_w.

Lemma 14. *Consider any instance V of* Validate *executed by T_w that returns* `true` *and let C_V be the configuration before the invocation of V. Then, for each triple $\langle x, -, d \rangle \in RS_{T_w}(C_V)$, d is consistent at C_V.*

Proof. Consider any triple $\langle x, -, d \rangle \in RS_{T_w}(C_V)$. We will prove that d is written by the last committed transaction that is serialized before C_V and updates x. By Observation 6, there is a unique update transaction T_d that writes d in x (line 76). Since $\langle x, -, d \rangle \in RS_{T_w}(C_V)$ (i.e. T_w reads the version d for x), Lemma 11 implies that CW_{T_w} precedes the completion of the instance of CHECKIFPERFORMED executed by T_w and since $CW_{T_d} < C_V$.

Assume, by the way of contradiction, that the last committed transaction that is serialized before C_V is $T_{d'} \neq T_d$ which writes the value $d' \neq d$ for x (line 76). Since $CW_{T_d} < C_V$, it must be that $CW_{T_d} < CW_{T_{d'}} < C_V$ since otherwise $T_{d'}$ would not be the transaction that is serialized last before C_V. Thus, Corollary 5 and Observation 6 imply that $d < d'$.

After C_V, T_w reads the version of x (line 59); let r be the first such `read`. Since $CW_{T_{d'}} < C_V < r$, and by inspection of the code (lines 76 and 77), $T_{d'}$ writes $d' > d$ for x before $CW_{T_{d'}}$, Observation 6 implies that r returns either d' or a value larger that d'. However, since V returns `true`, r must return d for x. This is a contradiction. □

Lemma 15. *RS_{T_w} is consistent at CW_{T_w}.*

Proof. Let $\langle x, -, d\rangle$ be any triple added to the read-set of T_w. We prove that d is written by the last committed transaction that is serialized before CW_{T_w} and updates x. By Observation 6, there is a unique update transaction T_d that writes d in x; let C_d be the configuration following this `write` (line 76).

Let V be the last instance of VALIDATE (line 53) executed by T_w before CW_{T_w}; let C_V be the configuration preceding the invocation of V. Lemma 14 implies that d is consistent at C_V.

Since T_w enters its waiting phase, by inspection of the code (lines 71 - 72), it follows that the instance D of LOCKDATASET executed by T_w returns `true`; let C_D be the configuration following this response. Since D returns `true`, by inspection of the code (lines 82, 90, and 91), it follows that the `CAS` of line 91 executed for x is successful. By inspection of the pseudocode (lines 83 - 84, and 90), this `CAS` uses d as the version of its second parameter. Since, it is successful no other transaction has written x between CW_{T_d} and CL_{x,T_w}.

Assume, by the way of contradiction, that the last committed transaction $T_{d'}$ that updates x and is serialized after C_V and before CW_{T_w}, writes the value $d' \neq d$ to x. Then, Corollary 5 implies that there is some configuration between CW_{T_d} and CL_{x,T_w} in which the version of x is $d' \neq d$. A contradiction. $\square$

Lemmas 13, 14 and 15 imply the following.

Theorem 16. WFR-TM *is an opaque* TM *algorithm.*

5.5.3 Progress

In this section, we show that read-only transactions in WFR-TM are wait-free, and that update transactions are not prone to deadlock.

Let α be an execution of WFR-TM. Let m_w be the maximum number of data items written by any update transaction in α and m_r be the maximum number of data items read by any read-only transaction in α.

Lemma 17. *Consider any transaction T executed by some process p_i in α. Then, $T \rightarrow beforeMe$ contains at most two transactions initiated by each process p_j, $1 \leq j \leq n$, $j \neq i$.*

Proof. Notice that a new element can be added to $T \rightarrow beforeMe$ only during the execution of CHECKIFPERFORMED by T; specifically, this occurs with the execution of line 38. We will prove that line 38 can be executed by T at most twice for each entry $A[j]$, $1 \leq j \leq n$, $j \neq i$. We remark that since $T \rightarrow wset = \emptyset$ during the execution of CHECKIFPERFORMED by T, T is considered as a read-only transaction as long as it executes its CHECKIFPERFORMED.

Fix any j, $1 \leq j \leq n$, $j \neq i$. To obtain a contradiction, suppose that line 38 is executed by T three times for $A[i]$. Notice that before executing line 38, T reads (on line 36) the `direc` record of some transaction, from $A[i]$; let r_1, r_2, and r_3 be the reads of line 36 in that `for` iteration in which the first, the second, and the third execution, respectively, of line 38 by T occurs.

Let T_1, T_2, and T_3 be the transactions returned by r_1, r_2, and r_3, respectively. Notice that T_1, T_2, and T_3 have the same initiator p_i. Since the first execution of line 38 occurs after r_1, the second after r_2, and the third after r_3, the inspection of the code (1st condition of line 37) implies that $T_1 \neq T_2 \neq T_3$. Moreover, by inspection of the code (3rd condition of line 37), the statuses of T_1, T_2, and T_3 are either `WAITING` or `COMMITTED` when the condition of the `if` statement of line 38 is executed. So, by inspection of the code (lines 72 - 73, 74, 77, and 79) T_1, T_2, and T_3 do not abort.

By inspection of the code (lines 30, 78 - 79, and 95 - 98), T_1, T_2, and T_3 call WAITREADERS after CW_{T_1}, CW_{T_2}, and CW_{T_3}, respectively. Recall that T is considered as a read-only transaction while executing its instance of CHECKIFPERFORMED.

Assume first that the announcement of T precedes the announcement of T_1, thus it also precedes CW_{T_1}. So, T_1 waits (line 98) until the instance of CHECKIFPERFORMED initiated by T returns. Therefore, p_i cannot initiate T_2 as long as T executes its instance of CHECKIFPERFORMED.This contradicts the fact that p_j reads T_2 in $A[i]$ while T executes its instance of CHECKIFPERFORMED.

Assume now that T_1 is announced before the announcement of T. Notice that T_2 is initiated by p_j after the completion of T_1. Since T reads T_1 on line 36 from $A[i]$ (though r_1) and r_1 follows the announcement of T (line 30), it follows that T is announced before the announcement of T_2, that is also before CW_{T_2}. Therefore, p_j cannot initiate T_3 as long as T executed I_{cip}. This contradicts the fact that p_i reads T_3 in $A[j]$ during the execution of I_{cip}. □

Lemma 18. *Consider any transaction T in α. Then, T completes* BeginTx *within $O(n^2)$ steps.*

Proof. T executes lines 24 to 30 in $O(1)$ steps. Thus, it remains to show that CHECKIFPERFORMED completes after $O(n^2)$ steps. Lemma 17 implies that no more than $2(n-1)$ elements are added in $T \rightarrow BeforeMe$. Thus, no more than $O(n)$ iterations of the `do while` are executed. Each such iteration completes in $O(n)$ additional steps since it reads n elements of the announce array. Additionally, each such iteration executes a search in $T \rightarrow beforeMe$. We remark that if we implement $beforeMe$ set of each transaction as a two-dimensional array of $2n$ elements (i.e. two elements per process), then this search can be executed in $O(1)$ steps. □

Theorem 19. *Each read-only transaction executed by some process that accesses m data items commits after $O(n^2 + m_r m_w)$ steps, i.e. the execution of each read-only transaction is wait-free.*

Proof. Lemma 18 implies that T_r completes BEGINTX within $O(n^2)$ steps.

We prove that each instance of READDI executed by T_r completes in $O(m_w)$ steps. Since T_r is a read-only transaction, $T_r \rightarrow wset = \emptyset$. Thus, lines 44 and 53 are executed in $O(1)$ steps. Lines 46, 47, and 52 execute only local computations. All remaining lines other than 50 are also executed in $O(1)$ steps. Notice that the second condition of line 50 performs a search on $beforeMe$ set of T_r for transaction $owner$. Recall that if we implement $beforeMe$ set of T_r as a two-dimensional array of $2n$ elements (i.e. two elements per process), then this search can be executed in $O(1)$ steps, given that the process id of $owner$ is stored in its `txrec`. Thus, the only condition that may cause the execution of more that $O(1)$ steps on line 50 is the verification of the condition "$tvar \in owner \rightarrow wset$". This costs $O(m_w)$ steps. Thus, each instance of READDI executed by T_r completes within $O(m_w)$ steps.

By inspection of the code (lines 68 to 70), it follows that COMMITTX, when called by a read-only transaction completed within $O(1)$ steps. Thus, T_r completes within $O(n^2 + m_r m_w)$ steps. □

If T_w is an update transaction, then Theorem 19 and the inspection of the code imply that, for each read-only transaction T_r, T_w waits only for a finite number of steps in order for T_r to complete, on lines 89 or 98.

Theorem 20. *In any infinite execution of* WFR-TM*, each update transaction T_w completes within a finite number of steps.*

Proof. Since sets $T_w \rightarrow beforeMe$, $T_w \rightarrow wset$, and T_w's read-set are finite, by inspection of the code, it follows that CREATEDI, WRITEDI, VALIDATE, and LOCKDATASET when called by T_w complete within a finite number of steps. By inspection of the code (lines 95 to 98), T_w may have to wait for the completion of at most $n - 1$ read-only transactions while executing WAITREADERS. Theorem 19 implies that, for each read-only transaction T_r, T_w waits for a finite number of steps in order for T_r to complete. Thus, WAITREADERS completes within a finite number of steps and therefore the same is true for COMMITTX. □

Theorem 21 proves that WFR-TM provides deadlock-freedom also among update transactions.

Theorem 21. *In any infinite execution α of* WFR-TM *in which infinitely many update transactions are initiated, infinitely many update transactions commit.*

Proof. To obtain a contradiction, suppose that no update transaction ever commits after some configuration C of α. Then, Theorem 20 implies that infinitely many transactions abort after C.

By inspection of the code (lines 53 - 55, 71, and 72), an update transaction T_w aborts either when one of the instances of VALIDATE (line 53) that T_w executes returns `false`, or when the single instance of LOCKDATASET that T_w executes during COMMITTX returns `false`. In the first case, by inspection of the code of VALIDATE, it follows that the version of at least one data item has changed since it has been initially read by T; let this update be performed by some transaction T'_w. By inspection of the code (lines 74 - 80) and Theorem 20, it follows that T'_w commits within a finite number of steps. Since, no transaction commits after C, it follows that only a finite number of instances of VALIDATE can return `false`, after C.

Let C' be the configuration following the return of the last instance of VALIDATE that returns `false`, after C. So, any update transaction T'_w initiated after C' aborts because the instance D' of LOCKDATASET it executes returns `false`. By inspection of the code (lines 86 - 88 and 91), D' returns `false` when a data item in $RS_{T'}$ is locked by some other transaction. By inspection of the code (line 82), each transaction acquires the locks of the data items it accesses in (ascending) order. So, there is at least one transaction that is initiated after C', for which the instance of LOCKDATASET executed by it will be able to acquire all the required locks and respond with `true`; that is a contradiction. $\square$

Chapter 6

The SemanticTM Software Transactional Memory Algorithm

6.1 General

In this chapter, we introduce SemanticTM, an STM algorithm which achieves fine-grain parallelism at the transactional instruction level. Additionally, for simple transactions, assuming compiler support, SemanticTM ensures that all transactions (both read-only and update) complete within a finite number of steps and never abort, by ensuring that no transactions conflict. Since SemanticTM ensures local-progress, it naturally supports irrevocable operations. In Section 6.2) the main ideas of SemanticTM are presented. Also, in Section 6.3 the pseudocode of SemanticTM is described, and in Section 6.4, the proof of correctness and progress properties ensured by SemanticTM is presented. Finally, in Section 6.5, the results of the experimental evaluation of SemanticTM are presented.

6.2 Main Ideas

SemanticTM uses a set of lists, called *di-lists*, one for each data item. A scheduler processes transactions one after the other and places the instructions of each transaction in the appropriate di-lists based on which data items each of them accesses. All the instructions of each transaction are placed in the di-lists before the instructions of any subsequent transaction. The scheduler also records any dependencies that may exist between the instructions of the same transaction. Each of the workers repeatedly chooses, *uniformly at random*, a di-list and executes the instructions of this list, starting from the first one. Processing transactions in this way ensures that conflicts never occur; so, transactions never abort. Recall that compiler support is employed to know, for each instruction, any dependency that leads to or originates from it. Figure 6.1 shows the main structure of SemanticTM.

For example, consider transactions T_1 and T_2 of Figure 6.2. Without loss of generality, assume that the instructions of T_1 are placed in the di-lists first. Then, the instructions of lines 1 and 2 of T_1 will be placed in the di-list for x before the write to x on line 6 of T_2. Similarly, the write to y of line 3 of T_1 will be placed in the di-list for y before the write to y of line 5 of T_2. Since the worker processes respect the order in which instructions have been inserted in the lists when they execute them, the instructions of T_1 on each data item will be executed before the instructions of T_2 on this data item, and thus no conflict between them will ever occur.

The set of data items accessed by a transaction is its *data set*. We call *control flow statements* the conditionals and loops, and we use the instruction `cond` to refer to such a

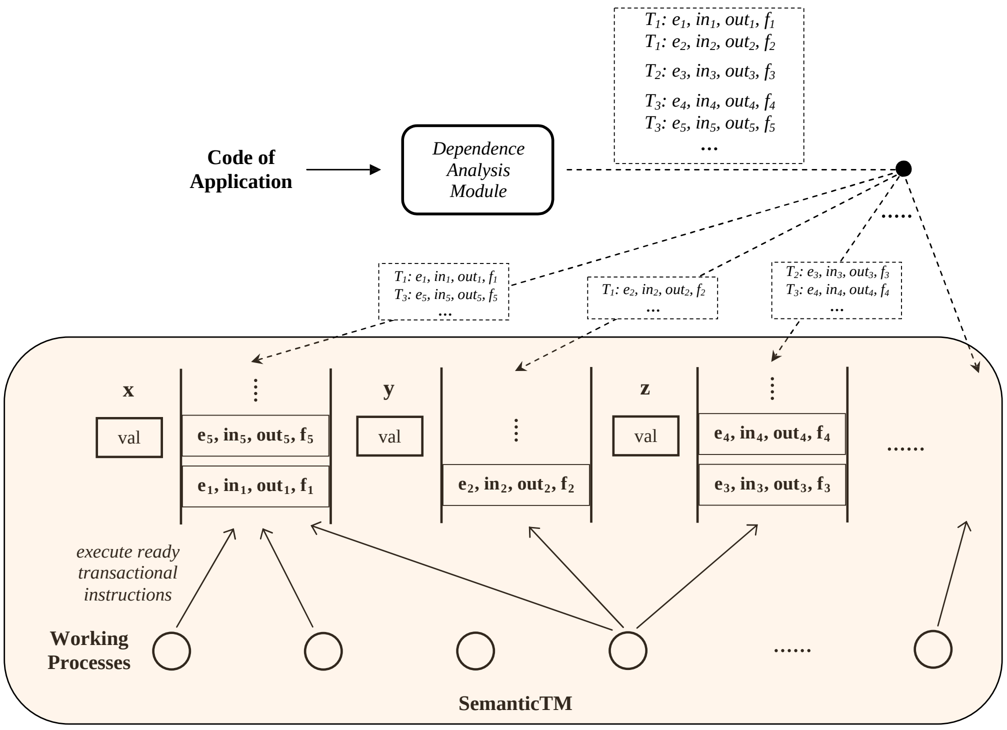

T_1: e_1, in_1, out_1, f_1
T_1: e_2, in_2, out_2, f_2
T_2: e_3, in_3, out_3, f_3
T_3: e_4, in_4, out_4, f_4
T_3: e_5, in_5, out_5, f_5
...
Code of Application
Dependence Analysis Module
x
y
z
val
e_5, in_5, out_5, f_5
e_1, in_1, out_1, f_1
e_2, in_2, out_2, f_2
e_4, in_4, out_4, f_4
e_3, in_3, out_3, f_3
execute ready transactional instructions
Working Processes
SemanticTM

Figure 6.1: Main Components of SemanticTM. Extraction of Transactional Instructions and Their Placement Into di-lists

statement. The *instructions* of a transaction are `read`, `write`, and `cond` instructions. We call the set of instructions in the body of a control flow statement a *block*; so each `cond` instruction is associated with a block.

6.2.1 Dependencies

If the execution of an instruction e_1 requires the result of the execution of another instruction e_2, then there is a *dependency* between e_1 and e_2. This dependency is an *input* dependency for e_1 and an *output* dependency for e_2. If e_1 requires the value read or written by e_2, then this dependency is called *data* dependency. Any other dependency that either leads to or originates from a `cond` instruction is called *control* dependency.

We remark that SemanticTM will place five instructions for T_1 (Figure 6.2) in the di-lists: e_1 which is a `write` to x (line 1), a `read` e_2 from x and a `write` e_3 to x (line 2), a `read` e_4 from x and a `write` e_5 to y for line 3. There is an output data dependency

```
1   $x := 3$
2   $x++$
3   $y := x$
```

T_1

```
4   $z := 2$
5   $y := z$
6   $x := y$
```

T_2

```
7   $x := 1$
8   if (...) then
9         $x := x + 2$
10  else
11        $x := x + 4$
12  $y := x$
```

T_3

```
13  $i := 1$
14  while ($i \leq 3$)   $cnt \in \{0, 1, 2, 3\}$
15        $j := 1$
16        while ($j \leq 5$)   $cnt \in \{0, 1, \ldots, 15\}$
17              $j := j + 1$   $startinneriter \in \{0, 5, 10\}$
18        $i := i + 1$
```

T_4

Figure 6.2: Transactions

from e_i to e_{i+1}, $1 \leq i < 5$. Notice that in order to execute an assignment between two data items (e.g. line 3, where the assignment on y depends on the value of x), SemanticTM places a `read` instruction (e_4 on x) before the corresponding `write` instruction (e_5 on y). By doing this, SemanticTM avoids maintaining data dependencies between `write` instructions (e.g. between e_3 and e_5). Also, for each control flow statement, SemanticTM places (to the appropriate di-lists) one `read` instruction for each of the data items used by it, before the corresponding `cond` instruction. By doing this, SemanticTM avoids to maintain data dependencies between the `cond` of this statement and each (if any) of the `write` instructions that writes some data item used by it.

Moreover, SemanticTM does not maintain input data dependencies for any `read` instruction e from a data item x, since all `writes` to x on which e depends have been placed in the di-list of x before e and thus the `read` can get the value from the metadata of x (by the way the algorithm works, this value will be consistent). Thus, SemanticTM records input data dependencies only for `write` and `cond` instructions (that originate from `read` instructions). For each such dependency, additional metadata is maintained, including the value of the data item, read by the corresponding `read` instruction, which is also called *value of the dependency*. We remark that each `read` (`write` or `cond`) instruction may have several output (input) data dependencies.

For each `cond` instruction, SemanticTM maintains an output control dependency from `cond` to each instruction e of the block associated with it. As an example, there is one output control dependency for instruction 8 (to 9) and another one for instruction 10 (to 11).

We assume that for each `write` instruction to a data item x or for each `cond` instruction e, a function f can be applied to the values of the input dependencies of e in order either to calculate the new value of x or to evaluate whether the condition is `true` or `false`, respectively. We remark that f should be applied after all the input data dependencies of e

have been resolved[1]. Table 6.1 provides a brief description of all possible dependencies for each instruction. The state of an instruction is *waiting*, if at least one of its input dependencies has not been resolved, otherwise, it is *ready*. An instruction that has an unresolved input control dependency is *inactive*; otherwise, it is *active*. Notice that a ready instruction is also active.

Recall that by using compiler support, the dependencies between the instructions of a transaction are known before the beginning of its execution. Each instruction, together with its dependencies (and function), is placed in the appropriate di-list, as a single *entry*.

6.2.2 Conditionals

Each conditional statement (`if`, `else if`, `else`) is associated with a `cond` instruction and a block. Therefore, for `if` ...`then` ...`else` statements, the two `cond`s (for the `if` and the `else` statement) and their blocks' transactional instructions will be placed in the appropriate di-lists. Then, at runtime, one of the two `cond` instructions will be evaluated as `false` so its block's instructions will be invalidated by the working thread that executes this `cond`. A `cond` instruction can be inserted in the di-list of any data item; in the current version of SemanticTM it is placed in the di-list of the first instruction of its block.

Notice that a transactional instruction of some block, may have *outside-block* dependencies which come from or lead to instructions that does not belong to the block. We remark that in SemanticTM outside-block dependencies are resolved directly (i.e. no extra metadata are maintained for them) because of the way that the transactional instructions are placed in the di-lists. For instance, there are input outside-block dependencies from the instruction of line 7 to the instructions of the `cond`s' blocks (lines 9 and 11). However, recall that in SemanticTM each of the lines 9 and 11 is replaced by a `read` from x and a `write` instruction to x, and no input data dependencies are maintained for `read`s. Moreover, with similar reasoning, the output outside-block dependencies from lines 9 and 11 to line 12 are also resolved directly.

[1]Computation on local variables can be included in the code of function f. For this reason, such a function may also be maintained for `read` instructions.

Transactional Instruction	Dependencies			
	Input		Output	
	Data Dep	Control Dep	Data Dep	Control Dep
$e = \texttt{read}(x)$	In SemanticTM, e has no input data dependencies	if e participates in some block, it has an input control dependency originating from the block's `cond`	e forwards the value it reads to `write` and `cond` instructions that depend on it	if e participates in some loop's block, an output control dependency originates from e to its block's `cond`
$e = \texttt{write}(x)$	e may have input data dependencies originating from `reads`	if e participates in some block, it has an input control dependency originating from the block's `cond`	In SemanticTM, e has no output data dependencies	if e participates in some loop's block, an output control dependency originates from e to its block's `cond`
$e = \texttt{cond}$	e may have input data dependencies originating from `reads`	if e participates in some block, it has an input control dependency originating from the block's `cond`; if e is a `cond` of a loop `cond`, it has an input control dependency originating from each of its block's instructions		e has output control dependencies to each of its block's instructions

Table 6.1: Data Dependencies Between Transactional Instructions

6.2.3 Loops

Let e be a transactional instruction that is included in a loop block; let c be the associated `cond`. SemanticTM places c and each instruction of the block in the appropriate di-lists only once independently of the number of times that the loop will be executed since this number may be known only at run time.

Additionally to the control dependency from c to e, an input control dependency of c from e is maintained. By doing this, c can initiate a new iteration only after its input control dependencies originating from its block instructions have been resolved, i.e. after all these instructions have been executed for the current iteration. Recall that these instructions can be executed only if their input control dependencies (from c) have been resolved. So, loop iterations are initiated one after the other, i.e. a new loop iteration is initiated only after the previous loop iteration has been completed.

In order to perform c (and e) multiple times, an *iteration counter* cnt_c is associated with c. This counter stores the current iteration number of the loop's execution. Moreover, the input control dependency of e is implemented with a counter cnt_e; the same counter is also used to implement the input control dependency of c from e. If $cnt_e = cnt_c - 1$, then the input control dependency of e is resolved for the (cnt_c)th iteration, otherwise, it is not; if $cnt_e = cnt_c$, then e has been performed and the input control dependency of c from e is resolved, for the (cnt_c)th iteration. Notice that cnt_e can be either equal to or smaller by one from cnt_c. This is so since loop iterations are initiated one after the other.

To ensure correctness, an *iteration number* is associated with each of the input data dependencies of e (or c). When the iteration number of an input data dependency $inDep$ of e (or c) is smaller than cnt_c, it follows that $inDep$ is unresolved for the current iteration; if all input data dependencies of e have their iteration number fields equal to cnt_c, then all data dependencies of e have been resolved and e can be executed. Once e is executed, it resolves the control dependency to c by writing cnt_c to cnt_e; recall that the same action marks e as performed for the current iteration. When all dependencies of c have been resolved c can be executed. If it decides to initiate the next iteration (i.e. its condition is evaluated as `true`) it increments cnt_c by one to resolve its output control dependencies for the next iteration.

We remark that the execution of e (and c) in some iteration may depend on the execution of some transactional instructions of the previous iteration; we call such a dependency *across-iteration*. Notice that SemanticTM does not maintain any of the across-iteration dependencies of e. Moreover, additionally to the instructions of the block of c, SemanticTM subsequently places (to the appropriate di-lists) one `read` instruction for each of the data items used by c. By doing this, SemanticTM avoids to maintain across iteration data dependencies between c and any of the `write` instructions (if any) that writes some data item used by c. So, c may only have input across-iteration dependencies from `read` instructions.

Notice that although (in SemanticTM) e does not have data dependencies with instructions outside the block of c, c may have input data dependencies from instructions both outside its block and inside its block; we call them *outer* and *inner* data dependencies, respectively. SemanticTM differentiates them so that inner data dependencies are not taken into consideration when c decides the initiation of the first loop iteration. Notice that, when the input control dependencies of c are resolved for some iteration, then its inner data dependencies are also resolved for the same iteration; so, c uses them to decide the initiation of the next loop iteration. Also notice that the inner data dependencies of c are the same with its across-iteration dependencies. So, c takes into consideration only its inner data dependencies, for iterations other than the first one.

Consider a di-list that contains two instructions e_1 and e_2, in this order, which are included in the same loop block, and assume that e_1 has been executed for the first loop iteration. Notice that e_1 is currently inactive until the next loop iteration is initiated. Also, recall that the next loop iteration is initiated by this loop's `cond` only after e_2 has also been executed for the first loop iteration. Since e_1 precedes e_2 in the di-list, the working processes may have to skip inactive instructions and search which element of the corresponding di-list is ready (instead of just checking whether the first element of the list is ready). Specifically, the di-list should be searched until an instruction is found that is either inactive or not ready, or does not participate in the same loop, or until the end of the list if such an instruction does not exist. In this way, the loop instructions are executed as if the loop was unfolded and all its instructions were executed in FIFO order. We remark that the loop in which an instruction participates can be determined using its input control dependency.

6.2.4 Nesting of Conditional Statements

Let c_2 be a `cond` that participates in the block of another `cond` c_1 (so the block of c_2 is *nested* in that of c_1). In SemanticTM, the output control dependencies of c_1 include only c_2 and not any instruction in the block of c_2; respectively, each instruction of the block of c_2 has an output control dependency with c_1.

We consider first the case where c_1 and c_2 are conditionals. During the execution of c_1 by some process p, p will resolve c_1's output control dependency to c_2. If c_2 is not invalidated, then c_2 and the instruction of its block are executed; otherwise, c_2 and the instructions of its block are invalidated.

We consider now the case where c_1 and c_2 are loops. Each time c_1 initiates a new iteration, c_2 is executed. During its execution, c_2 may execute several iterations of its block code. When its execution completes for some iteration of c_1, c_2 resolves its output control dependency to c_1. Recall that from this point on and until all the input control dependencies of c_2 from its block's instructions (including c_1) are resolved once more, c_2 is inactive.

Recall that c_2 has both inner and outer data dependencies. We remark that, in each iteration of c_1, c_2's outer data dependencies should be resolved before c_2 is executed for the first time for the current iteration of c_1. Notice that, a `cond` that does not participate in some loop is executed for the first time when its iteration counter equals 0. This is true for c_2, for the first iteration of c_1, but probably in next iterations of c_1 may have a value greater than 0.

To figure out the first time that c_2 is executed for the current iteration of c_1, a *start inner iteration number* $startinneriter_{c_2}$ is associated with c_2. Before the input control dependency of c_2 is resolved, $startinneriter_{c_2}$ is updated (by some working process executing c_1) so that $startinneriter_{c_2} = cnt_{c_2}$. For example, consider T_4 (Figure 6.2), with c_1 and c_2 be the `cond` instructions of lines 14 and 16, respectively. Notice that $startinneriter_{c_2}$ takes values $\{\perp, 0, 5, 10\}$ (in this order) where $\perp$ is its initial value and the rest are the values of cnt_{c_2} before c_2's loop starts in each iteration of c_1's loop, i.e. $(i-1)*5$, $1 \leq i \leq 3$, for the ith iteration of c_1's loop.

The case where c_2 is a conditional's `cond` and c_1 is a loop's `cond` is similar with the previous one, with the difference that c_2 is executed only once each time c_1 initiates another (outer) loop iteration. Finally, the case where c_2 is a loop's `cond` and c_1 is a conditional's `cond` is again similar with the above one (where c_1 and c_2 are both loops), with the difference that c_2's input control dependency is resolved at most once; specifically, if c_1 is evaluated as `true`.

6.2.5 Worker Processes

Since working processes choose the di-list to work on uniformly at random, it may happen that several working processes may (concurrently) execute the same instruction. To synchronize workers that execute the same instructions, the following synchronization techniques are employed. For each transactional instruction e, a $status$ field (with initial value `SIMULATING`) is maintained in its entry, indicating that e has not yet been performed. As soon as a working process completes the execution of e, it changes e's $status$ to `DONE`.

For each data item x, SemanticTM maintains a single `CAS` object which stores the value of x together with a *version number*. This is done in order to atomically update x. Recall that several instances of a `write` instruction e to some data item x which is contained in a loop block are executed (one for each iteration). The working processes executing the same instance of e should use the same old value for x, so that x is updated consistently; also, they should calculate the same new value for x for the current iteration. To ensure this, SemanticTM maintains a `CAS` object in the record of e which stores the old value of the data item to be updated by e and an iteration number; moreover, the new value of x, is calculated by all working processes using the values provided in input data dependencies of e for the current iteration.

In order to consistently resolve a data dependency, the value of this dependency is stored into a `CAS` object. Recall that an instruction participating in some loop has an iteration number associated with each of its data dependencies. In this case, this number is stored together with the value of the corresponding data dependency into a `CAS` object.

We consider now the case where a `cond` c_2 is nested under a `cond` c_1 and at least one of them is a loop's `cond`. In order the working processes executing c_1 to correctly update the $startinneriter_{c_2}$ field of c_2, this field is maintained into a `CAS` object. A worker process that wants to update this field for some iteration $j > 0$ of c_1, it first reads its current value, then it validates that the jth iteration of c_1 has not yet been initiated, and the updates this field (using `CAS`). By doing this, SemanticTM ensures that $startinneriter_{c_2}$ is updated exactly once for each iteration of c_1.

6.3 Pseudocode Description

6.3.1 Type Definitions

The type definitions of SemanticTM are presented in Figure 6.3. For each data item x, SemanticTM stores a record of type `direc` which consists of two fields: the value val of x and its version ver (initially 0); these `direcs` are maintained in some array $Ditem$ of size M, where M is the total number of data items. Also, SemanticTM associates with the ith data item x_i a di-list $List[i]$ which maintains the instructions to be applied on x_i; this list is implemented as a singly-linked list and each of its elements is of type `entry`. Specifically, $List[i]$ is a pointer to the first element of the di-list of x_i.

For each instruction e, SemanticTM stores a record called `entry`. The first field of `entry`, called ins, describes e and depending on the type ($iType$) of ins, it contains the following fields:

1. In case $iType =$ `read`: $outDD$, a `CAS` object containing a record of type `direc`, implementing the output dependency of e. The ver field of this `direc` is initialized

```
type direc                                  /* used to avoid ABA when updating $val$ or input data dependencies */
    $val$: value
    $ver$ : unsigned int

type oldvaluerec                            /* stores the old value of a data item */
    $oldv$: direc
    $inum$ : unsigned int

type entry
    $ins$: {
      ⟨$iType$ : read,
        $outDD$ : direc⟩,                   /* implements the output data dependencies of $ins$ */
      ⟨$iType$ : write,
        $inDD[]$ : ptr to direc,            /* implements the input data dependencies of $ins$ */
        $f$ : function,
        $oldvrec$ : oldvaluerec⟩,
      ⟨$iType$ : cond,
        $inDD[]$ : ptr to direc,            /* implements the outer input data dependencies of $ins$ */
        $inDDinner[]$ : ptr to direc,       /* implements the inner input data dependencies of $ins$ */
        $f$ : function,
        $outCD[]$ : ptr to entry,       /* together with $cnt$, implements the outer output control dependencies of $ins$ */
        $isloop$ : boolean,
        $cnt$ : unsigned int,           /* together with $outCD$, implements the outer output control dependencies of $ins$ */
        $startinneriter$ : unsigned int,
        $inCD[]$ : unsigned int⟩        /* implements the inner input control dependencies of $ins$; used to ensure that a new
                                           iteration starts after all its block instructions have been executed for the current iteration */
    }  /* end of $ins$ */
    $status$ : {SIMULATING, DONE}
    $inloop$ : boolean
    $pcond$ : ptr to entry
    $icond$ : unsigned int              /* index in the array of $pcond \to inCD$ where $ins$ should write (if needed) */
    $next$ : ptr to entry

/* Shared variables */
shared $Ditem[M]$: direc                        /* metadata for each data item */
shared $List[M]$: ptr to entry                  /* di-list of each data item */

/* Persistent local variables for process $p$ */
$innerExecutingCond_p$ : ptr to entry           /* maintain required information when executing
$innerExecutingCondIter_p$ : unsigned int       instructions participating in a loop */

local type iterations                           /* used to locally maintain inner and outer iterations of an instruction */
    $in$ : unsigned int
    $out$ : unsigned int
```

Figure 6.3: Type Definitions of SemanticTM

with the value 0; this initial value is the same with the initial value of the iteration counter of the `cond` instruction of the block in which e participates (if any).

2. In case $iType = \texttt{write}$ that is applied on some data item x: (a) $inDD$, an array of pointers to records of type `direc`. For each instruction *eptr* from which e depends, a pointer is maintained that points to $eptr \rightarrow outDD$, from where e should read its input value. (b) f, a function that can be applied to the values read through $inDD$ when e becomes ready, in order to calculate the new value of x. (c) *oldvrec*, a `CAS` object that contains an `oldvaluerec` record. It is used to maintain the old value of x. If e participates in a loop block, this value may be different for each loop iteration.
3. In case $iType = \texttt{cond}$: (a) $inDD$, an array that contains similar information for each of the (outer, in case e is a loop's `cond`) input data dependencies of e as for `writes` above. (b) $inDDinner$, an array that contains similar information for each of the inner input data dependencies of e as for $inDD$; it is used only when e is a loop's `cond`. (c) f, a function that can be applied to the values maintained in $inDD$ (or $inDDinner$, in case e is a loop's `cond` that decides the initiation of a loop iteration other than the first one) when e becomes ready, in order to evaluate whether `cond` is `true` or `false`. (d) *outCD*, an array that contains the output control dependencies of e; specifically, it contains a pointer to the `entry` record of each instruction participating in e's block. (e) *isloop*, a boolean that is `true` when e is a loop's `cond`; otherwise it is `false`. (f) *cnt*, a `CAS` object that contains an unsigned integer with initial value 0. It is used to determine its block's iteration and resolve e's output control dependencies. Each time the next iteration $k \geq 1$ is ready to start its execution, it is updated from $k-1$ to k and e's control dependencies are resolved for the kth iteration. (g) $startinneriter$, a `CAS` object that contains an unsigned integer with initial value 0. It is used when e participates in a block (i.e. when its block is nested under some other block). It maintains the value of *cnt* each time the input control dependency of e is resolved. (i) $inCD$, an array that maintains the input control dependencies of e. If e is a loop's `cond`, $inCD$ contains one element for each instruction included in `cond`'s loop block; otherwise, it is not used. Recall that each input control dependency is implemented as a counter; so each element of $inCD$ is a `CAS` object containing an unsigned integer which is initialized to 0.

In addition to ins, `entry` contains also the following fields: (a) $status$, a single bit that describes the $status$ of an instruction e. It is initially `SIMULATING` and changes to `DONE` after the execution of e has been completed; (b) *inloop*, a boolean that is `true` if

e participates in some loop's block; otherwise, it is `false`; (c) $pCond$, a pointer which is either `null` (if e does not participate in some block), or points to the `entry` record of the unique `cond` instruction from which e has an input control dependency (otherwise); (d) $iCond$, if e participates to some loop's block, then it is an index in the array $pCond \rightarrow inCD$ where e should write, to indicate that it has been executed for each specific iteration; (e) $next$, a pointer which is either `null`, or points to the next `entry` of the di-list containing e.

Recall that in order to execute several transactional instructions of a single loop which are contained in the same di-list, inactive instructions of this loop may have to be skipped. To implement this, locally in each process p, SemanticTM maintains a pointer $innerExecutingCond_p$ to some `entry` of a `cond` instruction that is currently *executing* (i.e. its output control dependency to its parent `cond` is not resolved) and the current inner iteration $innerExecutingCondIter_p$ of this block's `cond`.

The first time an instruction e has to be skipped and since e may be nested under other `cond`s, the parent `cond`s of e are visited until a `cond` that is currently executing is found, as follows. SemanticTM ensures that each new `cond` c visited participates in the same loop and the same iteration with e by checking the outer iteration of c against the inner iteration of c's parent `cond`, and then (if needed) performs the same check one level up between c's parent `cond` and c's grandparent `cond`, levelling up one level at a time. When a `cond` that is currently executing is found, SemanticTM maintains this `cond` and its inner current iteration into $innerExecutingCond_p$ and $innerExecutingCondIter_p$, respectively.

Then, whenever an instruction e' is reached that either has to be skipped or it is active, then the above procedure is repeated for e' and the parent `cond` $cin_{e'}$ of e' that is currently executing is found. Since e' may be nested under e, it may be that $cin_{e'} \neq innerExecutingCond_p$. In this case, SemanticTM validates that $cin_{e'}$ participates to the loop of $innerExecutingCond_p$ and to iteration $innerExecutingCondIter_p$, by repeating the above procedure. In case e' has to be skipped, $innerExecutingCond_p$ and $innerExecutingCondIter_p$ are updated with $cin_{e'}$ and its current iteration, respectively. By doing this, SemanticTM is able to ensure that the first active instruction found in some subsequent point (after e) of the corresponding di-list participates to the same block and it is executed for the same iteration.

Finally, SemanticTM's pseudocode uses local structure `iterations` to maintain the inner (if any) and outer iterations of some instruction.

```
21  APPLYINSTRUCTIONS()
22      $S = \{1, \ldots, M\}$                              /* $S$ is initialized with the indexes of all di-lists */
23      while ($S \neq \emptyset$) do                        /* as long as there is work */
24          $i$ := randomly choose an integer from $S$
25          while ($List[i] \neq$ null) do
26              $\langle eptr, iters \rangle$ := GETACTIVEINS($List[i]$)      /* find an active instruction */
27              if ($eptr$ = null) then break                 /* if no such instruction exists, choose some other di-list */

28              $\langle bool, arrayDD \rangle$ := CHECKDD($eptr, iters$)  /* check the input data dependencies of $eptr$ */
29              if ($bool$ = false) then break                /* if they are not resolved, $eptr$ is not ready */

30              EXECUTEINS($eptr, iters, \&Ditem[i], arrayDD$)        /* execute $eptr$ */
31              if ($List[i] \rightarrow status$ = DONE) then $List[i] := List[i] \rightarrow next$   /* update di-list */

32          if ($List[i]$ = null) then $S = S - \{i\}$        /* $List[i]$ is empty; remove index $i$ */
```

Figure 6.4: The Code of APPLYINSTRUCTIONS of SemanticTM

6.3.2 The Code of the SemanticTM Algorithm

SemanticTM's pseudocode is presented in Figures 6.4 to 6.8.

APPLYINSTRUCTIONS. APPLYINSTRUCTIONS repeatedly chooses a data item x (line 24) and executes consecutive ready instructions contained in its di-list, in order, starting from the first. More specifically, APPLYINSTRUCTIONS chooses the index of a data item maintained in the $Ditem$ array and the index of its corresponding di-list maintained in the $List$ array, from a set S that is initialized to contain all these indexes (line 22).

Recall that, if some instruction e in x's di-list participates in a loop's block and has been performed for the current iteration, other instructions in later positions of the list may be ready to execute for this iteration before e becomes ready again. Therefore, APPLYINSTRUCTIONS calls function GETACTIVEINS to find the first ready instruction in x's di-list.

If GETACTIVEINS returns $\langle \texttt{null}, - \rangle$, no instruction in this di-list is ready and APPLYINSTRUCTIONS selects some other di-list (line 27). Otherwise, $\langle eptr, iters \rangle$ is returned by GETACTIVEINS, where $eptr$ is active for iterations $iters$. So, GETACTIVEINS continues by checking if $eptr$'s data dependencies are resolved for the current iteration, by calling CHECKDD (line 28). If this is true, CHECKDD returns `true` together with the array containing the (`read`) transactional instruction from where the values of the input data dependencies of $eptr$ should be read (that is either $eptr \rightarrow inDD$ or $eptr \rightarrow inDDinner$); otherwise, it returns `false`. In the former case, GETACTIVEINS performs the ready instruction $eptr$, using function EXECUTEINS (line 30). In the latter case, APPLYINSTRUC-

```
⟨ptr to entry, iterations⟩ GETACTIVEINS ($pstart$ : ptr to entry)
    $innerExecutingCond_p := $ null
    $eptr := pstart$
    while ($eptr \neq$ null) do
        $iters :=$ READITERATIONS($eptr$)          /* read the inner (if any) and outer iterations */
                /* if at least one inactive loop instruction has been skipped, validate that $eptr$ participates in this loop, and update inner most
                executing cond of this loop and its current iteration (if needed); if validation fails then no active instruction has been found */
        if ($innerExecutingCond_p \neq$ null and
            PARTICIPATESINLOOP($eptr, innerExecutingCond_p,$
                                 $innerExecutingCondIter_p$) = false) then
            return ⟨null, $\perp$⟩
                                                    /* If the input control dependency of $eptr$ is unresolved
        if ($iters.out = 0$) then return ⟨null, $\perp$⟩     for its 1st iteration, no active instruction is found */
        if ($eptr \rightarrow inloop$ = true) then          /* if $eptr$ participates in some loop */
                /* if no instruction has been skipped, find the inner most executing cond of this loop and its current iteration, and
                validate that $eptr$ and its parent conds are executed for this iteration of the loop; if not restart from $pstart$ */
            if ($innerExecutingCond_p =$ null AND
                FINDINNEREXECUTINGCOND($eptr, iters.out$) = false) then
                $eptr := pstart$
                continue
                        /* if the input control dependency of $eptr$ from its parent cond has been resolved and $eptr$
                        either is not a cond, or it is a cond and the input control dependencies from the instructions
                        participating in its block are resolved, then $eptr$ is an active instruction */
            if (($iters.out = eptr \rightarrow pcond \rightarrow ins.inCD[eptr \rightarrow icond] + 1$) and
                ($eptr \rightarrow ins.iType \neq$ cond or CHECKINNERCD($eptr, iters.in$) = true) then
                return ⟨$eptr, iters$⟩
                        /* otherwise ($eptr$ does not participate in some loop and the input control
                        dependency of $eptr$'s parent, if any, is resolved), if $eptr$ is a cond */
        else if ($eptr \rightarrow ins.iType$ = cond)
                                                    /* if $eptr$ is not a loop's cond, then it is active */
            if ($eptr \rightarrow ins.isLoop$ = false) then return ⟨$eptr, iters$⟩
                        /* otherwise ($eptr$ is a loop's cond), if the input control dependencies
                        of $eptr$ from its block's instructions are resolved, then $eptr$ is active */
            else if (CHECKINNERCD($eptr, iters.in$) = true) then return ⟨$eptr, iters$⟩
                        /* otherwise ($eptr$ is an inactive loop's cond) and since $eptr$ should be skipped, it is added
                        it is maintained as the cond of the currently executing loop, together with its inner iteration */
            $innerExecutingCond_p := eptr$
            $innerExecutingCondIter_p := iters.in$
                        /* otherwise ($eptr$ does not participate in some loop, the input control dependency
        else return ⟨$eptr, iters$⟩   of $eptr$'s parent, if any, is resolved, and $eptr$ is either a read or a write), return $eptr$ */
                        /* $eptr$ is an inactive loop instruction; other ready instructions of the same loop
        $eptr := eptr \rightarrow next$      block in later positions of this di-list may have to be executed; so, skip $eptr$ */
    return ⟨null, $\perp$⟩
```

Figure 6.5: The Code of GETACTIVEINS of SemanticTM

```
55 iterations ReadIterations($eptr$ : ptr to entry)
56     if ($eptr \to ins.itype = $ cond) then            /* if $eptr$ is a cond */
                                                      /* if $eptr$ does not participate in some block $\to$ return inner iteration and 1 */
57         if ($eptr \to pcond = $ null) then return $\langle eptr \to ins.cnt, 1\rangle$
58         return $\langle eptr \to ins.cnt, eptr \to pcond \to ins.cnt\rangle$   /* otherwise, return both inner and outer iteration */
                                                      /* if $eptr$ participates in some block, then return its (outer) iteration */
59     else if ($eptr \to pcond \neq$ null) then return $\langle 0, eptr \to pcond \to ins.cnt\rangle$
60     else return $\langle 0, 1\rangle$                 /* otherwise, return that $eptr$ is executed for its 1st (and single) iteration */

61 boolean ParticipatesInLoop($eptr$ : ptr to entry, $loop$ : ptr to entry,
                               $loopIter$ : unsigned int)
62     $condptr := eptr \to pcond$                        /* $condptr$ is the parent cond of $eptr$ */
63     while ($condptr \neq$ null AND $condptr \neq loop$) then
64         $conditers :=$ ReadIterations($condptr$)
           /* if $condptr$ participates in some loop, has not yet been performed for its current outer iteration, and $innerExecutingCond_p$
           has not yet been updated, then maintain $condptr$ as the cond of the currently executing block of the loop */
65         if ($condptr \to pcond \neq$ null AND $condptr \to inloop =$ true AND
               $conditers.out = condptr \to pcond \to ins.inCD[condptr \to icond] + 1$ AND
               $innerExecutingCond_p = loop$) then
66             $innerExecutingCond_p := condptr$
67             $innerExecutingCondIter_p := conditers.in$

68         $condptr := condptr \to pcond$

69     if ($condptr =$ null OR $condptr \to ins.cnt \neq loopIter$) then return false
70     return true

71 boolean FindInnerExecutingCond($eptr$ : ptr to entry, $iter$ : unsigned int)
72     $outIter := iter$                                  /* remember the outer iteration of $eptr$ */
73     $condptr := eptr \to pcond$                        /* $condptr$ is the parent cond of $eptr$ */
74     while (true) do
75         $conditers :=$ ReadIterations($condptr$)
                                                      /* validate that $outIter$ is the current loop iteration of $condptr$ */
76         if ($outIter \neq conditers.in$) then return false
77                                 /* if $condptr$ participates in some loop and has not yet been performed for its current outer
                                   iteration, maintain it as the cond of the currently executing block of the loop and return true */
78         if ($condptr \to pcond =$ null OR $condptr \to inloop =$ false OR
               $conditers.out = condptr \to pcond \to ins.inCD[condptr \to icond] + 1$) then
79             $innerExecutingCond_p := condptr$
80             $innerExecutingCondIter_p := outIter$
81             return true

82         $outIter := conditers.out$                     /* remember the outer iteration of $condptr$ */
83         $condptr := condptr \to pcond$                 /* advance to the parent cond of $condptr$ */

84 boolean CheckInnerCD($eptr$ : ptr to entry, $iter$ : unsigned int)
       /* $eptr$ is a cond instruction; check whether its input control dependencies have been resolved for $iter$ */
85     for each element $el \in eptr \to ins.inCD$ do
86         if ($el \neq iter$) then return (false)        /* if not, return false */
87     return (true)                                      /* otherwise, return true */
```

Figure 6.6: The Code of ParticipatesInLoop, ReadIterations, UpdateLoopConds, and CheckInnerCD of SemanticTM

```
88  ⟨boolean, $arrayDD$[] : entry⟩ CHECKDD($eptr$ : ptr to entry, $iters$ : iterations)
                                   /* if $eptr$ is a read instruction, then it has no input data dependencies; so, return true */
89      if ($eptr \rightarrow itype$ = read) then return ⟨true, $\perp$⟩
                                   /* if $eptr$ is a write, or a cond that has not been executed at least once for $iters.out$ */
90      else if ($eptr \rightarrow ins.iType$ = write OR $eptr \rightarrow ins.startinneriter = iters.in$) then
                                   /* it is checked whether each of $eptr$'s input data dependencies is resolved for iteration $iters.out$ */
91          for each element $d \in eptr \rightarrow ins.inDD$ with index $j$ do
92              $tmp$ :=*$d$
93              if ($tmp.ver \neq iters.out$) then return ⟨false, $\perp$⟩      /* if this is not true, false is returned */
94          return ⟨true, $eptr \rightarrow ins.inDD$⟩      /* if this is true for all the dependencies of $eptr$, true is returned */
                                   /* $eptr$ is an active cond that has been executed at least once for $iters.out$, so it is also ready */
95      else return ⟨true, $eptr \rightarrow ins.inDDinner$⟩

96  EXECUTEINS($eptr$ : ptr to entry, $iters$ : iterations, $pdi$ : ptr to direc,
            $arrayDD$ : entry)
97      if ($eptr \rightarrow ins.iType$ = cond) then      /* $eptr$ is a cond */
        /* if $eptr$ is an if statement in some loop and has been initiated for the current loop iteration, then complete it for the current (outer) iteration */
98          if ( $eptr \rightarrow ins.isloop$ = false AND $eptr \rightarrow inloop$ = true AND
               $iters = \langle eptr \rightarrow ins.startinneriter + 1, eptr \rightarrow pcond \rightarrow ins.cnt\rangle$) then
99              $decision$ := false
100         else
101             $values[1..k] := \{arrayDD[1] \rightarrow val, \ldots, arrayDD[k] \rightarrow val\}$
102             $decision := eptr \rightarrow ins.f(values)$      /* otherwise, evaluate its condition */
103         if ($decision$ = true) then      /* if condition is evaluated as true, then its dependent */
                      /* if $eptr$ either handles a loop or participates in some loop, then the conds participating in its block are initialized */
104             if ($ins \rightarrow ins.isloop$ = true OR $ins \rightarrow inloop$ = true) then
105                 INITIALIZEDEPENDENTCONDS($eptr, iters.in + 1$)
                                   /* its control deps are resolved for iteration $iters.in + 1$ */
106             CAS($eptr \rightarrow ins.cnt, iters.in, iters.in + 1$)
                                   /* if $eptr$ neither participates in some loop nor handles some loop, mark it as completed */
107             if ($eptr \rightarrow inloop$ = false AND $eptr \rightarrow ins.isloop$ = false) then
                    $eptr \rightarrow status$ := DONE
                                            /* if its condition is evaluated as false and $eptr$ participates in
108         else if ($eptr \rightarrow inloop$ = true) then    some loop, then its output control dependency is resolved */
109             CAS($eptr \rightarrow pcond \rightarrow ins.inCD[eptr \rightarrow icond], iters.out - 1, iters.out$)
110         else                          /* otherwise, $eptr$ does not participate in some loop and its condition is
111             RESOLVECDINVALID($eptr$)      evaluated as false, resolve its control dependencies so that its dependent
112             $eptr \rightarrow status$ := DONE      instructions are marked as completed and mark it as completed */
113     else                 /* otherwise, $eptr$ is not cond. if $eptr$ is read, resolve its output data dependencies */
114         if ($eptr \rightarrow ins.iType$ = read) then RESOLVEDD($eptr, iters.out, pdi$)
115         else                          /* otherwise $eptr$ is a write, so calculate $pdi$'s new value and update it */
116             $values[1..k] := \{arrayDD[1] \rightarrow val, \ldots, arrayDD[k] \rightarrow val\}$
117             $newval := eptr \rightarrow ins.f(values)$
118             UPDATEDI($pdi, newval, eptr, iters.out$)
119         if ($eptr \rightarrow inloop$ = true) then      /* if $eptr$ participates in some loop, resolve its output control dependency */
120             CAS($eptr \rightarrow pcond \rightarrow ins.inCD[eptr \rightarrow icond], iters.out - 1, iters.out$)
121         else $eptr \rightarrow status$ := DONE      /* otherwise, mark $eptr$ as completed */
```

Figure 6.7: The Code of CHECKDD and EXECUTEINS of SemanticTM

```
122 UPDATEDI($pdi$ : ptr to direc, $newval$ : value, $eptr$ : ptr to entry,
        $cnt$ : unsigned int)
123     $data := *pdi$        /* read the current value of pdi's direc record and try to store it into $oldvrec$ field of $eptr$ */
124     CAS($eptr \rightarrow ins.oldvrec, \langle eptr \rightarrow ins.oldvrec.oldv, cnt - 1\rangle, \langle data, cnt\rangle$)
125     $\langle oldv, inum\rangle := eptr \rightarrow ins.oldvrec$        /* read $oldvrec$ field of $eptr$ */
126     if ($inum \neq cnt$) then return        /* if $eptr$ has already been performed for iteration $cnt$, then return */
127     CAS($pdi, oldv, \langle newval, oldv.ver + 1\rangle$)        /* update $pdi$ with $newval$ and increment its version */

128 RESOLVEDD($eptr$ : ptr to entry, $iter$ : unsigned int, $pdi$ : ptr to direc)
129     $val := pdi \rightarrow val$        /* read the value of the $pdi$'s $val$ field */
                                            /* read the current value of the $val$ field of $eptr$'s output
130     $curval := eptr \rightarrow ins.outDD.val$        data dependency and update it with the $val$ of $pdi$ using
131     CAS($eptr \rightarrow ins.outDD, \langle curval, iter - 1\rangle, \langle val, iter\rangle$)        this dependency's current $val$ and iteration $iter$ */

132 INITIALIZEDEPENDENTCONDS($eptr$ : ptr to entry, $newiter$ : unsigned int)
133     for each element $d \in eptr \rightarrow ins.outCD$ do        /* for each dependant $d$ of $eptr$ */
134         if ($d \rightarrow iType$ = cond) then        /* if $d$ is a cond */
135             $curinnerIter := d \rightarrow ins.cnt$        /* read the current inner iteration of $d$ */
136             $tmpstartinneriter := d \rightarrow ins.startinneriter$        /* read $d$'s maintained inner block iter */
137                     /* if the execution of $d$ for outer iteration $newiter$ has not started yet $d$'s, then
                        the maintained inner block iteration is updated with its current (inner) iteration */
138             if ($eptr \rightarrow pcond.ins.nct = newiter - 1$ AND
                    $tmpstartinneriter < curinnerIter$) then
139                 CAS($d \rightarrow ins.startinneriter, tmpstartinneriter, curinnerIter$)

140 RESOLVECDINVALID($eptr$ : ptr to entry)
141     for each element $d \in eptr \rightarrow ins.outCD$ do        /* for each dependant $d$ of $eptr$, if $d$ is a cond,
142         if ($d \rightarrow iType$ = cond) then RESOLVECDINVALID($d$)        then mark $d$'s dependants as completed, and
143         $d \rightarrow status$ := DONE        mark $d$ as completed */
```

Figure 6.8: The Code of UPDATEDI, RESOLVEDD, INITIALIZEDEPENDENTCONDS, and RESOLVECDINVALID of SemanticTM

TIONS selects some other di-list (line 29).

When the execution of e is completed (line 31), the head pointer of x's di-list is updated, so that it points to the next instruction (if any) after e in this di-list. If APPLYINSTRUCTIONS reaches the end of x's di-list, then since all the instructions of this di-list have been executed, the index of x is removed from S and APPLYINSTRUCTIONS selects some other di-list (line 32).

CHECKDD. CHECKDD (lines 89 - 95) takes as a parameter a pointer *eptr*[2] to some active transactional instruction's `entry` record and the iterations $iters$ of *eptr*. If *eptr* is a `read` instruction, then `true` is returned (line 89). If *eptr* is either a `write` instruction, or a `cond` instruction that has not been executed at least once for its outer iteration (line 90), then CHECKDD (lines 91 to 93) checks whether the iteration number of each input data dependency of *eptr* matches its current outer iteration (line 93); if this is true, CHECKDD returns `true` together with $eptr \rightarrow inDD$ (line 94), otherwise, it returns `false` (line 93). Otherwise, *eptr* is a `cond` instruction that has been executed at least once for its outer iteration and since it is active, it follows that its data dependencies are resolved and $\langle \texttt{true}, eptr \rightarrow inDDinner \rangle$ is returned (line 95).

GETACTIVEINS, PARTICIPATESINLOOP, READITERATIONS, UPDATELOOPCONDS, and CHECKINNERCD. GETACTIVEINS takes as a parameter a transactional instruction $pstart$ of a di-list L and returns a pointer (*eptr*) to the topmost element of L that is active (if any), as well as some information about its status, as described below; *eptr* is initialized with $pstart$. To implement this, for each working process p, GETACTIVEINS uses $innerExecutingCond_p$ (and $innerExecutingCondIter_p$) which is initialized (line 34).

Initially, it determines the inner (if any) and outer iterations of *eptr* by calling READITERATIONS (line 37). READITERATIONS (lines 56 - 60) returns both inner and outer iterations when *eptr* is a `cond` that participates in some block (line 58), it returns the inner iteration and value 1 for its outer iteration when *eptr* is a `cond` that does not participate in some block (line 57), it returns the (outer) iteration of *eptr* when it is a `read` or a `write` that participates in some block (line 59), and it returns the 1st and single (outer) iteration when *eptr* is a `read` or a `write` that does not participate in some block (line 60).

In case some inactive transactional instruction has been skipped (1st condition of line 38) and either *eptr* does not participate in some loop, or participates in a loop dif-

[2]Notice that from this point on, we use *eptr* to refer both to the data item's name and to the pointer to its `entry` record.

ferent than the one starting with $innerExecutingCond_p$, or in some iteration of this loop other than $innerExecutingCondIter_p$, then GETACTIVEINS returns $\langle \texttt{null}, \perp \rangle$, identifying that no active instruction has been found in L (line 39). GETACTIVEINS validates that *eptr* is executed for this loop and this iteration by calling PARTICIPATESINLOOP (lines 62 to 70), which traverses the parent `conds` of *eptr* until either $innerExecutingCond$ or `null` is found, and returns either `true` (line 70) or `false` (line 69), respectively. Moreover, PARTICIPATESINLOOP validates that the current iteration of the inner most executing `cond` ($innerExecutingCond_p$) has not changed (line 69); if this is not true, `false` is retuned. Finally, PARTICIPATESINLOOP updates (if needed) the inner most executing `cond` ($innerExecutingCond_p$) of this loop (lines 65 - 67). If PARTICIPATESINLOOP returns `false`, then GETACTIVEINS returns $\langle \texttt{null}, \perp \rangle$. Otherwise, GETACTIVEINS continues as follows.

If the outer iteration of *eptr* is 0 (line 40), then its input control dependency has not been resolved at least once, so *eptr* is inactive and GETACTIVEINS returns $\langle \texttt{null}, \perp \rangle$, identifying that no active instruction has been found in L. Otherwise, GETACTIVEINS continues as follows.

We consider first the case that *eptr* participates in some loop (line 41). If no instruction has been skipped, GETACTIVEINS finds the inner most executing `cond` of this loop and its current iteration, and validates that *eptr* and its parent `conds` are executed for this iteration of the loop, by calling FINDINNEREXECUTINGCOND (line 42). More specifically, FINDINNEREXECUTINGCOND (lines 72 to 83) traverses the parent `conds` of *eptr* (lines 73, 74, and 83) and checks that they are executed for the same loop iteration (line 76); if not, `false` is returned. When a `cond` is reached that either does not participate in some loop or it is not yet performed for its current outer iteration (line 78), FINDINNEREXECUTINGCOND maintains it together with its iteration in $innerExecutingCond_p$ and $innerExecutingCondIter_p$ (lines 79 - 80) as the `cond` and the iteration, respectively, of the currently inner most executing block of the loop and returns `true` (line 81).

If FINDINNEREXECUTINGCOND returns `false` (line 42), then GETACTIVEINS restarts with *eptr* equal to $pstart$. Otherwise, if FINDINNEREXECUTINGCOND returns `true`, GETACTIVEINS checks whether the input control dependency of *eptr* is resolved (1st condition of line 45) and, in case it is a `cond` the input control dependencies from instructions participating in *eptr*'s block are resolved (3rd condition of line 45). If this is `true`, then *eptr* is active and it is returned together with its iterations (line 46); otherwise, *eptr* is skipped (line 53). Notice that GETACTIVEINS checks the input control dependencies of *eptr* from instructions participating in its block by calling CHECKINNERCD(line 45). CHECKINNERCD (lines 85 to 87) returns `true` if all input control dependencies of *eptr* are resolved for the current iteration; otherwise, it returns `false`.

We consider now the case that *eptr* does not participates in some loop. Recall that the input control dependency of *eptr* (if any) is resolved (line 40). If *eptr* is a `cond` (line 47) but not a loop `cond` (line 48), then it is active and it is returned. If *eptr* is a loop `cond`, GETACTIVEINS checks whether the input control dependencies from instructions participating in *eptr*'s block are resolved (line 49). If this is `true`, then *eptr* is active and it is returned (line 49). Otherwise, since *eptr* has to be skipped, it is maintained as the `cond` of the currently executing block of the loop (line 50), together with its current inner iteration (line 51), and then *eptr* is skipped (line 53). Finally, if *eptr* is not a `cond` (line 52), then it is active and it is returned.

EXECUTEINS, UPDATEDI, RESOLVEDD, INITIALIZEDEPENDENTCONDS, and RESOLVECDINVALID. EXECUTEINS (lines 97 - 121) takes as parameters *eptr*, the iterations of *eptr*, a pointer to the `direc` record of the data item on which *eptr* is applied (when $eptr \rightarrow iType \in \{\texttt{read}, \texttt{write}\}$), and an array $arrayDD$ describing the transactional instruction from where the values of the input data dependencies of *eptr* should be read.

We consider first the case where *eptr* is a `cond`. If *eptr* is a conditional (1st condition of line 98) that participates in some loop (2nd condition of line 98), and has been initiated for the current loop iteration (3rd condition of line 98), then EXECUTEINS assigns `false` to the local variable $decision$, so that it completes the execution of *eptr* in the current (outer) iteration by resolving its output control dependency (line 109). Otherwise, its condition is evaluated (lines 101 - 102) and the result is stored in $decision$. If $decision$ is `true` (line 103) and *eptr* either handles a loop or participates in a loop (line 104), then its dependent `conds` are initialized by calling INITIALIZEDEPENDENTCONDS (line 105) with parameters *eptr* and the inner iteration of *eptr* incremented by one. Also, the output control dependencies of *eptr* are resolved (line 106) and in case *eptr* neither participates in some loop nor handles a loop it is marked as completed (line 107).

If $decision$ is `false` and *eptr* participates in some loop (line 108), then its output control dependency is resolved (line 109) and at the same time *eptr* is marked as inactive (for the current outer iteration). Otherwise, *eptr* neither participates in nor is nested under some loop, its condition is evaluated as `false` (line 110), and its control dependencies are resolved using RESOLVECDINVALID (line 111), so that its block's instructions and any block's instructions nested under it are marked as completed. Also, *eptr* is marked as completed (line 112). RESOLVECDINVALID (lines 141 - 143) marks as completed (line 143) all the instruction participating in *eptr*'s block (line 141) and all the instructions participating in any block nested under *eptr*'s block by iteratively calling RESOLVECDINVALID for any `cond` instruction reached (line 142).

INITIALIZEDEPENDENTCONDS (lines 133 - 139) takes as parameters *eptr* and the new iteration *newiter* to be initiated by *eptr*. If d participates in the block of e and is a `cond`, the current inner iteration (line 135) of d, and the maintained inner start iteration (line 136) of d are read. If the execution of d for iteration *newiter* has not yet started (first condition of line 138) and the maintained inner start iteration of d is smaller than the current inner iteration of d, then d's maintained inner start iteration is updated using d's current inner iteration (line 139).

We now discuss the case where *eptr* is not a `cond` (line 113). If the type of *eptr* is `read` (line 114), its output data dependencies are resolved for its (outer) iteration, by calling function RESOLVEDD. RESOLVEDD (lines 129 - 131) reads the current data *data* of *pdi* and updates the output dependency of *eptr*, using the old data of this dependency and the specified iteration number.

If the type of *eptr* is a `write` on some *pdi* (line 115), then the new value *newval* of *pdi* is calculated (lines 116 - 117) and then *pdi* is updated with *newval*, by calling function UPDATEDI (line 118). Specifically, UPDATEDI (lines 123 - 127) takes as parameters *di*, *newval*, *eptr*, and *cnt*. It starts by trying to store the current value of x's `direc` together with the current iteration (*cnt*) into *eptr* $\rightarrow$ *ins.oldvrec* (line 124). Then, it checks whether *eptr* is already performed (line 126); if this is true, it returns. Otherwise, the `direc` record of x is atomically (using `CAS`) updated (line 127) using *eptr* $\rightarrow$ *ins.oldvrec.oldv* (as the first parameter of this `CAS`) and *newval* together with the stored version of *di* (that is *eptr* $\rightarrow$ *ins.oldvrec.oldv.ver*) incremented by one (as the second parameter of this `CAS`).

Finally, if *eptr* participates in some loop (line 119) its output control dependency is resolved (line 120); otherwise, its *status* is updated to `DONE` (line 121).

6.4 Proof of the SemanticTM algorithm

Recall that SemanticTM focuses on relatively simple transactions that access a known set of data items, so the work of the scheduler can be performed statically at compile time. More specifically, the dependencies of each transactional instruction (`read`, or `write`, or `cond`) are statically known. Together with its dependencies, the instruction is placed into the appropriate di-list (as an `entry` record) based on which data item it accesses. Moreover, statically at compile time, the transactional instructions of each transaction are placed in the di-lists before the transactional instructions of any subsequent transaction.

So, roughly speaking, in order to prove SemanticTM's correctness it is enough to prove that i) the transactional instructions of each di-list are executed by worker processes in the order they are placed in the list, ii) each instruction e is executed only after its dependencies have been resolved, iii) the dependencies of e are resolved exactly once for each iteration in which it is performed, and iv) in each iteration of a block its transactional instructions are executed exactly once.

6.4.1 Definitions

Each transactional instruction e is associated with a unique `entry` record; the *status* of e is the value of field *status* in this record, which is initially `SIMULATING`. As long as the status of e is `SIMULATING`, we say that e is *not finished*. If its status becomes `DONE`, we say that e is *finished*. Throughout this proof we abuse notation and we use the same notation to refer both to the name of some transactional instruction and to its `entry` record.

Recall that each `cond` instruction is associated with a block of transactional instructions. Consider any transactional instruction e. If $e.pcond = null$, we say that e *participates* in the *main block*. If $e.pcond \neq null$, we say that e *participates* in the block of $e.pcond$, $e.pcond$ is the *parent* and an *ancestor* of e, and e is a *child* and a *descendant* of $e.pcond$. Notice that e may have several ancestor `cond`s, e.g. in case $e.pcond.pcond \neq null$, then $e.pcond.pcond$ is an ancestor `cond` of e, as well. More specifically, if ac is an ancestor `cond` of e and $ac.pcond \neq$ `null`, then $ac.pcond$ is an ancestor `cond` of e. Also, e is a descendant of each of its ancestors. If e is a `cond` and $e.ins.isloop =$ `true`, we say that e *handles* a loop. If any of the ancestors of e handles a loop, we say that e *participates* in the loop of that `cond`.

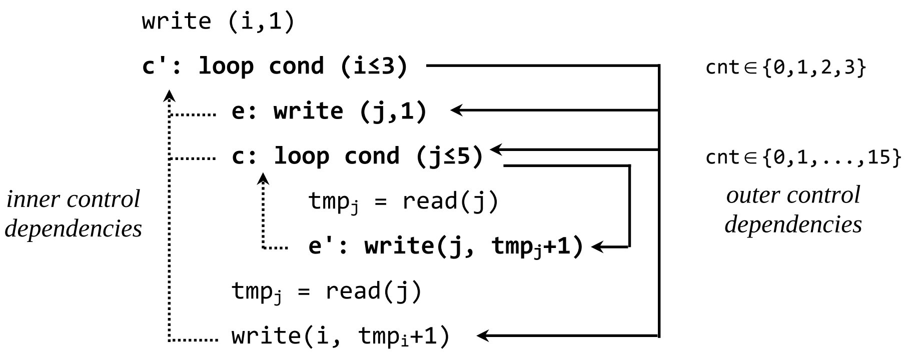
Consider any `write` or `cond` instruction wc. Recall that when an *outer* (or *inner*) *data dependency* exists between wc and a `read` instruction r that is *input* data dependency for wc and *output* data dependency for r, then an entry of $wc.inDD$ (or an entry of $wc.inDDinner$, in case wc is a `cond`) points to the $outDD$ field of r. [3]. Consider any `cond` instruction c. Recall that when an *outer control dependency* exists between c and a transactional instruction e that is *input* control dependency for e and *output* control dependency for c, then an entry of $c.outCD$ points to e. Moreover, in this case, $e.pcond = c$ and if c handles a loop, then c and e also have an *inner control dependency* that is *input* control dependency for c and *output* control dependency for e. Figure 6.9 presents an example of the control dependencies maintained in SemanticTM. In case some transactional instruction e participates in the main block, for simplicity, we assume that it has an (outer) input control dependency (from the fictitious `cond` of the main block to e).

Observation 1. *Consider a transactional instruction* e *and let* c *be either* $e.pcond$*, in case* $e.pcond \neq$ `null`*, or the fictitious* `cond` *of the main block, otherwise. Then, the following hold:*

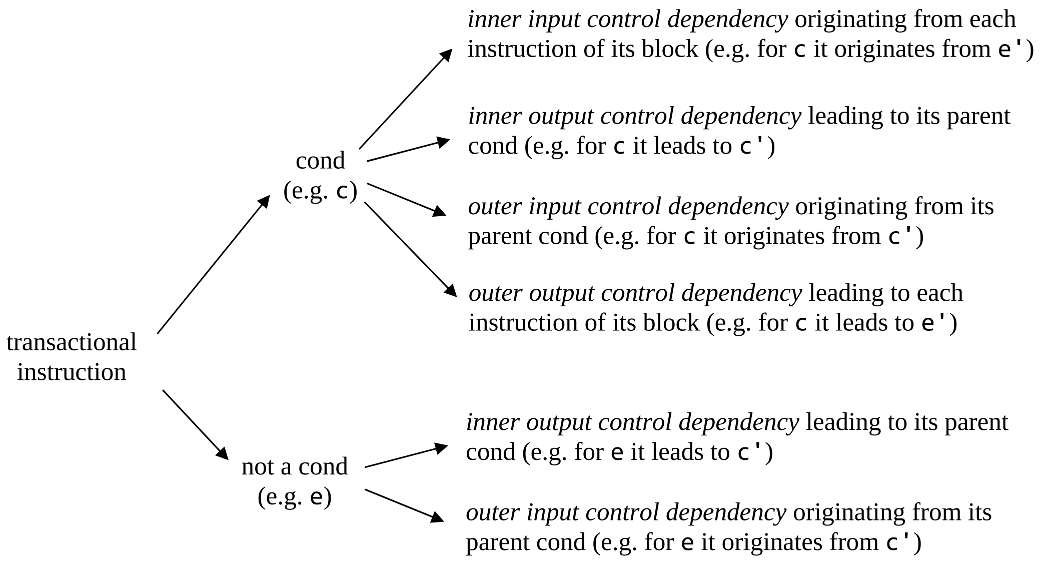
1. e *has an outer input control dependency originating from* c*,*
2. *if* e *participates in some loop, then it has an inner output control dependency leading to* c*,*
3. *if* e *is a* `cond`*, let* e' *be any of the transactional instructions participating in the block of* e*, then*
   (a) e *has an outer output control dependency leading to* e'*, and*
   (b) *if* e *participates in some loop, then* e *has an inner input control dependency originating from* e'*.*

Each data item x is associated with a unique `direc` record and a unique di-list. Throughout this proof we abuse notation and we use the same notation to refer both to the name of some data item and to its `direc` record. Each time a working process p successfully executes the `CAS` instruction of line 127 for some data item x with values $\langle v, l\rangle$, we say that p *updates* the value and the version of x with v and l, respectively, or *writes* the value v and version l to x. Recall that RESOLVEDD is only called by read instructions. Each time a working process p successfully executes the `CAS` instruction of line 131 for some data item

[3]Notice that *outDD* points to the opposite direction of the data dependency's direction. This is so, since in SemanticTM, each `write` instruction "reads" the required values by accessing the `direcs` of the corresponding `read` instructions, instead of having each `read` instruction "sending" the value read to its dependant `write` instructions.



```
write (i,1)
c': loop cond (i≤3)
    e: write (j,1)
    c: loop cond (j≤5)
        tmpj = read(j)
        e': write(j, tmpj+1)
    tmpj = read(j)
    write(i, tmpi+1)
```


transactional instruction
cond (e.g. c)
not a cond (e.g. e)
inner input control dependency originating from each instruction of its block (e.g. for c it originates from e')
inner output control dependency leading to its parent cond (e.g. for c it leads to c')
outer input control dependency originating from its parent cond (e.g. for c it originates from c')
outer output control dependency leading to each instruction of its block (e.g. for c it leads to e')
inner output control dependency leading to its parent cond (e.g. for e it leads to c')
outer input control dependency originating from its parent cond (e.g. for e it originates from c')

Example of instances of transactional instructions `c` and `c'` for their iterations:

```
c': <0,1>                         <1,1>                <2,1>               <3,1>
c :      <0,1> <1,1> ... <5,1>     <5,2> .. <10,2>      <10,3> .. <15,3>
```

For `c'` : `<0,1> <1,1> <2,1>` are instances of inner iterations and `<3,1>` is for outer iteration of `c'`
For `c` : `<0,1>.. <4,1> <5,2>.. <9,2> <10,3>.. <14,3>` are instances of inner iterations and `<5,1> <10,2> <15,3>` are for outer iterations of `c`

Figure 6.9: Control Dependencies in SemanticTM

x with values $\langle v, l \rangle$, all instructions that have an input data dependency originating from e can find the value read in $outDD$. Thus, once this `CAS` is executed, the value read becomes "visible" to the other processes. For this reason, when this `CAS` is successfully executed we say that p *reads* the value v for x.

6.4.2 Preliminaries

Consider any `cond` instruction c. The code (lines 26, 30, 37, 46, 48, 49, and 52) implies that the second parameter $iters$ of EXECUTEINS is returned by READITERATIONS (line 37) and the code (lines 56 - 58) implies that $iters.in = c.ins.cnt$. Recall that $c.ins.cnt$ is initialized with the value 0. Moreover, notice that $c.ins.cnt$ can only be updated with the `CAS` of line 106, which updates it from $iters.in$ to $iters.in + 1$.

Observation 2. *The following hold for the* $ins.cnt$ *field of a* `cond` *instruction:*

1. *it has the initial value* 0*,*
2. *it changes only on line 106, and*
3. *if it has the value* $l \geq 0$*, it can only change to* $l + 1$.

Fix any execution α of SemanticTM and consider a transactional instruction e. If e participates in the main block, denote by α_e the execution interval of α, that is defined as follows. If e finishes in α, then α_e starts with the initial configuration of α and ends with the configuration following the completion of e. If e does not complete in α, then $\alpha_e = \alpha$. We say that e has a single *outer iteration* whose execution interval is α_e.

If e is a `cond` instruction, let k be the larger integer assigned to $e.ins.cnt$ during α. We consider the execution intervals $\alpha^i_{e,in}$, $1 \leq i \leq k$, which are defined as follows. When the value i is written in $e.ins.cnt$, then $\alpha^i_{e,in}$ starts with the configuration following this assignment; also, if $i > 1$, $\alpha^{i-1}_{e,in}$ ends with the configuration preceding this assignment. If e completes, then $\alpha^k_{e,in}$ ends with the configuration following the completion of e. If e does not complete, then $\alpha^k_{e,in}$ is a suffix of α. We say that $\alpha^i_{e,in}$ is the *execution interval* of the ith *inner iteration* of e. Notice that when some working process p successfully executes the `CAS` instruction of line 106 and updates $e.ins.cnt$ to the value i, we say that p *initiates* the ith inner iteration of e and e is *completed* for its ith inner iteration. If $i > 1$, we also say that p *completes* the $(i - 1)$st inner iteration of e.

If e participates in the block of a `cond` instruction c, let k be the larger integer assigned to $c.ins.cnt$ during α. Then, let $\alpha^i_{e,out} = \alpha^i_{c,in}$, $1 \leq i \leq k$. We say that $\alpha^i_{e,out}$ is the execution interval of the ith *outer iteration* of e. Notice that each inner or outer iteration is associated with a unique number.

Consider a `cond` instruction c that either handles some loop or participates in some loop. Notice that c has both inner and outer iterations (given that it initiates at least one iteration of its loop), while any other `cond`, `read`, and `write` instruction has only outer iteration. Moreover, the execution interval of each of the inner iterations of c is included in (i.e. is a subsequence of) the execution interval of some outer iteration of c and the execution intervals of several inner iterations of c may be included in the execution interval of the same outer iteration of c. Also notice that, the execution interval of each of the inner iterations of c is also the execution interval of the outer iteration of each transactional instruction participating in c's block, and for each two instructions e_1 and e_2 participating in the block of c, it holds that the number k of outer iterations of e_1 is the same as that of e_2 and, for each i, $1 \leq i \leq k$, the execution interval of the ith outer iteration of e_1 is the same with the ith outer iteration of e_2.

We say that a working process p *reaches* a transactional instruction e for inner iteration in (if any) and outer iteration *out*, or simply for iteration $\langle in,out \rangle$, when an instance of READITERATIONS (line 37), executed during an instance *bi* of the body of the `while` loop of line 36 with $eptr = e$, returns $\langle in,out \rangle$; after p reaches e for $\langle in,out \rangle$ and before *bi* finishes, we say that p *examines* e for iteration $\langle in,out \rangle$. We say that a working process *skips* e for iteration $\langle in,out \rangle$, if it executes line 53 while it examines e for $\langle in,out \rangle$. We say that a working process *skips* e, if there exist iteration $\langle in, out \rangle$ such that the process skips e for $\langle in, out \rangle$. We say that a working process *selects* e for iteration $\langle in,out \rangle$, when $\langle e,\langle in,out \rangle \rangle$ is returned by an instance of GETACTIVEINS (line 26), initiated by this process; We say that a working process *selects* e, if there exist iteration $\langle in, out \rangle$ such that the process selects e for $\langle in, out \rangle$; in this case, we also say that the corresponding instance of GETACTIVEINS *selects* e. We say that a working process *executes* e for iteration $\langle in,out \rangle$, when it initiates an instance of EXECUTEINS (line 30) with first parameter e and second parameter $\langle in,out \rangle$. If e finishes while it is executed for outer iteration out, we suppose that e completes for out.

The code (lines 26, 30, 46, 48, 49, and 52) implies that p may execute a transactional instruction e for iteration $\langle in,out \rangle$ only if p selects e for this iteration, which happens after p reaches e for this iteration. The code (lines 30, 37, and 56 to 58) implies that EXECUTEINS takes as second parameter the $iters$ returned by READITERATIONS (line 37). If e participates in the main block, the code of READITERATIONS (lines 57 and 60) implies that $iters.out = 1$. Otherwise, the code of READITERATIONS (lines 58 and 59) implies that

$iters.out = e.pcond \rightarrow ins.cnt$. In this case, the code (lines 26, 27, 30, and 40) implies that a working process may execute an instance of EXECUTEINS only if $iters.out \geq 1$. So, in both cases $iters.out \geq 1$. Also, the inspection of the code of READITERATIONS and Observation 2 imply the following.

Observation 3. *Consider a transactional instruction e that is executed for iteration $\langle in, out \rangle$. Then, $in \geq 0$ and $out \geq 1$.*

Recall that a working process p initiates the ith inner iteration of e when it successfully executes the `CAS` instruction of line 106 and updates $e.ins.cnt$ to the value i. Before p initiates an inner iteration, it executes lines 98 to 102 to decide whether to initiate this iteration, or not. So, when p executes either line 99 or line 102, while executing a `cond` instruction c for inner iteration in, we say that p *decides* whether to initiate the $(in + 1)$st inner iteration of c (in which case $decision =$ `true`), or not (in which case $decision =$ `false`).

Consider any transactional instruction e that participates in some loop which is handled by a `cond` instruction c. The code (lines 30, 37, and 56 - 58) implies that the second parameter $iters$ of EXECUTEINS is returned by READITERATIONS (line 37). Since $e.pcond = c \neq null$, the code of READITERATIONS (lines 58 and 59) implies that $iters.out = e.pcond \rightarrow ins.cnt$. Recall that $c.ins.inCD[e.icond]$ is initialized with value 0. Moreover, notice that $c.ins.inCD[e.icond]$ can be updated with the `CAS` of either line 109 or line 120, which updates it from $iters.out - 1$ to $iters.out$, while e is executed for iteration $iters$. Observation 3 implies that $iters.out \geq 1$. Then, Observation 2 implies the following.

Observation 4. *The following hold for each element of the $ins.inCD$ array of a* `cond` *instruction:*

1. *it has the initial value* 0*,*
2. *it changes either on line 109, or on line 120, and*
3. *if it has the value $l \geq 0$, it can only change to $l + 1$.*

Consider any `cond` instruction c which has an outer control dependency d with some transactional instruction e, i.e. e is in the block of c. We say that d is *resolved* for the ith, $1 \leq i \leq k$ (where k is the larger integer assigned to $c.ins.cnt$ during α), inner iteration of c when the value i is written to $c.ins.cnt$. If $c.ins.cnt < i$, then d is *unresolved* for this iteration. Recall that $\alpha^i_{c,in}$ starts with the configuration following the update of $c.ins.cnt$ to i. So, each control dependency d from c to some transactional instruction e in c's block is *resolved* for the ith inner iteration of c, when $\alpha^i_{c,in}$ starts. So, when a working process initiates the ith inner iteration of c, it also *resolves* the outer output control dependencies of c for this iteration.

Moreover, recall that, if c either handles or participates in some loop, then c also has an inner control dependency d' originating from e. If $c.inCD[e.icond] = i$, $i \geq 1$, we say that d' is *resolved* for the ith inner iteration of c and e is *completed* for its ith (outer) iteration; otherwise, d' is *unresolved* and e is *not completed*, for this iteration. Observation 4 implies that any inner control dependency can be resolved at most once for each inner iteration. So, when some working process p executing e, successfully executes either the `CAS` of line 109 or the `CAS` of line 120, and updates $c.inCD[e.icond]$ from $i-1$ to i, we say that p *resolves* the inner output control dependency of e during its ith outer iteration and *completes* the execution of e for this iteration. Recall that $\alpha^i_{c,in} = \alpha^i_{e,out}$.

Assume now that c participates in some loop, i.e. it is in the block of a `cond` instruction c'. Then, if c is completed for the kth inner iteration of c', we say that each instruction e in c's block is completed for the kth inner iteration of c'. Notice that c' is an ancestor of e. This definition can be recursively applied on each other ancestor of e that is completed for some iteration.

We remark that, in the case that some transactional instruction e participates in the main block, its (outer) input control dependency is trivially *resolved* at the initial configuration.

Consider any `read` instruction r. The code (lines 30, 37, and 56 to 58) implies that EXECUTEINS takes as second parameter the $iters$ returned by READITERATIONS (line 37). If e participates in the main block, the code of READITERATIONS (lines 57 and 60) implies that $iters.out = 1$; otherwise, the code (lines 58 and 59) implies that $iters.out = e.pcond \rightarrow ins.cnt$. Recall that $r.ins.outDD.ver$ is initialized with value 0. Moreover, notice that $r.ins.outDD.ver$ can only be updated with the `CAS` of line 131, which updates it from $iters.out - 1$ to $iters.out$. This is so since RESOLVEDD is called on line 114 of EXECUTEINS with $iters.out$ as its second parameter. Observation 3 implies that $iters.out \geq 1$. So, Observation 2 implies the following.

Observation 5. *The following hold for the $ins.outDD.ver$ field of a* `read` *instruction:*

1. *it has the initial value* 0*,*

2. *it changes only on line 131, and*
3. *if it has the value $l \geq 0$, it can only change to $l + 1$.*

Consider any `write` or `cond` instruction wc which has an input data dependency from a `read` instruction r. We say that this dependency is *resolved* for the ith, $i \geq 1$, outer iteration of wc when $r.outDD.ver = i$; if $r.outDD.ver < i$, it is *unresolved* for this iteration. Observation 5 implies that any data dependency can be resolved at most once for each iteration. So, when some working process p successfully executes the `CAS` instruction of line 131 and updates $r.outDD.ver$ from $i-1$ to i, $i \geq 1$, we say that *p resolves* the output data dependencies of r for its ith (outer) iteration; also, we say that r is *applied* for its ith (outer) iteration when this `CAS` is successfully executed. If wc is a `cond` and e participates in the block of c, these concepts are defined for the inner iterations of c, similarly.

An instruction that has an unresolved input control dependency for some iteration is *inactive*; otherwise, it is *active* for this iteration. An instruction is in *waiting* state for some iteration, if at least one of its input (control or data) dependencies has not been resolved for this iteration; otherwise, it is *ready* for this iteration. By definition, each ready instruction is also active, for some iteration.

6.4.3 Correctness

Lemma 6. *Consider any transactional instruction* e *in α and assume that $\langle in, out \rangle$ are two integers such that* e *is selected for iteration $\langle in, out \rangle$. Let C be the final configuration of the execution interval of the first instance I_g of* GetActiveIns *at which* e *is selected. Then, the input control dependencies of* e *have been resolved for iteration $\langle$in,out$\rangle$, before C.*

Proof. Let p be the working process that executes I_g. We start by proving that the outer input control dependency of e has been resolved for outer iteration *out* before C. If e participates in the main block, then this claim holds trivially. Thus, assume that e participates in the block of some `cond` instruction c. To prove that the outer input control dependency of e has been resolved by C in this case, it suffices to argue that *c.inc.cnt* $\geq$ *out* by C. Let WL be the last instance of the body of the `while` loop of line 36 executed during I_g. By the code (lines 37, 58, and 59), it follows that *out* $=$ *c.ins.cnt* at the configuration that the instance of ReadIterations initiated by p on line 37 of WL returns. Since this configuration precedes C, the claim follows.

We continue by proving that the inner input control dependencies of e are resolved for inner iteration in before C. If e is not a `cond` instruction, or if it is a `cond` instruction that neither handles nor participates in some loop, then e has no inner iterations and this claim holds trivially. Assume that e is a `cond` instruction that either handles, or participates in some loop, or both. If e handles a loop without participating in some loop, the code (lines 47 to 50) implies that it is on line 49 that $\langle e, \langle in, - \rangle \rangle$ is returned; so, the instance of CHECKINNERCD initiated by p on line 49 of WL returns `true`. If e participates in some loop (independently of whether it handles a loop or not), the code (lines 41 - 46) implies that $\langle e, \langle in, - \rangle \rangle$ is returned on line 46; so, the instance of CHECKINNERCD initiated by p on line 45 of WL returns `true`. So, in both cases, the code of the corresponding instance of CHECKINNERCD (85 to 87) implies that the inner input control dependencies of e have been resolved for in before C. So, the lemma holds. □

Lemma 7. *Consider a* `cond` *instruction c that either handles or participates in some loop. Let e be any transactional instruction participating in the block of c and let cd be the entry for e in c.ins.inCD. Then, at any configuration of α, it holds that either cd $=$ c.ins.cnt or cd $=$ c.ins.cnt $- 1$.*

Proof. The proof is by induction on the configurations $C_1, C_2, \ldots$ of α. Fix any $i > 0$ and assume that the claim holds for C_{i-1}. We prove that the claim holds also for C_i. If $i = 1$, at the configuration C_1 (the initial configuration), recall that both *cd* and *c.inc.cnt* are initialized with value 0; so, the claim holds, in this case. Consider any $i > 1$ and let s be the step executed at C_{i-1} to get C_i. If s does not change neither *c.inc.cnt* nor *cd*, then claim holds by the induction hypothesis. We consider now that s changes either *c.inc.cnt* or *cd*.

We assume first that s changes *c.inc.cnt*. The code (lines 30 and 106) implies that *c.ins.cnt* can only be updated with the CAS of line 106, which updates it from $iters_c.in$ to $iters_c.in{+}1$, while c is executed for iteration $iters_c$. Observation 3 implies that $iters_c.in \geq 0$. The code (lines 26, 30, 37, 46, 48, 49, and 52) implies that the second parameter $iters_c$ of EXECUTEINS is returned by READITERATIONS (line 37). Since c is a `cond` instruction, the code (lines 56 - 58) implies that $iters_c.in =$ *c.ins.cnt*. If $iters_c.in = 0$, then s updates *c.ins.cnt* from 0 to 1 and, by the induction hypothesis, *cd* $= 0$ at C_i; so, the claim holds, in this case. We consider now that $iters_c.in > 0$. The code (lines 26, 30, 46, 48, 49, and 52) implies that c can be executed for iteration $iters_c$ only after c is selected for $iters_c$. Since c has an input inner control dependency from e, Lemma 6 implies that c can be selected for $iters_c$, only after this dependency is resolved for $iters_c.in$. So, at C_i it holds that $cd = iters_c.in$ and $c.ins.cnt = iters_c.in + 1$; so, the claim holds, in this case.

We assume now that s changes cd. The code (lines 30, 109, and 120) implies that cd can be updated with the `CAS` of either line 109 or line 120, which updates it from $iters_e.out - 1$ to $iters_e.out$, while e is executed for iteration $iters_e$. Observation 3 implies that $iters_e.out \geq 1$. The code (lines 26, 30, 37, 46, 48, 49, and 52) implies that the second parameter $iters_e$ of EXECUTEINS is returned by READITERATIONS (line 37). Since e participates in the block of c, the code (lines 58 and 59) implies that $iters_e.out = c.ins.cnt$. If $iters_e.out = 1$, then s updates cd from 0 to 1 and, by the induction hypothesis, $c.ins.cnt = 1$ at C_i; so, the claim holds, in this case. We consider now that $cd > 1$. The code (lines 26, 30, 46, 48, 49, and 52) implies that e can be executed for iteration $iters_e$ only after e is selected for $iters_e$. Since e has an input outer control dependency from c, Lemma 6 implies that e can be selected for $iters_e$, only after this dependency is resolved for $iters_e.out$. So, at C_i it holds that $c.ins.cnt = iters_e.out$ and $cd = iters_e.out$; so, the claim holds. □

Lemma 8. *Consider the di-list l_b of the bth, $b > 0$, data item x, and two consecutive transactional instructions e', e in l_b, where e' precedes e in l_b. Suppose that e' neither participates in some loop nor handles a loop, and let in and out be the maximum numbers of inner and outer iterations, respectively, such that e is selected for iteration $\langle in, out\rangle$. Then, for each i, j, $0 \leq i \leq in$, $1 \leq j \leq out$, e can be selected for iteration $\langle i, j\rangle$ only after e' finishes.*

Proof. To obtain a contradiction, suppose that there exist i, j such that some working process p selects e for iteration $\langle i,j\rangle$ before e' finishes. Consider the corresponding instance I_g of GETACTIVEINS executed by p that returns $\langle e,\langle i,j\rangle\rangle$. The code (lines 26, 35) implies that the first instance of the body of the `while` loop of line 36 during I_g is initiated with $eptr = List[b]$.

If I_g has returned (lines 46, 48, 49, and 52) the head of l_b, then $e = List[b]$. The code (line 31) implies that e becomes the head of l_b after e' finishes; this is a contradiction. So, I_g does not return $List[b]$. Then, the code (lines 26, 35, 43, and 53) implies that e is some transactional instruction following $List[b]$. Since, I_g returns e and e' precedes e in l_b, e' has to be skipped during some instance WL of the body of the `while` loop of line 36. Since, by assumption, e' neither participates in some loop nor handles a loop, and e' and e are consecutive in l_b, the code (lines 47, 48, and 52) implies that, during WL, I_g returns with e'

either on line 48 (if $e' = \texttt{cond}$) or on line 52 (if $e' \in \{\texttt{read}, \texttt{write}\}$). So, e' cannot be skipped during WL; this is a contradiction. □

Lemma 9. *Consider an instance I_g of* GetActiveIns *executed by some process p. Then, the following hold:*

1. *Let WL be an instance of the body of the* `while` *loop of line 36 during the execution of I_g in which p evaluates the statement of line 38 to false and let C be the configuration before this evaluation. If eptr points to some transactional instruction e and $innerExecutingCond_p \neq \texttt{null}$ at C, then $innerExecutingCond_p$ points to some transactional instruction e' that is an ancestor of e.*
2. *Let WL' be an instance of the body of the* `while` *loop of line 36 during the execution of I_g in which p selects a transactional instruction e. Assume that p skips at least one transactional instruction while executing I_g and let C' be the configuration before the last such skip. If $innerExecutingCond_p \neq \texttt{null}$ at C', then $innerExecutingCond_p$ points to some transactional instruction e' that is an ancestor of e.*

Proof. We start by proving claim 1. To obtain a contradiction, suppose that e' is not an ancestor `cond` of e. Since the statement of line 38 is evaluated by p as false, during WL and after C, and since $innerExecutingCond_p \neq null$ at C, the instance I_{pl} of PARTICIPATESINLOOP initiated by p for e returns `true`. By inspection of the code of I_{pl} (lines 62 - 63 and 68), it follows that $condptr$ starts from $e.pcond$ and traverses the ancestors of e, until either `null` or e' is reached. Since we assume that e' is not an ancestor `cond` of e, the code (line 69) implies that I_{pl} returns `false`; this is a contradiction.

We now prove claim 2. Notice that p selects e on one of the lines 46, 48, 49, and 52 during WL'. Thus, during the execution of WL', the statement of line 38 is evaluated by p as false; let C'' be the configuration before this evaluation. Notice that WL' is the last execution of the body of the `while` loop of line 36. So, the execution of WL' follows C' and C''. Since C' is the configuration before the last skip (of some transactional instruction) by p during I_g, the code (lines 53 and 36 - 38) implies that the value of $innerExecutingCond_p$ is the same at C and C'. Since we assume that $innerExecutingCond_p \neq null$ at C', then claim 1 implies the claim. □

Lemma 10. *Consider any instance I_g of* GetActiveIns *executed by some working process p. Then, during I_g and after the execution of line 35,*

1. *either* $innerExecutingCond_p = \texttt{null}$*, or* $innerExecutingCond_p \neq \texttt{null}$ *and* $innerExecutingCond_p$ *never changes back to* $\texttt{null}$*, and*
2. *if* $innerExecutingCond_p \neq \texttt{null}$*, then* $innerExecutingCond_p$ *points to some transactional instruction* e *that either handles or participates in some loop.*

Proof. The proof is by induction on the configurations $C_1, C_2, \ldots$ between the execution of line 35 and the response of I_g. Fix any $i > 0$ and assume that the claims hold for C_{i-1}. We prove that the claims hold also for C_i. If $i = 1$, then at the configuration C_1 (after the execution of line 35), $innerExecutingCond_p = \texttt{null}$, so the claims hold. Consider any $i > 1$ and let s be the step executed at C_{i-1} to get C_i. If s does not change $innerExecutingCond_p$, then claims hold by the induction hypothesis. We consider now that s changes $innerExecutingCond_p$. By inspection of the pseudocode, it follows that s (which follows C_1) is the execution of on one of the lines 50, 66, and 79. To prove claim 1 we argue below that s updates $innerExecutingCond_p$ with a value not equal to $\texttt{null}$.

If s is the execution of line 50, then *eptr* is written on $innerExecutingCond_p$. Then, the code (line 36) implies that *eptr* $\neq \texttt{null}$. Also, the code (lines 47 and 48) implies that *eptr* is a `cond` instruction and handles a loop; so, claim 2 holds, in this case.

If s is the execution of line 66, then *condptr* is written on $innerExecutingCond_p$. Then, the code (first condition of line 63) implies that *condptr* $\neq Null$. Also, the code (second condition of line 65) implies that *condptr* participates in some loop; so, claim 2 holds, in this case.

If s is the execution of line 79, then *condptr* is written on $innerExecutingCond_p$. Notice that *condptr* is initialized with *eptr.pcond* on line 73 and may be updated on line 83. Also, notice that FINDINNEREXECUTINGCOND is initiated with first parameter *eptr* (line 42). The code (line 36) implies that *eptr* $\neq \texttt{null}$; so, *eptr.pcond* $\neq \texttt{null}$. Also, the code (line 41) implies that *eptr* participates in some loop; so, *eptr.pcond* either handles or participates in some loop. Since *eptr.pcond* $\neq \texttt{null}$, the code (first condition of line 78 and line 83) implies that during the execution of the `while` loop (line 74) of FINDINNEREXECUTINGCOND, *condptr* may be updated with values different than `null`. So, claim 1 holds. Since *eptr.pcond* either handles or participates in some loop, the code (second condition of line 78 and line 83) implies that *condptr* either handles or participates in some loop. So, claim 2 holds. □

Lemma 11. *Consider a transactional instruction* e *that participates in some loop. Assume* *that a process p skips e and let C be the configuration preceding this skip. Then, at C:*

1. *$innerExecutingCond_p \neq$ `null` and points to some transactional instruction that is an ancestor `cond` of e, and*
2. *if $innerExecutingCondIter_p$ has the value $k \geq 0$, then the following hold: for each `cond` instruction c that is an ancestor of e and a descendant of the transactional instruction pointed to by $innerExecutingCond_p$, c has been completed for its outer iteration k.*

Proof. Recall that p skips e on line 53 during some instance WL of the body of the `while` loop of line 36 of some instance I_g of GETACTIVEINS (line 26).

We start by proving claim 1. Notice that during WL, p evaluates the statement of line 38 as false, since otherwise, WL would terminate on line 39 without executing line 53. We assume first that $innerExecutingCond_p = ac$ and $ac \neq$ `null` at the configuration C' before p evaluates the first condition of line 38, during WL. Thus, it must be that the instance I_{pl} of PARTICIPATESINLOOP initiated by p for e during WL returns `true`. Lemma 9 (claim 1) implies that ac is an ancestor of e. By inspection of the code of I_{pl} (lines 62 - 63 and 68), it follows that *condptr* starts from *e.pcond* and traverses the ancestors of e. Moreover, $innerExecutingCond_p$ may be updated with the value of *condptr* (line 66) only if *condptr* $\neq$ `null`. So, when I_{pl} returns, $innerExecutingCond_p \neq$ `null` and points to some transactional instruction that is an ancestor `cond` of e. Since e participates in some loop, line 41 is evaluated as true, so line 50 is not executed. Then, since $inerExecutingCond \neq$ `null`, the first condition of line 42 evaluates to false; thus, FINDINNEREXECUTINGCOND is not executed, so line 79 is not executed. Moreover, CHECKINNERCD does not change $innerExecutingCond$. Thus, $innerExecutingCond_p$ does not change by C. So, claim 1 holds, in this case.

We assume now that $innerExecutingCond_p =$ `null` at C'. Then, since e is skipped during WL, line 53 is executed. Since $innerExecutinCond =$ `null` at C', the first condition of line 38 is evaluated as false, so PARTICIPATESINLOOP is not executed (and thus line 66 is not executed). By the code (lines 41, 42, and 53), it follows that p evaluates the statement of line 42 as false during WL and before C (since otherwise it would execute the `continue` of line 44 and the skip would not occur). So, the instance I_f of FINDINNEREXECUTINGCOND initiated by p for e during WL returns `true`. By inspection of the code of FINDINNEREXECUTINGCOND (lines 79 - 81), it follows that $innerExecutingCond_p$ is updated with the value of *condptr* before I_f returns `true`. Moreover, by inspection of the code of FINDINNEREXECUTINGCOND (lines 73 - 74, 78 and 83), it follows that $condptr$ starts from $e.pcond$ (for which it holds $e.pcond \neq$ `null`, since e participates in some loop), traverses the ancestors of e, and $condptr$ never takes the value `null` (due to the first condition of line 78). So, when I_f returns, $innerExecutingCond_p \neq$ `null` and points to some transactional instruction that is an ancestor `cond` of e. Since CHECKINNERCD does not change $innerExecutingCond_p$ and line 50 is not executed, it follows that $innerExecutingCond_p$ does not change by C, claim 1 holds.

We finally prove claim 2. The code (line 34) implies that $innerExecutingCond_p$ is initialized to `null` during I_g. Since, by claim 1, $innerExecutingCond_p \neq$ `null` at C, it follows that, during I_g and before C, $innerExecutingCond_p$ has been updated at least once. Let C'' be the configuration before the last such update U of $innerExecutingCond_p$. Notice that $C'' < C$.

Since e participates is some loop, line 41 is evaluated as true, so line 50 is not executed. Therefore, by inspection of the code (lines 38 - 46), the only lines that can update $innerExecutingCond_p$ is either line 66 or line 79. Thus, U is a `write` of either line 66 or line 79 during some instance I_U of PARTICIPATESINLOOP or FINDINNEREXECUTINGCOND, respectively. By executing U, p writes the value of $condptr$ in $innerExecutingCond_p$ during an instance WL_U of the body of the while loop of either line 63 or line 74, respectively, with $condptr = innerExecutingCond_p$, during I_U. By the code (lines 62 - 63 and 68, and lines 73 - 74, 78 and 83, respectively), it follows that $condptr$ starts from $e.pcond$ and traverses the ancestors of e. The code (lines 67 and 80) implies that $innerExecutingCondIter_p$ is updated during WL_U and after the execution of U. If $innerExecutingCondIter_p$ is updated on line 67, it takes the value $conditer.in$. The code (lines 64 and 57 - 58) implies that $conditers.in = condptr \rightarrow ins.cnt$. If $innerExecutingCondIter_p$ is updated on line 80, it takes the value $outIter$. The code (line 76) implies that $outIter = conditers.in$. Also, the code (75 and 57 - 58) implies that $conditers.in = condptr \rightarrow ins.cnt$.

We consider any `cond` instruction c that is an ancestor of e and a descendant of the transactional instruction pointed to by $innerExecutingCond_p$. To obtain a contradiction, suppose that at C, c is not completed for its outer iteration with number k, the value stored in $innerExecutingCondIter_p$ at C. So, either c or an ancestor `cond` of c that is a descendant of $innerExecutingCond_p$ is not completed for its outer iteration; let c' be this `cond`. We start by proving the following Claim.

Claim 1. *Consider an instance WL' of the body of the while loop of either line 63 or line 74, executed by p before WL_U, with $condptr = ac$, where ac is an ancestor* `cond` *of e, during I_U. Then:*

1. *p evaluates the statement ST' of either line 65 or line 78 to false, and*
2. *$innerExecutingCond_p$ does not change before the beginning of WL_U.*

Proof. We start by proving claim 1. To obtain a contradiction, suppose that ST' is evaluated as true. In case ST' is the statement of line 78, the code (line 81) implies that WL_U is not executed; this is a contradiction. In case ST' is the statement of line 65, $innerExecutingCond_p$ takes the value ac on line 66, during WL'. During WL_U, the statement of line 65 is evaluated as true (since, otherwise, U is not executed) and therefore the fourth condition of this statement is evaluated as true. Thus, $innerExecutingCond_p = loop$ when the evaluation of the condition occurs, during WL_U. By inspection of the pseudocode, the value of $loop$ does not change during the execution of I_U. Since WL_U follows WL', WL_U is not the first instance of the body of the `while` loop of line 63 during which line 66 is executed. Thus, at the beginning of WL_U, $innerExecutingCond_p$ has the value that $condptr$ had at the beginning of some previous instance of this body. This is a contradiction, since then the second condition of the statement of the `while` loop of line 63 at the beginning of that instance would evaluate to false and that instance would not be executed. So, claim 1 holds.

We now prove claim 2. Claim 1 implies that if either line 65 or line 78 is executed during I_U and before the beginning of WL_U, then it evaluates to false. Thus, neither line 66 nor line 79 is executed during I_U and before the beginning of WL_U; so, claim 2 holds. □

We now continue with the proof of claim 2. Since c' is a descendant of $innerExecutingCond_p$ (and an ancestor of e), it follows that during I_U and before executing WL_U, p executes an instance WL' of the body of the while loop of either line 63 or line 74 with $condptr = c'$. Then, Claim 1 (claim 1) implies that during WL', p evaluates the statement ST' of either line 65 or line 78 to false. In the following, we derive a contradiction by arguing that ST' is evaluated as true.

Since $innerExecutingCond_p \neq$ `null`, Lemma 10 (claim 2) implies that $innerExecutingCond_p$ either handles or participates in some loop. Since c' is a descendant of $innerExecutingCond_p$, it follows that $c'.pcond \neq$ `null` and c' participates in some loop. So, if ST' is the statement of line 65, p evaluates to true the first two conditions of ST', whereas if ST' is the statement of line 78, p evaluates to false the first two conditions of ST'. We now argue that p evaluates the third condition of ST' to true. By inspection of the pseudocode (lines 64 and 75) $conditers$ has the value returned by READITERATIONS on line 64 or 75. Notice that READITERATIONS is called with parameter $condptr$, where $condptr$ points to c'. Since, $c'.pcond \neq \texttt{null}$, READITERATIONS returns on line 58. So, $conditers.out = c'.pcond \rightarrow ins.cnt$. Since, by assumption, c' is not completed for its outer iteration Lemma 7 implies that $c'.pcond \rightarrow ins.inCD[c'.icond] = c'.pcond \rightarrow ins.cnt - 1$, when ST' is executed. So, p evaluates the third condition of ST' to true.

In case ST' is the statement of line 78, then it is evaluated as true; this is a contradiction. We consider now that ST' is the statement of line 65. To prove that ST' is evaluated as true, we argue that the fourth condition of ST' is evaluated as true (since we have already argued that the other three condition of ST' are evaluated as true). Claim 1 (claim 2) implies that $innerExecutingCond_p$ does not change during I_U and before the beginning of WL_U. Since PARTICIPATESINLOOP is initiated with its second parameter equal to the value of $innerExecutingCondIter_p$ at the beginning of its invocation (line 38), then the fourth condition of ST' is evaluated as true; this is a contradiction. □

Recall that SemanticTM assumes that the transactions of an execution α are processed one after the other. Let $T_1, T_2, \ldots$ be this order. Let σ be the sequential execution of transactions $T_1, T_2, \ldots$ in this order. Let T_i, $i > 0$, be any of these transactions. Assume that $k^i \geq 0$ is the number of cond instructions of T_i. Let $c_0, c_1, \ldots, c_{k^i}$ be the cond instructions of T_i, in the order they appear in the code of T_i. Notice that c_0 is the fictitious cond of the main block of T_i. For each cond instruction c_j, $0 \leq j \leq k^i$, let $k^i_j \geq 0$ be the number of *iterations* executed in σ for c_j. We call these iterations *inner* iterations of c_j. If some cond c (or any other instruction e) participates in the block of c_j, then the inner iterations of c_j are *outer* iterations of c. Let e be any of the transactional instructions that participate in the block of c_j. If e is not a cond, then e does not have inner iterations. For each ℓ, $1 \geq \ell \geq k^i_j$, we call *instance* of e for iteration $\langle 0, l \rangle$ the ℓth instance of e in σ. For each cond $c_{j'}$, $j \geq j' \geq k^i_{j'}$, that participates in the block of c_j, if $k^i_{j,\ell}$ is the number of inner iterations of $c_{j'}$ executed for the ℓth inner iteration of c_j, for each ℓ', $0 \leq \ell' \leq k^i_{j,\ell}$, we call *instance* of e for iteration $\langle \ell', \ell \rangle$ the $(\sum_{t=1}^{\ell-1} k^i_{j,t} + \ell')$th occurrence of $c_{j'}$ in σ, where $\sum_{t=1}^{\ell-1} k^i_{j,t}$ is the number of occurrences of c'_j in the inner iterations of c_j that precede the

$(\ell - 1)$st inner iteration of c_j.

Consider an instance δ of a transactional instruction e for iteration $\langle in, out \rangle$ in σ. If e is executed for iteration $\langle in, out \rangle$ in α, then this execution of e in α, denoted by δ^{α}, is called the instance of e for iterations $\langle in, out \rangle$ in α and it corresponds to δ; otherwise, $\delta^{\alpha} = \perp$. If δ is not the last instance of e for its outer iteration *out* in σ, then c must be a `cond` that either handles or participates in some loop. We say that δ is *completed* in α when e is completed for its inner iteration in in α. Otherwise, δ is *completed* when e is completed for its outer iteration *out*. When a working process executes (selects or applies) e for $\langle in{,}out \rangle$ in α, we also say that p executes (selects or applies) δ, or δ is executed (selected or applied).

Observation 12. *Consider a transactional instruction* e *and let* in, *out be any two integers such that* δ *is an instance of* e *for iteration* $\langle in{,}out \rangle$ *in* σ*. Let* C *be a configuration at which* e *is completed. Then, the following hold:*

1. *if* $e \neq$ `cond`*, then* e *has no inner iterations (so* $in = 0$*) and* e *has a single instance for iteration out,*
   (a) e *is completed for its outer iteration out at* C*, and*
   (b) *if* e *participates in some loop, then the inner output control dependency of* e *has been resolved for its outer iteration out by* C*,*
2. *if* $e =$ `cond` *and if, before* C*, the process executing* e *has decided whether to initiate the next inner iteration* $in + 1$*, then*
   (a) *if* $decision =$ `false`*, then*
      i. *at* C*,* e *is completed for its outer iteration out, and*
      ii. *if* e *participates in some loop, then the inner output control dependency of* e *has been resolved for its outer iteration out by* C*.*
   (b) *if* $decision =$ `true`*, then at* C*:*
      i. e *has been completed for its* $(in + 1)$*st inner iteration, and*
      ii. *if* e *neither participates in some loop nor handles some loop, then* e *is completed for its outer iteration out.*

We remark that a partial order $\mathcal{PO}$, denoted by $<_{\mathcal{PO}}$, exists among the instances of the transactional instructions $(T_1, T_2, \ldots)$ executed in σ. Notice that this order is partial, since two transactional instructions may neither have any dependency with each other nor access the same data item; we say that these instructions are *independent*, otherwise, they are *dependent*. More specifically, considering two dependent transactional instructions e

and e', which are transactional instructions of transactions T_i and T_j, respectively, where $i, j > 0$:

- if a dependency exists among e, e' (so $i = j$) and if, in σ, e initiates $k_{in} \geq 0$ inner iterations and is executed in $k_{out} > 0$ outer iterations, then
  - if e has an output data dependency leading to e', and e, e' participate to the same block, then the instance δ of e for each iteration $\langle i, j\rangle$, $0 \leq i \leq k_{in}$, $1 \leq j \leq k_{out}$, *precedes* in $\mathcal{PO}$ the instance δ' of e' for this iteration, or $\delta <_{\mathcal{PO}} \delta'$,
  - if e has an outer output control dependency leading to e', then
    * the instance δ of e for each iteration $\langle i, j\rangle$, $0 \leq i \leq k_{in}$, $1 \leq j \leq k_{out}$, *precedes* in $\mathcal{PO}$ the instance δ' of e' for iteration $\langle in', i{+}1\rangle$, where $in' \geq 0$ is the number of the last inner iteration of e' executed for the ith outer iteration of e', or $\delta <_{\mathcal{PO}} \delta'$, and
    * if e handles a loop, then it has an inner input control dependency originating from e'. Let δ be the instance of e for each iteration $\langle i, j\rangle$, $1 \leq i \leq k_{in}$, $1 \leq j \leq k_{out}$, and δ' be the instance of e' for iteration $\langle in', i\rangle$, where $in' \geq 0$ is the number of the last inner iteration of e' executed for the ith outer iteration of e'. Then, δ' *precedes* δ in $\mathcal{PO}$ δ, or $\delta' <_{\mathcal{PO}} \delta$.
- if e, e' access the same data item x, no dependency exists among e, e', and either $i = j$ and e precedes e' in sequential order defined by the sequential semantics of T_i, or $i < j$, then the last instance δ of e *precedes* in $\mathcal{PO}$ the first instance δ' of e'.

Consider two instances δ and δ'. If $\delta <_{\mathcal{PO}} \delta'$, then $\delta' >_{\mathcal{PO}} \delta$ and we say that δ' follows δ in $\mathcal{PO}$. We also say that δ and δ' are *dependent* in $\mathcal{PO}$. If any instance δ'' exist such that $\delta <_{\mathcal{PO}} \delta'$ and $\delta' <_{\mathcal{PO}} \delta''$, then $\delta <_{\mathcal{PO}} \delta''$.

Let δ be an instance of a transactional instruction e for some iteration $\langle in, out\rangle$. We define the *next* instance of e to be some instance δ' of e for some iteration $\langle in', out'\rangle$ such that $\delta <_{\mathcal{PO}} \delta'$, and there is no other instance δ'' of e such that $\delta <_{\mathcal{PO}} \delta''$ and $\delta'' <_{\mathcal{PO}} \delta'$. Also, we say that an instance $delta'$ is *consecutive* to an instance δ, if $\delta <_{\mathcal{PO}} \delta'$, and there is no other instance δ'' such that $\delta <_{\mathcal{PO}} \delta''$ and $\delta'' <_{\mathcal{PO}} \delta'$. Consider the sequence of instances of transactional instructions that write the same data item x in σ. Then, we say that the ith, $i > 0$, instance of this sequence is the ith write on x.

Lemma 13. *Let δ be an instance of some transactional instruction e for iteration $\langle in, out\rangle$ and let δ_{next} be the next instance of e in σ. Then, if $\delta^{\alpha} \neq \bot$ and $\delta^{\alpha}_{next} \neq \bot$, then δ_{next} can be executed only after the completion of δ.*

Proof. Since e has more than one instances, e either handles or participates in some loop. If δ is not the last instance of e for out, then e handles a loop, δ is an instance of e for in, and δ_{next} is an instance of e for inner iteration $in + 1$. Let ch be any of the transactional instructions participating in the block of e. Consider any instance I_g of GETACTIVEINS at which ch is selected for iteration $\langle -, in + 1\rangle$. Lemma 6 implies that the input control dependencies of ch have been resolved for $\langle -, in + 1\rangle$, before the final configuration C of the execution interval of I_g. Since ch has an input control dependency from e, it follows that the outer input control dependency of ch from e has been resolved for $\langle -, in + 1\rangle$, i.e. $e.ins.cnt = in + 1$, by C. So, e is completed for its inth inner iteration by C. Therefore δ is completed by C. Moreover, since e has an (inner) input control dependency from ch, by following similar arguments, Lemma 6 implies that δ_{next} can be selected only after the completion of ch for its (in + 1)st outer iteration. Since before an instance is executed it has first to be selected, it follows that δ_{next} is executed only after the completion of δ. The claim holds, in this case.

We consider now that δ is the last instance of e for out. Then, δ is an instance of e for out and, since δ is not the last instance of e, e participates in some loop. Let $pc = e.pcond$. Since pc has an inner input control dependency from e, by following similar arguments, Lemma 6 implies that pc can be selected for its outth inner iteration only after the completion of δ. So, pc completes for its (out + 1)st inner iteration only after the completion of δ. Moreover, since e has an input control dependency from pc, by following similar arguments, Lemma 6 implies that δ_{next} can be selected only after the completion of pc for its (out + 1)st inner iteration. Since before an instance is executed it has first to be selected, it follows that δ_{next} is executed only after the completion of δ. The claim holds. □

Lemma 14. *Consider two transactional instructions e, e' such that e, e' are consecutive transactional instructions of the same block, e, e' access the bth, $b > 0$, data item x and they are placed in the di-list l_b of x, and e' is placed before e in l_b. If there exist two integers in, out such that δ_1 is an instance of e for iteration $\langle in, out\rangle$ in σ and δ_2 is an instance of e' in σ such that $d_2^{\alpha} \neq \bot$ and δ_1 is consecutive to δ_2. Then, if e is selected for iteration $\langle in, out\rangle$, this occurs after the completion of δ_2.*

Proof. To obtain a contradiction, suppose that some working process p selects e for $\langle in,out\rangle$ before the completion of δ_2. Notice that since e' and e participate to the same block δ_2 is the instance of e' for out.

Consider the instance I_g of GETACTIVEINS executed by p that returns $\langle e,\langle in,out\rangle\rangle$ during some instance WL of the body of the `while` loop of line 36; let C be the configuration before this happens (i.e. before p selects e for $\langle in,out\rangle$). The code (lines 26, 35) implies that the first instance of the body of the `while` loop of line 36 during I_g is initiated with $eptr = List[b]$. If I_g has returned (lines 46, 48, 49, and 52) the head of l_i, then $e = List[b]$. The code (line 31) implies that e becomes the head of l_b after the completion of e'; this is a contradiction.

So, I_g does not return $List[b]$. Then, the code (lines 26, 35, 43, and 53) implies that I_g returns some other transactional instruction following $List[b]$. Since, I_g returns e and e' precedes e in l_b, e' has to be skipped during some instance WL' of the body of the `while` loop of line 36; let C' be the configuration preceding the execution of this line (i.e. before p skips e'). Notice that $C' < C$. If e' neither (is a `cond` and) handles nor participates in some loop, then the code (lines 41, 47 - 48, and 52) implies that e' is returned either on line 48 or on line 52, and can not be skipped; this is a contradiction. So, e' either handles or participates in some loop.

We consider first that e' and e are consecutive in l_b. Since e' and e participate in the same block, it follows that e' participates in some loop and does not handle a loop (since, if e' handles a loop, then e' and e can not be consecutive in l_b). Since e' participates in some loop, the code (lines 41, 45, and 53) implies that before p skips e' it evaluates as false the statement of line 45, during WL'; so, the first condition of this statement is evaluated as false. However, since this condition is evaluated by p for e' during WL' and before C', $C' < C$, and e' completes for out after C', Lemma 7 implies that $e'.pcond \rightarrow ins.inCD[e'.icond] = out - 1$ when this condition is evaluated. So, this conditions is evaluated as true; this is a contradiction.

We consider now that e' and e are not consecutive in l_b; so $e' =$ `cond` and handle a loop. Let e'' be the last transactional instruction skipped by p, before e is selected by p for $\langle in,out\rangle$. Notice that e'' participates in the loop of e' and, since e' and e participate in the same block and are consecutive in this block, e' is one of the ancestor `conds` of e''. We remark that p skips e'' on line 53 during some instance WL'' of the body of the `while` loop of line 36; let C'' be the configuration preceding the execution of this line (i.e. before p skips e''). Notice that $C' < C'' < C$.

Since e'' participates in some loop and it is skipped before C'', Lemma 11 (claim 1) implies that $innerExecutingCond_p = ac \neq$ `null`, where ac is one of the ancestor `conds` of e'', before C''; also, let $innerExecutingCondIter_p = inIter$, where $inIter \geq 0$.

If e' and e do not participate in some loop, notice that since e is selected by p for iteration $\langle e$,$\langle in$,$out\rangle\rangle$, during WL the statement of line 38 is evaluated as false, after C''. Since $innerExecutingCond_p \neq null$ before C'', the first condition of line 38 is evaluates as true, during WL, and since e does not participate in some loop, the code (lines 38, 62 - 63, and 69) implies that the second condition of line 38 is also evaluated as true, during WL. So, the statement of line 38 is evaluated as true during WL; this is a contradiction.

So, e' and e participate in some loop; let *cpc* be $e.pcond$, or equivalently $e'.pcond$. Since $innerExecutingCond_p \neq null$ before C'', Lemma 9 (claim 2) implies that e can be selected during WL for $\langle in$,$out\rangle$ by p only if ac is either *cpc* or one of the ancestor `conds` of *cpc*. Since e' is an ancestor `cond` of e'' and a descendant of *cpc*, Lemma 11 (Claim 2) implies that this is possible only if e' is completed for the inner iteration with number $inIter$ of ac, before C''. Let out' be the maximum number of e''s outer iteration for which e' is completed before C''. Since, e' completes for *out* after C and $C > C''$, it follows that $out' < out$. During WL and before C, p initiates an instance I_{pl} of PARTICIPATESINLOOP with parameters $e, ac, inIter$, which returns `true`. Notice that e is reached for $\langle in, out\rangle$ during WL (line 37) and before the initiation of I_{pl}; let C''' be the configuration before e is reached. In case $ac = cpc$, then $inIter \leq out'$, or $inIter < out$. Also, at C''', $cpc.ins.cnt = out$, or equivalently $ac.ins.cnt \neq inIter$. In case $ac \neq cpc$, since e' is completed for both out' and $inIter$ before C'', and since ac is ancestor `cond` of *cpc*, by recursively applying Lemma 6, it follows that e can be reached for $\langle in, out\rangle$ only after ac initiates an iteration larger than $inIter$. So, in both cases, it holds that $ac.ins.cnt \neq inIter$ at C'''. Since ac is an ancestor `cond` of e, the code of I_{pl} (lines 62 - 63 and 68) implies that *condptr* takes the value ac during some instance of the body of the `while` loop of line 63. Then, since I_{pl} is initiated with second parameter ac, so $loop = ac$ during I_{pl}, the second condition of the subsequent execution of the statement of the `while` of line 63 evaluates to false. So, the statement of line 69 is evaluated with $condptr = ac$. Notice that I_{pl} is initialized with its third parameter $inIter$, so $loopIter = inIter$ during I_{pl}. Since, $ac \neq$ `null` and $ac.ins.cnt \neq inIter$, where $inIter = loopIter$, the statement of line 69 evaluates to false and I_{pl} returns `false`; this is a contradiction. □

Lemma 15. *Consider some transactional instruction e that does not participate in some loop and it is in the head of the di-list l_x of some data item x. Let in, out be integers such that δ is an instance of e that is not completed for iteration $\langle in, out\rangle$. Assume that the* *input data and control dependencies of* e *are resolved for* $\langle in, out\rangle$ *and let* C *be the first configuration at which all dependencies of* e *have been resolved for iteration* $\langle in, out\rangle$*. If the iteration of* e *does not change after* C*, then some process selects* e *for* $\langle in, out\rangle$ *after* C*.*

Proof. To obtain a contradiction, suppose that no process selects e for $\langle in, out\rangle$ after C. Since e is in the head of l_x, the code (line 26) implies that at least one instance of GETACTIVEINS is initiated with parameter e, after C. Then, the code (lines 35 and 36) implies that at least one iteration of the `while` loop of line 36 is executed with $eptr = e$; let WL be the first of them and let p be the working process executing WL.

During WL, p initiates an instance of READITERATIONS after C and since , by assumption, the outer input control dependency of e is resolved before $C_{\delta'}$, the iterations of e do not change after C, and e does not participates in some loop, the code of this instance (lines 56 - 60) implies that $in \geq 0$, $out = 1$, and p reaches e for iteration $\langle in, out\rangle$. Then, the code (line 34) implies that p evaluates to false the first condition (and the statement) of line 38 and, since $out = 1$, it also evaluates to false the statement of line 40.

Since e does not participate in some loop, p evaluates to false the condition of line 41. In case $e = \{\texttt{read}, \texttt{write}\}$, then GETACTIVEINS responds with $\langle e, \langle in, out\rangle\rangle$ (line 52); this is a contradiction. In case $e = \texttt{cond}$ and e does not handle a loop, then GETACTIVEINS responds with $\langle e, \langle in, out\rangle\rangle$ (line 48); this is a contradiction. In case $e = \texttt{cond}$ and e handles a loop, an instance I_{cd} of CHECKINNERCD is initiated with parameters e and in. Since , by assumption, all the inner input control dependencies of e are resolved before C and these iterations do not change after C, the code (lines 85 - 87) implies that I_{cd} returns true. So, the condition of line 49 evaluates to true and GETACTIVEINS responds with $\langle e, \langle in, out\rangle\rangle$; this is a contradiction. □

Lemma 16. *Consider some transactional instruction* e *that either handles or participates in some loop* L *and it is placed in the di-list* l_x *of some data item* x*. If* e *participates in* L*, let* e_1 *be the first transactional instruction of* L *that is placed in* l_x *(it may be that* $\mathrm{e}_1 = \mathrm{e}$*); otherwise, if* e *handles* L*, let* $\mathrm{e}_1 = \mathrm{e}$*. Assume that* e *is the kth,* $k \geq 1$*, transactional instruction of* L *that is placed in* l_x*. Assume that some process* p *initiates an instance* I_g *of* GetActiveIns *for* e_1 *and let* C *be the configuration preceding this initiation. Let* δ *be an instance of* e *for iteration* $\langle in, out\rangle$*. Assume that any instance of some instruction that precedes* δ *in* $\mathcal{PO}$ *is completed before* C *and that no other instance can be executed after* C*. Then,*

1. *if δ is not completed before C, then p selects e for $\langle in, out\rangle$.*
2. *otherwise, if δ is completed before C, then p skips e during the kth iteration of the* `while` *loop of line 36 of I_g.*

Proof. Since, by assumption, any instance of some instruction that precedes δ in $\mathcal{PO}$ is completed before C, it follows that the input control dependencies of e for $\langle in,\ out\rangle$ are resolved before C.

Let $S = e_1, e_2, \ldots, e_k$, $k \geq 0$, be the (possibly empty) sequence of transactional instructions that precede e in l_x that either handle (i.e. e_1) or participate in L. The proof is by induction on the value of k. Fix any $k > 0$ and assume that the claims hold for $k-1$. We prove that the claims hold also for k. Let WL be the kth instance of the body of the `while` loop of line 36, during I_g.

We first argue that p reaches e for $\langle in, out\rangle$ during WL. If $k = 1$, then the code (lines 35 and 36) implies that p executes WL with $eptr = e_1 = e$. If $k > 1$, then, by the induction hypothesis, claim 2 implies that p skips e_{k-1} during the $(k-1)$st instance of the body of the `while` loop of line 36 and the code (lines 53 and 36) implies that p executes WL with $eptr = e_k = e$. So, for $k \geq 1$, p initiates an instance of READITERATIONS (line 37) during WL. Since, by assumption, the input control dependencies of e for $\langle in,\ out\rangle$ are resolved before C and no instance of e is executed after C, by inspection of the code of READITERATIONS with parameter e (lines 56 - 60), it follows that p reaches e for iteration $\langle in, out\rangle$ during WL; so, claim holds. Let C' be the configuration before p reaches e for iteration $\langle in, out\rangle$ during WL.

We next argue that p evaluates to false the statement of line 38. If $k = 1$, the code (line 34) implies that p evaluates to false the first condition (and the statement) of line 38; so claim holds, in this case. If $k > 1$, then, by the induction hypothesis, claim 2 implies that p skips e_{k-1} during the $(k-1)$st instance of the body of the `while` loop of line 36. Observation 11 (Claim 1) implies that $innerExecutingCond_p \neq$ `null`, $innerExecutingCond_p = ac$, where ac is one of the ancestor `conds` of e_{k-1}, before C'; also, let $innerExecutingCondIter_p = inIter$. So, p evaluates as true the first condition of line 38 and continues by initiating an instance I_{pl} of PARTICIPATESINLOOP with parameters e, ac, and $inIter$. Since, by assumption, any instance of some instruction that precedes δ in $\mathcal{PO}$ is completed before C and that no other instance can be executed after C, it follows that ac is also an ancestor `cond` of e. Since, by assumption, the inner control dependencies of ac for $inIter$ are resolved before C and no other instance of ac is executed before the return of I_g, and the code (lines 38 and 62 - 70) imply that I_{pl} returns `true` and p evaluates to false the second condition (and the statement) of line 38; so, claim holds.

We now argue that p evaluates as false the statement of line 40. If e does not participate in some loop and handles a loop, the outer input control dependency of e is (trivially) resolved, i.e. $out = 1$. Otherwise, if e participates in some loop, by assumption, the outer input control dependency of e is resolved; so, $out \geq 1$. Thus, in both cases, p evaluates as false the statement of line 40; so, claim holds.

If e does not participate in some loop and handles a loop, then the code (lines 47 - 49) implies that p initiates an instance I_{cd} of CHECKINNERCD with parameters ept and in.

Otherwise, if e participates in some loop, e evaluates as true the condition of line 41 and continues by evaluating the statement of line 42. We argue that p evaluates as false this statement. If $k > 1$, then the induction hypothesis (claim 2) and Observation 11 (claim 1) imply that p evaluates as false the first condition (and the statement) of line 42; so, claim holds. If $k = 1$, then the code (line 34) implies that p evaluates as true the first condition of line 42 and continues by initiating an instance I_f of FINDINNEREXECUTINGCOND with parameters e and out (line 42). Since any instance of some instruction that precedes δ in $\mathcal{PO}$ is completed before C and since no subsequent instance of this instruction can be executed after C then the code (lines 97 - 121) implies that the iterations of all these instructions do not change while p is executing I_f. So, the code (lines 72 - 83) implies that I_f returns `true` and p evaluates as false the second condition of line 42; so, claim holds. So, p continues by evaluating the statement of line 45.

We now proceed to prove claim 1. We consider first that e does not participate in some loop and handles a loop. Since, by assumption, the inner control dependencies of e are resolved for in before C and (the iterations of) these dependencies do not change before I_g returns (that is also before I_{cd} returns), the code of I_{cd} (lines 85 - 87) implies that I_{cd} returns `true`. Thus, the code (line 49) implies that p selects e for $\langle in, out \rangle$.

We now consider that e participates in some loop. Since δ is not completed, p evaluates as true the first condition of line 45. If $e \in \{\texttt{read}, \texttt{write}\}$, then p evaluates as true the second condition of line 45 and selects e for $\langle in, out \rangle$. If $e = \texttt{cond}$, then p initiates an instance I'_{cd} of CHECKINNERCD with parameters e and in (third condition of line 45). Since, by assumption, the inner control dependencies of e are resolved for in before C and (the iterations of) these dependencies do not change before I_g returns (that is also before I'_{cd} returns), the code of I'_{cd} (lines 85 - 87) implies that I'_{cd} returns `true`. Thus, the code (line

46) implies that p selects e for $\langle in, out\rangle$. Thus, claim 1 holds.

Finally, we prove claim 2. We consider first that e does not participate in some loop and handles a loop. Since e either handles or participates in some loop, and δ is completed for iteration $\langle in, out\rangle$, it follows that $in \geq 1$. Since e handles a loop, by assumption, the inner input control dependencies of e are resolved for $in - 1$ before C and they cannot be resolved for a subsequent (inner) iteration of e before I_g returns (i.e. before the return of I_{cd}). So, by inspection of the code of I_{cd} (lines 85 - 87), I_{cd} returns `false` and the condition of line 49 is evaluated as false, during WL. Then, the code (line 50 - 51 and 53) implies claim 2, when e does not participate in some loop and handles a loop.

We consider now that e participates in some loop. Since e is completed for out, p evaluates as false the first condition of the statement of line 45 and p continues by executing line 53 (with which it skips e) during WL; so, claim 2 holds. □

Lemma 17. *Consider some transactional instruction e that is in the head of the di-list l_x of some data item x. Let δ be an uncompleted instance of e for iteration $\langle in, out\rangle$. Assume that some process p selects e for $\langle in, out\rangle$ and let C be the configuration preceding this. In case $e =$ `cond` and δ is the first instance of e for out, assume that $e.ins.startinneriter = in$ after (and at) C. Moreover, in case $e =$ `cond` and δ is not the first instance of e for out, assume that $e.ins.startinneriter \neq in$ after (and at) C. If the input data and control dependencies of e are resolved for $\langle in, out\rangle$ before C, then some process executes e for $\langle in, out\rangle$ after C.*

Proof. Since p selects e for $\langle in, out\rangle$ (line 26), it initiates an instance I_{dd} of CHECKDD on line 28 with parameters e and $\langle in, out\rangle$, after C.

We first argue that I_{dd} returns $\langle \texttt{true}, \perp\rangle$. In case $e =$ `read`, the code (line 89) implies that I_{dd} returns $\langle \texttt{true}, \perp\rangle$; so, claim holds. We consider now that $e \in \{\texttt{write}, \texttt{cond}\}$. In case, $e =$ `cond`, by assumption, $e.ins.startinneriter = in$, only if δ is the first instance of e for out. So, in case either $e =$ `cond` and δ is the first instance of e, or $e =$ `write`, the statement of line 90 evaluates as true. Moreover, since, by assumption, the input data dependencies of e are resolved for out before C, the code (lines 91 - 94) implies that I_{dd} returns $\langle \texttt{true}, -\rangle$ (line 94); so, claim holds. Otherwise, if $e =$ `cond` and δ is not the first instance of e for out, then, by assumption, $e.ins.startinneriter \neq in$. So, the statement of line 90 is evaluated as false and I_{dd} returns $\langle \texttt{true}, -\rangle$ (line 95); so, claim holds.

Since, I_{dd} returns $\langle \texttt{true}, - \rangle$, the code (lines 29 and 30) implies that p executes e for $\langle in, out \rangle$, after C. $\square$

Lemma 18. *Let δ be an uncompleted instance of some transactional instruction e for iteration $\langle in, out \rangle$. Assume that some process executes e for $\langle in, out \rangle$ and let p be the first of them. Let C be the configuration before p starts the execution of e for $\langle in, out \rangle$. Assume that any instance of e that precede δ in $\mathcal{PO}$ is completed before C and that no instance of e that follows δ completes after C. In case δ is the last instance of e for out, $e = \texttt{cond}$, e does not handle a loop, and e participates in some loop, assume that $e.ins.startinneriter = in - 1$, before C and does not change after C. Moreover, in case δ is not last instance of e for out, $e = \texttt{cond}$, e does not handle a loop, and e participates in some loop, assume that $e.ins.startinneriter \neq in - 1$, before C and does not change after C. Then, the following hold, in this order:*

1. *δ completes (i.e. if δ is not the last instance of e for out, then e completes for in, otherwise, if δ is the last instance of e, then e completes for out), and*
2. *if δ is the last instance of e and e does not participate in some loop, then*
   (a) *if e = cond and its condition is evaluated as false, then any instruction participating in the block of e, and*
   (b) *e finishes.*

Proof. Since, by assumption, any instance of e that precede δ in $\mathcal{PO}$ is completed before C and that no instance of e that follows δ completes after C, it follows that the control dependencies of e for $\langle in,\ out \rangle$ are resolved before C and after C they may be resolved only for iteration $\langle in,\ out \rangle$. Notice that in case either δ is not the last instance of e or e participates in some loop, then claim 2 trivially holds.

We consider first that $e \in \{\texttt{read}, \texttt{write}\}$. If e does not participate in some loop, then δ is the last instance of e and the code (lines 113 and 121) implies that some process finishes e (also, e completes for *out*); so, 1 and 2b hold. If e participates in some loop, then the code (lines 113 and 119 - 120) implies that at least one process executes an instance of the CAS of line 120; let CS' be the first instance of this CAS executed by these processes. Since δ is not completed and, by assumption, the control dependencies of e for $\langle in,\ out \rangle$ are resolved before C and after C they may be resolved only for iteration $\langle in,\ out \rangle$, then at C it holds that $e.pcond \rightarrow ins.inCD[e.icond] = out - 1$. So, since $CS' > C$, CS' is successful and claim 1 holds.

We consider now that $e = \texttt{cond}$. If δ is the last instance of e, it follows that either some process evaluates as true the condition of e and e neither handles a loop nor participates in some loop, or some process evaluates as false the condition of e, We consider first that some process evaluates as true the condition of e and e neither handles a loop nor participates in some loop. Then, at least one process executes the CAS of line 106; let CS'' be the first instance of this CAS executed by these processes. Since δ is not completed and, by assumption, the control dependencies of e for $\langle in, out\rangle$ are resolved before C and after C they may be resolved only for iteration $\langle in, out\rangle$, then at C it holds that $e.ins.cnt = in$. So, since $CS'' > C$, CS'' is successful and claim 1 holds. Moreover, the code (lines 107) implies that some process completes e; so, claim 2b holds.

We consider now that some process evaluates as false the condition of e. If e does not participate in some loop, then the code (lines 100 and 102) implies that $decision = \texttt{false}$ and the code (lines 110 - 112 and 141 - 143) implies that some process finishes any instruction participating in the block of e and e, in this order (also, e completes for out); so, claims 1, 2a, and 2b hold. Otherwise, if e participates in some loop, then e may either handle a loop or not. In the former case, the code (lines 100 and 102) implies that $decision = \texttt{false}$. In the latter, since δ is the last instance of e for out, by assumption, $e.ins.startinneriter = in - 1$ before C and can not change after C. Also, by assumption, $e.pcond \rightarrow ins.cnt = out$ before C and can not change after C. So, some process evaluates as true the statement of line 98 and the code (line 99) implies that $decision = \texttt{false}$. Then, in both cases, the code (lines 108 - 109) implies that at least one process executes the CAS of line 109; let CS''' be the first instance of this CAS executed by these processes. Since δ is not completed and, by assumption, the control dependencies of e for $\langle in, out\rangle$ are resolved before C and after C they may be resolved only for iteration $\langle in, out\rangle$, then at C it holds that $e.pcond \rightarrow ins.inCD[e.icond] = out - 1$. So, since $CS''' > C$, CS''' is successful and claim 1 holds.

If δ is not the last instance of e, then e either handles a loop or participates in some loop, and some process evaluates as true the condition of e. If e handles a loop, then the code (lines 100 and 102) implies that $decision = \texttt{true}$. Moreover, if e does not handle a loop, e participates in some loop, and δ is not the last instance of e for out, then, by assumption, $e.ins.startinneriter \neq in - 1$ before C and can not change after C. So, any process executing e for $\langle in, out\rangle$ evaluates to false the statement of line 98 and the code (lines 100 and 102) implies that $decision = \texttt{true}$. So, in the above two cases, the code (lines 103 and 106) implies that at least one process executes the CAS of line 106; let CS''''

be the first instance of this `CAS` executed by these processes. Since δ is not completed and, by assumption, the control dependencies of e for $\langle in, out\rangle$ are resolved before C and after C they may be resolved only for iteration $\langle in, out\rangle$, at C it holds that $e.ins.cnt = in$. So, since $CS'''' > C$, CS'''' is successful and claim 1 holds.

We consider now that e does not handle a loop, e participates in some loop, and δ is the last instance of e for out. Then, by assumption, $e.ins.startinneriter = in - 1$ before C and can not change after C. Also, by assumption, $e.pcond \rightarrow ins.cnt = out$ before C and can not change after C. So, some process evaluates as true the statement of line 98 and the code (line 99) implies that $decision = \texttt{false}$. Then, in both cases, the code (lines 108 - 109) implies that at least one process executes the `CAS` of line 109; let CS''''' be the first instance of this `CAS` executed by these processes. Since δ is not completed and, by assumption, the control dependencies of e for $\langle in, out\rangle$ are resolved before C and after C they may be resolved only for iteration $\langle in, out\rangle$, then at C it holds that $e.pcond \rightarrow ins.inCD[e.icond] = out - 1$. So, since $CS''''' > C$, CS''''' is successful and claim 1 holds. □

Lemma 19. *Let m be the number of subsets of equivalence classes of $\mathcal{PO}$. Let Δ be any of these subsets and let $|\Delta| = m'$. Then, let $S(\Delta) = \delta_1, \delta_2, \dots, \delta_{m'}$ be the sequence of instances of Δ such that for each j, $1 \leq j < m'$, δ_{j+1} is consecutive to δ_j. For each i, $1 \leq i \leq m'$, assume that δ_{i-1} is the instance a transactional instruction e' and δ_i is the instance of a transactional instruction e for iteration $\langle in, out\rangle$, in σ. Then,*

1. *If $\delta_i^\alpha \neq \bot$ and $\delta_{i-1}^\alpha \neq \bot$, then*
   (a) *If $e' = \texttt{read}$ and e' does not participates in the block of e, and e' and e does not participate in the same di-list, or $e' \in \{\texttt{write}, \texttt{cond}\}$, then δ_i can only be executed after δ_{i-1} is applied,*
   (b) *otherwise, δ_i can only be executed after δ_{i-1} is completed.*
2. *the following occur, in this order:*
   (a) *$\delta_i^\alpha \neq \bot$ and there is some configuration C_1 such that δ_i is executed at C_1,*
   (b) i. *if $e = \texttt{read}$, then there is some configuration C_3 such that the output data dependencies of e are resolved for its outth outer iteration and $C_3 > C_1$,*
      ii. *if $e = \texttt{write}$ on some data item x and δ_i is the dth, $d > 0$, write on x,*
         A. *the `CAS` of line 124 is successfully executed exactly once at some configuration C_2 updating $e.ins.oldvrec.inum$ from $out - 1$ to out and $C_2 > C_1$,*

B. *the* `CAS` *of line 127 is successfully executed exactly once at some configuration C_3 updating $x.ver$ from $d-1$ to d and $C_3 > C_2$,*

iii. *if $e =$* `cond`*, e either handles a loop or participates in some loop, and its decision is to initiate inner iteration $in+1$ of e, then for each* `cond` *instruction $c \in e.ins.outCD$: only if c has initiated at least one inner iteration during its inth outer iteration, then there exists some configuration C_3, where $C_3 > C_1$, such that if k is the value of $c.ins.cnt$ after c completed its last inner iteration during its inth outer iteration, then the* `CAS` *of line 139 is successfully executed exactly once updating $c.ins.startinneriter$ to k at C_3.*

(c) *there exists some configuration C_4 at which δ_i is completed and $C_4 > C_3$, and*

(d) *if δ_i is the last instance of e and e does not participate in some loop, then*

i. *if $e =$* `cond` *and its condition is evaluated as false, then there exists some configuration C_5 at which any instruction participating in the block of e finishes and $C_5 > C_4$,, and*

ii. *there exists some configuration C_6 at which e finishes and, in case $e =$* `cond`*, $C_6 > C_5$, otherwise, $C_6 > C_4$.*

Proof. The proof is by induction of the values of i. To obtain a contradiction, suppose that Lemma does not hold for the first time when δ is the bth instance in Δ, $b \geq 1$, and δ' is the $(b-1)$st instance in Δ.

Claim 1. Suppose that claim 1 does not hold. Let p be the working process executing e for iteration $\langle in,out \rangle$ and let C be the configuration preceding the initiation of the corresponding instance of EXECUTEINS (line 30) by p. When $b = 1$, δ is the first instance in Δ and claim trivially holds; so, $b > 1$.

Case 1. We consider first that e has an input dependency from e'.

Case 1.1. If e has an input control dependency from e', then Lemma 6 implies that e is selected for $\langle in,out \rangle$ after its input control dependency from e' is resolved for out. If $e' =$ `cond`, then by definition e' is also applied for $\langle in,out \rangle$ before e is selected for $\langle in,out \rangle$. If $e' =$ `read`, claims 2(b)i and 2c imply that e' is applied before it resolves its (inner) output control dependency with e, for out. In both cases, since p selects e for iteration $\langle in,out \rangle$ before it executes e for this iteration, that is also before C, this is a contradiction.

Case 1.2. So, e has an input data dependency from e'. Then, $e' =$ `read` and

$e \in$ write, cond. The code (lines 28 to 30) implies that an instance I_{dd} of CHECKDD with parameters $\langle e, \langle in, out \rangle \rangle$ is executed by p before C, which returns `true`. In case $e =$ write, the code of I_{dd} (first condition of line 90, and lines 91 - 94), implies that the input data dependencies of e are resolved for out before C, i.e. e' is applied for iteration $\langle in, out \rangle$ before C; this is a contradiction. So, $e =$ cond. Assume first that e has not initiated any inner iteration for out; so, e' participates in the same block with e. Then, since e can initiate an inner iteration for out after C, claim 2(b)iii implies that during the execution of I_{dd} it holds that $c.ins.startinneriter = c.ins.cnt$; so, p evaluates as true the second condition of line 90. Then, the code (lines 91 - 94) implies that the (outer) input data dependencies of e are resolved for out before C'; this is a contradiction. So, in is an inner iteration initiated by e during out, e' participates in the block of e, and e has an input inner control dependency from e'; this is a contradiction.

Case 2. We consider now that e has no input dependency from e'. So, e' and e participate in the same di-list l_x of some data item x. If e precedes e' in l_x, then $e =$ cond, and e' participates in some loop and in the block of e. So, e has an inner input control dependency from e'; this is a contradiction. So, e follows e' in l_x.

Case 2.1. Assume first that e and e' participate in different blocks. If e' participates in the block of some cond instruction c' that either handles or participates in some loop, then an instance $\delta'_c \in \Delta$ of c' follows δ' and precedes δ, $\delta' <_{\mathcal{PO}} \delta'_{c'} <_{\mathcal{PO}} I$; this is a contradiction. If e' participates in the block of some cond instruction than neither participates in some loop nor handles a loop, then $\delta' <_{\mathcal{PO}} \delta$ only when e' is the last instruction of the block of $e'.pcond$, and e' and e are consecutive in l_x. Then, Lemma 8 implies that p selects e for iteration $\langle in, out \rangle$ after the completion of e'; this is a contradiction, since p selects e for iteration $\langle in, out \rangle$ before it executes e for this iteration, that is also before C. So, e participates in the block of some cond instruction c. If $c = e'$, then e has an input control dependency from e'; this is a contradiction. So, $c \neq e$. Then, an instance $\delta_c \in \Delta$ of c precedes I and follows δ', $\delta' <_{\mathcal{PO}} \delta_c <_{\mathcal{PO}} \delta$; this is a contradiction.

Case 2.2. So, e' and e participate in the same block and they are consecutive in this block. So, since e' precedes e in l_x, Lemma 14 implies that p selects e for iteration $\langle in, out \rangle$ after the completion of e' for this iteration; this is a contradiction, since p selects e for iteration $\langle in, out \rangle$ before it executes e for this iteration, that is also before C.

Claim 2. Suppose that claim 2 does not hold. Suppose first that claim 2a does not hold. Let e be placed in the di-list l_x of some data item x. When $b > 1$, by assumption, δ'

completes and any instance $\delta'' <_{\mathcal{PO}} \delta'$ (and $\delta'' <_{\mathcal{PO}} \delta$) of some transactional instruction e'' completes. If $b > 1$, let $C_{\delta'}$ be the configuration following the completion of δ'; otherwise, let $C_{\delta'}$ be the initial configuration of α. Also, let S be the set containing those instructions whose instances precede I; notice that when $b = 1$ it holds that $S = \emptyset$, otherwise, $e' \in S$. By assumption, the instances of the instructions in S that precede I in $\mathcal{PO}$ are completed, before $C_{\delta'}$. Since δ neither is applied (claim 2(b)i) nor completes (claim 2c), claim 1 implies that none of the instances of the instructions in S that follow δ in $\mathcal{PO}$ is executed after $C_{\delta'}$.

We start by arguing that the input data and control dependencies of e are resolved for $\langle in.out\rangle$, before $C_{\delta'}$. If $b = 1$, then the input data (in case $e \in \{\texttt{read}, \texttt{cond}\}$) and control dependencies of e are (trivially) resolved. We consider now that $b > 1$. In case $e = \{\texttt{write}, \texttt{cond}\}$, Claim 2(b)i implies that the input data dependencies of e are resolved for out, before $C_{\delta'}$. In case e participates in the main block, then its (outer) input control dependency is trivially resolved for out. In case e participates in some block, then Claim 2c implies that its (outer) input control dependency is resolved for out, before $C_{\delta'}$. In case e handles a loop and δ is the first instance of e, then its (inner) input control dependencies are (trivially) resolved for in; otherwise, if δ is not the first instance of e, then Claim 2c implies that its (inner) input control dependencies are resolved for in, before $C_{\delta'}$. So, claim holds.

We next argue that some process selects e for $\langle in, out\rangle$, after $C_{\delta'}$. We consider first that e does not participate in some loop. If $b = 1$, then e participates in the main block and it is the head of l_x. If $b > 1$, δ' is the last instance of e'. Thus, by assumption, each instruction in S finishes, before $C_{\delta'}$, and the code (lines 25 - 31) implies that e becomes the head of l_x, after $C_{\delta'}$. So, for $b \geq 1$, Lemma 15 implies the claim, in this case.

We consider now that e participates in some loop L; so, $b > 1$. Let $S = e_1, e_2, \ldots, e_k$, $k \geq 0$, be the sequence of transactional instructions that precede e in l_x and either handle (i.e. e_1) or participate in L; notice that in case e is the first transactional instruction of L in l_x, then S is empty (and $k = 0$). If $k > 0$, let $e'' = e_1$, otherwise, let $e'' = e$. By assumption, any transactional instruction that precedes e'' in l_x is finished, before $C_{\delta'}$. So, the code (lines 25 - 31) implies that e'' becomes the head of l_x. Since e'' is in the head of l_x, the code (line 26) implies that at least one instance of GETACTIVEINS is initiated with parameter e''. Let C_{gai} be the configuration preceding the execution of some of these instances by some process; notice that $C_{gai} > C_{\delta'}$. To obtain a contradiction, suppose that no process selects e for $\langle in, out\rangle$, after $C_{\delta'}$. So, e is not finished before C_{gai} and Lemma 16 implies the claim; this is a contradiction.

In case δ is not the first instance of e for out, then e participates in some loop and since the first instance of e is completed, claim 2(b)iii, implies that $e.ins.startinneriter = in - 1$, before $C_{\delta'}$ and, by assumption, $e.ins.startinneriter$ does not change after $C_{\delta'}$. In case δ is the first instance of e for out, we consider the following cases. If e does not participate in some loop, then the code (lines 104 - 105 and 139) implies that $e.ins.startinneriter$ is never updated, so at $C_{delta'}$ and after $C_{\delta'}$, $e.ins.startinneriter$ has its initial value, that is 0. Also, since e does not participate in some loop, $in = 0$. So, in this case, $e.ins.startinneriter = in$. If e participates in some loop, then claim 2(b)iii implies that $e.ins.startinneriter = in$ before $C_{\delta'}$ and does not change after $C_{\delta'}$. Then, Lemma 17 implies that δ is executed by some process, after $C_{\delta'}$. So, claim 2a holds. This is a contradiction.

So, at least one process executes e for $\langle in, out\rangle$.

We consider fist that $e =$ `read` on x. The code (lines 113 - 114) implies that at least one process initiates an instance of RESOLVEDD with parameters e, out, and x. Then, the code (lines 129 - 131) implies that at least one process executes an instance of the `CAS` of line 131; let CS' be the first instance of this `CAS` executed by these processes. If δ is the first instance of e, recall that $e.ins.outDD$ is initialized to 0. If δ is not the first instance of e, by assumption, all instances of e that precede δ in $\mathcal{PO}$ are applied. Also, Lemma 13 implies that any instance of e that follows δ in $\mathcal{PO}$ can be executed only after δ is completed. So, the code (lines 130 - 131) implies that CS' is successful and claim 2(b)i holds.

We consider now that $e =$ `write` on x and e is the dth, $d > 0$, write on x. The code (lines 113, 115, and 118) implies that at least one process initiates an instance of UPDATEDI with parameters $x, -, e, out$. Then, the code (lines 129 - 131) implies that at least one process executes an instance of the `CAS` of line 124; let CS'' be the first instance of this `CAS` executed by these processes. If δ is the first instance of e, recall that $e.ins.oldvrec.inum$ is initialized to 0. If δ is not the first instance of e, by assumption, all instances of e that precede δ in $\mathcal{PO}$ are completed. Also, Lemma 13 implies that any instance of e that follows δ in $\mathcal{PO}$ can be executed only after δ is completed. So, CS'' is successful and claim 2(b)iiA holds.

Since, any instance of e that follows δ in $\mathcal{PO}$ can be executed only after δ is completed, the code (lines 125-126) implies that at least one process evaluates to true the statement of line 126. Thus, at least one process executes an instance of the `CAS` of line 127; let CS''' be the first instance of this `CAS` executed by these processes. If $d = 1$, recall that $x.ver$ is initialized to 0. If $d > 1$, by assumption, all instances of transactional instruction that write x and precede δ in $\mathcal{PO}$ are completed. Also, Lemma 13 implies that any instance of e that follows δ in $\mathcal{PO}$ can be executed only after δ is completed. So, CS''' is successful and claim 2(b)iiB holds.

We consider now that $e = \texttt{cond}$, e either handles a loop or participates in some loop, and its decision is to initiate inner iteration $in+1$. The code (lines 103 - 105) implies that at least one process initiates an instance of InitializeDependentConds with parameters e and $in + 1$. Then, consider a $\texttt{cond}$ instruction $c \in e.ins.outCD$. Let k be the value of $c.ins.cnt$ after c completed its last inner iteration during its inth outer iteration. If δ is the first instance of e, recall that $c.ins.cnt$ is initialized to 0. If δ is not the first instance of e, by assumption, all instances of e that precede δ in $\mathcal{PO}$ are completed. Also, Lemma 13 implies that any instance of e that follows δ in $\mathcal{PO}$ can be executed only after δ is completed. So, since $inIter = in + 1$ (since InitializeDependentConds is initiated with second parameter $in+1$), at least one process evaluates as true the first condition of line 138. Then, since, by assumption at least one inner iterations has been initiated for e during its inth outer iteration, claim 2(b)iii of the induction hypothesis implies that at least one process evaluates as true the second condition of line 138. Thus, at least one process executes an instance of the $\texttt{CAS}$ of line 139; let CS'''' be the first instance of this $\texttt{CAS}$ executed by these processes. If δ is the first instance of e, recall that $c.ins.startinneriter$ is initialized to 0. If δ is not the first instance of e, by assumption, all instances of e that precede δ in $\mathcal{PO}$ are completed. Also, Lemma 13 implies that any instance of e that follows δ in $\mathcal{PO}$ can be executed only after δ is completed. So, CS'''' is successful and claim 2(b)iii holds.

Suppose now that claims 2c and 2d do not hold. In case δ is not last instance of e for out, $e = \texttt{cond}$, e does not handle a loop, and e participates in some loop, (i.e. δ is the first instance of e for out) claim 2(b)iii implies that $e.ins.startinneriter = in$ (that is $e.ins.startinneriter \neq in - 1$), before $C_{I'}$ and does not change after $C_{I'}$. In case δ is the last instance of e for out, $e = \texttt{cond}$, e does not handle a loop, and e participates in some loop, (i.e. δ is the second instance of e for out) since the first instance of e is completed and by claim 2(b)iii, it follows that $e.ins.startinneriter = in - 1$, before $C_{I'}$ and, by assumption, $e.ins.startinneriter$ does not change after $C_{I'}$. Moreover, by Lemma 13, claim 1, and by assumption, it follows that all instances of e that precede δ are completed, before some process executes e for $\langle in, out\rangle$, and since δ does not complete no instance of e that follows δ completes. Then, Lemma 18 implies that claims 2c and 2d hold. This is a contradiction.

□

Recall that SemanticTM processes transactions one after the other. Let $\mathcal{S} = T_1, T_2, \ldots, T_k$, $k > 0$, be the sequence of transactions processed by SemanticTM. While processing each transaction T_i, $0 < i \leq k$, SemanticTM places each transactional instructions of T_i in the appropriate di-list, together with its dependencies and respecting the sequential semantics of T_i's instructions. So, in each di-list, the transactional instructions of the transactions in $\mathcal{S}$ respect the order of $\mathcal{S}$.

We say that a value of some data item x is *correct* at some configuration C, if it is the value written by the last instance of a `write` instruction that updates x before C. We say that an instance of a `read` instruction is *correct* at C, if it reads a value that is correct at C. We say that an instance of a `write` instruction is *correct* at C, if it updates x with a value v that is computed using values that are correct at C. We say that an instance of a `cond` instruction that evaluates its condition is *correct* at C, if this evaluation is performed using values that are correct at C.

Below we prove that all instances of transactional instructions that occur in α are correct. We remark that by doing so, we also prove that SemanticTM satisfies linearizability (and opacity).

Theorem 20. *The instances of transactional instructions that occur in α are correct.*

Proof. The proof is by induction on the place of some instance of a transactional instruction in PO. Fin any $k > 0$ and assume that the claim holds for all the instances of PO up to the $(k-1)$st. We prove that the claims holds also for kth instance. Let δ be the kth instance of $\mathcal{PO}$ and let δ be the instance of some transactional instruction e. Lemma 19 (claim 2a) implies that at least one process executes δ.

We consider first that $e =$ `read` on some data item x, placed in the di-list l_x of x. Lemma 19 (claim 2(b)i) implies that exactly one process reads (line 131) a value v of x for δ; let p be this process. The code implies that p read v on line 129 from the `direc` of x, while p executes δ. So, since all the `write` instructions that operate on x are contained in l_x, Lemma 19 (claims 1 and 2(b)iiB) implies the claim.

We consider now that $e =$ `write` on x. Lemma 19 (claim 2(b)iiB) implies that exactly one process writes (line 131) a value v in x for δ; let p' be this process. The code implies that p' calculates v on line 117 using values read on line 116 through the val field of each transactional instruction in $arrayDD$, while p' executes δ. The code (lines 30, 28, 94, and 95) implies that $arrayDD \in \{e.ins.inDD, e.ins.inDDnner\}$. Thus, Lemma 19 (claims 1 and 2(b)i) implies the claim.

We consider now that $e = \texttt{cond}$. In case δ is an instance of e that evaluates the condition of e, then the code implies that at least one process performs this evaluation on line 102 using values read on line 101 through the val field of each transactional instruction in $arrayDD$, while some process executes δ. The code (lines 30, 28, 94, and 95) implies that $arrayDD \in \{e.ins.inDD, e.ins.inDDnner\}$. Thus, Lemma 19 (claims 1 and 2(b)i) implies the claim. □

6.5 Experimental Evaluation

In this section, we present some experimental results on the performance of SemanticTM.

6.5.1 The system

We use a Core i7-4770 3.4 GHz Haswell processor, running Linux 3.9.1-64-net1 x86_64. This processor has 4 cores, each with 2 hyperthreads, and hyperthreads enabled. Each core has a private 32KB 8-way associative level-1 data cache and a 256KB 8-way level-2 data cache. The chip further includes a shared 8MB level-3 cache. The cache lines are each 64-bytes.

The benchmark code was written in C and compiled with GCC-4.8.1. We compare SemanticTM to GccSTM, the gcc's STM support which was introduced in GCC-4.7 [33]. GccSTM is considered as the industry STM standard.

6.5.2 Tested Workload

We study four micro-benchmarks that execute simple static transactions, testing different conflict patterns among them. In each of our benchmarks, we execute 10^5 transactions and have $N \in \{1, 2, \ldots, 8\}$ worker processes $W_1, \ldots, W_n$ work on N data items and their associated di-lists $V_1, \ldots, V_N$. For our experiments, we consider a simplified version of SemanticTM the code of which works as follows; the type definitions and its pseudocode are presented in Figure 6.10. Before the beginning of each experiment, a single process places

```
shared $Ditem[M]$: value                                /* the value of each data item */
shared $List[M]$: array of entry records                /* di-list of each data item */

type entry
    $ins$: {
        $\langle iType$ : read,
          $outDD$ : value$\rangle$,                          /* implements the output data dependencies of $ins$ */
        $\langle iType$ : write,
          $inDD[]$ : ptr to value,                       /* implements the input data dependencies of $ins$ */
          $f$ : function$\rangle$,
        $\langle iType$ : cond,
          $inDD[]$ : ptr to value,                       /* implements the input data dependencies of $ins$ */
          $f$ : function,
          $decision$ : boolean$\rangle$                       /* implements the output control dependencies of $ins$ */
    }

ApplyInstructions() by process $p$:
    for each element $eptr \in List[p]$ do                                  /* as long as there is work */
        if ($eptr \rightarrow pcond \neq$ null) then                          /* if $eptr$ participates in some block, wait
            wait until $eptr \rightarrow pcond \rightarrow decision \neq \perp$        its input control dependency to be resolved */
            if ($eptr \rightarrow pcond \rightarrow decision =$ false) then continue
                                 /* if $eptr$ is a read, resolve its output data dependencies */
        if ($eptr \rightarrow ins.iType =$ read) then $eptr \rightarrow outDD := Ditem[p]$
        else                     /* otherwise, read the values of the input data dependencies of $eptr$ */
            $values :=$ ReturnDDValues($eptr$)
                                 /* if $eptr$ is a write on data item $x$, calculate the new value of $x$, and update it */
            if ($eptr \rightarrow ins.iType =$ write) then $Ditem[p] := eptr \rightarrow f(values)$
                                 /* otherwise, $eptr$ is a cond, calculate its decision and resolve its output control dependencies */
            else $eptr \rightarrow decision := eptr \rightarrow ins.f(values)$

$\langle values[]$ : value$\rangle$ ReturnDDValues($eptr$ : ptr to entry) by process $p$:
    for each element $d \in eptr \rightarrow ins.inDD$ with index $j$ do        /* for each input data dependency $d$ of $eptr$
        wait until $*d \neq \perp$                                               wait until $d$ is resolved
        $values[j] := *d$                                                      and maintain its value into $values$ */
    return $values$
```

Figure 6.10: Type Definitions, and the Code of ApplyInstructions and ReturnDDValues of the Simplified Version of SemanticTM

```
$V_k \leftarrow V_{(k-2)\%N} + 1$
wait for some time
$V_{(k+1)\%N} \leftarrow V_{(k-1)\%N} + 1$
```

T_1

```
add read to $V_{(k-2)\%N}$
add write to $V_k$ dep on $V_{(k-2)\%N}$ followed by wait
add read to $V_{(k-1)\%N}$
add write to $V_{(k+1)\%N}$ dep on $V_{(k-1)\%N}$
```

T_1 - SemanticTM

```
$V_k \leftarrow V_k + 1$
wait for some time
$V_k \leftarrow V_k + 1$
```

T_2

```
$V_1 \leftarrow V_1 + 1$
wait for some time
$V_1 \leftarrow V_1 + 1$
```

T_3

```
$V_1 \leftarrow 10000$
while ($V_1 \neq 0$) do
    if ($V_1 > 0$) then
        $V_1 \leftarrow V_1 + 1$
        wait for some time
        $V_k \leftarrow V_k + 1$
```

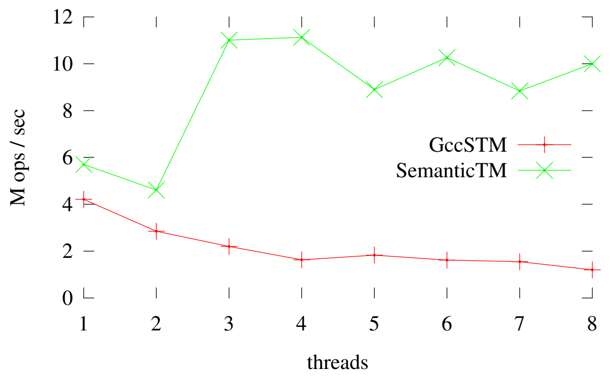
T_4

Figure 6.11: The Code of Transaction T_i, $1 \leq i \leq 4$, Executed by process k, $1 \leq k \leq N$

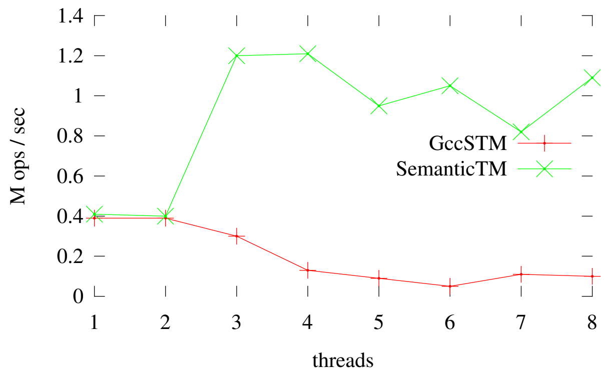
the instructions of each transaction in each di-list, with the difference that loops are now unfolded; we remark that in our experiments the number of times a loop is executed is known before the corresponding transaction is initiated. Specifically, considering a loop `cond` instruction c, instead of inserting only a single instance for c and its block's instructions into di-lists, multiple instances are inserted, one for each iteration of the loop; by doing this, any `cond` instruction is now manipulated as a conditional statement (i.e. `if`, `then`, `else`). Then, N workers are initiated and worker W_k, $1 \leq k \leq N$, processes all transactional instructions contained in di-list V_k; instead of having each worker to repeatedly choose, uniformly at random, a di-list and execute the instructions of this list. This static assignment of workers to lists trades wait-freedom for performance. We remark that no integration of gcc's STM support was required in order to implement this simplified version of SemanticTM.

The GccSTM code works on N variables as well, and initiates exactly N processes, each executing the same type and number of transactions as in SemanticTM. In both GccSTM and SemanticTM, in each benchmark, each worker process executes transactions of the same type. We denote by T_i, $1 \leq i \leq 4$, the transactions' type executed in our ith benchmark. The code of T_i executed by W_k is shown in Figure 6.11; also, the SemanticTM version of the code of T_1 is presented. Notice that all the accesses performed by each of our four benchmarks are shared accesses. So, there is no need to use selective instrumentation in our code (i.e., only instrument variables that are actually potential sources of contention).

We measure the throughput, i.e. the number of transactions that are executed successfully per second. The interesting thing about the workload of the 1st benchmark is that

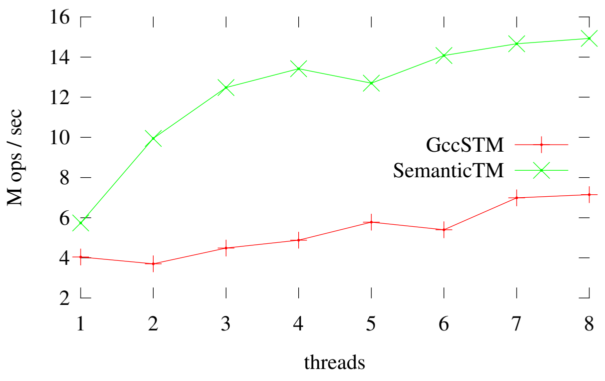


(a) T_1 (conflicts) - Short Wait Time

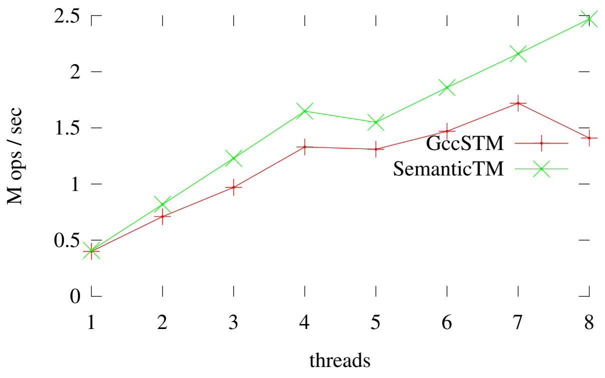


(b) T_1 (conflicts) - Long Wait Time

Figure 6.12: Transactions With Long and Short Wait Time, to Demonstrate the Impact of Different Amounts of Local Work, for T_1

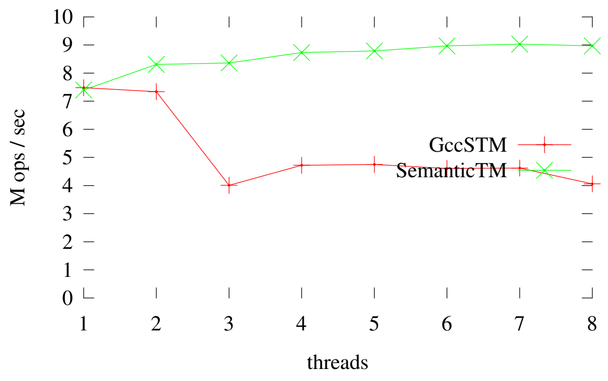


(a) T_2 (no-conflict) - Short Wait Time

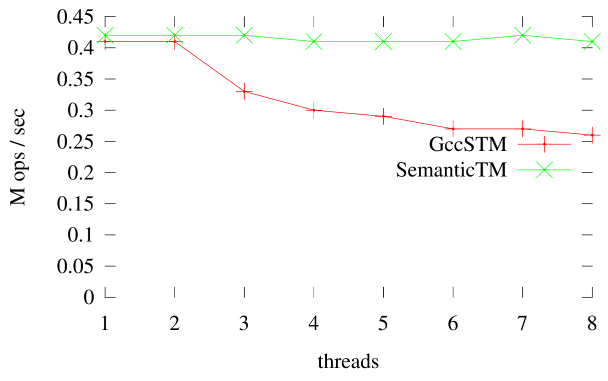


(b) T_2 (no-conflict) - Long Wait Time

Figure 6.13: Transactions With Long and Short Wait Time, to Demonstrate the Impact of Different Amounts of Local Work, for T_2

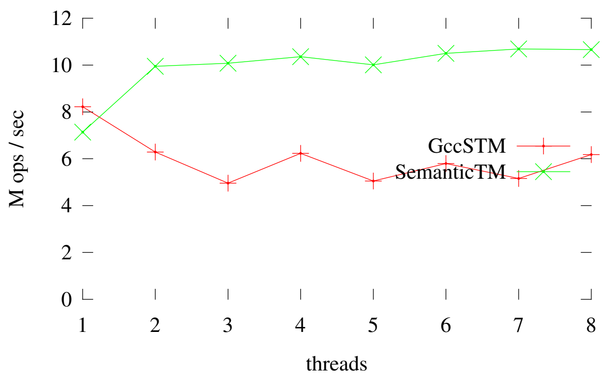


(a) T_3 (counter) - Short Wait Time

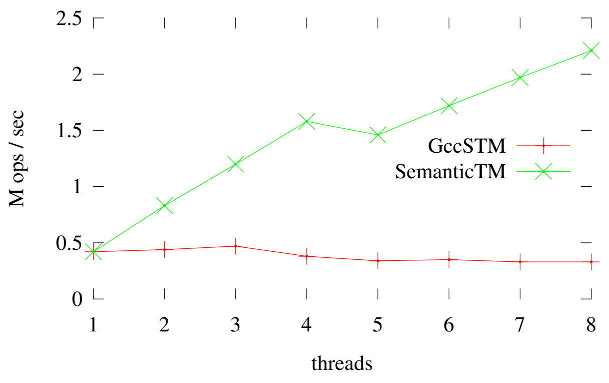


(b) T_3 (counter) - Long Wait Time

Figure 6.14: Transactions With Long and Short Wait Time, to Demonstrate the Impact of Different Amounts of Local Work, for T_3



(a) T_4 (cond) - Short Wait Time



(b) T_4 (cond) - Long Wait Time

Figure 6.15: Transactions With Long and Short Wait Time, to Demonstrate the Impact of Different Amounts of Local Work, for T_4

GccSTM, as well as any other optimistic TM algorithm, will abort all transactions while executing line 2. The reason is that while W_k waits by calling `wait`, W_{k-2} writes a data item (V_{k-2}) which is contained in W_k's read-set. However, W_k realizes that it has to abort only at commit time. Thus, the longer each transaction waits, the higher is the penalty (in terms of the number of aborted transactions) that an optimistic TM pays. We remark that the use of `wait` is realistic since it simulates the execution of local work which might be necessary.

In the 2nd benchmark, no transaction ever aborts, since each of them accesses a disjoint set of data items (W_k accesses only V_k). Using this benchmark, the overhead added by the SemanticTM's implementation is compared against the overhead added by GccSTM. In the 3rd benchmark each transaction increments by one the same shared counter (V_1). Finally, the 4th benchmark studies SemanticTM's performance for transactions with conditionals and loops.

6.5.3 Results

The graphs of Figures 6.12 to 6.15 show the performance advantage of SemanticTM in comparison to GccSTM. As expected, this advantage is significant in the 1st experiment (Figure 6.12), since the abort ratio of GccSTM is very high (for eight processes, it is 8 times faster than GccSTM when wait time is short and 20 times faster when wait time is long). In the 3rd experiment (Figure 6.14), GccSTM causes a smaller number of aborts, since at the first write, V_1 is locked due to its encounter-time-locking algorithm [33]. We remark that in encounter time locking V_1 is locked when WRITEDI is executed for it, as opposed to acquiring the lock later during commit time. Still its performance degrades. However, since the di-list of V_1 becomes a bottleneck, SemanticTM is only 2.5 times faster than GccSTM when wait time is short, and 2 times faster when wait time is long.

Finally, the 2nd experiment (6.13) where no conflicts occur, show that the overhead added by SemanticTM is less than the overhead added by GccSTM, since SemanticTM is almost 2 times faster than GccSTM in this experiment. For a small number of processes, the gap in performance is small for long wait time, since the overhead added by the GccSTM is amortized.

Chapter 7

Conclusion and Future Research

7.1 Synopsis of Contribution

In this thesis, we studied two well-established mechanisms for automatically executing sequential code segments in a concurrent environment from the perspective of achieving enhanced parallelism without sacrificing correctness and progress. We studied three major techniques for enhancing parallelism, namely disjoint-access parallelism, speculation, and fine-grained parallelism at instruction level.

In the avenue of studying disjoint-access parallel algorithms, we proved that it is not a coincidence that no algorithm in the literature ensures both disjoint-access parallelism and wait-freedom. Specifically, we proved that there is no linearizable universal construction that ensures both disjoint-access parallelism and wait-freedom. To prove our impossibility result we considered a data structure that can grow arbitrarily large in some execution; specifically, a singly liked-list.

For data structures that have a bound on the number of data items accessed by each operation they support, we also presented a universal construction (DAP-UC) that ensures both wait-freedom and disjoint-access parallelism. We further introduced and studied a weaker version of disjoint-access parallelism, which still however allow for increased parallelism, and we presented We presented a universal construction (TI-DAP-UC) that ensures wait-freedom and this version of disjoint-access parallelism.

In the STM context, as a first step towards achieving enhanced parallelism, we introduced WFR-TM, an STM algorithm which attempts to combine some of the advantages of pessimistic and optimistic STM. Finally, we introduced SemanticTM, an STM algorithm in which parallelism is achieved at the level of transactional instructions; i.e. not only the transactions themselves but also the instructions of each transaction may be executed concurrently. For simple transactions and assuming compiler support, SemanticTM guarantees that all transactions are wait-free, by ensuring that transactions never conflict.

When considering specific applications, some of the above algorithms may be advantageous over the others, in terms of achieved performance. For highly-contented applications, i.e. those that the concurrent execution of its operations result on highly-contented executions, DAP-UC can ensure both disjoint-access parallelism, strong progress guarantees, and high fault tolerance. For applications whose operations are read-intensive, WFR-TM offers strong progress guarantees and high fault tolerance for the read-only operations. Finally, for applications that contain simple operations such that the dependencies between the instructions of each operation are known at compile time, SemanticTM offers strong progress guarantees and high fault tolerance for all operations.

7.2 Directions for Future Work and Research

Although our work on disjoint-access parallel and wait-free universal constructions and STMs answers an open research problem, several problems arise that deserve further research. An interesting future research direction would be to introduce and study other meaningful weaker versions of disjoint-access parallelism that still allow for increased parallelism. A family of such properties could be derived by considering versions of disjoint-access parallelism which allows processes to interfere on a set of base objects. Considering different types of base objects and different sizes of such sets one could get an hierarchy of algorithms which satisfy *S-ignoring disjoint-access parallelism*, for which it is possible to provide both wait-free and disjoint-access parallel concurrent implementations for either any data structure or data structures with specific properties.

Another interesting future research direction is to identify the properties of data structures for which it is either possible or impossible to provide both wait-free and disjoint-access parallel concurrent implementations for any data structure. For instance, TI-DAP-UC ensures both timestamp-ignoring disjoint-access parallelism and wait-freedom for data structures whose operations have a bounded number of entry points. Extending the algorithm to work for data structures with unbounded number of entry points, or proving an impossibility result, is an open problem. Moreover, both DAP-UC and TI-DAP-UC requires $\Theta(n)$ space overhead per data item. It is an open problem whether a more space efficient universal construction can be designed. Similar questions arise regarding the step complexity of both algorithms.

WFR-TM attempts to reconcile positive aspects of the pessimistic and optimistic approaches in STM computing. Specifically, WFR-TM is a STM implementation that ensures wait-free execution of read-only transactions. It additionally, provides a deadlock-free, optimistic implementation of update transactions. Because of the fine-grained locking and the waiting mechanism, update transactions in WFR-TM are blocking. Recall that an update transaction may end up waiting for the termination of read-only transactions that are stalling or stopped. Future work could deal with eliminating these progress problems. Helping mechanisms can be introduced to the algorithm to make the update transaction non-blocking.

It is also interesting to study whether more efficient STM algorithms that WFR-TM and be designed by trading opacity. Moreover, it is interesting to investigate whether there are trade-offs between liveness and safety, i.e. can stronger progress properties be achieved by trading safety?

In the STM computing, several open problems arise in the direction of achieving more fine-grained parallelism, as well. The current version of SemanticTM assumes that each transaction accesses a known set of data items. This can be overcome by using wildcards; a *wildcard* is an instruction which accesses a data item but this data item is known only at runtime. As an example, consider a transaction that accesses an array; however, the exact elements of the array that it accesses become known only at runtime. To cope with this difficulty (or other similar cases), SemanticTM can maintain a di-list L for the entire array, as well as one list L_i, $1 \leq i \leq m$, for each of its elements, where m is the array size. The scheduler places each instruction e that accesses a (possibly unknown) element of the array in L. When later (at runtime), becomes known that the element is that in position i of the array, e is moved in list L_i. A similar strategy may work for supporting dynamic memory allocation, if we consider the memory heap as an array.

SemanticTM is currently achieving fine-grain parallelism at the level of transactional instructions by maintaining a di-list for each data item. Its space overhead can be decreased by maintaining a single di-list for a set of more than one data items. So, there is a tradeoff between the space overhead and the granularity of parallelism achieved by it.

Recall that in SemanticTM there are output dependencies from all instructions of a block to its `cond` and vice versa. However, the scheduler may choose to add such dependencies from the block's `cond` instruction only to those instructions that do not depend on other block instructions, since the rest have dependencies originating from them and therefore they will be executed after them. Moreover, no output control dependencies to a block's `cond` from those block instructions that do not contribute to the evaluation of the `cond` are needed. Such optimizations may have positive impact on the performance of SemanticTM.

Author’s Publications

BOOK CHAPTERS

Hillel Avni, Shlomi Dolev and Eleftherios Kosmas, “Proactive Contention Avoidance”, *Eds. Rachid Guerraoui, EPFL, and Paolo Romano, INESC-ID*, Springer-Verlag, 2014, under revision. I have contributed to the following sections: 1) No Aborts and No Serialization, 2) Prior TM Algorithms for Abort Elimination, 3) SemanticTM in a Nutshell, and 4) SemanticTM.

Panagiota Fatourou, Mykhailo Iaremko, Eleni Kanellou, and Eleftherios Kosmas, “Software Transactional Memory Algorithms”, *Eds. Rachid Guerraoui, EPFL, and Paolo Romano, INESC-ID*, Springer-Verlag, 2014, under revision. I have contributed to the following sections: 1) Introduction, 2) The system, 3) Transactional Memory Model, 4) STM Design Decisions and Mechanisms, 5) Interface for Transactional Operations, and 6) Non-Blocking Algorithms.

JOURNALS

Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers, “Universal Constructions that Ensure Disjoint-Access Parallelism and Wait-Freedom”, *submitted to Distributed Computing*, 2014.

CONFERENCES

Panagiota Fatourou, Eleni Kanellou, Eleftherios Kosmas, and M. Forhad Rabbi, “WFR-TM: Wait-Free Readers Without Sacrificing Speculation of Writers”, Presented in the *18th International Conference on Principles of Distributed Systems (OPODIS ’14)*, Cortina d’Ampezzo, Italy, December 2014. I have contributed to all parts of this paper, with my main contribution to be on designing the WFR-TM algorithm, describing its pseudocode, and writing the proof of its correctness and progress properties.

Hillel Avni, Shlomi Dolev, Panagiota Fatourou, and Eleftherios Kosmas, “Abort Free SemanticTM by Dependency Aware Scheduling of Transactional Instructions”, in *Proceedings of the 2014 International Conference on Networked Systems (NETYS ’14)*, Marrakech, Morocco, May 2014. I have contributed to the biggest part of this paper. My contribution on the Experimental Evaluation Section was on the selection of the appropriate benchmarks and on finalizing the implementation of the simplified version of SemanticTM.

Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers, "Universal Constructions that Ensure Disjoint-Access Parallelism and Wait-Freedom", *31st Annual ACM Symposium on Principles of Distributed Computing (PODC '12)*, Madeira, Portugal, July 2012. I mostly contributed to the production of the pseudocode of DAP-UC, its description, and the proof of its correctness, its progress, and its disjoint-access parallelism property. I also contributed contributed to the main idea of the impossibility result.

WORKSHOPS

Shlomi Dolev, Panagiota Fatourou and Eleftherios Kosmas, "Abort Free SemanticTM by Dependency Aware Scheduling of Transactional Instructions", *5th Workshop on the Theory of Transactional Memory (WTTM '13)*, Jerusalem, Israel, October 2013. I have contributed to all parts of this paper.

Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani and Corentin Travers, "Timestamp-Ignoring Wait-Free Universal Constructions for Unbounded Data Structures", *5th Workshop on the Theory of Transactional Memory (WTTM '13)*, Jerusalem, Israel, October 2013. I have contributed on defining timestamp-ignoring disjoint-access parallelism, designing the TI-DAP-UC algorithm, and writing its description.

Shlomi Dolev, Panagiota Fatourou and Eleftherios Kosmas, "Abort Free SemanticTM by Dependency Aware Scheduling of Transactional Instructions", *Euro-TM Workshop on Transactional Memory (WTM '13)*, Prague, Czech Republic, April 14, 2013. I have contributed to all parts of this paper.

Shlomi Dolev, Panagiota Fatourou, and Eleftherios Kosmas, "Abort Free SemanticTM by Dependency Aware Scheduling of Transactional Instructions", *8th Workshop on Transactional Computing (TRANSACT '13)*, Houston, TX, USA, March 2013. I have contributed to all parts of this paper, including the description of how SemanticTM works, designing the SemanticTM algorithm, describing its pseudocode, and describing its possible extensions. I also worked on the Introduction and Related Work Sections.

Bibliography

[1] M. Herlihy, "A methodology for implementing highly concurrent data structures," in *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP).* ACM, 1990, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/99163.99185

[2] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 124–149, 1991. [Online]. Available: http://doi.acm.org/10.1145/114005.102808

[3] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '95. New York, NY, USA: ACM, 1995, pp. 204–213. [Online]. Available: http://doi.acm.org/10.1145/224964.224987

[4] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," vol. 21, no. 2. New York, NY, USA: ACM, May 1993, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/173682.165164

[5] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[6] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08, New York, USA, 2008, pp. 175–184. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345233

[7] V. Bushkov, R. Guerraoui, and M. Kapalka, "On the liveness of transactional memory," in *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, ser. PODC '12. New York, NY, USA: ACM, 2012, pp. 9–18. [Online]. Available: http://doi.acm.org/10.1145/2332432.2332435

[8] J.-T. Wamhoff, T. Riegel, C. Fetzer, and P. Felber, "Robustm: A robust software transactional memory," in *Proceedings of the 12th International Symposium Stabilization, Safety, and Security of Distributed Systems (SSS)*, ser. Lecture Notes in Computer Science, vol. 6366. Springer, 2010, pp. 388–404.

[9] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou, "Disentangling multi-object operations (extended abstract)," in *Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing (PODC).* ACM, 1997, pp. 111–120. [Online]. Available: http://doi.acm.org/10.1145/259380.259431

[10] G. Barnes, "A method for implementing lock-free shared-data structures," in *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA).* ACM, 1993, pp. 261–270.

[11] H. Attiya, E. Hillel, and A. Milani, “Inherent limitations on disjoint-access parallel implementations of transactional memory,” *Theory Comput. Syst.*, vol. 49, no. 4, pp. 698–719, 2011.

[12] H. Attiya and E. Hillel, “A single-version stm that is multi-versioned permissive,” *Theory Comput. Syst.*, vol. 51, no. 4, pp. 425–446, 2012.

[13] D. Dice and N. Shavit, “What Really Makes Transactions Faster?” in *1st Workshop on Transactional Computing*, 2006, electronic, no proceedings.

[14] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, “Software transactional memory for dynamic-sized data structures,” in *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, ser. PODC ’03. New York, NY, USA: ACM, 2003, pp. 92–101. [Online]. Available: http://doi.acm.org/10.1145/872035.872048

[15] K. Fraser, “Practical lock freedom,” University of Cambridge, Computer Laboratory, Tech. Rep., 2003, http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[16] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, “Nonblocking memory management support for dynamic-sized data structures,” *ACM Trans. Comput. Syst.*, vol. 23, pp. 146–196, 2005. [Online]. Available: http://doi.acm.org/10.1145/1062247.1062249

[17] V. J. Marathe, W. N. S. III, and M. L. Scott, “Adaptive software transactional memory,” in *Proceedings of the 19th International Conference on Distributed Computing (DISC)*, ser. Lecture Notes in Computer Science, vol. 3724. Springer, 2005, pp. 354–368.

[18] F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang, “Nztm: nonblocking zero-indirection transactional memory,” in *Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2009, pp. 204–213.

[19] V. Gramoli, D. Harmanci, and P. Felber, “Toward a theory of input acceptance for transactional memories,” in *Proceedings of the 12th International Conference on Principles of Distributed Systems*, ser. OPODIS ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 527–533.

[20] R. Guerraoui, T. A. Henzinger, and V. Singh, “Permissiveness in transactional memories,” in *Proceedings of the 22Nd International Symposium on Distributed Computing*, ser. DISC ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 305–319.

[21] R. Guerraoui and M. Kapalka, “The semantics of progress in lock-based transactional memory,” *SIGPLAN Not.*, vol. 44, no. 1, pp. 404–415, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594834.1480931

[22] I. Keidar and D. Perelman, “On avoiding spare aborts in transactional memory,” in *In SPAA 2009*, pp. 59–68.

[23] D. Perelman, 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*, ser. PODC '10. New York, NY, USA: ACM, 2010, pp. 16–25. [Online]. Available: http://doi.acm.org/10.1145/1835698.1835704

[24] Y. Afek, A. Matveev, and N. Shavit, "Pessimistic software lock-elision," in *Proceedings of the 26th international conference on Distributed Computing*, ser. DISC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 297–311. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33651-5_21

[25] A. Matveev and N. Shavit, "Towards a fully pessimistic stm model," 2012.

[26] R. Guerraoui and M. Kapałka, "On obstruction-free transactions," in *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2008, pp. 304–313. [Online]. Available: http://doi.acm.org/10.1145/1378533.1378587

[27] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 1994, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/197917.198079

[28] F. Ellen, P. Fatourou, and E. Ruppert, "The space complexity of unbounded timestamps," in *Proceedings of the 21st International Conference on Distributed Computing*, ser. DISC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 223–237. [Online]. Available: http://dl.acm.org/citation.cfm?id=2393794.2393815

[29] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Proceedings of the 20th international conference on Distributed Computing*, ser. DISC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 194–208. [Online]. Available: http://dx.doi.org/10.1007/11864219_14

[30] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Proceedings of the 20th International Conference on Distributed Computing*, ser. DISC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 284–298. [Online]. Available: http://dx.doi.org/10.1007/11864219_20

[31] M. F. Spear, V. J. Marathe, W. N. S. III, and M. L. Scott, "Conflict detection and validation strategies for software transactional memory," in *Proceedings of the 20th International Symposium Distributed Computing (DISC)*, ser. Lecture Notes in Computer Science, vol. 4167. Springer, 2006, pp. 179–193.

[32] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 1793–1807, 2010.

[33] T. Riegel, “Software transactional memory building blocks,” Ph.D. dissertation, Technische Universität Dresden, Dresden, 01062 Dresden, Germany, 2013. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-115596

[34] P. Jayanti and S. Petrovic, “Efficiently implementing ll/sc objects shared by an unknown number of processes,” in *Proceedings of the 7th International Workshop on Distributed Computing (IWDC)*, ser. Lecture Notes in Computer Science, vol. 3741. Springer, 2005, pp. 45–56.

[35] C. H. Papadimitriou, “The serializability of concurrent database updates,” *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979. [Online]. Available: http://doi.acm.org/10.1145/322154.322158

[36] M. Herlihy, V. Luchangco, and M. Moir, “Obstruction-free synchronization: Double-ended queues as an example,” in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ser. ICDCS ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 522–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850929.851942

[37] H. Attiya and E. Dagan, “Universal operations: unary versus binary,” in *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 1996, pp. 223–232. [Online]. Available: http://doi.acm.org/10.1145/248052.248097

[38] H. Attiya and E. Hillel, “Built-in coloring for highly-concurrent doubly-linked lists,” *Theory Comput. Syst.*, vol. 52, no. 4, pp. 729–762, 2013.

[39] 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*, ser. ICDCN’11. Springer-Verlag, 2011, pp. 83–94. [Online]. Available: http://dl.acm.org/citation.cfm?id=1946143.1946151

[40] H. Attiya and E. Hillel, “Highly concurrent multi-word synchronization,” *Theor. Comput. Sci.*, vol. 412, no. 12-14, pp. 1243–1262, 2011.

[41] Y. Afek, D. Dauber, and D. Touitou, “Wait-free made fast,” in *Proceedings of the 27th annual ACM Symposium on Theory of Computing (STOC)*. ACM, 1995, pp. 538–547.

[42] P. Fatourou and N. D. Kallimanis, “The redblue adaptive universal constructions,” in *Proceedings of the 23rd International Symposium Distributed Computing (DISC)*, ser. Lecture Notes in Computer Science, vol. 5805. Springer, 2009, pp. 127–141.

[43] P. Fatourou and N. D. Kallimanis, “A highly-efficient wait-free universal construction,” in *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2011, pp. 325–334.

[44] J. H. Anderson and M. Moir, “Universal constructions for multi-object operations,” in *Proceedings of the 14th annual ACM symposium on Principles of distributed computing (PODC)*. ACM, 1995, pp. 184–193. [Online]. Available: http://doi.acm.org/10.1145/224964.224985

[45] J. H. Anderson and M. Moir, “Universal constructions for large objects,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 12, pp. 1317–1332, 1999.

[46] 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)*. ACM, 2010, pp. 335–344. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810538

[47] T. Crain, D. Imbs, and M. Raynal, “Towards a universal construction for transaction-based multiprocess programs,” *Theor. Comput. Sci.*, vol. 496, pp. 154–169, 2013.

[48] A. Welc, B. Saha, and A.-R. Adl-Tabatabai, “Irrevocable transactions and their applications,” in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, ser. SPAA ’08. New York, NY, USA: ACM, 2008, pp. 285–296. [Online]. Available: http://doi.acm.org/10.1145/1378533.1378584

[49] P. Felber, C. Fetzer, and T. Riegel, “Dynamic performance tuning of word-based software transactional memory,” in *PPoPP ’08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2008, pp. 237–246.

[50] S. M. Fernandes and J. a. Cachopo, “Lock-free and scalable multi-version software transactional memory,” in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP ’11. New York, NY, USA: ACM, 2011, pp. 179–188. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941579

[51] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar, “Smv: Selective multi-versioning stm.” in *DISC*, ser. Lecture Notes in Computer Science, D. Peleg, Ed., vol. 6950. Springer-Verlag, 2011, pp. 125–140.

[52] R. Guerraoui, M. Herlihy, and B. Pochon, “Toward a theory of transactional contention managers,” in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, ser. PODC ’05. New York, NY, USA: ACM, 2005, pp. 258–264. [Online]. Available: http://doi.acm.org/10.1145/1073814.1073863

[53] W. N. Scherer, III and M. L. Scott, “Advanced contention management for dynamic software transactional memory,” in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, ser. PODC ’05. New York, NY, USA: ACM, 2005, pp. 240–248. [Online]. Available: http://doi.acm.org/10.1145/1073814.1073861

[54] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 4–18. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92990-1_3

[55] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir, "Transactional contention management as a non-clairvoyant scheduling problem," in *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, ser. PODC '06. New York, NY, USA: ACM, 2006, pp. 308–315. [Online]. Available: http://doi.acm.org/10.1145/1146381.1146428

[56] H. Attiya and A. Milani, "Transactional scheduling for read-dominated workloads," in *Proceedings of the 13th International Conference on Principles of Distributed Systems*, ser. OPODIS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 3–17. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10877-8_3

[57] H. Attiya and D. Sainz, "Relstm: A proactive transactional memory scheduler," in *Proceedings of the 8th ACM SIGPLAN Workshop on Transactional Computing*, ser. TRANSACT '13, 2013.

[58] S. Dolev, D. Hendler, and A. Suissa, "Car-stm: scheduling-based collision avoidance and resolution for software transactional memory," in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, ser. PODC '08. New York, NY, USA: ACM, 2008, pp. 125–134. [Online]. Available: http://doi.acm.org/10.1145/1400751.1400769

[59] R. Motwani, S. Phillips, and E. Torng, "Non-clairvoyant scheduling," *Theor. Comput. Sci.*, vol. 130, no. 1, pp. 17–47.

[60] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, ser. SPAA '08. New York, NY, USA: ACM, 2008, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1378533.1378564

[61] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: a benchmark for software transactional memory," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 315–324. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273029

[62] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon, "Robust contention management in software transactional memory," in *OOPSLA '05 Workshop on Synchronization and Concurrency in Object-Oriented Lanugages (SCOOL '05)*, 2005.

[63] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: avoiding conflicts in transactional memories," in *Proceedings of the 28th ACM symposium on Principles*

of distributed computing, ser. PODC ’09. New York, NY, USA: ACM, 2009, pp. 7–16. [Online]. Available: http://doi.acm.org/10.1145/1582716.1582725

[64] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel, “Committing conflicting transactions in an stm,” in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP ’09. New York, NY, USA: ACM, 2009, pp. 163–172. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504201

[65] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, “Data-oriented transaction execution,” *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 928–939, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920959

[66] J. Steffan and T. Mowry, “The potential for using thread-level data speculation to facilitate automatic parallelization,” in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 2–. [Online]. Available: http://dl.acm.org/citation.cfm?id=822079.822712

[67] J. G. Steffan, C. B. Colohan, and T. C. Mowry, “Architectural support for thread-level data speculation,” School of Computer Science, Carnegie Mellon University, Tech. Rep. Tech. rep. CMU-CS-97-188, 1997.

[68] M. Cintra and D. R. Llanos, “Design space exploration of a software speculative parallelization scheme,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 6, pp. 562–576, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2005.69

[69] M. Cintra, J. F. Martínez, and J. Torrellas, “Architectural support for scalable speculative parallelization in shared-memory multiprocessors,” *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 13–24, May 2000. [Online]. Available: http://doi.acm.org/10.1145/342001.363382

[70] P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Luján, “Optimizing software runtime systems for speculative parallelization,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 39:1–39:27, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400698

[71] J. a. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, “Unifying thread-level speculation and transactional memory,” in *Proceedings of the 13th International Middleware Conference*, ser. Middleware ’12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 187–207. [Online]. Available: http://dl.acm.org/citation.cfm?id=2442626.2442639