

On the Limits of Efficient Database Consistency

Tim Baccaert

*Dissertation submitted in fulfillment of the
requirement for the degree of Doctor of Sciences.*

2026

Supervisor

Prof. dr. Bas Ketsman, Vrije Universiteit Brussel, Belgium

Jury

Prof. dr. Geraint Wiggins, Vrije Universiteit Brussel, Belgium (chair)

Prof. dr. Elisa Gonzalez Boix, Vrije Universiteit Brussel, Belgium

Prof. dr. Joost Vennekens, Vrije Universiteit Brussel, Belgium

Prof. dr. Frank Neven, Universiteit Hasselt, Belgium

dr. Nofar Carmeli, Inria Montpellier, France

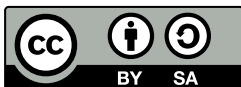
Prof. dr. Cristoph Koch, EPFL, Switzerland

Vrije Universiteit Brussel
Faculty of Sciences and Bioengineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel
Tel: +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

ISBN: 978-94-9346-177-2
NUR: 993
THEMA: UNA

“On the Limits of Efficient Database Consistency” © 2026 by Tim Baccaert
is licensed under Creative Commons Attribution-ShareAlike 4.0 International. To view a copy of this license, visit:
<https://creativecommons.org/licenses/by-sa/4.0/>



*In memory of my father,
who gave me a love for puzzles, computers,
and abysmal dad jokes.*

Abstract

Modern database systems are deployed in wildly different computing environments, from geo-distributed data centers to embedded systems. Despite this variety, all database systems are expected to ensure consistency of the data they store. We study two particular areas in which database systems have had difficulty scaling, despite hardware advances.

First, we study the theoretical limits of transaction processing systems, which are notoriously difficult to scale when there is high contention for shared resources. We show that optimally scheduling transactions with respect to their scheduling makespan, or the difference between the schedule's start and stop time, is an NP-complete problem. We then provide a polynomial time algorithm for transaction sets that contend for a single shared resource, and contribute several upper bounds on the makespan. Additionally, we derive approximation algorithms with an approximation ratio proportional to the transaction set size divided by the longest transaction's length.

Secondly, we study when it is strictly required for asynchronous distributed databases to utilize coordination in order to ensure consistency. We provide a generalization of the Consistency as Logical Monotonicity (CALM) theorem, which states that a problem has a consistent, coordination-free distributed implementation if and only if the problem is monotone. We extend the result so that it applies for problems that are non-deterministic such as leader election. We also make the theorem applicable to a wider range of systems, making the only-if direction less delicate. Lastly, we apply this generalized theorem to show that the classes of polynomial time problems and coordination-free problems are not equal.

Samenvatting

Moderne databanksystemen worden in gevarieerde omgevingen ingezet, gaande van geo-gedistribueerde datacenters tot geïntegreerde systemen. Ondanks deze variëteit, worden alle databanksystemen verwacht om hun opgeslagen data consistent te houden. We bestuderen twee specifieke domeinen waarin databanksystemen moeilijkheden hebben met opschalen, ondanks verbeteringen in hardware.

Als eerste, bestuderen we de theoretische limieten van transactieverwerkingssystemen. Deze zijn notoir moeilijk om op te schalen wanneer er hoge vraag is naar dezelfde gedeelde objecten. We tonen aan dat het optimaal plannen van transacties in verhouding tot hun makespan, of het verschil tussen de start- en eindtijden van hun schema, een NP-compleet probleem is. Daarnaast beschrijven we een algoritme dat in polynomiale tijd transactieverzamelingen kan plannen indien deze één gedeeld object hebben, en leiden we meerdere bovengrenzen op de makespan af. Bovendien construeren we meerdere approximatie algoritmen met een approximatieverhouding die proportioneel is aan de grootte van de transactieverzameling gedeeld door de lengte van de langste transactie.

Ten tweede, bestuderen we wanneer het strikt noodzakelijk is dat een asynchroon gedistribueerd systeem coördinatie nodig heeft om consistent te zijn. We leiden een generalisatie van de Consistentie als Logische Monotonociteit (CALM) stelling af, die stelt dat een probleem een consistente en coördinatie-vrije implementatie heeft als en slechts als het probleem monotoon is. We breiden het resultaat uit zodat het ook geldt voor non-deterministische problemen zoals Leadersverkiezing. Bovendien maken we de stelling toepasbaar op een breder scala aan systemen, zodat de slechts-als richting minder delicaat is. Als laatste, passen we de uitgebreide theorie toe om aan te tonen dat de klassen van polynomiale tijd en coördinatie-vrije problemen verschillend zijn.

Acknowledgements

I would like to thank my PhD supervisor Prof. dr. Bas Ketsman for being a good mentor, and for allowing me the room to experiment with different research topics. Even though this did not always result in work that got published, it did teach me valuable research skills which I consider to be just as valuable.

I want to thank the jury members: Prof. dr. Christoph Koch (EPFL), Dr. Nofar Carmeli (Inria), Prof. dr. Frank Neven (UHasselt), Prof. dr. Joost Vennekens (VUB), and Prof. dr. Elisa Gonzalez Boix (VUB) for feedback on the presentation and their constructive comments on this thesis.

I am also grateful towards my colleagues at the Software Languages Lab of the Vrije Universiteit Brussel for organizing research meetings; for the many discussions during lunches and coffee breaks; and for much more.

Een significant deel van deze thesis is geschreven tijdens een periode waar mijn vader is gestorven na een strijd met ziekte. Zonder de aanmoediging en ondersteuning van mijn moeder, familie, vriendin Caroline, en vriendenkring was deze thesis allicht nooit een feit geworden. Bij deze wil ik al deze mensen nogmaals van harte bedanken voor hun steun.

- Tim Baccaert

Contents

1	Introduction	1
1.1	Contributions	5
	A Formal Theory of Optimal Transaction Schedules .	5
	A Semantic Classification of Coordination-Freedom .	6
1.2	Publications	8
I	Transactions	11
2	Transaction Scheduling	13
2.1	Preliminaries: Permutations, Orders, Graphs	13
	Permutations	14
	Orders	14
	Directed Graphs	15
2.2	Preliminaries: Transaction Theory	15
	Transactions	15
	Schedules	18
	Conflict Serializability	22
2.3	Preliminaries: Computational Problems	23
	Decision Problems	23
	Optimization Problems	26
2.4	The Transaction Scheduling Problem	27
	Precedence Graphs	27
	Canonical Schedules	31
	Enumerative Transaction Scheduling	35
2.5	Summary	36
	Further Reading	37

3	Time Complexity	39
3.1	Preliminaries: Time Complexity Classes	39
	NP, P, NP-hardness and NP-completeness	39
	NPO, PO, and NP-hard Optimization	41
3.2	The Complexity of Transaction Scheduling	42
	From Colorability to Orientability	43
	From Orientability to Schedulability	44
3.3	Single Contention Point Scheduling	47
	Contention Points	47
	Mapping Transactions Onto Square Grids	48
3.4	Summary	53
	Further Reading	53
4	Approximation Algorithms	55
4.1	Preliminaries	55
	The Upper Bound Lemma	56
4.2	Two Contention Point Scheduling	58
	Algorithm	59
	Approximation Ratio	61
4.3	Three Contention Point Scheduling	64
	Maximal Total Overlap	65
	Algorithm	71
	Approximation Ratio	72
4.4	Fixed Contention Point Scheduling	78
	Adjacent Transposition Sequences	79
	Algorithm	81
	Makespan Bounds	82
4.5	Summary	89
	Summary of Bounds and Algorithms	89
	Further Reading	91
5	Related Work	93
5.1	Related Theory	93
	Serializability and Schedulability	93
	The Robustness and Allocation Problems	94
	The Job Shop Scheduling Problem	95
5.2	Related Systems	96
	Scheduling Systems	97
	Partitioning Systems	98
	Single-Threaded Systems	98

II	Queries	99
6	Transducer Networks	103
6.1	Preliminaries	103
	Schemas, Instances, and Mappings	103
	Networks, Distributed Schemas and Instances	104
	Distributed Behaviors	105
6.2	Generalized Transducer Networks	106
	Generalized Relational Transducers	107
	Configuration Constraints	108
	Configurations, Transitions, and Runs	109
	Distributed Output	111
6.3	Summary	112
	Further Reading	112
7	Expressivity of Transducer Networks	113
7.1	Computing Behaviors with Transducers	113
	Proof of Proposition 13	115
7.2	Expressing Leader Election	121
	Proof of Theorem 12	127
7.3	Expressing All Behaviors	133
7.4	Termination-Aware Transducers	136
7.5	Summary	142
8	Coordination-Freedom	143
8.1	Non-Deterministic Computation	144
	Proof of Theorem 15	152
8.2	Deterministic Computation	155
	Proof of Proposition 26	157
	Proof of Proposition 27	161
8.3	Polynomial Time Computation	164
	The Successor and Node Constraints	166
8.4	A Coordination Hierarchy	169
	Proof of Proposition 41	172
	Proof of Proposition 42	176
8.5	Related Work	184
8.6	Summary	185

9	Conclusion	187
9.1	Transaction Scheduling	187
9.2	Coordination-Freedom	188
A	Encodings for Transaction Scheduling	189
A.1	Transaction Scheduling as a SAT Problem	189
	Precedence Graphs	190
	Path Encoding	192
A.2	Transaction Scheduling as Mixed ILP	193
	Transaction Sets	193
	Schedules and Constraint Encoding	196

Introduction

This thesis is divided in two distinct parts. Each part discusses a different limit of efficient database consistency. The first part is concerned with the limits of transaction processing, while the second part is concerned with the limits of coordination-free distributed query processing.

Transaction Processing

Modern database systems are deployed in wildly disparate environments. This includes geo-distributed database systems which span many data centers across the globe, to embedded databases such as SQLite¹ which run on your watch, phone or tablet. Despite this variety, all database systems need to ensure a consistent state.

In the embedded scenario, advances in hardware have meant that almost all processors are now multi-core. Prior to 2006, as transistors (specifically MOSFETs²) got smaller, their power use would be proportional to their area, a phenomenon more commonly known as Dennard scaling [30]. Given that transistors have now gotten so unbelievably small that this no longer holds, the main way to increase transistor count is to add more cores to the processor. As a result, the clock frequency and power consumption per core have mostly stagnated, but the number of cores has increased.

Given this reality, the design of embedded database systems had to adapt if it wanted to continue benefiting from faster processors. While this has been mostly successful in the case of online analytical processing (OLAP) databases, which are optimized for complex

¹<https://sqlite.org/>

²Metal-Oxide-Semiconductor Field-Effect Transistor

read-only workloads, it has been much harder for online transaction processing (OLTP) databases [98]. Such OLTP systems process both read operations and complex write operations, which are bundled inside a logical unit called a transaction. These transactions are the core reason why scaling OLTP is hard.

Transactions know their origin in IBM's Information Management System (IMS), which was released in 1966. IMS was initially built to keep track of the bill of materials required to build the Apollo spacecraft, which landed astronauts on the moon [19]. The systems at the time had support for time-sharing, enabling multiple users to use the system concurrently³. The purpose of a transaction is to provide users with the illusion that their program runs independently of other programs running simultaneously on the system. The database system is responsible for making sure everything remains in a consistent state while this takes place.

There is much research about scalable approaches towards transaction processing. Today, most systems rely on *concurrency control*, which is a set of principled protocols that orchestrates access to shared resources such as pages and tuples. In the database systems literature, OLTP systems are evaluated by measuring their *throughput*, which is the number of transactions that are processed by the database per second, on various predetermined workloads such as YCSB [26] or TPC-C [28]. Reliably executing such workloads in a fair manner is not a trivial task, as database engineers typically write ad-hoc versions of these benchmarks, or they manipulate the workloads in non-standard ways. The systems community has spent a considerable amount of effort trying to make this evaluation method less biased [3, 31, 77, 96, 99].

While there is much experimental work, there is little to no theoretical understanding about the performance limits of transaction workloads. The current theory about transactions largely centers around *safety properties* [54, 72], and we elaborate on the thesis' relation to this literature in Chapter 5. In its most elementary form, safety is about deciding whether a given execution history (or schedule) was valid or not; with the gold standard definition for

³This is predominantly the IBM System/360 Model 65 and other systems in the System/360 series, the IBM 7010 on which the precursor (called DATE) to IMS was initially developed in 1965 had no support for time-sharing yet [19].

validity being *conflict serializability*. While the prior theory focuses on defining a space of schedules which is correct, *the first part of this thesis is concerned with identifying the space of schedules that are correct and optimal.*

Coordination-Free Distributed Query Processing

In (geo-)distributed database systems, results such as the Fischer-Lynch-Paterson (FLP) impossibility theorem [38], demonstrate an additional layer of complexity introduced by *asynchronous communication*. At a high level, the FLP result states that in asynchronous distributed systems, it is impossible to distinguish between system nodes that are slow or system nodes that have crashed, and it is therefore impossible to reach a consensus about the state of the system. In practice, the conditions of this theorem are stringent, and many real systems rely on coordination protocols such as Paxos [56] and Raft [70] to solve this problem reliably.

While protocols like Paxos, Raft, and their variations are effective in practice, they can often be too slow, too difficult to implement, or practically infeasible. The Consistency as Logical Monotonicity (CALM) theorem—which states that a (deterministic) problem has a consistent, coordination-free distributed implementation if and only if it is logically monotone—is one of the first results characterizing precisely when a problem requires the use of such coordination protocols. This result was first conjectured by Hellerstein [43], and later proven by Ameloot, Neven, and Van den Bussche [8] for deterministic computational problems (*i.e.*, deterministic queries).

The proof of the CALM theorem presented in [8] makes use of Relational Transducer Networks, an abstract model for asynchronous distributed systems where system nodes are programmed by specifying queries, and where all state is represented by the relational model. Other than which nodes are part of the network, the nodes have no other a priori knowledge about the global state of the system. Concretely, every node has access to two system-level relations: the **All**-relation, which lists the identifiers of all nodes in the system, and the **Id**-relation, which stores the identifier of the current node. The distribution of input data is always arbitrary, and is considered global state that nodes cannot access. Because of these

considerations, the “only-if” direction of the theorem does not apply to many of the systems that are typical in practical scenarios.

For example, the work by Zinn, Green, and Ludäscher [104] observed that some non-monotone problems are computable without coordination if nodes can somehow consult the data partitioning strategy. Based on this observation, the work by Ameloot, Ketsman, Neven and Zinn [7] extends the CALM theorem by taking such partitioning strategies into account. More concretely, it was shown that for two additional families of systems which have knowledge of the data partitioning strategy, it is possible to express problems that satisfy weaker notions of monotonicity.

The existing results reveal important aspects of the problem space, but the “only-if” direction of the CALM theorem remains delicate and is true only in systems conforming to the precise formulations of the prior works on CALM [7, 8, 104]. For systems that deviate even slightly, the analysis has to be done from scratch. *In the second part of this thesis, we introduce a more robust formal model and extend the earlier mentioned works by addressing the problems we discuss next.*

First, earlier work assumes systems where the data partitioning strategy can be consulted locally by each node. This is the case if a hash- or range-based partitioning strategy is used, but if the partitioning strategy is provided differently, such as through a distributed index [102], then prior results do not apply.

Secondly, there is more state that a system could make available to its nodes besides just the partitioning strategy. For instance, many programs implicitly rely on a global order on their underlying data values, such as through lexicographic string comparison or number comparison based on the total order for integers. Other examples are distributed systems which are organized in different network topologies that are not a complete graph, as is the case for distributed hash tables (DHTs) like Kademlia [64], or star topologies such as the leader-worker or client-server models.

Third, prior results only apply to problems that can be formulated as deterministic queries, where each problem input instance is associated to exactly one output instance. Furthermore, the problems are not allowed to refer to the location of data in the system at a semantic level, so we cannot express the scenarios that inspired

the CALM conjecture in the first place [5, 6, 43, 59]. In these scenarios, nodes are part of the problem formulation and multiple valid outputs could be associated with a single input instance.

1 Contributions

The contributions of this thesis can similarly be grouped into two main parts. The former relates to the problem of identifying the space of correct and optimal transaction schedules, and the latter is concerned with extending the CALM theorem.

A Formal Theory of Optimal Transaction Schedules

We formalize the execution of a transaction set as a schedule, which is the assignment of all operations in the set to positive integers. This intuitively represents their point of execution in time, and it lets us reason about operations that run in parallel, rather than working from the total order of operations as is customary when studying safety properties for transaction execution. Moreover, it lets us quantify the efficiency of a schedule as the elapsed time between a schedule's beginning and end. This is a quantity better known as the *makespan* in the scheduling literature [46, 73]. We focus on two central questions, the first being: “What is the complexity of constructing a serializable schedule of minimal makespan?”, and secondly: “Is there a bound on the optimal makespan for a given transaction set, and can we construct serializable schedules approximating this bound in polynomial time?”.

First, we demonstrate that the schedules with minimal makespan are among those schedules that can be constructed from a total order on the transactions. Hence, to find a minimal makespan schedule, it is sufficient to enumerate all total orders on the transaction set. This is the *enumeration scheduling algorithm*, presented at the end of Chapter 2.

Next, in Chapter 3, we show that for a set of transactions and a positive integer k , it is an NP-complete decision problem to decide whether there exists a serializable schedule with a makespan less than or equal to k . This demonstrates that we cannot fundamen-

tally improve on the naive enumeration scheduling algorithm presented in Chapter 2. We subsequently identify a fragment of the problem for which optimal scheduling is possible in polynomial time, namely, the transaction sets with a single contention point. Transactions in such a set are allowed to have a conflicting access to a single shared object, and this shared object is the same for all transactions in the set.

Lastly, we investigate bounds and approximation algorithms for the problem. In brief, an approximation algorithm relaxes the requirement that the output of the algorithm must be optimal. Instead, the output is allowed to be a bounded factor away from optimal. In order to get this information, we first need to derive bounds on the optimal makespan. We present a table of derived upper and lower bounds for the optimal makespan in Table 4.1, presented in Section 4.5.

These bounds are used to derive the quality metric of our approximation algorithms, the *approximation ratio*, which is the cost of the algorithm's output in proportion to the optimal output. An approximation algorithm is *tight* whenever there is an input on which the algorithm actually produces the optimal cost schedule. A summary of algorithms is given in Table 4.2, presented in Section 4.5.

A Semantic Classification of Coordination-Freedom

The first gap in prior work [7, 8, 104] is that it was not able to express the scenarios that inspired the initial CALM conjecture. To that extent, we introduce the notion of a (distributed) behavior as model for the class of problems that can be formulated by mappings from distributed instances to other distributed instances. Intuitively (a formal definition follows in Chapter 6), a distributed behavior is a non-deterministic query with the extra meaning that facts in its instances are tied to a specific node in the system. We call such instances *distributed instances*.

Example 1. For an example of a distributed instance, consider a set of facts of the form $\text{Leader}@x(y)$ with meaning that the node with identifier x deems the node with identifier y a leader. From a pure language perspective, the @ notation is

syntactic sugar, with $\text{Leader}@x(y)$ equivalent to $\text{Leader}(x, y)$. Operationally, however, we will interpret $\text{Leader}@x(y)$ as the fact $\text{Leader}(y)$ located on the node with identifier x .

For an example of a problem that can be formulated as a behavior, consider the leader election problem, which asks the nodes in a network to elect precisely one of the participating nodes. As (distributed) input instances to this mapping we can consider sets of facts of the form $\text{Node}@x()$, which simply reveal the identifiers of the nodes in the network. The distributed instance H , with

$$H := \{\text{Node}@a(), \text{Node}@b(), \text{Node}@c()\},$$

thus represents a three-node network with nodes named a , b and c . As output of this behavior we can consider sets of distributed instances of the previously considered type representing all valid leader-election scenarios. That is, on input H , it would give the set $\{K_1, K_2, K_3\}$, with

$$\begin{aligned} K_1 &:= \{\text{Leader}@a(a), \text{Leader}@b(a), \text{Leader}@c(a)\} \\ K_2 &:= \{\text{Leader}@a(b), \text{Leader}@b(b), \text{Leader}@c(b)\} \\ K_3 &:= \{\text{Leader}@a(c), \text{Leader}@b(c), \text{Leader}@c(c)\}. \end{aligned}$$

We refer to Example 11 for the complete formalization.

Secondly, we (strictly) generalize the relational transducer network model introduced in [8] by externalizing the definition of the system relations and making this definition an explicit parameter of the model. The latter allows to provide a CALM meta-result (Theorem 16) that covers the main results of [7, 8, 104] but is not restricted to the earlier considered definitions of system relations, nor by those specified in this thesis.

We also give tools to reason about the amount of coordination required by specific behaviors. We do this by proving results about the expressivity of each transducer class (*c.f.*, Figure 8.1 of Section 8.4). Besides determining *if a behavior requires coordination*, this allows us to say how much coordination, in relation to other problems.

The original definition of coordination-freedom used by Ameloot et al. [7, 8] is not suitable in these scenarios. It states that *a problem is coordination-free if there always exists an ideal distribution of the input data across nodes such that the desired output can be computed without communicating* and thus inherently assumes that data is never tied to a specific node. To address this limitation, we build on the alternative (yet equivalent) definition introduced in [7, 8]: *what can be computed without coordination is equivalent to what can be computed (possibly with coordination) in the restricted model in which nodes are oblivious to the set of identifiers of computing nodes*. To clarify, the nodes in the distributed network are fixed up front and do not change, but we assume a coordination free system to be a system where the nodes do not need to keep an up-to-date and accurate record of who the other nodes are (*i.e.*, they are oblivious to the network structure).

Lastly, in Chapter 8 we use our extensions to show there are polynomial time queries that are not (feasibly) coordination-free computable, an open problem posed by Hellerstein and Alvaro in [44].

2 Publications

The contents of this thesis are based on the following published works, with exception of the approximation algorithms presented in Chapter 4, the bounds were part of the paper, the algorithms are new and yet unpublished. The following two papers are concerned with the part of the thesis which is about coordination-freedom.

Baccaert, T. & Ketsman, B. *Distributed Consistency Beyond Queries* in *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023* (eds Geerts, F. et al.) (ACM, 2023), 47–58

Baccaert, T. & Ketsman, B. A Generalized CALM Theorem for Non-Deterministic Computation in Asynchronous Distributed Systems. *Information Systems* **138**, 102691 (2026)

This publication contains most of the content about optimal transaction scheduling which represents the first part of the thesis.

Baccaert, T. *et al.* *Bounding the Makespan of Transaction Schedules* in *29th International Conference on Database Theory, ICDT 2026, Tampere, Finland, March 24-27, 2026* (eds ten Cate, B. & Funk, M.) (Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2026), 10:1–10:20

The following publication was also written during the course of my PhD, and is related to the section about transaction scheduling, but it did not make it into the thesis. It concerns a preliminary version of a simulator for executing transaction schedules.

Baccaert, T. & Ketsman, B. *Cascade: Optimal Transaction Scheduling for High-Contention Workloads* in *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024* (2024), 5634–5638

Lastly, the appendix includes some unpublished encodings of the transaction scheduling problem as both a satisfiability problem, and as a mixed integer linear programming problem. This may aid people who wish to implement a transaction scheduler themselves.

Part I

Transactions

Transaction Scheduling

Most production quality database systems execute transactions in real time. Whenever a conflict occurs, this typically implies the database system has to restart or delay the transactions which caused the conflict to take place. However, if we buffer incoming transactions in a queue and process them in larger batches, we can make more holistic scheduling decisions. This chapter defines the transaction scheduling problem, which frames these batch scheduling decisions as a computational problem.

We first define preliminary mathematical concepts that are used often throughout the text. Then, we define what transactions and schedules are, and what it means for a schedule to be safe to execute. We subsequently give a brief overview of what decision and optimization problems are. And, finally, we define transaction scheduling as an optimization problem and present an algorithm that solves it.

1 Preliminaries: Permutations, Orders, Graphs

We will assume the reader to be familiar with set theoretic concepts such as sets, set builder notation, cartesian products, and relations (among which, equivalences, total functions and partial functions), and partitions. The remainder of this section repeats some standard mathematical notions that we will make heavy use of throughout the thesis. It may be useful to skip it, and come back to it in case there is particular notation you encounter that you are not familiar with.

Permutations

A total function $f : A \rightarrow B$ is called a *bijection* if for all $x, y \in A$, if $f(x) = f(y)$, then $x = y$ (i.e., f is *injective*); and for all $y \in B$ there is an $x \in A$ with $f(x) = y$ (i.e., f is *surjective*).

If f is a bijection, then a unique inverse function $f^{-1} : B \rightarrow A$ exists. That is, f^{-1} maps each $b \in B$ to a unique $f^{-1}(b) \in A$ such that we have $(f \circ f^{-1})(b) = b$, where $f \circ g$ is function composition.

For two positive integers i and j , we write $[i, j]$ to refer to the set $\{i, i + 1, \dots, j\}$. We also write $[i]$ as a shorthand for $[1, i]$.

Definition 1. Let S be a finite set, and let n be its size. We define a *permutation on S* as a bijection from $[n]$ to S .

A permutation on S is *non-empty* if S is not the empty set $\emptyset$. We will write $\langle x_1, x_2, \dots, x_n \rangle$ for a permutation on $\{x_1, x_2, \dots, x_n\}$ with $f(i) = x_i$ for all $i \in [n]$. **Throughout the text, we will use the term (*finite*) *sequence* to mean permutation, as this is more intuitive in the context of scheduling problems.**

Orders

Definition 2. We say that $\Sigma = (S, \leq_S)$ over a set S is a *partial order* if, for all $a, b, c \in S$, the binary relation $\leq_S$ satisfies:

- $a \leq_S a$ (*reflexivity*);
- if $a \leq_S b$ and $b \leq_S c$, then $a \leq_S c$ (*transitivity*); and
- if $a \leq_S b$ and $b \leq_S a$, then $a = b$ (*antisymmetry*).

We call Σ a *total order* if, in addition, we have that either $a \leq_S b$ or $b \leq_S a$ for all $a, b \in S$ (i.e., $\leq_S$ is *strongly connected*).

Notice that there is also a *strict total order* $<_S$ associated with $\leq_S$. Concretely, we have that $a <_S b$ if $a \leq_S b$ and $a \neq b$. While $<_S$ is still transitive, it is no longer reflexive, antisymmetric or strongly connected. Instead, for all $a, b \in S$ we have that: $a <_S a$ does not hold (*irreflexivity*); $a <_S b$ implies that $b <_S a$ does not hold (*asymmetry*); and if $a \neq b$ then either $a <_S b$ or $b <_S a$ (*connectivity*).

Directed Graphs

A *(finite) directed graph* is a pair $D = (V, E)$ where V is a finite set of elements called the *vertices*, and where E is a set of pairs of elements from V called the *directed edges*.

A *(directed, simple) path* in D is a non-empty finite sequence

$$p = \langle v_1, v_2, \dots, v_n \rangle$$

of distinct vertices $\{v_1, v_2, \dots, v_n\} \subseteq V$ such that

$$\{(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)\} \subseteq E.$$

A *(directed) cycle* is a path where $(v_n, v_1) \in E$, and a directed graph is said to be *cyclic* if it contains a cycle. It is called *acyclic* if there exists no cycle. We will write $\text{max-path}(D)$ to refer to the number of edges on the longest path in D .

We say that D is *strongly connected* if there exists a path between every pair of vertices in V . Additionally, D can be a partition of multiple subgraphs that are themselves strongly connected, in that case we call such subgraphs *strongly connected components* in D .

2 Preliminaries: Transaction Theory

Transactions

We first make abstraction of shared state. Let $O = \{a, b, c, \dots\}$ be a fixed and countably infinite set of *objects*. An object can represent a tuple in a relational database, a value in a concurrent hash map, or any other data structure which supports atomic operations. By *atomic*, we mean that the operation is either completely executed or not executed at all. Any outside observer, such as a different thread of execution, is unable to witness intermediate states.

Let $a \in O$ be an object. We say that an *operation on a* is either a *read on a* or a *write on a* , which we denote by $R[a]$ or $W[a]$ respectively. The former represents operations that query the object, hence, they can safely share access to the object in parallel. The latter represents updates and modifications, they require exclusive access in

order to correctly execute in parallel. In case the specific object does not matter, or is clear from the surrounding context, we will write *operation*.

We define transactions as finite sequences on sets of operations.

Definition 3. Let ℓ be a strictly positive integer, and let $o_1, o_2, \dots, o_\ell$ be operations on objects in O . Then, we define a *transaction* τ over O as a non-empty finite sequence on the set $\text{ops}(\tau) = \{o_1, o_2, \dots, o_\ell\}$.

In what follows, we will leave O implicit when it can be derived from the context, and write transaction τ instead of transaction τ over O . We write $|\tau|$ to refer to the transaction's length $|\text{ops}(\tau)| = \ell$. A transaction always induces a total order $\text{ord}(\tau) = (\text{ops}(\tau), \leq_\tau)$. Indeed, for all $o, p \in \text{ops}(\tau)$, if $\tau^{-1}(o) \leq \tau^{-1}(p)$ then $o \leq_\tau p$.

When considering a set of transactions T , we will assume that every transaction has a unique identifier i , and write τ_i to make this explicit. We will also assume that each transaction can read to each object at most once, and write to each object at most once. This assumption is often made in other work [54, 55, 89, 92], and it is significant for the complexity results we will consider later. Given a transaction $\tau_i \in T$, this lets us refer to a specific read or write over a in $\text{ops}(\tau_i)$ as $R_i[a]$ or $W_i[a]$ respectively.

Example 2. We base the running example on a selection of transaction specifications from the SmallBank benchmark [4, 90]. In brief, the benchmark describes a rudimentary banking system. Users own accounts with customer identifiers. We write the following in SQL.

```
CREATE TABLE account (
  cid INT PRIMARY KEY,
  name VARCHAR(128) NOT NULL
);
```

Each of the customers is associated with a checkings and savings account, which stores a given balance.

```
CREATE TABLE checking (
  cid INT PRIMARY KEY REFERENCES account(cid),
  bal DECIMAL(19,4) NOT NULL
);
```

```
CREATE TABLE savings (
  cid INT PRIMARY KEY REFERENCES account(cid),
  bal DECIMAL(19,4) NOT NULL
);
```

We will describe three stored procedures that represent Small-Bank transactions. The first is “Transact Savings”, which finds a customer’s identifier based on their name, and then adds to their savings balance.

```
CREATE PROCEDURE
  TransactSavings(n VARCHAR(128), b DECIMAL(19,4))
BEGIN
  SELECT cid INTO c FROM account WHERE name = n;
  UPDATE savings SET bal = bal + b WHERE cid = c;
END;
```

Next, we describe the “Amalgamate” transaction, which moves both the checking and savings balances from one customer to another customer.

```
CREATE PROCEDURE
  Amalgamate(n1 VARCHAR(128), n2 VARCHAR(128))
BEGIN
  SELECT cid INTO c1 FROM account WHERE name = n1;
  SELECT cid INTO c2 FROM account WHERE name = n2;
  SELECT bal INTO s FROM savings WHERE cid = c1;
  UPDATE savings SET bal = 0 WHERE cid = c1;
  SELECT bal INTO v FROM checking WHERE cid = c1;
  UPDATE checking SET bal = 0 WHERE cid = c1;
  UPDATE checking SET bal = bal + v + s WHERE cid = c2;
END;
```

Finally, we have the “Write Check” transaction, which occurs when a customer wants to write a check from their account.

```
CREATE PROCEDURE
  WriteCheck(n VARCHAR(128), b DECIMAL(19,4))
BEGIN
  SELECT cid INTO c FROM account WHERE name = n;
  SELECT bal into s FROM savings WHERE cid = c;
  SELECT bal into v FROM checking WHERE cid = c;
  IF (s + v) < b THEN
    UPDATE checking SET bal = bal - (b + 1) WHERE cid = c;
  ELSE
    UPDATE checking SET bal = bal - b WHERE cid = c;
  END IF;
END;
```

In our formal model, we need to know the objects (*i.e.*, tuples) being read and written, and this is not known before

execution. Hence, our model makes the assumption that we observe transactions after they execute, but *before they commit*. That is, before their changes are actually persisted in the database and made visible to other transactions. After analyzing the executed transactions, we decide an order in which transactions commit. This is precisely how a system such as Aria [60] operates.

Assume we execute the above three transactions as τ_1 , τ_2 and τ_3 respectively. We let a and b represent tuples in the account table, c and d are tuples in the checking table, and e represents a tuple in the savings table. We define the set of transactions $T = \{\tau_1, \tau_2, \tau_3\}$, where

$$\begin{aligned}\tau_1 &= \langle R_1[a], R_1[e], W_1[e] \rangle; \\ \tau_2 &= \langle R_2[a], R_2[b], R_2[e], W_2[e], R_2[c], W_2[c], R_2[d], W_2[d] \rangle; \text{ and} \\ \tau_3 &= \langle R_3[a], R_3[e], R_3[c], W_3[c] \rangle.\end{aligned}$$

In our model, an update is a read followed by a write. Conditional structures collapse into the branch that was taken during execution. Note also that multiple successive reads or writes on the same tuple are collapsed into a single read or write on that tuple.

Schedules

When executing multiple transactions, we have to ensure that operations do not interfere with each other. Two reads from different transactions on the same object can always execute in parallel, but any other combination is considered a problem. This is due to writes requiring exclusive access to the object. Formally, let T be a set of transactions. We define $\text{ops}(T)$ as the set $\bigcup_{\tau \in T} \text{ops}(\tau)$, and we say that two operations $o, p \in \text{ops}(T)$ are *conflicting (in T)* if o and p are on the same object, from different transactions, and at least one of them is a write.

We can now define a schedule for T , which assigns operations to a point in time. To model time we use the set of positive integers including 0, denoted by $\mathbb{Z}_{\geq 0}$.

Definition 4. Let T be a set of transactions. A *schedule* s for T is a total function $s : \text{ops}(T) \rightarrow \mathbb{Z}_{\geq 0}$, with:

1. for all $\tau \in T$, and for all $o, p \in \text{ops}(\tau)$, if $o <_{\tau} p$ then $s(o) < s(p)$; and
2. for all conflicting $o, p \in \text{ops}(T)$, it must hold that either $s(o) < s(p)$ or $s(p) < s(o)$.

Intuitively, the first property (1) expresses that schedules must preserve the order of operations that is already present in each transaction. The second property (2) gives the guarantee that conflicting operations do not execute at precisely the same time. Additionally, we will write $\text{time}(s, i)$ to refer to the set of operations at time i , namely $\{o \mid s(o) = i, o \in \text{ops}(T)\}$, given a schedule s for T .

A schedule s for T is called *dense* if, for every operation $o \in \text{ops}(T)$, either $s(o) = 0$ or there exists an operation $p \in \text{ops}(T)$ such that $s(o) = s(p) + 1$ and either o conflicts with p or they belong to the same transaction. In other words, this means that a schedule is dense if there are no operations that can be moved earlier in the schedule without invalidating the properties in Definition 4.

We say that a schedule is *serial* if it executes transactions one by one. Formally, a schedule s for T is *serial* if, for all $o, p \in \text{ops}(T)$ either $s(o) < s(p)$ or $s(p) < s(o)$ (i.e., it has no concurrent operations); and for all transactions $\tau_i, \tau_j \in T$ with $\tau_i \neq \tau_j$, and all distinct operations $o, p \in \text{ops}(\tau_i)$ with $s(o) < s(p)$, there exists no operation $q \in \text{ops}(\tau_j)$ such that $s(o) < s(q) < s(p)$ (i.e., it is non-interleaving).

Since we want to compare the efficiency of schedules, we need to define a metric that captures what this means. We choose the notion of makespan, which is common in the scheduling literature [20, 73], and adapt it to the setting of transactions. In short, it is the finishing time of the final operation in the schedule.

Definition 5. Let T be a set of transactions, and let s be a schedule for T . The *makespan* of s , written $\text{span}(s)$, is $\max\{s(o) + 1 \mid o \in \text{ops}(T)\}$.

In most of the related literature, schedules are defined as total orders of operations [54, 55, 72, 89, 92]. A notable exception to this is the work of Weikum and Vossen [95], which uses partial orders directly, compared to our function-based definition that induces a partial order. We deviate from this because we need to reason about the makespan of schedules, instead of exclusively considering their correctness.

To see the connection, consider some transaction set T and a schedule s for T . If schedules were defined as total orders instead, then for all pairs of distinct operations $o, p \in \text{ops}(T)$ that execute in parallel (i.e., $s(o) = s(p)$), we freely pick an order on $\{o, p\}$ to obtain a total order for $\text{ops}(T)$.

Example 3. Consider the set of transactions T defined in Example 2. We now define three different schedules. First, let s_1 be a schedule for T such that the following sets of operations appear at each time step:

$$\begin{aligned} \text{time}(s_1, 0) &= \{R_1[a], R_2[a], R_3[a]\}, & \text{time}(s_1, 1) &= \{R_2[b]\}, \\ \text{time}(s_1, 2) &= \{R_2[e]\}, & \text{time}(s_1, 3) &= \{W_2[e]\}, \\ \text{time}(s_1, 4) &= \{R_1[e], R_2[c], R_3[e]\}, & \text{time}(s_1, 5) &= \{W_1[e], W_2[c]\}, \\ \text{time}(s_1, 6) &= \{R_2[d], R_3[c]\}, & \text{time}(s_1, 7) &= \{W_2[d], W_3[c]\}. \end{aligned}$$

A more intuitive way to define and visualize s_1 is through the use of a Gantt chart as seen in Figure 2.1a. We additionally define the schedules s_2 and s_3 in Figure 2.1b and Figure 2.1c respectively. Furthermore, the makespan of the schedules is given by $\text{span}(s_1) = 8$, $\text{span}(s_2) = 10$, and $\text{span}(s_3) = 8$. Hence, both s_1 and s_3 are more efficient than s_2 .

	0	1	2	3	4	5	6	7	8	9	10
τ_1	$R_1[a]$				$R_1[e]$	$W_1[e]$					
τ_2	$R_2[a]$	$R_2[b]$	$R_2[e]$	$W_2[e]$	$R_2[c]$	$W_2[c]$	$R_2[d]$	$W_2[d]$			
τ_3	$R_3[a]$				$R_3[e]$		$R_3[c]$	$W_3[c]$			

(a) A schedule s_1 for T .

	0	1	2	3	4	5	6	7	8	9	10
τ_1	$R_1[a]$	$R_1[e]$	$W_1[e]$								
τ_2	$R_2[a]$	$R_2[b]$			$R_2[e]$	$W_2[e]$	$R_2[c]$	$W_2[c]$	$R_2[d]$	$W_2[d]$	
τ_3	$R_3[a]$			$R_3[e]$	$R_3[c]$	$W_3[c]$					

(b) A schedule s_2 for T .

	0	1	2	3	4	5	6	7	8	9	10
τ_1	$R_1[a]$	$R_1[e]$			$W_1[e]$						
τ_2	$R_2[a]$	$R_2[b]$	$R_2[e]$	$W_2[e]$	$R_2[c]$	$W_2[c]$	$R_2[d]$	$W_2[d]$			
τ_3	$R_3[a]$	$R_3[e]$	$R_3[c]$	$W_3[c]$							

(c) A schedule s_3 for T .

Figure 2.1 – Three schedules for T . Time advances from left to right, and is given at the top of each diagram. Each row is a transaction, and each operation is a box. Darker boxes signify membership in a conflict pair.

There is, however, an issue with s_3 . Namely, it causes an inconsistent state in the database. To see why, we have to specifically focus on the object e , which represents a tuple in the savings table:

1. τ_1 reads from e . Assume it reads a balance x .
2. Next, τ_2 reads from e , it also reads the balance x .
3. τ_2 then writes to e to zero the savings account, hence the updated balance is now $x' = 0$.
4. Finally, τ_1 writes to e , increasing the previously read x by a value b , hence the balance is now $x'' = x + b$.

We now end up in a situation where τ_2 's changes to e have been lost. This is a typical example of a *lost update* anomaly [18], illustrating that our definition of schedules does not prevent

anomalies from occurring.

Conflict Serializability

As Example 3 demonstrates, our current definition of schedules allows for executions that could invalidate shared state. Hence, we have to limit the schedules we consider to those that are *correct*. The standard notion of correctness for schedules is conflict serializability. It has quite some history, which we elaborate on in Section 5.1. In brief, a schedule is serializable if its resulting state is indistinguishable from a state that could have been obtained by the transactions executing one by one. Next, we will make this formal.

Let T be a set of transactions, and let s be a schedule for T . Given two conflicting operations o and p in $\text{ops}(T)$, if $s(o) < s(p)$, then p *depends on* o in s . We write this as $o \rightarrow_s p$. Now, we define conflict serializable schedules.

Definition 6 (Papadimitriou [72]). Let T be a set of transactions, and let s_1 and s_2 be schedules for T . We say that,

- s_1 is *conflict equivalent* to s_2 if $o \rightarrow_{s_1} p$ implies $o \rightarrow_{s_2} p$ and vice versa, for all conflicting $o, p \in \text{ops}(T)$; and
- s_1 is *conflict serializable* if there exists a serial schedule s_3 for T such that s_1 is conflict equivalent to s_3 .

Henceforth, we will also write *serializable* when we mean conflict serializable, as we will not be discussing other types of serializability until Section 5.1. To decide if a schedule s for T is serializable, we can avail of a connection to cycle checking in graphs.

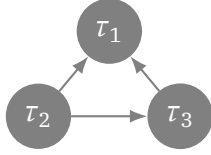
We can now define a *conflict graph* for s as a directed graph $\text{CG}(s) = (T, E)$ where $(\tau_i, \tau_j) \in E$ if there exist conflicting operations $o \in \tau_i$ and $p \in \tau_j$, with $\tau_i \neq \tau_j$, such that $o \rightarrow_s p$. Then, the following theorem can be applied.

Theorem 1 (Papadimitriou [72]). Let T be a transaction set. A schedule s for T is conflict serializable if, and only if, $\text{CG}(s)$ is acyclic.

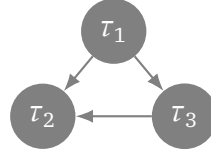
We briefly demonstrate this result in Example 4.

Example 4. Let us look at the conflict graphs of s_1 , s_2 , and s_3 in Figure 2.2.

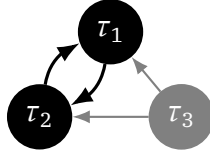
It is clear that s_1 and s_2 have an acyclic conflict graph (*cf.*, Figure 2.2a and Figure 2.2b respectively). Therefore, we can conclude that they are serializable schedules. However, the conflict graph of s_3 (*cf.*, Figure 2.2c) has a cycle, namely $\langle \tau_1, \tau_2 \rangle$. It is hence cyclic, and as such s_3 is not serializable.



(a) Conflict graph $CG(s_1)$.



(b) Conflict graph $CG(s_2)$.



(c) Conflict graph $CG(s_3)$ with a cycle colored in black.

Figure 2.2 – Conflict graphs for the schedules in Example 3.

3 Preliminaries: Computational Problems

Decision Problems

Most computer scientists are already familiar with decision problems, though we will briefly introduce them again at a higher level of abstraction, as they are used in this way in the rest of the thesis. This is also to explicitly contrast them to optimization problems, which are defined later. Intuitively, a decision problem asks us to formulate a solution in terms of an algorithm that outputs either *yes* or *no* (*i.e.*, it answers a proposition). In a formal treatment, we define formal languages and Turing machines first.

Briefly, a (*formal*) *language* is a set of finite sequences of elements (called *words*) drawn from a set called the *alphabet*. A *Turing machine* is a formal model for specifying algorithms which we (for our purposes) consider to be a black box that takes words as input and then either halts and outputs *accept* or *reject* for that input word, or it keeps running forever.

A language is referred to as a *decidable language* if there exists a Turing machine that always halts with *accept* for words that are part of that language, and halts with *reject* for those that do not. Note that a decider must halt on every input, if the machine does not halt on input words not in the language, it is called a *recognizable language* instead. A *decidable problem* is hence a problem that can be defined as a decidable language, this definition process is more commonly called *encoding*. There exist much more detailed treatments of these definitions elsewhere, for instance, in the work by Sipser [81].

Most computer scientists are also adequately convinced by a written algorithm specification, or by pseudocode, rather than a fully formal Turing machine definition¹. As such, we will not spend time belaboring the encoding of various problems, as this is only necessary if we would be working with Turing machines directly. Instead, we will define new decision problems using the box notation demonstrated next.

DECPROBLEM	
Input:	a definition for a valid (input) instance to the problem.
Output:	a definition for when to <i>accept</i> or <i>reject</i> the instance.

We can now apply this to the context of transactions, namely by formulating the conflict serializability problem. We do this in the next example.

¹To which I dearly hope you belong, otherwise I am deeply sorry for depriving you from the joy of decoding Turing machine state transition diagrams.

Example 5. We define the conflict serializability problem as follows.

SERIALIZABILITY	
Input:	a schedule s for a set of transactions T .
Output:	<i>accept</i> if s is conflict serializable, <i>reject</i> otherwise.

Algorithm 1 (described at a high-level by [72]) solves this problem by applying Theorem 1. It receives a schedule s for some transaction set T as input. The set T can be derived from s and does not necessarily need to be an input.

Algorithm 1 serializable?(s : schedule for T) $\rightarrow$ *accept* | *reject*

```

1:  $(T, E) \leftarrow \text{CG}(s)$ 
2:  $P \leftarrow$  an empty set (of partially processed vertices)
3:  $F \leftarrow$  an empty set (of fully processed vertices)
4: for all  $\tau_{\text{src}} \in (T \setminus F)$  do
5:    $S \leftarrow$  a new stack with the element  $\tau_{\text{src}}$ 
6:    $P \leftarrow P \cup \{\tau_{\text{src}}\}$ 
7:   while  $S$  is not empty do
8:      $\tau \leftarrow \text{head}(S)$ 
9:      $A \leftarrow \{\tau' \mid (\tau, \tau') \in E, \text{ and } \tau' \notin P \cup F\}$ 
10:    if  $A$  is not empty then
11:       $\tau' \leftarrow$  any transaction in  $A$ .
12:      if  $\tau' \in P$  then
13:        output reject
14:         $\text{push}(S, \tau')$ 
15:         $P \leftarrow P \cup \{\tau'\}$ 
16:      else
17:         $F \leftarrow F \cup \{\tau\}$ 
18:         $\text{pop}(S)$ 
19: output accept

```

We perform depth-first search through the conflict graph of s , starting at each node (lines 4–18). It immediately outputs *reject* (line 13) upon encountering a back edge. Recall that a

back edge is an edge that leads back to an ancestor node in the depth-first search path [27]. If no back edge was encountered, then the graph is acyclic, and the algorithm outputs *accept* (line 19).

Indeed, if the graph is acyclic then by applying Theorem 1, we can conclude that s is conflict serializable. Note that Algorithm 1 always terminates, as T is finite, and therefore E is also finite.

Optimization Problems

An often less familiar topic is that of optimization problems. A (*combinatorial*) *optimization problem* associates to each input instance: a set of *valid solutions*; a nonnegative *cost* (e.g., from $\mathbb{Z}_{\geq 0}$) associated to each valid solution; and an objective to either *minimize* or *maximize* the cost.

The goal of an optimization problem is to find a valid solution of either minimal or maximal cost for a given input instance. We will accordingly say *minimization* or *maximization* problem when the objective is to either minimize or maximize the cost, respectively. The minimal (*resp.* maximal) cost for an input instance x is denoted by $\text{opt}(x)$. As a shorthand, whenever we define an optimization problem, we will present it in a box format similar to that of decision problems. We provide an example of this below.

OPT	
Input:	a definition of a valid input instance for the problem.
Output:	a definition of a valid solution instance for the problem.
Goal:	a definition of the cost for a solution instance, and whether to minimize or maximize that cost.

4 The Transaction Scheduling Problem

We define the central problem in this thesis. The core idea is to find a serializable schedule with a minimal makespan for any given transaction set.

TxSCHED	
Input:	A set of transactions T .
Output:	A conflict serializable schedule s for T .
Goal:	Minimize $\text{span}(s)$.

In the remainder of this section, we will set up some useful tools and define an algorithm that solves this problem.

Precedence Graphs

In what follows, it is sometimes useful to consider a schedule without being concerned about the specific timing assigned to operations. The following schedule representation lets us focus exclusively on the relation between operations, and also provides a data structure for which we can build algorithms.

Definition 7. Let T be a set of transactions, and let s be a schedule for T . A *precedence graph* is defined as a directed graph $P_s = (\text{ops}(T), E_1 \cup E_2)$, with:

$$E_1 = \{(\tau(i), \tau(i+1)) \mid \tau \in T, i \in [|\tau| - 1]\}, \text{ and}$$

$$E_2 = \{(o, p) \mid o, p \in \text{ops}(T), o \rightarrow_s p\}.$$

Additionally, we can also make precedence graphs from total orders on the transaction set. Let T be a set of transactions, and let $\Sigma = (T, \leq_T)$ be a total order for T . Then the *precedence graph for Σ* , denoted P_Σ is defined by replacing E_2 in Definition 7 by the set of pairs (o, p) such that $o \in \text{ops}(\tau_i)$, $p \in \text{ops}(\tau_j)$ and $\tau_i <_T \tau_j$, where τ_i and τ_j are distinct transactions in T .

Example 6. We demonstrate what a typical precedence graph looks like in Figure 2.3. We use the serializable schedule s_1 defined in Example 3. By convention, black nodes denote that operations are part of a conflict pair, as do black edges. Gray nodes are not conflicting.

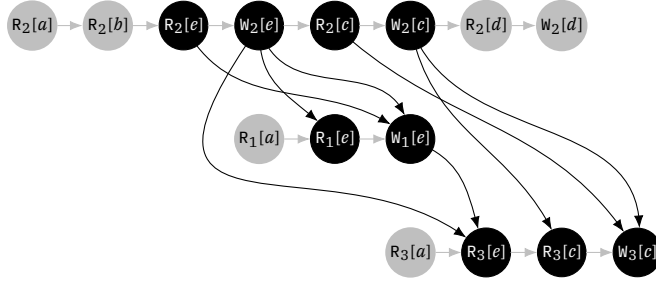


Figure 2.3 – Precedence graph P_{s_1} for s_1 .

Notice that we can obtain an identical precedence graph P_Σ if Σ orders T such that $\tau_2 <_T \tau_1 <_T \tau_3$. Corollary 2 will show that we can construct such a total order for any conflict serializable schedule.

As a consequence of Theorem 1, if our schedule is conflict serializable, then this necessarily implies that our precedence graph must be acyclic as well. Intuitively, this is because our precedence graph is almost homomorphic to the conflict graph. That is, if we make a function mapping each operation to its corresponding transaction, then applying it over all vertices gives us a directed graph with self-loops on transactions with more than one operation. If we ignore those self-loops we get precisely the conflict graph. We now prove this in a different and more formal manner.

Corollary 1. Let s be a schedule for a transaction set T . If s is conflict serializable, then P_s is acyclic.

Proof. We prove this by contradiction. Let s be a conflict serializable schedule, and let $\text{CG}(s) = (T, E_C)$ be its conflict graph, which is acyclic by Theorem 1. Assume $P_s = (\text{ops}(T), E_P)$ is cyclic. This

means that there is a path $\langle o_1, o_2, \dots, o_k \rangle$ in P_s such that $(o_k, o_1) \in E_P$. Hence, either o_1 and o_k are from the same transaction, or they are not.

Case 1 *If there is a transaction $\tau \in T$ such that $\{o_1, o_k\} \subseteq \text{ops}(\tau)$, then $o_1 <_\tau o_k$ and $o_k <_\tau o_1$. However, $(\text{ops}(\tau), <_\tau)$ is a strict total order. By asymmetry, if $o_1 <_\tau o_k$ then $o_k <_\tau o_1$ is false, and if we have that $o_k <_\tau o_1$ then $o_1 <_\tau o_k$ is false. Since both antecedents are simultaneously true, this means we have a contradiction.*

Case 2 *If there are distinct transactions $\tau_i, \tau_j \in T$ such that $o_1 \in \tau_i$ and $o_k \in \tau_j$, then we have that $o_1 \rightarrow_s o_k$ and $o_k \rightarrow_s o_1$. By the definition of $\text{CG}(s)$ this implies $\{(o_1, o_k), (o_k, o_1)\} \subseteq E_C$. However, we can then construct a cycle $\langle o_1, o_k \rangle$ in $\text{CG}(s)$, which is supposed to be acyclic per Theorem 1. This is a contradiction.*

Since both cases lead to a contradiction against our assumption that P_s is cyclic, we can conclude that P_s must be acyclic. $\square$

Another interesting consequence of Theorem 1, is that conflict serializable schedules reflect a certain total order over the transactions. This is also true for the precedence graphs that correspond with both the schedule and the order.

Corollary 2. Let s be a schedule for a transaction set T . If s is conflict serializable, then there is a total order Σ for T such that $P_\Sigma = P_s$.

Proof. We prove this by construction. Let s be a conflict serializable, and let $\text{CG}(s) = (T, E_C)$ be its conflict graph, which is acyclic by Theorem 1. Let $C = \{C_1, C_2, \dots, C_m\}$ be the partition of strongly connected components in $\text{CG}(s)$, and let $(C, \leq_C)$ be an arbitrarily chosen total order on C . Now, we define $(T, \leq_T)$ on T by letting $\tau_i \leq_T \tau_j$ for all $\tau_i, \tau_j \in T$ with either:

1. $\tau_i = \tau_j$; or
2. $(\tau_i, \tau_j) \in E_C$; or
3. $\tau_i \in C_h$ and $\tau_j \in C_q$ with distinct strongly connected components $C_h, C_q \in C$ such that $C_h \leq_C C_q$; or

4. there exists a $\tau_p \in T$ such that $\tau_i \leq_T \tau_p$ and $\tau_p \leq_T \tau_j$.

We now verify that $(T, \leq_T)$ is a total order. Reflexivity, strong connectedness, and transitivity immediately follow from (1), (3) and (4) respectively. To show antisymmetry, assume that $\tau_i \leq_T \tau_j$ and $\tau_j \leq_T \tau_i$ does not imply $\tau_i = \tau_j$. Then, we get $\{(\tau_i, \tau_j), (\tau_j, \tau_i)\} \subseteq E_C$ per (2), and hence a cycle $\langle \tau_i, \tau_j \rangle$ exists in $CG(s)$. This is a contradiction against $CG(s)$ being acyclic. Hence, our assumption does not hold and $\leq_T$ must be antisymmetric.

Finally, we show that $P_\Sigma = P_s$. Let $P_s = (\text{ops}(T), E_1 \cup E_2)$. Contrasting the definition of both, the only difference is the set of edges E_2 which is entirely determined by conflicts in s . Hence, given distinct operations $o \in \tau_i$ and $p \in \tau_j$ from distinct transactions $\tau_i, \tau_j \in T$, this boils down to showing $o \rightarrow_s p$ if, and only if, $\tau_i <_T \tau_j$. However, by definition of conflict graphs, $o \rightarrow_s p$ is equivalent to $(\tau_i, \tau_j) \in E_C$. And, by (2) this is equivalent to $\tau_i <_T \tau_j$.

This shows that there exists a total order Σ , such that $P_\Sigma = P_s$. $\square$

Finally, we demonstrate that there is a connection between the maximal path length in the precedence graph, and the makespan of conflict serializable schedules. This demonstrates that we can compute the maximal path in order to compute the makespan.

Proposition 1. Let T be a set of transactions. Then, for all conflict serializable schedules s for T , we have that $\text{max-path}(P_s) + 1 \leq \text{span}(s)$. If s is dense, then we have that $\text{max-path}(P_s) + 1 = \text{span}(s)$.

Proof. We prove $\text{max-path}(P_s) + 1 \leq \text{span}(s)$ by induction on the number of vertices k on the longest path in the precedence graph.

Base Case ($k = 1$) If T is a transaction set and s is a schedule for T such that $\langle o_1 \rangle$ is a longest path in P_s , then we show, by contradiction, that $1 \leq \text{span}(s)$. Assume that $\text{span}(s) < 1$. This implies $\text{span}(s) = 0$, since it can never go below 0. Hence, it must be that $\text{ops}(T) = \emptyset$. However, the longest path is $\langle o_1 \rangle$, hence $o_1 \in \text{ops}(T)$, which contradicts $\text{ops}(T) = \emptyset$. From this, we conclude $1 \leq \text{span}(s)$.

Inductive Case For the induction hypothesis (IH), assume for any transaction set T' , any schedule s' for T' , and any $P_{s'}$ with a longest path $\langle o_1, o_2, \dots, o_{k-1} \rangle$, that $(k-1) + 1 \leq \text{span}(s')$ holds.

Now, let T be a transaction set with a schedule s for T , and with P_s a precedence graph with a longest path $\langle o_1, o_2, \dots, o_{k-1}, o_k \rangle$. We show that $k+1 \leq \text{span}(s)$ by contradiction. Assume that $\text{span}(s) < k+1$. Now, if we remove all final operations o_k from s we get a schedule s'' such that $((k-1) + 1 = k) \leq \text{span}(s'')$ by the IH.

If we add o_k back to s'' to obtain s we can consider one of two cases. Either (a) there exists a transaction $\tau \in T$ with $\{o_{k-1}, o_k\} \subseteq \text{ops}(\tau)$, or (b) there exist distinct transactions $\{\tau_1, \tau_2\} \subseteq T$ with $o_{k-1} \in \text{ops}(\tau_1)$ and $o_k \in \text{ops}(\tau_2)$, such that $o_{k-1} \rightarrow_s o_k$ holds.

(a) By Definition 7, we get $o_{k-1} <_\tau o_k$ and hence $(o_{k-1}, o_k) \in E$; or

(b) By Definition 7, $o_{k-1} \rightarrow_s o_k$ implies $(o_{k-1}, o_k) \in E$.

In either case we get that $k+1 \leq \text{span}(s'') + 1$. Since s'' is s with the final operations removed we get that $k+1 \leq \text{span}(s)$. This contradicts against our assumption that $\text{span}(s) < k+1$, and hence we conclude that $k+1 \leq \text{span}(s)$.

What remains to be shown is that if s is dense, then $k+1 = \text{span}(s)$. By the definition of a dense schedule, for every $p \in \text{time}(s, i)$ with $i > 1$, there is some operation $o \in \text{time}(s, i-1)$ such that $(o, p) \in E$. This immediately implies a longest path in P_s has k vertices. Hence, $\text{span}(s) \leq k = \text{max-path}(s) + 1$. From this, $k+1 = \text{span}(s)$. $\square$

Canonical Schedules

Next, we establish a way to turn precedence graphs based on total orders into serializable schedules. This should enable us to make an algorithm that can cycle through all the total orders on the transaction set, use those as a basis for a precedence graph, and then turn that precedence graph into a schedule.

We will call a schedule based on a precedence graph its *canonical schedule*. Intuitively, it is the schedule that we obtain by placing operations towards the front of the schedule as much as possible while respecting the precedence graph edges, and without leaving

unnecessary gaps between operations (*i.e.*, the constructed schedule must be dense). Formally, we define it as follows.

Definition 8. Let T be a transaction set, let $\Sigma = (T, \leq_T)$ be a total order for T , and let $P_\Sigma = (\text{ops}(T), E)$ be the associated precedence graph. Then, *the canonical schedule for Σ* is defined as the schedule $\text{cs}(\Sigma)$ for T such that,

$$\text{time}(\text{cs}(\Sigma), 0) = \{o \mid o \in \text{ops}(T), \nexists p \in \text{ops}(T) : (p, o) \in E\}$$

and, for all $i \in [1, \text{max-path}(P_\Sigma)]$

$$\text{time}(\text{cs}(\Sigma), m(o)+1) = \{o \mid p \in S_{j < i}, o \in (\text{ops}(T) \setminus S_{j < i}), (p, o) \in E\}$$

where $S_{j < i} = \bigcup_{j=0}^{i-1} \text{time}(\text{cs}(\Sigma), j)$, and with $m(o)$ the maximal time assigned to all operations $p \in S_{j < i}$ with $(p, o) \in E$.

Example 7. If we apply this definition to the precedence graph we have given in Example 6, which corresponds to P_Σ where Σ orders T by $\tau_2 <_T \tau_1 <_T \tau_3$ then we get the schedule s_1 defined in Example 3.

By way of illustrating the definition, we provide Algorithm 2.

Algorithm 2 $\text{canonical}(\Sigma: \text{total order for } T) \rightarrow \text{schedule for } T$

```

1:  $(\text{ops}(T), E) \leftarrow P_\Sigma$ 
2: for all  $i \in [1, \text{max-path}(P_\Sigma)]$  do
3:    $\text{time}(\text{cs}(\Sigma), i) \leftarrow \emptyset$ 
4:   for all  $o \in \text{ops}(T), \nexists p \in \text{ops}(T) : (p, o) \in E$  do
5:      $\text{time}(\text{cs}(\Sigma), 0) \leftarrow \text{time}(\text{cs}(\Sigma), 0) \cup \{o\}$ 
6:    $S \leftarrow \text{time}(\text{cs}(\Sigma), 0)$ 
7:   for all  $o \in (\text{ops}(T) \setminus S)$  do
8:     if  $\text{cs}(\Sigma)(p)$  is defined for all  $p \in S$  with  $(p, o) \in E$  then
9:        $m \leftarrow \max\{\text{cs}(\Sigma)(p) \mid p \in S, (p, o) \in E\}$ 
10:       $\text{time}(\text{cs}(\Sigma), m+1) \leftarrow \text{time}(\text{cs}(\Sigma), m+1) \cup \{o\}$ 
11:     $S \leftarrow S \cup \{o\}$ 
12: output  $\text{cs}(\Sigma)$ 
```

The way we construct canonical schedules makes them dense.

Proposition 2. For all transaction sets T , and for all total orders Σ for T , the canonical schedule $\text{cs}(\Sigma)$ is dense.

Proof. Let T be a transaction set, let $\Sigma = (T, \leq_T)$ be a total order for T , let $P_\Sigma = (\text{ops}(T), E)$, and let $s = \text{cs}(\Sigma)$ be the corresponding canonical schedule. We continue by induction on the maximal path length $k = \text{max-path}(P_\Sigma)$.

Base Case ($k = 0$) In this case the schedule has makespan at most 1 by Proposition 1, hence for all operations $o \in \text{ops}(T)$ there exist no operations $p \in \text{ops}(T)$ such that $(o, p) \in E$ as otherwise the makespan would exceed 1. Therefore, by Definition 8 we get that $\text{time}(s, 0) = \text{ops}(T)$. Since all operations are scheduled at time 0, s is dense.

Inductive Case For the induction hypothesis (IH), assume every s with $\text{max-path}(P_\Sigma) < k$ is dense. We show that if $\text{max-path}(P_\Sigma) = k$, then s is also dense.

Let $s = \text{cs}(\Sigma)$ with $\text{max-path}(P_\Sigma) = k$, and let $p \in \text{time}(s, k)$. By Definition 8, there exists an operation

$$o \in \left(\bigcup_{i=0}^{k-1} \text{time}(s, i) \right) \setminus \{p\} \text{ with } (o, p) \in E.$$

Either (a) $o \in \text{time}(s, i)$ with $i < k - 1$, or (b) $o \in \text{time}(s, k - 1)$.

- (a) We know that o and p can not belong to the same transaction as $(o, p) \in E$ and this would imply $o \in \text{time}(s, k - 1)$ which is case (b). Hence, by repeated application of the IH, there must exist a chain of conflicting operations $o, o_1, o_2, \dots, o_m$ such that

$$s(o) + 1 = s(o_1), s(o_1) + 1 = s(o_2), \dots, s(o_{m-1}) + 1 = s(o_m)$$

with $o_m \in \text{time}(s, k - 1)$ and $(o_m, p) \in E$. In which case we have just ended up in case (b) again, if we redefine o to be o_m .

- (b) If $o \in \text{time}(s, k - 1)$ then we have $s(o) + 1 = s(p)$ and then $(o, p) \in E$ implies that there exists a $\tau \in T$ such that $o <_\tau p$ by Definition 7, or o and p are conflicting. This satisfies the definition of a dense schedule.

Hence, in both cases, we have an operation $o \in \text{time}(s, k - 1)$ such that $s(o) + 1 = s(p)$ and such that either o and p belong to the same transaction, or o and p conflict. This is precisely the definition of a dense schedule, hence we conclude the proof. $\square$

Using Corollary 2 and Proposition 1 in conjunction, we can prove the following proposition.

Proposition 3. For all transaction sets T , there exists a total order Σ for T such that:

$$\text{max-path}(P_\Sigma) + 1 = \text{span}(\text{cs}(\Sigma)) = \text{opt}(T).$$

Proof. Let T be a transaction set and let s be a conflict serializable schedule such that $\text{span}(s) = \text{opt}(T)$. We can assume that s is dense without loss of generality, as otherwise we can repeatedly pick problematic operations and move them earlier in the schedule until the schedule becomes dense. The resulting schedule s' is conflict equivalent to s and is therefore still conflict serializable. We have that $\text{span}(s') \leq \text{span}(s)$ and, by choice, we get $\text{span}(s') = \text{span}(s)$.

From Proposition 1, we have $\text{opt}(T) = \text{span}(s) = \text{max-path}(P_s) + 1$. And, from Corollary 2, that $\text{max-path}(P_\Sigma) + 1 = \text{max-path}(P_s) + 1$ for some total order Σ for T .

Finally, observe that $\text{cs}(\Sigma)$ is defined in terms of P_Σ . Hence, we have that $\text{max-path}(P_\Sigma) + 1 \leq \text{span}(\text{cs}(\Sigma))$. And, because $\text{cs}(\Sigma)$ is dense by Proposition 2, we have that $\text{max-path}(P_\Sigma) + 1 = \text{span}(\text{cs}(\Sigma))$. Since $\text{opt}(T) = \text{max-path}(P_\Sigma) + 1$, this concludes the proof. $\square$

It follows immediately from Proposition 3 that we can find the optimal makespan for T if we enumerate all total orders on T , compute their maximal path length, and then select an order that has the minimal of those maximal path lengths.

Corollary 3. For all transaction sets T , we have that:

$$\begin{aligned} \text{opt}(T) &= \min \{ \text{max-path}(P_\Sigma) + 1 \mid \Sigma \text{ is a total order for } T \} \\ &= \min \{ \text{span}(\text{cs}(\Sigma)) \mid \Sigma \text{ is a total order for } T \}. \end{aligned}$$

Enumerative Transaction Scheduling

Inspired by Corollary 3, we conclude this chapter by defining an algorithm that can find an optimal conflict serializable schedule for any set of transactions.

We will refer to the following algorithm as the “*enumerative transaction scheduling algorithm*”, as it consists of enumerating all total orders for the transaction set. For each total order, we consider its accompanying precedence graph and then compute its longest path. Note that we can only do this longest path computation efficiently because the precedence graph is acyclic. Formally, we define the pseudocode in Algorithm 3.

Algorithm 3 $\text{enum_tx_sched}(T: \text{transaction set}) \rightarrow \text{schedule for } T$

```

1:  $\text{span}_{\min} \leftarrow |\text{ops}(T)|$ 
2:  $\Sigma_{\text{opt}} \leftarrow$  arbitrary total order for  $T$ 
3: for all total orders  $\Sigma$  on  $T$  do
4:    $(\text{ops}(T), E) \leftarrow P_{\Sigma}$ 
5:    $d_{\max} \leftarrow 0$ 
6:   for all  $o_{\text{src}}, p \in \text{ops}(T)$  and  $(p, o_{\text{src}}) \notin E$  do
7:      $Q \leftarrow$  empty queue
8:      $\text{enqueue}(Q, (o_{\text{src}}, 0))$ 
9:     while  $Q$  is not empty do
10:       $(o, d) \leftarrow \text{serve}(Q)$ 
11:      if  $d > d_{\max}$  then
12:         $d_{\max} \leftarrow d$ 
13:        for all  $(o, p) \in E$  do
14:           $\text{enqueue}(Q, (p, d + 1))$ 
15:      if  $d_{\max} + 1 < \text{span}_{\min}$  then
16:         $\text{span}_{\min} \leftarrow d_{\max} + 1$ 
17:         $\Sigma_{\text{opt}} \leftarrow \Sigma$ 
18: output  $\text{canonical}(\Sigma_{\text{opt}})$ 

```

We now prove that this algorithm indeed solves the optimization problem TxNSCHED .

Theorem 2. Let T be any transaction set, and let s be the output of $\text{enum_tx_sched}(T)$, then s is a conflict serializable schedule with $\text{span}(s) = \text{opt}(T)$.

Proof. Let T be a transaction set, and let $s = \text{enum_tx_sched}(T)$.

To see that s is conflict serializable, note that s is a canonical schedule constructed from a total order $\Sigma_{\text{opt}} = (T, \leq_T)$ on T (line 18).

Hence, assume that s is not conflict serializable then there exists a cycle $\langle o_1, o_2, \dots, o_m \rangle$ in $P_{\Sigma_{\text{opt}}} = (\text{ops}(T), E)$. Notice that all $o_1, \dots, o_m$ are distinct by definition. By the definition of precedence graphs, this implies that there exist (possibly non-distinct) pairs of transactions $\tau_1, \tau_2 \in T$ such that $o_1 \in \text{ops}(\tau_1)$ and $o_m \in \text{ops}(\tau_2)$, and with $(o_m, o_1) \in E$.

In case $\tau_1 = \tau_2$ then $o_1 \leq_{\tau_1} o_m$ and $o_m \leq_{\tau_1} o_1$ even though it is the case that $o_1 \neq o_m$. This implies that the order of operations is not total (indeed it violates antisymmetry of $\leq_{\tau_1}$). This contradicts the definition of transactions.

Otherwise, if $\tau_1 \neq \tau_2$, then the cycle in $P_{\Sigma_{\text{opt}}}$ implies $\tau_1 \leq_T \tau_2$ and $\tau_2 \leq_T \tau_1$ even though $\tau_1 \neq \tau_2$, which again violates antisymmetry. This contradicts that Σ_{opt} is a total order. Since we obtain a contradiction in both cases, our assumption that s is not conflict serializable must be false, hence we conclude that s is conflict serializable.

To see that $\text{span}(s) = \text{opt}(T)$, we remark that lines 3–17 compute the longest path in the precedence graph P_{Σ} for every total order Σ on T . More concretely, by Corollary 3, we must compute

$$\min \{ \text{max-path}(P_{\Sigma}) + 1 \mid \Sigma \text{ is a total order for } T \},$$

which is equal to $\text{opt}(T)$. In Algorithm 3, $\text{max-path}(P_{\Sigma})$ is computed by lines 4–14 through a breadth-first search that keeps track of the distance from each source vertex. Then, we find the minimal $\text{max-path}(P_{\Sigma}) + 1$ by iterating over all orders keeping the smallest such value (*c.f.*, lines 15–17). This concludes the proof. $\square$

5 Summary

In this chapter, we have introduced the following core concepts.

- In Section 2.2, we defined operations, transactions, schedules, and conflict serializability as a formal definition for the safe execution of a set of transactions.

- Section 2.4 defined the transaction scheduling problem as an optimization problem. The rest of the section introduced an algorithm for this problem.
 - We defined precedence graphs as an alternative representation of schedules that let us more easily focus on the relations between operations.
 - Next, we defined canonical schedules as a way to turn precedence graphs back into schedules while preserving some interesting properties. Namely, the longest path in the precedence graph plus one (for the cost of executing the final operation) corresponds to the makespan of the canonical schedule.
 - Lastly, we defined the enumerative scheduling algorithm. This enumerates all total orders of the transaction set, then creates a precedence graph for this order, and computes the longest path. We keep track of the order that had the shortest longest path.

Further Reading

The material in this chapter was first introduced in an ICDT 2026 paper by Brecht Vandervoort, Bas Ketsman, and myself titled “*Bounding the Makespan of Transaction Schedules*” [14].

For more context on the formalization of transactions, some useful resources (in no particular order) are: the paper “*Concurrency Control for Database Theorists*” [54]; Brecht Vandervoort’s PhD Thesis [89]; Weikum and Vossen’s “*Transactional Information Systems*” [95]; and Papadimitriou’s “*The Theory of Database Concurrency Control*” [72].

For other background material, I used Sipser’s book titled “*Introduction to the Theory of Computation*” [81], and Halmos’ “*Naive Set Theory*” [42]. Anything else was probably knowledge that was latent in my brain somewhere. If a mistake managed to sneak in, I take full responsibility.

Time Complexity

1 Preliminaries: Time Complexity Classes

NP, P, NP-hardness and NP-completeness

Time complexity is concerned with the running time of algorithms, and classifying problems based on this running time. The main goal is to understand whether there exist efficient algorithms for a problem, or whether the problem is inherently difficult to solve. Most computer scientists first encounter time complexity in the form of asymptotic time complexity. A function $g : \mathbb{N} \rightarrow \mathbb{R}^+$ is an (*asymptotic*) *upper bound* for an algorithm if, there exist constants $c, n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ the number of steps of the algorithm is bounded by $c \cdot g(n)$.

Example 8. The algorithm `enum_tx_sched` we defined in Section 2.4 has an upper bound of $|T|! \cdot |\text{ops}(T)|^2$. To see why, we enumerate all total orders (of which there are $|S|!$ for a set S) in line (3), and for each of these orders we perform a number of breadth-first searches in lines (6–14). A breadth-first search is always linear in the size of the graph (V, E) hence this is either $|V| + |E|$ or $|V|^2$ operations depending on the representation of the graph as an adjacency list or adjacency matrix. Notice that $|V| + |E| \leq |V|^2$, hence we can use the latter as an upper bound for the breadth-first search.

If there exists an algorithm for a decision problem `DecProblem` with an upper bound $g(n)$, then we write $\text{DecProblem} \in \mathcal{O}(g(n))$. When g is polynomial in n , we say the algorithm runs in *polynomial time*.

Definition 9. A decision problem $\text{DEC}_{\text{PROBLEM}}$ is a member of P if $\text{DEC}_{\text{PROBLEM}} \in O(g(n))$ where $g(n)$ is a polynomial function.

Another important complexity class is called NP, which is the class of problems with polynomial time verifiable solutions. We can think of this as a game where someone hands you proofs that the instance accepts for this problem, and we have to check if their proof is correct in polynomial time. This proof of acceptance is more often called a *witness*, and is some mathematical structure like a set of numbers, a graph coloring, or a path in a graph.

Definition 10. A decision problem is a member of the NP class if its witnesses are polynomial time verifiable.

Clearly, $P \subseteq NP$. Whether $P = NP$ or $P \neq NP$ is still one of the biggest open problems in computer science. We briefly illustrate these concepts with the next example.

Example 9. For an example of a decision problem in P , consider the **SERIALIZABILITY** problem we defined in Example 5. It is clear that the algorithm `serializable?` performs a depth-first search in the conflict graph. This takes at most $|T|^2$ steps, which is clearly polynomial. Hence $\text{SERIALIZABILITY} \in O(|T|^2)$, and therefore $\text{SERIALIZABILITY} \in P$.

To see that $\text{SERIALIZABILITY} \in NP$, notice that the graph itself can serve as a witness. And, since we can verify that there are no cycles in this graph by doing a depth-first search (as we have previously established), we can verify this graph is indeed a witness in polynomial time.

Within NP, there is a subset of problems which is considered difficult to solve. Intuitively, it is a class of problems where a solution can only be obtained by enumerating all possible witnesses. Therefore, if we can prove that a problem is in this class, then it is reasonable to assume we cannot find a better algorithm other than guessing and checking. This class of problems is called NP-hard. If the problem is additionally in NP, then we say it is NP-complete.

To prove that a problem is NP-hard, we have to show there exists a polynomial time reduction from that problem to our problem. This

indicates that our problem is at least as hard as solving that original problem. Concretely, a (*Karp*) *reduction* from a decision problem DEC_1 to a decision problem DEC_2 is an algorithm that takes inputs for DEC_1 and transforms them into valid inputs for DEC_2 such that if DEC_2 accepts or rejects on that input, then DEC_1 must also accept or reject on that input. We write the expression $\text{DEC}_1 \leq_P \text{DEC}_2$, which reads as: “ DEC_1 *polynomial time reduces to* DEC_2 ”.

Definition 11. A decision problem $\text{DEC}_{\text{PROBLEM}}$ is *NP-hard* if there exists a polynomial time reduction from any NP-hard problem to $\text{DEC}_{\text{PROBLEM}}$. If $\text{DEC}_{\text{PROBLEM}}$ is additionally in NP, then we say that $\text{DEC}_{\text{PROBLEM}}$ is *NP-complete*.

We will see a few examples of Karp reductions in this chapter.

NPO, PO, and NP-hard Optimization

There are also complexity classes for optimization problems. The ones we will encounter in this chapter are the NPO and PO classes, or the “*NP Optimization problems*” and “*P Optimization problems*” to be more precise. These are more or less analogous to the classes NP and P for decision problems. We define NPO first.

Definition 12 (Hromkovič [46]). An optimization problem is a member of the NPO class if:

1. the inputs and solutions can be recognized in polynomial time;
2. the solution instances are polynomially sized; and
3. the cost of a solution instance can be computed in polynomial time.

Next, we define the class PO. Intuitively, this class corresponds to the class of optimization problems that is “efficient to compute”. Just as is the case for NP and P, it is still an open problem whether $\text{NPO} = \text{PO}$ or $\text{NPO} \neq \text{PO}$ [46].

Definition 13 (Hromkovič [46]). An optimization problem is a member of the PO class if:

1. it is a member of the NPO class, and
2. there exists a polynomial time algorithm that computes an optimal solution for every input instance.

One way to prove membership in NPO or PO for an optimization problem OPT is to show that its *threshold problem* is in NP or P respectively [46]. We use the notation $D(\text{OPT})$ for this. Formally, it is defined as follows.

$D(\text{OPT})$	
Input:	An input instance x for OPT , and a nonnegative cost k .
Output:	<i>true</i> if $\text{opt}(x) \leq k$ (resp. $\text{opt}(x) \geq k$) and OPT is a minimization (resp. maximization) problem, and <i>false</i> otherwise.

Finally, we introduce the notion of an *NP-hard optimization problem*. Notice that this is not entirely analogous to NP-hardness for decision problems. The reason is that reductions between two NPO-problems rarely preserve the cost function [46]. There exist more fine-grained notions for comparing the difficulty of NP-hard optimization problems, such as approximation complexity and parameterized complexity, but they are outside the scope of this thesis. We do discuss approximation algorithms in Chapter 4.

Definition 14 (Hromkovič [46]). An optimization problem is called an *NP-hard optimization problem* if its threshold problem is NP-hard.

2 The Complexity of Transaction Scheduling

We now investigate the time complexity of the transaction scheduling problem. In the previous chapter, we presented an algorithm that has to enumerate all possible total orders on the transaction set, which seems rather inefficient. In this section, we build up to the following result, which indicates that an algorithm doing something fundamentally faster than enumerating all total orders is unlikely to exist, unless $P = NP$.

Theorem 3. $D(\text{TxSCHED}) \in \text{NP-complete}$.

In what follows, we will refer to $D(\text{TxSCHED})$ as *SCHEDULABILITY*. From

the above result, the following corollary then immediately follows.

Corollary 4. TxSCHED is an NP-hard optimization problem.

To see that SCHEDULABILITY is in NP is straightforward. Indeed, we can consider schedules to be witnesses, and we can verify whether a schedule is a valid conflict serializable schedule by executing the polynomial time algorithm serializable? defined in Example 5 of Chapter 2. To see that the problem is NP-hard, we construct a reduction from a known NP-hard problem called 3-COLORABILITY to 2-ORIENTABILITY, and from 2-ORIENTABILITY to SCHEDULABILITY .

From Colorability to Orientability

We first introduce the 3-COLORABILITY problem. The more general COLORABILITY problem, where the number of colors is part of the input, is one of Karp's original 21 NP-complete problems [53]. Karp published this work in 1972, and it is a well-known paper introducing many graph-based NP-complete problems. In that paper, COLORABILITY was shown to be NP-complete by reduction from 3-SAT, which in turn was reduced from SAT, which is NP-complete by the Cook-Levin Theorem (also in [53]). In 1973, Stockmeyer showed that COLORABILITY is still NP-complete even if we require exactly three colors [84], which is the problem we present below.

3-COLORABILITY

Input: an undirected graph $G = (V, E)$.

Output: *accept* if there exists a function $f : V \rightarrow [3]$ (i.e., a 3-coloring) with $f(u) \neq f(v)$ if $\{u, v\} \in E$ for all $u, v \in V$, *reject* otherwise.

Theorem 4 (Stockmeyer [84]). 3-COLORABILITY is NP-complete.

Next, we introduce the 2-ORIENTABILITY problem, which is closely related to COLORABILITY. First, given an undirected graph $G = (V, E)$ and a total order $\leq_V$ on the nodes of G , we write $\text{OG}(G, \leq_V)$ to denote the *orientation* of G based on $\leq_V$, that is, the directed graph with nodes V and with directed edges $\{(u, v) \mid \{u, v\} \in E, u \leq_V v\}$. Now, the 2-ORIENTABILITY problem asks whether, for an undirected

graph, we can find an acyclic orientation for that graph such that the length of each directed path is bounded by 2. Formally, we define this as follows.

2-ORIENTABILITY	
Input:	an undirected graph $G = (V, E)$.
Output:	<i>accept</i> if there is an acyclic orientation $\text{OG}(G, \leq_V)$ for G such that the length of all paths in $\text{OG}(G, \leq_V)$ is at most 2, <i>reject</i> otherwise.

This problem and 3-COLORABILITY are closely related to one another through the following theorem that originates from the Graph Theory literature.

Theorem 5 (Gallai-Hasse-Roy-Vitaver [23]). Let G be an undirected graph. There exists a coloring for G with k colors, if and only if, there exists an acyclic orientation $\text{OG}(G, \leq_V)$ for G such all paths in $\text{OG}(G, \leq_V)$ are of length at most $k - 1$.

In other words, if a graph is colorable with k colors, then it is also $(k - 1)$ -orientable and vice versa. The next result about the problem's complexity thus readily follows from this insight.

Corollary 5. 2-ORIENTABILITY is NP-complete.

From Orientability to Schedulability

The following result concludes the proof of NP-completeness for the SCHEDULABILITY problem.

Proposition 4. $2\text{-ORIENTABILITY} \leq_P \text{SCHEDULABILITY}$

Given some graph $G = (V, E)$ as input for 2-ORIENTABILITY, we will explain the construction of a set T of transactions and will then argue that $G \in 2\text{-ORIENTABILITY}$ if, and only if, $(T, 3|E| + 3) \in \text{SCHEDULABILITY}$.

For the construction, we first fix an arbitrarily chosen order on the edges of E , which allows referring to them as $e_1, e_2, \dots, e_{|E|}$. We also associate to every edge e_i three different and unique objects x_i, y_i , and z_i . Then, the set T that we construct, will consist of precisely

$|V|$ transactions, such that there is one transaction for each node in V . We write T_v to denote the transaction associated to node $v \in V$. Every transaction T_v consists of precisely $3|E|$ operations, namely

$$W[\chi_1] \dots W[\chi_{|E|}] W[\gamma_1] \dots W[\gamma_{|E|}] W[\zeta_1] \dots W[\zeta_{|E|}]$$

The objects these operations interact with are chosen as follows: if $v \in e_i$, then χ_i , γ_i and ζ_i are the earlier defined objects x_i , y_i , and z_i , respectively. Otherwise, if $v \notin e_i$, then χ_i , γ_i and ζ_i are (all different) unique objects not occurring anywhere else in our construction.

We remark that this construction is indeed polynomial. To conclude the proof of Proposition 4, we make use of Proposition 1 and the next result. Some visual intuition for this lemma, in particular the construction of the longest paths in P_Σ , is given in Figure 3.1.

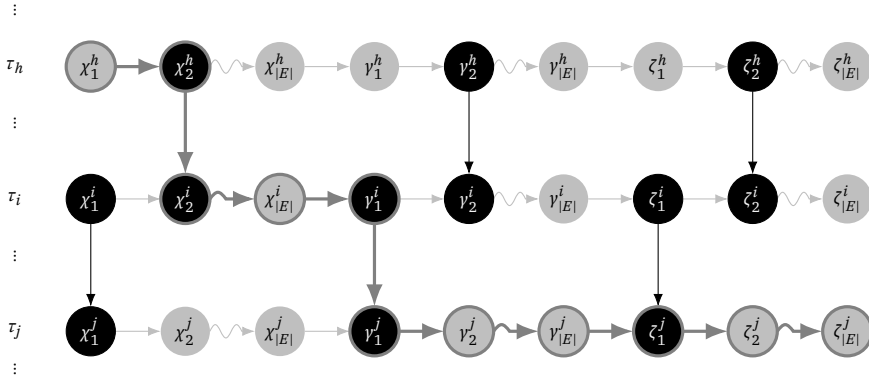


Figure 3.1 – A visualization of a longest path (marked in dark gray bold) in an arbitrary P_Σ , where the order Σ is defined as per Lemma 1. Each row represents a single transaction τ_h , τ_i or τ_j , sorted top-down consistent with $\leq_v$. Some cross-transaction edges are omitted for clarity of presentation. Waved lines denote one or more intermediate operations. We write χ_j^g , γ_j^g , and ζ_j^g to denote $W_g[\chi_j]$, $W_g[\gamma_j]$, and $W_g[\zeta_j]$, where g is the transaction identifier.

Lemma 1. For any total order $\leq_v$ on V , the longest path in $\text{OG}(G, \leq_v)$ counts at most two edges if and only if $\text{max-path}(P_\Sigma) + 1 \leq 3|E| + 3$, with $\Sigma = (T, \{(\tau_u, \tau_v) \mid u \leq_v v\})$.

Proof. Recall that the nodes of P_Σ coincide with the operations of transactions in T and that an edge (o, p) in P_Σ either means that p is the direct successor of o in some transaction, or that o and p are conflicting operations from different transactions. More precisely, in the latter case it follows from the construction of Σ that $o \in \text{ops}(\tau_v)$ and $p \in \text{ops}(\tau_u)$ with $v \leq_v u$ and with o the operation at position i in τ_v and p the operation at position i in τ_u for some integer i . We call the latter a cross transaction edge from τ_v to τ_u .

If direction The proof is by contraposition, *i.e.*, if $\text{max-path}(P_\Sigma) + 1 > 3|E| + 3$, then there is a path q in $\text{OG}(G, \leq_v)$ with three edges. For this, first recall that $\text{max-path}(P_\Sigma) + 1 \geq 3|E| + 3$ implies existence of a directed path q' in P_Σ with $3|E| + 3$ nodes, and hence with a length of $3|E| + 2$. Now, making use of the earlier developed intuition about P_Σ , we observe that every path q of length k contains at least $k - 3|E| - 1$ cross transaction edges (since every individual transaction is represented in P_Σ by precisely $3|E| - 1$ edges, and by the fact that cross transaction edges are always from transactions τ_v to τ_u with $v \leq_v u$). Since in our case: $k \geq 3|E| + 2$, it follows that there are at least three cross transaction edges. Hence, we can assume that there is an edge (o_1, o_2) , (p_1, p_2) and (r_1, r_2) with $o_1 \in \text{ops}(\tau_v)$, $o_2, p_1 \in \text{ops}(\tau_u)$, $p_2, r_1 \in \text{ops}(\tau_w)$ and $r_2 \in \text{ops}(\tau_s)$, which implies that a directed path $\langle (u, v), (v, w), (w, s) \rangle$ in G exists, which is clearly of length three.

Only-if direction This direction is by contraposition as well, *i.e.*, we show that if there exists a directed path in $\text{OG}(G, \leq_v)$ of length ≥ 3 then there exists a directed path in P_Σ of length $\geq 3|E| + 2$. For this, let $q = \langle (u, v), (v, w), (w, s) \rangle$ be such a path of precisely length three, with $e_{j_1} = \{u, v\}$, $e_{j_2} = \{v, w\}$, and $e_{j_3} = \{w, s\}$. Then, we construct a directed path q in P_Σ as follows: first traverse the consecutive operations of transaction τ_u up to (and including) its operation $W_u[x_{j_1}]$. Then traverse the cross transaction edge $(W_u[x_{j_1}], W_v[x_{j_1}])$, followed by the consecutive operations of transaction τ_v consecutive to $W_v[x_{j_1}]$ till (and including) operation $W_v[y_{j_2}]$. Then analog to before, traverse the cross transaction edge $(W_v[y_{j_2}], W_w[y_{j_2}])$ followed by consecutive operations of $W_w[y_{j_2}]$ in τ_w till (and including) $W_w[z_{j_3}]$ after which another cross transaction edge $(W_w[z_{j_3}], W_s[z_{j_3}])$ is traversed, followed by the remaining operations of τ_s consecutive

to $W_s[z_{j_3}]$. It can be verified that the constructed path has a length of precisely $3|E| + 2$, which concludes the proof. $\square$

3 Single Contention Point Scheduling

We just established that the general formulation of the transaction scheduling problem is an NP-hard optimization problem, meaning that we likely cannot beat brute force enumeration algorithms. It is natural to ask if there are restrictions we can place on the transaction sets to make the problem solvable more efficiently.

Contention Points

To this end, we will restrict classes of transactions based on the number of objects that are allowed to be present in conflict pairs. More formally, we define this as the set of contention points.

Definition 15. The *contention points* of a transaction set T are a set of objects $C(T) \subseteq O$, such that $a \in C(T)$ if, and only if there exist conflicting operations o and p on the object a in T .

Notice that the converse does not necessarily hold, that is, not every $a \in C(T)$ must appear in a conflicting operation pair. We will refer to an element of $C(T)$ as a *contention point*, and say that $\tau \in T$ *contends* if T contains an operation on a contention point.

Next, we introduce the set $\mathbb{T}_{\ell, m, n}$ of transaction sets with $T \in \mathbb{T}_{\ell, m, n}$ if $|\tau| \leq \ell$ for all $\tau \in T$, $|C(T)| \leq m$, and $|T| \leq n$. We will also use the set $\mathbb{T}_{\ell, 1, n}^+ \subseteq \mathbb{T}_{\ell, 1, n}$ where, for all $T \in \mathbb{T}_{\ell, 1, n}^+$, we have $T \in \mathbb{T}_{\ell, 1, n}$ such that τ contends for all $\tau \in T$.

Given a transaction set $T \in \mathbb{T}_{\ell, 1, n}^+$, we can see each transaction $\tau \in T$ as a finite sequence

$$\tau = \langle \underbrace{o_1, o_2, \dots, o_{r-1}}_{\text{prefix}}, W[a], \underbrace{o_{r+1}, \dots, o_{\ell-1}, o_{\ell}}_{\text{suffix}} \rangle,$$

where r is the position of $W[a]$ in this sequence, and where τ consists of a (potentially empty) subsequence of non-conflicting operations, followed by a conflicting operation to the contention point

a , and another (potentially empty) subsequence of non-conflicting operations. In the above transaction, we can also replace $W[a]$ by $R[a]$ as long as this does not remove the conflict from the transaction set. It does not affect the generality of the result we present next. We call the former subsequence the *prefix* of τ , and the latter the *suffix* of τ . We define $\text{pre}(\tau)$ and $\text{suf}(\tau)$ to be the number of operations before the first (respectively, the last) contention point.

Mapping Transactions Onto Square Grids

In what follows, we will show that the SCHEDULABILITY problem is decidable in polynomial time for transaction sets that are restricted to having a single contention point. More concretely, we prove the following result.

Theorem 6. For every $T \in \mathbb{T}_{\ell,1,n}$ and for every $k \in \mathbb{Z}_{\geq 0}$, it holds that SCHEDULABILITY $\in O(kn \log n)$.

We start by proving the result for the subset $\mathbb{T}_{\ell,1,n}^+ \subseteq \mathbb{T}_{\ell,1,n}$. That is, we assume that all transactions contend. We will relax this restriction to allow non-contending transactions later.

Intuitively, the algorithm works by trying to place the transactions into a specific square grid structure. If the transactions are able to fit within this grid then we accept, as we know that all possible longest paths will have a length bounded by k , given that $k+1$ is the grid's side length. See Figure 3.2 for an example of this grid.

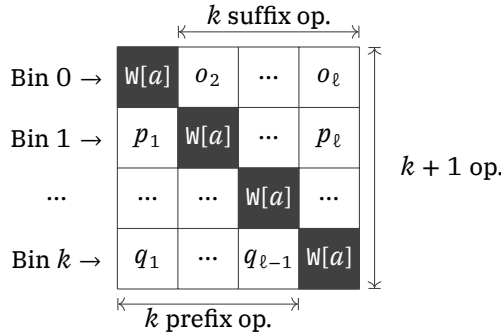


Figure 3.2 – Grid with contention point operations on the diagonal.

More formally, we will define injective functions called *allocations*

to model the placement of transaction operations within this grid. The definition is given below. Each transaction will be assigned to a bin, which corresponds to a row number of this grid (starting from 0). For each increasing bin number, the position of the contention point is moved from the first column to the last column. The rest of the grid positions are left open for the suffix and prefix operations.

Definition 16. Let $k \in \mathbb{Z}_{\geq 0}$. An allocation of $T \in \mathbb{T}_{\ell,1,n}^+$ to $[0, k]$ is an injective function α_k , which maps transactions $\tau \in T$ to $\alpha_k(\tau)$ such that $\text{pre}(\tau) \leq \alpha_k(\tau)$ and $\text{suf}(\tau) \leq k - \alpha_k(\tau)$. We call $\alpha_k(\tau)$ the *bin* of τ .

We will also denote the (partial) inverse of α_k by α_k^{inv} . Since α_k is injective, the inverse is only defined for the bins we actually assign transactions to. Notice that, $[0, k]$ means there are $k+1$ bins in total, as we start counting from 0.

For any allocation of a $T \in \mathbb{T}_{\ell,1,n}^+$ to $[0, k]$, we can use the allocation as a transaction order. If $\alpha_k(\tau_i) \leq \alpha_k(\tau_j)$ for $\tau_i, \tau_j \in T$, then we define an order $\text{ord}(\alpha_k) = (T, \leq_{\alpha_k})$ with $\tau_i \leq_{\alpha_k} \tau_j$. Conversely, we can define an allocation $\text{alloc}(\Sigma, k)$ to $[0, k]$ based on a total order Σ with $\text{span}(\text{cs}(\Sigma)) \leq k+1$. Let this total order be $\tau_1, \tau_2, \dots, \tau_n$. Then for each τ_i , with $i \in [n]$, we define $\text{alloc}(\Sigma, k)(\tau_i) = j$ such that $j \in [0, k]$ is the minimal bin with $\text{pre}(\tau_i) \leq j$, $\text{suf}(\tau_i) \leq k - j$, and $\text{alloc}(\Sigma, k)(\tau_h) < j$ for all $h < i$. We can now prove the following result.

Proposition 5. There exists an allocation α_k of $T \in \mathbb{T}_{\ell,1,n}^+$ to $[0, k]$, if, and only if, there exists a total order Σ for T such that $\text{span}(\text{cs}(\Sigma)) \leq k + 1$. Moreover, if α_k exists then $\Sigma = \text{ord}(\alpha_k)$. And, if Σ exists then $\alpha_k = \text{alloc}(\Sigma, k)$.

Proof. Let $T \in \mathbb{T}_{\ell,1,n}^+$. We first give the only-if direction and then provide the if direction.

Only-if direction Let α_k be an allocation of T to $[0, k]$. Furthermore, let p be a path in the precedence graph $P_{\text{ord}(\alpha_k)}$ which starts in the first operation of a transaction $\tau_i \in T$, and ends in the last operation in a transaction $\tau_j \in T$, hence the number of operations in p is $\text{pre}(\tau_i) + j - i + 1 + \text{suf}(\tau_j)$. We show that $\text{pre}(\tau_i) + j - i + 1 + \text{suf}(\tau_j) \leq k + 1$.

First, observe this bound on the number of operations in p :

$$\text{pre}(\tau_i) + j - i + 1 + \text{suf}(\tau_j) \leq \alpha_k(\tau_i) + j - i + 1 + k - \alpha_k(\tau_j).$$

This follows from $\text{pre}(\tau_i) \leq \alpha_k(\tau_i)$ and from $\text{suf}(\tau_j) \leq k - \alpha_k(\tau_j)$. Next, we have:

$$\alpha_k(\tau_i) + j - i + 1 + k - \alpha_k(\tau_j) \leq \alpha_k(\tau_i) + \alpha_k(\tau_j) - \alpha_k(\tau_i) + 1 + k - \alpha_k(\tau_j) = k + 1,$$

since $j - i \leq \alpha_k(\tau_j) - \alpha_k(\tau_i)$. This shows that every p has at most $k + 1$ operations, from which it follows that $\text{span}(\text{cs}(\text{ord}(\alpha_k))) \leq k + 1$ by Proposition 1.

If direction Let Σ be a total order of T such that $\text{span}(\text{cs}(\Sigma)) \leq k + 1$. Then, $f = \text{alloc}(\Sigma, k)$ is a valid allocation from T to $[0, k]$ by definition. First, it is clear that f is injective. Indeed, let $\{\tau, \tau'\} \subseteq T$ such that $\tau \neq \tau'$, then it is obvious that either $f(\tau) < f(\tau')$ or $f(\tau') < f(\tau)$ by definition, and therefore $f(\tau) \neq f(\tau')$. Additionally, for all $\tau \in T$ we have $\text{pre}(\tau) \leq f(\tau)$ and $\text{suf}(\tau) \leq k - f(\tau)$ by definition. This concludes the proof. $\square$

This characterization gives us a formal tool for constructing an optimal order, provided we have an algorithm that can create allocations. To that end, we define Algorithm 4.

Algorithm 4 $\text{allocate}(T \in \mathbb{T}_{\ell,1,n}^+, k \in \mathbb{Z}_{\geq 0}) \rightarrow \alpha_k \text{ or } \perp$

```

1: function suffix_lt( $\tau_i, \tau_j$ )
2:   if  $\text{suf}(\tau_i) \neq \text{suf}(\tau_j)$  then
3:     output  $\text{suf}(\tau_i) < \text{suf}(\tau_j)$ 
4:   else
5:     output  $\text{pre}(\tau_i) > \text{pre}(\tau_j)$ 
6: if  $T = \emptyset$  or  $k < \max\{|\text{ops}(\tau)| \mid \tau \in T\}$  then
7:   output  $\perp$ 
8: for all  $\tau \in \text{sorted\_by}(T, \text{suffix\_lt})$  do
9:   for all  $i \in [0, k + 1]$  do
10:    if  $\text{pre}(\tau) \leq i \wedge \text{suf}(\tau) \leq k - i \wedge \nexists \tau_h \in T : \alpha_k(\tau_h) = i$  then
11:       $\alpha_k(\tau) \leftarrow i$ 
12:    if  $\alpha_k(\tau)$  is undefined then
13:      output  $\perp$ 
14: output  $\alpha_k$ 

```

The above algorithm uses a function `sorted_by` which takes a set of transactions and a binary function f from the cartesian product of the transaction set with itself to true or false. The function f should return true if its first parameter is smaller than or equal to its second parameter, and false otherwise. This function is then used as a comparison function when sorting the set of transactions. This lets us abstract over the specific comparison-based sorting algorithm being used.

Algorithm 4 is sound by construction. If it does not terminate with $\perp$ for the input transaction set T and threshold k , it terminates with a valid allocation of T to $[0, k]$. Showing completeness, that is, termination with $\perp$ implies there exists no valid allocation from T to $[0, k]$, requires slightly more argumentation. The following lemma is a useful tool.

Lemma 2. Let $T \in \mathbb{T}_{\ell,1,n}^+$. For every $T' \subseteq T$, and every allocation of T on $[0, k]$, we have that $|T'| + i + j - 1 \leq k$.

Proof. Consider any subset $T' \subseteq T$ with i and j respectively the minimum prefix and minimum suffix length of transactions in T' . By definition of allocations for T to $[0, k]$, it follows for every transaction $\tau \in T'$ that $i \leq \text{pre}(\tau) \leq \alpha_k(\tau)$. Moreover, since T' contains $|T'|$ many different such transactions, we know that for at least one of these transactions, say τ' , we have $i + |T'| - 1 \leq \alpha_k(\tau')$. In addition, we also know that $j \leq \text{suf}(\tau') \leq k - \alpha_k(\tau')$, thus implying $\alpha_k(\tau') \leq k - j$. Combining the two inequalities gives us $i + |T'| - 1 \leq k - j$, which is equivalent to the desired inequality $|T'| + i + j - 1 \leq k$. $\square$

Completeness follows from assuming a smaller allocation exists despite the algorithm yielding $\perp$, and then showing it contradicts Lemma 2. Proposition 5 ensures no order exists either. We remark that k is at most $\sum_{\tau \in T} |\text{ops}(\tau)|$, hence k depends entirely on the transaction set, and its time complexity is polynomial.

Proposition 6. For any $T \in \mathbb{T}_{\ell,1,n}^+$ and $k \in \mathbb{Z}_{\geq 0}$, Algorithm 4 runs in $O(k|T| \log |T|)$ time and returns an allocation of T to $[0, k]$ if it exists and $\perp$ otherwise.

Proof. It is straightforward to see that if Algorithm 4 does not terminate with $\perp$, then its output is an allocation. It is less obvious that if the algorithm returns $\perp$, then there is no α_k for T . If for all $\tau \in T$ there is no $i \leq k$ with $\text{pre}(\tau) \leq i$ and $\text{suf}(\tau) \leq k - i$, then no allocation of T on $k + 1$ can exist. Otherwise (if there is at least one such $i \leq k$) all appropriate positions have already been taken by transactions considered in earlier iterations of the algorithm. Now let j be the last position in which transaction τ would hypothetically fit (thus with $\text{pre}(\tau) \leq j$ and $\text{suf}(\tau) \leq k - j$).

Let h be the minimum position preceding j such that all positions between (and including) h and j are taken by transactions. We remark that all positions in which τ can hypothetically fit are part of this sequence, since at least all positions between $\text{pre}(\tau)$ and j are taken, but h may be smaller than $\text{pre}(\tau)$. Let $T' \subseteq T$ be a transaction set occupying the spots between h and j inclusive. Now, notice that the minimum prefix length of all transactions in $T' \cup \{\tau\}$ is precisely h . Indeed, this observation is trivial in case $h = 0$ and otherwise follows directly from the assignment strategy of our algorithm and the fact that position $h - 1 \geq 0$ is empty. Also notice that the minimum suffix length of all transactions in $T' \cup \{\tau\}$ is precisely $\text{suf}(\tau) = k - j$.

We obtain a contradiction against Lemma 2. Assume there exists an allocation of T on $[0, k]$, then by Lemma 2 it must be the case that $|T' \cup \{\tau\}| + (k - j) + h - 1 \leq k$. However, we can simplify the left-hand side of that inequality to $j - h + 2 + k - j + h - 1$, which simplifies further to $k + 1$. Hence we get $k + 1 \leq k$ or $1 \leq 0$ which is a contradiction. Therefore, there is no allocation from T on $[0, k]$.

For the time complexity of this algorithm, observe that line (8) involves sorting all transactions, which can be done in $|T| \log |T|$ time, in the worst case. Lastly, line (9) involves at most $k + 1$ or $O(k)$ steps. Hence, the overall time complexity is $O(k|T| \log |T|)$. $\square$

Until now, we have considered transactions from $\mathbb{T}_{\ell,1,n}^+$, but this restriction can be easily relaxed to any $T \in \mathbb{T}_{\ell,1,n}$ by applying Algorithm 4 to the subset of transactions $T' \subseteq T$ which contend. To obtain an ordering for the full T we can append the remaining transactions ($T \setminus T'$) arbitrarily onto the front or back of the allocation order. Theorem 6 then follows as a corollary of Proposition 5 and Proposition 6.

4 Summary

In this chapter, we presented the following main results.

- In Section 3.2, we proved that the decision variant of the transaction scheduling problem is NP-complete, meaning that the optimization problem can be considered an NP-hard optimization problem.
 - We first introduced the 3-COLORABILITY problem, which was proven NP-complete by Stockmeyer [84]. We then defined the 2-ORIENTABILITY problem which is closely related to it via the Gallai-Hasse-Roy-Vitaver theorem [23].
 - We then defined a reduction from the 2-ORIENTABILITY problem to the SCHEDULABILITY problem, which is a decision variant of the transaction scheduling optimization problem we introduced in the previous chapter.
- Next, in Section 3.3, we introduced a restricted variant of the problem which does allow the SCHEDULABILITY problem to be solved in polynomial time.
 - We introduced a restriction on transaction sets in terms of the number of allowed conflicting objects, we refer to these as the contention points of a transaction set.
 - For transaction sets T with a single contention point and a threshold $k \in \mathbb{Z}_{\geq 0}$, we then introduced an algorithm that can decide SCHEDULABILITY in $\mathcal{O}(k|T| \log |T|)$.

Further Reading

The material in this section originates from the ICDT 2026 paper by Brecht Vandevoort, Bas Ketsman, and myself titled “*Bounding the Makespan of Transaction Schedules*” [14].

The background material on the time complexity of decision problems and asymptotic analysis is based on the work of Sipser titled: “*Introduction to the Theory of Computation*” [81], and the book: “*Algorithmics for Hard Problems*” by Hromkovič was consulted for the parts about the time complexity of optimization problems.

Approximation Algorithms

Until now, we have presented algorithms that always give an optimal conflict serializable schedule for the input transaction set. However, we could also try to relax the optimality guarantee, and this gives us an approximation algorithm. Informally, such an algorithm runs in polynomial time, and provides us with a solution of a cost that is within a bounded factor of the optimal cost. Notice that this is a stronger guarantee than that provided by heuristic algorithms, which do not provide any such guarantees on the quality of the output.

1 Preliminaries

Given that our formalization of transaction scheduling is a minimization problem, we will only define approximation algorithms for minimization problems. The definition for maximization problems is similar, however.

Definition 17 (Hromkovič [46]). Let MIN be a minimization problem with the set of inputs I . An α -approximation algorithm A_{MIN} for MIN is a polynomial time algorithm which, for all inputs $x \in I$, gives an output $A_{\text{MIN}}(x)$ such that

$$\frac{c(A_{\text{MIN}}(x))}{\text{opt}(x)} \leq \alpha,$$

where $c(A_{\text{MIN}}(x))$ is the cost of output $A_{\text{MIN}}(x)$, $\text{opt}(x)$ is the optimal cost for input x , and where $\alpha \geq 1$ is more often referred to as the *approximation ratio* of A_{MIN} .

We also say that the approximation algorithm A_{MIN} is *tight* if we can construct an instance $x \in I$ for which $c(A_{\text{MIN}}(x)) = \alpha \cdot \text{opt}(x)$.

We now introduce some extra notation for working with the position of contention points in transactions. Since our definition of transactions allows at most one read and one write operation on the same object we cannot uniquely identify an operation based on the object it acts on. *For the purposes of this chapter, we will assume that there is only a single conflicting operation per contention point in each transaction.* Let $\mathbf{x} = \langle x_1, x_2, \dots, x_k \rangle$ be a finite sequence of distinct objects. We will write $\mathbf{x}_{(i)}$ to denote the i -th object in $\mathbf{x}$. For a transaction τ in a transaction set T , we will let $\text{seq}(\tau)$ represent the sequence of contention points ordered in the way τ accesses them, and write $\text{seq}(\tau, i)$ to refer to $\mathbf{x}_{(i)}$ given that $\mathbf{x} = \text{seq}(\tau)$. We write $\text{cpos}(\tau, i) \in [|\text{ops}(\tau)|]$ to refer to the position of the operation on the contention point $\text{seq}(\tau, i)$ in some transaction τ . Furthermore, we define $T_{(\mathbf{x})}$ as $\{\tau \mid \tau \in T, \text{seq}(\tau) = \mathbf{x}\}$.

By convention, we will always refer to the contention points in lexicographic order. Hence, if we have m contention points, then we refer to the first one as a , the second one as b , and so on. For example, in a set $T \in \mathbb{T}_{\ell,3,n}$ the contention points $C(T)$ are always a , b , and c . We will say that a transaction $\tau \in \mathbb{T}_{\ell,m,n}$ is k -contending if τ accesses $k \leq m$ contention points in its operations, irrespective of the order in which they are accessed. For instance, if a transaction τ is 2-contending, then it can either access a and then b , or b and then a . Recall from the previous chapter that the notation $\mathbb{T}_{\ell,m,n}^+$ is defined as $\mathbb{T}_{\ell,m,n}^+ \subseteq \mathbb{T}_{\ell,m,n}$ where for all $T \in \mathbb{T}_{\ell,m,n}^+$ it is the case that every transaction is m -contending.

The Upper Bound Lemma

In what follows, we will provide algorithms for transaction sets in $\mathbb{T}_{\ell,m,n}$, and we want to prove upper bounds on the makespan of the schedules produced by these algorithms. In the interest of reducing the number of cases in these proofs, we will first identify which kinds of transaction sets behave the worst in this sense.

This next lemma shows that we should consider $\mathbb{T}_{\ell,m,n}^+ \subseteq \mathbb{T}_{\ell,m,n}$ (the set of all transaction sets where all transactions are m -contending, defined in the previous chapter) such that $|T| = n$ and such that

all $\tau \in T$ are of length ℓ . If one of the transactions has fewer operations on contention points, or a transaction is missing, or one of the transactions is shorter, then the upper bound will be lower than what we could encounter otherwise.

Lemma 3. Let $T^+ \in \mathbb{T}_{\ell, m, n}^+$ such that $|T^+| = n$ and $|\text{ops}(\tau)| = \ell$ for all $\tau \in T^+$. If $T = (T^+ \setminus \{\tau_{old}\}) \cup T_{new}$ with $\tau_{old} \in T^+$ such that:

1. $T_{new} = \{\tau_{new}\}$ where τ_{new} is k -contending and $k < m$; or
2. $T_{new} = \emptyset$; or
3. $T_{new} = \{\tau_{new}\}$ with $|\text{ops}(\tau_{new})| < \ell$;

then, for any total order Σ^+ on T^+ , it holds that:

$$\text{span}(\text{cs}(\Sigma)) \leq \text{span}(\text{cs}(\Sigma^+)),$$

where Σ is the total order Σ^+ with τ_{old} replaced by τ_{new} . In case (2), we leave out τ_{new} and concatenate any gaps present in Σ in order to obtain a total order.

Proof. Let T^+ and T be the sets from the lemma statement, and let Σ^+ and Σ be total orders on T^+ and T respectively, also defined in the lemma statement. We now continue based on three cases, depending on the definition of T_{new} .

Case (1): τ_{new} is k -contending with $k < m$. The difference between P_{Σ^+} and P_{Σ} is that $k - m$ of the operations on contention points present in τ_{old} are no longer conflicting operations in τ_{new} . Any longest path q^+ chosen in P_{Σ^+} is also a longest path q in P_{Σ} unless q^+ passes through operations of τ_{old} . Depending on the operations that are missing, we could either have that q has the same length as q^+ if the operations on contention points have disappeared from the middle of τ_{old} , or they can be removed from the front or back, in which case the maximal reduction in length is $k - m$ operations. Hence, to conclude, we get that:

$$\text{span}(\text{cs}(\Sigma^+)) - (k - m) \leq \text{span}(\text{cs}(\Sigma)) \leq \text{span}(\text{cs}(\Sigma^+)).$$

From this, we can conclude that $\text{span}(\text{cs}(\Sigma)) \leq \text{span}(\text{cs}(\Sigma^+))$.

Case (2): $T_{new} = \emptyset$. The only difference between P_{Σ^+} and P_{Σ} is that the operations of τ_{old} are missing from P_{Σ} . Hence, any longest path

q^+ that we pick in P_{Σ^+} would also be a longest path q in P_{Σ} *unless* q^+ passes through operations of τ_{old} . The fewest number of operations q^+ can traverse through that no q can traverse through is a single operation on a contention point, and the greatest number of operations is ℓ , as transaction length is bounded by ℓ . Hence, if q and q^+ are identical then $\text{span}(\text{cs}(\Sigma)) = \text{span}(\text{cs}(\Sigma^+))$, and otherwise:

$$\text{span}(\text{cs}(\Sigma^+)) - \ell \leq \text{span}(\text{cs}(\Sigma)) \leq \text{span}(\text{cs}(\Sigma^+)) - 1.$$

From this, we can conclude that $\text{span}(\text{cs}(\Sigma)) \leq \text{span}(\text{cs}(\Sigma^+))$.

Case (3): τ_{new} with $|\text{ops}(\tau_{new})| < \ell$. Here, the only difference between P_{Σ^+} and P_{Σ} is that there are fewer operations in τ_{new} than there are in τ_{old} . Any longest path q^+ in P_{Σ^+} would also be a longest path q in P_{Σ} *unless* q^+ traverses through operations of τ_{old} . In that case, there exists a similar path in q that traverses through τ_{new} . Given that τ_{new} contains strictly fewer than ℓ operations, this path will be at least 1 and at most $|\text{ops}(\tau_{new})|$ operations shorter. That is, if q and q^+ are identical then $\text{span}(\text{cs}(\Sigma)) = \text{span}(\text{cs}(\Sigma^+))$, and otherwise:

$$\text{span}(\text{cs}(\Sigma^+)) - |\text{ops}(\tau_{new})| \leq \text{span}(\text{cs}(\Sigma)) \leq \text{span}(\text{cs}(\Sigma^+)) - 1$$

Thus, we can similarly conclude: $\text{span}(\text{cs}(\Sigma)) \leq \text{span}(\text{cs}(\Sigma^+))$. □

2 Two Contention Point Scheduling

In this section, we are concerned with the following problem. Note that, in the remainder of this chapter, we will always assume that $\ell \geq m$ for all $\mathbb{T}_{\ell, m, n}$, as this ensures all transactions are long enough to potentially have an operation on each contention point.

2-TxSCHED

Input: A transaction set $T \in \mathbb{T}_{\ell, 2, n}$ with $\ell \geq 2$.

Output: A conflict serializable schedule s for T .

Goal: Minimize $\text{span}(s)$.

To schedule transaction sets with two contention points, we intuitively start by dividing the transaction sets into disjoint subsets of 2-contending, 1-contending and 0-contending transactions. We then schedule each subset separately, and merge them together in a way that does not increase the makespan. Hence, to find an overall upper bound, we only need to consider the 2-contending case.

Algorithm

We refer to a *no-op transaction* as a transaction consisting of a single *no-op*, which is defined as a read operation $R[\perp]$ on a (special) bottom object $\perp \in O$. Intuitively, this operation does nothing. We will use these transactions as placeholders when constructing sequences. Furthermore, we will write $|\mathbf{x}|$ to denote the length of the sequence $\mathbf{x}$. We use the functions `append` and `sorted_by` (defined in Section 3.3). The function `append` concatenates sequences, *i.e.*, if $\mathbf{x}$ and $\mathbf{y}$ are sequences then it produces a sequence $\mathbf{z}$ s.t. $|\mathbf{z}| = |\mathbf{x}| + |\mathbf{y}|$ and with $\mathbf{x}_{(i)} = \mathbf{z}_{(i)}$ for all $i \in [|\mathbf{x}|]$, and $\mathbf{y}_{(j)} = \mathbf{z}_{(|\mathbf{x}|+j)}$ for all $j \in [|\mathbf{y}|]$.

First, we introduce some helper functions. We define the functions `sort_dilation` (Alg. 5) and `sort_suffix` (Alg. 6). Both functions take a set of transactions $T \in \mathbb{T}_{\ell, m, n}$ and a flag which states to either sort in ‘ascending’ order (denoted *gte*, for *greater-than or equal*) or in ‘descending’ order (denoted *lte*, for *less-than or equal*). The function inputs differ in the sense that the former takes $T \in \mathbb{T}_{\ell, m, n}$ where $m \geq 2$ while the latter requires that $T \in \mathbb{T}_{\ell, 1, n}$.

The `sort_dilation` function sorts transactions based on the number of operations between their first two contention points when sorting in *gte* order, and it sorts based on the distance between the last two contention points when sorting in *lte* order. We sometimes refer to these distances as the ‘dilation’ of the transaction.

The algorithm `sort_suffix` is similar to `sort_dilation`, except it operates on transaction sets with a single contention point and sorts by way of the suffix rather than the dilation between two contention points.

The helper function `merge` takes two sequences and the index of a conflict point, and merges the two sequences on that conflict point index. The sequences are here referred to as ‘runs’, as we assume

Algorithm 5 $\text{sort_dilation}(T : \mathbb{T}_{\ell, (m \geq 2), n}, d : \text{gte/lte}) \rightarrow \text{tx. seq.}$

```

1: function dilation_gte( $\tau, \tau'$ )
2:   output  $\text{cpos}(\tau, 2) - \text{cpos}(\tau, 1) \geq \text{cpos}(\tau', 2) - \text{cpos}(\tau', 1)$ 
3: function dilation_lte( $\tau, \tau'$ )
4:   output  $\text{cpos}(\tau, m) - \text{cpos}(\tau, m-1) \leq \text{cpos}(\tau', m) - \text{cpos}(\tau', m-1)$ 
5:  $c\_fn \leftarrow$  if  $d = \text{gte}$  then dilation_gte else dilation_lte
6: output sorted_by( $T, c\_fn$ )

```

Algorithm 6 $\text{sort_suffix}(T : \mathbb{T}_{\ell, 1, n}, d : \text{gte/lte}) \rightarrow \text{tx. seq.}$

```

1: function suffix_gte( $\tau, \tau'$ )
2:   output  $\text{suf}(\tau) \geq \text{suf}(\tau')$ 
3: function suffix_lte( $\tau, \tau'$ )
4:   output  $\text{suf}(\tau') \leq \text{suf}(\tau)$ 
5:  $c\_fn \leftarrow$  if  $d = \text{gte}$  then suffix_gte else suffix_lte
6: output sorted_by( $T, c\_fn$ )

```

they have been sorted by some other function before. This is a similar primitive to the merge being utilized in merge sort.

Algorithm 7 $\text{merge}(\text{runL}, \text{runR} : \text{tx. seq.}; p : \text{confl. index}) \rightarrow \text{tx. seq.}$

```

1:  $\text{run} \leftarrow$  a sequence of  $|\text{runL}| + |\text{runR}|$  no-op transactions
2:  $i \leftarrow 1; j \leftarrow 1$ 
3: for all  $k \in [|\text{runL}| + |\text{runR}|]$  do
4:   if  $\text{cpos}(\text{runL}_{(i)}, p) \leq \text{cpos}(\text{runR}_{(j)}, p)$  then
5:      $\text{run}_{(k)} \leftarrow \text{runL}_{(i)}; i \leftarrow i + 1$ 
6:   else
7:      $\text{run}_{(k)} \leftarrow \text{runR}_{(j)}; j \leftarrow j + 1$ 

```

Finally, we can define the scheduling algorithm AS_2 (see Algorithm 8 on the next page). Since we return a canonical schedule, this means it is also conflict serializable. In the algorithm we convert sequences into orders, this is defined next. Let $\mathbf{x}$ be a sequence of transactions, then we define a total order $\text{ord}(\mathbf{x}) = (\bigcup_{i \in |\mathbf{x}|} \mathbf{x}_{(i)}, \leq_{\mathbf{x}})$ such that $\mathbf{x}_{(i)} \leq \mathbf{x}_{(j)}$ iff $i \leq j$.

Algorithm 8 $\text{AS}_2(T : \text{set in } \mathbb{T}_{(\ell \geq 2), 2, n}) \rightarrow \text{schedule for } T$

- 1: $\text{runAB} \leftarrow \text{sort_dilation}(T_{\langle a, b \rangle}, \text{gte})$
 - 2: $\text{runA} \leftarrow \text{sort_suffix}(T_{\langle a \rangle}, \text{gte})$
 - 3: $\text{runAB}' \leftarrow \text{merge}(\text{runAB}, \text{runA}, 1)$
 - 4: $\text{runBA} \leftarrow \text{sort_dilation}(T_{\langle b, a \rangle}, \text{lte})$
 - 5: $\text{runB} \leftarrow \text{sort_suffix}(T_{\langle b \rangle}, \text{lte})$
 - 6: $\text{runBA}' \leftarrow \text{merge}(\text{runBA}, \text{runB}, 1)$
 - 7: $\text{runNC} \leftarrow$ any sequence on all non-conflicting transactions in T
 - 8: **output** canonical(ord(append(runAB' , runBA' , runNC)))
-

Approximation Ratio

The algorithm AS_2 runs in time quadratic in the number of operations, and quasilinear in the number of transactions, and yields a schedule with a makespan of at most $2\ell + n - 2$. We prove this next.

Proposition 7. For all $T \in \mathbb{T}_{\ell, 2, n}$ with $\ell \geq 2$ it is the case that:

1. $\text{span}(\text{AS}_2(T)) \leq 2\ell + n - 2$, and
2. $\text{AS}_2(T)$ runs in $\mathcal{O}(p^2 + n \log n)$ where $p = \sum_{\tau \in T} |\text{ops}(\tau)|$.

Proof. To show (1), let all $\tau \in T$ be 2-contending, let $|T| = n$, and let all transactions be length ℓ . We are allowed to do this as assuming otherwise would result in a lower-than-or-equal makespan by Lemma 3.

Let Σ be an order on T such that $\text{cs}(\Sigma) = \text{AS}_2(T)$. This order lets us refer to the elements of T by $\tau_1, \tau_2, \dots, \tau_n$. We will let q refer to the longest path in P_Σ and let o and p be the first (respectively, last) operation in q such that $o \in \text{ops}(\tau_i)$ and $p \in \text{ops}(\tau_j)$ for some integers $i, j \in [n]$. We now compute bounds on the number of operations on the path q . The reason for not using the (edge) path length is that the number of nodes (i.e. number of operations) is anyway equal to the makespan (as this is the path length increased by one), and this makes the proofs slightly easier to follow.

We continue with a proof by cases. By construction of Algorithm 8, the following cases can occur:

- C1) $\text{seq}(\tau_i) = \langle a, b \rangle$ and $\text{seq}(\tau_j) = \langle a, b \rangle$,
- C2) $\text{seq}(\tau_i) = \langle b, a \rangle$ and $\text{seq}(\tau_j) = \langle b, a \rangle$, and

C3) $\text{seq}(\tau_i) = \langle a, b \rangle$ and $\text{seq}(\tau_j) = \langle b, a \rangle$.

In cases (C1) and (C2), Algorithm 8 constructs a path starting in τ_i , transitioning through $(j-i)$ transactions on either contention point a (in case C1) or b (in case C2) and ends in transaction τ_j , this gives the following bound on the number of operations in q :

$$\text{pre}(\tau_i) + 1 + (j - i) + (\text{cpos}(\tau_j, 2) - \text{cpos}(\tau_j, 1)) + 1 + \text{suf}(\tau_j) \leq \quad (4.1)$$

$$(\ell - 1) + 1 + (j - i) + (\ell - 1) \leq \quad (4.2)$$

$$2\ell - 1 + (j - i) \leq \quad (4.3)$$

$$2\ell + n - 2$$

The inequality (4.1) follows from the fact that $\text{pre}(\tau_i) \leq \ell - 1$ and $(\text{cpos}(\tau_j, 2) - \text{cpos}(\tau_j, 1)) + 1 + \text{suf}(\tau_j) \leq \ell - 1$. Then, inequality (4.2) follows from simplification (namely $2(\ell - 1) + 1 = 2\ell - 1$). Finally, inequality (4.3) follows from the worst case assignment of $i = 1$ and $j = n$, which then simplifies the expression to $2\ell - 1 + (n - 1)$ or $2\ell + n - 2$.

For case (C3), Algorithm 8 constructs a path starting in τ_i , which transitions through $(j - i)$ other transactions on contention point b , to ultimately end up in τ_j . This gives the following bound on the number of operations in q :

$$\text{pre}(\tau_i) + 1 + (\text{cpos}(\tau_i, 2) - \text{cpos}(\tau_i, 1)) + (j - i) + 1 + \text{suf}(\tau_j) \leq \quad (4.4)$$

$$(\ell - 1) + (j - i) + 1 + (\ell - 1) \leq \quad (4.5)$$

$$2\ell - 1 + (j - i) \leq \quad (4.6)$$

$$2\ell + n - 2$$

The inequality (4.4) follows from the fact that $\text{suf}(\tau_j) \leq \ell - 1$, and the fact that $\text{pre}(\tau_i) + 1 + (\text{cpos}(\tau_i, 2) - \text{cpos}(\tau_i, 1)) \leq \ell - 1$. The remainder of the argument for (4.5) and (4.6) is identical to the previous two cases. In all cases, $\max\text{-path}(P_\Sigma) + 1 \leq 2\ell + n - 2$, hence by Corollary 3 we can conclude that $\text{span}(\text{canonical}(\Sigma)) \leq 2\ell + n - 2$.

For showing (2), it is clear to see that lines 1, 2, 4, and 5 of Algorithm 8 involve sorting which can be done in $\mathcal{O}(n \log n)$ time; then lines 3 and 6 involve a merge which can be done in $\mathcal{O}(n)$ time. And finally, concatenating the runs and building the canonical schedule (line 8) can be done in $\mathcal{O}(p^2 + n)$ time where $p = \sum_{\tau \in T} |\text{ops}(\tau)|$. This gives an overall bound of $\mathcal{O}(p^2 + n \log n)$. $\square$

The above proposition gives us the guarantee that the makespan will not exceed $2\ell + n - 2$ whenever we use the algorithm A_2 for some transaction in $\mathbb{T}_{\ell,2,m}$. Next, we show that there are transaction sets in $\mathbb{T}_{\ell,2,m}$ that *do* actually have this as an optimal makespan. In this sense, such a set serves as a lower bound.

Proposition 8. There exists a transaction set $T \in \mathbb{T}_{\ell,2,n}$ with $n \geq 2$, such that $\text{opt}(T) = 2\ell + n - 2$.

Proof. Let T be a transaction set such that $T = T_{\langle a,b \rangle} \cup T_{\langle b,a \rangle}$ and, if n is odd then $|T_{\langle a,b \rangle}| = \lfloor n/2 \rfloor$ and $|T_{\langle b,a \rangle}| = \lfloor n/2 \rfloor + 1$, otherwise $|T_{\langle a,b \rangle}| = n/2$ and $|T_{\langle b,a \rangle}| = n/2$.

All transactions in the subset $T_{\langle a,b \rangle}$ are of the following form:

$$\tau_{a,b} = \langle W[a], R[u_1], R[u_2], \dots, R[u_{\ell-2}], W[b] \rangle,$$

and all transactions in the subset $T_{\langle b,a \rangle}$ are of the following form:

$$\tau_{b,a} = \langle W[b], R[u_1], R[u_2], \dots, R[u_{\ell-2}], W[a] \rangle.$$

Let $\Sigma = (T, \leq_T)$ be a total order on T , so we can refer to T 's elements by $\tau_1, \tau_2, \dots, \tau_n$. Because the directed edges in P_Σ must respect Σ for conflicting operations, there exists a path $\langle W_1[a], W_2[a], \dots, W_n[a] \rangle$ and a path $\langle W_1[b], W_2[b], \dots, W_n[b] \rangle$ which are both n operations long. Furthermore, there exists a path $\langle W_j[a], \dots, W_j[b], W_{j+1}[b], \dots, W_{j+1}[a] \rangle$ in P_Σ , where $j = |T_{\langle a,b \rangle}|$, which contains 2ℓ operations. If we connect these paths as per Figure 4.1, we see a grid-like pattern with 2ℓ columns and n rows which can together be combined to get a minimally required makespan of $2\ell + n - 2$. Indeed, we count $|T_{\langle a,b \rangle}|$ operations of type $W_i[a]$, followed by the transition from $T_{\langle a,b \rangle}$ to $T_{\langle b,a \rangle}$ which takes 2ℓ operations, followed by $|T_{\langle b,a \rangle}|$ operations of type $W_i[b]$. This concludes the proof. $\square$

The following theorem is a corollary of Propositions 7 and 8. While we prove an exact approximation ratio here, we can refer to A_2 as a $\mathcal{O}(n/\ell)$ -approximation algorithm, which is slightly easier to remember and talk about.

Theorem 7. AS_2 is a tight $\left(\frac{n-2}{\ell} + 2\right)$ -approximation algorithm for 2-TXSCHED.

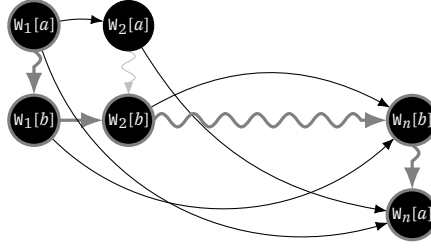


Figure 4.1 – A longest path (marked in dark gray bold) in a precedence graph P_Σ for any total order Σ on T . Waved lines denote one or more intermediate operations.

Proof. The trivial lower bound on the makespan of any transaction set in $\mathbb{T}_{\ell,2,n}$ is ℓ , and Proposition 7 gives us an upper bound on the makespan of $2\ell + n - 2$ in polynomial time. Therefore, the approximation ratio can be derived as follows:

$$\frac{2\ell + n - 2}{\ell} = \frac{2\ell}{\ell} + \frac{n - 2}{\ell} = \frac{n - 2}{\ell} + 2.$$

Furthermore, by Proposition 8 we know that there exist transaction sets T in $\mathbb{T}_{\ell,2,n}$ with $n \geq 2$ such that $\text{opt}(T) = 2\ell + n - 2$, showing that A_2 is a tight approximation in case $n \geq 2$. For the case that $n = 1$ the makespan outputted by A_2 is trivially ℓ . Hence, A_2 is a tight $((n - 2)/\ell + 2)$ -approximation algorithm, concluding the proof. $\square$

3 Three Contention Point Scheduling

We now increase the amount of contention points to three. As we will see, the main thing that changes is that lower bounds become harder to find. While we obtained a tight approximation in the previous section, this will no longer be the case here.

$(3, n)$ -TxSCHED

Input:	A transaction set $T \in \mathbb{T}_{\ell,3,n}$ with $\ell \geq 3$.
Output:	A conflict serializable schedule s for T .
Goal:	Minimize $\text{span}(s)$.

The algorithm for three contention points still relies on the same building blocks introduced in the previous section. What changed is the amount of permutations that operations on contention points can be in within a single transaction. Given that we have 3 contention points, there are now $3! = 6$ permutations to handle instead of only $\langle a, b \rangle$ and $\langle b, a \rangle$. Hence, we now need to think about the most efficient way to combine groups of transactions that operate on the contention points in the same order.

Maximal Total Overlap

To briefly illustrate this problem, in case we have a transaction set $T \in \mathbb{T}_{\ell,3,n}$ such that $n \geq 6$, then we can divide T into 6 subsets, one for each order on contention points. In this case, that is:

$$T = T_{\langle a,b,c \rangle} \cup T_{\langle b,c,a \rangle} \cup T_{\langle c,b,a \rangle} \cup T_{\langle b,a,c \rangle} \cup T_{\langle a,c,b \rangle} \cup T_{\langle c,a,b \rangle}.$$

If we schedule transactions such that transactions from $T_{\langle a,b,c \rangle}$ occur before those in $T_{\langle c,a,b \rangle}$ then any longest path going through these sets will move from $\tau_i \in T_{\langle a,b,c \rangle}$ to $\tau_j \in T_{\langle c,a,b \rangle}$ on operations working with the contention point c . Hence, transactions from $T_{\langle a,b,c \rangle}$ can not overlap with transactions in $T_{\langle c,a,b \rangle}$, increasing overall path length. However, if we scheduled them such that $T_{\langle c,a,b \rangle}$ is before $T_{\langle a,b,c \rangle}$, then the path would transfer from τ_j to τ_i through an operation on the contention point a . This would allow for an overlap of one contention point (*i.e.*, operations on a from $T_{\langle a,b,c \rangle}$ can execute at the same time as operations on b from $T_{\langle c,a,b \rangle}$ are executing).

To make this more precise, we first define the concept of “maximal overlap between transactions”. Intuitively, this refers to the maximal number of conflicting operations that two transactions can execute concurrently without breaking the rules of conflict serializability. Since we will make abstraction of operations, and rather

focus on contention point sequences, we will more formally define this as follows.

Definition 18. Let $\mathbf{x}$ and $\mathbf{y}$ be two length m sequences of distinct objects. We define the *maximal overlap between $\mathbf{x}$ and $\mathbf{y}$* , denoted $\text{ov}(\mathbf{x}, \mathbf{y})$, as the maximal number $k \in [1, m - 1]$ such that for all $j \in [k]$ and for all $i \in [j + m - k, m]$ we have $\mathbf{x}_{(i)} \neq \mathbf{y}_{(j)}$; if no such k exists, then we let $\text{ov}(\mathbf{x}, \mathbf{y}) = 0$.

Given a totally ordered finite set of same-length distinct object sequences $(K, \leq_K)$, its *total overlap* $\text{ov}^+(K, \leq_K)$ is defined as the sum $\sum_{i \in [|K| - 1]} \text{ov}(\mathbf{x}_i, \mathbf{x}_{i+1})$, where all $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{|K|}$ are ordered as per $\leq_K$. Note that the prior subscripts are not positions in $\mathbf{x}$, but identifiers for distinct object sequences (the former are written $\mathbf{x}_{(i)}$, not $\mathbf{x}_i$). Lastly, the *maximal total overlap* for K is

$$\text{ov}_{\max}^+(K) = \max\{\text{ov}^+(K, \leq_K) \mid \text{with } (K, \leq_K) \text{ a total order on } K\}.$$

Furthermore, for any two disjoint ordered sets $(T_1, \leq_1)$ and $(T_2, \leq_2)$, we define the *concatenation* of T_1 and T_2 as $T_1 \parallel T_2 = (T_1 \cup T_2, \leq_{\text{cat}})$ where $\tau_i \leq_{\text{cat}} \tau_j$ is either $\tau_i \leq_1 \tau_j$ with $\{\tau_i, \tau_j\} \subseteq T_1$, or $\tau_i \leq_2 \tau_j$ with $\{\tau_i, \tau_j\} \subseteq T_2$, or $\tau_i \in T_1$ and $\tau_j \in T_2$. Notice that this operation is associative, so we will write $T_1 \parallel T_2 \parallel T_3$ for $(T_1 \parallel T_2) \parallel T_3$.

Now, for the algorithm to work, we always need to be able to find a sequence of contention points which has a maximal total overlap that is minimal. In this case, the lowest we can achieve is an overlap of 1 operation on a contention point between each type of set $T_{\mathbf{x}}$. Formally, this translates into the following result.

Proposition 9. For all sets K with $|K| \geq 3$ and

$$K \subseteq \{\langle a, b, c \rangle, \langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle, \langle c, b, a \rangle\},$$

we have that $\text{ov}_{\max}^+(K) = |K| - 1$.

Proof. In what follows, we will represent

$$K \subseteq \{\langle a, b, c \rangle, \langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle, \langle c, b, a \rangle\}$$

with $|K| \geq 3$ as a directed graph $G = (K, A)$ where, for all $\{\mathbf{x}, \mathbf{y}\} \subseteq K$, we have $(\mathbf{x}, \mathbf{y}) \in A$ if $\text{ov}(\mathbf{x}, \mathbf{y}) = 1$.

We first recall definitions about directed graphs. First, notice that all our directed graphs are *simple*, in the sense of $(x, x) \notin A$ for all $x \in K$, since $ov(x, x) = 2$. The *indegree* $d^-(x)$ or *outdegree* $d^+(x)$ of a vertex $x \in K$ is the number of incoming arcs $|\{(y, x) \in A \mid y \in K\}|$ or outgoing arcs $|\{(x, y) \in A \mid y \in K\}|$ respectively. The *minimum indegree* $\delta^-(G)$ is defined as $\min\{d^-(x) \mid x \in K\}$ and the *minimum outdegree* $\delta^+(G)$ is defined as $\min\{d^+(x) \mid x \in K\}$. A *Hamiltonian (directed) path* is a (directed) path in G that visits each vertex precisely once.

To show $ov_{\max}^+(K) = |K| - 1$, it is sufficient to show that G has a Hamiltonian path. Indeed, the path gives a total order $K_{\leq} = (K, \leq_K)$ such that $ov^+(K_{\leq}) = |K| - 1$, since the overlap between each object sequence is precisely 1, and all sequences in K are visited by the path. We now show this is the case for each gradual increase in the minimum indegree and outdegree.

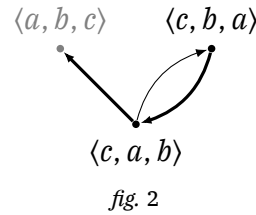
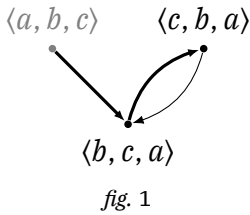
If $\delta^-(G) = 0$ or $\delta^+(G) = 0$, then G has a Hamiltonian path. Without loss of generality, assume $d^-(a b c) = 0$. This implies that

$$\{\langle a, c, b \rangle, \langle b, a, c \rangle, \langle c, a, b \rangle\} \not\subseteq K$$

since $ov(\langle a, c, b \rangle, \langle a, b, c \rangle) = 1$, $ov(\langle b, a, c \rangle, \langle a, b, c \rangle) = 1$ and since we have that $ov(\langle c, a, b \rangle, \langle a, b, c \rangle) = 1$. And, since $|K| \geq 3$, it follows that

$$K = \{\langle a, b, c \rangle, \langle c, b, a \rangle, \langle b, c, a \rangle\}.$$

A Hamiltonian path for these vertices is highlighted below, in bold (see *fig. 1*).



If $\delta^-(G) = 1$ or $\delta^+(G) = 1$, then G has a Hamiltonian path. Assume w.l.o.g. that $d^-(a b c) = 1$. This implies that either $\langle a, c, b \rangle \in K$, $\langle b, a, c \rangle \in K$ or $\langle c, a, b \rangle \in K$. The only possibilities for K are therefore given by fig. 2–10 (note that fig. 2 is given above), possible Hamiltonian paths are marked in bold.

Similarly, assume w.l.o.g. that $d^+(\langle a, b, c \rangle) = 1$. This implies that either $\langle a, c, b \rangle \in K$, $\langle b, a, c \rangle \in K$, or $\langle b, c, a \rangle \in K$. The only possibilities for K are given by fig. 1, 3, 5, 7, and 11–14 (note that fig. 1 is given above), possible Hamiltonian paths are marked in bold.

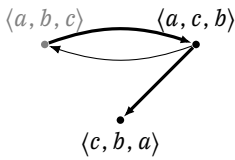


fig. 3

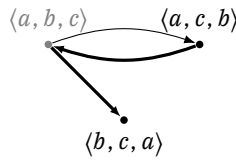


fig. 4

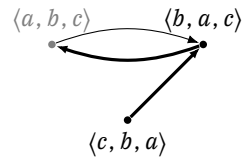


fig. 5

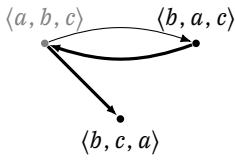


fig. 6

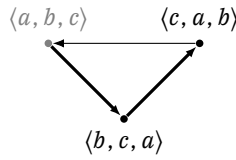


fig. 7

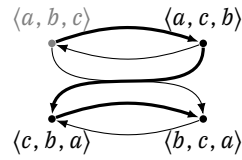


fig. 8

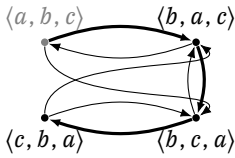


fig. 9

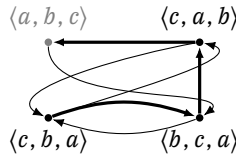


fig. 10

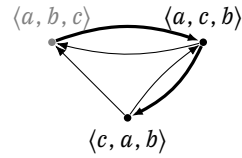


fig. 11

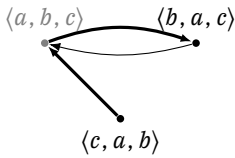


fig. 12

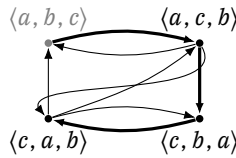


fig. 13

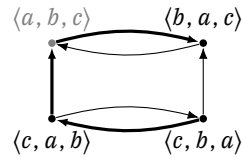


fig. 14

Claim: If $\delta^-(G) = 2$ or $\delta^+(G) = 2$, then G has a Hamiltonian path.

Assume w.l.o.g. that $d^-(a b c) = 2$. This implies that we either have that $\{\langle a, c, b \rangle, \langle b, a, c \rangle\} \subseteq K$, or $\{\langle a, c, b \rangle, \langle c, a, b \rangle\} \subseteq K$, or we have that

$\{\langle b, a, c \rangle, \langle c, a, b \rangle\} \subseteq K$. Notice that many cases for K lead to a graph G with $\delta^-(G) < 2$. Indeed, figures 11, 12, and 14 would normally be included but they contain vertices x with $d^-(x) = 1$, therefore contradicting with $\delta^-(G) = 2$. The only possibilities for K where $\delta^-(G) = 2$ are therefore given by figure 13, and figures 15–18, possible Hamiltonian paths are marked in bold.

Similarly, assume w.l.o.g. that $d^+(a b c) = 2$. This implies that either $\{\langle a, c, b \rangle, \langle b, a, c \rangle\} \subseteq K$, or $\{\langle a, c, b \rangle, \langle b, c, a \rangle\} \subseteq K$, or $\{\langle b, a, c \rangle, \langle b, c, a \rangle\} \subseteq K$. Analogously as before, many cases for K lead to a graph G with $\delta^-(G) < 2$, which we therefore exclude. The remaining cases for K where $\delta^-(G) = 2$ are given by figure 9, 17–20. The possible Hamiltonian paths are marked in bold.

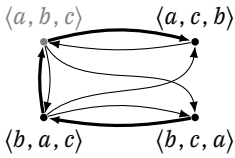


fig. 15

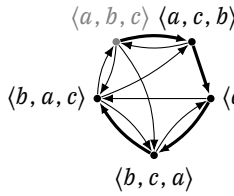


fig. 16

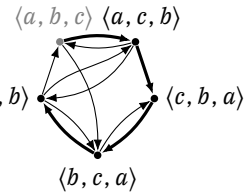


fig. 17

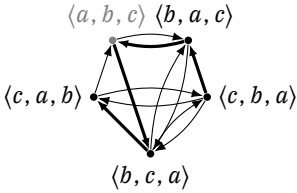


fig. 18

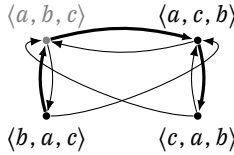


fig. 19

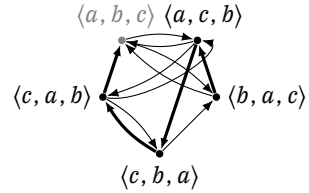


fig. 20

If $\delta^-(G) = 3$ or $\delta^+(G) = 3$, then G has a Hamiltonian path. Assume w.l.o.g. that $d^-(a b c) = 3$, this implies that

$$\{\langle a, c, b \rangle, \langle b, a, c \rangle, \langle c, a, b \rangle\} \subseteq K.$$

The possibilities for K are normally fig. 19, 20, 21, and

$$K = \{\langle a, b, c \rangle, \langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle\};$$

but since the former two and the last K do not satisfy $\delta^-(G) = 3$, we are left with only fig. 21. Indeed, for the last K , we have $d^-(\langle c, a, b \rangle) = 2$. A possible Hamiltonian path is given in bold.

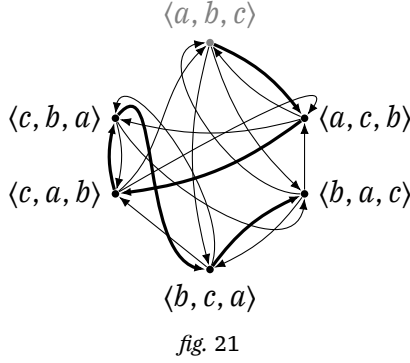
Similarly, assume w.l.o.g. that $d^+(\langle a, b, c \rangle) = 3$, then

$$\{\langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle\} \subseteq K.$$

The possibilities for K are fig. 15, 16, 21, and

$$K = \{\langle a, b, c \rangle, \langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle\}.$$

Again, we have that 15, 16, and the last K do not satisfy $\delta^+(G) = 3$. Hence, the only valid K is fig. 21.



There exists no K such that $\delta^-(G) = 4$ or $\delta^+(G) = 4$. Assume that there exists a K such that $\delta^-(G) = 4$ or $\delta^+(G) = 4$, this implies there is a $\mathbf{x} \in K$ such that $d^-(\mathbf{x}) = 4$ or $d^+(\mathbf{x}) = 4$. However, all incoming neighbours of $\mathbf{x}$ are given by

$$\{\langle \mathbf{x}_{(1)}, \mathbf{x}_{(3)}, \mathbf{x}_{(2)} \rangle, \langle \mathbf{x}_{(2)}, \mathbf{x}_{(1)}, \mathbf{x}_{(3)} \rangle, \langle \mathbf{x}_{(3)}, \mathbf{x}_{(1)}, \mathbf{x}_{(2)} \rangle\}$$

and hence $d^-(\mathbf{x}) = 3$. Similarly the outgoing neighbours of $\mathbf{x}$ are given by

$$\{\langle \mathbf{x}_{(1)}, \mathbf{x}_{(3)}, \mathbf{x}_{(2)} \rangle, \langle \mathbf{x}_{(2)}, \mathbf{x}_{(1)}, \mathbf{x}_{(3)} \rangle, \langle \mathbf{x}_{(2)}, \mathbf{x}_{(3)}, \mathbf{x}_{(1)} \rangle\},$$

hence $d^+(\langle \mathbf{x} \rangle) = 3$. In both cases, we arrive at a contradiction against our initial assumption that $\delta^+(G) = 4$ or $\delta^-(G) = 4$, hence there cannot exist a K such that $\delta^-(G) = 4$ or $\delta^+(G) = 4$.

In conclusion, since we can always build a Hamiltonian path in G for every chosen K such that $|K| \geq 3$, we can use that Hamiltonian path as a total order on the elements of K such that $\text{ov}_{\max}^+(K) = |K| - 1$. This concludes the proof. $\square$

Algorithm

In what follows, we will denote by $\text{permute_seq}(T)$ a sequence of contention point sequences in which the maximal total overlap is $|K(T)| - 1$, where $K(T) = \{\text{seq}(\tau) \mid \tau \in T\}$ constructed in the way described by the previous proof (*i.e.*, by copying over the Hamiltonian paths), with $K = K(T)$. In the next section we will introduce a generic algorithm for this, but one could implement this by building a hard-coded lookup table based on each Hamiltonian path given in the proof above.

We now define AS_3 (Algorithm 9) for sets in $\mathbb{T}_{(\ell \geq 3), 3, (n \geq 3)}$. We also require that $n \geq 3$ because of Proposition 9. We can obtain a better approximation for $\mathbb{T}_{\ell, m, 2}$ and $\mathbb{T}_{\ell, m, 1}$ which are trivial cases we discuss in the summary at the end.

Algorithm 9 $\text{AS}_3(T : \text{set in } \mathbb{T}_{(\ell \geq 3), 3, (n \geq 3)}) \rightarrow \text{schedule for } T$

```

1:  $run \leftarrow$  an empty sequence  $\langle \rangle$ 
2:  $ps \leftarrow \text{permute\_seq}(T)$ 
3: for all  $x$  in  $\langle ps_{(1)}, ps_{(2)}, \dots, ps_{(|ps|-1)} \rangle$  do
4:    $run3 \leftarrow \text{sort\_dilation}(T_x, \text{gte})$ 
5:    $run2 \leftarrow \text{sort\_dilation}(T_{\langle x_{(1)}, x_{(2)} \rangle}, \text{gte})$ 
6:    $run1 \leftarrow \text{sort\_suffix}(T_{\langle x_{(1)} \rangle}, \text{gte})$ 
7:    $run-x \leftarrow \text{merge}(\text{merge}(run3, run2, 1), run1, 1)$ 
8:    $run \leftarrow \text{append}(run, run-x)$ 
9:  $y \leftarrow ps_{(|ps|)}$ 
10:  $run3 \leftarrow \text{sort\_dilation}(T_y, \text{lte})$ 
11:  $run2 \leftarrow \text{sort\_dilation}(T_{\langle y_{(1)}, y_{(2)} \rangle}, \text{lte})$ 
12:  $run1 \leftarrow \text{sort\_suffix}(T_{\langle y_{(2)} \rangle}, \text{lte})$ 
13:  $run-y \leftarrow \text{merge}(\text{merge}(run3, run2, 1), run1, 1)$ 
14:  $run \leftarrow \text{append}(run, run-y)$ 
15:  $runNC \leftarrow$  any sequence on all non-conflicting transactions in  $T$ 
16:  $run \leftarrow \text{append}(run, runNC)$ 
17: output  $\text{canonical}(\text{ord}(run))$ 

```

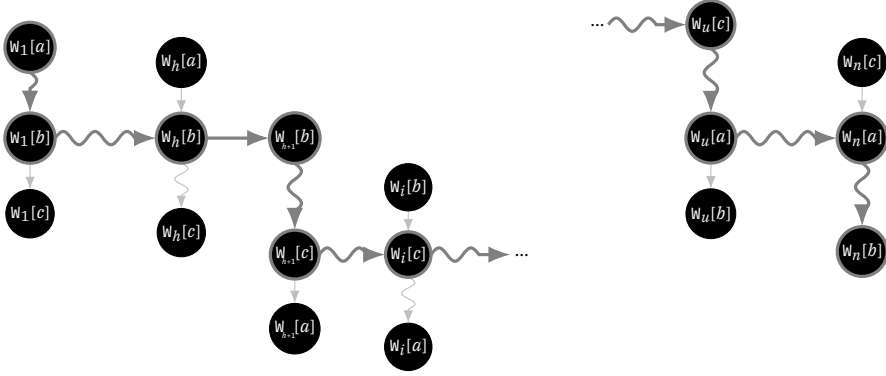


Figure 4.2 – A longest path (bold line) in the (implicit) precedence graph generated from a schedule yielded by AS_3 . Waved lines represent one or multiple intermediate operations. Edges between conflicting operations that do not contribute to the longest path are omitted for clarity.

Approximation Ratio

The algorithm AS_3 also runs in time quadratic in the number of operations and quasilinearly in the number of transactions, similar to AS_2 from the previous section. The approximation ratio is still in the order of $\mathcal{O}(n/\ell)$, which we will derive next. We start by proving an upper bound on the makespan of the schedules that AS_3 yields. Figure 4.2 can be used as a visual aid.

Proposition 10. For all $T \in \mathbb{T}_{\ell,3,n}$ with $\ell \geq 3$ it is the case that:

1. $\text{span}(AS_3(T)) \leq n\ell - n + 1$ if $3 \leq n \leq 6$,
2. $\text{span}(AS_3(T)) \leq 7\ell + n - 14$ if $7 \leq n$, and
3. $AS_3(T)$ runs in $\mathcal{O}(p^2 + n \log n)$ where $p = \sum_{\tau \in T} |\text{ops}(\tau)|$.

Proof. Let $T \in \mathbb{T}_{\ell,3,n}$ with $\ell \geq 3$ and assume that $|T| = n$, that all transactions are length ℓ , and that all $\tau \in T$ are 3-contending. We are allowed to do this without loss of generality due to Lemma 3.

For (1), let $3 \leq n \leq 6$. Then, define a total order $\Sigma = (T, \leq_T)$ based on $AS_3(T)$. Furthermore, let $(K, \leq_K)$ be the total order defined by $\text{seq}(\tau_i) \leq_K \text{seq}(\tau_j)$ if, and only if $\tau_i \leq_T \tau_j$ for all τ_i and τ_j in T . By construction, we know P_Σ has a path starting in each $w_i[\mathbf{x}_{(1)}]$, and transitioning to the next transaction at $(w_i[\mathbf{x}_{(2)}], w_{i+1}[\mathbf{y}_{(1)}])$ for $\mathbf{x} \leq_K \mathbf{y}$

and for each $i \in [n - 1]$.

In the worst case, $\text{pre}(\tau_j) = \text{suf}(\tau_j) = 0$ and the number of operations between $W_j[z_{(1)}]$ and $W_j[z_{(2)}]$ is $\ell - 3$ for each $j \in [n]$ and $z \in K$. We also need to count $W_j[z_{(1)}]$ and $W_j[z_{(2)}]$ for a cost of $\ell - 1$ per transaction in T . Plus an additional operation $W_n[w_{(3)}]$ at the end (with $w \in K$). In total we therefore have that $\text{span}(\text{cs}(P_\Sigma)) \leq n(\ell - 1) + 1 = n\ell - n + 1$.

For (2), let $7 \leq n$. We order T by AS_3 allowing to refer to its elements by $\tau_1, \tau_2, \dots, \tau_n$, we will refer to the order itself by Σ . Now, let q be a longest path in P_Σ , with $o \in \tau_i$ and $p \in \tau_j$ respectively the first and last operation of q . Our goal is to derive the amount of nodes on this path (as this is precisely the makespan, *i.e.*, the amount of edges + 1) Now, we have that either: $\text{seq}(\tau_i) = \text{seq}(\tau_j)$; or $\text{seq}(\tau_i) \neq \text{seq}(\tau_j)$ such that all transactions in $\text{seq}(\tau_i)$ appear before all transactions in $\text{seq}(\tau_j)$, by construction of AS_3 .

In case that $\text{seq}(\tau_i) = \text{seq}(\tau_j)$: if q starts in a transaction and ends in a distinct transaction with the same contention point order, then q can take many possible paths from τ_i to τ_j , however, all of these paths *must* include at least $\text{pre}(\tau_i) + \text{suf}(\tau_j) + (j - i) + 3$. The remaining path length is given by:

$$\max \{ \delta_{(1,2)}^i, \delta_{(1,2)}^{i+1}, \dots, \delta_{(1,2)}^j \} + \max \{ \delta_{(2,3)}^i, \delta_{(2,3)}^{i+1}, \dots, \delta_{(2,3)}^j \},$$

where $\delta_{(x,y)}^k = \text{cpos}(\tau_k, y) - \text{cpos}(\tau_k, x)$.

In the worst case, all such $\delta^k(x, y)$ equal $\ell - 3$. Hence, we get $\text{pre}(\tau_i) + \text{suf}(\tau_j) + (j - i) + 2\ell - 3$ for the entire maximal length of q . For $(j - i)$, notice that every conflict point order (there are $3! = 6$ of these) must be present, hence $j - i$ can maximally be $n - 5$. Furthermore, $\text{pre}(\tau_i)$ and $\text{suf}(\tau_j)$ can both maximally be $\ell - 3$.

In conclusion, the maximal path length of q is $(\ell - 3) + (\ell - 3) + n - 5 + 2\ell - 3$, which is equal to $4\ell + n - 14$. Since $4\ell + n - 14 < 7\ell + n - 14$ we can conclude that the length of q is in this case bounded by $7\ell + n - 14$.

In case that $\text{seq}(\tau_i) \neq \text{seq}(\tau_j)$: Here, we will assume without loss of generality that $\text{ord}(\text{permute_seq}(T))$ equals:

$$\langle \langle a, b, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle, \langle a, c, b \rangle, \langle c, b, a \rangle, \langle b, a, c \rangle \rangle.$$

Notice that its maximal total overlap is 5. The path q starts in a transaction and ends in a distinct transaction with a different contention point sequence. In the worst case we can visit all transactions, hence $i = 1$ and $j = n$. The part of q that starts in τ_i (which must satisfy $\text{seq}(T_i) = \langle a, b, c \rangle$ due to AS_3), and includes all transactions τ for which $\text{seq}(\tau) = a b c$, contributes:

$$\text{pre}(T_1) + 1 + \max\{\delta_{(1,2)}^1, \delta_{(1,2)}^2, \dots, \delta_{(1,2)}^{n_{\langle a,b,c \rangle}}\} + 1 + (n_{\langle a,b,c \rangle} - 1), \quad (4.7)$$

where $n_{\langle a,b,c \rangle} = |T_{\langle a,b,c \rangle}|$, and $\delta_{(x,y)}^k = \text{cpos}(\tau_k, y) - \text{cpos}(\tau_k, x)$. Due to the construction of AS_3 , the quantity $\delta_{(1,2)}^1$ will be maximal, hence in the worst case $\delta_{(1,2)}^1 = \ell - 3$ and $\text{pre}(\tau_1) = 0$. Therefore the Equation 4.7 is bounded by $1 + \ell - 3 + 1 + (n_{\langle a,b,c \rangle} - 1) = \ell + n_{\langle a,b,c \rangle} - 2$.

The part of q that ends in τ_n (which satisfies $\text{seq}(\tau_n) = \langle b, a, c \rangle$ by construction), and includes all $\tau \in T$ such that $\text{seq}(\tau) = b a c$, contributes for:

$$1 + \max\{\delta_{(1,2)}^{n-n_{\langle b,a,c \rangle}}, \delta_{(1,2)}^{n-n_{\langle b,a,c \rangle}+1}, \dots, \delta_{(1,2)}^n\} + n_{\langle b,a,c \rangle} \\ + \max\{\delta_{(2,3)}^{n-n_{\langle b,a,c \rangle}}, \delta_{(2,3)}^{n-n_{\langle b,a,c \rangle}+1}, \dots, \delta_{(2,3)}^n\} + 1 + \text{suf}(\tau_n),$$

where $n_{\langle b,a,c \rangle} = |T_{\langle b,a,c \rangle}|$. Both $\max\{\delta_{(1,2)}^x \mid x \in [n - n_{\langle b,a,c \rangle}, n]\}$ and the quantity $\max\{\delta_{(2,3)}^x \mid x \in [n - n_{\langle b,a,c \rangle}, n]\}$ are each bounded by $\ell - 3$. Therefore, in total, this expression can be at most $1 + \ell - 3 + n_{\langle b,a,c \rangle} + \ell - 3 + 1 + 0$, or simplifying further: $2\ell + n_{\langle b,a,c \rangle} - 4$.

Lastly, the part of q that goes through all τ with $\text{seq}(\tau) = \mathbf{x}$ such that $\mathbf{x} \in \{\langle b, c, a \rangle, \langle c, a, b \rangle, \langle a, c, b \rangle, \langle c, b, a \rangle\}$, follows the following pattern:

$$1 + \max\{\delta_{(1,2)}^k, \delta_{(1,2)}^{k+1}, \dots, \delta_{(1,2)}^{k+n_{\mathbf{x}}}\}$$

where $n_{\mathbf{x}} = |T_{\mathbf{x}}|$. Again, in the worst case this is bounded by $1 + \ell - 3 = \ell - 2$ for every $T_{\mathbf{x}}$. Since there are 4 such sequences, and since each $T_{\mathbf{x}}$ contains $n_{\mathbf{x}}$ transactions, we get an overall bound of $4(\ell - 2) + n_{\langle b,c,a \rangle} + n_{\langle c,a,b \rangle} + n_{\langle a,c,b \rangle} + n_{\langle c,b,a \rangle}$.

Aggregating the costs for all segments together we have:

$$(\ell + n_{\langle a,b,c \rangle} - 2) + (2\ell + n_{\langle b,a,c \rangle} - 4) + \\ (4\ell - 8 + n_{\langle b,c,a \rangle} + n_{\langle c,a,b \rangle} + n_{\langle a,c,b \rangle} + n_{\langle c,b,a \rangle}) = \\ 7\ell + n_{\langle a,b,c \rangle} + n_{\langle b,c,a \rangle} + n_{\langle c,a,b \rangle} + n_{\langle a,c,b \rangle} + n_{\langle c,b,a \rangle} + n_{\langle b,a,c \rangle} - 14 = \\ 7\ell + n - 14.$$

Hence, we have that the number of nodes on q is bounded by $7\ell + n - 14$. The result for (2) then follows from Corollary 3.

Finally, to show result (3), observe that generating the permutations is constant time (since we have a lookup table), lines 3–16 involve sorting 6 times and merging 12 times which comes down to a time complexity of $\mathcal{O}(n \log n)$, and step 17 involves building a canonical schedule, which takes $\mathcal{O}(p^2)$ operations where p is defined as $\sum_{\tau \in T} |\text{ops}(\tau)|$. In total, this gives a time complexity of $\mathcal{O}(p^2 + n \log n)$, which concludes the proof. $\square$

Next, we construct a number of transaction sets that have a high optimal makespan. For the case where $3 \leq n \leq 6$ this follows from Proposition 9, and this will give us a tight approximation. However, once we allow $7 \leq n$, a new sub-problem arises. That is, we need to optimally schedule all T_x , since $|T_x| \geq 2$ for some $x = \text{seq}(\tau)$ and $\tau \in T$. Notice that each $\tau \in T_x$ is of the form:

$$\tau = \overbrace{\dots}^{\delta^1} W[\mathbf{x}_{(1)}] \overbrace{\dots}^{\delta^2} W[\mathbf{x}_{(2)}] \overbrace{\dots}^{\delta^3} W[\mathbf{x}_{(3)}] \overbrace{\dots}^{\delta^4}$$

where, for all $j \in [4]$, the dots “...” represent $\delta^j \in [0, \ell - 3]$ read operations on unique objects, such that $\delta^1 + \delta^2 + \delta^3 + \delta^4 = \ell - 3$. We write these transactions as tuples $(\delta^1, \delta^2, \delta^3, \delta^4)$ if the x is irrelevant. To maximize the optimal makespan, we need assignments for all δ^j in all $\tau \in T_x$, such that $\text{opt}(T_x)$ is maximal.

We now give our best estimates. If $|T_x| = 2$, then we will define $T_2 = \{(0, 0, \ell - 3, 0), (0, \ell - 3, 0, 0)\}$. If $|T_x| = 3$, then this is defined as $T_3 = \{(0, \lfloor \ell - 3/2 \rfloor, 0, \lfloor \ell - 3/2 \rfloor)\} \cup T_2$. And finally, when $|T_x| = 4$ then $T_4 = \{(0, \ell - 4, 1, 0), (0, 1, \ell - 4, 0)\} \cup T_2$. To go beyond this, we can repeat any $\tau \in T_4$ until $|T_x| = n$. Figure 4.3 shows some optimal schedules for these sets. The above T_x are then concatenated with a maximal total overlap of $n - 1$ (c.f., Proposition 9). Hence, the first 5 orders x each add $\ell - 2 + |T_x|$ to the makespan, the final order y adds $\text{opt}(T_y)$. This leads to the following transaction sets

Proposition 11. There exist $T \in \mathbb{T}_{\ell, 3, n}$ such that:

1. if $3 \leq n \leq 6$ then $\text{opt}(T) = n\ell - n + 1$,
2. if $7 \leq n \leq 17$ then $\text{opt}(T) = 6\ell + n - 11$,
3. if $18 \leq n \leq 23$ then $\text{opt}(T) = 6\ell + \lfloor (\ell - 3)/2 \rfloor + n - 11$, and
4. if $24 \leq n$ then $\text{opt}(T) = 7\ell + n - 15$.

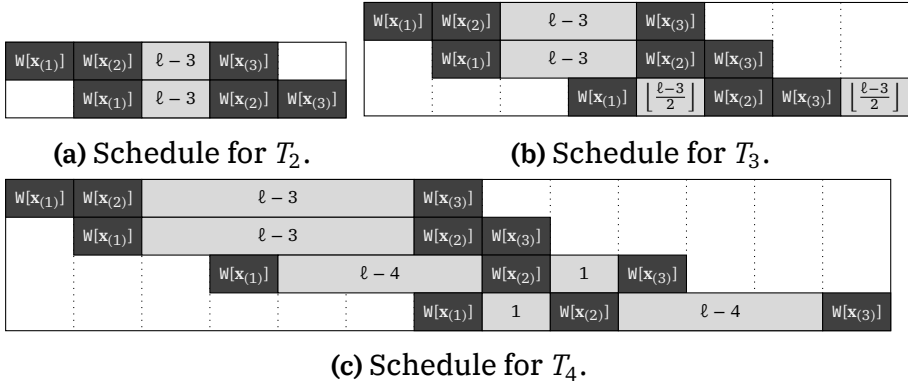


Figure 4.3 – Schedules for T_x such that $\text{opt}(T_x)$ is maximized, for $|T_x| \in \{2, 3, 4\}$. Light gray boxes with numbers indicate there are that amount of read operations on unique objects.

Proof. For (1), we construct a set $T_{[3,6]}$. First, let

$$K \subseteq \{\langle a, b, c \rangle, \langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle, \langle c, b, a \rangle\}$$

with $|K| = n$ such that $3 \leq n \leq 6$. Then, for every $i \in [n]$, add a transaction τ_i to $T_{[3,6]}$ such that:

$$\tau_i = \{W_i[x_{(1)}], R_i[u_1], \dots, R_i[u_{\ell-3}], W_i[x_{(2)}], W_i[x_{(3)}]\},$$

where $u_1, \dots, u_{\ell-3}$ are random unique objects. Each transaction is ℓ operations long, but precisely $\text{ov}(K) = |K| - 1$ operations are overlapping (due to Proposition 9). Since $|K| = n$, we then have

$$\text{opt}(T_{[3,6]}) = |K|\ell - (|K| - 1) = n\ell - (n - 1) = n\ell + n - 1.$$

For (2), (3), and (4), we remark that we can order

$$\{\langle a, b, c \rangle, \langle a, c, b \rangle, \langle b, a, c \rangle, \langle b, c, a \rangle, \langle c, a, b \rangle, \langle c, b, a \rangle\}$$

into a sequence $\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^6$ such that $\mathbf{x}_{(2)}^i = \mathbf{x}_{(1)}^{i+1}$ for all $i \in [5]$. Indeed, just take the order

$$\langle a, b, c \rangle, \langle b, c, a \rangle, \langle c, b, a \rangle, \langle b, a, c \rangle, \langle a, c, b \rangle, \langle c, a, b \rangle,$$

or more generally:

$$\begin{aligned} &\langle \mathbf{x}_{(1)}, \mathbf{x}_{(2)}, \mathbf{x}_{(3)} \rangle, \langle \mathbf{x}_{(2)}, \mathbf{x}_{(3)}, \mathbf{x}_{(1)} \rangle, \langle \mathbf{x}_{(3)}, \mathbf{x}_{(2)}, \mathbf{x}_{(1)} \rangle, \\ &\langle \mathbf{x}_{(2)}, \mathbf{x}_{(1)}, \mathbf{x}_{(3)} \rangle, \langle \mathbf{x}_{(1)}, \mathbf{x}_{(3)}, \mathbf{x}_{(2)} \rangle, \langle \mathbf{x}_{(3)}, \mathbf{x}_{(1)}, \mathbf{x}_{(2)} \rangle. \end{aligned}$$

Therefore, we can always place the T_x with the minimal cost at $\mathbf{x}^6$, and secondly, the makespan is always

$$5(\ell - 2) + \sum_{i=1}^5 |T_{\mathbf{x}^i}| + \text{opt}(T_{\mathbf{x}^6}). \quad (4.8)$$

since the transaction $\tau \in T_{\mathbf{x}^i}$ with the largest $\text{cpos}(\tau, 2) - \text{cpos}(\tau, 1)$ always satisfies $\text{cpos}(\tau, 2) - \text{cpos}(\tau, 1) = \ell - 3$. We also add $+1$ to obtain $\ell - 2$ since we always have to traverse within one transaction from $\mathbf{x}_{(1)}$ to $\mathbf{x}_{(2)}$ (see Figure 4.3).

Now, we compute bounds on Equation 4.8 depending on the bounds we have for n .

In case (2), when $7 \leq n \leq 17$: Observe that $|T_{\mathbf{x}^6}| \in \{1, 2\}$. Hence, $\text{opt}(T_{\mathbf{x}^6}) = \ell - 1 + |T_{\mathbf{x}^6}|$ (see Figure 4.3a for $|T_{\mathbf{x}^6}| = 2$). Therefore, in the worst case, Equation 4.8 further reduces to:

$$5(\ell - 2) + \ell - 1 + \sum_{i=1}^6 |T_{\mathbf{x}^i}| = 6\ell + n - 11.$$

In case (3), when $18 \leq n \leq 23$: Note that $|T_{\mathbf{x}^6}| \in \{2, 3\}$. Hence (see Figures 4.3a and 4.3b),

$$\text{opt}(T_{\mathbf{x}^6}) \in \{\ell - 1 + |T_{\mathbf{x}^6}|, \ell - 1 + \lfloor \ell - 3/2 \rfloor + |T_{\mathbf{x}^6}|\}.$$

Therefore, in the worst case, Equation 4.8 further reduces to:

$$5(\ell - 2) + \ell - 1 + \lfloor \ell - 3/2 \rfloor + \sum_{i=1}^6 |T_{\mathbf{x}^i}| = 6\ell + \lfloor \ell - 3/2 \rfloor + n - 11.$$

In case (4), when $24 \leq n$: Note that $|T_{\mathbf{x}^6}| \geq 4$. Hence (see Figures 4.3c),

$$\text{opt}(T_{\mathbf{x}^6}) = 2\ell - 5 + |T_{\mathbf{x}^6}|$$

Therefore, in the worst case, Equation 4.8 further reduces to:

$$5(\ell - 2) + 2\ell - 5 + \sum_{i=1}^6 |T_{\mathbf{x}^i}| = 7\ell + n - 15.$$

This concludes the proof. □

We now conclude this section by showing that AS_3 is a tight approximation algorithm when $3 \leq n \leq 6$ and not tight otherwise. In both cases the approximation ratio is in the order of $\mathcal{O}(n/\ell)$.

Theorem 8. AS_3 is a tight $(\frac{1-n}{\ell} + n)$ -approximation algorithm for $(3, n)$ -TxSCHED if $3 \leq n \leq 6$ and $\ell \geq 3$. And, AS_3 is a $(\frac{n-14}{\ell} + 7)$ -approximation algorithm for $(3, n)$ -TxSCHED if $7 \leq n$ and $\ell \geq 3$.

Proof. In the case where $3 \leq n \leq 6$ the upper bound given by Proposition 10 is $n\ell - n + 1$. The trivial lower bound which holds for all instances is ℓ (i.e., all transactions are scheduled in parallel), therefore the approximation ratio is given by:

$$\begin{aligned} \frac{n\ell - n + 1}{\ell} &= \frac{n\ell}{\ell} - \frac{n + 1}{\ell} \\ &= n + \frac{-n + 1}{\ell} \\ &= \frac{1 - n}{\ell} + n \end{aligned}$$

Given that there exist transaction sets $T \in \mathbb{T}_{\ell, 3, n}$ such that $\text{opt}(T) = n\ell - n + 1$ (c.f., Proposition 11), this shows that AS_3 is tight when $3 \leq n \leq 6$.

In the case where $7 \leq n$, the upper bound from Proposition 10 is $7\ell + n - 14$, hence the approximation ratio is:

$$\frac{7\ell + n - 14}{\ell} = \frac{7\ell}{\ell} + \frac{n - 14}{\ell} = \frac{n - 14}{\ell} + 7$$

The sets with highest optimal makespan that we found in Proposition 11 do not have this makespan, hence we cannot say that AS_3 is tight in this case. This concludes the proof. $\square$

4 Fixed Contention Point Scheduling

The final algorithm we will present is concerned with the following problem. We now lift the restriction on m , and just require that m is a fixed quantity known up front. We still require that $\ell \geq m$. What changes is that we can no longer call our algorithm an approximation algorithm, since it does not run in polynomial time.

(m, n) -TxSCHED

Input: A transaction set $T \in \mathbb{T}_{\ell, m, n}$ with $\ell \geq m$.
Output: A conflict serializable schedule s for T .
Goal: Minimize $\text{span}(s)$.

Adjacent Transposition Sequences

We start by introducing another important tool, which will be useful for generalizing Proposition 9. The terminology originates from combinatorics, precisely the study of permutations, where transpositions can refer to swapping two arbitrary elements within a given permutation.

Definition 19. Let $\mathbf{x}$ and $\mathbf{y}$ be equal-length sequences of $z \geq 1$ unique objects, then $\mathbf{y}$ is an *adjacent transposition* of $\mathbf{x}$ if

$$\mathbf{y} = \mathbf{x}_{(1)} \mathbf{x}_{(2)} \dots \mathbf{x}_{(i+1)} \mathbf{x}_{(i)} \dots \mathbf{x}_{(z)}$$

for some $i \in [z - 1]$.

The notation $\mathbf{x} \curvearrowright \mathbf{y}$ denotes that $\mathbf{y}$ is an adjacent transposition of $\mathbf{x}$. Intuitively, $\mathbf{y}$ swaps precisely two adjacent elements in $\mathbf{x}$. We will write $\mathbf{x} \curvearrowright \mathbf{y} \curvearrowright \mathbf{z}$ to mean that both $\mathbf{x} \curvearrowright \mathbf{y}$ and $\mathbf{y} \curvearrowright \mathbf{z}$. Additionally, we call $\text{seq}(m) = \langle \mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^{m!} \rangle$ where each sequence has m operations an *adjacent transposition sequence for m* if $\mathbf{x}^1 \curvearrowright \mathbf{x}^2 \curvearrowright \dots \curvearrowright \mathbf{x}^{m!}$. Adjacent transpositions have the following property.

Lemma 4. If $\mathbf{x}$ and $\mathbf{y}$ are equal-length sequences of $m \geq 3$ unique objects, and $\mathbf{x} \curvearrowright \mathbf{y}$, then $\text{ov}(\mathbf{x}, \mathbf{y}) = m - 2$.

Proof. Let $m \geq 3$, and let $\mathbf{x}$ and $\mathbf{y}$ be two equal-length sequences of m distinct objects such that $\mathbf{x} \curvearrowright \mathbf{y}$. We then define sets $Y_{(z)}$ containing all possible objects which can appear as $\mathbf{y}_{(z)}$ with $z \in [m]$. Formally,

we have that:

$$\begin{aligned} Y_{(1)} &= \{\mathbf{x}_{(1)}, \mathbf{x}_{(2)}\}, \\ Y_{(i)} &= \{\mathbf{x}_{(i-1)}, \mathbf{x}_{(i)}, \mathbf{x}_{(i+1)}\}, \text{ for all } i \in [2, m-1], \\ Y_{(m)} &= \{\mathbf{x}_{(m-1)}, \mathbf{x}_{(m)}\}. \end{aligned}$$

To show that $\text{ov}(\mathbf{x}, \mathbf{x}) = m - 2$, we need to prove that (1) for all $j \in [m - 2]$ and $i \in [j + 2, m]$ it holds that $\mathbf{x}_{(i)} \neq \mathbf{y}_{(j)}$; and (2) we need to show that j can not take on a value higher than $m - 2$ while still satisfying $\mathbf{x}_{(i)} \neq \mathbf{y}_{(j)}$.

To show (1), we define sets $X_{(j+2)}$ for all $j \in [m - 2]$ containing all $\mathbf{x}_{(i)}$ for $i \in [j + 2, m]$. That is:

$$X_{(k)} = \{\mathbf{x}_{(k)}, \mathbf{x}_{(k+1)}, \dots, \mathbf{x}_{(m)}\}, \text{ for all } k \in [3, m].$$

It then immediately becomes clear that:

$$\begin{aligned} (X_{(3)} = \{\mathbf{x}_{(3)}, \dots, \mathbf{x}_{(m)}\}) &\not\subseteq (\{\mathbf{x}_{(1)}, \mathbf{x}_{(2)}\} = Y_{(1)}), \\ (X_{(4)} = \{\mathbf{x}_{(4)}, \dots, \mathbf{x}_{(m)}\}) &\not\subseteq (\{\mathbf{x}_{(1)}, \mathbf{x}_{(2)}, \mathbf{x}_{(3)}\} = Y_{(2)}), \\ &\dots \\ (X_{(m)} = \{\mathbf{x}_{(m)}\}) &\not\subseteq (\{\mathbf{x}_{(m-3)}, \mathbf{x}_{(m-2)}, \mathbf{x}_{(m-1)}\} = Y_{(m-2)}). \end{aligned}$$

Hence, for all $j \in [m - 2]$ and $i \in [j + 2, m]$ it follows that $\mathbf{x}_{(i)} \neq \mathbf{y}_{(j)}$.

To show (2), that is, $m - 2$ is the maximal value j can take for which the above is possible. We show that there exists a $j \in [m - 1]$ and $i \in [j + 2, m]$ such that $\mathbf{x}_{(i)} = \mathbf{y}_{(j)}$. Indeed, if we let $i = m$ and $j = m - 1$ we get: $(X_{(m)} = \{\mathbf{x}_{(m)}\}) \subseteq (\{\mathbf{x}_{(m-2)}, \mathbf{x}_{(m-1)}, \mathbf{x}_{(m)}\} = Y_{(m-1)})$. In other words, if we define $\mathbf{y} = \mathbf{x}_{(1)}, \mathbf{x}_{(2)}, \dots, \mathbf{x}_{(m)}, \mathbf{x}_{(m-1)}$, we get the desired result. This concludes the proof. $\square$

The next insight follows from the above lemma and the Steinhaus-Johnson-Trotter algorithm, which for a permutation of m objects, generates an adjacent transposition sequence for m [51, 83, 88].

For instance, for three objects the algorithm would start from a single permutation on one object, a . It would then place the second object, in lexicographical descending order, every possible legal position to get the two permutations on two objects; namely: a, b and b, a . And finally, it would place the final object in every possible

position. First in descending lexicographic order for a, b and then in ascending lexicographic order for b, a . This yields a, b, c ; a, c, b ; and c, a, b for the former, followed by c, b, a ; b, c, a ; and b, a, c for the latter. This sequence is then the final output of the algorithm. We can keep extending this recursive process for larger and larger m .

There are many variations of this algorithm, among some that use constant space per iteration. But this prior example illustrates the general idea. In the remainder of this section, we will refer to this algorithm by the notation `steinhaus_johnson_trotter( $m$ )` in algorithm specifications. This algorithm does not run in polynomial time, as it generates all possible sequences of m objects (*i.e.*, it outputs $m!$ items).

Lemma 5. If $m \geq 4$, $\ell \geq m$, $n \geq m!$, and $T \in \mathbb{T}_{\ell, m, n}$ with $K = \{\text{seq}(\tau) \mid \tau \in T\}$ containing all equal-length sequences of m distinct objects, then $\text{ov}_{\max}^+(K) = (n - 1)(m - 2)$.

Proof. Let K be the set from the formulation of the lemma. We can apply the Steinhaus-Johnson-Trotter algorithm [51, 83, 88] to obtain a total order $(K, \leq_K)$ on all elements in K , say $\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^{m!}$, such that $\mathbf{x}^1 \curvearrowright \mathbf{x}^2 \curvearrowright \dots \curvearrowright \mathbf{x}^{m!}$. The desired value for $\text{ov}_{\max}^+(K)$ then readily follows from applying Lemma 4. $\square$

Algorithm

We can now define a generalization of AS_3 and AS_2 to an arbitrary (yet fixed) amount of contention points. The algorithm makes use of the Steinhaus-Johnson-Trotter algorithm.

Algorithm 10 $AS_m(T : \text{set in } \mathbb{T}_{(\ell \geq m), m, n}) \rightarrow \text{schedule for } T$

```

1:  $run \leftarrow$  an empty sequence  $\langle \rangle$ .
2:  $ats \leftarrow \text{steinhaus\_johnson\_trotter}(m)$ 
3: for all  $\mathbf{x}$  in  $\langle ats_{(1)}, ats_{(2)}, \dots, ats_{(m-1)} \rangle$  do
4:    $run\text{-}\mathbf{x} \leftarrow$  an empty sequence  $\langle \rangle$ .
5:   for all  $i \in \langle m, m-1, \dots, 2 \rangle$  do
6:      $run\text{-}part \leftarrow \text{sort\_dilation}(T_{\langle \mathbf{x}_{(1)}, \mathbf{x}_{(2)}, \dots, \mathbf{x}_{(i)} \rangle}, gte)$ 
7:      $run\text{-}\mathbf{x} \leftarrow \text{merge}(run\text{-}\mathbf{x}, run\text{-}part)$ 
8:      $run\text{-}part \leftarrow \text{sort\_suffix}(T_{\langle \mathbf{x}_{(1)} \rangle}, gte)$ 
9:      $run\text{-}\mathbf{x} \leftarrow \text{merge}(run\text{-}\mathbf{x}, run\text{-}part)$ 
10:     $run \leftarrow \text{append}(run, run\text{-}\mathbf{x})$ 
11:  $\mathbf{x} \leftarrow ats_{(m)}$ 
12:  $run\text{-}\mathbf{x} \leftarrow$  an empty sequence  $\langle \rangle$ .
13: for all  $i \in \langle 1, 2, \dots, m-1 \rangle$  do
14:    $run\text{-}part \leftarrow \text{sort\_dilation}(T_{\langle \mathbf{x}_{(i)}, \mathbf{x}_{(i+1)}, \dots, \mathbf{x}_{(m)} \rangle}, lte)$ 
15:    $run\text{-}\mathbf{x} \leftarrow \text{merge}(run\text{-}\mathbf{x}, run\text{-}part)$ 
16:    $run\text{-}part \leftarrow \text{sort\_suffix}(T_{\langle \mathbf{x}_{(1)} \rangle}, lte)$ 
17:    $run\text{-}\mathbf{x} \leftarrow \text{merge}(run\text{-}\mathbf{x}, run\text{-}part)$ 
18:  $run \leftarrow \text{append}(run, run\text{-}\mathbf{x})$ 
19:  $runNC \leftarrow$  any sequence on all non-conflicting transactions in  $T$ 
20:  $run \leftarrow \text{append}(run, runNC)$ 
21: output  $\text{canonical}(\text{ord}(run))$ 

```

Makespan Bounds

We can adjust the first upper bound of Proposition 10 to work for an arbitrary number of contention points. However, this bound is not based on AS_m but rather on an argument that is not constructive. A better upper bound would require a generalization of Proposition 9, which we have not been able to achieve.

Proposition 12. If $m \geq 4$, $\ell \geq m$, and $3 \leq n < m!$, then for all $T \in \mathbb{T}_{\ell, m, n}$ it is the case that $\text{opt}(T) \leq n\ell - n + 1$.

Proof. To show this result, we show that for

$$K \subseteq \{\mathbf{x} \mid \mathbf{x} \text{ is a permutation of } m \text{ objects}\}$$

with $3 \leq |K| < m!$, then there exists a sequence $\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^{|K|}$ such that $\text{ov}(\mathbf{x}^i, \mathbf{x}^{i+1}) = 1$ for all $i \in [m-1]$.

Indeed, if this were true, it implies that there exists a sequence for transactions in T such that each transaction overlaps with the next one by 1 operation. In the worst case where each transaction is of the form:

$$\left\langle \overbrace{w[x_{(1)}], \dots, w[x_{(2)}], \dots, w[x_{(m-1)}], \dots, w[x_{(m)}]}^{\ell-1 \text{ operations}} \right\rangle$$

This means that the overall makespan will be $n(\ell-1)+1$ or $n\ell-n+1$. We continue by induction on $|K|$.

In the base case, $|K| = 3$: Let $K = \{x, y, z\}$. Now, assume there exists no total ordering for K such that $ov(x^i, x^{i+1}) = 1$ for $i \in [2]$. This implies that:

1. $x_{(m)} = y_{(1)}$ and $x_{(1)} = y_{(m)}$; and
2. $x_{(m)} = z_{(1)}$ and $x_{(1)} = z_{(m)}$; and
3. $y_{(m)} = z_{(1)}$ and $y_{(1)} = z_{(m)}$.

Therefore, $x_{(1)} = x_{(m)}$, since $x_{(1)} = y_{(m)} = z_{(1)} = x_{(m)}$. However, this is a contradiction, since objects in x must be unique (otherwise they are not a permutation). Hence, there must exist a total ordering for K such that $ov(x^i, x^{i+1}) = 1$ for $i \in [2]$.

Inductive Case. Now, assume the property holds for some $k < m!$ and K' with $|K'| = k$, then, we show that it holds for a K with $K' \subset K$ and $|K| = k + 1$. Let w be the additional element in K .

Now, assume there exists no total ordering for K with $ov(x^i, x^{i+1}) = 1$ for $i \in [k]$. By the induction hypothesis, there exists a sequence $x^1, x^2, \dots, x^k$ for K' such that $ov(x^i, x^{i+1}) = 1$ for $i \in [k-1]$. This implies that:

1. $x^k_{(m)} = w_{(1)}$ and $w_{(m)} = x^1_{(1)}$ (i.e., we cannot place w at the beginning or the end); and
2. $x^k_{(m)} = x^1_{(1)}$ and $x^{k-1}_{(m)} = w_{(1)}$ (i.e., we cannot place x^k in the beginning and place w in its place at the end).

This implies that $w_{(1)} = w_{(m)}$, since $w_{(1)} = x^k_{(m)} = x^1_{(1)} = w_{(m)}$. However, this is a contradiction since objects in w must be unique.

Hence, there must exist a total ordering for K such that $\text{ov}(\mathbf{x}^i, \mathbf{x}^{i+1}) = 1$ for $i \in [k]$. This concludes the proof. $\square$

In case that $n = m!$ or $n > m!$, we can determine a makespan bound on the schedules given by AS_m . As we will see, its running time is dominated by a factor $m!$ since we have to generate all permutations on m elements in a specific sequence (*i.e.*, the Steinhaus-Johnson-Trotter algorithm).

Theorem 9. For all $T \in \mathbb{T}_{\ell, m, n}$ such that $m \geq 4$ and $\ell \geq m$ we have that:

1. If $n = m!$, then $\text{span}(\text{AS}_m(T)) \leq n\ell - (n-1)(m-2)$,
2. If $n > m!$, then:

$$\text{span}(\text{AS}_m(T)) \leq (m! + m - 2)\ell + n - (m-1)(m! + m - 2),$$

3. The algorithm $\text{AS}_m(T)$ runs in time $\mathcal{O}(m! + p^2 + mn \log n)$ with $p = \sum_{\tau \in T} |\text{ops}(\tau)|$.

Proof. Let $T \in \mathbb{T}_{\ell, m, n}$ with $\ell \geq m$ and assume that $|T| = n$, that all transactions are length ℓ , and that all $\tau \in T$ are m -contending. We are allowed to do this without loss of generality due to Lemma 3. During the invocation of $\text{AS}_m(T)$ we construct a total order $\leq_K$ on $K = \{\text{seq}(\tau) \mid \tau \in T\}$ given by the Steinhaus-Johnson-Trotter algorithm [51, 83, 88] (*c.f.*, line 2). We will also refer to the $\text{ord}(\text{run})$ derived on line 20 as the total order $\Sigma = (T, \leq_T)$. We now show that any longest path q (in number of nodes) in P_Σ is bounded by the upper bounds given in the theorem statement, given a bound on n . From this, an application of Corollary 3 then gives us the desired results for (1) and (2). We will prove (3) afterwards.

For (1), let $n = m!$ and consider that q starts in transaction $\tau_i \in T$ and ends in transaction $\tau_j \in T$, we continue by considering a number of cases. We start by excluding the case $\tau_i = \tau_j$, since then the longest path q can be at most of length ℓ . Since $\ell \leq n\ell - (n-1)(m-2)$, it follows that q is bounded by $n\ell - (n-1)(m-2)$.

In Case $\text{seq}(\tau_i) = \text{seq}(\tau_j)$: The number of nodes on path q has the following number of operations:

$$\begin{aligned} & \text{pre}(\tau_i) + \max\{\delta_{(1,2)}^k \mid k \in [i, j]\} + \dots + \\ & \max\{\delta_{(m-1,m)}^k \mid k \in [i, j]\} + \text{suf}(\tau_j) + j - i + m, \end{aligned}$$

where $\delta_{(u,u+1)}^k = \text{cpos}(\tau_k, u + 1) - \text{cpos}(\tau_k, u)$, given $k \in [i, j]$ and $u \in [1, m - 1]$. In the worst case, we have the following:

- $j = n$ and $i = 1$;
- for τ_i , either $\text{pre}(\tau_i) = \ell - m$, or $\text{pre}(\tau_i) = 0$ with $u \in [m - 1]$ such that $\delta_{u,u+1}^i = \ell - m$;
- for τ_j , either $\text{suf}(\tau_j) = \ell - m$, or $\text{suf}(\tau_j) = 0$ with $u \in [m - 1]$ such that $\delta_{u,u+1}^j = \ell - m$;
- for each $k \in [i + 1, j - 1]$ there is precisely one $u \in [m - 1]$ such that $\delta_{u,u+1}^k = \ell - m$.

Therefore, we get the following bound on the number of operations:

$$n(\ell - m) + n + m - 1 = n\ell - mn + m + n - 1.$$

This bound is lower than $n\ell - (n - 1)(m - 2)$, since:

$$\begin{aligned} n\ell - mn + m + n - 1 &\leq n\ell - (n - 1)(m - 2) && \Leftrightarrow \\ n\ell - mn + m + n - 1 &\leq n\ell - [mn - 2n - m + 2] && \Leftrightarrow \\ n\ell - mn + m + n - 1 &\leq n\ell - mn + 2n + m - 2 && \Leftrightarrow \\ n - 1 &\leq 2n - 2. \end{aligned}$$

It follows that q 's number of nodes is bounded by $n\ell - (n - 1)(m - 2)$ when this is the case.

In case $\text{seq}(\tau_i) \neq \text{seq}(\tau_j)$: Here, q goes through a chain of multiple contention point orders $T_{\mathbf{x}^1}, T_{\mathbf{x}^2}, \dots$ where $\{\mathbf{x}^1, \mathbf{x}^2, \dots\} \subseteq K$. Due to Lemma 4, we know that $\mathbf{x}_{(2)}^i = \mathbf{x}_{(1)}^{i+1}$ for all $i \in [|K|]$, therefore, the path q has the following operations:

$$\begin{aligned} & \text{pre}(\tau_i) + \delta_{(1,2)}^i + \delta_{(1,2)}^{i+1} + \dots + \delta_{(1,2)}^{j-1} + \\ & \max\{\delta_{(u,u+1)}^j \mid u \in [1, m - 1]\} + \text{suf}(\tau_j) + 2(n - 1) + m \end{aligned}$$

Note that the first $n - 1$ transactions on q access the operations $W[\mathbf{x}_{(1)}]$ and $W[\mathbf{x}_{(2)}]$, hence a contribution of $2(n - 1)$ to the path cost. The final transaction accesses all m operations on contention points. In the worst case, we get that:

- $i = 1$ and $j = n$;
- $\delta_{(1,2)}^k = \ell - m$ for $k \in [i, j - 1]$, and therefore $\text{pre}(\tau_i) = 0$;
- $\max \{ \delta_{(u,u+1)}^j \mid u \in [1, m - 1] \} = \ell - m$, and therefore $\text{suf}(\tau_j) = 0$.

Therefore, we get the following bound:

$$\begin{aligned}
 & (n - 1)(\ell - m) + (\ell - m) + 2(n - 1) + m = \\
 & n\ell - nm - \ell + m + \ell - m + 2n - 2 + m = \\
 & n\ell + 2n - nm + m - 2 = \\
 & n\ell + 2n - nm + m - 2 = \\
 & n\ell - [nm - 2n - m + 2] = \\
 & n\ell - (n - 1)(m - 2).
 \end{aligned}$$

Hence q is of node length at most $n\ell - (n - 1)(m - 2)$. This concludes the proof for (1).

For (2), let $n > m!$ and let q start in $\tau_i \in T$ and end in $\tau_j \in T$. Again we do not consider the case where $\tau_i = \tau_j$, as then the longest path is trivially length ℓ . Hence, the cases that remain are either $\text{seq}(\tau_i) = \text{seq}(\tau_j)$, or $\text{seq}(\tau_i) \neq \text{seq}(\tau_j)$.

In case $\text{seq}(\tau_i) = \text{seq}(\tau_j)$: The length of q is given by:

$$\text{pre}(\tau_i) + (m - 1)(\ell - (m + 1)) + \text{suf}(\tau_j) + (j - i + 1) \quad (4.9)$$

That is, the path starts in the prefix of τ_i , visits the maximal space between $\mathbf{x}_{(1)}$ and $\mathbf{x}_{(2)}$, visits the next maximal space between $\mathbf{x}_{(2)}$ and $\mathbf{x}_{(3)}$, and so on, until a transaction that visits the maximal space between $\mathbf{x}_{(m-1)}$ and $\mathbf{x}_{(m)}$ (with $\mathbf{x} = \text{seq}(\tau_i) = \text{seq}(\tau_j)$). Finally, the path ends in the suffix of τ_j .

First, we know that $\text{pre}(\tau_i) \leq \ell - m$ and $\text{suf}(\tau_j) \leq \ell - m$. Next, each of the transactions τ either traverses through a contention point contributing 1, or it traverses through one contention point plus some maximal space, contributing $1 + \text{cpos}(\tau, i + 1) - \text{cpos}(\tau, i)$ a

grand total of $(m - 1)$ times. This overall quantity is bounded by $(m - 1)(\ell - (m + 1)) + (j - i + 1)$ or, in the worst case when $j = n$ and $i = 1$, we get $(m - 1)(\ell - (m + 1)) + n$. Hence, we can rewrite equation 4.9 as follows:

$$\begin{aligned}
& 2(\ell - m) + (m - 1)(\ell - (m + 1)) + n \\
&= 2(\ell - m) + (m - 1)(\ell - m - 1) + n \\
&= 2\ell - 2m + (m - 1)\ell - (m - 1)m - (m - 1) + n \\
&= (m + 1)\ell + n - 2m - m^2 - 2m - 1 \\
&= (m + 1)\ell + n - m^2 - 4m - 1.
\end{aligned}$$

This bound is lower than $(m! + m - 2)\ell + n - (m - 1)(m! + m - 2)$, hence the path is bounded appropriately in this case.

In case $\text{seq}(\tau_i) \neq \text{seq}(\tau_j)$: The node length of q is given by:

$$\begin{aligned}
& \text{pre}(\tau_i) + 1 + \delta_{1,2}(\tau_i) \\
&+ (m! - 2)(\ell - (m + 1)) + (m - 2)(\ell - (m + 1)) \\
&+ \delta_{m-1,m}(\tau_j) + 1 + \text{suf}(\tau_j) + (j - i + 1)
\end{aligned}$$

where $\delta_{x,y}(\tau) = \text{cpos}(\tau, y) - \text{cpos}(\tau, x)$. We justify all the operations that go rightwards (and not downwards into the next transaction as those are accounted for by $(j - i + 1)$). The path starts in τ_i and visits up to and including $\mathbf{W}[\mathbf{x}_{(2)}]$, since we know all transactions overlap by $m - 2$ (c.f., Lemma 5). We then repeatedly visit the next grouping, which contributes $(\ell - m + 1)$ a total of $(m! - 2)$ times. We then end up in the last grouping which contributes $(m - 2)(\ell - (m + 1))$ for all the spaces between $\mathbf{x}_{(1)}$ and $\mathbf{x}_{(2)}$, between $\mathbf{x}_{(2)}$ and $\mathbf{x}_{(3)}$, and so on until $\mathbf{x}_{(m-2)}$ and $\mathbf{x}_{(m-1)}$. And finally, we visit $\delta_{m-1,m}(\tau_j)$ operations and end with $1 + \text{suf}(\tau_j)$.

In the worst case, we have $j = n$ and $i = 1$. And, we have that $\text{pre}(\tau_i) = 0$ and $\delta_{1,2}(\tau_i) = \ell - m$; and furthermore that $\text{suf}(\tau_j) = 0$ and $\delta_{m-1,m}(\tau_j) = \ell - m$, due to Σ . This means we can rewrite the above expression as:

$$\begin{aligned}
& m!(\ell - (m + 1)) + (m - 2)(\ell - (m + 1)) + n \\
&= (\ell - m - 1)(m! + m - 2) + n \\
&= (m! + m - 2)\ell - (m - 1)(m! + m - 2) + n
\end{aligned}$$

This bound is $(m! + m - 2)\ell + n - (m - 1)(m! + m - 2)$, showing (2).

For (3), observe that the Steinhaus-Johnson-Trotter algorithm used on line 2 of AS_m runs in time $O(m!)$, since it has to generate all possible permutations on m , and then subsequently iterates over all of them. Next, for each of these permutations we have to sort all transactions within (c.f., lines 4–10) which can be done in time $O(m \cdot n \log n)$ (since we need to sort $m - 1$ times). Furthermore, we do this again for the final contention point sequence on lines 11–18. We then concatenate the transactions without contention points at the back, which can be done in $O(n)$ and finally, we output the canonical schedule, which can be computed in $O(p^2)$ where $p = \sum_{\tau \in T} |\text{ops}(\tau)|$. Hence, this gives the overall bound of $O(m! + p^2 + mn \log n)$. This concludes the proof. $\square$

Lastly, if this were an approximation algorithm, then in the case that $n = m!$ we would be able to achieve a tight approximation due to the following result.

Theorem 10. There exists a $T \in \mathbb{T}_{\ell, m, n}$ with $m \geq 4$, $\ell \geq m$ and $n = m!$ such that $\text{opt}(T) = n\ell - (n - 1)(m - 2)$.

Proof. If T contains n transactions of length ℓ such that each transaction has a different order of contention points, and such that $n = m!$, then Lemma 5 says we have a maximal total overlap of $(n - 1)(m - 2)$ for $K = \{\text{seq}(\tau) \mid \tau \in T\}$. We can therefore claim that $\text{opt}(T) = n\ell - (n - 1)(m - 2)$ if it is the case that none of the non-conflicting operations $\ell - m$ are distributed inbetween any of the operations on contention points $W[\mathbf{x}_{(2)}], W[\mathbf{x}_{(3)}], \dots, W[\mathbf{x}_{(m)}]$ (with $\mathbf{x} \in K$); otherwise the makespan will be lower than $n\ell - (n - 1)(m - 2)$ as more of the $\ell - m$ non-conflicting operations will be part of the region that overlaps. We can define a T for which the former is the case, namely:

$$\{\langle W_i[\mathbf{x}_{(1)}], R_i[u_{i,1}], \dots, R_i[u_{i,\ell-m}], W_i[\mathbf{x}_{(2)}], \dots, W_i[\mathbf{x}_{(m)}] \rangle \mid \mathbf{x} \in K, i \in [n]\}.$$

The desired result then follows. $\square$

5 Summary

In this chapter we looked at different approximation algorithms for scheduling transactions. In the process of doing this, we also derived worst-case lower bounds and upper bounds on transaction sets that satisfy certain properties.

Summary of Bounds and Algorithms

In the following table, what we mean by a worst-case lower bound (denoted by $\uparrow$) is that there exists a transaction set which has that bound as an optimal makespan. Furthermore, by an *upper bound* (denoted by $\downarrow$), we mean that there exists an algorithm which yields a schedule of at most that makespan. When neither the symbols $\uparrow$ or $\downarrow$ are present, it means this is both a worst-case lower bound and upper bound.

	$m = 0$	$m = 1$	$m = 2$	$m = 3$	$4 \leq m$
$n = 1$	ℓ	ℓ	ℓ	ℓ	ℓ
$n = 2$		$\ell + 1$	2ℓ	2ℓ	2ℓ
$3 \leq n < m!$	$3 \leq m!$	$3 \leq m!$	$3 \leq m!$	$n\ell - n + 1$	$n\ell - n + 1 \downarrow$
$n = m!$		ℓ	2ℓ		$n\ell - (n-1)(m-2)$
$m! < n < m!m$	ℓ	$\ell + n - 1$	$2\ell + n - 2$	$7\ell + n - 14 \downarrow$ $6\ell + n - 11 \uparrow$	$(m! + m - 2)\ell + n - (m-1)(m! + m - 2) \downarrow$
$m!m \leq n < (m+1)!$				$7\ell + n - 14 \downarrow$ $6\ell + \lfloor \frac{\ell-3}{2} \rfloor + n - 11 \uparrow$	
$(m+1)! \leq n$				$7\ell + n - 14 \downarrow$ $7\ell + n - 15 \uparrow$	

Table 4.1 – All bounds derived in this chapter given that $\ell \geq m$.

The values for $\mathbb{T}_{\ell, m, 1}$, $\mathbb{T}_{\ell, 0, n}$, and $\mathbb{T}_{\ell, 1, n}$ in this table haven't been discussed in prior sections, but this cost is straightforward to determine. Indeed, if $n = 1$ the makespan of all schedules is simply ℓ . Similarly, if $m = 0$ we can always execute all transactions entirely in parallel, hence the bounds are also ℓ . Lastly, if $m = 1$, the lone operation on the contention point occurs in the same position in every transaction in all input instances T for which $\text{opt}(T)$ is maximal. An optimal schedule will shift each operation in each consecutive transaction forwards by one time unit, hence we have the bound $\ell + n - 1$ given $n \geq 2$.

Similarly, the bound for $\mathbb{T}_{\ell, m, 2}$ is also straightforward. The upper bound can be obtained by scheduling both transactions serially in which case we get a makespan of 2ℓ . The worst-case lower bound is essentially the same as the worst-case lower bound for the transactions in $\mathbb{T}_{\ell, 2, 2}$, since we only need transactions $\langle W[a], \dots, W[b] \rangle$ and $\langle W[b], \dots, W[a] \rangle$ where the operations in-between (*i.e.*, the ‘...’) can be arbitrary, *including other operations on contention points*. Hence, if we add any more operations, it does not make the bound higher.

In the table below, we summarize the algorithms we have introduced thus far. The “Algorithm Type” column indicates whether the algorithm is exact or an approximation. The α column gives the approximation ratio, and the “Ref.” points to the definition site.

m	Algorithm Type	α	Tight?	Time Complexity	Ref.
0	Exact	1	Yes	$O(p^2)$	See Ahead
1	Exact	1	Yes	$O(n^2 \log n + p^2)$	
2	Approx.	$\frac{n-2}{\ell} + 2$	Yes	$O(p^2 n \log n)$	Alg. 8
3	Approx.	$\frac{1-n}{\ell} + n$ if $n \in [3, 6]$ $\frac{n-14}{\ell} + 7$ if $n \geq 7$	Yes No		Alg. 9
≥ 4	Approx. ($\notin P$)	$\frac{(m-2)(1-n)}{\ell} + n$ if $n = m!$ $O(n/\ell + \ell!)$ if $n > m!$	Yes No	$O(m! + p^2 + mn \log n)$	Alg. 10
	Exact ($\notin P$)	1	Yes	$O(n! \cdot p^2 + p^2)$	Alg. 3

Table 4.2 – Summary of all algorithms introduced in this part of the thesis; the parameter p is defined as $\sum_{\tau \in T} |\text{ops}(\tau)|$.

The algorithm for the $m = 0$ case is essentially to sort T arbitrarily and construct a canonical schedule, hence the $O(p^2)$ bound on running time.

Furthermore, for $m = 1$, we have to repeatedly apply Algorithm 4 for values of $k \in [\ell, \ell + n]$. Hence, the overall running time becomes $O((\ell + n - \ell)n \log n) = O(n^2 \log n)$.

Finally, for the approximation ratio of Algorithm 10, in case that $n = m!$, we get $\alpha = (n\ell - (n-1)(m-2))/\ell$ which can be rewritten into $((m-2)(1-n)/\ell) + n$. And, for $n > m!$ we get the following bound on

α (recall that $\ell \geq m$):

$$\begin{aligned}
& \frac{(m! + m - 2)\ell + n - (m - 1)(m! + m - 2)}{\ell} \leq \\
& \frac{(m! + \ell - 2)\ell}{\ell} + \frac{n}{\ell} - \frac{(\ell - 1)(m! + \ell - 2)}{\ell} = \\
& m! + \ell - 2 + \frac{n}{\ell} + \frac{m!}{\ell} - m! - \ell - \frac{2}{\ell} + 3 = \\
& \frac{n}{\ell} + \frac{m!}{\ell} - \frac{2}{\ell} + 1 \leq \\
& \frac{n}{\ell} + \frac{\ell(\ell - 1)!}{\ell} - \frac{2}{\ell} + 1 = \\
& \mathcal{O}\left(\frac{n}{\ell} + \ell!\right).
\end{aligned}$$

Further Reading

The material in this section originates partially from the ICDT 2026 paper by Brecht Vandevoort, Bas Ketsman, and myself titled “*Bounding the Makespan of Transaction Schedules*” [14]. The paper establishes the bounds presented here; the conversion of those bounds into approximation algorithms is new to this chapter.

For background material on approximation algorithms, the book “*Algorithmics for Hard Problems*” by Hromkovič was consulted [46].

Related Work

1 Related Theory

Serializability and Schedulability

For years, serializability has been the gold standard for ensuring data consistency in databases. In brief, we recall from Section 2.2 that a transaction schedule is serializable if it is equivalent (given some definition of equivalence) to a serial execution of the given set of transactions.

One natural equivalence notion is view equivalence, which states that two schedules are equivalent if all read operations observe the same values and the final database state is the same for both schedules. A famous result by Papadimitriou [71] states that view serializability is NP-complete to decide. Although view serializability is a natural notion of serializability, this negative result has led to the adoption of conflict serializability as the de facto standard for serializability in practice, and is frequently simply referred to as “serializability”.

Contrasting view equivalence, conflict equivalence puts a more direct restriction on the order of operations. In particular, all pairs of conflicting operations must be ordered the same in both schedules. It is well known that conflict serializability of a schedule can be decided in polynomial time (as we have demonstrated with Algorithm 1), and that every conflict serializable schedule is view serializable as well [72]. The core observation is that the serializability problem is orthogonal to the scheduling problem. The former is concerned with deciding serializability for a given schedule, while the latter is concerned with finding the most efficient serializable

schedule for a given set of transactions.

To the best of our knowledge, the only prior results on scheduling are also from Papadimitriou [71], who established that a scheduler which allows precisely all view serializable schedules will likely be inefficient unless $P = NP$. This result is to be expected, given the NP-completeness of deciding view serializability. Contrasting these results by Papadimitriou, the results presented in this thesis restrict the search space of schedules to those that are conflict serializable, a property which can be tested in polynomial time. The NP-completeness result for transaction scheduling (Theorem 3) therefore strengthens this earlier work, and shows that a tractable notion of serializability does not necessarily lead to tractable scheduling algorithms.

The Robustness and Allocation Problems

Orthogonal to finding more efficient schedules that guarantee (conflict) serializability, database systems frequently offer lower levels of isolation, such as read committed or snapshot isolation. In fact, many commercial database systems offer these lower isolation levels by default, and some do not even support conflict serializability [15]. The relaxed consistency guarantees of these lower isolation levels allow for more concurrency, and therefore higher throughput. This performance increase comes at a cost, however, as these lower isolation levels no longer guarantee serializability. Recent work from the database theory community has looked into the transactional robustness problem [37, 54, 55, 89].

The robustness problem is concerned with the following question: given a set of transactions and a specific isolation level, is every allowed schedule (i.e., consistent with the isolation level) a serializable schedule? This robustness property guarantees that the set of transactions can be safely executed under the lower isolation level, thereby increasing performance, while ensuring serializability.

The allocation problem [37, 91] extends upon this problem by considering a setting where isolation levels can be mixed (i.e., different transactions are allocated to different isolation levels), and tries to identify the optimal allocation that still guarantees serializability for the given workload.

We emphasize that this line of work relies on an existing scheduler. A key assumption is that, although this scheduler is guaranteed to produce a schedule in line with the chosen isolation levels, the produced output schedule cannot be chosen. Instead, robustness involves verifying serializability of every possible output schedule. The robustness and allocation problems are orthogonal to the scheduling problem, as they rely on an existing scheduler to execute transactions, rather than directly constructing an efficient serializable schedule over the provided workload.

The Job Shop Scheduling Problem

Outside of the database literature, the job shop scheduling problem [20, 73] is the closest relative to the transaction scheduling problem. Indeed, one can see objects as machines, transactions as jobs, and operations as always lasting a single unit of time. The associated decision problem is similarly known to be NP-complete, even when limited to 3 machines [82], known to be approximable only if the operations are unit-time [57, 63], and has a polynomial time approximation scheme (PTAS) if the maximal job length and the number of machines is fixed [40]. However, the results do not carry over. The job shop scheduling problem is distinct from transaction scheduling because its resulting schedules are not conflict serializable.

Lastly, the polynomial time algorithm for scheduling transaction sets with a single contention point, given in Section 3.3, resembles a polynomial-time algorithm for the Job Shop Scheduling problem called *Johnson's Rule* [50]. We illustrate it is subtly different.

Example 10. Job shop problem inputs consist of jobs (analogous to our transactions), containing a sequence of operations lasting a duration $x \in \{1, 2, \dots\}$ on machines (analogous to our objects). Johnson's rule is applicable when all jobs have two operations on two machines in a single order, hence we represent jobs as pairs (x, y) where x and y are the durations spent on the first and second operation.

Consider the job shop instance

$$\mathcal{J} = \{A = (1, 3), B = (1, 2), C = (2, 1)\}.$$

Johnson's rule repeatedly picks the operation with the shortest duration and places it in the first free position starting from the front of the schedule if that duration is for the first machine, and places it in the first free position starting from the end of the schedule if it is for the last machine, while breaking ties arbitrarily. A valid schedule would be B, A, C .

If we now create transactions $\tau_J \in T$ for each job $(x, y) \in J$ such that $\text{pre}(\tau_J) = x$ and $\text{suf}(\tau_J) = y$, then the allocation for T matching the order B, A, C is $\alpha_5(\tau_B) = 1$, $\alpha_5(\tau_A) = 2$, and $\alpha_5(\tau_C) = 3$. However, an allocation α_4 for T is given by our algorithm, namely, $\alpha_4(\tau_A) = 1$, $\alpha_4(\tau_B) = 2$, $\alpha_4(\tau_C) = 3$. Hence, Johnson's rule does not give an optimal schedule for the transaction scheduling problem.

2 Related Systems

The systems most closely related to our work process transactions in batches, which entails that the input to the transaction processing system is a set of multiple transactions. This most closely corresponds to the formal models that most of the theory of transactions considers. Batching allows us to perform dynamic analysis based on the conflicts present in the batch, and this typically increases throughput when the cost of analysis is lower than the cost to outright process the batch without performing analysis first. These kinds of systems are more commonly known in the database literature as *deterministic transaction processing systems* [9, 25, 32, 34, 36, 45, 58, 60, 61, 65, 67, 78–80, 86, 100].

These systems were first studied around the turn of the century [49, 66], when distributed computing at the data-center scale was starting to become more prevalent. Determinism in this setting means that the system, in some way, deterministically decides the order in which transactions are executed by performing dynamic analysis on incoming transactions. Precisely what this dynamic analysis entails or how execution is handled varies wildly from system to system. Since this early work, it has become an incredibly active domain. We point to a number of survey articles for general context [1, 85], and now highlight relevant subcategories.

Scheduling Systems

The only system doing explicit scheduling based on makespan, to the best of our knowledge, is the system from Cheng et al. [24]. This demonstrates that scheduling is a viable approach in practice.

The system relies on a heuristic that clusters transactions based on the most frequently occurring contention point. The work introduces the shortest makespan first (SMF) heuristic, which relies on placing transactions on hot keys far apart in the schedule, instead of relying on the full read and write set as we do in our formal model. Their approach is to train a KNN-classifier to recognize hot objects, and it uses this hot set to make scheduling decisions. The KNN-classifier is trained on traces that previously executed on the system, and it has to be retrained periodically. Furthermore, they make the observation that focusing on hot keys for scheduling provides the greatest benefit, *which we formally prove by showing that the fragment of single contention point transaction sets (i.e., a single hot object) can be scheduled optimally in polynomial time*. The results show that their approach performs well on workloads that have high contention for a small set of shared objects. For workloads with lower contention, the approach is on par with other systems. Hence, the overhead for this approach is manageable.

The work on lock scheduling by Huang et al. [47] and Tian et al. [87] focuses on scheduling that minimizes latency to acquire locks. Their heuristics focus on giving transactions sets with a larger set of dependencies priority over other sets. Related to this is the work by Wang et al. [94] which studied scheduling strategies which specifically minimize lock thrashing events.

Cao et al. [21] have investigated the complexity of a blend between partitioning and scheduling. Their implementation works with existing partitioning algorithms such as [75, 97]. In summary, their algorithm starts from an existing partition, and then subsequently weaves transactions from the leftover residual set (which still has conflicts with one or more subsets in the partition) into the conflict-free disjoint subsets of the partition. Their approach can still end up with leftover transactions in the residual set. In such cases, *our general formulation of transaction scheduling is complementary and can be used to explicitly schedule this residual set*.

Partitioning Systems

Partitioning systems [16, 21, 35, 52, 75, 76, 97, 98, 101, 103] work by partitioning the batch into disjoint sets of transactions that conflict on the same database objects, thereby avoiding conflicting operations between the different sets. Each of these sets is assigned to a single core and each transaction within them is executed one by one. Because there are no conflicts between the sets, we can do this concurrently, without the need for concurrency control. We can find such a partition by choosing it beforehand, based on the dataset and workload [52], or by computing which partition minimizes runtime conflicts [21, 75, 97].

The worst case behavior of batch-partitioning is triggered when the entire batch ends up in a single set. Typically, the system will fall back to online concurrency control [21, 75] such as two-phase locking or multi-versioning in this scenario, instead of executing everything on one core. *Alternatively, one could use our algorithms to sort this remaining set.*

Single-Threaded Systems

TigerBeetle¹ is a special-purpose financial transaction processing database. It is not a traditional general-purpose relational database in the sense that it implements a hard-coded set of transactions and only stores numeric data. Furthermore, one of its defining features is that it runs single-threaded. Additional workers are only there for replication purposes, and are kept in sync using View-stamped Replication [69]. One of the main drivers making this performant is the choice to batch transactions. While it does no scheduling, it still manages to outright outperform most other databases for the same kinds of workloads according to their benchmarks². It would be interesting to study whether explicitly scheduling using one of our algorithms could increase throughput.

¹<https://tigerbeetle.com>

²<https://www.infoq.com/presentations/redesign-oltp/>

Part II

Queries

We now turn our attention to the second part of this thesis. Recall from the introduction that we are concerned with the problem of whether a computational problem can (or can not) be computed in a manner that does not involve a coordination protocol. One of the first positive results in this area is the Consistency as Logical Monotonicity (CALM) Theorem, which states that a computational problem has a consistent and coordination-free implementation, if and only if, it is logically monotone.

We will address two main limitations of prior work on this theorem. First, we will make the only-if direction of the theorem less delicate. We will update the formal model of computation used in the proof of the theorem to allow for arbitrary system configurations. Secondly, the prior work only relates to computational problems that may be formulated as *deterministic queries*, which map a single input to a single output. Instead, we will allow the model to also express non-deterministic computational problems. We then study how powerful the only-if direction has to be in order to express non-determinism. This is the central topic we start with in Chapter 6 and continue in Chapter 7.

Finally, we will apply this updated model to derive new insights about coordination freedom in Chapter 8. We show that this new model can express prior work on CALM, and we use this to investigate the question of whether all queries that are polynomial-time computable are also coordination-free computable. We also provide a coordination hierarchy, which demonstrates how expressive certain models are with respect to one another.

Transducer Networks

We introduce Distributed Behaviors and Generalized Transducer Networks, which are respectively a way of semantically modeling distributed computational problems and a computational model for shared-nothing distributed systems. We first work up to the definition of a distributed behavior, which is a special kind of non-deterministic queries that allow for explicitly modeling the location of facts. Then, we introduce Generalized Transducer Networks along with run semantics.

1 Preliminaries

Schemas, Instances, and Mappings

We assume an infinite set **dom** called the *domain* of data values. A *schema* σ is a finite set of relation names R associated with an arity $ar_\sigma(R) \in \mathbb{N}$ and a label $acc_\sigma(R) \in \{input, output\}$. With this label we define a primitive access constraint on relations, indicating that a relation can be accessed in the normal way (*output*) or only through membership testing (*input*).¹ We elaborate on its use further on in this thesis. When σ is clear from the context, we write $ar(R)$ and $acc(R)$ without index. As a shorthand, we write $R[k]$ to mean $R \in \sigma$ with $ar_\sigma(R) = k$ and only write $acc_\sigma(R)$ explicitly if $acc_\sigma(R) = input$.

We say that $\mathbf{f} = R(t_1, \dots, t_k)$ is a *fact* over schema σ if $R \in \sigma$, $ar(R) = k$, and $(t_1, \dots, t_k) \in \mathbf{dom}^k$. An *instance* I over σ is a finite set of facts over σ . The set of all instances over σ is denoted by $\text{inst}(\sigma)$. For an instance I and schema σ , we use $I|_\sigma$ to denote the instance obtained

¹Notice that we use the same terminology as [68], although applied to entire relations instead of on individual attributes.

by removing all facts from I that are not over σ . We write $\text{Adom}(I)$ to denote the *active domain* of I which is the set of data values occurring in I in relations R with $\text{acc}(R) = \text{output}$.

Let σ_1 and σ_2 be disjoint schemas. A *mapping* M from σ_1 to σ_2 is a (left-)total relation $M \subseteq \text{inst}(\sigma_1) \times \text{inst}(\sigma_2)$. We call σ_1 the *input schema* and σ_2 the *output schema* of M .

We will assume that M is *domain-preserving*, i.e., for every $(I, J) \in M$ we have $\text{Adom}(J) \subseteq \text{Adom}(I)$; and that M is *generic*, i.e., for every bijective renaming $\beta : \mathbf{dom} \rightarrow \mathbf{dom}$ and every instance I over σ_1 and instance J over σ_2 , we have that $(I, J) \in M$ if, and only if $(\beta(I), \beta(J)) \in M$. From now on we will only consider domain-preserving generic mappings and therefore simply call them mappings. We say that a mapping M from σ_1 to σ_2 is *deterministic* when for every instance I over σ_1 there exists exactly one J over σ_2 such that $(I, J) \in M$.

Networks, Distributed Schemas and Instances

A *network* $\mathcal{N}$ is a finite set whose elements we call *node identifiers* (or simply *nodes*).

In order to model the distribution of facts from an instance (over some schema σ) over the nodes of a network $\mathcal{N}$, we define the *distributed extension* of a schema σ , denoted $\text{dist}(\sigma)$, to be the schema obtained by increasing all arities of relations in σ by one. A *distributed fact* over σ then is as a fact $\mathbf{f} = R(x, \bar{a})$ over $\text{dist}(\sigma)$ and a *distributed instance* H over schema σ is an instance over $\text{dist}(\sigma)$. Intuitively, a distributed fact $\mathbf{f} = R(x, \bar{a})$ signifies that a fact $R(\bar{a})$ over σ is located on node x . Henceforth, we will often write $R@x(\bar{a})$ instead of $R(x, \bar{a})$ to highlight the special meaning of x . Notice that this notation is consistent with the literature [2, 59]. We will also write $\text{netw}(\mathbf{f})$ to refer to x , and $\text{netw}(H)$ to denote the set of all node identifiers in facts of the distributed instance H .

Sometimes it will be necessary to convert distributed instances to (regular) instances, in which case we write $\text{flat}(H)$ for a distributed instance H over some distributed schema $\text{dist}(\sigma)$ to denote the instance over σ obtained by removing the first positions (which store node identifiers) of all facts in H . For a node $x \in \text{netw}(H)$ we will

write $H(x)$ to denote the instance over σ located on x ,

$$H(x) := \{R(\bar{a}) \mid R(x, \bar{a}) \in H\},$$

thus slightly abusing notation and pretending that H is a mapping from $\text{netw}(H)$ to $\text{inst}(\sigma)$ instead of an instance.

As a convention, we will refer to (non-distributed) instances with letters I and J , and to distributed instances with the letters H , K and U .

Distributed Behaviors

Lastly, we define distributed behaviors.

Definition 20. A (*distributed*) *behavior* B from σ_1 to σ_2 is a mapping from $\text{dist}(\sigma_1)$ to $\text{dist}(\sigma_2)$ with the additional property that $(H, K) \in B$ implies $\text{netw}(K) \subseteq \text{netw}(H)$, for every distributed instance H over σ_1 and K over σ_2 .

We call two behaviors B from σ_1 to σ_2 and B' from σ'_1 to σ'_2 *equivalent* if there exists a bijection β from $\sigma_1 \cup \sigma'_1$ to $\sigma_2 \cup \sigma'_2$ such that $\sigma_2 = \beta(\sigma_1)$, $\sigma'_2 = \beta(\sigma'_1)$ and $(H, K) \in B$ if, and only if,

$$(\{\beta(R)(\bar{a}) \mid R(\bar{a}) \in H\}, \{\beta(R)(\bar{a}) \mid R(\bar{a}) \in K\}) \in B'.$$

That is, B and B' are equivalent if they are identical up to relation renaming.

We now formally define the behavior described in Example 1.

Example 11. For every schema σ , we define $L(\sigma)$ as the behavior with input schema σ and output schema $\sigma' := \{\text{Leader}[1]\}$, such that $(H, K) \in L(\sigma)$ if H and K satisfy:

1. For every $x \in \text{netw}(H)$ there exists a $y \in \text{netw}(H)$ such that $\text{Leader}@x(y) \in K$, and
2. For every $\text{Leader}@x(y), \text{Leader}@z(w) \in K$, it holds that $y = w$.

Recall from earlier that $\text{Leader}[1]$ is notation for a relation Leader with arity 1. Intuitively, each node x has a $\text{Leader}@x(y)$ -fact and the leader node y is the same for all nodes $x \in \text{netw}(H)$.

2 Generalized Transducer Networks

As our model of computation, we introduce *generalized transducer networks*, which are based on transducer networks [8] but without restrictions on the system schema, and instead specifying a constraint that explicitly relates input instances to system instances. Furthermore, the nodes in our model communicate by broadcasting to all nodes whereas the model in [8] communicates by sending messages to its neighbours according to a network topology. We instead model this network topology using constraints (*cf.*, Example 16). We do this to more explicitly model that the network topology is state which is shared between the nodes (even though nodes may not explicitly store it). We proved that our model is a generalization of the earlier relational transducer network models in the appendix of [13].

Relational transducer networks are a model for asynchronous and shared-nothing distributed systems. What makes it distinctive is that computing nodes are defined declaratively using relational state machines. Concretely, we provide a declarative definition for a single program (*i.e.*, the transducer), of which multiple identical copies run on each of the nodes in the system. We program transducers using mappings, which can be seen as non-deterministic relational programs. We do not specify a concrete syntax or language (*e.g.*, Datalog, SQL, ...), and we instead define such mappings purely based on (computable) semantics with the goal of keeping the model general.

All nodes have their own state which is not shared with the other nodes (*i.e.*, shared-nothing). This local state is modeled as a relational database with set-based semantics, hence, each stored tuple appears at most once. Nodes do not have direct access to the state of other nodes, and instead exchange state by sending messages through an asynchronous “oblivious broadcast” mechanism.

Nodes can send a message only by broadcasting to all other nodes through an abstract network layer. In practice, such a network layer can be a distributed message queue (*e.g.*, Apache Kafka, ZeroMQ, ...). From the perspective of the model, however, we only care about the fact that message sends are broadcasts, and that we cannot

directly send point-to-point messages to a single node. Furthermore, we assume that messages can arrive multiple times, out of order, and with an arbitrary finite delay. In concrete systems, a finite delay can be achieved by restarting nodes after they crash, and then broadcasting any messages that were potentially dropped during the crash (*i.e.*, at-least-once delivery guarantees). In the definition we will later see that the network's abstract message buffer is modelled using a multiset to allow for message duplication, while a node's local state is modelled using a set.

Generalized Relational Transducers

A transducer is defined with respect to a transducer schema. A (*generalized*) *transducer schema*

$$Y = (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{mem}}, Y_{\text{msg}}, Y_{\text{sys}})$$

is a tuple of disjoint relation schemas, defining the schema of the input instance (Y_{in}), the output instance (Y_{out}), the local memory instance (Y_{mem}), the messages (Y_{msg}), and a system instance (Y_{sys}).

Definition 21. We define a (*generalized, relational*) *transducer* over Y as a tuple $\Pi = (C, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$, with:

- A configuration constraint C (whose definition we postpone to the next section) from Y_{in} to a behavior from Y_{in} to Y_{sys} ; and
- The *output mapping* M_{out} from Y_{all} to Y_{out} ; and
- An *insert mapping* M_{in} and *delete mapping* M_{del} from Y_{all} to Y_{mem} ; and
- The *send mapping* M_{snd} from Y_{all} to Y_{msg} ;

where $Y_{\text{all}} := Y_{\text{in}} \cup Y_{\text{out}} \cup Y_{\text{mem}} \cup Y_{\text{sys}}$. When we wish to make the configuration constraint C explicit, we call Π a C -transducer.

Intuitively, the mappings define how the local state of an individual computing node should be updated based on the current local state and newly received messages. More precisely, M_{out} specifies which facts are added to the output instance; M_{in} and M_{del} specify which facts are inserted into and deleted from memory, respectively; and M_{snd} specifies which messages are sent as side effect of the state change. The input instance over Y_{in} as well as the system

instance over Y_{sys} are fixed at the start of a computation and cannot be changed. The output relations are *inflationary*, a transducer can insert tuples into them, and tuples can never be removed from them. Hence, the state only grows or ‘inflates’.

Configuration Constraints

The purpose of the earlier mentioned configuration constraint is to map any input instance over Y_{in} , which is distributed over nodes of the system, to specific instances over Y_{sys} (on those nodes) that we consider valid configurations of the system. Hence, it constrains the system to be in a valid configuration. The system relations are there to model “system state”, such as a data distribution policy, which node is a leader, a list of all nodes in the system, and so on. In earlier work [7, 8, 104], this relationship was fixed through an ad-hoc definition; instead, we make it a central parameter of the model.

Formally, a configuration constraint is defined as follows.

Definition 22. A (configuration) constraint $\mathbf{C}$ is a total function mapping every schema σ to a behavior from σ (to some schema σ' that can be specific to σ).

We will say that a configuration constraint is *compatible* with a transducer schema Y if $\mathbf{C}(Y_{\text{in}})$ is a behavior from Y_{in} to Y_{sys} , and from now on require that a transducer is always parameterized with a constraint that is compatible with its schema. We will speak of a *C-transducer* to make $\mathbf{C}$ explicit.

Example 12. For examples of constraints, consider the **Id** and **All** constraints based on the **Id** and **All** relations in [7, 8]: The constraint **Id**, whose behavior $\mathbf{Id}(Y_{\text{in}})$ is from Y_{in} to $\{\mathbf{Id}[1]\}$ with $(H, K) \in \mathbf{Id}(Y_{\text{in}})$ iff H over Y_{in} and

$$K = \{\mathbf{Id}@x(x) \mid x \in \text{netw}(H)\}.$$

Intuitively, **Id** makes every node aware of its identifier by requiring that every node’s local **Id** relation is populated with a unique value resembling the node’s own identifier.

Furthermore, **All** is a constraint defined as a behavior $\mathbf{All}(Y_{\text{in}})$

from Y_{in} to $\{\text{All}[1]\}$ with $(H, K) \in \text{All}(Y_{\text{in}})$ iff H over Y_{in} and $K = \{\text{All}@x(y) \mid x \in \text{netw}(H) \text{ and } y \in \text{netw}(H)\}$. The latter enforces that a node's local All relation is populated with precisely all identifiers of all nodes in the network.

We recall that behaviors are (left-)total, hence for every transducer schema Y , every compatible constraint C , and every distributed instance H over Y_{in} , we require that there always exists at least one distributed instance K over Y_{sys} with $(H, K) \in C(Y_{\text{in}})$. Intuitively, a configuration constraint thus never restricts the input instances that a transducer network can accept.

The careful reader may have observed that the above two examples allow for a simpler definition of constraints:

1. The output schemas for **Id** and **All** are fixed and thus independent of Y_{in} ; and
2. **Id** and **All** are functions mapping distributed instances H over Y_{in} onto precisely one distributed instance for Y_{sys} .

In general, however, we want to allow constraints whose output schema depends on the input schema. Furthermore, we want to allow for multiple possible output distributions.

As another example of a constraint, we want to call attention to the $L(\sigma)$ -behaviors defined in Example 11. From now on, we refer to $L(\sigma)$ as the **L-constraint** or **leader constraint**. We also emphasize that a constraint is not computed, but serves as a representation of the knowledge that a system of interest has (in some way) encoded in its system relations.

Going forward, we will define multiple other constraints and also regularly combine them. Formally, we call two constraints C and C' *disjoint* if for every schema σ , $C(\sigma)$ and $C'(\sigma)$ are behaviors to disjoint schemas. For disjoint configuration constraints, we will write $C+C'$ to denote the constraint with $(C+C')(\sigma)(H) := C(\sigma)(H) \cup C'(\sigma)(H)$.

Configurations, Transitions, and Runs

The semantics of generalized relational transducers is defined in terms of runs over distributed instances. At a high level, a run is an infinite sequence of network states (which we call *configurations*) in

which every transition represents a local transition (and its side-effects such as sent messages and generated output tuples) for precisely one node.

Formally, given a C-transducer Π over some transducer schema Y , a *configuration* $\rho = (S, M, \Omega)$ of Π is a pair of distributed instances S, M , and a total function Ω defined as:

- The *state instance* S over $Y_{\text{out}} \cup Y_{\text{mem}}$, containing the output and memory facts at each node;
- the *message instance* M over Y_{msg} , specifying all messages currently in transit towards a node; and
- the *multiplicity function* Ω mapping each $x \in \text{netw}(M)$ to a function from $M(x)$ to $\mathbb{N}$, for how many times a message was delivered.

A *transition* of Π on a distributed instance H over Y_{in} is a quintuple $\tau = (\rho_i, x, R, W, \rho_j)$, where $x \in \text{netw}(H)$ is a node, $\rho_i = (S_i, M_i, \Omega_i)$ is a configuration of Π referred to as the *source configuration* of τ , R is a subset of $M_i(x)$ denoting messages read by x (i.e., received by x in τ), and W is an instance over Y_{msg} denoting messages written to other nodes by x (i.e., broadcast by x in τ). We let $\rho_j = (S_j, M_j, \Omega_j)$ be the *target configuration* of the transition. First, let D be a distributed instance defined by

$$D := H(x) \cup U(x) \cup S_i(x) \cup R, \text{ with } (H, U) \in \mathbf{C}(Y_{\text{in}}),$$

then we define the target's state instance S_j as

$$\begin{aligned} S_j(x)|_{Y_{\text{out}}} &= S_i(x)|_{Y_{\text{out}}} \cup O, \text{ with } (D, O) \in M_{\text{out}} \\ S_j(x)|_{Y_{\text{mem}}} &= [S_i(x)|_{Y_{\text{mem}}} \cup (I \setminus J)] \setminus (J \setminus I), \text{ with} \\ &\quad (D, I) \in M_{\text{in}} \text{ and } (D, J) \in M_{\text{del}} \\ S_j(y) &= S_i(y) \text{ for all } y \in \text{netw}(H) \setminus \{x\}, \end{aligned}$$

with $(D, W) \in M_{\text{snd}}$. The message instance M_j is defined by

$$\begin{aligned} M_j(x) &= M_i(x) \setminus R \\ M_j(y) &= M_i(y) \cup W \text{ for all } y \in \text{netw}(H) \setminus \{x\}. \end{aligned}$$

And the multiplicity function Ω_j is defined as

$$\begin{aligned}\Omega_j(x)(\mathbf{f}) &= \Omega_i(x)(\mathbf{f}) - k, \text{ for } \mathbf{f} \in M_j(x) \text{ and an arbitrary} \\ &\quad \text{number of messages read } k \leq \Omega_i(x)(\mathbf{f}) \\ \Omega_j(y)(\mathbf{f}) &= \Omega_i(y)(\mathbf{f}), \text{ for } \mathbf{f} \in M_j(y) \setminus W(y) \\ &\quad \text{with } y \in \text{netw}(H) \setminus \{x\} \\ \Omega_j(y)(\mathbf{f}) &= \Omega_i(y)(\mathbf{f}) + 1, \text{ for } \mathbf{f} \in W.\end{aligned}$$

A transition where $R = \emptyset$ is a *heartbeat* transition.

We define a *run* $\mathcal{R}$ of Π on a distributed instance H over Y_{in} as an infinite sequence of transitions of Π on H

$$(\rho_1, x_1, R_1, W_1, \rho_2), (\rho_2, x_2, R_2, W_2, \rho_3), \dots$$

where $\rho_1 = (S_1, M_1, \Omega_1)$ such that $S_1 = \emptyset$, $M_1 = \emptyset$, and with Ω_1 the empty function, as the *start configuration* of $\mathcal{R}$. Note that each subsequent transition has the target configuration of the previous transition as its source configuration. We always consider *fair* runs, which means that:

1. each node is the active node in an infinite number of transitions, and
2. facts that occur infinitely often in the message buffer of a node are infinitely often delivered to that node.

In other words, each node participates in the run, and we have at-least-once delivery guarantees for messages. Note that this fairness constraint is necessary since runs are not deterministic, each transition specifies some node that becomes active and which messages are delivered. We require fairness to guarantee liveness.

Distributed Output

We now define the *distributed output instance* at some configuration $\rho = (S, M, \Omega)$, denoted $\text{do}(\rho)$, as

$$\{R @ x(\bar{a}) \mid x \in \text{netw}(H), R(\bar{a}) \in S(x)|_{Y_{\text{out}}}\},$$

and define the *distributed quiescence configuration* of $\mathcal{R}$ as the configuration ρ_i at which each subsequent configuration ρ_j in the run

with $j > i$ satisfies $\text{do}(\rho_i) = \text{do}(\rho_j)$. The *distributed output instance* of a run $\mathcal{R}$ is defined as $\text{do}(\mathcal{R}) = \text{do}(\rho_i)$ where ρ_i is $\mathcal{R}$'s quiescence configuration.

We remark that even if a run leads to a quiescence configuration, the nodes in the network can not necessarily recognize that quiescence has been reached. We elaborate on this observation in Section 7.4 on Termination-Aware Transducers.

3 Summary

In this chapter, we have introduced the following core concepts.

- We introduced distributed behaviors in Section 6.1 as a way to model distributed computation semantically.
- Secondly, in Section 6.2 we defined the generalized transducer network model and defined its semantics in terms of fair runs.

Further Reading

The material in this section was first introduced in a PODS 2023 paper by Bas Ketsman and myself titled “*Distributed Consistency Beyond Queries*” [11]. This was later extended into a journal article titled “*A Generalized CALM Theorem for Non-Deterministic Computation in Asynchronous Distributed Systems*” [13].

Relational transducer networks were first used in [8] to formalize the CALM (*i.e.*, Consistency as Logical Monotonicity) Theorem, and later extended in [7] to include data partitioning strategies. We refer to Chapter 8 (specifically, Section 8.5 titled “Related Work”), for further discussion of formal models that are related to generalized transducer networks.

Expressivity of Transducer Networks

This chapter studies the computation of behaviors with the generalized transducer networks introduced in the previous chapter. We will first introduce the elementary concepts of computation and characterization of behaviors, and give general properties that relate the two concepts. Next, we establish that leader election is not always expressible by every transducer network, and we introduce the necessary configuration constraints that do allow us to express it. We then discuss the configuration constraints required for expressing all possible distributed behaviors. Lastly, we introduce termination-aware transducers, which are capable of deciding when they have reached a quiescence configuration.

1 Computing Behaviors with Transducers

We now study generalized transducer networks that compute behaviors. In other words, transducer networks with runs that produce a distributed quiescence configuration corresponding to a distributed output instance as specified by the behavior.

Definition 23. Let Π be a **C**-transducer over a transducer schema Y , and B a behavior from Y_{in} to Y_{out} . We say that

- Π *computes* B if, for each distributed instance H over Y_{in} , each fair run $\mathcal{R}$ of Π on H satisfies $(H, \text{do}(\mathcal{R})) \in B$; and
- B *characterizes* Π if Π computes B and for each $(H, K) \in B$ there is a fair run $\mathcal{R}$ of Π on H with $\text{do}(\mathcal{R}) = K$.

There is a subtle but important distinction between computation and characterization that only becomes apparent when faced with non-determinism. Under the definition of computation, if H is a distributed input instance and K is the output of a fair run of a transducer on H , then there must exist an admissible pair $(H, K) \in B$. Under the definition of characterization, however, for all admissible pairs $(H, K) \in B$ there must exist a fair run of the transducer on the distributed input H which yields K . We further illustrate this with the next example.

Example 13. Consider a transducer Π that takes as input any instance H for $L(\sigma)$, and as output always picks the highest node identifier as a leader. Similarly, we choose Π' to be a transducer which always picks the lowest node identifier as a leader. For this, we assume that there is a known order on domain elements captured by the system relations, we later formally model this with the **Ord**-constraint which is defined and discussed in Section 7.2.

It is apparent that both Π and Π' will always output something valid with respect to $L(\sigma)$, in the sense that for every input H there will exist some K for which $(H, K) \in L(\sigma)$. Indeed, according to the definition both Π and Π' compute the same behavior, in the sense that we can swap Π for Π' if our only concern is that all nodes agree on some leader.

However, Π and Π' are obviously not identical transducers. For the input $H = \{\text{Node@10}, \text{Node@20}\}$, we have that Π 's runs on H yield

$$K = \{\text{Leader@1(2)}, \text{Leader@2(2)}\}$$

while the runs of Π' on H always yield

$$K' = \{\text{Leader@1(1)}, \text{Leader@2(1)}\}.$$

To express this fact, we say that Π and Π' do not characterize $L(\sigma)$. A transducer that characterizes $L(\sigma)$ must allow for at least one run on H that yields K and at least one run on H that yields K' .

The class of behaviors that is equivalent to behaviors computable by a $\mathbf{C}$ -transducer is denoted by $\mathcal{F}(\mathbf{C})$. Similarly, the class of behaviors equivalent to the behaviors characterized by a $\mathbf{C}$ -transducer is denoted as $\mathcal{G}(\mathbf{C})$. The class of all computable behaviors is denoted as $\mathcal{B}$. A behavior is computable if there exists a Turing machine that on input H enumerates all K with $(H, K) \in B$. We remark that the use of equivalence in these definitions is only to prevent artificial results, like *e.g.*, $\text{Id}(\sigma) \notin \mathcal{G}(\text{Id})$, which would otherwise follow from the required disjointness of transducer schemas.

For two behaviors B and B' we say that $B \subseteq B'$ if they are over the same schema and for all distributed instances H and K we get that $(H, K) \in B$ implies $(H, K) \in B'$.

The next proposition summarizes basic properties of behaviors and their computability and characterizability.

Proposition 13. For every constraint $\mathbf{C}$:

1. every $\mathbf{C}$ -transducer is characterized by a behavior;
2. $\mathcal{G}(\mathbf{C}) \subseteq \mathcal{F}(\mathbf{C})$;
3. $B \in \mathcal{F}(\mathbf{C})$ implies $B' \in \mathcal{F}(\mathbf{C})$ if $B \subseteq B'$;
4. $B \in \mathcal{F}(\mathbf{C})$ if, and only if, there is a $B' \in \mathcal{G}(\mathbf{C})$ with $B' \subseteq B$;
5. $\mathcal{F}(\mathbf{C}) \subseteq \mathcal{F}(\mathbf{C} + \mathbf{C}')$ for constraints $\mathbf{C}'$ disjoint from $\mathbf{C}$; and
6. $\mathbf{C} \in \mathcal{F}(\mathbf{C} + \mathbf{C}')$ for constraints $\mathbf{C}'$ disjoint from $\mathbf{C}$.

Proposition 13 follows from the Lemmas 6-11 in the next subsection.

Proof of Proposition 13

We can consider the semantics of any $\mathbf{C}$ -transducer as a relation between all of its distributed input and output pairs. We prove that this relation is a behavior in the next lemma.

Lemma 6. For all constraints $\mathbf{C}$, and for all $\mathbf{C}$ -transducers Π , there exists a behavior B which characterizes Π .

Proof. We construct a relation B based on the input and output pairs of a transducer, and subsequently show that this relation is a behavior from the transducer's Y_{in} to its Y_{out} . Let $\mathbf{C}$ be a constraint, let Y be a transducer schema, and let Π be a $\mathbf{C}$ -transducer over Y .

Then, for all distributed instances H over Y_{in} , we define a relation B such that $(H, \text{do}(\mathcal{R})) \in B$ for all fair runs $\mathcal{R}$ of Π on H .

To prove that B is a behavior, we must prove that it is a mapping (*i.e.*, left-total, domain-preserving, and generic) and that it preserves node identifiers:

- a) B is *left-total*: B is defined for all possible distributed instances H over Y_{in} ;

Given that B is defined for every distributed instance H over Y_{in} , this property follows directly from its definition.

- b) B is *domain-preserving*: for all $(H, K) \in B$, $\text{Adom}(K) \subseteq \text{Adom}(H)$;

Let H be a distributed instance over Y_{in} , and let $\mathcal{R}$ be a fair run of Π on H . We prove that $\text{Adom}(\text{do}(\mathcal{R})) \subseteq \text{Adom}(H)$ by induction on the sequence of transitions in $\mathcal{R}$.

For the base case, let $\tau_1 = (\rho_1, x_1, R_1, W_1, \rho_2)$ be the first transition in $\mathcal{R}$ with source configuration $\rho_1 = (S_1, M_1, \Omega_1)$. We show $\text{Adom}(\text{do}(\rho_1)) \subseteq \text{Adom}(H)$:

$$\begin{aligned} & \text{Adom}(\text{do}(\rho_1)) \\ &= \text{Adom}(\{R @ x(\bar{a}) \mid x \in \text{netw}(H), R(\bar{a}) \in S_1(x)|_{Y_{\text{out}}}\}) \quad (7.1) \\ &= \text{Adom}(\text{netw}(H) \cup S_1(x)|_{Y_{\text{out}}}) \quad (7.2) \\ &= \text{Adom}(\text{netw}(H) \cup \emptyset) \quad (7.3) \\ &\subseteq \text{Adom}(H), \quad (7.4) \end{aligned}$$

where (7.1) applies the definition of distributed output, (7.2) writes $\text{do}(\rho_1)$ in terms of the sets from which it was constructed, (7.3) is due to $S_1 = \emptyset$ since ρ_1 is the start configuration of $\mathcal{R}$. And finally, (7.4) follows from the fact that $\text{netw}(H)$ is a restriction of the set of values in H .

For the induction case, let $\tau_j = (\rho_i, x_i, R_i, W_i, \rho_j)$ be the j -th transition in $\mathcal{R}$. For the *induction hypothesis* (IH), assume that the inclusion $\text{Adom}(\text{do}(\rho_i)) \subseteq \text{Adom}(H)$ holds. We show this implies the

inclusion $\text{Adom}(\text{do}(\rho_j)) \subseteq \text{Adom}(H)$ holds:

$$\begin{aligned} & \text{Adom}(\text{do}(\rho_j)) \\ &= \text{Adom}(\{R @ x(\bar{a}) \mid x \in \text{netw}(H), R(\bar{a}) \in S_j(x)|_{Y_{\text{out}}}\}) \end{aligned} \quad (7.5)$$

$$= \text{Adom}(\text{netw}(H) \cup S_j|_{Y_{\text{out}}}) \quad (7.6)$$

$$= \text{Adom}(\text{netw}(H) \cup S_i|_{Y_{\text{out}}} \cup O) \quad (7.7)$$

where $(H_i(x_i) \cup U_i(x_i) \cup S_i(x_i) \cup R_i, O) \in M_{\text{out}}$

and $(H_i, U_i) \in \mathbf{C}(Y_{\text{in}})$

$$\subseteq \text{Adom}(H), \quad (7.8)$$

where (7.5) applies the definition of distributed output, where (7.6) rewrites $\text{do}(\rho_j)$ in terms of the sets from which it was constructed, and where (7.7) applies the definition of S_j . Lastly, (7.8) follows from the following inclusions:

- $\text{Adom}(\text{netw}(H)) \subseteq \text{Adom}(H)$, as $\text{netw}(H)$ is a restriction of H ;
- $\text{Adom}(S_i|_{Y_{\text{out}}}) \subseteq \text{Adom}(\text{do}(\rho_i)) \subseteq \text{Adom}(H)$ by the IH; and
- $\text{Adom}(O) \subseteq \text{Adom}(H)$ by M_{out} being domain-preserving.

This shows $\text{Adom}(\text{do}(\mathcal{R})) \subseteq \text{Adom}(H)$, thus B is domain-preserving.

c) B is *generic*: for all bijective renamings $\beta : \mathbf{dom} \rightarrow \mathbf{dom}$, we have $(H, K) \in B$ if, and only if, $(\beta(H), \beta(K)) \in B$; and

($\Rightarrow$) Let $\beta : \mathbf{dom} \rightarrow \mathbf{dom}$ be a bijective renaming, let H be a distributed instance over Y_{in} , and let $\mathcal{R}$ be a fair run of Π on H with distributed output $\text{do}(\mathcal{R})$, such that $(H, \text{do}(\mathcal{R})) \in B$. We construct a fair run $\mathcal{R}^\beta$ of Π on $\beta(H)$ and show that $(\beta(H), \text{do}(\mathcal{R}^\beta)) \in B$ and $\text{do}(\mathcal{R}^\beta) = \beta(\text{do}(\mathcal{R}))$.

Let $\tau_i = (\rho_i, x_i, R_i, W_i, \rho_j)$ be the i -th transition in $\mathcal{R}$ with the source $\rho_i = (S_i, M_i, \Omega_i)$ and target $\rho_j = (S_j, M_j, \Omega_j)$. Then, we define the i -th transition of $\mathcal{R}^\beta$ as $\tau_i^\beta = (\rho_i^\beta, x_i^\beta, R_i^\beta, W_i^\beta, \rho_j^\beta)$, where $x_i^\beta := \beta(x_i)$, $R_i^\beta := \beta(R_i)$, $W_i^\beta := \beta(W_i)$, and with:

$$\rho_i^\beta := (S_i^\beta, M_i^\beta, \Omega_i^\beta) \text{ w. } S_i^\beta := \beta(S_i), M_i^\beta := \beta(M_i) \text{ and } \Omega_i^\beta := \beta \circ \Omega_i, \quad (7.9)$$

$$\rho_j^\beta := (S_j^\beta, M_j^\beta, \Omega_j^\beta) \text{ w. } S_j^\beta := \beta(S_j), M_j^\beta := \beta(M_j) \text{ and } \Omega_j^\beta := \beta \circ \Omega_j. \quad (7.10)$$

This construction yields a run $\mathcal{R}_\beta$ of Π on $\beta(H)$, which is fair because $\mathcal{R}$ is fair. And, since B contains all pairs of fair runs of Π for any distributed instance H over Y_{in} , hence $(\beta(H), \text{do}(\mathcal{R}_\beta)) \in B$.

Furthermore, with $\rho_q := (S_q^\beta, M_q^\beta, \Omega_q^\beta)$ the quiescence configuration of $\mathcal{R}^\beta$, we can rewrite $\text{do}(\mathcal{R}^\beta)$ as:

$$\text{do}(\mathcal{R}^\beta) = \{R @ x(\bar{a}) \mid x \in \text{netw}(\beta(H)), R(\bar{a}) \in S_q^\beta(x)|_{Y_{\text{out}}}\} \quad (7.11)$$

$$= \beta(\{R @ x(\bar{a}) \mid x \in \text{netw}(H), R(\bar{a}) \in S_q(x)|_{Y_{\text{out}}}\}), \quad (7.12)$$

$$= \beta(\text{do}(\mathcal{R})), \quad (7.13)$$

where (7.11) and (7.13) apply the definition of distributed output; and (7.12) distributes out β , since $\beta(\text{netw}(H)) = \text{netw}(\beta(H))$, and since $\beta(S_q)$ is defined as S_q^β by (7.9) and (7.10). This shows that $\text{do}(\mathcal{R}^\beta) = \beta(\text{do}(\mathcal{R}))$.

We have shown that $(\beta(H), \text{do}(\mathcal{R}_\beta)) \in B$ and $\text{do}(\mathcal{R}^\beta) = \beta(\text{do}(\mathcal{R}))$, hence this proves that $(\beta(H), \beta(\text{do}(\mathcal{R}))) \in B$.

($\Leftarrow$) This argument is analog to the ($\Rightarrow$)-direction if we replace “ H ” by $\beta(H)$ and “ $\beta(H)$ ” by $\beta^{-1}(H)$. The inverse of β exists since it is a bijection.

d) *B preserves node-identifiers*: for all pairs $(H, K) \in B$, we have that $\text{netw}(K) \subseteq \text{netw}(H)$.

Let H be a distributed instance over Y_{in} , and let $\mathcal{R}$ be a fair run of Π on H , hence $(H, \text{do}(\mathcal{R})) \in B$. Additionally, let $\rho_q = (S_q, M_q, \Omega_q)$ be the quiescence configuration of $\mathcal{R}$. We can rewrite $\text{netw}(\text{do}(\mathcal{R}))$ as:

$$\text{netw}(\text{do}(\mathcal{R})) = \text{netw}(\{R @ x(\bar{a}) \mid x \in \text{netw}(H), R(\bar{a}) \in S_q(x)|_{Y_{\text{out}}}\}) \quad (7.14)$$

$$= \text{netw}(H) \cup \text{netw}(S_q|_{Y_{\text{out}}}) = \text{netw}(H) \quad (7.15)$$

where (7.14) applies the definition of distributed output, and where (7.15) rewrites the distributed output in terms of the sets from which it was constructed. The second equality follows from $\text{netw}(S_q|_{Y_{\text{out}}}) = \emptyset$, since M_{out} is a mapping from Y_{all} to Y_{out} , and Y_{out} does not have node identifier positions for tuples specified by its schema (*i.e.*, to add node identifiers to facts we need to define an augmented version of the schema through the $\text{dist}(Y_{\text{out}})$ construct). This proves that $\text{netw}(\text{do}(\mathcal{R})) \subseteq \text{netw}(H)$.

By claims (a), (b), and (c) we have shown that B is a mapping, and by claim (d) we have shown that B is a behavior. $\square$

Lemma 7. For each constraint $\mathbf{C}$, we have that $\mathcal{G}(\mathbf{C}) \subseteq \mathcal{F}(\mathbf{C})$.

Proof. This result follows directly from Definition 23. $\square$

If a transducer computes a behavior B , then any behavior B' with $B \subseteq B'$ must also be computable. Indeed, we can simply use an answer for B as an answer for B' . Notice that this is not the case for characterization.

Lemma 8. For each constraint $\mathbf{C}$, $B \in \mathcal{F}(\mathbf{C})$ implies $B' \in \mathcal{F}(\mathbf{C})$ if $B \subseteq B'$.

Proof. Let $\mathbf{C}$ be a constraint and let $B \in \mathcal{F}(\mathbf{C})$ be a behavior from σ_1 to σ_2 . Then, let Π be a $\mathbf{C}$ -transducer over transducer schema $\mathbf{Y}$ with $Y_{\text{in}} := \sigma_1$ and $Y_{\text{out}} := \sigma_2$, such that Π computes B . Next, let B' be a behavior from σ_1 to σ_2 such that $B \subseteq B'$. We prove that $B' \in \mathcal{F}(\mathbf{C})$.

Let H be a distributed instance H over Y_{in} . Since Π computes B , there must be a fair run $\mathcal{R}$ of Π on H with $(H, \text{do}(\mathcal{R})) \in B$. Since $B \subseteq B'$, we have $(H, \text{do}(\mathcal{R})) \in B'$. Hence, Π also computes B' , and therefore $B' \in \mathcal{F}(\mathbf{C})$. $\square$

Every computable behavior has some subset that can be characterized by a transducer. And, conversely, for every behavior that is characterized by a transducer there exists an extension that can be computed by a transducer.

Lemma 9. For each constraint $\mathbf{C}$, we have $B \in \mathcal{F}(\mathbf{C})$ if, and only if, there exists a $B' \in \mathcal{G}(\mathbf{C})$ with $B' \subseteq B$.

Proof. Let $B' \in \mathcal{G}(\mathbf{C})$ be a behavior. By Lemma 7, we get $B' \in \mathcal{F}(\mathbf{C})$. Let B be a behavior with $B' \subseteq B$. By Lemma 8 we get $B \in \mathcal{F}(\mathbf{C})$.

Conversely, we let $B \in \mathcal{F}(\mathbf{C})$ with Π a $\mathbf{C}$ -transducer computing B . By Lemma 6, we know that Π is characterized by a behavior B' (that is, $B' \in \mathcal{G}(\mathbf{C})$). Since Π computes B , we have that $B' \subseteq B$. $\square$

Lemma 10. For all constraints $\mathbf{C}$ and $\mathbf{C}'$, such that $\mathbf{C}'$ is disjoint from $\mathbf{C}$, we have that $\mathcal{F}(\mathbf{C}) \subseteq \mathcal{F}(\mathbf{C} + \mathbf{C}')$.

Proof. Let C' and C be two disjoint constraints. Let B be a behavior from σ_1 to σ_2 computed by a C -transducer Π over Y , such that $Y_{\text{in}} := \sigma_1$, and $Y_{\text{out}} := \sigma_2$. We show there exists a $C+C'$ -transducer which also computes B .

Let Π' be a $C+C'$ -transducer over Y' where the mappings of Π' are defined to be identical to the ones of Π , and where Y' is identical to Y except for the system schema Y'_{sys} which is compatible with $C+C'$ and satisfies $Y_{\text{sys}} \subseteq Y'_{\text{sys}}$.

Let H be a distributed instance over Y_{in} and let $\mathcal{R}$ be a fair run of Π on H . Since Π computes B we have that $(H, \text{do}(\mathcal{R})) \in B$. However, since the mappings of Π' are identical to the mappings of Π , $\mathcal{R}$ is also a valid fair run of Π' on H . Therefore, Π' also computes B . This shows the existence of a $C+C'$ -transducer computing B . It follows that $B \in \mathcal{F}(C+C')$. $\square$

Lemma 11. For all constraints C and C' , such that C' is disjoint from C , we have that $C \in \mathcal{F}(C+C')$.

Proof. Let σ be an arbitrary schema, let $C(\sigma)$ and $C'(\sigma)$ be two constraints that are disjoint, and let Y be a transducer schema such that $Y_{\text{in}} := \sigma$. We define $\Pi = (C+C', M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$ where M_{out} is defined as $C(Y_{\text{in}})$, and where M_{in} , M_{del} and M_{snd} are defined as the empty mappings.

Let H be a distributed instance over Y_{in} . Now, we construct a run $\mathcal{R}$ of Π on H such that $(H, \text{do}(\mathcal{R})) \in C(Y_{\text{in}})$. Let $(\leq, \text{netw}(H))$ be a total order on all nodes in H . Then, let $\mathcal{R}$ be the run of Π on H where in every transition a node x from $\text{netw}(H)$ is chosen from low to high according to $\leq$. In each transition, this node x becomes active and computes M_{out} over $H(x)$. After precisely $|\text{netw}(H)|$ transitions, every node has been active once. Beyond this point, the run reaches quiescence since M_{in} , M_{del} , and M_{snd} are empty mappings which do not produce new input for M_{out} .

This sequence of $|\text{netw}(H)|$ transitions repeats infinitely often in $\mathcal{R}$, and since each node is chosen once according to $\leq$, we know that $\mathcal{R}$ is a fair run. Furthermore, we know that $(H, \text{do}(\mathcal{R})) \in M_{\text{out}}$ iff $(H, \text{do}(\mathcal{R})) \in C(Y_{\text{in}})$ since $M_{\text{out}} = C(Y_{\text{in}})$ by definition. This shows that Π computes $C(Y_{\text{in}})$ and hence $C \in \mathcal{F}(C+C')$. $\square$

2 Expressing Leader Election

The **Id+All**-transducers are the most expressive variant considered in the literature, and they can express every computable query [8] (of course, given that the language of transducer mappings is expressive enough). Based on these results, the reader may be inclined to believe that all computable behaviors (*i.e.*, those in $\mathcal{B}$) are computable with an **Id+All**-transducer, or $\mathcal{F}(\mathbf{Id+All}) = \mathcal{B}$, but this is not true. In particular, the next theorem shows that $\mathbf{L}(\sigma)$ (*c.f.*, Example 11) is not computable with an **Id+All**-transducer.

Theorem 11. $\mathbf{L}(\sigma) \notin \mathcal{F}(\mathbf{Id+All})$.

Theorem 11 is a consequence of the fact that transducers are based on generic queries and therefore cannot break the symmetry in input instances. This argument closely resembles earlier impossibility results [10, 62] for synchronous distributed system models, showing that symmetry has to be broken somehow in order to solve the leader-election problem.

We first define terminology. An *automorphism* for a distributed instance H over σ is a bijective mapping $\beta : \text{Adom}(H) \rightarrow \text{Adom}(H)$ with $\beta(H) = H$. We call an automorphism a *k-cycle*, if for every element $n \in \text{Adom}(H)$ we have that $\beta^k(n) = n$, where β^k is notation for k function compositions of β . Theorem 11 is a corollary of the following result. Intuitively, we require β to be a $|\text{netw}(H)|$ -cycle to prevent distributed input instances that have asymmetry in the input, that is, one node that has facts which the other nodes do not have. This asymmetry can be exploited to create an **Id+All**-transducer that computes leader election for that specific asymmetric instance. We will elaborate on this further with an example after the proof has been presented.

Proposition 14. Let B be a behavior computable by a **C**-transducer over a transducer schema $\mathbf{Y}$. If $(H, U) \in \mathbf{C}(\mathbf{Y}_{\text{in}})$, and β is a $|\text{netw}(H)|$ -cycle for $H \cup U$, then β is an automorphism for at least one distributed instance K with $(H, K) \in B$.

Proof. Let Π be the **C**-transducer implied by the proposition. For

notational convenience we will assume, without loss of generality, that $\text{netw}(H) = [1, k]$, for some positive integer k denoting the size of the network. The proof is by construction of a fair run $\mathcal{R} = \rho_0, \rho_1, \dots$ for transducer Π on distributed instance H that produces the output described by the proposition.

Let β be a k -cycle of $H \cup U$, for $(H, U) \in \mathbf{C}(Y_{\text{in}})$. We call a set $S \subseteq \text{Adom}(H \cup U)$ an *orbit* of n in β if $S = \{\beta^i(n) \mid i = 1, 2, \dots, k\}$ for some $n \in \text{Adom}(H \cup U)$. We will assume that $[1, k]$ represents the nodes in cycle notation for β . This particularly means that for every i either $\beta(i) = i + 1$ or $\beta(i) = i - k$ (due to our assumption that β is a k -cycle).

Intuitively, we construct $\mathcal{R}$ as an MPC computation [17], which is organized as a series of synchronized rounds in which nodes perform a computation in parallel and wait for all messages to arrive before continuing with the next round. Formally, a round is a contiguous sequence of k transitions, with one transition per node in $[1, k]$. An arbitrary round $i \geq 1$ thus starts at configuration $\rho_{(i-1)k}$ and ends at configuration $\rho_{i.k}$. When a node x becomes active in round $i \geq 1$ it will always consume precisely the fragment of the message instance that was present at the start of the round, that is, $M_{(i-1)k}(x)$. An arbitrary transition in round $i \geq 1$ thus takes the following form

$$(\rho_w, x, M_{(i-1)k}(x), O_{\text{snd}}, \rho_{w+1}) \text{ where } \rho_w := (S_w, M_w, \Omega_w), \text{ and } \quad (7.16) \\ \rho_{w+1} := (S_{w+1}, M_{w+1}, \Omega_{w+1})$$

with $S_{w+1}(y)|_{Y_{\text{out}}} := S_w(y)|_{Y_{\text{out}}} \cup \beta(O_{\text{out}})$, and

$$S_{w+1}(y)|_{Y_{\text{mem}}} := [S_w(y)|_{Y_{\text{mem}}} \cup (\beta(O_{\text{in}}) \setminus \beta(O_{\text{del}}))] \setminus (\beta(O_{\text{del}}) \setminus \beta(O_{\text{in}})),$$

such that $(D_x, O_{\text{out}}) \in M_{\text{out}}$, $(D_x, O_{\text{in}}) \in M_{\text{in}}$, $(D_x, O_{\text{snd}}) \in M_{\text{snd}}$, and $(D_x, O_{\text{del}}) \in M_{\text{del}}$ for $D_x := H(x) \cup S_w(x) \cup M_{(i-1)k}(x) \cup U(x)$ where $(H, U) \in \mathbf{C}(Y_{\text{in}})$.

We let ρ_0 be the start configuration of Π and proceed with an inductive construction of $\mathcal{R}$ in which we will specify more precisely the choices of transitions that are still left open. More precisely, we will describe the construction of round $i \geq 1$, assuming the following condition:

$$(\dagger) \quad \beta(S_{(i-1)k}(x)) = S_{(i-1)k}(\beta(x)) \text{ and } \beta(M_{(i-1)k}(x)) = M_{(i-1)k}(\beta(x)), \text{ for every } x \in [1, k].$$

Notice that $(\dagger)$ is true for round $i = 1$, by choice of configuration ρ_0 . Further on in the proof, we will show that $(\dagger)$ is also true for round $i + 1$, implying that our construction is complete and $(\dagger)$ holds for all configurations that start (or end) a round.

A round i , for which we assume $(\dagger)$ to hold, consists of k transitions for nodes $[1, k]$, made active in natural order. As a transition, we either take an arbitrary transition (*c.f.*, Eq. 7.16) or a transition that imitates one of the earlier transitions of the round. Specifically, node $y \in [1, k]$ has an arbitrary transition if $\beta^{-1}(y) \neq y - 1$ (*i.e.*, y is the first node of its orbit in β that becomes active) and an imitating transition if $\beta^{-1}(y) = y - 1$ (*i.e.*, $y - 1$ is a node of the same orbit and has already been active). For the case $\beta^{-1}(y) = y - 1$, taking the transition in Eq. 7.16 as transition of $x = \beta^{-1}(y)$, the imitating transition for y goes as follows:

$$(\rho_u, y, M_{(i-1)k}(y), O_{\text{snd}}, \rho_{u+1}) \text{ where } \rho_u := (S_u, M_u, \Omega_u), \text{ and } \quad (7.17)$$

$$\rho_{u+1} := (S_{u+1}, M_{u+1}, \Omega_{u+1}),$$

with $S_{u+1}(y)|_{Y_{\text{out}}} := S_u(y)|_{Y_{\text{out}}} \cup \beta(O_{\text{out}})$, and

$$S_{u+1}(y)|_{Y_{\text{mem}}} := [S_u(y)|_{Y_{\text{mem}}} \cup (\beta(O_{\text{in}}) \setminus \beta(O_{\text{del}}))] \setminus (\beta(O_{\text{del}}) \setminus \beta(O_{\text{in}})).$$

To see that the construction of round i is valid, we need to convince ourselves that $(D_y, \beta(O_{\text{out}})) \in M_{\text{out}}$, $(D_y, \beta(O_{\text{in}})) \in M_{\text{in}}$ and $(D_y, \beta(O_{\text{del}})) \in M_{\text{del}}$, for $D_y := H(y) \cup S_u(y) \cup M_{(i-1)k}(y) \cup U(y)$ where $(H, U) \in \mathbf{C}(Y_{\text{in}})$.

The inclusions are clearly true if $\beta(D_x) = D_y$, as then $(D_x, O_{\text{out}}) \in M_{\text{out}}$ implies $(D_y, \beta(O_{\text{out}})) \in M_{\text{out}}$, $(D_x, O_{\text{in}}) \in M_{\text{in}}$ implies $(D_y, \beta(O_{\text{in}})) \in M_{\text{in}}$, and $(D_x, O_{\text{del}}) \in M_{\text{del}}$ implies $(D_y, \beta(O_{\text{del}})) \in M_{\text{del}}$, following directly from the genericity of mappings. That $\beta(D_x) = D_y$ is indeed true is a consequence of $(\dagger)$ for round i and the fact that nodes x and y become active at most once per round, thus with $S_w(x) = S_{(i-1)k}(x)$ and $S_u(y) = S_{(i-1)k}(y)$.

To see that $(\dagger)$ is true for round $i + 1$, we take an arbitrary node $x \in [1, k]$, with $y = \beta(x)$. We first show $\beta(S_{i,k}(x)) = S_{i,k}(y)$. The latter follows directly from the constructions if $y = x + 1$, because (taking Eq. 7.16 and Eq 7.17 as the respective transitions) $S_{(i-1)k}(x) = S_x(x)$, $S_{(i-1)k}(y) = S_y(y)$, $S_{i,k}(x) = S_{x+1}(x)$, $S_{i,k}(y) = S_{y+1}(y)$, and $(\dagger)$ is true

for round i , thus we have

$$\begin{aligned}
S_{i.k}(y)|_{Y_{\text{out}}} &= S_{(i-1).k}(y)|_{Y_{\text{out}}} \cup \beta(O_{\text{out}}) \\
&= \beta(S_{(i-1).k}(x)|_{Y_{\text{out}}}) \cup O_{\text{out}} \\
&= \beta(S_{i.k}(x)|_{Y_{\text{out}}}), \text{ and} \\
S_{i.k}(y)|_{Y_{\text{mem}}} &= [S_{(i-1).k}(y)|_{Y_{\text{mem}}} \cup (\beta(O_{\text{in}}) \setminus \beta(O_{\text{del}}))] \setminus (\beta(O_{\text{del}}) \setminus \beta(O_{\text{in}})) \\
&= \beta([S_{(i-1).k}(x)|_{Y_{\text{mem}}} \cup (O_{\text{in}} \setminus O_{\text{del}})] \setminus (O_{\text{del}} \setminus O_{\text{in}})) \\
&= \beta(S_{i.k}(x)|_{Y_{\text{mem}}}),
\end{aligned}$$

If $y \neq x + 1$ it must be that $y = x - k + 1$ (due to assumptions on β , particularly that all orbits over network nodes agree with the order k). In other words, $\beta^{k-1}(y) = x$. More precisely, by assumptions on β , $y = \beta^{k-1}(x)$. We can now apply a similar analysis as for the previous case

$$\begin{aligned}
S_{i.k}(y)|_{Y_{\text{out}}} &= S_{(i-1).k}(y)|_{Y_{\text{out}}} \cup O_{\text{out}} \\
&= \beta(S_{(i-1).k}(x)|_{Y_{\text{out}}}) \cup \beta^k(O_{\text{out}}) \\
&= \beta(S_{(i-1).k}(x)|_{Y_{\text{out}}}) \cup \beta^{k-1}(O_{\text{out}}) \\
&= \beta(S_{i.k}(x)|_{Y_{\text{out}}}), \text{ and} \\
S_{i.k}(y)|_{Y_{\text{mem}}} &= [S_{(i-1).k}(y)|_{Y_{\text{mem}}} \cup (O_{\text{in}} \setminus O_{\text{del}})] \setminus (O_{\text{del}} \setminus O_{\text{in}}) \\
&= \beta([S_{(i-1).k}(x)|_{Y_{\text{mem}}} \\
&\quad \cup (\beta^{k-1}(O_{\text{in}}) \setminus \beta^{k-1}(O_{\text{del}}))] \setminus (\beta^{k-1}(O_{\text{del}}) \setminus \beta^{k-1}(O_{\text{in}}))) \\
&= \beta(S_{i.k}(x)|_{Y_{\text{mem}}}).
\end{aligned}$$

It remains to show $\beta(M_{i.k}(x)) = M_{i.k}(y)$. For this, notice that for an arbitrary node z :

$$\begin{aligned}
M_{i.k}(z) &= ([M_{(i-1).k}(z) \cup \{\mathbf{f} \in O_{\text{snd}, v} \mid v \in [1, z-1]\}] \setminus M_{(i-1).k}(z)) \\
&\quad \cup \{\mathbf{f} \in O_{\text{snd}, v} \mid v \in [z+1, k]\},
\end{aligned}$$

with the multiplicity function $\Omega_{i.k}(z)$ as

$$\begin{aligned}
\Omega_{i.k}(z)(\mathbf{f}) &= (\Omega_{(i-1).k}(z)(\mathbf{f}) + |\{\mathbf{f} \in O_{\text{snd}, v} \mid v \in [1, z-1]\}|) \\
&\quad + |\{\mathbf{f} \in O_{\text{snd}, v} \mid v \in [z+1, k]\}|
\end{aligned}$$

and with $(D_v, O_{\text{snd}, v}) \in M_{\text{snd}}$ denoting the facts sent by node v during round i . Building further on the above equation (again, updating

the multiplicity function $\Omega_{i,k}$ accordingly) we obtain

$$\begin{aligned}
M_{i,k}(y) &= \{\mathbf{f} \in O_{\text{snd},v} \mid v \in [1, y-1]\} \cup \{\mathbf{f} \in O_{\text{snd},v} \mid v \in [y+1, k]\} \\
&= \{\mathbf{f} \in O_{\text{snd},v} \mid v \in [1, k]\} \setminus O_{\text{snd},y} \\
&= \beta(\{\mathbf{f} \in O_{\text{snd},v} \mid v \in [1, k]\} \setminus O_{\text{snd},x}) \\
&= \beta(\{\mathbf{f} \in O_{\text{snd},v} \mid v \in [1, x-1]\} \cup \{\mathbf{f} \in O_{\text{snd},v} \mid v \in [x+1, k]\}) \\
&= \beta(M_{i,k}(x)).
\end{aligned}$$

for every $x \in [1, k]$ and $y = \beta(x)$.

To conclude the proof, we note that $\mathcal{R}$ is a fair run, since all rounds are finite and messages are read after every round. Since Π computes behavior B , run $\mathcal{R}$ must thus have a quiescence configuration representing a distributed output K such that $(H, K) \in B$. Finally, the claim of the proposition now follows immediately from $(\dagger)$. $\square$

We now demonstrate why this condition of β being a $|\text{netw}(H)|$ -cycle is important in the above proposition. Intuitively, it prevents the distributed instance H from being asymmetric, as leader election is often computable with **Id+All**-transducers in this case. For instance, consider the following example.

Example 14. Consider a behavior B which picks one node to be the leader if it has precisely one fact $T()$ present in its part of the input, otherwise it picks an arbitrary leader. Concretely, given the distributed instance

$$H = \{N@1(), N@2(), N@3(), T@1()\}$$

where $N[0]$ is a relation that denotes that this node is a node, and where $T[0]$ is a nullary relation representing the presence of a token. We have that the only valid output for B is (H, K) with $K = \{\text{Leader}@1(1), \text{Leader}@2(1), \text{Leader}@3(1)\}$ since there is precisely one node, namely node 1, with $T()$ in its input.

Indeed, we can demonstrate that for this specific instance, the condition of Proposition 14 does not hold. H has precisely two possible automorphisms, namely:

$$\beta_1 = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3\} \text{ and } \beta_2 = \{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 2\}.$$

Here, β_1 is a 1-cycle and β_2 is a 2-cycle, hence neither are $|\text{netw}(H)|$ -cycles (i.e., 3-cycles).

We can easily build an **Id+All**-transducer that computes B specifically for asymmetric instances such as H (but not in general). Indeed, all nodes simply broadcast their input, and each node outputs $\text{Leader}(x)$ for the node x that has the fact $T()$.

In order to have transducers that can deal with symmetric instances, a trivial solution is to bake asymmetry into a constraint. For instance, we can add the presence of a leader into the system by using the **L**-constraint.

Corollary 6. $L(\sigma) \in \mathcal{F}(\mathbf{Id+All+L})$.

Proof. This follows from Proposition 13, specifically Lemma 11. $\square$

Another approach to break symmetry is order. For this, we define a configuration constraint that makes a relation $\leq[2]$ available which specifies an order over the values of the active domain of the distributed input instance, thus reflecting the relevant part of an arbitrary order $\leq$ over the domain **dom**. Formally, we define configuration constraint **Ord** to map schemas σ to behaviors from σ to $\{\leq[2]\}$ where $\text{acc}(\leq) = \text{input}$, and with $(H, U) \in \mathbf{Ord}(\sigma)$ iff H over σ and $U = \{\leq @x(a, b) \mid a, b \in \text{Adom}(H), a \leq b\}$.

We remark that $\text{acc}(\leq) = \text{input}$ is important, as it allows nodes to compare data values that they already know (either because the value was in the local input or system instance, or because it arrived during the computation as part of a message) without leaking undesirable additional information. Indeed, with $\text{acc}(\leq) = \text{output}$ the nodes would, in addition to a relative order, also know, from the start, the complete active domain of the system.

Example 15. We construct a **Id+All+Ord**-transducer capable of computing $L(\sigma)$. The transducer picks the smallest node identifier in its **All**[1]-relation to be the leader.

Let Y be a transducer schema, where Y_{in} is defined as $\{\text{Node}[0]\}$, with Y_{out} defined as $\{\text{Leader}[1]\}$, and with $Y_{\text{mem}} := \emptyset$, $Y_{\text{msg}} := \emptyset$, and $Y_{\text{sys}} := \{\leq[2]\}$. Let H be a distributed instance over Y_{in} , and let $(H, U) \in \mathbf{Id+All+Ord}(Y_{\text{in}})$. Next, define

$$\Pi = (\mathbf{Id+All+Ord}, M_{\text{in}}, M_{\text{out}}, M_{\text{del}}, M_{\text{snd}})$$

to be a transducer over Y where given any $a, b, c \in \text{Adom}(H(x))$, M_{out} maps each $\text{All}(a)$, $\text{All}(b)$, $\text{All}(c) \in U(x)$ with $\leq(a, b) \in U(x)$ and $\leq(c, a) \notin U(x)$ to $\text{Leader}(a)$; and with M_{in} , M_{del} , and M_{snd} defined as the empty mappings.

Both **Id+All+Ord**-transducers and **Id+All+L**-transducers are capable of characterizing the same behaviors. Indeed, order can be used to grant the leader status to a specific node that all nodes of the network agree on, and a leader node can choose an arbitrary order over the active domain and broadcast its choice to the other nodes of the network. We summarize this insight as follows.

Theorem 12. $\mathcal{F}(\text{Id+All+Ord}) = \mathcal{F}(\text{Id+All+L})$.

Proof of Theorem 12

We show the two inclusions for Theorem 12 one by one in Proposition 15 and Proposition 16.

Proposition 15. $\mathcal{F}(\text{Id+All+Ord}) \subseteq \mathcal{F}(\text{Id+All+L})$

Proof. Let σ_1 and σ_2 be two schemas, and let B be a behavior from σ_1 to σ_2 that is computable by **Id+All+Ord**-transducers.

Transducer Construction Let $\Pi_{\leq}$ be a **Id+All+Ord**-transducer. It computes B and has the mappings $M_{\text{in}}^{\leq}$, $M_{\text{out}}^{\leq}$, $M_{\text{del}}^{\leq}$ and $M_{\text{snd}}^{\leq}$. We construct a **Id+All+L**-transducer Π over Y , where Y is a transducer schema with Y_{in} and Y_{out} defined as σ_1 and σ_2 respectively, and with Y_{sys} given by the output schema of **Id+All+L**(Y_{in}). The remaining Y_{mem} and Y_{msg} can be derived from Π 's mappings, which we define later. Π will operate in three phases:

- P1:* In the first phase, all transducers send their input relations to the leader;
- P2:* the next phase triggers once the leader has received all facts. This constitutes a reconstruction of H . We can then use this to obtain a distributed instance U where $(H, U) \in \text{Ord}(Y_{\text{in}})$. Then, we broadcast the relevant parts of U to all other transducers; and

P3: finally, the last phase triggers once all transducers have received the order facts. At this point, the transducer can mimic a **Id+All+Ord**-transducer such as $\Pi_{\leq}$.

P1 We now define M_{in} and M_{snd} in tandem; the mappings M_{out} and M_{del} are left empty until phase 3. If a transducer does not have both the $\text{Leader}(x)$ and $\text{Id}(x)$ facts present in its system memory, it is not a leader node, and then the M_{snd} mapping broadcasts all input facts $R(\bar{x})$ to every other transducer under a specially prefixed relation name $\text{SndR}(y, \bar{x})$, where y is the node identifier for that transducer accessible through Id .

In turn, the leader (*i.e.*, the transducer with $\text{Leader}(x)$ and $\text{Id}(x)$ locally present) uses the M_{in} mapping to store incoming $\text{SndR}(y, \bar{x})$ fact as both a $\text{NetR}(y, \bar{x})$ fact, and $R(\bar{x})$. Next, for each $\text{SndR}(\bar{x})$ fact received, the leader broadcasts an $\text{AckR}(\bar{x})$ fact through M_{snd} . Once each non-leader transducer observes a $\text{AckR}(\bar{x})$ fact for every $R(\bar{x})$ fact in its local input, it broadcasts an $\text{Ack}(x)$ fact via M_{snd} , with x the node identifier obtained from $\text{Id}(x)$.

Finally, once the leader node observes a $\text{Ack}(x)$ fact for every $\text{All}(x)$ fact such that x is not the leader's node identifier, it broadcasts $\text{P1Done}()$ via M_{snd} and stores it locally with M_{in} . This concludes the first phase.

P2 When the leader has the fact $\text{P1Done}()$ locally, Π enters phase two. Since the leader now has all facts from all transducers, it can construct H by restricting its local state to the NetR relation and the local $R(\bar{x})$ relation annotated with the leader's local identifier x . Notice that we have to strip away the Net prefix for this, we annotate all R relations with their respective node identifier. This gives us a distributed instance H stored on the leader. We can then take U where $(H, U) \in \mathbf{Ord}(Y_{\text{in}})$ and, for every $\leq @y(a, b) \in U$ broadcast $\text{Snd}\leq(y, a, b)$ facts via M_{snd} and store $\leq(a, b)$ locally via M_{in} if $y = x$ where $\text{Id}(x)$.

In response, the non-leader transducers store each $\text{Snd}\leq(y, a, b)$ fact locally as $\leq(a, b)$ via M_{in} if y matches its node identifier, and broadcast $\text{Ack}\leq(y, a, b)$ via M_{snd} with y the identifier of that transducer (*i.e.*, y given $\text{Id}(y)$ on that transducer).

The leader transducer then listens for all $\text{Ack}_{\leq}(x, a, b)$ facts to come in via M_{in} for each identifier x in its All-relation excluding $x = y$ with $\text{Id}(y)$ (given that y is the current leader node), and all $\leq(a, b)$ facts it has locally; in other words, it waits until all non-leader transducers have received the full order relation. Once this is the case it broadcasts $\text{P2Done}()$ via M_{snd} , additionally storing it locally via M_{in} . This concludes the second phase.

P3 Once $\text{P2Done}()$ is present on a transducer, it simulates $\Pi_{\leq}$ by using its $M_{\text{in}}^{\leq}$, $M_{\text{del}}^{\leq}$, $M_{\text{out}}^{\leq}$ and $M_{\text{snd}}^{\leq}$ mappings. This concludes the third phase.

Computation Let H be a distributed instance over σ_1 . We show that for all fair runs $\mathcal{R}_{\leq}$ of $\Pi_{\leq}$ on H , there exists a fair run $\mathcal{R}$ of Π on H such that $\text{do}(\mathcal{R})$ is well-defined and $(H, \text{do}(\mathcal{R})) \in B$.

Let $\mathcal{R}_{\leq}$ be a run of $\Pi_{\leq}$ on H . We construct a run $\mathcal{R}$ of Π on H which has $\mathcal{R}_{\leq}$ as a suffix (for phase 3 of Π). We structure the definition of $\mathcal{R}$ into repeated rounds in which all $n = |\text{netw}(H)|$ nodes become active. We always let the leader be the last transducer that becomes active. The rounds in $\mathcal{R}$ are defined as follows:

- R1: First, we add $n - 1$ configurations where each transducer that is not the leader becomes active. Each transducer in the round broadcasts all of its input relations R . Next, we have a configuration where the leader becomes active, and it broadcasts acknowledgement facts for all SndR -facts.
- R2: Next, we add another round of $n - 1$ configurations in which each non-leader transducer becomes active; it now sees that it has received AckR -facts for each relation R in its input, and sends a Ack -fact back to the leader. Then, the leader becomes active and observes Ack -facts for each of the transducers in its All-relation besides itself. It stores and broadcasts $\text{P1Done}()$ in the same configuration.
- R3: We add $n - 1$ configurations in which all non-leader transducers become active and receive $\text{P1Done}()$. Next, we add a configuration in which the leader becomes active and broadcasts $\text{Snd}_{\leq}$ -facts.

R4: Subsequently, we add $n-1$ configurations in which non-leader transducers broadcast $\text{Ack}_{\leq}(x, a, b)$ facts, where x is the node identifier of the active node for that configuration. We then add a new configuration in which the leader becomes active. It observes the $\text{Ack}_{\leq}(x, a, b)$ facts for all identifiers in its All-relation (besides the leader itself) and broadcasts and stores a $\text{P2Done}()$ fact.

R5: Lastly, all transducers become active in a round of n configurations. Each of them receive $\text{P2Done}()$.

After R5, all configurations of $\mathcal{R}_{\leq}$ are concatenated.

If $\mathcal{R}_{\leq}$ has a quiescence configuration, and hence $\text{do}(\mathcal{R}_{\leq})$ is defined, then $\mathcal{R}$ also has a quiescence configuration with $\text{do}(\mathcal{R}) = \text{do}(\mathcal{R}_{\leq})$. This follows by definition. Since $(H, \text{do}(\mathcal{R}_{\leq})) \in B$, we also have that $(H, \text{do}(\mathcal{R})) \in B$. Hence, $\mathcal{F}(\text{Id+All+Ord}) \subseteq \mathcal{F}(\text{Id+All+L})$. $\square$

Proposition 16. $\mathcal{F}(\text{Id+All+L}) \subseteq \mathcal{F}(\text{Id+All+Ord})$

Proof. Let σ_1 and σ_2 be two schemas, and let B be a behavior from σ_1 to σ_2 that is computable by an **Id+All+L**-transducer.

Transducer Construction Let Π_L be a **Id+All+L**-transducer computing B , with the mappings $M_{\text{in}}^L, M_{\text{del}}^L, M_{\text{out}}^L$, and M_{snd}^L . We construct a **Id+All+Ord**-transducer Π over Y , where Y is a transducer schema with Y_{in} and Y_{out} defined as σ_1 and σ_2 respectively, and with Y_{sys} given by the output schema of **Id+All+Ord**(Y_{in}). The Y_{mem} and Y_{msg} schemas can be derived from the mappings of Π , that we are about to define. The transducer Π works in three phases:

P1: First, the transducer with the smallest node ID according to the $\leq$ -system relation is designated as temporary leader. All nodes send their input to this intermediary leader.

P2: The temporary leader has access to H at this point, and it can get access to $(H, U) \in L(Y_{\text{in}})$. The ‘real’ leader that is present in U is then broadcast to all other nodes.

P3: The real leader mimics a **Id+All+L**-transducer such as Π_L .

P1 We now define M_{in} and M_{snd} at the same time; the mappings M_{out} and M_{del} are left empty until phase 3. If a transducer does not have both the lowest node identifier according to All and the $\leq$ -relation present in its system memory, it is not a temporary leader node, and then the M_{snd} mapping broadcasts all input facts $R(\bar{x})$ to every other transducer under a specially prefixed relation name $\text{SndR}(y, \bar{x})$, where y is the node identifier for that transducer accessible through Id.

In turn, the temporary leader uses the M_{in} mapping to store each incoming $\text{SndR}(y, \bar{x})$ fact as both a $\text{NetR}(y, \bar{x})$ fact, and $R(\bar{x})$. Furthermore, for every $\text{SndR}(\bar{x})$ fact received, the temporary leader broadcasts an $\text{AckR}(\bar{x})$ fact through M_{snd} . Once each non-leader transducer observes a $\text{AckR}(\bar{x})$ fact for every $R(\bar{x})$ fact in its local input, it broadcasts an $\text{Ack}(x)$ fact via M_{snd} , with x the node identifier obtained from $\text{Id}(x)$.

Finally, once the temporary leader node observes a $\text{Ack}(x)$ fact for every $\text{All}(x)$ fact such that x is not the leader's node identifier, it broadcasts $\text{P1Done}()$ via M_{snd} and stores it locally with M_{in} . This concludes the first phase.

P2 When the temporary leader has the fact $\text{P1Done}()$ locally, Π enters phase two. Since the leader now has all facts from all transducers, it can construct H by restricting its local state to the NetR relation and the local $R(\bar{x})$ relation annotated with the temporary leader's local identifier x . Notice that we have to strip away the Net prefix for this, we annotate all R relations with their respective node identifier. This gives us a distributed instance H stored on the leader. We can then take U where $(H, U) \in \mathbf{L}(Y_{\text{in}})$ and, for every $\text{Leader}@y(a) \in U$ broadcast $\text{SndLead}(y, a)$ facts via M_{snd} and store $\text{Leader}(a)$ locally via M_{in} if $y = x$ where $\text{Id}(x)$.

In response, the non-leader transducers store each $\text{SndLead}(y, a)$ fact locally as $\text{Leader}(a)$ via M_{in} if y matches its node identifier, and broadcast $\text{AckLead}(y)$ via M_{snd} with y the identifier of that transducer (*i.e.*, y given $\text{Id}(y)$ on that transducer).

The temporary leader transducer then listens for all $\text{Ack}\leq(x)$ facts to come in via M_{in} for each identifier x in its All-relation excluding $x = y$ with $\text{Id}(y)$ (given that y is the current leader node); in other

words, it waits until all non-leader transducers have received the new leader relation. Once this is the case it broadcasts $\text{P2Done}()$ via M_{snd} , additionally storing it locally via M_{in} . This concludes the second phase.

P3 Once $\text{P2Done}()$ is present on a transducer, it simulates Π_L by using its M_{in}^L , M_{del}^L , M_{out}^L and M_{snd}^L mappings. This concludes the third phase.

Computation Let H be a distributed instance over σ_1 . We show that, for every fair run $\mathcal{R}_L$ of Π_L on H , there exists a fair run $\mathcal{R}$ of Π on H such that $\text{do}(\mathcal{R})$ is well-defined and $(H, \text{do}(\mathcal{R})) \in B$.

Let $\mathcal{R}_L$ be a fair run of Π_L on H . We construct a run $\mathcal{R}$ of Π on H which has $\mathcal{R}_L$ as a suffix (for phase 3 of Π). We structure the definition of $\mathcal{R}$ into repeated rounds in which all $n = |\text{netw}(H)|$ nodes become active. We always let the temporary leader become active as the last transducer in a round.

Concretely, the rounds in $\mathcal{R}$ are given as follows:

- R1: First, we add $n - 1$ configurations in which every transducer that is not the temporary leader becomes active. Each transducer in this round broadcasts all of its input relations R to the lowest node identifier in the All-relation. We also add a configuration where the temporary leader becomes active, and it broadcasts acknowledgement facts for all SndR-facts.
- R2: We then add another round of $n - 1$ configurations in which each non-leader transducer becomes active; they now send a Ack-fact back to the temporary leader. Then, the temporary leader becomes active and observes Ack-facts for all transducers in its All-relation besides itself. It stores and broadcasts $\text{P1Done}()$ in the same configuration.
- R3: We add n configurations in which each transducer becomes active and receives $\text{P1Done}()$.
- R4: We then add a round of $n - 1$ heartbeat transitions for all nodes that are not the temporary leader. Next, the temporary leader becomes active and broadcasts SndLead-facts.

R5: Next, we add a round of $n - 1$ non-leader transducers becoming active which observe this leader relation, broadcasting AckLead-facts in response. Then, we add a configuration in which the temporary leader becomes active. It observes the AckLead facts for all identifiers in its All-relation (that is, besides the temporary leader itself) and broadcasts and stores a P2Done() fact.

R6: Lastly, we add n configurations in which all transducers become active. Each of them receives P2Done().

After R6, all configurations of $\mathcal{R}_L$ are concatenated.

Notice that if $\mathcal{R}_L$ has a quiescence configuration, then $\mathcal{R}$ will also have a quiescence configuration. Hence, $\text{do}(\mathcal{R})$ exists and is equal to $\text{do}(\mathcal{R}_L)$. And since $(H, \text{do}(\mathcal{R}_L)) \in B$ by definition, we also have that $(H, \text{do}(\mathcal{R})) \in B$. Therefore, $\mathcal{F}(\text{Id}+\text{All}+\text{L}) \subseteq \mathcal{F}(\text{Id}+\text{All}+\text{Ord})$. $\square$

3 Expressing All Behaviors

As a leader node can decide not only an order, but can in fact make any decision for the other nodes in the network. Unsurprisingly, **Id+All**-transducers equipped with **L** (or **Ord**, due to Theorem 12) can compute any computable behavior.

Theorem 13. $\mathcal{F}(\text{Id}+\text{All}+\text{L}) = \mathcal{B}$.

Notice that $\mathcal{F}(\text{Id}+\text{All}+\text{L}) \subseteq \mathcal{B}$ is trivial, since $\mathcal{F}(\text{Id}+\text{All}+\text{L})$ is a set of behaviors. The other direction is shown in Proposition 17. Intuitively, **Id**, **All**, and **L** allow the network to simulate a centralized computer. The leader node collects the complete distributed input, computes an output, and redistributes that output to the appropriate nodes.

Proposition 17. $\mathcal{B} \subseteq \mathcal{F}(\text{Id}+\text{All}+\text{L})$

Proof. Let σ_1 and σ_2 be two schemas, and let B be a behavior from σ_1 to σ_2 . We define an **Id+All+L**-transducer computing B .

Transducer Construction Let Y be a transducer schema such that $Y_{\text{in}} = \sigma_1$, $Y_{\text{out}} = \sigma_2$, and such that Y_{sys} is the output schema for the behavior **Id+All+L**(Y_{in}). The schemas Y_{mem} and Y_{msg} can be inferred from the mapping definitions we will give below. The **Id+All+L**-transducer Π over Y , which we will define next, operates in three distinct phases:

- P1*: Each transducer sends its fragment of the input to the leader node.
- P2*: The leader node computes B , as it can construct the input instance, and it redistributes the output accordingly.
- P3*: Once the output is in the correct place according to B , each node places its output facts into the output instance.

P1 We define M_{in} and M_{snd} at the same time, M_{del} is the empty mapping, and the definition of M_{out} is postponed to P3. Let H be some distributed input instance over Y_{in} , and let x be a node identifier in $\text{netw}(H)$.

For every $R(\bar{y}) \in H(x)$ such that x is not the identifier of the leader, we broadcast a fact $\text{SndR}(x, \bar{y})$ with M_{snd} . The leader node, via M_{in} , stores $\text{CopyR}(x, \bar{y})$ for every received $\text{SndR}(x, \bar{y})$. For all $\text{CopyR}(x, \bar{y})$ facts present on the leader, it broadcasts $\text{AckR}(x, \bar{y})$ via M_{snd} . Then, in response, if a non-leader node receives a $\text{AckR}(x, \bar{y})$ -fact for every one of its $R(\bar{y})$ -facts, it broadcasts $\text{P1Ack}(x)$ via M_{snd} . Finally, once the leader node has obtained a $\text{P1Ack}(x)$ for every non-leader identifier x , it stores $\text{P1Done}()$ via M_{in} .

P2 Once the leader has $\text{P1Done}()$, it can define an H by letting $R@x(\bar{y}) \in H$ if $\text{CopyR}(x, \bar{y})$ is stored locally, or if $R(\bar{y})$ is stored locally and x is the identifier of the leader node itself.

We can then let $(H, K) \in B$ and send each fact $S@x(\bar{y}) \in K$ to all other nodes as $\text{SndS}(x, \bar{y})$ via the M_{snd} mapping. Additionally, the leader node stores those $S@x(\bar{y})$ -facts where x is the leader node identifier as $\text{CopyS}(\bar{y})$ in its storage via M_{in} . The non-leader nodes also store $\text{CopyS}(\bar{y})$ via M_{in} upon getting each $\text{SndS}(x, \bar{y})$ where x is their own node identifier. Furthermore, each non-leader node broadcasts $\text{AckS}(x, \bar{y})$ -facts for each received fact via M_{snd} .

Once the leader node receives $\text{AckS}(x, \bar{y})$ -facts for every $\text{S}@x(\bar{y}) \in K$, it stores $\text{P2Done}()$ via M_{in} , and broadcasts $\text{SndP2Done}()$ via M_{snd} . All non-leader nodes store $\text{P2Done}()$ via M_{in} every time they receive a $\text{SndP2Done}()$ fact.

P3 Once a node has $\text{P2Done}()$, it outputs $\text{S}(\bar{y})$ for every $\text{CopyS}(\bar{y})$ in its input instance, via M_{out} . This concludes Π 's construction.

Computation Let H be a distributed instance over Y_{in} . We show that for all fair runs $\mathcal{R}$ of Π on H , it is the case that $\text{do}(\mathcal{R})$ exists, and $(H, \text{do}(\mathcal{R})) \in B$.

Let $\mathcal{R}$ be a fair run of Π on H . Since $\mathcal{R}$ is fair, we can assume without loss of generality that computation is organized in repeating rounds of $|\text{netw}(H)|$ configurations where every node becomes active once. We additionally assume that the leader is always the last node to become active in a round. Notice that this indeed preserves generality as we can model such a round by splitting it into one round in which all non-leader nodes become active and where the leader node does a heartbeat transition, and one round where the leader node is active and where all non-leader nodes perform a heartbeat transition.

The following rounds occur in $\mathcal{R}$ by construction of Π :

- R1: In the first round, all non-leader nodes send their input relations $\text{R}(\bar{y})$ to the leader node, and the leader node sends back an acknowledgement for every received fact.
- R2: Next, all non-leader nodes receive these acknowledgements and return an acknowledgement to the leader. The leader receives these acknowledgements and concludes the first phase by storing $\text{P1Done}()$.
- R3: The non-leader nodes perform a heartbeat transition. Furthermore, the leader node sends output facts $\text{SndS}(x, \bar{y})$ to all non-leader nodes, storing $\text{CopyS}(\bar{y})$ -facts for the fragment of the output that belongs on the leader.
- R4: The non-leader nodes become active and store $\text{CopyS}(\bar{y})$ facts for each $\text{SndS}(x, \bar{y})$ if x matches the node identifier of this non-leader transducer. Additionally, each non-leader node sends

an acknowledgement back to the leader for each received fact. The leader stores $P2Done()$ upon receiving the acknowledgements, and broadcasts $SndP2Done()$.

R5: All non-leader nodes become active and receive $SndP2Done()$, in response, they store $P2Done()$. The leader node outputs $S(\bar{y})$ for every $CopyS(\bar{y})$.

R6: The non-leader nodes become active one last time and output $S(\bar{y})$ for every $CopyS(\bar{y})$. The leader node performs a heartbeat transition.

All remaining rounds consist entirely of heartbeat transitions, and therefore $\mathcal{R}$ reaches quiescence after R6, hence the distributed output $do(\mathcal{R})$ is well-defined.

To see that $(H, do(\mathcal{R})) \in B$, we point to the fact that the leader node computes $(H, K) \in B$ and places the relevant parts of K on each node. In conclusion, we have that Π computes B , and hence we have that $B \subseteq \mathcal{F}(\mathbf{Id} + \mathbf{All} + \mathbf{L})$. $\square$

4 Termination-Aware Transducers

In the next chapter, we encounter transducers with constraints that cannot express all behaviors, and then we are interested in comparing classes of transducers parameterized by these constraints.

A hindering feature of transducers for this purpose is that a transducer can compute a behavior (*c.f.*, Definition 23) without knowing that it has finished computing the behavior (i.e., without knowing that no new facts will be inserted in the output relations in the future). As a consequence, even though a constraint C itself is computable using a transducer parameterized by another constraint C' , i.e., $C(\sigma) \in \mathcal{F}(C')$ for every σ , it does not follow in general that $\mathcal{F}(C) \subseteq \mathcal{F}(C')$. This is for example the case with $\mathbf{Id}$ -transducers computing the $\mathbf{All}$ relation on inputs over $\{\text{Node}[0]\}$.

The following proposition provides a set of restrictions that *do* allow such reasoning. For a constraint C , we denote by $\bar{C}$ its extension providing for every relation R with $acc(R) = input$ a relation $\bar{R}$ containing the complement of R . Formally, we will write $\bar{\sigma}$ to

denote the schema $\{\bar{R}[k] \mid R[k] \in \sigma \text{ with } \text{acc}(R) = \text{input}\}$ and with $\text{acc}(\bar{R}) = \text{input}$ for all its relations. Then, for every σ and with $\mathbf{C}(\sigma)$ a behavior from σ to σ' , $\bar{\mathbf{C}}(\sigma)$ is the behavior from σ to $\sigma' \cup \bar{\sigma}'$ with $(H, U) \in \mathbf{C}(\sigma)$ iff $(H, U \cup \{\bar{R}@x(\bar{a}) \mid R@x(\bar{a}) \notin H \text{ with } R \in \sigma, \text{acc}(R) = \text{input}, \bar{a} \subseteq \text{Adom}(H) \text{ and } x \in \text{netw}(H)\})$.

The intuition underlying Proposition 18 is that materializing a relation R and its complement in parallel in an inflationary way, allows a transducer to recognize when R is complete with respect to its local active domain. Since relations R with $\text{acc}(R) = \text{input}$ are anyway only readable as far as the local active domain allows, this is sufficient to emulate the presence of R .

Proposition 18. For disjoint constraint $\mathbf{C}_1$ and $\mathbf{C}_2$ with for every σ having $\bar{\mathbf{C}}_1(\sigma) \in \mathcal{F}(\mathbf{C}_2)$ and with $\text{acc}(R) = \text{input}$ for every $R \in \mathbf{C}_1(\sigma)$, we have $\mathcal{F}(\mathbf{C}_1 + \mathbf{C}_2) \subseteq \mathcal{F}(\mathbf{C}_2)$.

Proof. Let B be a behavior in $\mathcal{F}(\mathbf{C}_1 + \mathbf{C}_2)$, and let Π be some $\mathbf{C}_1 + \mathbf{C}_2$ -transducer over a schema $Y = (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{mem}}, Y_{\text{msg}}, Y_{\text{sys}})$ computing this behavior B .

Let Π' be the $\mathbf{C}_2$ -transducer that computes $\bar{\mathbf{C}}(Y_{\text{in}})$. Without loss of generality, we will assume that it is over a schema

$$Y' = (Y'_{\text{in}}, Y'_{\text{out}}, Y'_{\text{mem}}, Y'_{\text{msg}}, Y'_{\text{sys}})$$

that is disjoint from Y .

We construct a $\mathbf{C}_2$ -transducer Π'' that simulates Π and Π' . More precisely, Π'' is defined over $Y = (Y_{\text{in}}, Y_{\text{out}}, Y''_{\text{mem}}, Y''_{\text{msg}}, Y'_{\text{sys}})$ such that $Y''_{\text{msg}} := Y_{\text{msg}} \cup Y'''_{\text{msg}}$ with Y'''_{msg} a primed copy of Y_{msg} , $Y''_{\text{mem}} := Y_{\text{sys}} \cup Y_{\text{mem}} \cup Y_{\text{msg}} \cup Y'_{\text{mem}}$.

Intuitively, transducer Π'' will simulate both Π and Π' , but in such a way that transitions of Π' are only executed over states that are indistinguishable from the states of a $\mathbf{C}_1 + \mathbf{C}_2$ -transducer. Formally, every mapping of Π executes in parallel the respective mapping of Π' ; then checks if the local memory relations populated by Π' are each other's complement w.r.t. the current active domain of the entire local instance. If the latter is true, the respective mapping of Π is applied as well (otherwise, the mappings of Π are not applied). In addition, mapping M_{in} of Π'' will also store arriving facts over

Y'''_{msg} in respective memory relations of Y_{msg} ; and M_{del} removes the facts in relations of Y_{msg} if the earlier mentioned check is true.

The intuition of the latter two is that we need a mechanism to ensure that no messages intended for the simulation of Π get lost during transitions in which the check is false and thus the mappings of Π are not executed.

For correctness, we notice every run produced by the constructed transducer over some distributed instance H yields a valid run by Π . For this to see notice that Π' does not interfere with Π due to the disjoint schemas (up to the output relations of Π' which serve as system relations for Π), that at any point in time that a transition of Π is executed, the system relation is computed as far as Π can read it due to the $\text{acc}(R) = \text{input}$ access constraints.

Furthermore, since every fair run will eventually reach a quiescence configuration for the relations produced by Π' , it is guaranteed that the constructed transducer will at some point start executing the mappings of Π and from then on keep executing these mappings for every further transition, thus yielding a fair run of Π which is guaranteed to reach a quiescence point with desired distributed output. $\square$

To extend Proposition 18 to behaviors $\mathbf{C}_1(\sigma)$ having also relations R with $\text{acc}(R) = \text{output}$, we propose a stricter form of transducer computation. We consider the following formal definition.

Definition 24. A transducer Π is *termination-aware* if it is defined over a transducer schema Y with dedicated relation $\text{Terminated}[0] \in Y_{\text{mem}}$, and the property that for every distributed instance H over Y_{in} , every fair run $\mathcal{R}$ of Π over H , and every node $x \in \text{netw}(H)$, there should be a relation Terminated that is true on x at some point in the run, and once it is true:

- (i) it remains true on x forever in the run; and
- (ii) no further tuples are inserted in the local output relations $R \in Y_{\text{out}}$ on node x having $\text{acc}(R) = \text{output}$.

Intuitively, when the Terminated flag on a node switches to true, it follows that the node's output relation is complete up to perhaps

some output relations R with $acc(R) = input$.

Example 16. For an example of a termination-aware transducer, consider a scenario where the network is always organized with a specific graph overlay topology (which we assume to be connected for this example) and nodes initially know only of their direct neighbors in the topology.

A constraint **Top** with associated behavior **Top**(σ) from σ to $\{\text{Neighbour}[1]\}$ could be used to formalize this scenario, *i.e.*, $(H, U) \in \text{Top}(\sigma)$ iff:

$$U := \{\text{Neighbor}@x(y) \mid y \text{ being a neighbor of } x \text{ in } \lambda\}$$

for some topology λ of $\text{netw}(H)$ with appropriate properties.

Then, $\text{All}(\sigma) \in \mathcal{F}_T(\text{Top})$ because nodes can broadcast their id to all other nodes in the network, and synchronize with nodes of received ids to ensure also all their neighbors are known. When a node knows the entire network and has synchronized with all those nodes, it can set the Terminated flag.

Henceforth, we refer by $\mathcal{F}_T(\mathbf{C})$ to the class of behaviors computable with a termination-aware $\mathbf{C}$ -transducer. We remark that a termination-aware transducer can still be involved in interactions essential for other nodes to complete their output, even after the Terminated flag is set. The following theorem shows how the set $\mathcal{F}_T(\mathbf{C})$ is used to reason about the relationship between classes of behaviors computable with different constraints.

Theorem 14. For every pair of constraints $\mathbf{C}$ and $\mathbf{C}'$:

1. $\mathcal{F}_T(\mathbf{C}) \subseteq \mathcal{F}(\mathbf{C})$; and
2. $\mathcal{F}(\mathbf{C} + \mathbf{C}') \subseteq \mathcal{F}(\mathbf{C}')$ if $\overline{\mathbf{C}}(\sigma) \in \mathcal{F}_T(\mathbf{C}')$ for every σ .
3. $\mathcal{F}(\mathbf{C}) \subseteq \mathcal{F}(\mathbf{C}')$ if $\mathbf{C}(\sigma) \in \mathcal{F}_T(\mathbf{C}')$ for every σ and with $acc(R) = output$ for every $R \in \sigma$.

The proof of Theorem 14 follows directly from Proposition 19 and Proposition 20, which are given next.

Proposition 19. For each $\mathbf{C}$, we have that $\mathcal{F}_T(\mathbf{C}) \subseteq \mathcal{F}(\mathbf{C})$.

Proof. Let $\mathbf{C}$ be an arbitrary constraint, and let $B \in \mathcal{F}_T(\mathbf{C})$ be a behavior from σ_1 to σ_2 . Furthermore, let Π be a termination-aware $\mathbf{C}$ -transducer over $\mathbf{Y}$ with $Y_{\text{in}} := \sigma_1$ and $Y_{\text{out}} := \sigma_2$, computing B . Note that, by definition of $\mathcal{F}_T(\mathbf{C})$ each termination-aware $\mathbf{C}$ -transducer computing B is also a $\mathbf{C}$ -transducer computing B . Therefore, we have that $B \in \mathcal{F}(\mathbf{C})$. $\square$

The following result states that if we can compute $\overline{\mathbf{C}}$ using a terminating $\mathbf{C}'$ -transducer, then $\mathbf{C}$ adds no additional expressive power on top of $\mathbf{C}'$. Notice that (3) in Theorem 14 is a special case of this result.

Proposition 20. For all pairs of constraints $\mathbf{C}$ and $\mathbf{C}'$, $\mathcal{F}(\mathbf{C}+\mathbf{C}') \subseteq \mathcal{F}(\mathbf{C}')$ if $\overline{\mathbf{C}}(\sigma) \in \mathcal{F}_T(\mathbf{C}')$ for every σ .

Proof. Let $\mathbf{C}$ and $\mathbf{C}'$ be configuration constraints as in the proposition. We proceed by proving for an arbitrary behavior $B \in \mathcal{F}(\mathbf{C})$ that $B \in \mathcal{F}(\mathbf{C}')$. For this, let σ_1 and σ_2 be the input and output schema of B and let $\Pi = (\mathbf{C}, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$ be a $\mathbf{C}$ -transducer computing B . Let $\mathbf{Y} = (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{mem}}, Y_{\text{msg}}, Y_{\text{sys}})$ be the schema that Π is over. We remark that $Y_{\text{in}} := \sigma_1$ and $Y_{\text{out}} := \sigma_2$.

From the proposition it follows that there exists a termination-aware $\mathbf{C}'$ -transducer Π' over some transducer schema

$$\mathbf{Y}' = (Y_{\text{in}}, Y_{\text{sys}} \cup \overline{Y_{\text{sys}}}, Y'_{\text{mem}}, Y'_{\text{msg}}, Y'_{\text{sys}})$$

computing behavior $\overline{\mathbf{C}}(Y_{\text{in}})$ from Y_{in} to $Y_{\text{sys}} \cup \overline{Y_{\text{sys}}}$, on which we will base our construction. Let $Y'_{\text{all}} := Y_{\text{in}} \cup Y_{\text{sys}} \cup \overline{Y_{\text{sys}}} \cup Y'_{\text{mem}} \cup Y'_{\text{msg}} \cup Y'_{\text{sys}}$.

Transducer Construction We now construct the transducer Π'' over transducer schema

$$\mathbf{Y}'' = (Y_{\text{in}}, Y_{\text{out}}, Y''_{\text{mem}}, Y''_{\text{msg}}, Y''_{\text{sys}})$$

with, $Y''_{\text{mem}} := Y'_{\text{out}} \cup Y'_{\text{mem}} \cup Y_{\text{mem}}$, and $Y''_{\text{msg}} := Y'_{\text{msg}} \cup Y_{\text{msg}}$. Lastly,

$$\Pi'' = (\mathbf{C}', M''_{\text{out}}, M''_{\text{in}}, M''_{\text{del}}, M''_{\text{snd}})$$

is defined with the following mappings:

- The transducer emulates Π' ; stores the active domain values received from other nodes; stores all messages intended for Π'' ; and, if the terminated fact is present and complemented relations are indeed all each others complement under the current active domain, then emulate Π over the current instance, but taking as messages the stored messages.

Formally, we have that $(I, J) \in M_{\text{in}}$ iff J is a minimal instance with

1. $J' \subseteq J$ for some J' with $(I|_{Y'_{\text{all}}}, J') \in M'_{\text{in}}$;
2. $\text{AdomSnd}(a) \in I$ implying $\text{Adom}(a) \in J$; and
3. $R(\bar{a}) \in I$ and $R \in Y_{\text{msg}}$ implying $\text{StoredMsg}_R(\bar{a}) \in J$;
4. if $\text{Terminated}() \in I$, and $R(\bar{a}) \in I$ iff $\bar{R}(\bar{a}) \notin I$ for every $R[k] \in Y_{\text{sys}}$ and $\bar{a} \subseteq \text{Adom}(I)^k$, then $J' \subseteq J$ for some J' with $(I') \in M_{\text{in}}$ with I' the instance obtained by renaming in I the relation StoredMsg_R to R and removing all facts not in the schema of Π .

- Intuitively, the transducer broadcasts its active domain values, emulates Π' , and (as before) if all constraints hold also emulate Π , but over the stored messages. $(I, J) \in M''_{\text{snd}}$ iff J is a minimal instance with

5. $a \in \text{Adom}(I)$ implying $\text{AdomSnd}(a) \in J$;
6. $J' \subseteq J$ for some J' with $(I|_{Y'_{\text{all}}}, J') \in M'_{\text{snd}}$;
7. if $\text{Terminated}() \in I$, and $R(\bar{a}) \in I$ iff $\bar{R}(\bar{a}) \notin I$ for every $R[k] \in Y_{\text{sys}}$ and $\bar{a} \subseteq \text{Adom}(I)^k$, then $J' \subseteq J$ for some J' with $(I') \in M_{\text{snd}}$ with I' the instance obtained by renaming in I the relation StoredMsg_R to R and removing all facts not in the schema of Π .

- $(I, J) \in M''_{\text{del}}$ iff

8. $J' \subseteq J$ for some J' with $(I|_{Y'_{\text{all}}}, J') \in M'_{\text{del}}$;
9. if $\text{Terminated}() \in I$, and $R(\bar{a}) \in I$ iff $\bar{R}(\bar{a}) \notin I$ for every $R[k] \in Y_{\text{sys}}$ and $\bar{a} \subseteq \text{Adom}(I)^k$, then $J' \subseteq J$ for some J' with $(I') \in M_{\text{del}}$ with I' the instance obtained by renaming in I

the relation StoredMsg_R to R and removing all facts not in the schema of Π .

- $(I, J) \in M''_{\text{out}}$ iff
 10. $J' \subseteq J$ for some J' with $(I|_{Y'_{\text{all}}}, J') \in M'_{\text{out}}$;
 11. if $\text{Terminated}() \in I$, and $R(\bar{a}) \in I$ iff $\bar{R}(\bar{a}) \notin I$ for every $R[k] \in Y_{\text{sys}}$ and $\bar{a} \subseteq \text{Adom}(I)^k$, then $J' \subseteq J$ for some J' with $(I') \in M_{\text{out}}$ with I' the instance obtained by renaming in I the relation StoredMsg_R to R and removing all facts not in the schema of Π .

It can be verified straightforwardly that the transducer is correct, because Π is guaranteed to set the $\text{Terminated}()$ flag and create the complementary relations. Furthermore, from the perspective of Π' , the computation proceeds as in a normal run. $\square$

5 Summary

In this chapter we introduced the following concepts and results.

- We defined what it means for a behavior to be computed by a transducer, and what it means for a behavior to be characterized by a transducer. The former is represented by the class $\mathcal{F}(\mathbf{C})$, and the latter is represented by the class $\mathcal{G}(\mathbf{C})$ for some constraint $\mathbf{C}$.
- We showed that leader election is not computable by an **Id+All**-transducer, which indicates that the model must be extended with some way to break symmetry. We created both **Id+All+L** and **Id+All+Ord**-transducers for this purpose.
- We showed that the aforementioned **Id+All+L** and **Id+All+Ord**-transducers can compute all distributed behaviors.
- Finally, we introduced the class of termination-aware transducers, which are capable of deciding when they have reached a quiescence state. We represent this by the class $\mathcal{F}_T(\mathbf{C})$ for some constraint $\mathbf{C}$.

Coordination-Freedom

In this chapter, we investigate which classes of behaviors require coordination. We do so by first looking at the class of all behaviors, which includes non-determinism. Next, we restrict ourselves to those behaviors that are deterministic, and demonstrate that we can prove results from prior work using generalized transducer networks. Lastly, we present a hierarchy of behavior classes based on how much coordination is required to compute them.

First, we have to concretely define coordination-free behaviors:

Definition 25. We call a behavior *coordination-free* if it is computable with a **Id+Ord**-transducer. We say that a behavior requires a *weak form of coordination* if there exists a configuration constraint $\mathbf{C}$ with $\text{All}(Y_{\text{in}}) \notin \mathcal{F}_T(\mathbf{C})$ such that this behavior is computable with a $\mathbf{C}$ -transducer Π with schema Y .

This definition of coordination-freedom is inspired by an important insight in [8] which states that queries computable without coordination (for a specific definition of coordination) are precisely the queries that can be computed with **Id**-transducers. We additionally include the **Ord** constraint, as the **Id**-transducers are otherwise not expressive enough to pick the correct output of a behavior in case the full network is not known yet by all nodes, despite that behavior intuitively seeming monotone-like. For instance, see Example 17 ahead, where Top_{span} cannot be computed by an **Id**-transducer despite the behavior being node-monotone. Given (deterministic) queries, this order is not required, and **Id**-transducers are sufficient. We prove this in Section 8.2, where we show that the existing results from [7, 8] carry over to this model.

1 Non-Deterministic Computation

We define and study the computability for class(es) of (*strictly*) *node monotone* behaviors. Intuitively, these behaviors should be able to cope with arbitrary (yet finite) delays in communication. Such delays can cause a subset of the network to compute output before knowing all nodes that are present in the network. To deal with this, we need to guarantee that output computed over a subset of the network can be combined with output from larger subsets.

Definition 26. We define a behavior B from σ_1 to σ_2 to be

- *node monotone* if for every pair of distributed instances H_1 and H_2 over σ_1 , with $\text{netw}(H_1) \cap \text{netw}(H_2) = \emptyset$, and every $(H_1, K_1) \in B$ and $(H_2, K_2) \in B$ there exists a $(H_1 \cup H_2, K) \in B$ such that $K_1 \cup K_2 \subseteq K$, and
- *strictly node monotone* if, in addition, for each such $(H_1 \cup H_2, K) \in B$ it is the case that $K_1 \cup K_2 \subseteq K$.

We denote the classes of behaviors corresponding to these properties by $\mathcal{M}_{\text{node}}$ and $\mathcal{M}_{\text{node-}\forall}$, respectively. We name these properties *node monotone* as they extend the notion of monotonicity to a non-deterministic setting where facts are tied to computing *nodes*. Indeed, for a (deterministic) query q over a schema σ to be monotone we require that, if $q(I_1) = J_1$ and $q(I_2) = J_2$ for instances I_1, I_2 over σ then $J_1 \cup J_2 \subseteq q(I_1 \cup I_2)$. The definition above extends this to binary relations, with the added requirement that for the distributed instances H_1 and H_2 given in the definition, it must be that H_2 originates from *node(s) residing in a part of the network different from $\text{netw}(H_1)$* . This intuitively reflects that H_1 and H_2 are combined after a network partition is resolved.

Example 17. For an example of a node monotone behavior, consider the variant $\mathbf{Top}_{\text{span}}$ of $\mathbf{Top}$ that only considers network topologies representing a spanning tree over all nodes. Indeed, for any σ , the behavior $\mathbf{Top}_{\text{span}}(\sigma)$ is node-monotone. To see this, take two arbitrary distributed instances H_1 and H_2 over σ with disjoint network (*i.e.*, $\text{netw}(H_1) \cap \text{netw}(H_2) = \emptyset$), and any interpretation of the system relations K_1 and K_2 with

$(H_1, K_1), (H_2, K_2) \in \mathbf{Top}_{\text{span}}$. Since K_1 and K_2 represent spanning trees over $\text{netw}(H_1)$ and $\text{netw}(H_2)$, respectively, the distributed instance

$$K = K_1 \cup K_2 \cup \{\text{Neighbor}@x(y), \text{Neighbor}@y(x)\}$$

taking any node $x \in \text{netw}(H_1)$ and $y \in \text{netw}(H_2)$ represents a spanning tree for network $\text{netw}(H_1) \cup \text{netw}(H_2)$. It is now easy to see that $K_1 \cup K_2 \subseteq K$ and $(H_1 \cup H_2, K) \in \mathbf{Top}_{\text{span}}(\sigma)$. Behavior $\mathbf{Top}_{\text{span}}(\sigma)$ is not strictly node monotone, as clearly (if the network of $\text{netw}(H_1)$ or $\text{netw}(H_2)$ is large enough) there exist also spanning trees for $\text{netw}(H_1) \cup \text{netw}(H_2)$ with completely different structure.

The variant of **Top** requiring that the topology is complete (*i.e.*, fully connected) is an example of a strict node monotone behavior, as can be easily verified.

While strict node monotonicity is a rather strong condition, it allows us to explore the capabilities of **Id**-transducers. It turns out that any strictly node monotone behavior computed by an **Id+All**-transducer can also be computed by a transducer without **All**.

Proposition 21. $\mathcal{F}(\mathbf{Id+All}) \cap \mathcal{M}_{\text{node-v}} \subsetneq \mathcal{F}(\mathbf{Id})$.

Proof. Let $B \in \mathcal{F}(\mathbf{Id+All}) \cap \mathcal{M}_{\text{node-v}}$ be a behavior from σ_1 to σ_2 , and let Y be a transducer schema such that $Y_{\text{in}} = \sigma_1$, $Y_{\text{out}} = \sigma_2$, and with Y_{sys} the output schema of $\mathbf{Id}(\sigma_1)$. The definition of Y_{mem} and Y_{msg} can be inferred from the mappings we define later.

Transducer Construction We define an **Id**-transducer Π over Y , repeatedly cycling through the following phases:

- P1:* Build a quorum of transducers which agree on a (potentially) partial **All**-relation. To achieve this, transducers use a **gossip protocol**.
- (a) Transducers broadcast their own identifier, and broadcast all identifiers they receive from other transducers. Each gossip message is tagged with the identifier of the sending transducer.

When transducers receive a *gossip message*, they store it locally. This builds a local data structure indicating which transducers have seen which identifiers.

- (b) Once a transducer locally determines that other transducers have an identical record of identifiers, it will cease sending gossip messages and instead send a *propose message* to all such transducers.

Upon receiving a propose message, a transducer will locally determine if that sending transducer has an identical record of identifiers. If this is the case, it will cease sending gossip messages and send out its own propose message back. Otherwise, it still has in-flight gossip messages left to handle, and then it waits.

- (c) Transducers which have sent a propose message to every identifier in their local record, and have obtained a propose message in return for that identifier, are in a quorum. Their local identifier record is then used to create an All-relation.

P2: Each transducer executes *B*, and writes the output to its output instance.

We now define the mappings M_{in} , M_{snd} , M_{del} and M_{out} (postponed to *P2*) in tandem, for each of the phases.

P1 In part (a), as long as there is no QuorumAdom(*a*)-fact present (defined in phase *P1b*) where *a* is an active domain value, each transducer sends a SndId(*x*, *x*)-fact via M_{snd} , where *x* is the node identifier of the transducer. The first attribute of this fact represents the origin of the message, the second represents an identifier this transducer knows about. Secondly, the transducer stores All(*x*) via M_{in} .

For each received SndId(*y*, *z*)-fact, a transducer with identifier *x* stores All(*z*) and RemoteAll(*y*, *z*) with M_{in} and broadcasts SndId(*x*, *z*) with M_{snd} .

Next, we continue with (b). If there exists a node identifier *y* such that, for all facts All(*z*) there exists a RemoteAll(*y*, *z*) and vice versa, then this transducer sends Propose(*x*, *y*) via M_{snd} , where *x* is the

identifier of the sending transducer. Additionally, it adds a fact $\text{QuorumAdom}(a)$ via M_{in} for every active domain value a , effectively stopping the transmission of gossip messages until new facts have been derived.

Upon receipt of a $\text{Propose}(x, y)$, the transducer with id y checks if there exists $\text{RemoteAll}(x, z)$ for all $\text{All}(z)$ facts (and vice versa). If this is the case, the node stores $\text{RecvPropose}(x)$ with M_{in} and sends back its own $\text{Propose}(y, x)$, and adds $\text{QuorumAdom}(a)$ via M_{in} facts for all its active domain values a . Otherwise, it does nothing and waits.

Lastly, for (c), if for all facts $\text{All}(y)$ except $y = x$ where x is the id of the current transducer, there exists a $\text{RecvPropose}(y)$ and vice versa, then this node stores $\text{P1Done}(a)$ for all active domain values a using M_{in} .

P2 If this transducer has a $\text{P1Done}(a)$ fact for every active domain value, then it computes its part of B and records its output via the mapping M_{out} . That is, the mapping M_{out} is defined to compute B only if $\text{P1Done}(a)$ is present for all a . Once B has been computed for the current input instance, we add $\text{QuorumAdom}(a)$ and $\text{P1Done}(a)$ facts to our delete instance via M_{del} . This makes us cycle back to the first phase.

Computation We show that for every fair run $\mathcal{R}$ of Π we have that $(H, \text{do}(\mathcal{R})) \in B$. Let $\mathcal{R}$ be a fair run of Π on H . There are three general types of finite delays or partitions that can happen between nodes in the network, based on the size of the set of nodes that are in common between two islands of disconnected nodes. We remark that we assume there are two sets of disconnected nodes without loss of generality. Assume the networks $\mathcal{N}_1 \subseteq \text{netw}(H)$, $\mathcal{N}_{1 \cup 2} \subseteq \mathcal{N}_1$, and $\mathcal{N}_2 = \mathcal{N}_{1 \cup 2} \cup \text{netw}(H) \setminus \mathcal{N}_1$, we outline the three possibilities as follows:

1. $\mathcal{N}_{1 \cup 2} = \emptyset$, therefore $\mathcal{N}_1 \cap \mathcal{N}_2 = \emptyset$, hence $\mathcal{N}_1$ and $\mathcal{N}_2$ have nothing in common;
2. $\mathcal{N}_{1 \cup 2} \neq \emptyset$ with $\mathcal{N}_{1 \cup 2} \subsetneq \mathcal{N}_1$ and $\mathcal{N}_{1 \cup 2} \subsetneq \mathcal{N}_2$, hence both $\mathcal{N}_1$ and $\mathcal{N}_2$ share a non-empty proper subset of $\mathcal{N}_1$ (or $\mathcal{N}_2$); and
3. $\mathcal{N}_{1 \cup 2} = \mathcal{N}_1 = \mathcal{N}_2$, meaning that there is no partition.

In case (3), we note that Π trivially reaches a quiescence state after P1 and P2 are completed once. We focus our attention to cases (1) and (2).

For case (1), assume that $\mathcal{N}_{1 \cup 2} = \emptyset$. And, without loss of generality, assume that $\mathcal{R}$ starts with nodes in $\mathcal{N}_1$ becoming active, while nodes in $\mathcal{N}_2$ are arbitrarily yet finitely delayed. Due to the construction of Π there will be some configuration in $\mathcal{R}$ where each node in $\mathcal{N}_1$ has constructed a quorum after P1.

Next, $\mathcal{R}$ reaches a configuration in which it has computed some distributed output $(H_1, \text{do}(\rho_i)) \in B$ where

$$H_1 = \{R @ x(\bar{a}) \in H \text{ with } x \in \mathcal{N}_1 \text{ for all } R[k] \in Y_{\text{in}}\}$$

due to Π 's construction. Note that $\text{do}(\rho_i)$ is written to the output instances of all nodes in $\mathcal{N}_1$, and can no longer be retracted.

Next, the delay between $\mathcal{N}_1$ and $\mathcal{N}_2$ is resolved, and $\mathcal{N}_2$ receives all messages from nodes in $\mathcal{N}_1$. This results in a configuration in $\mathcal{R}$ where a quorum is reached for all of $\text{netw}(H)$. Due to the construction of Π and the strict node monotonicity of B , there will be a configuration ρ_j in $\mathcal{R}$ with $j > i$ where the output of ρ_i is extended (i.e., $\text{do}(\rho_i) \subseteq \text{do}(\rho_j)$) and such that $(H, \text{do}(\rho_j)) \in B$. At this point, no more messages are read since the quorum spans all of $\text{netw}(H)$, and therefore $\text{do}(\rho_j) = \text{do}(\mathcal{R})$.

For case (2), let $\mathcal{N}_{1 \cup 2} \neq \emptyset$ and $\mathcal{N}_{1 \cup 2} \subsetneq \mathcal{N}_1$. First, without loss of generality, assume that $\mathcal{R}$ starts with $\mathcal{N}_1$ becoming active while nodes in $\mathcal{N}_2 \setminus \mathcal{N}_{1 \cup 2}$ are arbitrarily yet finitely delayed. Nodes in $\mathcal{N}_1$ exchange all their identifiers and build a record of each other's All-relations due to the construction of Π but they stop right before they can make a quorum. Then, nodes in $\mathcal{N}_2$ become active while the nodes in $\mathcal{N}_1$ are arbitrarily (but finitely) delayed. They similarly exchange their identifier and build a record of each other's All-relations. At this point, nodes in $\mathcal{N}_1$ have a record of all of $\mathcal{N}_1$, nodes in $\mathcal{N}_2$ have a record of all of $\mathcal{N}_2$, and nodes in $\mathcal{N}_{1 \cup 2}$ know all of $\text{netw}(H)$. Neither subset computes B since neither has quorums.

Next, we have the situation where $\mathcal{N}_1$ becomes active while $\mathcal{N}_2$ is delayed (the case where $\mathcal{N}_2$ is before $\mathcal{N}_1$ is analog). All nodes in $\mathcal{N}_1$ reach a quorum and compute B and write it to their output. However, the nodes in $\mathcal{N}_2$ still do not know if they have a quorum since

$\mathcal{N}_{1 \cup 2}$ has not responded yet. Hence, assume $\mathcal{N}_{1 \cup 2}$ becomes active again, this causes the missing All-facts to be sent to nodes in $\mathcal{N}_1 \setminus \mathcal{N}_2$. At this point $\mathcal{N}_2$ knows all of $\text{netw}(H)$ and computes B as well. Due to strict node monotonicity of B the outputs remain correct and combinable, hence $(H, \text{do}(\mathcal{R})) \in B$. $\square$

Transducers with order and identity constraints are entirely expressive enough to characterize all node monotone behaviors.

Theorem 15. $\mathcal{G}(\text{Id+Ord}) = \mathcal{M}_{\text{node}}$.

The proof of this theorem is postponed until a later section in this chapter titled “Proof of Theorem 15”.

We can further generalize this observation by noting a relationship between the power of the transducer network, captured in a constraint $\mathbf{C}$, and the preconditions placed upon the distributed instances being mapped by the behavior. Indeed, baking in progressively stronger constraints allows for progressively weaker forms of node monotonicity.

In the below definition we use the following notation. For a distributed instance H and subset $X \subseteq \text{netw}(H)$ of its nodes, let $H_X := \{R @ x(\bar{a}) \in H \mid x \in X\}$. Then, we write $H \parallel_X$ to denote

$$\{R @ x(\bar{a}) \in H_X \mid \text{either, } \text{acc}(R) = \text{output} \text{ or, } \text{acc}(R) = \text{input} \text{ and } \bar{a} \subseteq \text{Adom}(H_X)\}.$$

Intuitively, the notation $H \parallel_X$ removes from H_X the facts that can never be observed if H_X were a stand-alone network.

Definition 27. Let $\mathbf{C}$ be a constraint with behavior $\mathbf{C}(\sigma_1)$ to σ_3 . A behavior B from σ_1 to σ_2 is $\mathbf{C}$ -monotone if for distributed instances $H_0, \dots, H_k$ over σ_1 , $K_0, \dots, K_k$ over σ_2 , and U over σ_3 with:

- $\text{netw}(H_j) \cap \text{netw}(H_\ell) = \emptyset$ for all $j \neq \ell \in [0, k]$;
- $(H, U) \in \mathbf{C}(\sigma_1)$ with $H = \bigcup_{i=0}^k H_i$; and
- $(H_i, U \parallel_{\text{netw}(H_i)}) \in \mathbf{C}(\sigma_1)$ for all $i \in [1, k]$

there exists a distributed instance K over σ_2 with $(H, K) \in B$ such that $\bigcup_{i=0}^k K_i \subseteq K$ holds.

The reader may wonder why in Definition 27 it is necessary to consider $k > 1$. To answer this question, we first argue that $k = 1$ is indeed sufficient for many concrete configuration constraints $\mathbf{C}$ considered in this paper. The reason for this observation is that most of the example constraints in this paper have a special property that for some U , $\{(H_1, U \upharpoonright_{\text{netw}(H_1)}), (H_2, U \upharpoonright_{\text{netw}(H_2)})\} \subseteq \mathbf{C}(\sigma)$ with $\text{netw}(H_1) \cap \text{netw}(H_2) = \emptyset$ implies $(H_1 \cup H_2, U) \in \mathbf{C}(\sigma)$. In general, however, this property is not true, as illustrated next.

Example 18. Consider the variant $\mathbf{Top}(\sigma)$ in which every edge of the topology λ has a direction, and we add node a to the Neighbor relation of a node b only if a and b are adjacent in λ with the witnessing edge pointing from b to a . Now consider a distributed instance H over some schema σ with $\text{netw}(H) = [0, k]$ and with $\lambda = \{1 \rightarrow 0, 2 \rightarrow 0, \dots, k \rightarrow 0\}$. Let U be such that $(H, U) \in \mathbf{Top}(\sigma)$ and let H_i and $U \upharpoonright_{H_i}$ be the distributed instance based on H , respectively U , singling out node i and its local instance. We observe that $(H_i, U \upharpoonright_{\text{netw}(H_i)}) \in \mathbf{Top}(\sigma)$ is true for every $i \in [1, k]$ while $(\bigcup_{i \in S} H_i, U \upharpoonright_S) \in \mathbf{Top}(\sigma)$ is clearly false for every set $S \subseteq [1, k]$ with more than two nodes. (Notice that $0 \notin [1, k]$.)

Let $\mathcal{M}_{\mathbf{C}}$ denote the set of all the behaviors that are $\mathbf{C}$ -monotone. Clearly, $\mathcal{M}_{\text{node}} \subseteq \mathcal{M}_{\mathbf{C}}$ for every $\mathbf{C}$.

Theorem 16. For any configuration constraint $\mathbf{C}$,

$$\mathcal{G}(\mathbf{Id} + \mathbf{Ord} + \mathbf{C}) = \mathcal{M}_{\mathbf{C}}.$$

To illustrate the power of this result, we introduce weaker forms of node monotonicity by considering stronger constraints.

Example 19. For an example of a constraint expressing the presence of a distributed index, consider the constraint $\mathbf{Ind}$ with, for every schema σ , $\mathbf{Ind}(\sigma)$ denoting a behavior from σ to $\{\text{Index}_R[k+1] \mid R[k] \in \sigma\} \cup \{\text{Node}[1]\}$ with $\text{acc}(\text{Index}_R) = \text{input}$, and such that $(H, U) \in \mathbf{Ind}(\sigma)$ iff U is a distributed instance such that:

- for every $R[k] \in \sigma$ and $a_1, \dots, a_k \in \text{Adom}(H)$ at least one pair of (not necessarily different) nodes $x, y \in \text{netw}(H)$

with $\text{Index}_R @ x(y, a_1, \dots, a_k) \in U$;

- every $\text{Index}_R @ x(y, \bar{a}) \in U$ and $R(\bar{a}) \in \text{flat}(H)$ implying $R @ y(\bar{a}) \in H$; and
- every $\text{Index}_R @ x(y, \bar{a}) \in U$ implying $\text{Node} @ x(y) \in U$.

Intuitively, **Ind** requires that for every input fact there is a node that stores said input fact. The index entries on each node make this mapping between nodes and facts explicit, and when a node does not store a fact there is also no entry for it in the index. Every node has a relation **Node** revealing all nodes that its index points to. Notice that the latter information is non-redundant because $\text{acc}(\text{Index}_R) = \text{input}$.

As another example, consider the variant **Ind**⁺ of **Ind** that, additionally, requires that for every domain element a there is at least one node with index entries for all facts containing a . Formally, **Ind**⁺(σ) is defined with $(H, U) \in \text{Ind}^+(\sigma)$ iff

- $(H, U) \in \text{Ind}(\sigma)$; and
- there is a function $\alpha : \text{Adom}(H) \rightarrow \mathcal{P}^+(\text{netw}(H))$ such that for every $R[k] \in \sigma$ and every sequence $a_1, \dots, a_k$ of elements from $\text{Adom}(H)$ we have

$$\text{Index}_R @ x(y, a_1, \dots, a_k) \in U$$

iff $x \in \alpha(a_i)$ for some $i \in [1, k]$.

The attentive reader may recognize that **Ind** and **Ind**⁺ show similarities with the policy-aware and domain-guided transducers introduced in previous work [7, 104] which make nodes aware of the facts they are responsible for in the network. Indeed, these constraints provide nodes with precisely the same information, only encoded in a way not restricting the distributed input instances that can be considered.

Now, applying Theorem 16, we obtain the following corollary.

Corollary 7.

1. $\mathcal{G}(\text{Id} + \text{Ord} + \text{Ind}) = \mathcal{M}_{\text{Ind}}$; and
2. $\mathcal{G}(\text{Id} + \text{Ord} + \text{Ind}^+) = \mathcal{M}_{\text{Ind}^+}$.

Proof of Theorem 15

This result is an extension of the proof of Proposition 21. Our only change in the construction is to subdivide P2 into two subphases (a) and (b).

Proposition 22. $\mathcal{M}_{\text{node}} \subseteq \mathcal{G}(\text{Id+Ord})$.

Proof. First, we let $B \in \mathcal{M}_{\text{node}}$ be a behavior from σ_1 to σ_2 , and let Y be a transducer schema where $Y_{\text{in}} = \sigma_1$, $Y_{\text{out}} = \sigma_2$, and $Y_{\text{sys}} = \text{Id+Ord}$. Similar to the proof of Proposition 21, we remark that the definition of Y_{mem} and Y_{msg} can be inferred from the definition of the transducer's mappings we define later.

Transducer Construction The **Id+Ord**-transducer Π over Y is the same as the transducer constructed for the proof of Proposition 21. Indeed, we simply replace P2 by the following phase:

- P2: (a) Each transducer sends its input to the transducer with the lowest identifier in its quorum.
- (b) The transducer with the lowest identifier in the quorum computes B and redistributes the outputs to the appropriate nodes in its quorum according to B . Each node receives this output and writes it to its output instance.

P2 For (a), if this transducer has a $\text{P1Done}(a)$ -fact for every active domain value a , then it sends out $\text{SndR}(y, z, \bar{x})$ -facts via M_{snd} for every input relation $R(\bar{x})$, where z is the identifier of the sending transducer, and where y is the transducer identifier which is minimal for this transducer's quorum according to the system relation $\leq$.

In response, the transducer with identifier y receives these input facts $\text{SndR}(\bar{x})$ and stores them in $\text{CopyR}(z, \bar{x})$ -facts via M_{in} . In (b), the transducer with id y computes B by using all $\text{CopyR}(z, \bar{x})$ -facts as the distributed input instance H . The output distributed instance K is then sent out as $\text{CopyOutR}(z, \bar{x})$ for all output facts $R(\bar{x})$ via M_{snd} . The transducers receive each of these facts and output $R(\bar{x})$ to their respective output instances. We finally add $\text{QuorumAdom}(a)$ and

P1Done(a) to the delete instance via M_{del} . This cycles the transducer back to P1.

Computation & Characterization The argument that for any distributed input instance H over σ_1 , any fair run of Π on H satisfies $(H, \text{do}(\mathcal{R})) \in B$ trivially follows from the argument given in the proof of Proposition 21. Indeed, the only change is that now the node with the lower identifier can make choices for the quorum, allowing more behaviors to become computable.

The fact that for each $(H, K) \in B$ there is a fair run $\mathcal{R}$ of Π on H with $\text{do}(\mathcal{R}) = K$ follows from the insight that transducers with the lowest identifier in each quorum can choose precisely those (H', K') pairs in B which, with successive extensions, lead to $(H, K) \in B$. $\square$

Proposition 23. $\mathcal{G}(\text{Id+Ord}) \subseteq \mathcal{M}_{\text{node}}$.

Proof. Let $B \in \mathcal{G}(\text{Id+Ord})$ be an arbitrary behavior characterized by a transducer $\Pi = (\text{Id+Ord}, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$ over a transducer schema $\mathcal{Y}$. Our goal is to show that $B \in \mathcal{M}_{\text{node}}$, in other words, for any pair H_1, H_2 over $\mathcal{Y}_{\text{in}}$ such that $\text{netw}(H_1) \cap \text{netw}(H_2) = \emptyset$ we show for each $(H_1, K_1) \in B$ and $(H_2, K_2) \in B$ there exists a $(H_1 \cup H_2, K) \in B$ such that $K_1 \cup K_2 \subseteq K$.

For this, let H_1 and H_2 be distributed instances with $\text{netw}(H_1) \cap \text{netw}(H_2) = \emptyset$. Since we assume that B is characterized by Π , there exists at least one run of Π on H_1 for each $(H_1, K_1) \in B$. Let $\mathcal{R}_1$ be a run of Π on H_1 generating an arbitrary such K_1 . Similarly, let $\mathcal{R}_2$ be a run of Π on H_2 generating an arbitrary K_2 with $(H_2, K_2) \in B$. Now consider the distributed instance $H := H_1 \cup H_2$. By the definition of fair runs, there exist runs such that messages between nodes in $\text{netw}(H_1)$ and nodes in $\text{netw}(H_2)$ are delayed for an arbitrary yet finite period of time. During this time, nodes in $\text{netw}(H_1)$ can not distinguish if they are part of $\text{netw}(H)$ or $\text{netw}(H_1)$ (similarly for nodes in $\text{netw}(H_2)$ which cannot distinguish between $\text{netw}(H_2)$ and $\text{netw}(H)$). In particular, there is a run $\mathcal{R}$ of Π on H which start with the prefix of $\mathcal{R}_1$ till and including its quiescence configuration, followed by the prefix of $\mathcal{R}_2$ till and including the quiescence configuration ρ of $\mathcal{R}_2$, with $\text{do}(\rho)$ (in $\mathcal{R}$) having $(H_1 \cup H_1) \subseteq \text{do}(\rho)$. Now extend the run in any fair way. Since Π computes B the run $\mathcal{R}$

will eventually reach a quiescence configuration as well, thus with, $(H_1 \cup H_2, \text{do}(\mathcal{R})) \in B$ and $K_1 \cup K_2 \in \text{do}(\mathcal{R})$ since output facts cannot be retracted. This shows that $B \in \mathcal{M}_{\text{node}}$. $\square$

Proof of Theorem 16

Proposition 24. For any configuration constraint $\mathbf{C}$, $\mathcal{M}_{\mathbf{C}} \subseteq \mathcal{G}(\text{Id}+\text{Ord}+\mathbf{C})$.

Proof. Let $\mathbf{C}$ be a constraint, and let $B \in \mathcal{M}_{\mathbf{C}}$ be a behavior from σ_1 to σ_2 . Let $Y := (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{msg}}, Y_{\text{mem}}, Y_{\text{sys}})$ with $Y_{\text{in}} := \sigma_1$, and $Y_{\text{out}} := \sigma_2$, Y_{sys} consistent with $\mathbf{C}$, and with the definitions of Y_{mem} and Y_{msg} inferred from the definition of the mappings we describe later. We construct a transducer

$$\Pi := (\text{Id}+\text{Ord}+\mathbf{C}, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$$

over Y which is characterized by B , which is based on the transducer construction given in the proof of Proposition 22.

Transducer Construction The definitions of the mappings of Π are effectively identical to the mappings of the transducer given in the proof of Proposition 22, with the only difference being that P2(b) is replaced by something else:

P2: b) The transducer with the lowest identifier in the quorum checks whether all the input relations it received satisfies $\mathbf{C}$. If it does, the transducer computes B and sends the output to the appropriate transducers, who in turn write it to their output channel, ultimately going back to P1. Otherwise, we wait for the quorum to expand by going back to P1.

P2 For (b), the transducer with the lowest identifier according to **Ord** within the quorum uses its $\text{CopyR}(z, \bar{x})$ -facts as a distributed instance H' and verifies whether there exists a $(H', U') \in \mathbf{C}$. If this is the case, it will distribute the distributed instance K' from $(H', K') \in B$ to the correct nodes by sending $\text{CopyOutR}(z, \bar{x})$ -facts via M_{snd} for all output facts $R(\bar{x})$ in K' , where z is the id of the transducer on

which $R(\bar{x})$ resides. Upon receiving these facts, the transducers then output $R(\bar{x})$ to their output instance. If $(H', U') \in \mathbf{C}$ did not exist, we send nothing. After this, we delete $\text{QuorumAdom}(a)$ and $\text{P1Done}(a)$ via M_{del} . This cycles the transducer to P1.

Computation & Characterization It can be verified that Π computes B , the argument is analogous to the argument given in the proof of Proposition 22. The only additions are that Π now waits in case $(H', U') \in \mathbf{C}$ does not exist, but it is otherwise analogous. $\square$

Proposition 25. For any configuration constraint $\mathbf{C}$, $\mathcal{G}(\text{Id+Ord+C}) \subseteq \mathcal{M}_{\mathbf{C}}$.

Proof. Let $\mathbf{C}$ be a constraint, and let $B \in \mathcal{G}(\text{Id+Ord+C})$ be a behavior characterizing an Id+Ord+C -transducer Π over the transducer schema Y . Next, assume distributed instances $H_0, \dots, H_k$ over Y_{in} , $K_0, \dots, K_k$ over Y_{out} , and U over Y_{sys} , with $\text{netw}(H_a) \cap \text{netw}(H_b) = \emptyset$ for $a \neq b \in [0, k]$. Furthermore, assume that $(H_i, U|_{\text{netw}(H_i)}) \in \mathbf{C}(Y_{\text{in}})$, $(H_i, K_i) \in B$ for all $i \in [1, k]$, and that $(\bigcup_{i=0}^k H_i, U) \in \mathbf{C}(Y_{\text{in}})$. Let H be $\bigcup_{i=0}^k H_i$. We show the existence of a fair run $\mathcal{R}$ of Π on H such that $\text{do}(\mathcal{R}) = K$ is a distributed instance over Y_{out} with $(H, K) \in B$ and $\bigcup_{i=0}^k K_i \subseteq K$. First, assume fair runs $\mathcal{R}_i$ of Π on H_i with $i \in [1, k]$, by choice of $\mathcal{R}$ we can always consider $\text{do}(\mathcal{R}_i) = K_i$ and hence $(H_i, \text{do}(\mathcal{R}_i)) \in B$. Next, for the construction of $\mathcal{R}$, consider the run composed of all runs $\mathcal{R}_i$, back-to-back, followed by a section where all the nodes in $\text{netw}(H_0)$ become active (due to run fairness). At this point, we have $(H, U) \in \mathbf{C}(Y_{\text{in}})$ and $\mathcal{R}$ will reach a quiescence point at $\text{do}(\mathcal{R}) = K$ by choice of $\mathcal{R}$. Since the output of runs $\mathcal{R}_i$ are already written to the output instance of each node we have $K_i \subseteq K$, which shows $B \in \mathcal{M}_{\mathbf{C}}$. $\square$

2 Deterministic Computation

Let $\mathcal{D}$ denote the set of all computable deterministic behaviors. That is, $\mathcal{D}$ contains all behaviors B such that B is a deterministic mapping. Furthermore, for a behavior B , let B^* denote the behavior

defined as

$$B^* := \{(H, K) \mid (H, K') \in B \text{ with } K(n) := \text{flat}(K'), n \in \text{netw}(H)\}.$$

(It is easy to see that B^* is indeed a behavior *c.f.*, Definition 20.)

Proposition 26. For every configuration constraint $\mathbf{C}$:

1. $\mathcal{F}(\mathbf{C}) \cap \mathcal{D} = \mathcal{G}(\mathbf{C}) \cap \mathcal{D}$;
2. $\mathcal{F}(\mathbf{Id+All}) \cap \mathcal{D} = \mathcal{D}$;
3. $\mathcal{F}(\mathbf{Id}) \cap \mathcal{D} = \mathcal{M}_{\text{node}} \cap \mathcal{D}$; and
4. $\mathcal{F}(\mathbf{Id+C}) \cap \mathcal{D} = \mathcal{M}_{\mathbf{C}} \cap \mathcal{D}$ for any constraint $\mathbf{C}$;
5. $B \in \mathcal{F}(\mathbf{C})$ iff $B^* \in \mathcal{F}(\mathbf{C})$.

The proof of this proposition is postponed to Section 8.2.

For a query Q from σ_1 to σ_2 we consider the behavior B_Q from σ_1 to σ_2 with $(H, K) \in B_Q$ if and only if $\text{flat}(K) = Q(\text{flat}(H))$ (taking of course all other constraints implied by the definition of behavior into account). In other words, B_Q represents all possible computations of Q in distributed networks ignoring the positioning of input and output facts on specific nodes. Behavior B_Q^* then represents precisely the scenarios where every computation of Q ends with all nodes having the entire output for Q . Particularly notice that $B_Q^* \subseteq B_Q$ and $B_Q^* \in \mathcal{D}$.

Recall from [7] that a query Q from σ_1 to σ_2 is *semi-monotone*¹ if for every pair of instances I_1, I_2 over σ_1 where I_2 does not contain any facts with arity 0, and every fact $\mathbf{f} \in I_2$ satisfying $\text{Adom}(\mathbf{f}) \setminus \text{Adom}(I_1) \neq \emptyset$ it is the case that $Q(I_1) \subseteq Q(I_1 \cup I_2)$. Intuitively, adding facts is not allowed to remove answers, provided that each added fact introduces at least one new value. Furthermore, we call Q *(domain-)disjoint-monotone* if for every two instances I_1, I_2 over σ_1 where I_2 does not contain facts with arity 0, and each fact $\mathbf{f} \in I_2$ satisfying $\text{Adom}(\mathbf{f}) \cap \text{Adom}(I_1) = \emptyset$ we have $Q(I_1) \subseteq Q(I_1 \cup I_2)$. Intuitively, this means that adding a completely separate body of data is not allowed to invalidate answers about the original data.

We have the following relationships between queries and behaviors that compute queries:

¹Also known as (domain-)distinct-monotone [7].

Proposition 27. For every computable query Q :

1. Q is monotone iff $B_Q^* \in \mathcal{M}_{\text{node}}$;
2. Q is semi-monotone iff $B_Q^* \in \mathcal{M}_{\text{Ind}}$;
3. Q is disjoint-monotone iff $B_Q^* \in \mathcal{M}_{\text{Ind}^+}$;

We postpone the proof of this proposition to Section 8.2.

By combining Proposition 26 and 27, we can now easily reproduce many of the results in [7, 8].

Theorem 17. For all computable queries Q , the following holds:

1. Q is monotone iff $B_Q \in \mathcal{F}(\text{Id})$;
2. Q is semi-monotone iff $B_Q \in \mathcal{F}(\text{Id}+\text{Ind})$;
3. Q is disjoint-monotone iff $B_Q \in \mathcal{F}(\text{Id}+\text{Ind}^+)$; and
4. Q is computable iff $B_Q \in \mathcal{F}(\text{Id}+\text{All})$.

Proof of Proposition 26

Proposition 28. $\mathcal{F}(\mathbf{C}) \cap \mathcal{D} = \mathcal{G}(\mathbf{C}) \cap \mathcal{D}$.

Proof. $\mathcal{G}(\mathbf{C}) \cap \mathcal{D} \subseteq \mathcal{F}(\mathbf{C}) \cap \mathcal{D}$ is trivial. Further, for every behavior $B \in \mathcal{F}(\mathbf{C})$ from σ_1 to σ_2 there is a $\mathbf{C}$ -transducer Π that computes it. The latter means that any fair run $\mathcal{R}$ of Π over H will result in a distributed instance $K = \text{do}(\mathcal{R})$ over σ_2 with $(H, K) \in B$. Since $B \in \mathcal{D}$ implies that K is also the only distributed instance that is paired with H in B , it is immediate that such a run exists for *every* such K . By generality of the argument, it now follows immediately that B indeed characterizes Π , hence $B \in \mathcal{G}(\mathbf{C})$. $\square$

Proposition 29. $\mathcal{F}(\text{Id}+\text{All}) \cap \mathcal{D} = \mathcal{D}$

Proof. We remark that $\mathcal{F}(\text{Id}+\text{All}) \cap \mathcal{D} \subseteq \mathcal{D}$ is trivial.

Conversely, let $B \in \mathcal{D}$ be a behavior from schema σ_1 to schema σ_2 .

Transducer Construction Let $Y = (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{mem}}, Y_{\text{msg}}, Y_{\text{sys}})$ transducer schema, with

$$\begin{aligned}
Y_{\text{in}} &:= \sigma_1; \\
Y_{\text{out}} &:= \sigma_2; \\
Y_{\text{sys}} &:= \{\text{Id}[1], \text{All}[1]\}; \\
Y_{\text{mem}} &:= \{\text{Done}[1]\} \cup \{\text{ExtR}[k+1] \mid R[k] \in Y_{\text{in}}\} \\
&\quad \cup \{\text{AckR}[k+1] \mid R[k] \in Y_{\text{in}}\}, \text{ and} \\
Y_{\text{msg}} &:= \{\text{DoneSnd}[1]\} \cup \{\text{SndR}[k+2] \mid R[k] \in Y_{\text{in}}\} \\
&\quad \cup \{\text{SndAckR}[k+2] \mid R[k] \in Y_{\text{in}}\}.
\end{aligned}$$

We will construct a transducer $\Pi = (\mathbf{Id} + \mathbf{All}, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$ such that Π computes B .

Let M_{snd} be such that $(I, J) \in M_{\text{snd}}$ if, and only if, J is the smallest instance with the properties:

1. For every $R[k] \in Y_{\text{in}}$,
 - (a) $R(\bar{a}), \text{All}(y) \in I$ implies $\text{SndR}(y, x, \bar{a}) \in J$;
 - (b) $\text{Id}(x), \text{ExtR}(y, \bar{a}) \in I$ implies $\text{SndAckR}(y, x, \bar{a}) \in J$;
 - (c) if for all $\text{All}(y) \in I$ and $R(\bar{a}) \in I$ (with $R[k] \in Y_{\text{in}}$) there exists a $\text{AckR}(y, \bar{a}) \in I$ then $\text{DoneSnd}(x) \in J$.

Intuitively, the transducer will thus broadcast its entire local instance, send out acknowledgments for facts that it has received, and announce it is done if it has received an acknowledgment back from all nodes for all its own facts.

Let M_{in} be such that $(I, J) \in M_{\text{in}}$ if, and only if, J is a minimal instance with the following properties:

2. (a) $\text{SndR}(x, y, \bar{a}) \in I$ implies $\text{ExtR}(y, \bar{a}) \in J$;
- (b) $\text{SndAckR}(x, y, \bar{a}) \in I$ implies $\text{AckR}(y, \bar{a}) \in J$;
- (c) $\text{DoneSnd}(y) \in I$ implies $\text{Done}(y) \in J$.

In other words, the transducer stores all facts it receives in its memory relations.

Let M_{out} be such that $(I, J) \in M_{\text{out}}$ with J a minimal instance adhering to:

3. If $\text{Done}(y) \in I$ for every $\text{All}(y) \in I$, Let $\text{Id}(x) \in I$ and K be such that $(\{\text{R}@y(\bar{a}) \mid \text{ExtR}(y, \bar{a}) \in I, \text{R}[k] \in Y_{\text{in}}\}, K) \in B$:
 - (a) $\text{S}@x(\bar{a}) \in K$ implies $\text{S}(\bar{a}) \in J$ for every $\text{S}[p] \in Y_{\text{out}}$.

In other words, if the node has received all local instances from all nodes in the network, then it computes the intended behavior locally and forwards the output facts relevant for itself to its local output relations.

Lastly, we define M_{del} as the empty mapping, that is $(I, \emptyset) \in M_{\text{del}}$ for every possible I . This concludes the definition of Π .

Transducer Correctness We verify that Π computes B . First, we assume the arbitrary distributed instance H over Y_{in} , and show that there exists a run $\mathcal{R}$ of Π on H with $(H, \text{do}(\mathcal{R})) \in B$. We construct a run $\mathcal{R}$ of Π on H that is organized in rounds in which every node becomes active precisely once. In the first round, $\mathcal{R}$ broadcasts its full input instance to every other node in the network (*c.f.*, 1a). In the next round, we let each node consume all messages from its message instance at once, storing all external input in ExtR relations in the node's local memory instance (*c.f.*, 2a). The following round will make each node send acknowledgments for the received input back to each sender (*c.f.*, 1b), followed by another round where each node reads the full message instance and records the acknowledgments as AckR relations in its memory instance (*c.f.*, 2b). Next, we have a round where each node generates a DoneSnd -fact for every node as it has received an acknowledgement from every node for all facts in its own local instance (*c.f.*, 1c). This message confirms that the other node has indeed received the full instance.

Moving to the next round, each node witnesses a SndDone -fact from every node and stores it in its local memory (*c.f.*, 2c). Then, in the next round, every node will observe the Done -fact for every node, indicating that the node has a copy of the entire distributed instance in its local memory, thus allowing to compute B and writing the fragment of the output corresponding to its own node to its output instance (*c.f.*, 3a). Since Π is defined in terms of B and has access to all of H , it must be that $(H, \text{do}(\mathcal{R})) \in B$. Note that $\mathcal{R}$ is fair, since it is

organized in a finite number of rounds where each node becomes active precisely once.

It is straightforward to verify that each run of $\mathcal{R}$ of Π on H reaches a quiescence point and satisfies $(H, \text{do}(\mathcal{R})) \in B$ since nodes store all received messages in their memory and eventually every node receives the entire instance, all acknowledgements, and DoneSnd-facts due to the fairness condition. Then, since B is only computed after every node indeed has the complete distributed instance in its memory relation, and since B is deterministic, every node will compute the same K and therefore making the network eventually reflect the distributed output instance K . This observation concludes the proof. $\square$

Proposition 30. $\mathcal{M}_C \cap \mathcal{D} \subseteq \mathcal{F}(\text{Id}+C) \cap \mathcal{D}$ for every C .

Proof. We assume the construction of Π from the proof of Proposition 24 with a few minor differences:

- each behavior B additionally satisfies $B \in \mathcal{D}$, and the arbitrary constraint used in Π satisfies $C(Y_{\text{in}}) \in \mathcal{D}$;
- Π no longer has access to $\leq[2]$, and the definition of Y_{sys} is adapted accordingly, hence we change M_{snd} (the only mapping using $\leq[2]$) to send input data to every node instead of just those smaller than itself, making each node compute B instead of only the smallest node in the quorum.

To show that Π is characterized by B we can use largely use the proof for Proposition 24, given the above changes. $\square$

Proposition 31. $\mathcal{F}(\text{Id}+C) \cap \mathcal{D} \subseteq \mathcal{M}_C \cap \mathcal{D}$ for every C .

Proof. The proposition follows from:

- Proposition 13, implying $\mathcal{F}(\text{Id}+C) \subseteq \mathcal{F}(\text{Id}+\text{Ord}+C)$,
- Proposition 26, implying $\mathcal{F}(\text{Id}+\text{Ord}+C) \cap \mathcal{D} = \mathcal{G}(\text{Id}+\text{Ord}+C) \cap \mathcal{D}$, and
- Theorem 16, implying $\mathcal{G}(\text{Id}+\text{Ord}+C) = \mathcal{M}_C$.

$\square$

Proposition 32. $\mathcal{F}(\text{Id}) \cap \mathcal{D} = \mathcal{M}_{\text{node}} \cap \mathcal{D}$

Proof. We first observe that $\mathcal{M}_{\text{node}} \cap \mathcal{D} = \mathcal{M}_{\mathbf{C}} \cap \mathcal{D}$ where $\mathbf{C}$ is the empty constraint; that is, with $\mathbf{C}(\sigma)$ the behavior from σ to $\emptyset$ where $(H, \emptyset) \in \mathbf{C}(\sigma)$ for each H over σ . The result of the proposition now follows directly from Proposition 30 and Proposition 31. $\square$

Proposition 33. $B \in \mathcal{F}(\mathbf{C})$ if, and only if $B^* \in \mathcal{F}(\mathbf{C})$.

Proof. For the if direction, we recall that $B^* \subseteq B$ and thus $B^* \in \mathcal{F}(\mathbf{C})$ implies $B \in \mathcal{F}(\mathbf{C})$ due to Proposition 13(3).

For the only-if direction, we recall that $B \in \mathcal{F}(\mathbf{C})$ implies existence of a $\mathbf{C}$ -transducer Π that computes B . Now let us extend the send-mapping of Π by simulating (in addition to its current definition) the output-mapping and letting every node broadcast its output facts via dedicated message relations to all other nodes in the network. Furthermore, let us extend the output-mapping of Π to (in addition to its current definition) store all output facts received via messages into the local output relations of the receiving node. Then it is easy to see that this extended transducer computes precisely B^* and, by construction, that B^* is indeed in $\mathcal{F}(\mathbf{C})$. $\square$

Proof of Proposition 27

Proposition 34. Q is monotone if $B_Q^* \in \mathcal{M}_{\text{node}}$.

Proof. Let Q be an arbitrary query from σ_1 to σ_2 , with behavior B_Q from σ_1 to σ_2 such that $(H, K) \in B_Q$ if, and only if $\text{flat}(K) = Q(\text{flat}(H))$, and with behavior $B_Q^* = \{(H, K) \mid (H, K') \in B \text{ with } K(n) = \text{flat}(K')\}$ from σ_1 to σ_2 and $B_Q^* \in \mathcal{M}_{\text{node}}$. We show that $Q(I_1) \subseteq Q(I_1 \cup I_2)$ for every pair of instances I_1, I_2 over σ_1 . Assume two distributed instances H_1, H_2 over σ_1 with $\text{netw}(H_1) \cap \text{netw}(H_2) = \emptyset$ and assume $(H_1, K_1) \in B_Q^*$ and $(H_2, K_2) \in B_Q^*$. By node monotonicity and determinism of B_Q^* there exists exactly one $(H_1 \cup H_2, K) \in B_Q^*$, and K satisfies $K_1 \cup K_2 \subseteq K$. Now, assume $I_1 = \text{flat}(H_1)$ and $I_2 = \text{flat}(H_2)$, then by definition of B_Q^* we have that $Q(I_1) = \text{flat}(K_1)$ and $Q(I_2) = \text{flat}(K_2)$,

and $Q(I_1 \cup I_2) = \text{flat}(K)$. And since $\text{flat}(K_1) \cup \text{flat}(K_2) \subseteq \text{flat}(K)$, this shows that $Q(I_1) \subseteq Q(I_1 \cup I_2)$, hence Q is monotone. $\square$

Proposition 35. $B_Q^* \in \mathcal{M}_{\text{node}}$ if Q is monotone.

Proof. Let Q be an arbitrary monotone query from σ_1 to σ_2 . And, let B_Q be a behavior from σ_1 to σ_2 such that $(H, K) \in B_Q$ if, and only if $\text{flat}(K) = Q(\text{flat}(H))$, and with behavior $B_Q^* = \{(H, K) \mid (H, K') \in B \text{ with } K(n) = \text{flat}(K')\}$ from σ_1 to σ_2 . We show that $B_Q^* \in \mathcal{M}_{\text{node}}$. Assume instances I_1, I_2 over Y_{in} , due to monotonicity of Q we have $Q(I_1) \subseteq Q(I_1 \cup I_2)$ and $Q(I_2) \subseteq Q(I_1 \cup I_2)$. Then, assuming disjoint networks $\mathcal{N}_1$ and $\mathcal{N}_2$ we define H_1 as $\{R@x(\bar{a}) \mid R(\bar{a}) \in I_1 \text{ for each } R[k] \in \sigma_1, \text{ and for all } x \in \mathcal{N}_1\}$ and H_2 as $\{R@x(\bar{a}) \mid R(\bar{a}) \in I_2 \text{ for each } R[k] \in \sigma_1, \text{ and for all } x \in \mathcal{N}_2\}$. Note that $\text{netw}(H_1) \cap \text{netw}(H_2) = \emptyset$ by definition. Next, notice that $(H_1, K_1), (H_2, K_2) \in B_Q^*$ and $(H_1 \cup H_2, K) \in B_Q^*$ by definition of B_Q^* and that $K_1 \cup K_2 \subseteq K$ by the monotonicity of Q . Hence, $B_Q^* \in \mathcal{M}_{\text{node}}$. $\square$

Proposition 36. Q is semi-monotone if $B_Q^* \in \mathcal{M}_{\text{Ind}}$.

Proof. Let Q be a query as in the proposition, from some σ_1 to a schema σ_2 . Let I_1 and I_2 be arbitrary instances over σ_1 with property that I_2 contains precisely the same nullary facts as I_1 and for every fact $R(\bar{a}) \in I_2$ it is the case that at least one of the values, say a , in $\bar{a}$ does not occur in $\text{Adom}(I_1)$.

Now we construct a distributed instance $H_1 := \{R@x(\bar{a}) \mid R(\bar{a}) \in I_1\}$ by placing every fact in I_1 on precisely one node x . Let $H_0 := \{R@y(\bar{a}) \mid R(\bar{a}) \in I_2\}$ be the distributed instance placing all facts of I_2 on a node $y \neq x$.

Now consider the distributed instances

$$U_1 = \{\text{Index}_R@x(x, \bar{a}) \mid R(\bar{a}) \in I_1\} \cup \{\text{Node}@x(x)\}, \text{ and} \\ U_2 = \{\text{Index}_R@y(y, \bar{a}) \mid R(\bar{a}) \in I_2\} \cup \{\text{Node}@y(y)\}.$$

Here, (H_1, U_1) and $(H_1 \cup H_0, U_1 \cup H_0)$ are in $\text{Ind}(\sigma_1)$ thus $B_Q^* \in \mathcal{M}_{\text{Ind}}$ implies $(H_1, K_1) \in B_Q^*$ and $(H_1 \cup H_0, K) \in B_Q^*$ with $K_1 \subseteq K$. The latter implies $Q(\text{flat}(H_1)) = \text{flat}(K_1) \subseteq \text{flat}(K) = Q(\text{flat}(H_1 \cup H_2))$, thus concluding the proof. $\square$

Proposition 37. $B_Q^* \in \mathcal{M}_{\text{Ind}}$ if Q is semi-monotone.

Proof. Let Q be an arbitrary semi-monotone query from σ_1 to σ_2 . We show $B_Q^* \in \mathcal{M}_{\text{Ind}}$. For this, let $H_0, H_1, \dots, H_k$ be distributed instances over σ_1 , let $K_0, K_1, \dots, K_k$ be distributed instances over σ_2 and let U be a distributed system instance as per the definition of $\mathcal{M}_{\text{Ind}}$, furthermore let $H = H_0 \cup H_1 \cup \dots \cup H_k$.

It now suffices to show that $K_i \subseteq K$ for every $i \in [1, k]$. To see this, take an arbitrary K_i . Now observe that for any fact $R(\bar{a}) \notin \text{flat}(H_i)$ with $\bar{a} \subseteq \text{Adom}(H_i)$ it must be that there are nodes $x, y \in \text{netw}(H_i)$ with $\text{Index}_R @ x(y, \bar{a}) \in H_i$ and $R @ y(\bar{a}) \notin H_i$. Therefore, due to disjointness of the network, it follows that $\text{Index}_R @ x(y, \bar{a}) \in H$ and $R @ y(\bar{a}) \notin H$. It then follows from $(H, U) \in \text{Ind}(\sigma_1)$ that $R(\bar{a}) \notin \text{flat}(H)$.

From this, the definition of semi-monotonicity applies to H_i and H , thus with $Q(H_i) \subseteq Q(H)$. It now follows that

$$K_i = \{R @ x(\bar{a}) \mid R(\bar{a}) \in Q(H_i)\} \subseteq \{R @ x(\bar{a}) \mid R(\bar{a}) \in Q(H)\} = K,$$

which concludes the proof. $\square$

Proposition 38. Q is disjoint-monotone if $B_Q^* \in \mathcal{M}_{\text{Ind}^+}$.

Proof. Let Q be an arbitrary query from σ_1 to σ_2 . Let I_1 and I_2 be two instances over σ_1 with $\text{Adom}(I_1) \cap \text{Adom}(I_2) = \emptyset$ and with I_2 having precisely the same nullary facts as I_1 .

Now consider the same distributed instance H_1, H_0, U_1 and U_0 as in the proof of Proposition 36. We note that, indeed (H_1, U_1) and $(H_1 \cup H_0, U_1 \cup U_0)$ are in $\text{Ind}^+(\sigma_1)$, and thus it follows from $B_Q^* \in \mathcal{M}_{\text{Ind}^+}$ that for $(H_1, K_1) \in B_Q^*$ and $(H_1 \cup H_0, K) \in B_Q^*$ that $K_1 \subseteq K$, which means $\text{flat}(K_1) \subseteq \text{flat}(K)$, as desired. $\square$

Proposition 39. $B_Q^* \in \mathcal{M}_{\text{Ind}^+}$ if Q is disjoint-monotone.

Proof. The proof is analogous to the proof of Proposition 37, except that an extra condition applies making nodes responsible for specific values. That is, in the network $\text{netw}(H_i)$ a function α applies

that for every value a in $\text{Adom}(H_i)$ makes at least one node $x \in \text{netw}(H_i)$ responsible for a implying $\text{Index}_R @ x(a, \dots, a) \in U \upharpoonright_{\text{netw}(H_i)}$. Now suppose towards a contradiction that $\text{flat}(H)$ contains a fact $R(\bar{a})$ not in H_i but using a value a from $\text{Adom}(H_i)$. Then, $(H, U) \notin \text{Ind}^+(\sigma_1)$, because $\text{Index}_R @ x(a, \dots, a) \in U \upharpoonright_{\text{netw}(H_i)}$ implies $x \in \alpha(a)$ for U and thus that $\text{Index}_R @ x(\bar{a}) \in U$ and therefore that $R @ x(\bar{a}) \in H_i$ which is the desired contradiction.

Since we have argued that the definition of disjoint-monotonicity applies on $\text{flat}(H_i)$ and $\text{flat}(H)$, it follows from the definition of B_Q^* that

$$\begin{aligned} K_i &= \{R @ x(\bar{a}) \mid R(\bar{a}) \in Q(\text{flat}(H_i)), x \in \text{netw}(K_i)\} \\ &\subseteq \{R @ x(\bar{a}) \mid R(\bar{a}) \in Q(\text{flat}(H)), x \in \text{netw}(K)\} \\ &= K. \end{aligned}$$

This concludes the proof. □

3 Polynomial Time Computation

In their CACM article on the CALM theorem, Hellerstein and Alvaro [44] questioned if it is possible that all polynomial-time queries over ordered databases can be computed in a coordination-free manner. That is, do the conditions of the Immerman-Vardi theorem [48, 93] align with the conditions of the CALM theorem? We investigate this question by making each node powerful enough to utilize a total order over its active domain elements. To this end, we extend the order constraint **Ord** (defined in Section 7.2) to also install on each node the minimum and maximum element of the active domain.

Let Ord^+ be the configuration constraint from σ to behaviors on $\{\leq[2], \text{Min}[1], \text{Max}[1]\}$. Each relation is labeled with $\text{acc}(\leq) = \text{input}$, $\text{acc}(\text{Min}) = \text{input}$ and $\text{acc}(\text{Max}) = \text{input}$, and with $(H, U) \in \text{Ord}^+(\sigma)$ if there is an order $\leq$ over $\text{Adom}(H)$, and with

- $\leq(a, b) \in U$ iff $a \leq b$ (for $a, b \in \text{Adom}(H)$);
- $\text{Min}(a) \in U$ iff a is the smallest according to $\leq$ in $\text{Adom}(H)$; and
- $\text{Max}(a) \in U$ iff a is the largest according to $\leq$ in $\text{Adom}(H)$.

Let PTIME be the class of all polynomial time computable queries.

Theorem 18. $Q \in \text{PTIME}$ exists with $B_Q \notin \mathcal{F}(\text{Id} + \text{Ord}^+ + \text{Ind})$.

Proof. Let Q from $\sigma_1 := \{E[2]\}$ to $\sigma_2 := \{O[0]\}$ with $O() \in Q(I)$ iff there exists $x, y \in \text{Adom}(I)$ with $E(x, y) \in I$ and

$$E(x, v), E(v, x), E(y, w), E(w, y) \notin I$$

for all $v, w \in \text{Adom}(I) \setminus \{x, y\}$. In other words, Q is a boolean query that yields true whenever there is a disconnected edge in the directed graph specified by the relation $E[2]$. Notice that Q is computable in polynomial time using depth-first search.

We show that Q is not computable by a $\text{Id} + \text{Ord}^+ + \text{Ind}$ -transducer by showing that its query-computing behavior B_Q is not $\text{Ord}^+ + \text{Ind}$ -monotone. First, let B_Q be a behavior from σ_1 to σ_2 with $(H, K) \in B_Q$ iff $\text{flat}(K) = Q(\text{flat}(H))$. Observe that

$$\mathcal{G}(\text{Id} + \text{Ord}^+ + \text{Ind}) \subseteq \mathcal{G}(\text{Id} + \text{Ord} + \text{Ord}^+ + \text{Ind})$$

by Proposition 13 (assuming primed versions for relations in the system schema to not have collisions between relations for Ord and Ord^+). We now use this to show that B_Q is not $\text{Ord}^+ + \text{Ind}$ -monotone, and hence not computable with a $\text{Id} + \text{Ord}^+ + \text{Ind}$ -transducer according to Theorem 16.

Consider the following definitions:

- Let $\sigma_3 = \{\leq[2], \text{Min}[1], \text{Max}[1]\}$ unioned with

$$\{\text{Index}_R[k+1], \text{Node}[1] \mid \text{for every } R[k] \in \sigma_1\},$$

such that:

- $\text{acc}(\leq) = \text{input}$,
- $\text{acc}(\text{Min}) = \text{input}$,
- $\text{acc}(\text{Max}) = \text{input}$, and
- $\text{acc}(\text{Index}_R) = \text{input}$ for each $R[k] \in \sigma_1$;
- Next, let $H_0, H_1, \dots, H_k$ be distributed instances over σ_1 such that $\text{netw}(H_a) \cap \text{netw}(H_b) = \emptyset$ for all $a \neq b \in [0, k]$;

- let U be a distributed instance over σ_3 such that $(\bigcup_{i=0}^k H_i, U) \in \mathbf{Ord}^+ + \mathbf{Ind}(\sigma_1)$, and $(H_i, U|_{\text{netw}(H_i)}) \in \mathbf{Ord}^+ + \mathbf{Ind}(\sigma_1)$; and
- let $K_0, K_1, \dots, K_k$ be distributed instances over σ_2 such that we have $(H_i, K_i) \in B_Q$ for all $i \in [1, k]$.

We show for all distributed instance K over σ_2 with $(\bigcup_{i=0}^k H_i, K) \in B_Q$ that $\bigcup_{i=0}^k K_i \not\subseteq K$ (i.e., B_Q is not $\mathbf{Ord}^+ + \mathbf{Ind}$ -monotone). Since B_Q is a deterministic behavior there is exactly one K w. $(\bigcup_{i=0}^k H_i, K) \in B_Q$, hence it is sufficient to show $\bigcup_{i=0}^k K_i \not\subseteq K$ for this unique K .

Without loss of generality, let each $\text{flat}(H_i)$ with $i \in [1, k]$ contain a disconnected edge according to the definition of Q (i.e., $\mathbf{O}\mathbf{O} \in \text{flat}(K_i)$ for each $i \in [1, k]$). Furthermore, let $\text{flat}(H_0)$ contain graph edges connecting all disconnected edges in every $\text{flat}(H_i)$ together, such that $\text{flat}(\bigcup_{i=0}^k H_i)$ contains a fully connected graph. Hence, we have that $\mathbf{O}\mathbf{O} \notin K$ but $\mathbf{O}\mathbf{O} \in \text{flat}(K_i)$ for each $i \in [1, k]$, showing that $\bigcup_{i=0}^k K_i \not\subseteq K$. This shows $B_Q \notin \mathcal{M}_{\mathbf{Ord}^+ + \mathbf{Ind}}$, concluding the proof. $\square$

The reader may wonder if the previous result remains true if the order relation in $\mathbf{Ord}^+$ is replaced by a successor relation. For active domain values a and b if $a \leq b$, then there may still be an active domain value c such that $a \leq c \leq b$. If c is present on some other transducer, we may falsely conclude that there are no elements in between a and b . We tackle this next by explicitly tracking which active domain values directly follow each other.

The Successor and Node Constraints

We define the *successor constraint* $\mathbf{Ord}^s$ as a configuration constraint mapping schemas σ on behaviors to $\{\text{Min}[1], \text{Max}[1], \text{Suc}[2]\}$ with $\text{acc}(\text{Min}) = \text{input}$, $\text{acc}(\text{Max}) = \text{input}$, $\text{acc}(\text{Suc}) = \text{input}$, and $(H, U) \in \mathbf{Ord}^s(\sigma)$ if there is a strict order $<$ over $\text{Adom}(H)$ and

- $\text{Min}(a) \in U$ iff a is the smallest according to $<$ in $\text{Adom}(H)$;
- $\text{Max}(a) \in U$ iff a is the largest according to $<$ in $\text{Adom}(H)$;
- $\text{Suc}(a, b) \in U$ iff $a < b$ for $a, b \in \text{Adom}(H)$ and there exists no $c \in \text{Adom}(H)$ such that $a < c < b$.

Hence, each Ord^s -transducer can test if two elements are direct successors of one another, for all active domain elements. That is,

$$\text{Min}(a), \text{Succ}(a, b), \text{Succ}(b, c), \dots, \text{Succ}(y, z), \text{Max}(z)$$

for all active domain elements $a, b, c, \dots, y, z \in \text{Adom}(H)$ given some input-instance H .

Even though we can now enumerate the active domain, we cannot tell the difference between an active domain element that is a node identifier, and an active domain element that is not. The node constraint, formally defined below, allows us to test if an active domain value is a network identifier. In practical systems, this could simply be implemented as a test which checks whether a data value is a valid IP address, or a valid network identifier. This constraint is different to *All* in that it does not allow to list the nodes of the network, only to test if (an already known) domain value is a node identifier or not.

We define a *node constraint* Nd as a configuration constraint mapping schemas σ on behaviors to $\{\text{Node}[1]\}$ with label $\text{acc}(\text{Node}) = \text{input}$, and with $(H, U) \in \text{Nd}(\sigma)$ if $U = \{\text{Node}@x(y) \mid x \in \text{netw}(H)\}$.

This now allows us to show that these transducers can compute all behaviors, which is shown by the following proposition.

Proposition 40. $B = \mathcal{F}(\text{Id} + \text{Nd} + \text{Ord}^s)$.

Proof. Let σ_1 and σ_2 be arbitrary schemas, and let B be any computable behavior from σ_1 to σ_2 . We show that $B \in \mathcal{F}(\text{Id} + \text{Nd} + \text{Ord}^s)$ by constructing a $\text{Id} + \text{Nd} + \text{Ord}^s$ -transducer.

Consider some arbitrary transducer with identifier x in $\text{netw}(H)$, on a high-level it will proceed as follows:

1. First, x broadcasts its input $H(x)$ to all other transducers in the network using the message $\text{SndR}(\bar{a})$, for all relations $R \in Y_{\text{in}}$. Incoming facts are placed into $\text{ExtR}(\bar{a})$, notice that the local instance will also be part of $\text{ExtR}(\bar{a})$ since x will send messages to itself.
2. At some point, x will be able to verify that it has discovered the entire active domain by enumerating from $\text{Min}(a)$ to $\text{Max}(z)$

using $\text{Suc}[2]$. This means the node can construct the **All**-relation by using the **Node**-relation given by the **Nd** constraint. We also pick the lowest node identifier to be the leader.

3. At this point we apply the same construction as the proof for Theorem 13, that is, the proof that $\mathcal{B} = \mathcal{F}(\mathbf{Id}+\mathbf{All}+\mathbf{L})$.

Transducer Construction First, we define a transducer schema

$$Y = (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{msg}}, Y_{\text{mem}}, Y_{\text{sys}})$$

where $Y_{\text{in}} := \sigma_1$, $Y_{\text{out}} := \sigma_2$, and

$$Y_{\text{msg}} := \{\text{SndR}[k] \in Y_{\text{in}}\}$$

$$Y_{\text{mem}} := \{\text{ExtR}[k], \text{All}[2], \text{Leader}[1], \text{AllComplete}[0] \mid R[k] \in Y_{\text{in}}\}$$

$$Y_{\text{sys}} \text{ defined as per } \mathbf{Id}+\mathbf{Nd}+\mathbf{Ord}^s$$

We construct a transducer $\Pi = (\mathbf{Id}+\mathbf{Nd}+\mathbf{Ord}^s, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$ over Y as follows. First, we define M_{snd} with $(I, J) \in M_{\text{snd}}$ if, and only if J is a minimal instance satisfying the following properties:

1. $\text{SndR}(\bar{a}) \in J$ with $R(\bar{a}) \in I$ for all $R[k] \in Y_{\text{in}}$.
2. If $\text{AllComplete}() \in I$, simulate the construction of M_{snd} given in the proof of Theorem 13.

Next, we define M_{in} with $(I, J) \in M_{\text{in}}$ if, and only if J is a minimal instance satisfying the following properties:

3. $\text{SndR}(\bar{a}) \in I$ implies $\text{ExtR}(\bar{a}) \in J$.
4. If there exists a chain

$$\text{Min}(a), \text{Suc}(a, b), \text{Suc}(b, c), \dots, \text{Suc}(y, z), \text{Max}(z)$$

for $a, b, c, \dots, y, z \in \text{Adom}(I)$ without gaps (*i.e.*, the active domain is complete), then for all active domain values $a \in \text{Adom}(I)$ test if they are nodes with the **Node**-relation and store this as **All**(a) fact in J if this is the case. Then, pick the smallest node identifier encountered in this way and store this as a **Leader**-fact in J . Lastly, add the fact **AllComplete**() in J .

5. If $\text{AllComplete}() \in I$, simulate the construction of M_{in} given in the proof of Theorem 13.

And lastly, for M_{del} and M_{out} we take the definitions used in the construction of Theorem 13 as soon as $\text{AllComplete}() \in I$.

Transducer Correctness Let H be an arbitrary, distributed instance over Y_{in} . First, we show there exists a run $\mathcal{R}$ of Π on H such that $(H, \text{do}(\mathcal{R})) \in B$. Indeed, consider $\mathcal{R}$ where all transducers are immediately active, send and receive their entire input instance with (1) and (3). Then, since each node has the entire active domain, the construction rule (4) is triggered and each node gets **All**, **Leader** and **AllComplete**. Then, the construction rules (2) and (5) are triggered and the run construction of Theorem 13 can be followed.

Note that for each fair run $\mathcal{R}$ of Π on H , it is the case that $(H, \text{do}(\mathcal{R})) \in B$. This amounts to arguing that if arbitrary transducers start to get delayed or hindered in some way, the output is still a valid run output. Indeed, this is the case due to (4), since a transducer can only continue if its active domain is complete. $\square$

4 A Coordination Hierarchy

Figure 8.1 summarizes the relationships between the most important classes of computable behaviors that we considered in this paper. The upcoming propositions formalize the remaining inclusions and non-inclusions.

All containment relationships follow directly from Propositions 13, 12 and 18, Theorems 13 and 14, and the next insights:

Proposition 41. For every schema σ :

1. $\overline{\text{Ord}}(\sigma) \in \mathcal{F}(\text{Id} + \text{Ord}^+)$;
2. $\overline{\text{Ind}}(\sigma) \in \mathcal{F}(\text{Id} + \text{Ind}^+)$;
3. $\text{Ord}(\sigma) \in \mathcal{F}(\text{Id} + \text{L})$;
4. $\text{Ind}^+(\sigma) \in \mathcal{F}_T(\text{Id} + \text{All})$.

For the non-containment and strictness of containment relationships we have the following proposition:

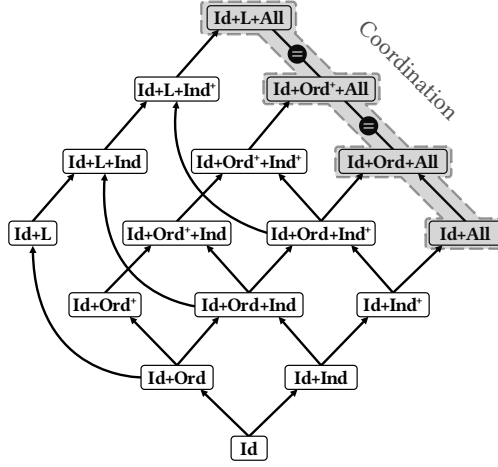


Figure 8.1 – Classes are connected through a path iff there is a containment relationship. All arrows indicate strict containment and point towards the containing class.

Proposition 42.

- $\mathcal{F}(\mathbf{Id}+\mathbf{L}) \setminus \mathcal{F}(\mathbf{Id}+\mathbf{Ind}+\mathbf{Ord}^+) \neq \emptyset$;
- $\mathcal{F}(\mathbf{Id}+\mathbf{Ord}^+) \setminus \mathcal{F}(\mathbf{Id}+\mathbf{L}+\mathbf{Ind}) \neq \emptyset$;
- $\mathcal{F}(\mathbf{Id}+\mathbf{Ord}) \setminus \mathcal{F}(\mathbf{Id}+\mathbf{All})$;
- $\mathcal{F}(\mathbf{Id}+\mathbf{Ind}) \setminus [\mathcal{F}(\mathbf{Id}+\mathbf{L}) \cup \mathcal{F}(\mathbf{Id}+\mathbf{Ord}^+)] \neq \emptyset$; and
- $\mathcal{F}(\mathbf{Id}+\mathbf{Ind}^+) \setminus [\mathcal{F}(\mathbf{Id}+\mathbf{Ind}+\mathbf{L}) \cup \mathcal{F}(\mathbf{Id}+\mathbf{Ind}+\mathbf{Ord}^+)] \neq \emptyset$.

Furthermore, we observe that although $\mathcal{F}(\mathbf{Id}+\mathbf{Ord}^+)$ and $\mathcal{F}(\mathbf{Id}+\mathbf{L})$ are incomparable they collide if we ignore single-node networks, as made formal by the following proposition:

Proposition 43. For every behavior $B \in \mathcal{F}(\mathbf{Id}+\mathbf{Ord}^+)$ there is a behavior $B' \in \mathcal{F}(\mathbf{Id}+\mathbf{L})$ such that for every distributed instance H with $|\text{Adom}(H)| \geq 2$: $(H, K) \in B$ iff $(H, K) \in B'$.

Proof. Consider the variant **C** of **Ord**⁺ that leaves the order, max and min relations empty if there is only one active domain element. To prove the proposition, it suffices to prove that $\mathcal{F}(\mathbf{Id}+\mathbf{L}) = \mathcal{F}(\mathbf{Id}+\mathbf{C})$, and thus, due to Proposition 18, that $\overline{\mathbf{C}}(\sigma) \in \mathcal{F}(\mathbf{Id}+\mathbf{L})$.

We construct a transducer $\Pi = (\mathbf{Id}+\mathbf{L}, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$ over the transducer schema $\mathbf{Y} = (\mathbf{Y}_{\text{in}}, \mathbf{Y}_{\text{out}}, \mathbf{Y}_{\text{mem}}, \mathbf{Y}_{\text{msg}}, \mathbf{Y}_{\text{sys}})$ such that Π com-

puts $\overline{\mathbf{C}}(\sigma)$. We define $Y_{\text{in}} := \sigma$,

$$Y_{\text{out}} := \{\leq[2], \text{Min}[1], \text{Max}[1], \overline{\leq}[2], \overline{\text{Min}}[1], \overline{\text{Max}}[1]\}$$

with each R in Y_{out} labelled $\text{acc}(R) = \text{input}$, $Y_{\text{mem}} := \{\text{Adom}[1]\}$, $Y_{\text{msg}} := \{\text{AdomSnd}[1]\}$, and $Y_{\text{sys}} := \{\text{Id}[1], \text{Leader}[1]\}$. First, we define the mappings $M_{\text{snd}}, M_{\text{in}}, M_{\text{del}}, M_{\text{out}}$ as follows:

- $(I, J) \in M_{\text{snd}}$ iff:
 - $a \in \text{Adom}(I|_{Y_{\text{in}}})$ implies $\text{AdomSnd}(a) \in J$ (nodes broadcast local active domain); and
 - $R(\overline{a}), \text{Leader}(x), \text{Id}(x) \in I$ implies $\text{SndR}(\overline{a}) \in J$ for all $R[k] \in Y_{\text{out}}$ (leader broadcasts its computed output to all other nodes);
- $(I, J) \in M_{\text{in}}$ iff $J = \{\text{Adom}(a) \mid \text{AdomSnd}(a) \in I\}$ (every node stores the active domain elements it receives in its local memory);
- $(I, J) \in M_{\text{del}}$ iff $J = \emptyset$ (nothing is deleted);
- $(I, J) \in M_{\text{out}}$ iff:
 - $\text{Adom}(a), \text{Adom}(b), \text{Leader}(x), \text{Id}(x) \in I$ with arbitrary domain order $\leq$ and $a \leq b$ implies $\leq(a, b), \overline{\leq}(b, a), \text{Min}(a), \overline{\text{Min}}(b), \text{Max}(b), \overline{\text{Max}}(a) \in J$ (when the first two active domain elements are discovered by the leader, generate an arbitrary order over them, update the minimum and maximum, and complements accordingly);
 - If $\leq(a, b), \overline{\leq}(b, a), \text{Min}(a), \overline{\text{Min}}(b), \text{Max}(b), \overline{\text{Max}}(a), \text{Adom}(c), \text{Adom}(d), \text{Leader}(x), \text{Id}(x) \in I$ with arbitrary domain order $\leq$ and $c \leq d$, then $\leq(c, d), \leq(d, b), \overline{\leq}(d, c), \overline{\leq}(c, a) \in J$ (after the first two active domain elements are processed by the leader, keep adding new elements before the maximal element); and
 - $\text{SndR}(\overline{a}) \in I$ implies $R(\overline{a}) \in J$ for all $R[k] \in Y_{\text{out}}$.

To show correctness of Π , note that each fair run $\mathcal{R}$ of Π over a distributed instance H will eventually lead to a configuration ρ where every node has $\text{Adom}(\text{flat}(H))$ stored in its local Adom relation. One of these nodes is the leader, and picks a minimum, maximum, and

their complements with the first two incoming active domain elements. Then it gradually extends the order relation and its complements by inserting the elements behind the maximum. During this process the computed output is always broadcast to all other nodes in the network. It is clear that each node eventually computes $\overline{\mathbf{C}}(\sigma)$, this concludes the proof. $\square$

Proof of Proposition 41

Proposition 44. $\overline{\mathbf{Ord}}(\sigma) \in \mathcal{F}(\mathbf{Id}+\mathbf{Ord}^+)$, for every schema σ .

Proof. Let σ be some schema. The proof is by construction of a $\mathbf{Id}+\mathbf{Ord}^+$ -transducer that computes $\overline{\mathbf{Ord}}(\sigma)$. Recall that $\overline{\mathbf{Ord}}(\sigma)$ is a behavior from σ to $\{\leq[2], \overline{\leq}[2]\}$. For the construction will assume the output schema is primed (to prevent collisions with relation names in the system relations), thus equaling $\{\leq'[2], \overline{\leq}'[2]\}$.

For this, consider transducer schema

$$Y = (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{mem}}, Y_{\text{msg}}, Y_{\text{sys}})$$

with $Y_{\text{in}} := \sigma$, $Y_{\text{out}} := \{\leq[2], \overline{\leq}[2]\}$, $Y_{\text{mem}} := \{\text{Adom}[1]\}$, and with $Y_{\text{msg}} := \{\text{AdomSnd}[1]\}$, and finally $Y_{\text{sys}} := \{\leq[2], \overline{\leq}[2], \text{Min}[1], \text{Max}[1]\}$.

Now consider the transducer

$$\Pi = (\mathbf{Id}+\mathbf{Ord}^+, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$$

with $(I, J) \in M_{\text{out}}$ iff

$$J = \{\leq'(\overline{a}) \mid \overline{a} \subseteq \text{Adom}(I) \text{ with } \leq(\overline{a}) \in I\} \\ \cup \{\overline{\leq}'(\overline{a}) \mid \overline{a} \subseteq \text{Adom}(I) \text{ with } \leq(\overline{a}) \notin I\}.$$

With $(I, J) \in M_{\text{snd}}$ iff $J = \{\text{AdomSnd}(a) \mid a \in \text{Adom}(I)\}$, and with $(I, J) \in M_{\text{in}}$ iff $J = \{\text{Adom}(a) \mid a \in \text{Adom}(I)\}$, and finally with $(I, J) \in M_{\text{del}}$ iff $J = \emptyset$.

Intuitively, Π lets every node broadcast its active domain, store the active domain of the network in its local memory and forward its $\leq$ system relation (and its complement) to the output. Correctness of the transducer is straightforward. This concludes the proof. $\square$

Proposition 45. $\overline{\text{Ind}}(\sigma) \in \mathcal{F}(\text{Id}+\text{Ind}^+)$, for every schema σ .

Proof. The proof is completely analogous to the proof of Proposition 44, except that $\leq'$ becomes Index_R and Node for every $R \in \sigma$ and $\leq'$ becomes Index'_R for every $R \in \sigma$. $\square$

Proposition 46. $\overline{\text{Ord}}(\sigma) \in \mathcal{F}(\text{Id}+\text{L})$, for every schema σ .

Proof. Let σ be an arbitrary schema. We construct a transducer

$$\Pi = (\text{Id}+\text{L}, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$$

over the transducer schema $Y = (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{mem}}, Y_{\text{msg}}, Y_{\text{sys}})$ computing the behavior $\overline{\text{Ord}}(\sigma)$ to σ' .

Transducer Construction We define the schemas in the transducer schema Y as follows, for each $R[k] \in Y_{\text{in}}$ we have:

$$\begin{aligned} Y_{\text{in}} &= \sigma, \\ Y_{\text{out}} &= \{\leq[2]\} \text{ with } \text{acc}(\leq) = \text{input}, \\ Y_{\text{mem}} &= \{\text{ExtR}[k+1], \text{AckR}[k], \text{ExtDone}[1], \leq[2]\}, \\ Y_{\text{msg}} &= \{\text{SndR}[k+2], \text{SndAckR}[k+1], \text{DoneSnd}[1], \text{Snd}\leq[3]\}, \\ Y_{\text{sys}} &= \{\text{Id}[1], \text{Leader}[1]\}. \end{aligned}$$

First, we define $(I, J) \in M_{\text{snd}}$ if, and only if J is a minimal instance satisfying the properties:

1. if $\text{Id}(x) \in I$, then for every $R[k] \in Y_{\text{in}}$:
 - (a) $R(\bar{a}), \text{Leader}(y) \in I$ implies $\text{SndR}(y, x, \bar{a}) \in J$;
 - (b) $\text{ExtR}(y, \bar{a}), \text{Leader}(x) \in I$ implies $\text{SndAckR}(y, \bar{a}) \in J$;
 - (c) if for all $R(\bar{a}) \in I$ (with $R[k] \in Y_{\text{in}}$) there exists an $\text{AckR}(\bar{a}) \in I$ then $\text{DoneSnd}(x) \in J$; and
 - (d) If $\leq(a, b), \text{Leader}(x), \text{ExtR}(y, \bar{a}) \in I$, then $\text{Snd}\leq(y, a, b) \in J$.

Hence, each node (including the leader) sends its entire local input instance to the leader node. The leader sends back an acknowledgment for each fact, and the sending node announces to the leader that it is done once it has received acknowledgments for every fact. At that point, the leader broadcasts a new order relation.

Next, we define $(I, J) \in M_{\text{in}}$ if, and only if J is a minimal instance satisfying the upcoming properties:

2. If $\text{Id}(x) \in I$, then for each $R[k] \in Y_{\text{in}}$:
 - (a) $\text{SndR}(x, y, \bar{a}), \text{Leader}(x) \in I$ implies $\text{ExtR}(y, \bar{a}) \in J$;
 - (b) $\text{SndAckR}(x, \bar{a}), \in I$ implies $\text{AckR}(\bar{a}) \in J$;
 - (c) $\text{DoneSnd}(y) \in I$ implies $\text{ExtDone}(y) \in J$;
 - (d) $\text{Leader}(x), \text{ExtDone}(y) \in I$ and there is no $\leq$ -fact in I yet, then for some arbitrary linear order $\leq$ there exist $\leq$ -facts in J encoding $\leq$ over $\text{Adom}(I)$; and
 - (e) If $\text{Leader}(x), \text{ExtDone}(y) \in I$ and there are $\leq$ -facts in I already, then for some linear order $\leq^+$ with a lower bound starting at the upper bound of the $\leq$ -order already there, we add $\leq$ -facts in J encoding $\leq^+$ over the fragment of values from $\text{Adom}(I)$ which does not have $\leq$ -facts yet.

The leader receives input facts from all of the nodes, and each time it is done receiving one node's full local instance it adds to the global order relation. The lower bound of the new order relation starts at the upper bound of the order already present (or it generates an arbitrary one if none is present yet), the M_{snd} mapping is then responsible for broadcasting it to every other node.

Subsequently, we define $(I, J) \in M_{\text{out}}$ if, and only if J is a minimal instance such that:

3. $\text{Snd}\leq(x, a, b), \text{Id}(x) \in I$ implies $\overline{\leq}(a, b) \in J$.

Intuitively, M_{out} listens for the order broadcast from the leader, and writes it to its output channel immediately, building up $\overline{\text{Ord}}(\sigma)$.

Lastly, we define M_{del} as the empty mapping, that is $(I, \emptyset) \in M_{\text{del}}$ for each instance I . This concludes the definition of Π .

Transducer Correctness We show that Π computes $\overline{\text{Ord}}(\sigma)$. Let H be an arbitrary distributed instance over Y_{in} . We show that there exists a run $\mathcal{R}$ of Π on H such that $(H, \text{do}(\mathcal{R})) \in \overline{\text{Ord}}(\sigma)$. Let $\mathcal{R}$ be a run of Π on H , which is organised in rounds where each node in $\text{netw}(H)$ becomes active precisely once. In the first round every node broadcasts its instance to the leader (*c.f.*, 1a), in the next round, the leader reads its entire message instance and stores facts locally (*c.f.*, 2a). Subsequently, there is a round where the leader sends acknowledgments to all nodes for the facts it received (*c.f.*, 1b), which are then read from the message instance by the other nodes in the next round (*c.f.*, 2b). In the round that follows, all nodes notify the leader that they are done with transmitting their instance (*c.f.*, 1c), the leader reads these notifications from the message instance and stores them locally (*c.f.*, 2c). Then, there is a round where the leader generates order facts matching an a linear order $\leq$ (*c.f.*, 2d), which are sent to all other nodes in the following round (*c.f.*, 1d). Lastly, all nodes read their entire message instance and store the order relation in their output instance (*c.f.*, 3), at this point the $\mathcal{R}$ reaches a quiescence point $\text{do}(\mathcal{R})$. In this case, for any $a, b \in H$ we have computed an order relation $a \leq b$ which is the complement of some order relation $b \leq^- a$, hence $(H, \text{do}(\mathcal{R})) \in \overline{\text{Ord}}(\sigma)$.

Next, we show that each fair run $\mathcal{R}$ of Π on H satisfies $(H, \text{do}(\mathcal{R})) \in \overline{\text{Ord}}(\sigma)$. Let $\mathcal{R}$ be any fair run of Π on H . Consider that $\text{netw}(H)$ is subdivided into two subnetworks, one network $\mathcal{N}_1$ containing the leader and one network $\mathcal{N}_2$ not containing the leader. First the leader will compute an ordering and place it on the nodes in $\mathcal{N}_1$ since it cannot distinguish between $\mathcal{N}_1 \subsetneq \text{netw}(H)$ and $\mathcal{N}_1 = \text{netw}(H)$. Next, the delay is restored and the leader generates a new arbitrary linear ordering starting at the upper bound of the previously generated order (*c.f.*, 2e). This is then broadcast to every other node and output as usual (*c.f.*, 1d, 3). At this point it is clear that $\mathcal{R}$ reaches a quiescence point and $(H, \text{do}(\mathcal{R})) \in \overline{\text{Ord}}(\sigma)$.

This shows that Π computes $\overline{\text{Ord}}(\sigma)$, which concludes the proof. $\square$

Proposition 47. $\text{Ind}^+(\sigma) \in \mathcal{F}_T(\text{Id}+\text{All})$, for every schema σ .

Proof. Consider the **Id+All**-transducer Π constructed in Proposi-

tion 29, with the following changes:

- Let Y_{in} be defined as σ .
- Instead of computing an arbitrary behavior $B \in \mathcal{D}$, define B to be a variation of $\text{Ind}^+(\sigma)$ that creates an index fact on each node for every fact in the local input of that node and then additionally includes a $\text{Termination}()$ fact. Note that $\text{Ind}^+(\sigma) \in \mathcal{D}$ as well.

The argumentation for showing that Π computes this variation of $\text{Pol}^+(\sigma) \in \mathcal{D}$ is analogous to Proposition 29, since this is simply a special case of the latter proof. $\square$

Proof of Proposition 42

Proposition 48. $\mathcal{F}(\text{Id}+\text{L}) \setminus \mathcal{F}(\text{Id}+\text{Ind}+\text{Ord}^+) \neq \emptyset$.

Proof. First, we show $\text{L}(\sigma) \in \mathcal{F}(\text{Id}+\text{L})$ by constructing a transducer

$$\Pi = (\text{Id}+\text{L}, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$$

over a transducer schema $Y = (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{mem}}, Y_{\text{msg}}, Y_{\text{sys}})$ such that Π computes B . First, we define the schemas $Y_{\text{in}} = \{\text{Node}[0]\}$, $Y_{\text{out}} = \{\text{Leader}'[1]\}$, $Y_{\text{mem}} = Y_{\text{msg}} = \emptyset$, and $Y_{\text{sys}} = \{\text{Id}[1], \text{Leader}[1]\}$. Next, we define $M_{\text{in}}, M_{\text{del}}$ and M_{snd} as the empty mappings, *i.e.*, $(I, \emptyset) \in M_{\text{in}}$, $(I, \emptyset) \in M_{\text{del}}$, and $(I, \emptyset) \in M_{\text{snd}}$ for all instances I over Y_{in} . And lastly, we define $(I, J) \in M_{\text{out}}$ if, and only if J is the smallest instance such that $\text{Leader}(x) \in I$ implies $\text{Leader}'(x) \in J$. This concludes the construction of Π . It is trivial to see that Π computes B .

Next, we show that $\text{L}(\sigma) \notin \mathcal{F}(\text{Id}+\text{Ind}+\text{Ord}^+)$. First, observe that

$$\mathcal{F}(\text{Id}+\text{Ind}+\text{Ord}^+) \subseteq \mathcal{F}(\text{Id}+\text{Ord}+\text{Ord}^++\text{Ind})$$

by Proposition 13, which we use to show that $\text{L}(\sigma) \notin \mathcal{M}_{\text{Ord}^++\text{Ind}}$, showing that

$$\text{L}(\sigma) \notin \mathcal{F}(\text{Id}+\text{Ord}+\text{Ord}^++\text{Ind}), \text{ and}$$

$$\text{L}(\sigma) \notin \mathcal{F}(\text{Id}+\text{Ind}+\text{Ord}^+).$$

Let $H_0, H_1, \dots, H_k$ over Y_{in} , U over Y_{sys} and $K_0, K_1, \dots, K_k$ over Y_{out} be distributed instances, such that $\text{netw}(H_a) \cap \text{netw}(H_b) = \emptyset$ for all

$a \neq b \in [0, k]$, $(H_i, U|_{\text{netw}(H_i)}) \in \mathbf{Ind+Ord}^+(Y_{\text{in}})$, $(H_i, K_i) \in B_Q$ for all $i \in [1, k]$, and $(\bigcup_{i=0}^k H_i, U) \in \mathbf{Ind+Ord}^+(Y_{\text{in}})$. We show there exists a distributed instance K over Y_{out} with $(\bigcup_{i=0}^k H_i, K) \in B$ satisfying $\bigcup_{i=0}^k K_i \not\subseteq K$. Let each K_i with $i \in [1, k]$ designate a leader according to the minimal node identifier in $U|_{\text{netw}(H_i)}$ with $(H_i, U|_{\text{netw}(H_i)}) \in \mathbf{Ind+Ord}^+(Y_{\text{in}})$, hence $\text{Leader}(x) \in K_i$. And, let K_0 contain $\text{Leader}(z)$ with z smaller than any other $\text{Leader}(x) \in K_i$. Then we can see that if K is a distributed instance with $(\bigcup_{i=0}^k H_i, K) \in B$, each of the K_i satisfies $K_i \subseteq K$ since K contains $\text{Leader}(z)$ and K_i does not contain $\text{Leader}(z)$. This shows that $L(\sigma) \notin \mathcal{M}_{\text{Ord}^+ + \text{Ind}}$, concluding the proof. $\square$

Proposition 49. $\mathcal{F}(\mathbf{Id+Ord}^+) \setminus \mathcal{F}(\mathbf{Id+L+Ind}^+) \neq \emptyset$.

Proof. Consider the behavior B from

$$\sigma := \{\text{Node}[0]\} \text{ to } \sigma' := \{\text{SingleNodeNetwork}[0]\}$$

with for every distributed instance H over σ :

$$(H, \{\text{SingleNodeNetwork}@a() \mid a \in \text{netw}(H)\}) \in B$$

if $|\text{netw}(H)| = 1$ and $(H, \emptyset) \in B$ otherwise. Clearly, $B \in \mathcal{F}(\mathbf{Id+Ord}^+)$ because $|\text{netw}(H)| = 1$ implies $|\text{Adom}(H)| = 1$ and thus, assuming $\text{netw}(H) = \{a\}$, $(H, K) \in \mathbf{Id+Ord}^+$ iff

$$K = \{\text{Id}(a), \leq(a, a), \text{Min}@a(a), \text{Max}@a(a)\}.$$

In all other scenarios (with $\text{netw}(H) > 1$), we have $(H, K) \in \mathbf{Id+Ord}^+$ with $\text{Min}@x(a), \text{Max}@x(b) \in H$ and $a \neq b$ for every $x \in \text{netw}(H)$. More precisely, the witnessing transducer can simply check if local Min and Max relations. If they both contain the same value, the fact `SingleNodeNetwork` should be produced. In all other cases (thus either containing different values, or if one or both are empty - implying an element is not yet part of the local active domain) the fact should not be produced.

It remains to show $B \notin \mathcal{F}(\mathbf{Id+L+Ind}^+)$. Towards a contradiction suppose there is a $\mathbf{Id+L+Ind}^+$ -transducer Π that computes B . Now let $H_1 = \{\text{Node}@a()\}$ and $H_2 = \{\text{Node}@b()\}$. Clearly,

$$(H_1, \{\text{SingleNodeNetwork}@a()\}), (H_2, \{\text{SingleNodeNetwork}@b()\}) \in B.$$

Remark that B is deterministic, thus B characterizes Π (as shown in Proposition 26). Therefore, there must also exist runs $\mathcal{R}_1$ on H_1 and $\mathcal{R}_2$ on H_2 of Π , with $\text{do}(\mathcal{R}_1) = \text{do}(\mathcal{R}_2) = \{\text{Node}@b()\}$.

Now, $H := \{\text{Node}@a(), \text{Node}@b()\}$ and arbitrary U such that $(H, U) \in \mathbf{Id}+\mathbf{L}+\mathbf{Ind}^+$. Without loss of generality we assume $\text{Leader}@a(a) \in U$ (otherwise the argument is analogous taking $\text{Leader}@b(b) \in U$).

In that case, we have $(H \cup K)(a) = (H_1 \cup K_1)(a)$. Therefore, we can construct a run $\mathcal{R}$ of Π on H in which node a does not receive messages for a significant amount of time and performs the same transitions as in $\mathcal{R}_1$ till the transition that leads to the quiescence configuration of $\mathcal{R}_1$, implying $\text{SingleNodeNetwork}@a \in \text{do}(\mathcal{R})$. Since $\mathcal{R}$ can still be extended to a fair run we have found the desired contradiction and conclude that Π (and by assumption no $\mathbf{Id}+\mathbf{L}+\mathbf{Ind}^+$ -transducer) can compute B . $\square$

Proposition 50. $\mathcal{F}(\mathbf{Id}+\mathbf{Ord}) \setminus \mathcal{F}(\mathbf{Id}+\mathbf{All}) \neq \emptyset$.

Proof. Consider a behavior B from $\sigma_1 := \{\mathbf{R}[1]\}$ to $\sigma_2 := \{\text{LocalMin}[1]\}$ which is defined by $(H, K) \in B$ iff K is a minimal, distributed instance with for every $x \in \text{netw}(H)$ exactly one fact $\text{LocalMin}@x(a)$ with $a \in \text{Adom}(H(x))$, and there exists an order $\leq$ over $\text{Adom}(H)$ such that for every $x \in \text{netw}(H)$, the fact $\text{LocalMin}@x(a)$ implies that a is the lowest value in $\text{Adom}(H)$ (according to $\leq$).

To see that $B \in \mathcal{F}(\mathbf{Id}+\mathbf{Ord})$, we define a transducer

$$\Pi = (\mathbf{Id}+\mathbf{Ord}, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$$

over transducer schema

$$\mathbf{Y} = (\mathbf{Y}_{\text{in}}, \mathbf{Y}_{\text{out}}, \mathbf{Y}_{\text{mem}}, \mathbf{Y}_{\text{msg}}, \mathbf{Y}_{\text{sys}})$$

computing B . We define the schemas of $\mathbf{Y}$ by $\mathbf{Y}_{\text{in}} = \sigma_1$, $\mathbf{Y}_{\text{out}} = \sigma_2$, $\mathbf{Y}_{\text{mem}} = \mathbf{Y}_{\text{msg}} = \emptyset$, and $\mathbf{Y}_{\text{sys}} = \{\leq[2], \text{Id}[1]\}$. Next, we define $M_{\text{in}}, M_{\text{del}}$ and M_{snd} as the empty mappings, i.e., $(I, \emptyset) \in M_{\text{in}}$, $(I, \emptyset) \in M_{\text{del}}$, and $(I, \emptyset) \in M_{\text{snd}}$ for all instances I over $\mathbf{Y}_{\text{in}}$. And lastly, we define $(I, J) \in M_{\text{out}}$ if, and only if J is the smallest instance such that $\mathbf{R}(x), \leq(z, x) \in I$ with $\mathbf{R}(z) \notin I$ then for $\mathbf{R}(x) \in I$ with no $\leq(z, x) \in I$ implies $\text{LocalMin}(x) \in J$. It can now be verified easily that Π computes B , hence $B \in \mathcal{F}(\mathbf{Id}+\mathbf{Ord})$.

To see that $B \notin \mathcal{F}(\mathbf{Id}+\mathbf{All})$ we make use of Proposition 14. For this, consider the distributed instance

$$H := \{\mathbf{R}@x(x), \mathbf{R}@x(y), \mathbf{R}@y(x), \mathbf{R}@y(y)\}.$$

Clearly, for

$$U := \{\mathbf{Id}@x(x), \mathbf{Id}@y(x), \mathbf{All}@x(x), \mathbf{All}@x(y), \mathbf{All}@y(x), \mathbf{All}@y(y)\},$$

we have $(H, U) \in \mathbf{Id}+\mathbf{Ord}(\sigma_1)$. Next, it is clear that $\beta : \{x \mapsto y, y \mapsto x\}$ is an automorphism for $H \cup U$ with order 1 and orbit size 1 for x and y . Now, Proposition 14 implies that β is an automorphism for at least one of the distributed instance K with $(H, K) \in B$. Since for the constructed distributed instance H there are only two options for K :

$$\begin{aligned} K_1 &:= \{\mathbf{LocalMin}@x(x), \mathbf{LocalMin}@y(x)\} \text{ or} \\ K_2 &:= \{\mathbf{LocalMin}@x(y), \mathbf{LocalMin}@y(y)\}, \end{aligned}$$

it is easy to verify that $\mathbf{LocalMin}@y(y) \in \beta(K_1)$ and thus $\beta(K_1) \neq K_1$ and, analogously, $\mathbf{LocalMin}@x(x) \in \beta(K_2)$ thus $\beta(K_2) \neq K_2$. This concludes the proof. $\square$

Proposition 51. $\mathcal{F}(\mathbf{Id}+\mathbf{Ind}) \setminus [\mathcal{F}(\mathbf{Id}+\mathbf{L}) \cup \mathcal{F}(\mathbf{Id}+\mathbf{Ord}^+)] \neq \emptyset$.

Proof. Let Q be a query from $\sigma := \{\mathbf{E}[2]\}$ to $\sigma' := \{\bar{\mathbf{E}}[2]\}$ with

$$Q(I) = \{\bar{\mathbf{E}}(a, b) \mid a, b \in \text{Adom}(H) \text{ with } E(a, b) \notin I\}.$$

First, we show that $B_Q \in \mathcal{F}(\mathbf{Id}+\mathbf{Ind})$. For this, let

$$\Pi = (\mathbf{Id}+\mathbf{Ind}, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$$

over schema $\mathbf{Y} = (\mathbf{Y}_{\text{in}}, \mathbf{Y}_{\text{out}}, \mathbf{Y}_{\text{mem}}, \mathbf{Y}_{\text{msg}}, \mathbf{Y}_{\text{sys}})$ with $\mathbf{Y}_{\text{in}} := \sigma_1, \mathbf{Y}_{\text{out}} := \sigma_2$,

$$\begin{aligned} \mathbf{Y}_{\text{mem}} &:= \{\mathbf{Adom}[1], \mathbf{IndexExt}_E[2]\}, \\ \mathbf{Y}_{\text{msg}} &:= \{\mathbf{AdomSnd}[1], \mathbf{IndexSnd}_E[3]\}, \text{ and} \\ \mathbf{Y}_{\text{sys}} &:= \{\mathbf{Id}[1], \mathbf{Index}_E[3], \mathbf{Node}[1]\}. \end{aligned}$$

We define mappings $M_{\text{snd}}, M_{\text{in}}, M_{\text{del}}, M_{\text{out}}$ as follows:

- $(I, J) \in M_{\text{snd}}$ iff

$$J = \{\text{AdomSnd}(a) \mid a \in \text{Adom}(I|_{Y_{\text{in}}})\} \cup \{\text{IndexSnd}_E(x, \bar{a}) \mid \text{Index}_E(x, \bar{a}) \in I\}$$

(every node broadcast its local active domain as well as its index entries);

- $(I, J) \in M_{\text{in}}$ iff

$$J = \{\text{Adom}(a) \mid \text{AdomSnd}(a) \in I\} \cup \{\text{IndexExt}_E(\bar{a}) \mid \text{IndexSnd}_E(x, \bar{a}) \in I, \text{Id}(x) \in I\}$$

(every node stores the active domain elements it receives in its local memory instance);

- $(I, J) \in M_{\text{del}}$ iff $J = \emptyset$ (nothing is deleted);
- $(I, J) \in M_{\text{out}}$ with

$$J = \{\bar{E}(a, b) \mid \{a, b\} \subseteq \text{Adom}(I|_{Y_{\text{in}} \cup Y_{\text{mem}}}), \text{IndexExt}_E(a, b) \in I, E(a, b) \notin I\}$$

(a node tests for every combination of (not necessarily different) active domain elements a and b if it is responsible for $E(a, b)$. If then $E(a, b)$ is not in the local input instance, $\bar{E}(a, b)$ should be in the output).

For the correctness of Π , we observe that every fair run $\mathcal{R}$ of Π over a distributed instance H will eventually lead to a configuration ρ where every node has $\text{Adom}(\text{flat}(H))$ stored in its local relation Adom . Since **Ind** ensures that for every fact $E(a, b)$, with $\{a, b\} \in \text{Adom}(H)$ there is at least one node x in the network such that the system instance contains $\text{Index}_E(x, a, b)$, it follows that $\bar{E} @ x(a, b) \in \text{do}(\mathcal{R})$, and thus $\bar{E}(a, b) \in \text{flat}(\text{do}(\mathcal{R}))$. By generality, we thus have that $Q(\text{flat}(H)) \subseteq \text{flat}(\text{do}(\mathcal{R}))$.

Since a node x will only produce $\bar{E}(a, b)$ if $\text{Policy}_E(x, a, b)$ exists, it follows from the absence of $E(a, b)$ in its local input instance that $\bar{E}(a, b) \notin \text{flat}(H)$ and thus, by generality, that $\text{flat}(\text{do}(\mathcal{R})) \subseteq Q(\text{flat}(H))$, which concludes the correctness proof.

To see that $B_Q \notin \mathcal{F}(\mathbf{Id}+\mathbf{L})$ we recall that $\mathbf{Ord}(\sigma) \in \mathcal{F}(\mathbf{Id}+\mathbf{L})$ (c.f., Proposition 41) implies $\mathcal{F}(\mathbf{Id}+\mathbf{L}) = \mathcal{F}(\mathbf{Id}+\mathbf{Ord}+\mathbf{L})$, and thus we can apply

Theorem 16. To this end, let $H_1 := \{E(a, b)\}$ and

$$U_1 := \{Id@a(a), \leq@a(a, a), \leq@a(a, b), le@a(b, b), Leader@a(a)\}.$$

Clearly, $(H_1, U_1) \in \mathbf{Id+Ord+L}(\sigma_1)$. Also, $(H_1, K_1) \in B_Q$ for $K_1 := \{\bar{E}@a(b, a), \bar{E}@a(b, b), \bar{E}@a(a, a)\}$. Now consider:

$$H_0 := \{E@b(b, a), E@b(b, b), E@b(a, a)\}, \text{ and}$$

$$K_0 := \{Id@b(b), \leq@b(a, a), \leq@b(a, b), \leq@b(b, b), Leader@b(a)\}.$$

Hence, $(H_1 \cup H_2, U_1 \cup U_2) \in \mathbf{Id+Ord+L}(\sigma_1)$, but the only K for which $(H_1 \cup H_2, K) \in B_Q$ is $K := \emptyset$ thus with $K_1 \not\subseteq K$, which is the desired contradiction showing that B_Q is indeed not in $\mathcal{F}(\mathbf{Id+L})$.

The argument for $B_Q \notin \mathcal{F}(\mathbf{Id+Ord}^+)$ is completely analogous, since

$$\mathcal{F}(\mathbf{Id+Ord}^+) = \mathcal{F}(\mathbf{Id+Ord}^+ + \mathbf{Ord})$$

due to Proposition 41. Only, U_1 should be replaced by

$$U'_1 := (K_1 \setminus \{Leader@a(a)\}) \cup \{Min@a(a), Max@a(b)\}$$

and U_0 by $U'_0 := (U_0 \setminus \{Leader@b(a)\}) \cup \{Min@b(a), Max@b(b)\}$, as then (H_1, U'_1) and $(H_1 \cup H_0, U'_1 \cup U'_0)$ are in $\mathbf{Id+Ord}^+ + \mathbf{Ord}(\sigma_1)$.

This concludes the proof. □

Proposition 52.

$$\mathcal{F}(\mathbf{Id+Ind}^+) \setminus [\mathcal{F}(\mathbf{Id+Ind+L}) \cup \mathcal{F}(\mathbf{Id+Ind+Ord}^+)] \neq \emptyset.$$

Proof. Consider the query Q from $\sigma_1 := \{E[2]\}$ to $\sigma_2 := \{NC[2]\}$ with $(I, J) \in Q$ iff

$$J := \{NC(a, b) \mid \{a, b\} \subseteq \text{Adom}(I)$$

and there is no sequence of facts

$$E(a, c_1), E(c_1, c_2), \dots, E(c_k, b)\} \text{ in } I\}.$$

First, we show $B_Q \in \mathcal{F}(\mathbf{Id+Ind}^+)$ by defining a $\mathbf{Id+Ind}^+$ -transducer Π that computes it. For this, consider schema

$$Y = (Y_{\text{in}}, Y_{\text{out}}, Y_{\text{mem}}, Y_{\text{msg}}, Y_{\text{sys}})$$

with $Y_{\text{in}} := \sigma_1, Y_{\text{out}} := \sigma_2,$

$Y_{\text{msg}} := \{\text{ESnd}[2], \text{AckSnd}[3], \text{CompleteSnd}[2], \text{IndexSnd}_E[3]\},$

and

$Y_{\text{mem}} := \{\text{EExt}[2], \text{Complete}[1], \text{Ack}[2], \text{IndexExt}_E[2]\}.$

Let $\Pi = (\mathbf{Id} + \mathbf{Ind}^+, M_{\text{out}}, M_{\text{in}}, M_{\text{del}}, M_{\text{snd}})$ with schema Y and mappings as follows:

- $(I, J) \in M_{\text{in}}$ iff

$$\begin{aligned} J = & \{\text{EExt}(a, b) \mid \text{ESnd}(a, b) \in I\} \\ & \cup \{\text{Ack}(a, b, x) \mid \text{AckSnd}(a, b, x) \in I\} \\ & \cup \{\text{Complete}(a) \mid \text{CompleteSnd}(a, x), \text{Id}(x) \in I\} \\ & \cup \{\text{IndexExt}_E(\bar{a}) \mid \text{IndexSnd}_E(x, \bar{a}) \in I, \text{Id}(x) \in I\} \end{aligned}$$

(every node stores all facts it receives in its local memory instance);

- $(I, J) \in M_{\text{snd}}$ iff

$$\begin{aligned} J = & \{\text{ESnd}(a, b) \mid \text{E}(a, b) \in I\} \\ & \cup \{\text{AckSnd}(a, b, x) \mid \text{EExt}(a, b), \text{Id}(x) \in I\} \\ & \cup \{\text{CompleteSnd}(a, x) \mid \text{IndexExt}_E(a, a) \in I \\ & \quad \text{and for every fact } \text{E}(\bar{c}) \in I \\ & \quad \text{with } \bar{c} \text{ containing } a : \text{Ack}(\bar{c}, x) \in I\} \\ & \cup \{\text{IndexSnd}_E(x, \bar{a}) \mid \text{Index}_E(x, \bar{a})\} \end{aligned}$$

(every node broadcast its local E relation, sends an acknowledgment for every ESnd message it receives, and also sends $\text{CompleteSnd}(a, x)$ messages if it is responsible for that a and has received an ack from x for all E -facts in its local instance mentioning a);

- $(I, J) \in M_{\text{del}}$ iff $J = \emptyset$ (nothing is deleted);
- $(I, J) \in M_{\text{out}}$ iff

$$\begin{aligned} J = & \{\text{NC}(a, b) \mid a, b \in \text{Adom}(I|_{\{E[2]\}}) \\ & \text{and for all values } c \text{ in the components of } a \text{ and } b \\ & \text{we have that } \text{Complete}(c) \in I\}. \end{aligned}$$

(a node tests for every combination of (not necessarily different) active domain elements a and b if these are part of a component that cannot be extended in any direction in the future.)

Correctness of Π follows from the definition of $\mathbf{Ind}^+$.

We proceed by showing $B_Q \notin \mathcal{F}(\mathbf{Id}+\mathbf{Ind}+\mathbf{L})$. For this, we again make use of $\mathbf{Ord}(\sigma) \in \mathcal{F}(\mathbf{Id}+\mathbf{L})$ (c.f., Proposition 41) implying $\mathcal{F}(\mathbf{Id}+\mathbf{L}) = \mathcal{F}(\mathbf{Id}+\mathbf{Ord}+\mathbf{L})$ and thus that we can apply Theorem 16. Let $H_1 := \{E(a, a), E(b, b)\}$ and $H_0 := \{Ea, b\}$. Let

$U_1 := \{\mathbf{Id}@a(a), \mathbf{Leader}@a(a), \mathbf{Index}_E@a(a, a, a), \mathbf{Node}@a(a)\}$, and

$$\begin{aligned} U_0 := & \{\mathbf{Index}_E@b(b, b, b), \\ & \mathbf{Leader}@b(a), \\ & \mathbf{Index}_E@b(b, a, b), \\ & \mathbf{Index}_E@b(b, b, a), \\ & \mathbf{Node}@b(b)\}. \end{aligned}$$

Clearly, (H_1, U_1) and $(H_1 \cup H_0, U_1 \cup U_0)$ are in $\mathbf{Id}+\mathbf{Ord}+\mathbf{L}$. Furthermore, for $K_1 := \{\mathbf{NC}@a(a, b), \mathbf{NC}@a(b, a)\}$ we have $(H_1, K_1) \in B_1$. Now, the conclusion follows from $(H_1 \cup H_0, K) \in B_1$ implying $K = \emptyset$, and thus $B_Q \notin \mathcal{M}_L$.

The argument for $B_Q \notin \mathcal{F}(\mathbf{Id}+\mathbf{Ind}+\mathbf{Ord}^+)$ is completely analogous, but now taking $H_1 := \{E(a, a), E(b, b)\}$ and $H_0 := \{E(a, c), E(c, b)\}$ with

$$\begin{aligned} U_1 := & \{\mathbf{Id}@a(a), \mathbf{Index}_E@a(a, a, a), \\ & \mathbf{Index}_E@a(a, a, b), \mathbf{Index}_E@a(a, b, a), \\ & \mathbf{Index}_E@a(a, b, b), \leq@b(a, a), \\ & \leq@b(a, b), \leq@b(b, b)\}, \text{ and} \\ U_0 := & \{\mathbf{Index}_E@b(b, c, c), \mathbf{Index}_E@b(b, a, c), \mathbf{Index}_E@b(b, c, a), \\ & \mathbf{Index}_E@b(b, b, c), \\ & \mathbf{Node}@b(b), \mathbf{Index}_E@b(b, c, b), \\ & \leq@b(a, a), \leq@b(a, b), \leq@b(b, b), \\ & \leq@b(a, c), \leq@b(b, c), \leq@b(c, c)\}. \end{aligned}$$

□

5 Related Work

As mentioned at various points throughout this part of the thesis, we built strongly on the work by Ameloot et al. [7, 8].

To the best of our knowledge (as of May 2026), the only other published work from the database theory community about the topic of the CALM theorem is the work by Power et al. [74]. The authors present the semiautomaton model, which models state as elements of an arbitrary set rather than a database instance, and models state transitions as deterministic mappings between states. This allows their model to express arbitrary data such as CRDTs, which are harder to express in the relational model. In comparison, generalized transducer networks exclusively model state as relational data, but allow for non-deterministic insert, update, delete and output mappings. Furthermore, as long as the update function in the semiautomaton is both generic and domain-independent, one can define this function as a constraint to be used in a generalized transducer. Secondly, the authors present the *free-termination property* for pairs of queries and instances, intuitively meaning that a semiautomaton is able to individually terminate execution by setting a ready flag. In Section 7.4 (based on [11]), we expressed an identical property with a termination-aware transducer, aside from the flag now being called “terminated” instead of “ready”.

Lastly, our system model bears resemblance to models used for reasoning about shared knowledge in multi-agent systems [33]. In particular, the message passing systems considered by Chandy and Misra [22]. However, they do not study the same problems we do.

6 Summary

In this chapter we introduced the following concepts and results.

- We studied the classes of node monotone and strict node monotone behaviors and their corresponding transducers, defining the concept of monotonicity for behaviors.
- We defined the class of C -monotone behaviors, which allows us to generalize the idea of progressively weaker forms of node monotonicity given progressively stronger constraints C .
- We reproduced prior results for queries within the model of behaviors and generalized transducer networks.
- Finally, we presented a coordination spectrum, providing an overview of the most important classes of behaviors we have studied in this part of the thesis.

Conclusion

1 Transaction Scheduling

There is still a significant gap between the number of contention points needed in the construction used in the NP-completeness proof of transaction schedulability (Theorem 3), and the single contention point which was needed for the polynomial time scheduling algorithm introduced in Section 3.3. An obvious direction in which the work could be extended, is to look for a dichotomy result based on the number of contention points (*e.g.*, the problem is NP-hard when the number of contention points exceeds 3, and otherwise it is in P). A possible place in which to look for inspiration could be the job shop scheduling literature, given that the problem is known to be NP-hard for 3 machines (*i.e.*, objects in transaction scheduling parlance), and given that a transaction schedule is also a valid (yet not optimal!) job shop schedule, but not vice versa.

Furthermore, there are still a number of lower bounds missing from Table 4.1. Specifically for the sets $\mathbb{T}_{\ell,3,n}$ with $7 \leq n$, and $\mathbb{T}_{\ell,m,n}$ in case $n \neq m!$ and $3 \leq n$. It remains open whether the upper bounds can be lower than the ones provided, or whether the lower bounds can be made much higher. A possible avenue to investigate would be approaches based on the Lovász local lemma, which is often used for such lower bounds in the scheduling literature.

Both the work about complexity and bounds is relevant in order to investigate the approximation complexity of transaction scheduling. The domain holds many interesting results for the job shop scheduling (*e.g.*, Mastrolili et al. [63], and Fishkin et al. [39, 40]). It involves constructing approximation-preserving reductions (*e.g.* L-reductions [29, 46]) from existing problems in discrete optimiza-

tion, which then enable us to decide membership in the class of approximable optimization problems (APX), or the class that admits a polynomial-time approximation scheme (PTAS). The former would indicate that there are tractable algorithms for which the output schedule has a makespan within a constant factor of its optimal makespan, and the latter implies that we can find a tractable algorithm that gets arbitrarily close to the optimal makespan.

Lastly, the algorithms provided in this work require knowledge of the transaction set before execution. To bridge this gap, queuing theory may be an inspiration [41]. In such a model, the arrival of transactions into the system becomes a stochastic process, potentially increasing the complexity further.

2 Coordination-Freedom

The work around coordination-freedom in relational transducer networks can take several directions. First it would be interesting to enhance transducers with the ability to do value invention, as this would allow more closely modeling algorithms that occur in the real world, where counters and arithmetic are plentiful. We also noticed that under the absence of order and a leader, but under the presence of an **All**-relation, leader election is sometimes expressible but sometimes not, depending on whether the distributed input itself exhibits enough asymmetry. A characterization of this problem seems to be essential to be able to semantically characterize classes of behaviors computable without a tie-breaking mechanism in the system configuration.

Lastly, we remark that it may be interesting to compare classes of configurations with respect to additional properties of behaviors, as we did focusing on deterministic behaviors. One possibly interesting property would be to study the computability of behaviors that are independent of the specific non-deterministic choice of system instance.

Encodings for Transaction Scheduling

We provide two encodings of the transaction scheduling optimization problem into different NP complete problems for which well-optimized solvers exist. The first encoding is to Satisfiability (SAT), and the second encoding is to the Julia library JuMP¹, a Mixed Integer Linear Programming (ILP) solver.

1 Transaction Scheduling as a SAT Problem

We start with a CNF encoding for schedules, we denote the constructed CNF formula $\text{Sched}(T)$ for some transaction set T . Let T be an arbitrary set of transactions. A schedule assigns to each transaction $\tau \in T$ exactly one position from $\{1, \dots, |T|\}$. To encode this, we introduce a number of clauses. In what follows, we can read $S_{i,j}$ being true as τ_i being scheduled to run as the j 'th transaction in the schedule.

$$\text{for all } \tau_i \in T, \quad S_{i,1} \vee S_{i,2} \vee \dots \vee S_{i,|T|} \quad (\text{A.1})$$

$$\text{for all } \tau_i \in T, \text{ and } j \neq k \in \{1, \dots, |T|\}, \quad \neg S_{i,j} \vee \neg S_{i,k} \quad (\text{A.2})$$

Furthermore, we must make sure that each position is occupied by exactly one transaction. For this, we introduce two more kinds of clauses.

$$\text{for all } j \in \{1, \dots, |T|\}, \quad S_{1,j} \vee S_{2,j} \vee \dots \vee S_{|T|,j} \quad (\text{A.3})$$

$$\text{for all } j \in \{1, \dots, |T|\}, \text{ and } \tau_i \neq \tau_k \in T \quad \neg S_{i,j} \vee \neg S_{k,j} \quad (\text{A.4})$$

¹<https://jump.dev/>

This results in a quadratic number of variables and clauses.

Lemma 12. $\text{Sched}(T)$ contains $\mathcal{O}(|T|^2)$ variables and $\mathcal{O}(|T|^2)$ clauses.

Proof. For the number of variables, observe that (1) introduces $|T|^2$ variables. This readily gives us the $\mathcal{O}(|T|^2)$ bound on the number of variables.

Next, observe that (1) and (3) jointly introduce $2|T|$ clauses. Furthermore, (2) and (4) each introduce pairwise combinations of variables chosen from $|T|$ positions. This results in exactly $2\binom{|T|}{2}$ clauses, where $\binom{n}{k}$ is defined as $n!/k!(n-k)!$. This is also known as the binomial coefficient. A well known upper-bound for the binomial coefficient is $n^k/k!$, which gives $\mathcal{O}(2|T| + 2(|T|^2/2)) = \mathcal{O}(|T|^2)$ clauses in total. This concludes the proof. $\square$

Precedence Graphs

We now encode the structures needed for calculating costs of precedence graphs, which we denote by the CNF formula $\text{Prec}(T)$, which includes $\text{Sched}(T)$.

A naive way to encode each precedence graph, is to include a variable for each edge in the graph. Hence, for the schedule s in Example 3, we would have the edges $(W_1[c], R_1[a])$ and $(R_1[a], R_1[b])$ for τ_1 , $(R_2[a], W_2[c])$ and $(W_2[c], R_2[c])$ for τ_2 , and so on, together with a number of edges corresponding with conflicts between different transactions. Instead we will introduce variables for each conflict, and annotate how many operations are between this conflict and the next one by means of a weight. When there is no prior conflict, or no next conflict, we use the special nodes $\top$ and $\perp$ which respectively represent a source and sink. This transformation is depicted in Figure A.1.

We define a *compression* of a precedence graph P_s as a weighted directed graph $\text{comp}(P) = (V, A, w)$ where

- V consists of $\top, \perp$ and all conflicting operations in the schedule s ; and

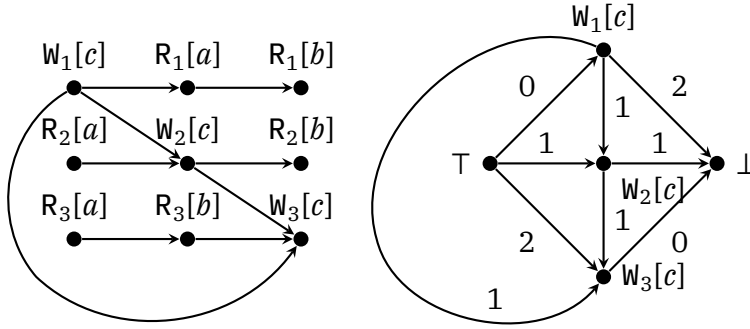


Figure A.1 – The left shows a regular precedence graph. The right shows a compressed precedence graph, only showing conflicting operations.

- if $o_1, o_2, \dots, o_k$ are the conflicting operations of a single transaction in T , then the arcs $(\top, o_1), (o_1, o_2), \dots, (o_k, \perp)$ are in A ; and
- their weights $w((u, v))$ are defined by the length of the shortest path between u and v . If o_i is the first (respectively, last) operation in the transaction, then $w((\top, o_i))$, (respectively, $w((o_i, \perp))$) is defined to be 0.

Furthermore, if P_s contains an arc (o_i, o_j) where $o_i \in \tau_i$ and $o_j \in \tau_j$ with $\tau_i \neq \tau_j \in T$ then (o_i, o_j) is also present in A , and $w((o_i, o_j))$ is defined to be 1, we define $comp'(P_s)$ to be $comp(P_s)$ without such (o_i, o_j) arcs, which we refer to as *cross-transaction*. Note that $comp'(P_s)$ is identical for every possible precedence graph of T .

Let τ be any transaction in T , and consider its $comp'(P_s) = (V, A, w)$. Then, we introduce the following clause.

$$\text{for all } (o_i, o_j) \in A, \quad A_{o_i, o_j, w((o_i, o_j))} \quad (\text{A.5})$$

The presence of cross-transaction edges is determined by the specific position of a transaction in the schedule. We now introduce the following clauses which make such an edge true, if it is allowed by the position of its constituent transactions. Hence, for all $\tau_i \neq$

$\tau_j \in T$, with conflicting $o_i \in \tau_i, o_j \in \tau_j$, we add:

$$\neg S_{i,|T|} \vee \neg A_{o_i,o_j,1} \quad (\text{A.6})$$

$$\neg S_{j,1} \vee \neg A_{o_i,o_j,1} \quad (\text{A.7})$$

$$\forall \ell < m : \{l, m\} \subseteq \{2, \dots, |T| - 1\}, \quad \neg S_{i,\ell} \vee \neg S_{j,m} \vee A_{o_i,o_j,1} \quad (\text{A.8})$$

At this point, every model of the constructed $\text{Prec}(t)$ represents a possible $\text{comp}(P_s)$.

Lemma 13. $\text{Prec}(T)$ contains $\mathcal{O}(|T|^2 \cdot |C|^2)$ variables and $\mathcal{O}(|T|^2 \cdot |C|^2)$ clauses, where C is the set of all conflicting operations in T .

Proof. For the number of variables, consider that (5) adds one variable per pair of conflicting operations (i.e., $\mathcal{O}(|C|)$ variables). Next, (6) and (7), both add a variable for each combination of conflicting operations, which yields $\mathcal{O}(2^{\binom{|C|}{2}}) = \mathcal{O}(|C|^2)$ variables. And finally, (8) adds a variable for each combination of conflicting operations, for each ordering of indices, which yields $\mathcal{O}(\binom{|T|}{2} \binom{|C|}{2}) = \mathcal{O}(|T|^2 |C|^2)$ variables. In total, this gives $\mathcal{O}(|C| + |C|^2 + |T|^2 |C|^2) = \mathcal{O}(|T|^2 \cdot |C|^2)$ variables. The number of clauses is analogous to the number of variables. $\square$

Path Encoding

To get the length of the longest paths in the precedence graph, we add the transitive closure over all possible precedence graphs to the CNF, denoted as the CNF formula $\text{Length}(T)$ for a set of transactions T .

For this to make sense, we have to cut off this process at a given maximal length as cycles in the graph can cause infinite loops to occur. Hence, we fix the maximal length ω to be the sum of all the costs of all the arcs in $\text{comp}'(P_s)$ together with the number of cross-transaction arcs that each compressed precedence graph has. For T in Example 3, ω would be equal to 9.

We add the following clauses recursively, as long as $w_1 + w_2 \leq \omega$:

$$\neg A_{o_1,o_2,w_1} \vee \neg A_{o_2,o_3,w_2} \vee A_{o_1,o_3,w_1+w_2} \quad (\text{A.9})$$

Finally, the only clause we still need to encode the decision variant $D(\text{TxSCHED})$ is the following, given a k , we add a clause for each $i \in \{k, k + 1, \dots, \omega\}$:

$$\neg A_{\top, \perp, i} \quad (\text{A.10})$$

To compute the optimisation variant (that is, TxSCHED), we use clauses (1) through (9) as the base CNF, and repeatedly call a SAT solver using (10) as an assumption, and letting k range from 1 to ω .

2 Transaction Scheduling as Mixed ILP

We now encode transaction scheduling into a Mixed ILP instance that can be passed to the JuMP solver.

Transaction Sets

We start by representing a transaction set using the following data structure.

```
struct TransactionSet
    # order-3 tensor containing all
    # read operations (transaction × operation × object)
    reads::BitArray{3}
    # order-3 tensor containing all
    # write operations (transaction × operation × object)
    writes::BitArray{3}
    # captures the indices of the final operation
    # of each transaction
    finals::Vector{Int64}
    # captures the conflicts for each transaction
    # i.e.,  $j \Rightarrow \{ [i, k], \dots \}$ 
    # where  $j$  is a transaction,  $i$  is an operation and  $k$  an object
    conflicts::Dict{Int64, Set{Tuple{Int64, Int64}}}}
    # map object identifiers to user-provided objects
    oid2obj::Vector{Any}
    # map user-provided objects to object identifiers
    obj2oid::Dict{Any, Int64}
    # number of distinct objects referenced by all transactions
    k::Int64
    # max number of operations in any transaction
    l::Int64
    # number of transactions
    n::Int64

    # [ ... methods are yet to be declared ...]
end
```

In order to construct one of these transaction sets, we provide a convenience function that takes in a list of transactions of the following form:

```
T = [[(:r 1) (:w 1) (:r 3) (:w 3)] [(:r 4) (:w 4) (:r 2) (:w 2)]]
```

We can then convert it into a `TransactionSet` with the following constructor function. This function is defined within the scope of the earlier defined struct.

```
function TransactionSet(transactions)
    n = length(transactions)
    finals = length(transactions)
    l = maximum(finals)

    objs = sort(unique(obj for tx in transactions for (_, obj) in tx))
    k = length(objs)

    obj2oid = Dict{obj => i for (i, obj) in enumerate(objs)}
    oid2obj = collect(objs)

    reads = falses(n, l, k)
    writes = falses(n, l, k)

    op_type = Dict(
        :read => :read, :r => :read, :R => :read,
        :write => :write, :w => :write, :W => :write
    )

    for (j, tx) in enumerate(transactions)
        for (i, (op, obj)) in enumerate(tx)
            if op_type[op] == :read
                reads[j, i, obj2oid[obj]] = 1
            elseif op_type[op] == :write
                writes[j, i, obj2oid[obj]] = 1
            else
                throw(DomainError(op, """
                    TransactionSet(..) was called with a transaction
                    containing an unrecognized operation type. Pick
                    operations with types such as `:r`, `:read`, `:R`,
                    `:w`, `:write` or `:W`.
                    """))
            end
        end
    end

    conflicts = _conflicts(reads, writes, n, l, k)

    new(reads, writes, finals, conflicts, oid2obj, obj2oid, k, l, n)
end
```

The above function makes use of a helper function `_conflicts` which finds which operations in the transaction set are conflicting

with one another. This is something we define next. We recall that two operations are conflicting if: (1) they are from different transactions, (2) they act on the same object, and (3) at least one of the operations is a write. Given that the read and write set are given to this function as order-3 tensors, we also call attention to what each dimension represents. The first dimension gives the object identifiers, the second dimension indicates the operation, and the third dimension indicates the transaction.

The way the algorithm operates, in general terms, is to first identify the objects that are conflicting objects, and then filter out the indices which correspond to operations on conflicting objects. The results are dictionary mappings that correspond transaction indices with sets of pairs of operation and object indices.

```
function _conflicts(rs, ws, n, l, k)
    # results are j => {(i1, k1), (i2, k2), ... } where:
    # - j is the transaction index,
    # - i1, i2 are conflicting operation indices for j
    # - k1, k2 are conflicting object indices for j
    res = Dict{Int64, Set{Tuple{Int64, Int64}}}()
    # The 'w' matrix is the addition of all write operations
    # of all transactions into a single matrix (order-2 tensor)
    # the rows of this matrix that sum up to a value >= 2 are
    # write-write conflicting objects.
    w = reshape(sum(ws, dims=1), l, k)
    # this function extracts all read-write/write-read conflicts
    # given two transaction indices j1 and j2 such that j1 < j2.
    # since these conflicts are symmetric we do not need to
    # check (j2, j1) conflicts if we already check (j1, j2), hence
    # why j1 < j2 is sufficient.
    function _conflict_objects(j1, j2)
        # maximum of 'w + reads of i1' and 'w + reads of j2' gives
        # a matrix where either j1 had a rw/wr conflict or j2
        # had a rw/wr conflict. We do not want to consider
        # read-read pairings as these are not conflicts.
        # We then sum all rows (i.e., dimension=1) to get
        # a single row.
        c = sum(max.(w + rs[j1, :, :], w + rs[j2, :, :]),
                dims = 1)

        # We then find all indices in that vector that sum up to
        # at least 2. The previous operation returns
        # CartesianIndices since it is done directly on a matrix,
        # so we just get the index from the first dimension.
        map(idx -> idx[2], findall(c .>= 2))
    end

    function _obj2opindex(j, k)
        # find all the operation indices i which contain
        # an operation on a conflicting object index k
        # in transaction j

```

```

        op_idx = findall(max.(rs[j, :, k], ws[j, :, k]) .== 1)

        # neatly package the indices as a tuple instead of a
        CartesianIndex
        map(i -> (i, k), op_idx)
    end

    # Loop over all pairs of distinct transactions such
    # that j1 < j2 and consider all their conflicting
    # object indices k
    for j1 = 1:n, j2 = j1+1:n, k = _conflict_objects(j1, j2)
        # add conflicting operations of j1 and j2 to res
        res[j1] = get(res, j1, union(
            Set{Tuple{Int64, Int64}}{()},
            _obj2opindex(j1, k)
        ))
        res[j2] = get(res, j2, union(
            Set{Tuple{Int64, Int64}}{()},
            _obj2opindex(j2, k)
        ))
    end

    res
end
end

```

Schedules and Constraint Encoding

Schedules are an data type for representing our solution. We keep track of some extra information besides just the schedule, such as the solving time and the resulting makespan.

```

struct Schedule
    serial_order::Vector{Int64}
    schedule::Matrix{Int64}
    makespan::Int64
    solver_timing_secs::Number
end

```

Finally, we can now give the encoding of the problem into Mixed ILP. The below syntax makes use of JuMP's macro-based DSL for specifying the program.

```

function optschedule(transactions::TransactionSet)::Schedule
    model = Model(HiGHS.Optimizer)
    n = transactions.n
    l = transactions.l
    finals = transactions.finals
    conflicts = transactions.conflicts

    # Precedes(transaction, transaction)
    @variable(model, precedes[1:n, 1:n], Bin)

```

```

# SerialOrder(transaction)
@variable(model, 1 <= serial_order[1:n] <= n, Int)
@constraint(model, [j = 1:n], precedes[j, j] == 0)

for j1 = 1:n, j2 = [1:(j1-1); (j1+1):n]
    @constraint(model, precedes[j1, j2] + precedes[j2, j1] == 1)
    @constraint(model, precedes[j1, j2] --> {
        serial_order[j1] + 1 <= serial_order[j2]
    })
end

# Schedule(transaction, operation)
@variable(model, 0 <= schedule[1:n, 1:1] ≤ n * 1 - 1, Int)

# operations from the same transaction should follow
# that transaction's order of operations
@constraint(model, [j = 1:n, i = 1:(finals[j] - 1)],
    schedule[j, i] + 1 <= schedule[j, i+1])

# conflicting operations from different transactions
# should follow the serial ordering of transactions
for j1 = 1:n,
    j2 = [(1:j1-1); (j1+1):n],
    (i1, k1) in get(conflicts, j1, Set{Tuple{Int64, Int64}}{}()),
    (i2, k2) in get(conflicts, j2, Set{Tuple{Int64, Int64}}{}())
    if k1 == k2
        @constraint(model, precedes[j1, j2] --> {
            schedule[j1, i1] + 1 <= schedule[j2, i2]
        })
    end
end

# Objective: minimize the makespan
@variable(model, 1 ≤ makespan <= n * 1, Int)

@constraint(
    model,
    [j = 1:n],
    schedule[j, finals[j]] + 1 <= makespan
)

@objective(model, Min, makespan)

set_silent(model)
optimize!(model)

res_serial_order = round.(Int64, value(serial_order))
res_schedule = round.(Int64, value(schedule))
res_makespan = round(Int64, value(makespan))

return Schedule(res_serial_order, res_schedule, res_makespan,
    solve_time(model))
end

```

Bibliography

1. Abadi, D. J. & Faleiro, J. M. An overview of deterministic database systems. *Commun. ACM* **61**, 78–88 (2018).
2. Abiteboul, S., Antoine, É., Miklau, G., Stoyanovich, J. & Testard, J. *Rule-based application development using Webdamlog* in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22–27, 2013* (eds Ross, K. A., Srivastava, D. & Papadias, D.) (ACM, 2013), 965–968. doi:10.1145/2463676.2465251.
3. Aken, D. V., Difallah, D. E., Pavlo, A., Curino, C. & Cudré-Mauroux, P. *BenchPress: Dynamic Workload Control in the OLTP-Bench Testbed* in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (eds Sellis, T. K., Davidson, S. B. & Ives, Z. G.) (ACM, 2015), 1069–1073. doi:10.1145/2723372.2735354.
4. Alomari, M., Cahill, M. J., Fekete, A. D. & Röhm, U. *The Cost of Serializability on Platforms That Use Snapshot Isolation* in *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7–12, 2008, Cancún, Mexico* (eds Alonso, G., Blakeley, J. A. & Chen, A. L. P.) (IEEE Computer Society, 2008), 576–585. doi:10.1109/ICDE.2008.4497466.
5. Alvaro, P., Conway, N., Hellerstein, J. M. & Marczak, W. R. *Consistency Analysis in Bloom: a CALM and Collected Approach* in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR 2011), January 9–12, 2011, Asilomar, CA, USA* (CIDR Conference, Online, 2011), 249–260.
6. Alvaro, P. *et al.* *Dedalus: Datalog in Time and Space* in *Proceedings of Datalog Reloaded: 1st International Workshop (Datalog 2010), March 16–19, 2010, Oxford, UK* (Springer, Berlin, Germany, 2010), 262–281.

7. Ameloot, T. J., Ketsman, B., Neven, F. & Zinn, D. Weaker Forms of Monotonicity for Declarative Networking: A More Fine-Grained Answer to the CALM-Conjecture. *ACM Trans. Database Syst.* **40**, 21:1–21:45. doi:10 . 1145 / 2809784 (2016).
8. Ameloot, T. J., Neven, F. & den Bussche, J. V. Relational transducers for declarative networking. *J. ACM* **60**, 15:1–15:38. doi:10 .1145/2450142 .2450151 (2013).
9. An, S., Cao, Y. & Zhao, W. *Competitive Consistent Caching for Transactions* in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022* (IEEE, 2022), 2154–2167.
10. Angluin, D. *Local and Global Properties in Networks of Processors* in *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC 1980), April 28-30, 1980, Los Angeles, CA, USA* (ACM, New York, NY, USA, 1980), 82–93. doi:10 .1145 / 800141 .804655.
11. Baccaert, T. & Ketsman, B. *Distributed Consistency Beyond Queries* in *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA , June 18-23, 2023* (eds Geerts, F., Ngo, H. Q. & Sintos, S.) (ACM, 2023), 47–58.
12. Baccaert, T. & Ketsman, B. *Cascade: Optimal Transaction Scheduling for High-Contention Workloads* in *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024* (2024), 5634–5638.
13. Baccaert, T. & Ketsman, B. A Generalized CALM Theorem for Non-Deterministic Computation in Asynchronous Distributed Systems. *Information Systems* **138**, 102691 (2026).
14. Baccaert, T., Vandevoort, B. & Ketsman, B. *Bounding the Makespan of Transaction Schedules* in *29th International Conference on Database Theory, ICDT 2026, Tampere, Finland, March 24-27, 2026* (eds ten Cate, B. & Funk, M.) (Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2026), 10:1–10:20.
15. Bailis, P. *et al.* Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* **7**, 181–192. doi:10 .14778 / 2732232 .2732237 (2013).
16. Balakrishnan, M. *et al.* *Tango: distributed data structures over a shared log* in *ACM SIGOPS 24th Symposium on Operating Systems*

- Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (eds Kaminsky, M. & Dahlin, M.) (ACM, 2013), 325–340.
17. Beame, P., Koutris, P. & Suciu, D. Communication Steps for Parallel Query Processing. *Journal of the ACM* **64**, 40:1–40:58. doi:10.1145/3125644 (2017).
 18. Berenson, H. *et al.* A Critique of ANSI SQL Isolation Levels in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995* (eds Carey, M. J. & Schneider, D. A.) (ACM Press, 1995), 1–10. doi:10.1145/223784.223785.
 19. Berman, U. *The Birth of IMS/360* tech. rep. (Computer History Museum, 2007).
 20. Brucker, P. *Scheduling Algorithms* isbn: 978-3-540-20524-1 (Springer, 2004).
 21. Cao, Y., Fan, W., Ou, W., Xie, R. & Zhao, W. Transaction Scheduling: From Conflicts to Runtime Conflicts. *Proc. ACM Manag. Data* **1**, 26:1–26:26 (2023).
 22. Chandy, K. M. & Misra, J. How Processes Learn. *Distributed Comput.* **1**, 40–52 (1986).
 23. Chang, G. J., Tong, L., Yan, J. & Yeh, H. A Note on The Gallai-Roy-Vitaver Theorem. *Discrete Mathematics* **256**, 441–444. doi:10.1016/S0012-365X(02)00386-2 (2002).
 24. Cheng, A. *et al.* Towards Optimal Transaction Scheduling. *Proc. VLDB Endow.* **17**, 2694–2707. doi:10.14778/3681954.3681956 (2024).
 25. Choi, I., Michael, E., Li, Y., Ports, D. R. K. & Li, J. *Hydra: Serialization-Free Network Ordering for Strongly Consistent Distributed Applications* in *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023* (eds Balakrishnan, M. & Ghobadi, M.) (USENIX Association, 2023), 293–320.
 26. Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R. & Sears, R. *Benchmarking cloud serving systems with YCSB* in *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010* (eds Hellerstein, J. M., Chaudhuri, S. & Rosenblum, M.) (ACM, 2010), 143–154. doi:10.1145/1807128.1807152.

27. Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. *Introduction to Algorithms, 3rd Edition* isbn: 978-0-262-03384-8 (MIT Press, 2009).
28. Council, T. P. P. TPC Benchmark C Standard Specification Revision 5.2 (2006).
29. Crescenzi, P. *A Short Guide to Approximation Preserving Reductions* in *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity, Ulm, Germany, June 24-27, 1997* (IEEE Computer Society, 1997), 262–273. doi:10.1109/CCC.1997.612321.
30. Dennard, R. *et al.* Design of Ion-implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits* **9**, 256–268. doi:10.1109/JSSC.1974.1050511 (1974).
31. Difallah, D. E., Pavlo, A., Curino, C. & Cudré-Mauroux, P. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* **7**, 277–288. doi:10.14778/2732240.2732246 (2013).
32. Dong, Z. *et al.* Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *Proc. VLDB Endow.* **16**, 1930–1943 (2023).
33. Fagin, R., Halpern, J. Y., Moses, Y. & Vardi, M. Y. *Reasoning About Knowledge* (MIT Press, 1995).
34. Faleiro, J. M., Abadi, D. & Hellerstein, J. M. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* **10**, 613–624 (2017).
35. Faleiro, J. M. & Abadi, D. J. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.* **8**, 1190–1201 (2015).
36. Faleiro, J. M., Thomson, A. & Abadi, D. J. *Lazy evaluation of transactions in database systems* in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014* (eds Dyreson, C. E., Li, F. & Özsu, M. T.) (ACM, 2014), 15–26.
37. Fekete, A. D. *Allocating isolation levels to transactions* in *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA* (ed Li, C.) (ACM, 2005), 206–215. doi:10.1145/1065167.1065193.

38. Fischer, M. J., Lynch, N. A. & Paterson, M. *Impossibility of Distributed Consensus with One Faulty Process* in *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA* (eds Fagin, R. & Bernstein, P. A.) (ACM, 1983), 1–7. doi:10.1145/588058.588060.
39. Fishkin, A. V., Jansen, K. & Mastrolilli, M. *On Minimizing Average Weighted Completion Time: A PTAS for the Job Shop Problem with Release Dates* in *Algorithms and Computation, 14th International Symposium, ISAAC 2003, Kyoto, Japan, December 15-17, 2003, Proceedings* (eds Ibaraki, T., Katoh, N. & Ono, H.) **2906** (Springer, 2003), 319–328. doi:10.1007/978-3-540-24587-2_34.
40. Fishkin, A. V., Jansen, K. & Mastrolilli, M. Grouping Techniques for Scheduling Problems: Simpler and Faster. *Algorithmica* **51**, 183–199. doi:10.1007/s00453-007-9086-6 (2008).
41. Gross, D., Shortle, J. F., Thompson, J. M. & Harris, C. M. *Fundamentals of Queueing Theory, Fourth Edition* doi:10.1002/9781118625651 (Wiley, 2008).
42. Halmos, P. R. *Naive Set Theory* isbn: 0-387-90092-6 (Springer-Verlag, New York, 1974).
43. Hellerstein, J. M. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *SIGMOD Rec.* **39**, 5–19 (2010).
44. Hellerstein, J. M. & Alvaro, P. Keeping CALM: when distributed consistency is easy. *Communications of the ACM* **63**, 72–81. doi:10.1145/3369736 (2020).
45. Hildred, J., Abebe, M. & Daudjee, K. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proc. VLDB Endow.* **17**, 469–482 (2023).
46. Hromkovič, J. *Algorithmics for Hard Problems - Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics, Second Edition* isbn: 978-3-642-07909-2. doi:10.1007/978-3-662-05269-3 (Springer, 2004).
47. Huang, J., Mozafari, B., Schoenebeck, G. & Wenisch, T. F. A Top-Down Approach to Achieving Performance Predictability in Database Systems in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017*,

- Chicago, IL, USA, May 14-19, 2017 (eds Salihoglu, S., Zhou, W., Chirkova, R., Yang, J. & Suciu, D.) (ACM, 2017), 745–758.
48. Immerman, N. *Relational Queries Computable in Polynomial Time (Extended Abstract)* in *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA* (eds Lewis, H. R., Simons, B. B., Burkhard, W. A. & Landweber, L. H.) (ACM, 1982), 147–152. doi:10.1145/800070.802187.
 49. Jiménez-Peris, R., Patiño-Martínez, M. & Arévalo, S. *Deterministic Scheduling for Transactional Multithreaded Replicas* in *19th IEEE Symposium on Reliable Distributed Systems, SRDS'00, Nürnberg, Germany, October 16-18, 2000, Proceedings* (IEEE Computer Society, 2000), 164–173.
 50. Johnson, S. M. Optimal Two- and Three-stage Production Schedules with Setup Times Included. *Naval Research Logistics Quarterly* **1**, 61–68 (1954).
 51. Johnson, S. M. Generation of Permutations by Adjacent Transposition. *Mathematics of Computation* **17**, 282–285 (1963).
 52. Kallman, R. *et al.* H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* **1**, 1496–1499 (2008).
 53. Karp, R. M. *Reducibility Among Combinatorial Problems* in *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA* (eds Miller, R. E. & Thatcher, J. W.) (Plenum Press, New York, 1972), 85–103. doi:10.1007/978-1-4684-2001-2_9.
 54. Ketsman, B., Koch, C., Neven, F. & Vandevoort, B. Concurrency Control for Database Theorists. *SIGMOD Record* **51**, 6–17. doi:10.1145/3582302.3582304 (2022).
 55. Ketsman, B., Koch, C., Neven, F. & Vandevoort, B. Deciding Robustness for Lower SQL Isolation Levels. *ACM Transactions on Database Systems* **47**, 13:1–13:41. doi:10.1145/3561049 (2022).
 56. Lamport, L. Fast Paxos. *Distributed Comput.* **19**, 79–103. doi:10.1007/S00446-006-0005-X (2006).

57. Leighton, F. T., Maggs, B. M. & Richa, A. W. Fast Algorithms for Finding $O(\text{Congestion} + \text{Dilation})$ Packet Routing Schedules. *Comb.* **19**, 375–401. doi:10.1007/s004930050061 (1999).
58. Li, J., Michael, E. & Ports, D. R. K. *Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control* in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017* (ACM, 2017), 104–120.
59. Loo, B. T. *et al.* Declarative networking. *Commun. ACM* **52**, 87–95. doi:10.1145/1592761.1592785 (2009).
60. Lu, Y., Yu, X., Cao, L. & Madden, S. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* **13**, 2047–2060 (2020).
61. Lu, Y., Yu, X., Cao, L. & Madden, S. Epoch-based Commit and Replication in Distributed OLTP Databases. *Proc. VLDB Endow.* **14**, 743–756 (2021).
62. Lynch, N. A. *Distributed Algorithms* isbn: 1-55860-348-4 (Morgan Kaufmann, Burlington, MA, USA, 1996).
63. Mastrolilli, M. & Svensson, O. Hardness of Approximating Flow and Job Shop Scheduling Problems. *J. ACM* **58**, 20:1–20:32. doi:10.1145/2027216.2027218 (2011).
64. Maymounkov, P. & Mazières, D. *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric* in *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers* (eds Druschel, P., Kaashoek, M. F. & Rowstron, A. I. T.) **2429** (Springer, 2002), 53–65.
65. Mu, S., Cui, Y., Zhang, Y., Lloyd, W. & Li, J. *Extracting More Concurrency from Distributed Transactions* in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014* (eds Flinn, J. & Levy, H.) (USENIX Association, 2014), 479–494.
66. Narasimhan, P., Moser, L. E. & Melliar-Smith, P. M. *Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications* in *The Eighteenth Symposium on Reliable Distributed Systems, SRDS 1999, Lausanne, Switzerland, October 19-22, 1999, Proceedings* (IEEE Computer Society, 1999), 263–273.

67. Narula, N., Cutler, C., Kohler, E. & Morris, R. T. *Phase Reconciliation for Contended In-Memory Transactions* in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014* (eds Flinn, J. & Levy, H.) (USENIX Association, 2014), 511–524.
68. Nash, A. & Ludäscher, B. *Processing First-Order Queries under Limited Access Patterns* in *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France* (ACM, New York, NY, USA, 2004), 307–318. doi:10.1145/1055558.1055601.
69. Oki, B. M. & Liskov, B. *Viewstamped Replication: A General Primary Copy* in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988* (ed Dolev, D.) (ACM, 1988), 8–17. doi:10.1145/62546.62549.
70. Ongaro, D. & Ousterhout, J. K. *In Search of an Understandable Consensus Algorithm* in *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014* (eds Gibson, G. & Zeldovich, N.) (USENIX Association, 2014), 305–319.
71. Papadimitriou, C. H. The Serializability of Concurrent Database Updates. *J. ACM* **26**, 631–653. doi:10.1145/322154.322158 (1979).
72. Papadimitriou, C. H. *The Theory of Database Concurrency Control* isbn: 0-88175-027-1 (Computer Science Press, 1986).
73. Pinedo, M. L. *Scheduling: Theory, Algorithms, and Systems* isbn: 978-3-031-05923-0 (Springer, 2023).
74. Power, C., Koutris, P. & Hellerstein, J. M. *The Free Termination Property of Queries over Time* in *28th International Conference on Database Theory, ICDT 2025, March 25-28, 2025, Barcelona, Spain* (eds Roy, S. & Kara, A.) **328** (2025), 32:1–32:22.
75. Prasaad, G., Cheung, A. & Suciu, D. *Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning* in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020* (eds Maier, D. et al.) (ACM, 2020), 527–542.
76. Qadah, T. M. & Sadoghi, M. *QueCC: A Queue-oriented, Control-free Concurrency Architecture* in *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France,*

- December 10-14, 2018 (eds Ferreira, P. & Shrira, L.) (ACM, 2018), 13–25.
77. Qu, L. *et al.* *Application-Oriented Workload Generation for Transactional Database Performance Evaluation* in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022* (IEEE, 2022), 420–432. doi:10.1109/ICDE53745.2022.00036.
 78. Ren, K., Thomson, A. & Abadi, D. J. Lightweight Locking for Main Memory Database Systems. *Proc. VLDB Endow.* **6**, 145–156 (2012).
 79. Ren, K., Thomson, A. & Abadi, D. J. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *Proc. VLDB Endow.* **7**, 821–832 (2014).
 80. Ren, K., Thomson, A. & Abadi, D. J. VLL: a lock manager redesign for main memory database systems. *VLDB J.* **24**, 681–705 (2015).
 81. Sipser, M. *Introduction to the Theory of Computation* isbn: 978-0-534-94728-6 (PWS Publishing Company, 1997).
 82. Sotskov, Y. N. & Shakhlevich, N. V. NP-hardness of Shop-scheduling Problems with Three Jobs. *Discret. Appl. Math.* **59**, 237–266. doi:10.1016/0166-218X(95)80004-N (1995).
 83. Steinhaus, H. *One hundred problems in elementary mathematics* 174 (Basic Books, Inc., Publishers, New York, 1964).
 84. Stockmeyer, L. J. Planar 3-colorability is polynomial complete. *SIGACT News* **5**, 19–25. doi:10.1145/1008293.1008294 (1973).
 85. Thomson, A. & Abadi, D. J. The Case for Determinism in Database Systems. *Proc. VLDB Endow.* **3**, 70–80 (2010).
 86. Thomson, A. *et al.* *Calvin: fast distributed transactions for partitioned database systems* in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012* (eds Candan, K. S., Chen, Y., Snodgrass, R. T., Gravano, L. & Fuxman, A.) (ACM, 2012), 1–12.
 87. Tian, B., Huang, J., Mozafari, B. & Schoenebeck, G. Contention-Aware Lock Scheduling for Transactional Databases. *Proc. VLDB Endow.* **11**, 648–662 (2018).
 88. Trotter, H. F. Algorithm 115: Perm. *Commun. ACM* **5**, 434–435. doi:10.1145/368637.368660 (1962).

89. Vandevoort, B. *Optimizing Concurrency Control: Robustness Against Read Committed Revisited* PhD thesis (Hasselt University, Belgium, 2021).
90. Vandevoort, B., Ketsman, B., Koch, C. & Neven, F. Robustness against Read Committed for Transaction Templates with Functional Constraints. *Logical Methods in Computer Science* **19**. doi:10.46298/LMCS-19(4:39)2023 (2023).
91. Vandevoort, B., Ketsman, B. & Neven, F. *Allocating Isolation Levels to Transactions in a Multiversion Setting* in *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023* (eds Geerts, F., Ngo, H. Q. & Sintos, S.) (ACM, 2023), 69–78. doi:10.1145/3584372.3588672.
92. Vandevoort, B., Ketsman, B. & Neven, F. Allocating Isolation Levels to Transactions in a Multiversion Setting. *ACM Transactions on Database Systems* **50**, 8:1–8:24. doi:10.1145/3716374 (2025).
93. Vardi, M. Y. *The Complexity of Relational Query Languages (Extended Abstract)* in *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA* (eds Lewis, H. R., Simons, B. B., Burkhard, W. A. & Landweber, L. H.) (ACM, 1982), 137–146. doi:10.1145/800070.802186.
94. Wang, D., Cai, P., Qian, W. & Zhou, A. *Predictive Transaction Scheduling for Alleviating Lock Thrashing in Database Systems for Advanced Applications - 25th International Conference, DAS-FAA 2020, Jeju, South Korea, September 24-27, 2020, Proceedings, Part I* (eds Nah, Y. et al.) **12112** (Springer, 2020), 139–156.
95. Weikum, G. & Vossen, G. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery* isbn: 1-55860-508-8 (Morgan Kaufmann, 2002).
96. Weng, S. et al. Lauca: A Workload Duplicator for Benchmarking Transactional Database Performance. *IEEE Trans. Knowl. Data Eng.* **36**, 3180–3194. doi:10.1109/TKDE.2024.3360116 (2024).
97. Yao, C. et al. Exploiting Single-Threaded Model in Multi-Core In-Memory Systems. *IEEE Trans. Knowl. Data Eng.* **28**, 2635–2650 (2016).

98. Yu, X., Bezerra, G., Pavlo, A., Devadas, S. & Stonebraker, M. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* **8**, 209–220 (2014).
99. Zhang, H. *et al.* Dike: A Benchmark Suite for Distributed Transactional Databases in Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023 (eds Das, S., Pandis, I., Candan, K. S. & Amer-Yahia, S.) (ACM, 2023), 95–98. doi:10.1145/3555041.3589710.
100. Zhang, I., Sharma, N. K., Szekeres, A., Krishnamurthy, A. & Ports, D. R. K. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* **35**, 12:1–12:37 (2017).
101. Zheng, Q. *et al.* Lion: Minimizing Distributed Transactions Through Adaptive Replica Provision in 40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024 (IEEE, 2024), 2012–2025.
102. Zhou, J., Bruno, N. & Lin, W. Advanced partitioning techniques for massively distributed computation in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012 (eds Candan, K. S., Chen, Y., Snodgrass, R. T., Gravano, L. & Fuxman, A.) (ACM, New York, NY, USA, 2012), 13–24.
103. Zhou, X., Yu, X., Graefe, G. & Stonebraker, M. Lotus: Scalable Multi-Partition Transactions on Single-Threaded Partitioned Databases. *Proc. VLDB Endow.* **15**, 2939–2952 (2022).
104. Zinn, D., Green, T. J. & Ludäscher, B. Win-move is coordination-free (sometimes) in 15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012 (ed Deutsch, A.) (ACM, 2012), 99–113. doi:10.1145/2274576.2274588.