2013 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

DOCTORAATSPROEFSCHRIFT

Declarative Networking: Models and Conjectures

Proefschrift voorgelegd tot het behalen van de graad van doctor in wetenschappen, informatica, te verdedigen door:

Tom Ameloot

Promotor: prof. dr. Jan Van den Bussche
Copromotor: prof. dr. Frank Neven

D/2013/2451/15

Maastricht University
universiteit hasselt
KNOWLEDGE IN ACTION

Acknowledgments

I thank my supervisor Jan Van den Bussche and my co-supervisor Frank Neven for introducing me to the topics of cloud computing and declarative networking, at a time when these were hot topics and new interesting conjectures were posed to the research community.

I like to express special thanks to Jan for the many inspiring discussions, and for the valuable insights that resulted from them. I additionally thank Jan for giving me much advice about writing good technical texts and for patiently reading my many proofs. Jan has also shared with me his interest in nature, and birds in particular, that I have found very inspiring.

I thank my family for their continuous support and general advice.

I thank my fellow PhD students for the things I have learned from them and the conversations we have had. I also thank my colleagues in general for creating a friendly atmosphere, like organizing celebrations for a paper-acceptance.

I thank the Research Foundation Flanders (FWO) for providing this opportunity for doing a PhD thesis.

Nederlandstalige Samenvatting (Dutch Summary)

Deze thesis gaat over *declarative networking*, een onderzoeksgebied waarin men gedistribueerde programma's implementeert met beschrijvende talen zoals *Datalog* [2], waarin men voornamelijk specificeert *wat* de uitvoer is, en niet zozeer *hoe* deze uitvoer moet berekend worden. Het voordeel van zulke talen is dat complexe algoritmen kunnen beschreven worden met redelijk korte programma's [45, 37]. Ter vergelijking, om gegevensbanken te ondervragen gebruikt men standaard de beschrijvende taal *SQL* in plaats van een imperatieve programmeertaal zoals *Java*, omdat men voor deze toepassing vaak veel kortere en overzichtelijkere programma's kan schrijven in SQL.

In deze thesis gebruiken we twee bestaande modellen. Het eerste model bestaat uit *relational transducers* [7, 28, 29, 30, 54]; deze stellen een abstracte computer voor. Wij definiëren zelf netwerken van zulke relational transducers. Het tweede model wordt gevormd door de taal *Dedalus* [13, 14, 37]; dit is een programmeertaal geïnspireerd door Datalog. Wij veronderstellen in beide modellen dat een netwerk willekeurige vertragingen kan veroorzaken tussen het versturen en aankomen van boodschappen; dit is de zogenaamde asynchrone communicatie zoals typisch voorkomt op het Internet.

Hoofdstuk 1 is de Engelstalige inleiding, en Hoofdstuk 2 herhaalt de nodige voorkennis, waaronder de hierboven genoemde modellen.

In Hoofdstuk 3 onderzoeken we de *CALM conjecture*, een probleemstelling geformuleerd door Hellerstein [36, 37]: men stelt de vraag of er een verband is tussen enerzijds de uitdrukkingskracht van gedistribueerde programma's en anderzijds hun vermogen om berekeningen uit te voeren zonder teveel boodschappen te versturen (coördinatie). We onderzoeken deze probleemstelling in een model met relational transducers. Ons belangrijkste resultaat is dat gedistribueerde programma's die geen coördinatie uitvoeren beperkt zijn tot monotone berekeningen, d.w.z. als de invoer groeit dan groeit de uitvoer van de berekening. We onderzoeken ook bijkomstig de uitdrukkingskracht van variaties in het gebruikte model.

In Hoofdstuk 4 onderzoeken we of het mogelijk is om via een computeralgoritme automatisch te beslissen of een gegeven gedistribueerd programma, voor elke invoer afzonderlijk, altijd dezelfde uitvoer zal geven. Indien het gedistribueerd programma deze eigenschap heeft, zal het dus vertragingen van boodschappen kunnen tolereren die door het netwerk worden veroorzaakt. We onderzoeken deze probleemstelling in een model met relational transducers. Ons belangrijkste resultaat is dat we dit inderdaad kunnen beslissen, mits we voldoende syntactische beperkingen opleggen aan zulke gedistribueerde programma's. Als bijkomend resultaat tonen we ook aan dat er nog steeds nuttige berekeningen kunnen uitgevoerd worden onder deze beperkingen.

In Hoofdstuk 5 tonen we aan dat het mogelijk is om de werking van een (gedistribueerd) Dedalus programma volledig te beschrijven via een declaratieve semantiek, d.w.z. we kunnen met formele regels modellen beschrijven die de volledige werking samenvatten. Dit kan een interessante en alternatieve kijk bieden op de gedistribueerde werking van een Dedalus programma.

In Hoofdstuk 6 onderzoeken we de *CRON conjecture*, een tweede probleemstelling geformuleerd door Hellerstein [36, 37]: men stelt de vraag of er een verband is tussen enerzijds de volgorde waarin boodschappen mogen afgeleverd worden aan een gedistribueerd programma en anderzijds de aard van de berekeningen (monotoon versus niet-monotoon) waarin die boodschappen worden gebruikt. We onderzoeken deze probleemstelling voor de taal Dedalus. Ons belangrijkste resultaat is dat de volgorde inderdaad niet van belang is bij programma's waarin geen negatieve testen kunnen uitgevoerd worden, waardoor deze programma's beperkt zijn tot monotone berekeningen. Zulke programma's blijven, voor elke input afzonderlijk, dezelfde uitvoer produceren zelfs als de volgorde van de boodschappen volledig door elkaar gehaald wordt (door het netwerk of wegens een andere oorzaak).

Abstract

Declarative networking is a field in which people implement distributed protocols and applications in high-level declarative languages like Datalog. Such a declarative formalism is believed to help the programmer express complex distributed computations with relatively few lines of code. In this thesis, we investigate many different aspects of declarative networking. The model of networked relational transducers is one main topic, for which we investigate the expressivity; the link between expressivity and distributed coordination; and, the decidability of eventual consistency. Another important topic consists of the language Dedalus, a Datalog-inspired language, for which we define a purely declarative semantics. We also show how this semantics can be used to reason about distributed message causality.

Contents

Chapter 1

Introduction

This work is situated in the field of *declarative networking* [45, 36, 37], in which people describe and implement distributed protocols and algorithms in high-level declarative languages. Such formalisms reduce the technical details exposed to the programmer, so he can focus on just the distributed algorithm. Interesting questions can be asked, such as how suitable a formalism is for expressing certain problems, what runtime properties are exhibited, and how easy it is to automatically verify semantical properties.

We consider two existing models, around which we develop several results (see below). The first model is the *relational transducer* model, introduced by Abiteboul and Vianu, that is well established in database theory research as a model for data-centric agents reacting to inputs [7, 28, 29, 30, 54]. Relational transducers are firmly grounded in the theory of database queries [5, 6] and also have close connections with Abstract State Machines [25]. It thus seems natural to consider networks of relational transducers, as we will do.

Declarative networking initially developed around Datalog [45]. For this reason, our second model (or language) is *Dedalus*, which is one of the latest Datalog-inspired languages proposed in declarative networking [13, 14, 37], and has influenced other recent language designs for distributed and cloud computing such as Webdamlog [1] and Bloom [12]. In recent years we are also seeing a more general resurgence of interest in Datalog, e.g., [27, 38]. A notable promising property of Datalog is that it allows complex distributed algorithms and protocols to be expressed in relatively few lines of code [39, 37].

A major hurdle in declarative networking is the nondeterminism inherent to such systems. This nondeterminism is typically due to the asynchronous communication between the nodes of a network. Accordingly, one of the challenges is to design distributed programs so that the same outputs can eventually be produced on the same inputs, no matter how messages between nodes have been delayed or received in different orders. When a program has this property, we say it is "eventually consistent" [57, 36, 37, 12]. This is actually a fuzzy term, and in our results we will use two distinct interpretations of this notion: *(i)* all infinite (fair) computation traces yield the same output, simply called *consistency*; and, *(ii)* every finite computation trace can be extended to yield the outputs obtained in other finite computation traces, called *confluence*. We will always make clear which interpretation is used.

We now present the obtained results, divided in four main chapters. First, we mention that Chapter 2 gives preliminaries on standard database notions and also formalizes the two mentioned models for declarative networking. Chapter 7 is the conclusion.

1.1 Relational Transducers for Declarative Networking

In his keynote speech at PODS 2010 [36, 37], Hellerstein made a number of intriguing conjectures concerning the expressiveness of declarative networking. In particular, the CALM conjecture (Consistency And Logical Monotonicity) suggests a strong link between, on the one hand, "eventually consistent" and "coordination-free" distributed computations, and on the other hand, expressibility in monotonic Datalog (without negation or aggregate functions). The conjecture was not fully formalized, however; indeed, as Hellerstein notes himself, a proper treatment of this conjecture requires crisp definitions of eventual consistency and coordination, which have been lacking so far. Moreover, it also requires a formal model of distributed computation.

We formally investigate the CALM conjecture in Chapter 3. First, in order to address the expressiveness issues raised by Hellerstein, we present a model for distributed database querying based on networks of relational transducers and a formalization of "eventual consistency" for such networks.

It is less clear, however, how to formalize the intuitive notion of "coordination". We do not claim to resolve this issue definitively, but we propose a new, non-obvious definition that appears workable. Distributed algorithms requiring coordination are viewed as less efficient than coordination-free algorithms. Hellerstein has identified *monotonicity* as a fundamental property connected with coordination-freeness. Indeed, monotonicity enables "embarrassing parallelism" [37]: agents working on parts of the data in parallel can produce parts of the output independently, without the need for coordination.

One side of the CALM conjecture now states that any database query expressible in monotonic Datalog can be computed in a distributed setting in an eventually consistent, coordination-free manner. This is the easy side of the conjecture, and indeed we formally confirm it in the following broader sense: any monotone query can be computed by a network of "oblivious" transducers. Oblivious transducers are unaware of the network extent (in a sense that we will make precise), and every oblivious transducer network is coordination-free. Here, we should note that the transducer model is parameterized by the query language $\mathcal{L}$ that the transducer can use to update its local state. Formally, the monotone query to be computed must be expressible in the while-closure of $\mathcal{L}$ for the above confirmation to hold. If the query is defined in Datalog, for example, then $\mathcal{L}$ can just be unions of conjunctive queries.

The other side of the CALM conjecture, that the query computed by an eventually consistent, coordination-free distributed program is always expressible in Datalog, is false when taken literally, as we will point out. Nevertheless, we do give an extended version of the conjecture that holds. More importantly, we confirm the conjecture in the following more general form: coordination-free networks of transducers can compute only monotone queries. Note that here we are using our newly proposed formal definition of coordination-free.

In the last part of Chapter 3, we investigate the expressiveness of the model by considering different local query languages $\mathcal{L}$ to implement the transducers. From this investigation, we learn that the transducer model is quite natural, in the sense that it only adds a notion of iteration to each considered language $\mathcal{L}$.

1.2 Deciding Eventual Consistency

As mentioned above, it can be desirable to design systems that are "eventually consistent" [57, 36, 37, 12]. In Chapter 4, we investigate the decidability of this property. There, we will view eventual consistency as a confluence notion. On any fixed input, let J be the union of all outputs that can be produced during any possible execution of the distributed program. Then in our definition of eventual consistency, we require that for any two different outputs $J_1 \subseteq J$ and $J_2 \subseteq J$ resulting from two (partial) executions on the same input, the same output J can be produced in an extension of either partial execution. So, intuitively, the prior execution of the program will not prevent outputs from being produced if those outputs can be produced with another execution (on the same input).

The model used in Chapter 4 is again based on relational transducers, but we immediately limit attention to transducers whose functionality is implemented with unions of conjunctive queries with negation, henceforth referred to as "rules". Given the affinity between conjunctive queries and Datalog, we expect our results to apply also to other declarative networking formalisms, although we have not yet worked out these applications.

Our first main result is the identification of a number of syntactic restrictions on the rules used in the transducers, *not* so that eventual consistency always holds, but so that checking it becomes decidable. Informally, the restrictions comprise the following.

- The cluster must be recursion-free: the different rules among all local programs cannot be mutually recursive through positive subgoals. Recursive dependencies through negative subgoals are still allowed.

- The local programs must be inflationary: deletions from state relations are forbidden.

- The rules are message-positive: negation on message relations is forbidden.

- The state-update rules must satisfy a known restriction which we call "message-boundedness". This restriction is already established in the verification of relational transducers: it was first identified under the name "input-boundedness" by Spielmann [54] and was investigated further by Deutsch et al. [29, 30].

- Finally, the message-sending rules must be "static" in the sense that they may not depend on state relations; they can still depend on input relations and on received messages.

The last two restrictions are the most fundamental; in fact, even if just the last restriction is dropped and all the others are kept in place, the problem is already back to undecidable. The first three restrictions can probably be slightly relaxed without losing decidability, and indeed we just see our work as a step in the right direction. Eventual consistency is not an easy problem to analyze.

The second result of Chapter 4 is an analysis of the expressive power of clusters of relational transducers satisfying our above five restrictions; let us call such clusters "simple". Specifically, we show that simple clusters can compute *exactly* all distributed queries expressible by unions of conjunctive queries with negation, or equivalently, the existential fragment of first-order logic, without any further restrictions. So, this result shows that simple clusters form indeed a rather weak computational model, but not as weak as to be totally useless.

1.3 Declarative Semantics for Dedalus

It is well understood how an *operational* semantics for declarative networking can be defined formally [30, 48, 35]. Such a formal semantics is typically defined as a transition system. The transition system is infinite even if the distributed computation is working on a finite input database, because computing nodes can run indefinitely; moreover, they can keep on sending messages so that an unbounded number of messages can be floating around in the network. In addition, the transition system is highly nondeterministic, because nodes work concurrently, communication is asynchronous, and messages can be delayed and eventually be delivered out of order by the network.

On the other hand, it remains unclear how (and if) a purely *declarative* formal semantics can be given for the languages used in declarative (!) networking. This has been lacking so far, and the purpose of Chapter 5 is to contribute towards filling this gap, specifically for the language Dedalus. Concretely, the following approach is used.

1. First, in Chapter 2 we give a formal operational semantics for Dedalus. As mentioned above, this part is quite standard. Our definition leads to the notion of fair runs of a Dedalus program $\mathcal{P}$ on an input distributed database instance H. Runs represent distributed computations and, due to the nondeterminism mentioned above, there are typically many fair runs of $\mathcal{P}$ on H.

2. We continue in Chapter 5, where we note that each run respects a causal order (which is a partial order) that relates the local steps of the different compute nodes through chains of local steps and communicated messages. This order indicates what events "happened before" which other events [23]. Now, the computation of each run can be described by a structure which we call a trace, which includes for each compute node in the network the detailed information about the local steps it has performed and about the messages it has sent and received. The trace conforms to the causal order.

3. The main idea now is that the traces of runs can be obtained precisely as the set of *stable models* [32] of $\mathcal{P}$ on H. A few manipulations are needed before we can aim for such a result, however, because the Dedalus program $\mathcal{P}$ is not really a

Datalog$^\neg$ program.[1] Indeed, the language Dedalus provides special "inductive rules" and "asynchronous rules" that are used for respectively persisting memory across local computation steps and sending messages. First, we will transform these rules into Datalog$^\neg$ rules that simulate their effect, where asynchronous rules will nondeterministically choose the arrival times of messages [40, 52]. Furthermore, $\mathcal{P}$ is augmented with a fixed, input- and network-independent set of rules that express causality on the messages. Applying the stable model semantics to such transformed Dedalus programs constitutes the declarative semantics.

We believe that our result is interesting because it shows the equivalence between two quite different ways to define the semantics of a Dedalus program.

Perhaps most importantly, the result is of interest for grounding a representative database language for distributed and cloud computing in a well-motivated model-theoretic semantics. Indeed, our characterization provides a purely declarative axiomatization of fair distributed program behaviors in terms of the stable models of a logical theory (finite set of Datalog$^\neg$ rules). Specifically, we have succeeded in capturing in Datalog$^\neg$ the operational notion of causality, which focuses on just the key features of the distributed computation: the state of each compute node at each local time, and at what local times the nodes send and receive messages. Hence our declarative semantics reasons from the perspective of the local times of each node, which is a justified approach since there is no common "global clock" in a distributed environment [23].

It should be noted that the studied declarative semantics can perhaps not directly be used in practical applications, because in the semantics there is a reference to an infinite number of local computation steps of the nodes. But we hope our work can provide insights for the design of a more intuitive declarative semantics, used by developers of distributed applications. See also Chapter 7 for a discussion.

As mentioned, many Datalog-inspired languages have been proposed to implement distributed applications [45, 48, 35, 1], and they contain several features such as aggregation and non-determinism (choice), that result in powerful languages. But the essential features that all these languages possess, are reasoning about distributed state and representing message sending. We think of the language Dedalus, as we define it here, as a minimalistic extension of Datalog to provide just these essential features. For this reason, we expect that the current work can serve as a theoretical base that can be extended to more powerful language features as well.

1.4 The CRON Conjecture

In Chapter 6, we use the context of Dedalus to investigate a second conjecture by Hellerstein [36, 37]: the CRON conjecture (Causality Required Only for Non-monotonicity).

Causality stands for the physical constraint that an effect can only happen after its cause. Applied to message delivery, this intuitively means that a sent message can only be delivered in the future, not in the past. Now, the conjecture relates the causal delivery of messages to the nature of the computations that those messages

[1] Datalog$^\neg$ stands for "Datalog with negation".

participate in, like monotone versus non-monotone, and asks us to think about the cases where causality is really needed.

There seem to be interesting real-world applications of the CRON conjecture, one of which is crash recovery. During crash recovery, a program can read an old checkpointed state and a log of received messages, which is disjoint from that state. These messages could appear to come from the "future" when put side-by-side with the old state because according to the old state, those messages have yet to be sent. Then, it is not always clear how the program should combine the old state and the message log, certainly if negation and more generally non-monotone operations are involved. One can understand the CRON conjecture as saying that during recovery, for non-monotone operations, messages from the log should be read in causal order, like the order in which they are received, and they should not be exposed all at once.

From the other direction, if you know that only monotone operations are involved, the recovery could perhaps become more efficient by reading the messages all at once. This can be useful for the following reason. Distributed computations happen often in large clusters of compute nodes, where failure of nodes is not uncommon [58], and indeed distributed computing software should be robust against failures [23]. We want to avoid restarting entire computations when only a few nodes fail, and therefore it seems natural to use some lightweight crash recovery facility for individual nodes that can still make the computation succeed, although perhaps some partial results might have to be recomputed. The CRON conjecture could help us better understand how such recovery facilities can be designed.

In Chapter 6, we formally investigate the CRON conjecture. Continuing on the results of Chapter 5, it turns out that stable models [32] provide a way to reason about non-causality, and we use this to formalize the CRON conjecture. A strong interpretation of the conjecture posits that causality is not needed if and only if the query computed by a Dedalus program is monotone. Neither the "if" nor the "only if" direction holds, however, as we will demonstrate. Therefore we have turned attention to a more syntactic version of the conjecture, and there we indeed find that causal message ordering is not needed for positive Dedalus programs in order to compute meaningful results, if these programs already behave correctly in a causal operational semantics. This is the main result of Chapter 6.

Chapter 2

Preliminaries

2.1 Basic Database Notions

We first recall some basic notions from database theory [2]. A *database schema* $\mathcal{D}$ is a finite set of pairs (R, k) where R is a *relation name* and $k \in \mathbb{N}$ its associated *arity*. A relation name occurs at most once in a database schema. We often write a pair (R, k) as $R^{(k)}$. For the cases $k = 0$, $k = 1$, and $k = 2$, we often say relation R is respectively *nullary*, *unary*, and *binary*.

We assume some infinite universe **dom** of atomic data values. A *fact* $\boldsymbol{f}$ is a pair $(R, \bar{a})$, often denoted as $R(\bar{a})$, where R is a relation name and $\bar{a}$ is a tuple of values over **dom**. For a fact $R(\bar{a})$, we call R the *predicate*. We say that a fact $R(a_1, \ldots, a_k)$ is *over* database schema $\mathcal{D}$ if $R^{(k)} \in \mathcal{D}$. A database *instance* I over $\mathcal{D}$ is a set of facts over $\mathcal{D}$. For a subset $\mathcal{D}' \subseteq \mathcal{D}$, we write $I|_{\mathcal{D}'}$ to denote the subset of facts in I whose predicate is a relation name in $\mathcal{D}'$. The *active domain* of I, denoted $adom(I)$, is the set of data values occurring in facts of I. For a fact $\boldsymbol{f}$, we also write $adom(\boldsymbol{f})$ to denote the set of values occurring in $\boldsymbol{f}$. For a function $h : \mathbf{dom} \to \mathbf{dom}$, we define $h(I) = \{R(h(a_1), \ldots, h(a_k)) \mid R(a_1, \ldots, a_k) \in I\}$.

A *query* $\mathcal{Q}$ *over input database schema* $\mathcal{D}$ *and output database schema* $\mathcal{D}'$ is a partial function mapping database instances over $\mathcal{D}$ to database instances over $\mathcal{D}'$. A special but common kind of query are those where the output database schema contains just one relation. A query $\mathcal{Q}$ is called *generic* if for all input instances I and all permutations h of **dom**, the query $\mathcal{Q}$ is also defined on the isomorphic instance $h(I)$ and $\mathcal{Q}(h(I)) = h(\mathcal{Q}(I))$. We recall that a generic query $\mathcal{Q}$ is *domain-preserving*, in the sense that $adom(\mathcal{Q}(I)) \subseteq adom(I)$ for all input instances I. We use the word "query" in this text to mean generic query, unless explicitly specified otherwise.

2.2 Multisets

A *multiset* m over a universe $\mathcal{U}$ is a function that maps each element $e \in \mathcal{U}$ to a natural number $m(e)$ that represents the number of times that e occurs in m. The set operators $\cap$, $\cup$, and $\setminus$ can be defined for multisets in a natural way. For two multisets m_1 and m_2, we write $m_1 \sqsubseteq m_2$ to denote that $m_1(e) \leq m_2(e)$ for each $e \in \mathcal{U}$. For a multiset m, we define the set $set(m) = \{e \in \mathcal{U} \mid m(e) \geq 1\}$, i.e., the collapse of m to a set. The *size* of a multiset m is defined as $\sum_{e \in \mathcal{U}} m(e)$.

2.3 Network and Distributed Data

A *network* $\mathcal{N}$ is a finite, connected, and undirected graph whose nodes are all in **dom**, and where each edge is between two distinct nodes. We write $nodes(\mathcal{N})$ and $edges(\mathcal{N})$ to denote the nodes and edges of $\mathcal{N}$ respectively. For $x \in nodes(\mathcal{N})$, we write $neighbor(x, \mathcal{N})$ to denote the set $\{y \mid (x, y) \in edges(\mathcal{N})\}$. If $|nodes(\mathcal{N})| = 1$ then we call $\mathcal{N}$ a *single-node* network.

We now formalize how input data is distributed across a network. A *distributed database schema* $\mathcal{E}$ is a pair $(\mathcal{N}, \eta)$ where $\mathcal{N}$ is a network, and η is a function that maps each node x of $\mathcal{N}$ to an ordinary database schema. A *distributed database instance* H *over schema* $\mathcal{E}$ is a function that assigns to each node x of $\mathcal{N}$ an ordinary database instance $H(x)$ over the local schema $\eta(x)$.

2.4 Transducers

Our first model for declarative networking uses *relational transducers* [7, 15, 28, 29, 30, 54]. A relational transducer is an abstract computing device, formalized in this section. The complete network model is presented in Section 2.5.

A *transducer schema* Υ is a tuple $(\Upsilon_{\text{in}}, \Upsilon_{\text{out}}, \Upsilon_{\text{msg}}, \Upsilon_{\text{mem}}, \Upsilon_{\text{sys}})$ of database schemas, called respectively "input", "output", "message", "memory" and "system". A relation name can occur in at most one database schema of Υ. We fix Υ_{sys} to always contain two unary relations `Id` and `All`. A *transducer state* for Υ is a database instance over $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$.

We make a distinction between two kinds of relational transducers, depending on how they communicate with their environment. This leads to *epidemic* and *addressing* transducers. If the kind is understood from the context, we will just say "transducer".

2.4.1 Epidemic Transducer

An *epidemic (relational) transducer* Π over Υ is a collection of queries:

- for each $R^{(k)} \in \Upsilon_{\text{out}}$ there is a query $\mathcal{Q}^R_{\text{out}}$ having output schema $\{R^{(k)}\}$;
- for each $R^{(k)} \in \Upsilon_{\text{mem}}$ there are queries $\mathcal{Q}^R_{\text{ins}}$ and $\mathcal{Q}^R_{\text{del}}$ both having output schema $\{R^{(k)}\}$;
- for each $R^{(k)} \in \Upsilon_{\text{msg}}$ there is a query $\mathcal{Q}^R_{\text{snd}}$ having output schema $\{R^{(k)}\}$;

where each of these queries has the input schema $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{msg}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$. These queries will form the internal mechanism that a computing node uses to update its local storage and to send messages. The transducer model is parameterized by a generic query language $\mathcal{L}$: this language is used to concretely specify the above queries, in which case we call Π an $\mathcal{L}$-transducer.

A *local transition* of Π is a 4-tuple $(I, I_{\text{rcv}}, J, J_{\text{snd}})$, also denoted as $I, I_{\text{rcv}} \rightarrow J, J_{\text{snd}}$, where I and J are transducer states for Υ, and I_{rcv} and J_{snd} are instances over Υ_{msg}, such that (abbreviating $I' = I \cup I_{\text{rcv}}$):

$$\begin{aligned}
J|_{\Upsilon_{\text{in}},\Upsilon_{\text{sys}}} &= I|_{\Upsilon_{\text{in}},\Upsilon_{\text{sys}}}; \\
J|_{\Upsilon_{\text{out}}} &= I|_{\Upsilon_{\text{out}}} \cup \bigcup_{R^{(k)} \in \Upsilon_{\text{out}}} \mathcal{Q}^R_{\text{out}}(I'); \\
J|_{\Upsilon_{\text{mem}}} &= \bigcup_{R^{(k)} \in \Upsilon_{\text{mem}}} (I|_R \cup R^+(I')) \setminus R^-(I') \\
J_{\text{snd}} &= \bigcup_{R^{(k)} \in \Upsilon_{\text{msg}}} \mathcal{Q}^R_{\text{snd}}(I'),
\end{aligned}$$

where, following the presentation in [59],

$$\begin{aligned}
R^+(I') &= \mathcal{Q}^R_{\text{ins}}(I') \setminus \mathcal{Q}^R_{\text{del}}(I'); \text{ and,} \\
R^-(I') &= \mathcal{Q}^R_{\text{del}}(I') \setminus \mathcal{Q}^R_{\text{ins}}(I').
\end{aligned}$$

Intuitively, on receipt of the message facts I_{rcv}, a local transition updates the old transducer state I to new transducer state J and sends the facts in J_{snd}. When compared to I, in J potentially more output facts are produced; and, the update semantics for each memory relation R adds the facts produced by *insertion* query $\mathcal{Q}^R_{\text{ins}}$, removes the facts produced by *deletion* query $\mathcal{Q}^R_{\text{del}}$, and there is no-op semantics in case a fact is both added and removed at the same time [54]. Output facts can not be removed. Note that local transitions are deterministic in the following sense: if $I, I_{\text{rcv}} \rightarrow J, J_{\text{snd}}$ and $I, I_{\text{rcv}} \rightarrow J', J'_{\text{snd}}$ then $J = J'$ and $J_{\text{snd}} = J'_{\text{snd}}$.

2.4.2 Addressing Transducer

In the literature on declarative networking, a seemingly common feature seems to be that nodes send each message to a specifically addressed neighbor [45, 35, 14, 49]. Let Υ be a transducer schema. An *addressing (relational) transducer* Π over Υ is defined just like an epidemic transducer, except that for each $R^{(k)} \in \Upsilon_{\text{msg}}$ there is a query $\mathcal{Q}^R_{\text{snd}}$ having output schema $\{R^{(k+1)}\}$ instead of $\{R^{(k)}\}$. The extra component will contain the addressee of each message, which is by convention the first component. The semantics for local transitions is defined in the same way as for epidemic transducers, except that now each fact in J_{snd} has an extra component.

2.5 Transducer Networks

Here we formalize our first model for declarative networking. Concrete examples of this model can be found in Chapters 3 and 4. A *transducer network* $\boldsymbol{\mathcal{N}}$ is a triple $(\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ where $\mathcal{N}$ is a network, $\boldsymbol{\Upsilon}$ is a function mapping each $x \in \textit{nodes}(\mathcal{N})$ to a transducer schema, and $\boldsymbol{\Pi}$ is a function mapping each $x \in \textit{nodes}(\mathcal{N})$ to a transducer over schema $\boldsymbol{\Upsilon}(x)$. For technical convenience, we assume that all transducer schemas use the same message relations (with the same arities). This is not really a restriction because the transducers are not obliged to use all message relations.

2.5.1 Distributed Schemas

Naturally, we can define the distributed input database schema $in^{\boldsymbol{\mathcal{N}}}$ for $\boldsymbol{\mathcal{N}}$ that maps each $x \in \mathit{nodes}(\mathcal{N})$ to the input subschema of $\boldsymbol{\Upsilon}(x)$. The distributed schemas $out^{\boldsymbol{\mathcal{N}}}$ and $mem^{\boldsymbol{\mathcal{N}}}$ are defined similarly, but now using the output and memory subschemas.

2.5.2 Operational Semantics

Any distributed database instance H over $in^{\boldsymbol{\mathcal{N}}}$ can be given as input to $\boldsymbol{\mathcal{N}}$. A *configuration of* $\boldsymbol{\mathcal{N}}$ *on* H is a pair $\rho = (s, b)$ of functions s and b where for each $x \in \mathit{nodes}(\mathcal{N})$,

- letting $\mathcal{D}_1 = \boldsymbol{\Upsilon}(x)_{\text{in}}$ and $\mathcal{D}_2 = \boldsymbol{\Upsilon}(x)_{\text{sys}}$, function s maps x to a transducer state $s(x)$ for $\boldsymbol{\Upsilon}(x)$ such that $s(x)|_{\mathcal{D}_1} = H(x)$ and $s(x)|_{\mathcal{D}_2} = \{\mathtt{Id}(x)\} \cup \{\mathtt{All}(y) \mid y \in \mathit{nodes}(\mathcal{N})\}$; and,
- b maps x to a finite multiset of facts over the shared message schema of $\boldsymbol{\mathcal{N}}$.

A configuration describes a snapshot of the network at some moment during its evolution. We call s the *state* function and b the *(message) buffer* function. Function s maps each node x to its state $s(x)$, where instance H provides the local input of x, and the system relations `Id` and `All` provide transducer $\boldsymbol{\Pi}(x)$ respectively the identity of x and the identities of all nodes. Next, the function b maps x to its message buffer $b(x)$, that is a multiset of message facts, with the intuition that these are the messages sent to x but that are not yet delivered. A multiset allows us to represent duplicates of the same message (sent at different times).

The *start configuration of* $\boldsymbol{\mathcal{N}}$ *on* H, denoted $\mathit{start}(\boldsymbol{\mathcal{N}}, H)$, is the unique configuration $\rho = (s, b)$ of $\boldsymbol{\mathcal{N}}$ on H where for each $x \in \mathit{nodes}(\mathcal{N})$, letting $\mathcal{D} = \boldsymbol{\Upsilon}(x)_{\text{out}} \cup \boldsymbol{\Upsilon}(x)_{\text{mem}}$, we have $s(x)|_{\mathcal{D}} = \emptyset$ and $b(x) = \emptyset$.

We now describe the actual computation of the transducer network. A *global transition* of $\boldsymbol{\mathcal{N}}$ on input H is a 4-tuple (ρ_1, x, m, ρ_2), also denoted as $\rho_1 \xrightarrow{x,m} \rho_2$, where $x \in \mathit{nodes}(\mathcal{N})$, and $\rho_1 = (s_1, b_1)$ and $\rho_2 = (s_2, b_2)$ are configurations of $\boldsymbol{\mathcal{N}}$ on H such that

- $m \sqsubseteq b_1(x)$ and there exists a J_{snd} such that
$$s_1(x),\ \mathit{set}(m) \to s_2(x),\ J_{\text{snd}}$$
is a local transition of transducer $\boldsymbol{\Pi}(x)$;
- for each $y \in \mathit{nodes}(\mathcal{N}) \setminus \{x\}$ we have $s_2(y) = s_1(y)$;
- regarding the message buffers, we have
  - (i) $b_2(x) = b_1(x) \setminus m$;
  - (ii) for each $y \in \mathit{neighbor}(x, \mathcal{N})$ we have $b_2(y) = b_1(y) \cup J_{\text{snd}}^{\to y}$ where $J_{\text{snd}}^{\to y} = J_{\text{snd}}$ if $\boldsymbol{\Pi}(x)$ is epidemic and $J_{\text{snd}}^{\to y} = \{R(\bar{a}) \mid R(y, \bar{a}) \in J_{\text{snd}}\}$ if $\boldsymbol{\Pi}(x)$ is addressing; and,
  - (iii) for all other nodes y we have $b_2(y) = b_1(y)$.

We call x the *active node* and m the *delivered messages*. Intuitively, in a global transition, we select an arbitrary node x and allow it to receive some arbitrary submultiset m from its message buffer. The messages in m are then delivered at node x (as a set, i.e., without duplicates) and x performs a local transition, in which it updates its memory and output relations, and possibly sends some new messages. If $\mathbf{\Pi}(x)$ is epidemic, the sent messages are broadcast to all neighbors, and if $\mathbf{\Pi}(x)$ is addressing, the addressee is used to put the message in the right neighbor buffer.[1] If $m = \emptyset$, we call this global transition a *heartbeat* transition. A heartbeat transition corresponds to the real life situation in which a node does a computation step when a local timer goes off and no messages have been received from the network.

A *run* $\mathcal{R}$ of transducer network $\boldsymbol{\mathcal{N}}$ on input distributed database instance H is a sequence of global transitions $\rho_i \xrightarrow{x_i, m_i} \rho_{i+1}$ for $i = 1, 2, 3, \ldots$, where $\rho_1 = \mathit{start}(\boldsymbol{\mathcal{N}}, H)$, and the i^{th} transition with $i \geq 2$ operates on the resulting configuration of the previous transition $i - 1$. Runs can be finite or infinite, but in each of the chapters we will say which option is taken.

Note, when a node changes its output or memory relations during one global transition, then these changes are visible to that node only starting from the next global transition in which that node is active. Also, several facts can be delivered together during a transition, regardless of whether they were sent during different earlier transitions or during the same earlier transition.

We have not defined global transitions that are concurrent, i.e., global transitions in which multiple nodes simultaneously receive messages from their own message buffer and do a local transition. This can be simulated by multiple sequential global transitions: let the nodes become active in some arbitrary order, and each active node just reads its own message buffer. Because local transitions are deterministic, the nodes will update their state and send messages in the same way as they would during a concurrent transition.

2.5.3 Fairness

When infinite runs are considered, in the literature on process models it is customary to require certain "fairness" conditions [31, 22, 41]. Let $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \mathbf{\Upsilon}, \mathbf{\Pi})$ be a transducer network. An infinite run of $\boldsymbol{\mathcal{N}}$ on some input distributed database instance is called *fair* if *(i)* every node of $\mathcal{N}$ is active in an infinite number of transitions and *(ii)* if for some node a fact occurs in its message buffer in an infinite number of configurations, then this fact is delivered to that node during an infinite number of transitions. Intuitively, the last condition demands that no sent messages are infinitely delayed. All infinite runs that we consider are assumed to be fair, unless explicitly specified otherwise.

Note, every transducer network has an infinite fair run for every input because heartbeat transitions are still possible even when the message buffers have become empty.

[1]The first component of each fact in J_{snd} is regarded as the addressee, and this component is projected away during the transfer of the message to the buffer of that addressee. Messages having an addressee that is not a neighbor are lost.

2.5.4 Message Delivery Constraints

We may want to impose a size-constraint on the delivered message multisets. Indeed, for a transducer network $\mathcal{N}$ and a natural number $k \geq 1$, we can restrict our attention to runs of $\mathcal{N}$ where the sizes of the delivered message multisets are of size at most k. This is the *k-delivery* semantics for $\mathcal{N}$. No such bound is assumed, unless explicitly mentioned.

2.6 Conjunctive Queries

We now recall the query language *unions of conjunctive queries with (safe) negation*, abbreviated $\text{UCQ}^{\neg}$. This language is equivalent to the existential fragment of first-order logic [2]. It will be convenient to use a slightly unconventional formalization of conjunctive queries.

Let **var** be a universe of *variables*, disjoint from **dom**. An *atom* is of the form $R(u_1, \ldots, u_k)$ where R is a relation name and $u_i \in \mathbf{var} \cup \mathbf{dom}$ for $i = 1, \ldots, k$. We call R the *predicate*. If an atom contains no data values, then we call it *constant-free*. A *literal* is an atom, or an atom with "$\neg$" prepended; these literals are respectively called *positive* and *negative*.

A *conjunctive query* (or simply *rule*) φ is a triple

$$(\mathit{head}_\varphi,\ \mathit{pos}_\varphi,\ \mathit{neg}_\varphi)$$

where head_φ is an atom, and pos_φ and neg_φ are sets of atoms. The components head_φ, pos_φ and neg_φ are called respectively the *head*, the *positive body atoms* and the *negative body atoms*. The union of the last two sets is called the *body atoms*. Note that in our formalization, the set neg_φ contains just atoms, not negative literals. Every rule φ must have a head, whereas pos_φ and neg_φ may be empty. If $\mathit{neg}_\varphi = \emptyset$ then φ is called *positive*. If all atoms comprising φ are constant-free, then φ is called *constant-free*.

A rule φ may be written in the conventional syntax. For instance, if $\mathit{head}_\varphi = T(\mathtt{u}, \mathtt{v})$, $\mathit{pos}_\varphi = \{R(\mathtt{u}, \mathtt{v})\}$ and $\mathit{neg}_\varphi = \{S(\mathtt{v})\}$, with $\mathtt{u}, \mathtt{v} \in \mathbf{var}$, then we can write φ as

$$T(\mathtt{u}, \mathtt{v}) \leftarrow R(\mathtt{u}, \mathtt{v}),\ \neg S(\mathtt{v}).$$

The specific ordering of literals to the right of the arrow is arbitrary. We will often refer to the body literals more directly, by prepending the symbol "$\neg$" to the negative body atoms. For the previous example, the body literals are $R(\mathtt{u}, \mathtt{v})$ and $\neg S(\mathtt{v})$.

We call φ *safe* if the variables occurring in head_φ and neg_φ also occur in pos_φ. The set of variables of φ is denoted $\mathit{vars}(\varphi)$. If $\mathit{vars}(\varphi) = \emptyset$ then φ is called *ground*, in which case we will consider $\{\mathit{head}_\varphi\} \cup \mathit{pos}_\varphi \cup \mathit{neg}_\varphi$ to be a set of facts.

Let $\mathcal{D}$ be a database schema. A rule φ is said to be *over schema $\mathcal{D}$* if for each atom $R(u_1, \ldots, u_k) \in \{\mathit{head}_\varphi\} \cup \mathit{pos}_\varphi \cup \mathit{neg}_\varphi$ we have $R^{(k)} \in \mathcal{D}$. Suppose φ is over $\mathcal{D}$. A *valuation* for φ is a total function $V : \mathit{vars}(\varphi) \to \mathbf{dom}$. Note, there is only one valuation for ground rules, namely, the one having an empty domain. Now, we define the *application* of V to an atom $R(u_1, \ldots, u_k)$ of φ, denoted $V(R(u_1, \ldots, u_k))$, as the *fact* $R(a_1, \ldots, a_k)$ where for $i = 1, \ldots, k$ we have $a_i = V(u_i)$ if $u_i \in \mathbf{var}$ and $a_i = u_i$ otherwise. In words: the application of valuation V replaces the variables by data values and leaves the old data values unchanged. This notation is naturally extended to a set of atoms, which results in a set of facts. Now, let I be an instance over $\mathcal{D}$. The valuation V is said to be *satisfying for* φ *on* I if $V(pos_\varphi) \subseteq I$ and $V(neg_\varphi) \cap I = \emptyset$. If this is so, then φ is said to *derive* the fact $V(head_\varphi)$. The *result of* φ *applied to* I, denoted $\varphi(I)$, is the set of facts derived by all satisfying valuations for φ on I.

A *union of conjunctive queries with negation* over a database schema $\mathcal{D}$ is a finite set Φ of rules over $\mathcal{D}$ that all have the same head predicate. The resulting language is denoted UCQ$^\neg$, and Φ will also be called a UCQ$^\neg$*-program*. Let I be a database instance. The *result of* Φ *applied to* I, denoted $\Phi(I)$, is defined as $\bigcup_{\varphi\in\Phi}\varphi(I)$. Note, if $\Phi = \emptyset$ then $\Phi(I) = \emptyset$.

The language UCQ is the sublanguage of UCQ$^\neg$ where only positive rules can be used.

2.7 Datalog

We recall the language Datalog with negation [2], which we abbreviate as Datalog$^\neg$.

Let $\mathcal{D}$ be a database schema. A Datalog$^\neg$ *program* P *over* $\mathcal{D}$ is a set of safe rules over $\mathcal{D}$ (defined in Section 2.6). We call P *constant-free* if all rules in P are constant-free. We will write $sch(P)$ to denote the database schema that P is over. We define $idb(P) \subseteq sch(P)$ to be the database schema consisting of all relations occurring in rule-heads of P. We abbreviate $edb(P) = sch(P) \setminus idb(P)$.[2]

An *input* for P is a database instance over $sch(P)$. Note, we allow inputs to already contain facts over $idb(P)$, and the reason for this slightly unconventional input definition will become clear in Section 2.8.2.

2.7.1 Positive and Semi-positive

Let P be a Datalog$^\neg$ program. We say that P is *positive* if it has only positive rules. Positive Datalog$^\neg$ is commonly referred to as simply Datalog [2]. We say that P is *semi-positive* if for each rule $\varphi \in P$, all predicates used in atoms of neg_φ are contained in $edb(P)$. Naturally, positive programs are semi-positive.

Let P be a semi-positive Datalog$^\neg$ program. We now give the semantics of P [2]. We define the *immediate consequence operator* T_P that maps each instance J over $sch(P)$ to the instance $J' = J \cup A$ where A is the set of facts derived by all possible satisfying valuations for the rules of P on J. Note that $adom(J') \subseteq adom(J)$.

Let I be an instance over $sch(P)$. Consider the infinite sequence I_0, I_1, I_2, etc, that is inductively defined as follows: $I_0 = I$ and $I_i = T_P(I_{i-1})$ for each $i \geq 1$. We define the *output of* P *on input* I, denoted $P(I)$, as $\bigcup_j I_j$; this is the *minimal fixpoint* of the T_P operator. Note, $I \subseteq P(I)$. When I is finite, the fixpoint is finite and can be computed in polynomial time (under data complexity [56]).

2.7.2 Stratified Semantics

We now recall the stratified semantics of a Datalog$^\neg$ program P [2]. To improve readability, as a slight abuse of notation, here we will treat $idb(P)$ as a set of only

[2]The abbreviation “idb” stands for “intensional database schema” and “edb” stands for “extensional database schema” [2].

relation names (without associated arities). The program P is called *syntactically stratifiable* if there is a function $\sigma : \mathit{idb}(P) \to \{1, \ldots, |\mathit{idb}(P)|\}$ such that for each rule $\varphi \in P$, having some head predicate T, the following conditions are satisfied:

- $\sigma(R) \leq \sigma(T)$ for each $R(\bar{v}) \in \mathit{pos}_{\varphi}|_{\mathit{idb}(P)}$;
- $\sigma(R) < \sigma(T)$ for each $R(\bar{v}) \in \mathit{neg}_{\varphi}|_{\mathit{idb}(P)}$.

For $R \in \mathit{idb}(P)$, we call $\sigma(R)$ the *stratum number* of R. For technical convenience, we may assume that if there is an $R \in \mathit{idb}(P)$ with $\sigma(R) > 1$ then there is an $S \in \mathit{idb}(P)$ with $\sigma(S) = \sigma(R) - 1$.

Intuitively, the function σ partitions P into a sequence of semi-positive Datalog$^\neg$ programs $P_1, \ldots, P_k$ with $k \leq |\mathit{idb}(P)|$ such that for each $i = 1, \ldots, k$, the program P_i is the set of rules of P whose head predicate has stratum number i. Rules with the same head predicate are always in the same semi-positive program. This sequence of semi-positive programs is called a *syntactic stratification* of P. We can now apply the *stratified semantics* to P: for an input I over $\mathit{sch}(P)$, we first compute the fixpoint $P_1(I)$, then the fixpoint $P_2(P_1(I))$, etc. The *output of P on input I*, denoted $P(I)$, is then defined as $P_k(P_{k-1}(\ldots P_1(I)\ldots))$. It is well known that the output of P does not depend on the chosen syntactic stratification (in the case that more than one exists).

Not all Datalog$^\neg$ programs are syntactically stratifiable.

2.7.3 Stable Model Semantics

We now recall the stable model semantics of a Datalog$^\neg$ program P [32, 52]. Let I be a database instance over $\mathit{sch}(P)$. Let $\varphi \in P$. Let V be a valuation for φ whose image is contained in $\mathit{adom}(I)$. Valuation V does not have to be satisfying for φ on I. Together, V and φ give rise to a ground rule ψ, that is precisely φ except that each $u \in \mathit{vars}(\varphi)$ is replaced by $V(u)$. We call ψ a *ground rule of φ with respect to I*. Let $\mathit{ground}(\varphi, I)$ denote the set of all ground rules of φ that we can make with respect to I. The *ground program of P on input I*, denoted $\mathit{ground}(P, I)$, is defined as $\bigcup_{\varphi \in P} \mathit{ground}(\varphi, I)$.

Let M be a set of facts over the schema $\mathit{sch}(P)$. We write $\mathit{ground}_M(P, I)$ to denote the program obtained from $\mathit{ground}(P, I)$ as follows:

1. remove every rule $\psi \in \mathit{ground}(P, I)$ for which $\mathit{neg}_{\psi} \cap M \neq \emptyset$;
2. remove the negative (ground) body atoms from all remaining rules.

Note that $\mathit{ground}_M(P, I)$ is a positive program. We say that M is a *stable model of P on input I* if M is the output of $\mathit{ground}_M(P, I)$ on input I. Hence, $I \subseteq M$ and $\mathit{adom}(M) \subseteq \mathit{adom}(I)$ by the semantics of positive Datalog$^\neg$ programs.

Not all Datalog$^\neg$ programs have stable models on every input.

2.8 Dedalus

We now recall the language Dedalus [13, 14, 37], that can be used to describe distributed computations. This constitutes our second model for declarative networking.

Essentially, Dedalus is an extension of Datalog$^\neg$ to represent updatable memory for the nodes of a network and to provide a mechanism for communication between these nodes, similar to transducer networks. Concrete example programs are given in Chapters 5 and 6.

In the context of Dedalus, a *(computer) network* is a just nonempty finite set $\mathcal{N}$ of nodes, which are again values in **dom**. Communication channels (edges) are not explicitly represented because we allow a node x to send a message to any node y, as long as x knows about y by means of input relations or received messages. When using Dedalus for general distributed or cluster computing, the delivery of messages is handled by the network layer, which is abstracted away. But Dedalus programs can also describe the network layer itself [45, 37], in which case we would restrict attention to programs where nodes only send messages to nodes to which they are explicitly linked; these nodes would again be provided as input.

Also, each node is running the same program. Consequently, a *distributed database instance* H is over a network $\mathcal{N}$ and one database schema $\mathcal{D}$: H is a function mapping every node of $\mathcal{N}$ to an ordinary finite database instance over $\mathcal{D}$. This represents how data over the same schema $\mathcal{D}$ is spread over the nodes of a network.

2.8.1 Syntax

We present Dedalus as Datalog$^\neg$ extended with a simple annotation mechanism, which keeps the notations simpler.[3] Let $\mathcal{D}$ be a database schema. We write $\mathbf{B}\{\bar{\mathtt{v}}\}$, where $\bar{\mathtt{v}}$ is a tuple of variables, to denote any sequence β of literals over database schema $\mathcal{D}$, such that the variables in β are precisely those in the tuple $\bar{\mathtt{v}}$. Let $R(\bar{\mathtt{u}})$ denote any atom over $\mathcal{D}$. There are three types of Dedalus rules over $\mathcal{D}$:

- A *deductive* rule is a normal Datalog$^\neg$ rule over $\mathcal{D}$.
- An *inductive* rule is of the form
$$R(\bar{\mathtt{u}})\bullet \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}.$$
- An *asynchronous* rule is of the form
$$R(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}.$$

For inductive rules, the annotation '$\bullet$' can be likened to the transfer of "tokens" in a Petri net from the old state to the new state. For asynchronous rules, the annotation '$\mid \mathtt{y}$' with $\mathtt{y} \in \mathbf{var}$ means that the derived head facts are transferred ("piped") to the node represented by $\mathtt{y}$. Deductive, inductive and asynchronous rules will express respectively local computation, updatable memory, and message sending (cf. Section 2.8.2). Like for Datalog$^\neg$, a Dedalus rule is called *safe* if all its variables occur in at least one positive body atom.

To illustrate, if $\mathcal{D} = \{R^{(2)}, S^{(1)}, T^{(2)}\}$, then the following three rules are examples of, respectively, deductive, inductive and asynchronous rules over $\mathcal{D}$:

$$T(\mathtt{u},\mathtt{v}) \leftarrow R(\mathtt{u},\mathtt{v}),\ \neg S(\mathtt{v}).$$

[3] These annotations correspond to syntactic sugar in the previous presentations of Dedalus.

$$T(\mathtt{u},\mathtt{v})\bullet \leftarrow R(\mathtt{u},\mathtt{v}).$$

$$T(\mathtt{u},\mathtt{v}) \mid \mathtt{y} \leftarrow R(\mathtt{u},\mathtt{v}),\ S(\mathtt{y}).$$

Now consider the following definition:

Definition 2.1. A Dedalus *program over a schema $\mathcal{D}$* is a set of deductive, inductive and asynchronous Dedalus rules over $\mathcal{D}$, such that all rules are safe, and the set of deductive rules is syntactically stratifiable.

We will additionally assume that Dedalus programs are constant-free, as is common in the theory of database query languages, and which is not really a limitation, since constants that are important for the program can always be indicated by unary relations in the input.

Let $\mathcal{P}$ be a Dedalus program. We write $sch(\mathcal{P})$ to denote the schema that $\mathcal{P}$ is over. The definitions of $idb(\mathcal{P})$ and $edb(\mathcal{P})$ are like for Datalog$^\neg$ programs.

An *input* for $\mathcal{P}$ is a *distributed* database instance H over some network $\mathcal{N}$ and the schema $edb(\mathcal{P})$. The next section gives the operational semantics for Dedalus.

2.8.2 Operational Semantics

In this section we define our operational semantics for Dedalus. We describe how a network executes a Dedalus program $\mathcal{P}$ when an input distributed database is given. This operational semantics is in line with earlier work on declarative networking [30, 48, 35, 1], and also the operational semantics for transducer networks (Section 2.5.2).

The essence of the operational semantics is as follows. By definition, the input distributed database is over a certain network. Every node of the network runs the *same* Dedalus program, and a node has access only to its own local state and any received messages. The nodes are made active one by one in some arbitrary order, and this continues an infinite number of times. During each active moment of a node x, called a *local (computation) step*, node x receives message facts and applies its deductive, inductive and asynchronous rules. Concretely, the deductive rules, forming a stratified Datalog$^\neg$ subprogram, are applied to the incoming messages and the previous state of x. Next, the inductive rules are applied to the output of the deductive subprogram, and these allow x to store facts in its memory: these facts become visible in the next local step of x. Finally, the asynchronous rules are also applied to the output of the deductive subprogram, and these allow x to send facts to the other nodes or to itself. These facts become visible at the addressee after some arbitrary delay, which represents asynchronous communication. We will refer to local steps simply as "steps".

In the next subsections, we make the above sketch more concrete. Fix some Dedalus program $\mathcal{P}$.

2.8.2.1 Configurations

A configuration describes the network at a certain point in its evolution. Let H be an input distributed database instance for $\mathcal{P}$, over a network $\mathcal{N}$. We define a *configuration ρ of $\mathcal{P}$ on H* to be a pair (s, b) where

- s is a function that maps each node of $\mathcal{N}$ to a set of facts over $sch(\mathcal{P})$;

- b is a function that maps each node of $\mathcal{N}$ to a set of pairs of the form $(i, \boldsymbol{f})$, where $i \in \mathbb{N}$ and $\boldsymbol{f}$ is a fact over $\mathit{idb}(\mathcal{P})$.

We call s the *state* function and b the *(message) buffer* function respectively. They have the same meaning as in the operational semantics for transducer networks. The reason for having numbers i, called *send-tags*, attached to facts in the image of b is merely a technical convenience: these numbers help separate multiple instances of the same fact when it is sent at different moments (to the same addressee), and these send-tags will not be visible to the Dedalus program.[4]

The *start configuration of* $\mathcal{P}$ *on input* H, denoted $\mathit{start}(\mathcal{P}, H)$, is the configuration $\rho = (s, b)$ of $\mathcal{P}$ on H defined by

- $s(x) = H(x)$ for each $x \in \mathcal{N}$;
- $b(x) = \emptyset$ for each $x \in \mathcal{N}$.

In words: for every node, the state is initialized with its local input fragment in H, and there are no sent messages.

2.8.2.2 Subprograms

We look at the operations that are executed locally during each step of a node. We have mentioned above that the three types of Dedalus rules each have their own purpose in the operational semantics. For this reason, we split the program $\mathcal{P}$ into three subprograms, that contain respectively the deductive, inductive and asynchronous rules. In Section 2.8.2.3, we describe how these subprograms are used in the operational semantics.

- First, we define $\mathit{deduc}_{\mathcal{P}}$ to be the Datalog$^\neg$ program consisting of precisely all deductive rules of $\mathcal{P}$.
- Secondly, we define $\mathit{induc}_{\mathcal{P}}$ to be the Datalog$^\neg$ program consisting of all inductive rules of $\mathcal{P}$ after the annotation '$\bullet$' in their head is removed.
- Thirdly, we define $\mathit{async}_{\mathcal{P}}$ to be the Datalog$^\neg$ program consisting of precisely all rules
$$T(\mathtt{y}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \mathtt{y}\}$$
where
$$T(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \mathtt{y}\}$$
is an asynchronous rule of $\mathcal{P}$. So, we basically put the variable $\mathtt{y}$ as the first component in the (extended) head atom. The intuition for the generated head facts is that the first component will represent the addressee.

Note, the programs $\mathit{deduc}_{\mathcal{P}}$, $\mathit{induc}_{\mathcal{P}}$ and $\mathit{async}_{\mathcal{P}}$ are just Datalog$^\neg$ programs over the schema $\mathit{sch}(\mathcal{P})$. Moreover, $\mathit{deduc}_{\mathcal{P}}$ is syntactically stratifiable because the deductive rules in every Dedalus program must be syntactically stratifiable. It is possible however that $\mathit{induc}_{\mathcal{P}}$ and $\mathit{async}_{\mathcal{P}}$ are not syntactically stratifiable. Now we define the semantics of each of these three subprograms.

[4]We could have equivalently modeled message buffers with multisets, where messages are not tagged. The tags however are technically more convenient in showing our results about Dedalus.

Let I be a database instance over $sch(\mathcal{P})$. During each step of a node, the intuition of the deductive rules is that they "complete" the available facts by adding all new facts that can be logically derived from them. This calls for a fixpoint semantics, and for this reason, we define the *output of* $deduc_{\mathcal{P}}$ *on input* I, denoted as $deduc_{\mathcal{P}}(I)$, to be given by the stratified semantics. This implies $I \subseteq deduc_{\mathcal{P}}(I)$. Importantly, I is allowed to contain facts over $idb(\mathcal{P})$, and the intuition is that these facts were derived during a previous step (by inductive rules) or received as messages (as sent by asynchronous rules). This will become more explicit in Section 2.8.2.3.

During each step of a node, the intuition behind the inductive rules is that they store facts in the memory of the node, and these stored facts will become visible during the next step. There is no notion of a fixpoint here because facts that will become visible in the next step are not available in the current step to derive more facts. For this reason, we define the *output of* $induc_{\mathcal{P}}$ *on input* I to be the set of facts derived by the rules of $induc_{\mathcal{P}}$ for all possible satisfying valuations in I, in just one derivation step. This output is denoted as $induc_{\mathcal{P}}\langle I\rangle$. Possibly $I \cap induc_{\mathcal{P}}\langle I\rangle = \emptyset$.

During each step of a node, the intuition behind the asynchronous rules is that they generate "message" facts that are to be sent around the network. The *output for* $async_{\mathcal{P}}$ *on input* I is defined in the same way as for $induc_{\mathcal{P}}$, except that we now use the rules of $async_{\mathcal{P}}$ instead of $induc_{\mathcal{P}}$. This output is denoted as $async_{\mathcal{P}}\langle I\rangle$. The intuition for not requiring a fixpoint for $async_{\mathcal{P}}$ is that a message fact will arrive at another node, or at a later step of the sender node, and can therefore not be read during sending.

Regarding data complexity [56], for each subprogram the output can be computed in PTIME with respect to the size of its input.

2.8.2.3 Transitions and Runs

Let H be as above. We define how to go from one configuration ρ_1 to another configuration ρ_2. This is quite similar to the operational semantics for transducer networks (cf. Section 2.5.2). Again, a *transition* describes how one active node does a local computation step to update its state and to send messages around the network. Such transitions are chained in a *run* to describe a full execution of the Dedalus program on the given input.

As a small notational aid, for a set m of pairs of the form $(i, \boldsymbol{f})$, we define $untag(m) = \{\boldsymbol{f} \mid \exists i \in \mathbb{N} : (i, \boldsymbol{f}) \in m\}$.

A *transition with send-tag* $i \in \mathbb{N}$ is a five-tuple $(\rho_1, x, m, i, \rho_2)$, also denoted as

$$\rho_1 \xrightarrow[i]{x,m} \rho_2,$$

where $\rho_1 = (s_1, b_1)$ and $\rho_2 = (s_2, b_2)$ are configurations of $\mathcal{P}$ on input H, $x \in \mathcal{N}$, $m \subseteq b_1(x)$, and, letting

$$\begin{aligned} I =&\quad s_1(x) \cup untag(m), \\ D =&\quad deduc_{\mathcal{P}}(I), \\ \delta^{i\to y} =&\quad \{(i,\, R(\bar{a})) \mid R(y, \bar{a}) \in async_{\mathcal{P}}\langle D\rangle\} \text{ for each } y \in \mathcal{N}, \end{aligned}$$

for x we have

$$\begin{aligned} s_2(x) &= H(x) \cup \mathit{induc}_{\mathcal{P}}\langle D\rangle, \\ b_2(x) &= (b_1(x) \setminus m) \cup \delta^{i\to x}, \end{aligned}$$

and for each $y \in \mathcal{N} \setminus \{x\}$ we have

$$\begin{aligned} s_2(y) &= s_1(y) \\ b_2(y) &= b_1(y) \cup \delta^{i\to y}. \end{aligned}$$

We say this transition is *of* the *active* node x. Intuitively, the transition expresses how the active node x reads its old state $s_1(x)$ together with the received facts in $\mathit{untag}(m)$ (thus without the tags), and then completes this information with subprogram $\mathit{deduc}_{\mathcal{P}}$. Next, the state of x is changed to $s_2(x)$, which always contains the input facts of x, over schema $\mathit{edb}(\mathcal{P})$, and it also includes all facts derived by subprogram $\mathit{induc}_{\mathcal{P}}$, which is applied to the deductive fixpoint. So, input facts are never lost, and relations in $\mathit{idb}(\mathcal{P})$ have mutable state, where only the facts that are explicitly derived by $\mathit{induc}_{\mathcal{P}}$ are remembered. Only the state of the active node x changes. Lastly, the subprogram $\mathit{async}_{\mathcal{P}}$ is also applied to the deductive fixpoint. The generated facts are *messages*, and by the syntax of $\mathit{async}_{\mathcal{P}}$, these have an additional location specifier as their first component to indicate the addressee. For each $y \in \mathcal{N}$, the set $\delta^{i\to y}$ contains all messages addressed to y: we drop the addressee-component because all facts are destined for y, and we attach the send-tag i. The set $\delta^{i\to y}$ is then added to the buffer of y. Messages with an addressee outside the network are ignored. This transition semantics closely corresponds to that of the language Webdamlog [1].

A *run* $\mathcal{R}$ *of* $\mathcal{P}$ *on input* H is an *infinite* sequence of transitions, such that *(i)* the very first configuration is $\mathit{start}(\mathcal{P}, H)$, *(ii)* the target configuration of each transition is the source configuration of the next transition, and *(iii)* the transition at ordinal i of the run uses send-tag i. The resulting transition system is highly non-deterministic because in each transition we can choose the active node and also what messages from its buffer we want to deliver. An infinite number of transitions is always possible because the set of delivered messages may be empty.

A nice aspect of the operational semantics given here is that every message is just a fact over $\mathit{idb}(\mathcal{P})$. This allows the local Dedalus rules of a recipient node to treat received message facts in the same way as facts in its old state, i.e., there is no noticeable difference. From this viewpoint, communication is in some sense transparent to the nodes, which is one of the design principles of Dedalus.

As a final remark, transitions as they are defined here can also simulate *concurrent* transitions, in which multiple nodes are active at the same time and receive messages from their respective buffers (cf. Section 2.5.2).

2.8.2.4 Fairness and Arrival Function

Note, Dedalus runs are by definition always infinite. We impose on Dedalus runs the same fairness conditions as in Section 2.5.3. This is formalized next. Let H be an input distributed database instance for $\mathcal{P}$, over a network $\mathcal{N}$. Let $\mathcal{R}$ be a run of $\mathcal{P}$ on H. For each transition ordinal i, let $\rho_i = (s_i, b_i)$ denote the source configuration of transition i. Now, run $\mathcal{R}$ is called *fair* if:

- every node is the active node in an infinite number of transitions; and,
- for every transition ordinal i, for every $y \in \mathcal{N}$, for every pair $(j, \boldsymbol{f}) \in b_i(y)$, there is a transition having ordinal k with $i \leq k$ in which $(j, \boldsymbol{f})$ is delivered to y.

In this second condition, possibly $k = i$, and in that case $(j, \boldsymbol{f})$ is delivered in the transition immediately following configuration ρ_i. Because the pair $(j, \boldsymbol{f})$ can be in the message buffer of multiple nodes, this k is not unique for the pair $(j, \boldsymbol{f})$ by itself. But, when we also consider the addressee y, it follows from the operational semantics that this k is unique for the triple $(j, y, \boldsymbol{f})$. This reasoning gives rise to a function $\alpha_{\mathcal{R}}$, called the *arrival function for* $\mathcal{R}$, that is defined as follows: for every transition i, for every node y, for every fact $(i, \boldsymbol{f}) \in \delta^{i \to y}$ (i.e., $\boldsymbol{f}$ is sent to addressee y during i), the function $\alpha_{\mathcal{R}}$ maps $(i, y, \boldsymbol{f})$ to the transition ordinal k in which $(i, \boldsymbol{f})$ is delivered to y. We always have $\alpha_{\mathcal{R}}(i, y, \boldsymbol{f}) > i$. Indeed, the delivery of a message can only happen after it was sent. So, when the delivery of one message causes another to be sent, then the second one is delivered in a later transition. This is related to the topic of “causality”, discussed in Section 5.5.3.

Chapter 3

Relational Transducers for Declarative Networking

3.1 Outline

In this chapter we investigate the CALM conjecture in the model of transducer networks, and we investigate the expressiveness of such networks. First, Section 3.2 gives related work. Section 3.3 gives technical remarks specific to this chapter. Section 3.4 investigates the use of transducer networks for expressing conventional database queries in a distributed fashion. Section 3.5 discusses the issue of coordination, and looks into the CALM conjecture. Section 3.6 gives results about the expressiveness of transducer networks. Section 3.7 discusses a variation of the used transducer model.

3.2 Related Work

The desire to better understand coordination in the field of declarative networking is evidenced by the steadily growing literature on this subject. First, Alvaro et al. [12] look at coordination as a quantitative property that can be minimized. They describe a program analysis technique to identify syntactical locations in the code where coordination is not needed. The goal then, is to help the programmer iteratively reduce the number of locations where coordination is used.

Our conference paper [15] has also inspired follow-up work by others. In particular, Zinn et al. [59] have generalized our results to also include a "partitioning policy", which is a strategy to initialize every node of a network with input data before the computation starts. The basic idea is that each node is given local relations that provide information about how data is distributed, and in particular what data each node can autonomously reason about, i.e., without coordination with other nodes. This allows a node to sometimes perform nonmonotone operations without the need for communication. It turns out that in some variations of the model considered, all database queries can be implemented in a "coordination-free" manner. However, such a partitioning policy might be expensive in terms of how much additional data each node needs.

One of our results is that a monotone query can in principle be computed without coordination, but it remains open in what exact way the best performance can be achieved in a practical scenario. The work of Loo et al. [44] and Nigam et al. [49], however, provides concrete algorithms for the case of distributed Datalog programs. They want to efficiently update the state (i.e., the derivations) on nodes of the network whenever some input facts change. It would be too costly to completely recompute the state of every node when an update happens. Instead, they propose a technique that propagates only the incremental changes that have to be distributedly applied. This allows for sending less messages around the network, and does not require needless recomputations of data. Their algorithms require no coordination, can handle recursive Datalog rules, and can tolerate messages that are delivered out of order by the network.

This chapter is the extended version of our conference paper [15]. Some proof details are not included in this chapter, but can be found in the technical report [16].

3.3 General Remarks

In this chapter we restrict attention to so-called *homogeneous* transducer networks in which each node is running the same transducer (over the same schema). This model appears the most natural when considering standard database queries in the distributed setting, as we will do in this chapter. A homogeneous transducer network is denoted $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \Upsilon, \Pi)$, showing the network $\mathcal{N}$, the single transducer schema Υ and the single transducer Π over Υ. Importantly, to allow for a simpler presentation, we build up the definitions and results with *epidemic* transducers only. Homogeneous-epidemic transducer networks seem a basic but good model to reason about cloud computing. For completeness, in Section 3.7, we also relate the results to addressing transducers.

We will use infinite (fair) runs in this chapter, to express that the transducer networks run indefinitely.

For a homogeneous transducer network $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \Upsilon, \Pi)$, an input distributed database instance H is now also simplified because each node uses the same input schema Υ_{in}: instance H is a total function mapping each node of $\mathcal{N}$ to an ordinary database instance over Υ_{in}.

We recall some database query languages. First, the rule-based languages UCQ, UCQ$^{\neg}$, Datalog, and Datalog$^{\neg}$, are formalized in Chapter 2. Also recall the following languages [2]:

- FO: first order logic (relational calculus),
- While: FO with iteration,
- NrDatalog: Datalog with no recursion,
- NrDatalog$^{\neg}$: Datalog$^{\neg}$ with no recursion.[1]

Among all these query languages, UCQ has the least expressive power.

[1]We assume that the stratified semantics is used for NrDatalog$^{\neg}$.

3.4 Expressing Queries

What does it take for a transducer network to compute some global query? Here we propose a formal definition based on the two properties of *consistency* and *network-independence*. This is discussed in the following subsections.

3.4.1 Transducer Kinds

We will use the following terminology for transducers. We call a transducer *oblivious* if it does not read the relations `Id` and `All` in its queries. Intuitively, this means that the transducer is unaware of the network context, because it does not know about the node it is running on, and it does not know about the other nodes. A transducer is called *inflationary* if it never deletes facts from its memory relations, i.e., deletion queries for memory relations always return the empty set. A transducer is called *monotone* if all its queries are monotone. The later Example 3.4 describes a transducer that is at the same time oblivious, inflationary, and monotone.

3.4.2 Input and Output

To relate better to standard database queries, we frame the input and output of a transducer network in the context of ordinary (non-distributed) database instances. Let $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \Upsilon, \Pi)$ be a transducer network. Let I be an instance over Υ_{in}. This instance can be given as input to $\boldsymbol{\mathcal{N}}$ by partitioning it across the nodes, where the same fact can be given to multiple nodes. Formally, a distributed database instance H over $\mathcal{N}$ and Υ_{in} is said to be a *horizontal partition* of I if $I = \bigcup_{x \in nodes(\mathcal{N})} H(x)$.

Let $\rho = (s, b)$ be a configuration of $\boldsymbol{\mathcal{N}}$ on input H. Naturally, ρ defines an output database instance $out(\rho)$ over the schema Υ_{out} as follows:

$$out(\rho) = \bigcup_{x \in nodes(\mathcal{N})} s(x)|_{\Upsilon_{\text{out}}}.$$

Let $\mathcal{R}$ be a run of $\boldsymbol{\mathcal{N}}$ on H. Let ρ_1, ρ_2, etc, denote the sequence of configurations of $\mathcal{R}$. If there is a number $i \geq 1$ such that $out(\rho_i) = out(\rho_j)$ for all $j > i$, then we call i a *quiescence point* for $\mathcal{R}$. We call a configuration ρ_i of $\mathcal{R}$ a *quiescence configuration* if i is a quiescence point. Only quiescence configurations can follow a quiescence configuration, and all quiescence configurations define the same output database instance. Because the input is finite and the transducer queries are generic, only a finite number of distinct output facts are possible. The following property is now clear:

Proposition 3.1. For every transducer network, on every input, every run contains a quiescence configuration.

The *output* of $\mathcal{R}$ is now defined as $out(\rho_i)$ where ρ_i is a quiescence configuration of $\mathcal{R}$. Note, our notion of output does not specify what node is responsible for what piece of the output, as is common in cloud computing. Also, nodes are still allowed to send messages once a quiescence point is reached.

3.4.3 Consistency

We say that a transducer network $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \Upsilon, \Pi)$ is *consistent* if individually for all database instances I over Υ_{in}, on all horizontal partitions of I over $\mathcal{N}$, all runs of $\boldsymbol{\mathcal{N}}$ produce the same output, denoted $\boldsymbol{\mathcal{N}}(I)$.

When $\boldsymbol{\mathcal{N}}$ is consistent, this function $\boldsymbol{\mathcal{N}}(.)$ has the characteristic of a query, except that it need not be generic. For example, the "query" that asks for all data elements in the input that are not node identifiers, can be computed by a consistent transducer network. We mainly focus on the computation of generic queries. Naturally, $\boldsymbol{\mathcal{N}}$ is said to *compute a query* $\mathcal{Q}$ over input schema Υ_{in} and output schema Υ_{out} if $\boldsymbol{\mathcal{N}}$ is consistent and $\boldsymbol{\mathcal{N}}(I) = \mathcal{Q}(I)$ for every instance I over Υ_{in} on which $\mathcal{Q}$ is defined.

Because the individual queries that make up a transducer are generic, we can make the following observation:

Proposition 3.2. The function $\boldsymbol{\mathcal{N}}(.)$ is generic for each consistent transducer network $\boldsymbol{\mathcal{N}}$ in which the transducer is oblivious.

3.4.4 Examples

First, we explain our notational conventions for specifiying concrete transducers. Because FO is equivalent to NrDatalog$^\neg$ [2], we will frequently use the more readable rule-based syntax of NrDatalog$^\neg$ to specify FO-transducers. The answer relations of NrDatalog$^\neg$ programs will be clearly marked. For example, for a memory-insertion query $\mathcal{Q}^R_{\text{ins}}$, the answer relation of the corresponding NrDatalog$^\neg$ program is R_{ins}; for an output query $\mathcal{Q}^T_{\text{out}}$, the answer relation is T_{out}; for a message-sending query $\mathcal{Q}^S_{\text{snd}}$, the answer relation is S_{snd}. We leave a blank line between the NrDatalog$^\neg$ rules that belong to different queries. Unmentioned queries of a transducer are assumed to always return the empty set.

We now give some examples of transducer networks.

Example 3.3. For a simple example of a consistent transducer network, let the input be a binary relation R. Each node outputs the identical pairs from its part of the input. No messages are sent. This network computes the equality selection $\sigma_{\$1=\$2}(R)$. This is easily programmed on an FO-transducer, which is specified as follows. The transducer schema is $\Upsilon_{\text{in}} = \{R^{(2)}\}$, $\Upsilon_{\text{out}} = \{T^{(2)}\}$, $\Upsilon_{\text{msg}} = \emptyset$, $\Upsilon_{\text{mem}} = \emptyset$, and the single transducer rule is:

$$T_{\text{out}}(\mathtt{u}, \mathtt{u}) \leftarrow R(\mathtt{u}, \mathtt{u}).$$

□

Example 3.4. To give an example of a consistent transducer network involving communication, we compute the transitive closure of a binary relation R in a well-known distributed manner [45]. We present here a naive, unoptimized version. Each node sends its part of the input to its neighbors. Specifically, each node also forwards all messages it receives to its neighbors. This way, the input is flooded to all nodes. Each node accumulates the input facts it receives in a binary memory relation S. Finally, each node has an output relation T in which we repeatedly insert $R \cup S \cup (T \circ T)$, where $\circ$ stands for relational composition. Thanks to the monotonicity of the transitive closure, this transducer network is consistent. We can implement this idea with just an UCQ-transducer. The transducer schema is $\Upsilon_{\text{in}} = \{R^{(2)}\}$, $\Upsilon_{\text{out}} = \{T^{(2)}\}$, $\Upsilon_{\text{msg}} = \{U^{(2)}\}$, $\Upsilon_{\text{mem}} = \{S^{(2)}\}$, and the transducer rules are:

$$U_{\text{snd}}(\mathtt{u},\mathtt{v}) \leftarrow R(\mathtt{u},\mathtt{v}).$$
$$U_{\text{snd}}(\mathtt{u},\mathtt{v}) \leftarrow U(\mathtt{u},\mathtt{v}).$$

$$S_{\text{ins}}(\mathtt{u},\mathtt{v}) \leftarrow U(\mathtt{u},\mathtt{v}).$$

$$T_{\text{out}}(\mathtt{u},\mathtt{v}) \leftarrow R(\mathtt{u},\mathtt{v}).$$
$$T_{\text{out}}(\mathtt{u},\mathtt{v}) \leftarrow S(\mathtt{u},\mathtt{v}).$$
$$T_{\text{out}}(\mathtt{u},\mathtt{v}) \leftarrow T(\mathtt{u},\mathtt{w}),\, T(\mathtt{w},\mathtt{v}).$$

Note, the transducer is oblivious. There is no need to reason explicitly about node identifiers, because we only let the nodes steadily accumulate all input facts across the network and incrementally produce chunks of output. The transducer is also inflationary and monotone, reflecting the essential nature of the transitive closure computation. □

Example 3.5. Let us see a simple example of a transducer network that is *not* consistent. Consider a network with at least two nodes. The input is a unary relation R. Each node sends its part of R to its neighbors. Next, each node outputs the received messages on condition that the output is still empty. When there are at least two nodes and at least two different R-facts, different runs may deliver the messages in different orders, so different outputs can be produced, even for the same input distributed database instance. We can write an FO-transducer Π to implement this idea. The transducer schema is $\Upsilon_{\text{in}} = \{R^{(1)}\}$, $\Upsilon_{\text{out}} = \{T^{(1)}\}$, $\Upsilon_{\text{msg}} = \{U^{(1)}\}$, $\Upsilon_{\text{mem}} = \emptyset$, and the transducer rules are:

$$U_{\text{snd}}(\mathtt{u}) \leftarrow R(\mathtt{u}).$$

$$\mathtt{block}(\,) \leftarrow T(\mathtt{u}).$$
$$T_{\text{out}}(\mathtt{u}) \leftarrow \neg\mathtt{block}(\,),\, U(\mathtt{u}).$$

□

Undecidability for testing consistency of a transducer network readily follows from undecidability of satisfiability of FO (see [16] for the proof).

3.4.5 Network-Independence

We are mainly interested in the case where a query can be correctly computed by a transducer regardless of the network.

Let Π be a transducer over a schema Υ. We say that Π is *network-independent* if for all networks $\mathcal{N}$, the homogeneous transducer networks $(\mathcal{N}, \Upsilon, \Pi)$ are consistent and compute the same query $\mathcal{Q}$. We say that $\mathcal{Q}$ is the query *distributedly computed* by Π. The transducer from Example 3.4 is network-independent. Now consider the following example.

Example 3.6. We give a simple example of a transducer that gives rise to consistent transducer networks but that is *not* network-independent. Suppose we have a unary input relation R. Each node sends its own identifier to its neighbors. This way the edges of the network can be discovered. The discovered edges are forwarded to every node, and when a node detects that the collected edges together form a complete graph, then the node outputs its local input for relation R. If the network is indeed a complete graph, by fairness eventually all nodes will detect this, and then the transducer network computes the identity query. But on other network topologies the empty query is computed.

For completeness, we specify an FO-transducer Π to implement this idea. We define the transducer schema Υ as $\Upsilon_{\text{in}} = \{R^{(1)}\}$, $\Upsilon_{\text{out}} = \{T^{(1)}\}$, $\Upsilon_{\text{msg}} = \{A^{(1)}, B^{(2)}\}$, $\Upsilon_{\text{mem}} = \{S^{(2)}\}$. The rules of the transducer are:

$$A_{\text{snd}}(\mathtt{u}) \leftarrow \mathtt{Id}(\mathtt{u}).$$

$$B_{\text{snd}}(\mathtt{u},\mathtt{v}) \leftarrow A(\mathtt{u}),\ \mathtt{Id}(\mathtt{v}).$$
$$B_{\text{snd}}(\mathtt{u},\mathtt{v}) \leftarrow B(\mathtt{u},\mathtt{v}).$$

$$S_{\text{ins}}(\mathtt{u},\mathtt{v}) \leftarrow B(\mathtt{u},\mathtt{v}).$$

$$\mathtt{missing}(\,) \leftarrow \mathtt{All}(\mathtt{u}),\ \mathtt{All}(\mathtt{v}),\ \mathtt{u} \neq \mathtt{v},\ \neg S(\mathtt{u},\mathtt{v}).$$
$$T_{\text{out}}(\mathtt{u}) \leftarrow R(\mathtt{u}),\ \neg\mathtt{missing}(\,).$$

□

Testing network-independence for FO-transducers is undecidable (see [16] for the proof).

3.4.6 Preliminary Observations

We now give several preliminary results about expressing queries with transducers, that are important for deriving later results.

First, we present two lemmas which show that at each node a transducer can always assemble a local copy of all input facts available on the network.

Lemma 3.7. Let $\mathcal{D}$ be a database schema. There is a transducer schema Υ with $\Upsilon_{\text{in}} = \mathcal{D}$ and an oblivious, inflationary, monotone UCQ-transducer Π over Υ such that for every transducer network for Π, for every instance I over $\mathcal{D}$, on every horizontal partition of I, every fair run reaches a configuration where every node has a local copy of the entire instance I in its memory.

Proof. Because the construction is straightforward, we only provide a sketch. The idea is to let all nodes send out their local input facts and forward any message they receive. The local inputs, together with the received inputs, are accumulated in local memory relations. In any fair run, eventually all nodes will have received all input facts. Relations `Id` and `All` are not needed. We also do not need deletions on the memory relations. This technique has already been illustrated by Example 3.4. □

We can refine the technique of Lemma 3.7 to let a node know *when* it has collected every input fact in memory:

Lemma 3.8. Let $\mathcal{D}$ be a database schema. There is a transducer schema Υ with $\Upsilon_{\text{in}} = \mathcal{D}$ and an UCQ$^{\neg}$-transducer Π over Υ such that for every transducer network for Π, for every instance I over $\mathcal{D}$, on every horizontal partition of I, every fair run reaches a configuration where every node has a local copy of the entire instance I in its memory, and an additional flag '`ready`' is true (implemented by a nullary memory relation). Moreover, the flag '`ready`' does not become true at a node before that node has the entire instance I in its memory.

The transducer Π is not oblivious, but can be made inflationary when using locally the language NrDatalog$^{\neg}$ instead of UCQ$^{\neg}$.[2]

Proof. We provide a sketch; see [16] for the full construction. The idea is that a node x will send its local input facts over relation $R^{(k)} \in \mathcal{D}$ to every other node, with an additional last component that contains the identifier of x, to indicate the origin of the fact. We call this last component the "tag". Next, when a node y receives a tagged input fact, it removes the tag and stores the fact in its memory. This already lets each node incrementally accumulate all inputs across the network. Now, for each fact that y receives from x, node y also sends an acknowledgment back to x. The node x checks whether y has (eventually) acknowledged all the input facts of x. If yes, then x sends out $\texttt{done}(x, y)$. From the viewpoint of y, if y has received $\texttt{done}(x, y)$ from all other nodes x then it knows that it has accumulated all the input facts on the network, and the `ready`-flag is created at y. The relations `Id` and `All` are used heavily in this protocol. □

The following theorem indicates that our transducer model has enough expressive power to study queries in the distributed context:

Theorem 3.9. Let $\mathcal{L}$ be a language containing UCQ$^{\neg}$. Then every query expressible in $\mathcal{L}$ can be distributedly computed by an $\mathcal{L}$-transducer. In particular, if $\mathcal{L}$ is a computationally complete query language, every partial computable query can be distributedly computed by an $\mathcal{L}$-transducer.

Proof. Let $\mathcal{Q}$ be a query expressible in $\mathcal{L}$. Let $\mathcal{D}$ and $\mathcal{D}'$ be respectively the input and output schema of $\mathcal{Q}$. We construct an $\mathcal{L}$-transducer Π to compute $\mathcal{Q}$ in two steps. In the first step, we use the partial specification of Π from Lemma 3.8 to obtain the entire input instance at every node. The language UCQ$^{\neg}$ suffices for this. This transducer has input schema $\Upsilon_{\text{in}} = \mathcal{D}$ but does not produce any output yet. Now, in the second step, we set $\Upsilon_{\text{out}} = \mathcal{D}'$. And because $\mathcal{Q}$ is expressible in $\mathcal{L}$, once the flag `ready` becomes true, we can output $\mathcal{Q}$ in the next local transition, by implementing for each output relation an $\mathcal{L}$-query that reads only the collected input facts. □

In the context of the CALM conjecture, monotone queries will play an important role. For now, we observe that oblivious transducers are sufficient to compute them:

Theorem 3.10. Let $\mathcal{L}$ be a query language containing UCQ. Then every *monotone* query expressible in $\mathcal{L}$ can be distributedly computed by an *oblivious* $\mathcal{L}$-transducer.

[2]This is because a NrDatalog$^{\neg}$ program allows auxiliary relations to be declared, to which negation can be applied.

In particular, if $\mathcal{L}$ is computationally complete, every partial computable monotone query can be distributedly computed by an oblivious $\mathcal{L}$-transducer. Moreover, these oblivious transducers can be made inflationary and monotone.

Proof. Let $\mathcal{Q}$ be a monotone query expressible in $\mathcal{L}$. The idea is the same as in the proof of Theorem 3.9, but we now use the oblivious, inflationary, monotone transducer from Lemma 3.7, to let every node gradually collect all inputs facts available on the network. Now, because $\mathcal{Q}$ is expressible in $\mathcal{L}$, in every local transition we can execute $\mathcal{L}$-queries for the output relations that read the part of the input already accumulated in memory. Since $\mathcal{Q}$ is monotone, no incorrect tuples are output this way. Eventually, all nodes have accumulated all the input across the network, and no new outputs will be produced. □

3.5 The CALM Conjecture

The following was conjectured by Hellerstein:

Conjecture 3.11 (CALM Conjecture [37])**.** A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.

Before we can rigorously investigate this conjecture, we want to formalize the notion of "coordination-freeness". This is presented in Section 3.5.1. Next, we will present our formal CALM conjecture and its associated results in Section 3.5.2. Additional results are in Section 3.5.3.

3.5.1 Coordination-free

The CALM conjecture hinges on an intuitive notion of "coordination" of certain distributed computations. We illustrate this notion with a few examples.

In the well-known two-phase commit protocol [33], each node is responsible for executing some part of a distributed transaction. To keep the distributed database consistent in the face of runtime crashes, either all parts should be committed or none is. To this purpose, after executing its part of the distributed transaction, but before actually committing the results, a node checks that *every* node can commit its results. This way, the distributed commit can proceed only if all individual nodes can commit. Naturally, the nodes have to exchange messages to determine if they can commit or not.

As another example, the multicast protocol of Lemma 3.8 also relies on heavy coordination: the nodes exchange many messages, including acknowledgments, before they all obtain the flag '`ready`'.

Generalizing both examples, the main idea behind coordination is that a large set of nodes needs to obtain a consensus. For two-phase commit this is the global decision whether all nodes should commit or not, and for Lemma 3.8 the consensus is that all nodes have the same data. Reaching a consensus is known to be difficult in the distributed context [23]. Because of the complexity of consensus, the involved nodes sometimes have to wait relatively long before continuing with the actual computation. This is also called "global barrier" [37].

It should be clear that coordination typically decreases the efficiency of distributed computations, because while the coordination is under way, the nodes are just waiting. So, it seems useful to understand precisely when coordination can be avoided, for which we will use the term "coordination-freeness". This is what the CALM conjecture is all about. It seems hard to give a definitive formalization of coordination-freeness. Still, we offer here a nontrivial definition that appears interesting. A very drastic, too drastic, definition of coordination-free would be that the full output can always be computed without communication, regardless of the input partition. Our definition is much less severe and only requires that the computation can succeed without communication on "suitable" horizontal partitions. It actually does not matter what a suitable partition is, as long as it exists.

Formally, let Π be a transducer over a schema Υ. Let $\boldsymbol{\mathcal{N}}$ be a transducer network for Π. We call $\boldsymbol{\mathcal{N}}$ *coordination-free* if individually for every database instance I over Υ_{in}, there exists a horizontal partition H of I and a run of $\boldsymbol{\mathcal{N}}$ on H in which a quiescence configuration is already reached by performing only heartbeat transitions (zero or more). Intuitively, if the horizontal partition is right, then no communication is required to correctly compute the query. The property of coordination-freeness is mainly interesting for consistent transducer networks, because then at the quiescence configuration that was reached with only heartbeat transitions, the produced output is the same as produced by any other fair run. We call transducer Π *coordination-free* if for every network its corresponding transducer network is coordination-free.

Example 3.12. Consider again the transitive closure computation from Example 3.4. When every node has the full input, they can each individually compute the transitive closure with only heartbeats. Hence, this transducer is coordination-free. □

The transitive closure query is monotone and this example can actually be generalized in the following proposition. This proposition is implicit in the literature on embarrasingly parallel computation [37, 44, 49], and our main result (Theorem 3.21) will provide a converse to it.

Proposition 3.13. Let $\mathcal{L}$ be a query language containing UCQ. Every monotone query $\mathcal{Q}$ expressible in $\mathcal{L}$ can be distributedly computed by a coordination-free $\mathcal{L}$-transducer.

Proof. Recall from the proof of Theorem 3.10 that there is an oblivious $\mathcal{L}$-transducer that distributedly computes $\mathcal{Q}$. Using the same intuition as in Example 3.12, this transducer is coordination-free. □

Note, this proposition would not hold under the drastic definition of coordination-freeness from above.

The reader should not be lulled into believing that with a coordination-free program it is always sufficient to give the full input at all nodes, as the following example shows:

Example 3.14. Consider the following query $\mathcal{Q}$, having as input two nullary relations A and B, and a nullary output relation T: create the non-empty output (representing "true") if at least one of A and B is nonempty. This query is monotone. Consider the following (contrived) transducer Π to compute $\mathcal{Q}$. If the network has only one node (which can be tested by looking at the relation `All`), the transducer simply outputs the answer to the query. Otherwise, it first tests if its local input fragments of A and B are *both* nonempty. If this is the case, nothing is output locally yet, but a nullary fact C is sent out. Any node that receives the message C will output it. When precisely one of A and B is nonempty locally, the transducer simply outputs the correct output directly. The transducer is network-independent. Also, the transducer is coordination-free, because on networks with at least two nodes there always is a partition of the data under which no node has both A and B locally nonempty, and the query can be computed without communication. Moreover, when A and B are both nonempty, and every node has the entire input, no run will reach a quiescence configuration without communication. □

The following two examples show that network-independence for a transducer does not guarantee coordination-freeness, and vice versa.

Example 3.15. We provide an example of a transducer that is network-independent but not coordination-free, i.e., requires communication. Let $\mathcal{Q}$ be the following "emptiness" query, having a nullary input relation R, and a nullary output relation T: create the non-empty output (representing "true") iff R is empty. This query is nonmonotone. We now describe a transducer to distributedly compute $\mathcal{Q}$. Since every node can have a part of the input, the nodes coordinate with each other to be certain that R is empty at every node. Every node sends out its identifier (using the relation `Id`) on condition that its local relation R is empty. Received messages are forwarded, so that if R is globally empty, eventually all nodes will have received the identifiers of all nodes, which can be checked using the relation `All`. When this happens, the transducer at each node outputs a nullary fact.

For completeness, we specify an FO-transducer Π to implement this idea. The transducer schema Υ is as follows: $\Upsilon_{\text{in}} = \{R^{(0)}\}$, $\Upsilon_{\text{out}} = \{T^{(0)}\}$, $\Upsilon_{\text{msg}} = \{U^{(1)}\}$ and $\Upsilon_{\text{mem}} = \{S^{(0)}\}$. The rules are:

$$U_{\text{snd}}(\mathtt{u}) \leftarrow \mathtt{Id}(\mathtt{u}),\ \neg R(\,).$$
$$U_{\text{snd}}(\mathtt{u}) \leftarrow U(\mathtt{u}).$$

$$S_{\text{ins}}(\mathtt{u}) \leftarrow \mathtt{Id}(\mathtt{u}),\ \neg R(\,).$$
$$S_{\text{ins}}(\mathtt{u}) \leftarrow U(\mathtt{u}).$$

$$\mathtt{missing}(\,) \leftarrow \mathtt{All}(\mathtt{u}),\ \neg S(\mathtt{u}).$$
$$T_{\text{out}}(\,) \leftarrow \neg\mathtt{missing}(\,).$$

□

Example 3.16. We give a transducer that is coordination-free, and that is consistent on every network, but is not network-independent. The transducer has two unary input relations R and S, and it has a unary output relation T. Using relations `Id` and `All`, the transducer can detect if there is only one node, or if there are more nodes. If there is just one node, the single node outputs the union of R and S. If there are at least two nodes, then all nodes will copy their local inputs into their memory; they also broadcast their input facts to each other, so that all nodes accumulate all inputs of the network; and, the nodes will continuously output the intersection of the accumulated R-facts with the accumulated S-facts.

First, we see that on each network this transducer is consistent. Indeed, on a single-node network the union of R and S is output, and on a multi-node network the intersection of R and S is output. This different output behavior prevents the transducer from being network-independent. Finally, the transducer is coordination-free because on a single-node network the output is computed with only heartbeats, and on a multi-node network we can consider the partition where each node has the entire input, and then the intersection of R and S can already be computed with only heartbeats. $\square$

Coordination-freeness seems a useful property for a transducer to have. However, it cannot be decided automatically in general:

Proposition 3.17. Coordination-freeness is undecidable for FO-transducers.

Proof. We reduce the finite satisfiability problem for FO to deciding coordination-freeness for FO-transducers. Let φ be an FO-sentence over a database schema $\mathcal{D}$. We construct an FO-transducer Π that is coordination-free iff φ is *not* finitely satisfiable.

Consider the transducer Π in Example 3.15, that is over schema Υ. We may assume without loss of generality that the relation names of Υ do not occur in $\mathcal{D}$. We obtain a new transducer schema Υ' from Υ by adding $\mathcal{D}$ to Υ_{in}; by adding new message relations $\{(C^{\text{msg}}, k) \mid C^{(k)} \in \mathcal{D}\}$; and, by adding new memory relations $\{(C^{\text{mem}}, k) \mid C^{(k)} \in \mathcal{D}\}$. We obtain a new transducer Π' over Υ' by modifying Π to let all nodes gradually accumulate all input facts by means of message forwarding. Moreover, besides keeping the old output condition "$\neg$`missing`", we will only produce an output if additionally φ is satisfied on the accumulated $\mathcal{D}$-facts so far (in memory).

For the first direction, suppose that φ is finitely satisfiable on a database instance I over $\mathcal{D}$. We show that Π' is not coordination-free. We can regard I as a database instance over Υ'_{in}, where relation R is empty. Let $\mathcal{N}$ be a network containing two nodes x and y. Let $\boldsymbol{\mathcal{N}}$ denote the transducer network based on $\mathcal{N}$ and Π'. Suppose there is some horizontal partition H of I over $\mathcal{N}$ and a run $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on input H in which a first quiescence configuration is already reached by doing only heartbeat transitions. Because I does not contain $R(\,)$, the nodes send the messages $U(x)$ and $U(y)$. Because of fairness, these messages must be delivered to y and x respectively, which can happen only after the first quiescence point because before the quiescence point there are only heartbeat transitions. Eventually, every node will find $\neg$`missing`$(\,)$ to be true. The same reasoning can be applied to the relations of $\mathcal{D}$: whether I is empty or not, there must be a configuration after the first quiescence point, in which all nodes have accumulated I in the memory relations. Then φ also becomes true, and thus we know that every node eventually outputs $T()$. Note that this fact cannot be in the first quiescence configuration because it requires the delivery of at least one of the messages $U(x)$ or $U(y)$. So, the initial quiescence configuration that was reachable by only heartbeat transitions cannot exist. Thus, the network $\mathcal{N}$ and input I are a proof that Π' is not coordination-free.

For the other direction, suppose that φ is not finitely satisfiable. Then no transducer network based on Π' can produce output, no matter what the input instance over Υ'_{in} or horizontal partition of that instance is. Hence, the start configuration of every run is already a quiescence configuration, and Π' is coordination-free. $\square$

Although coordination-freeness is undecidable for FO-transducers (and by extension more powerful transducers), we can identify a syntactic class of transducers that is guaranteed to be coordination-free, and that will prove to have the same expressive power as the class of coordination-free transducers. Importantly, the syntactic restriction does not guarantee network-independence. Recall from Section 3.4.1 that an *oblivious* transducer does not read the system relations `Id` and `All`. For now we observe:

Proposition 3.18. Let $\mathcal{L}$ be a query language. Every network-independent, oblivious $\mathcal{L}$-transducer is coordination-free.

Proof. Let Π be a network-independent, oblivious $\mathcal{L}$-transducer over a schema Υ. Let $\mathcal{Q}$ be the query distributedly computed by Π.

First, on a single-node network, the single node is always given the entire input and there can only be heartbeat transitions. Then, for an input instance I over Υ_{in}, a quiescence configuration containing $\mathcal{Q}(I)$ is always reached by doing only heartbeat transitions.

Now consider any other network $\mathcal{N}$, any instance I over Υ_{in}, and the horizontal partition H that places the entire instance I at every node. Since Π is oblivious, nodes cannot detect that they are on a network with multiple nodes unless they receive a message. So, by doing only heartbeat transitions initially, every node will act the same as if in a single-node network and will already output the entire $\mathcal{Q}(I)$. Because Π is network-independent, the nodes cannot output more than $\mathcal{Q}(I)$ when they receive messages afterwards. □

3.5.2 Main Results

Now we can formalize the original Conjecture 3.11. We will take the terms "program" and "to have an execution strategy" to mean "query" and "to be distributedly computed by a transducer", respectively. The term "eventually consistent" is then formalized by our notions of consistency and network-independence. Under this interpretation, the conjecture becomes:

Conjecture 3.19. A query can be distributedly computed by a coordination-free transducer if and only if it is expressible in Datalog.

Let us immediately get the if-side of this conjecture out of the way. It holds, because a query in Datalog is monotone, and then by Theorem 3.10 there exists an oblivious transducer to compute the query, but we have seen in Proposition 3.18 that oblivious transducers are coordination-free.

As to the only-if side, the explicit mention of Datalog is a bit of a nuisance because Datalog is limited to polynomial time whereas there certainly are monotone queries outside PTIME. We also mention the celebrated paper [8] where Afrati, Cosmadakis and Yannakakis show that even within PTIME there exist queries that are monotone but not expressible in Datalog.

But Datalog aside, however, the true emphasis of the CALM Conjecture clearly lies in the monotonicity aspect. Indeed, we confirm it in this sense:

Theorem 3.20. Let $\mathcal{L}$ be a query language. Every query that is distributedly computed by a coordination-free $\mathcal{L}$-transducer is monotone.

Proof. Let Π be a coordination-free $\mathcal{L}$-transducer over a schema Υ that distributedly computes a query $\mathcal{Q}$. Let I and J be two database instances over the schema Υ_{in} such that $I \subseteq J$. We must show that $\mathcal{Q}(I) \subseteq \mathcal{Q}(J)$. Consider a fact $\boldsymbol{f} \in \mathcal{Q}(I)$. Consider a network $\mathcal{N}$ with at least two nodes. Let $\boldsymbol{\mathcal{N}}$ denote the transducer network based on $\mathcal{N}$ and Π. Since Π is coordination-free and network-independent, there exists a horizontal partition H of I and a run $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on input H in which a quiescence configuration, containing the facts $\mathcal{Q}(I)$, is already reached by letting the nodes do only heartbeat transitions. Let x be a node where $\boldsymbol{f}$ is output in the quiescence configuration. Let y be a node different from x and consider a horizontal partition H' of J where $H'(x) = H(x)$ and $H'(y) = J$. Let n be the number of initial heartbeat transitions of x in run $\mathcal{R}$ that were needed to output $\boldsymbol{f}$ at x. Consider a prefix of a run of $\boldsymbol{\mathcal{N}}$ on input H' in which we initially do n heartbeat transitions, all with active node x. Because local transitions are deterministic, the node x goes through the same state changes as in run $\mathcal{R}$ before $\boldsymbol{f}$ is output and therefore $\boldsymbol{f}$ is output again in this prefix. The prefix can be extended to a full fair run $\mathcal{R}'$ of $\boldsymbol{\mathcal{N}}$ on input H'. Since $\boldsymbol{\mathcal{N}}$ is consistent, the fact $\boldsymbol{f}$ will be output on any partition of J, during any fair run. Hence, $\boldsymbol{f}$ belongs to the query computed by $\boldsymbol{\mathcal{N}}$ applied to J. Moreover, Π is network-independent, so $\boldsymbol{f}$ belongs to $\mathcal{Q}(J)$. □

We can now obtain the following result:

Theorem 3.21. Let $\mathcal{L}$ be a query language containing UCQ. For every query $\mathcal{Q}$ that is expressible in $\mathcal{L}$, the following are equivalent:

1. $\mathcal{Q}$ can be distributedly computed by a coordination-free $\mathcal{L}$-transducer;
2. $\mathcal{Q}$ can be distributedly computed by an oblivious $\mathcal{L}$-transducer; and,
3. $\mathcal{Q}$ is monotone.

Proof. Theorem 3.10 yields *(3) ⇒ (2)*; Proposition 3.18 yields *(2) ⇒ (1)*; Theorem 3.20 yields *(1) ⇒ (3)*. □

In particular, if $\mathcal{L}$ is computationally complete, then the previous equivalences hold for any computable query. As a small remark, now it is of no surprise that Example 3.15 required coordination; indeed, there we distributedly compute a non-monotone query.

3.5.2.1 Discussion

In practice, Theorem 3.21 can be used as follows. Essentially, by restricting a language, its execution can in general be optimized more thoroughly than the unrestricted language. A well-known example is SQL versus a Turing-complete programming language. For our situation, the programmer of a distributed (query) algorithm can write a program in a high-level declarative formalism, like our transducer model, or a Datalog-variant like e.g. [44, 14, 1]. Suppose the query is monotone. Then by Theorem 3.21, the query can be implemented in a coordination-free manner. Moreover, we can prevent the programmer from abusing coordination using the syntactic restriction of obliviousness. The main idea is that the programmer is given only a few communication primitives, like sending a message to neighboring nodes, and a syntactic restriction is imposed to prevent the programmer from using network relations like Id or All (or equivalent information). Next, the programmer, or a software tool, needs to assert that the program is network-independent, i.e., on every network, all fair runs produce the desired outcome. Then, using Theorem 3.21, if the runtime is told that the program is oblivious and network-independent, the runtime can execute the program without any coordination. By contrast, if the programmer uses Id and All, then this semantic property is no longer guaranteed, and one would have to resort to a general execution strategy that has built-in coordination, which seems a waste if the program expresses a monotone computation. This way, obliviousness could be a useful guiding principle for distributed query evaluation. The works of Loo et al. [44] and Nigam et al. [49] provide coordination-free distributed execution engines for Datalog.

3.5.3 Further Results

It is natural to wonder about variations of our model. One question may be about the system relations Id and All. Without them (the oblivious case), we know that we are always coordination-free and thus monotone.

What if we would read precisely one system relation; only Id or only All? As to coordination-freeness, the argument given in the proof of Proposition 3.18 still works when the transducer reads only Id, because then nodes still cannot detect that they are on a network with multiple nodes. However, the argument fails when the transducer reads only All, and indeed we have the following counterexample.

Example 3.22. We describe a transducer that is network-independent, reads only All, but that is not coordination-free. The query expressed is simply the identity query on a unary relation R. The transducer can observe the difference between a single-node and a multi-node network by looking at the relation All. If it is a single-node network, the node simply outputs the result directly. If it is a multi-node network, every node sends out a message. Only upon receiving a message will a node then output the result. Thus on a multi-node network, regardless of the horizontal partition, communication is needed for the transducer network to produce the required output. □

So, coordination-freeness is not guaranteed when reading only All, but yet, monotonicity is not lost.

Theorem 3.23. Let $\mathcal{L}$ be a query language. Every query distributedly computed by an $\mathcal{L}$-transducer that reads only relation All, is monotone.

Proof. Let Π be a network-independent transducer that reads only All. As a technical convenience, we assume that runs can use *concurrent* global transitions, in which multiple nodes can be active at the same time, each receiving messages from their own message buffer. At the end of such a concurrent global transition, for each node, its message buffer is extended with the multiset union of all messages sent to it by its neighbors. These concurrent transitions can be simulated by a sequence of ordinary single-node transitions, as remarked at the end of Section 2.5.2.

Let Υ be the schema of Π. Let $\mathcal{Q}$ be the query distributedly computed by Π. Let I and J be two database instances over Υ_{in} such that $I \subseteq J$. Let $\boldsymbol{f} \in \mathcal{Q}(I)$. We have to show that $\boldsymbol{f} \in \mathcal{Q}(J)$. The main trick used in this proof is that although Π can count the number of nodes of a network (using relation `All`), it cannot directly observe the *edges* of the network. So, when $\boldsymbol{f}$ is output on input I in one network, we can fool the transducer to output $\boldsymbol{f}$ on input J in another network that has only slightly different edges.

Run on I Consider a network $\mathcal{N}_1$ in the form of a ring, containing at least four nodes. See Figure 3.1 for an example. Let $\boldsymbol{\mathcal{N}}$ denote the transducer network based on $\mathcal{N}_1$ and Π. Let H_1 be the horizontal partition of I that places I on every node of $\mathcal{N}_1$.

We show now that there exists a run $\mathcal{R}_1$ of $\boldsymbol{\mathcal{N}}$ on input H_1 with sequence of configurations $\rho_1 = (s_1, b_1)$, $\rho_2 = (s_2, b_2)$, …, such that for each $i \geq 1$ and each $x, y \in \mathit{nodes}(\mathcal{N}_1)$ we have $s_i(x) = s_i(y)$ and $b_i(x) = b_i(y)$. In words: in every configuration, all nodes have the same transducer state and the same message buffer. We inductively construct $\mathcal{R}_1$. For the base case ($i = 1$), configuration ρ_1 satisfies the property because it is the start configuration: all nodes are given the entire input I, and all message buffers are empty. For the induction hypothesis, assume that the property holds for i. For the inductive step, we show how to continue the partially constructed run $\mathcal{R}_1$ so that the property holds for $i + 1$. Denote $m = b_i(x)$ for some node x. Possibly $m = \emptyset$. We next do a concurrent global transition in which *each* node is the recipient of delivered message multiset m. This is possible by using the induction hypothesis. So, we are delivering the entire message buffer at once to each node. Again by the induction hypothesis, all nodes have the same state in configuration ρ_i, and since local transitions are deterministic, all nodes will have the same state in configuration ρ_{i+1}. Also, if one node sends a message set J_{snd} on delivery of m, then all nodes will send this set on delivery of m. Hence, because $\mathcal{N}_1$ is a ring, for each node, the messages of J_{snd} will have been added *twice* to its message buffer at the end of the concurrent global transition. Since all nodes emptied their message buffer at the beginning of the concurrent transition, we see that in ρ_{i+1} the nodes again have the same message buffer.

The run $\mathcal{R}_1$ can be converted to a fair run $\mathcal{R}'_1$ with only non-concurrent global transitions and that produces the same output as $\mathcal{R}_1$. Moreover, because Π is network-independent, we know that $\mathcal{R}'_1$ outputs $\mathcal{Q}(I)$, and thus $\mathcal{R}_1$ outputs $\mathcal{Q}(I)$. Therefore, we can consider a node u of $\mathcal{N}_1$ and an index $k \geq 1$ such that u outputs $\boldsymbol{f}$ during the k^{th} concurrent global transition of $\mathcal{R}_1$.

Run on J Let u be the node as previously defined. Let z be a node of $\mathcal{N}_1$ that is not a neighbor of u. We obtain a new network $\mathcal{N}_2$ from $\mathcal{N}_1$ by adding an edge between the two neighbors of z. Because $\mathcal{N}_1$ is a ring with at least four nodes, we know that this edge was not previously there and thus $\mathcal{N}_2$ contains a smaller ring without node z. Let $\boldsymbol{\mathcal{N}}'$ denote the transducer network based on $\mathcal{N}_2$ and Π. Importantly, note that $\mathcal{N}_1$ and $\mathcal{N}_2$ have precisely the same nodes. Let H_2 be the horizontal partition of J that places I on every node except z and that places $J \setminus I$ on z.

Let us abbreviate $N = \mathit{nodes}(\mathcal{N}_2) \setminus \{z\}$. Recall the sequence of configurations ρ_1, ρ_2, …, of run $\mathcal{R}_1$ from above. We now show that there exists an (unfair) run $\mathcal{R}_2$ of $\boldsymbol{\mathcal{N}}'$ on input H_2 with sequence of configurations $\rho'_1 = (s'_1, b'_1)$, $\rho'_2 = (s'_2, b'_2)$, …, such that for each $i \geq 1$ and each $y \in N$ we have $s'_i(y) = s_i(y)$ and $b'_i(y) = b_i(y)$. In words: the

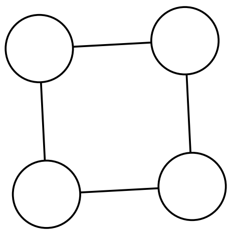

Figure 3.1: Ring network topology

smaller ring of nodes N follows exactly the states and message buffers of run $\mathcal{R}_1$. We inductively construct $\mathcal{R}_2$. For the base case ($i = 1$), the property is satisfied because input partition H_2 initializes the nodes of N in the same way as input partition H_1. For the induction hypothesis, we assume that the property holds for index i. For the inductive step, we show that the property holds for index $i+1$. As in the construction of $\mathcal{R}_1$, we next do a concurrent global transition in which we deliver to every node of N the contents of its entire message buffer. Using the induction hypothesis, this causes each node of N to send the same message instance J_{snd} to their neighbors. This message instance was also sent during the corresponding global transition of $\mathcal{R}_1$. Let y_1 and y_2 denote the two neighbors of node z in $\mathcal{N}_1$. We have $\{y_1, y_2\} \subseteq N$. Because we have added the extra edge between y_1 and y_2 in $\mathcal{N}_2$, node y_1 sends J_{snd} to z *and* to y_2. This is similar for y_2. Node z does not send anything because it is ignored. So, like in $\mathcal{R}_1$, both y_1 and y_2 have J_{snd} added precisely twice to their message buffer at the end of the concurrent global transition. The rest of the reasoning is the same as in the inductive step for constructing $\mathcal{R}_1$. We obtain that the nodes of N have the same state and message buffers in configuration ρ'_{i+1} as in configuration ρ_{i+1}.

Consider again the run $\mathcal{R}_2$. Because $u \in N$, the fact $\boldsymbol{f}$ is eventually output at u during $\mathcal{R}_2$, during some global transition k. But $\mathcal{R}_2$ is clearly not fair because the node z is ignored. However, we can make a new run $\mathcal{R}'_2$ by copying only the first k global transitions of $\mathcal{R}_2$, converting each of them to a sequence of ordinary (non-concurrent) global transitions and then extending this prefix arbitrarily to a full fair run. Thus, we obtain that $\boldsymbol{f}$ is output in a fair run of $\boldsymbol{\mathcal{N}}'$ on input H_2. Since Π is network-independent, we obtain that $\boldsymbol{f} \in \mathcal{Q}(J)$, as desired. □

As a corollary, we can add two more statements to the three equivalent statements of Theorem 3.21:

Corollary 3.24. Let $\mathcal{L}$ be a query language containing UCQ. The following statements are equivalent for any query $\mathcal{Q}$ expressible in $\mathcal{L}$:

1. $\mathcal{Q}$ can be distributedly computed by an oblivious $\mathcal{L}$-transducer;
2. $\mathcal{Q}$ can be distributedly computed by an $\mathcal{L}$-transducer that is given only `Id`; and,
3. $\mathcal{Q}$ can be distributedly computed by an $\mathcal{L}$-transducer that is given only `All`.

Proof. The directions *(1) ⇒ (2)* and *(1) ⇒ (3)* are immediate because an oblivious transducer is given neither of `Id` or `All`. For *(2) ⇒ (1)*, when only `Id` is read, the query $\mathcal{Q}$ is monotone as argued above. Then, by also using that $\mathcal{Q}$ is expressible in $\mathcal{L}$, we can apply Theorem 3.10 to know that $\mathcal{Q}$ is computable by an oblivious $\mathcal{L}$-transducer. The direction *(3)* $\Rightarrow$ *(1)* is similar, but now we use Theorem 3.23. $\square$

To conclude this section, we note that distributed algorithms involving a form of coordination typically require the participating nodes to have some knowledge about the other participating nodes [23]. This justifies our modeling of this knowledge in the form of the system relations `Id` and `All`. Importantly, we have shown that these relations are only necessary if one wants to compute a nonmonotone query in a distributed fashion.

3.6 Expressiveness Analysis

In this section we want to better understand the transducer model itself. The main question we would like to address is how the transducer model can be combined with a local query language to express a global query. It is not obvious what peculiarities of the model can be exploited in the local queries, and how. It will turn out actually that the global query language expressed by the transducer is the while-closure of its local query language. Intuitively, this is because each node can do multiple local transitions in a run, which can be seen as iterations of an implicit while-loop. This is very natural, and we believe this shows that our (distributed) transducer model is relatively elegant, because it respects previous results about well-known query languages [2].

Table 3.1 summarizes the expressiveness results.

Queries expressible in While = queries computable by FO-transducers = queries computable by UCQ$^{\neg}$-transducers
Monotone queries expressible in While = queries computable by oblivious FO-transducers
Queries expressible in Datalog = queries computable by inflationary NrDatalog-transducers
Queries in PSPACE = queries computable by multi-node FO-transducer networks under 1-delivery semantics (cf. Section 2.5.4)

Table 3.1: Expressiveness summary

3.6.1 While versus FO

We first show the following property, and although the result might not sound very surprising, writing out the details turned out to be rather intricate.

Lemma 3.25. A query is expressible in While if and only if it is computable by an FO-transducer on a single-node network.

proof (sketch). We sketch the main ideas of the proof (see [16] for the full details). For the only-if direction, we have to simulate a While-program on a single-node FO-transducer network. A While-program can be simulated by iterated heartbeats using well-known techniques [4]. The main idea is that the loops in the while-program are rewritten with explicit "goto" statements. The statements of this rewritten program can then be simulated by an FO-transducer that keeps track of which statement is to be executed next, and goto-statements can make the simulation jump back to a previous statement (simulating a loop).

For the if-direction, let Π be an FO-transducer over a schema Υ that computes a query $\mathcal{Q}$ on a single-node network $\boldsymbol{\mathcal{N}}$. A While-program that computes the query $\mathcal{Q}$ has to use exactly the same input and output schema as Π, namely, Υ_{in} and Υ_{out} respectively. The While-program is however allowed to declare any number of temporary relations. We may assume that Π does not read message relations in its internal queries, because no messages can be received on a single-node network. As a first case, let us additionally assume that the internal FO-queries of Π do not read relations `Id` and `All` (the oblivious case). Now, because the memory relations of Π start empty, and temporary relations declared in the While-program also start empty, we can easily construct a While-program P that consists of one big loop, of which one iteration performs the same state changes as Π during one heartbeat transition. In order to terminate, P must detect repetition of transducer states, because this implies that Π has repeated a state and will output no new output facts. Detecting such a repetition is possible by using the technique of Abiteboul and Simon [3].

Let us now consider the second case where Π reads `Id` or `All` (or both) in its internal queries. These relations can not be simulated by the While-program. Indeed, these relations are always non-empty from the perspective of Π, and a While-program can not create temporary relations to represent them: indeed, when the input is empty, the While-program can not invent a value to store in `Id` and `All`, and when the input is nonempty, the While-program can in general not choose one value to store in `Id` and `All`. Therefore, we will first eliminate the use of `Id` and `All` from the queries of Π. Once this is done, we can apply the above translation for the oblivious case.

Remove relation `All` Note that in the FO-queries of Π we can replace the use of relation `All` by `Id` because, on a single-node network, both relations have the same contents. Formally, in a transducer state there is a fact `Id`(a) iff there is a fact `All`(a).

Remove relation `Id` Assume that relation `All` is not used in Π. Next, we remove the use of relation `Id` from Π. We use the work of Van den Bussche and Cabibbo [55], who have shown how to convert an ordinary (untyped) FO-formula to a *typed* formula that computes the same query. In typed formulas, each variable is of a specific *sort*, meaning that it ranges over an isolated domain of values. In our case, we distinguish between two sorts: *(i)* values in the active domain of an input database instance over Υ_{in}; and, *(ii)* the identifier x of the single node in $\boldsymbol{\mathcal{N}}$ (with $\boldsymbol{\mathcal{N}}$ as defined above).[3] We will denote these sorts as respectively *adom* and *id*. A *type* τ is a tuple of sort symbols, like $(adom, id, id)$.

[3]By genericity of the queries in Π, we may assume that the node-identifier does not occur in the input, giving rise to these separated sorts.

Based on Π, we construct a second transducer Π^2 as follows. For each relation $R^{(k)} \in \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}$ of Π and each type τ of arity k, transducer Π^2 has a relation $R_\tau^{(k)}$. Transducer Π^2 also has a memory relation `Adom` in which it stores all values from its input. We now describe how Π^2 updates such a relation $R_\tau^{(k)}$. Let φ denote the FO-formula used by Π to insert tuples in relation R (deletion is similar). If $\tau = (adom, adom, \ldots, id)$ for instance, then transducer Π^2 will use a formula of the form

$$\psi(\mathtt{u}_1, \ldots, \mathtt{u}_k) \wedge \mathtt{Adom}(\mathtt{u}_1) \wedge \mathtt{Adom}(\mathtt{u}_2) \wedge \ldots \wedge \mathtt{Id}(\mathtt{u}_k)$$

to insert into R_τ the tuples of type τ that are computed by φ. The formula ψ is basically the formula φ, but modified to cope with the separation of tuples by their type: each time φ reads a tuple from a relation $S^{(l)}$, formula ψ reads a tuple from the union $\bigcup_{\tau\in\alpha} S_\tau^{(l)}$, where α is all types of arity l. This way, Π^2 also computes the same query as Π.

Now, we can apply Proposition 1 of [55] to the formulas in Π^2 to obtain new formulas in which there is no explicit reference to relations `Adom` and `Id`. Instead, the converted formulas use variables of two sorts (the *adom* and *id* sorts). In a last step, we can syntactically eliminate any reference to *id* variables, and obtain back normal FO-formulas. These can be used in a new transducer Π^3, which is oblivious, to compute the same query as Π. □

Now we can obtain the following result:

Theorem 3.26. A query is expressible in While if and only if it can be distributedly computed by an FO-transducer.

Proof. For the if-direction, let Π be an FO-transducer that distributedly computes a query $\mathcal{Q}$. Because Π is network-independent, the query $\mathcal{Q}$ must also be computed when executing Π on a single-node network. Then, by using Lemma 3.25, there is a While-program that computes $\mathcal{Q}$.

For the only-if direction, let $\mathcal{Q}$ be a query that can be computed by a While-program. We specify an FO-transducer to compute $\mathcal{Q}$ in two steps. First we use Lemma 3.8 to obtain the entire input instance at every node. Every node can then act as if it was alone, ignoring any further messages, and simulate the While-program again using Lemma 3.25. □

For monotone queries we have the following, more specific result:

Theorem 3.27. Every *monotone* query expressible in While can be distributedly computed by an *oblivious* FO-transducer.

Proof. Let $\mathcal{Q}$ be a monotone query expressible in While. We construct an oblivious FO-transducer to compute $\mathcal{Q}$. Note, Theorem 3.10 is not applicable, because it would give us an oblivious While-transducer, and not an oblivious FO-transducer. But the proof idea of the theorem can still be used.

First, we use the simple UCQ-protocol of Lemma 3.7 to let all nodes accumulate all input facts in memory. This does not require `Id` or `All`. Next, every time a node receives a new input fact, it starts or restarts a simulation of the While-program for $\mathcal{Q}$. The simulation uses the techniques of the proof of Lemma 3.25 (only-if direction), where specifically the output facts are first computed in temporary memory relations before being officially output. Checking whether a new input fact is received is done by comparing a received input fact with the previously accumulated input facts in memory. The restarting of the simulation of the While-program is done by emptying all memory relations, and restarting the program counter. The restart can happen at the moment a simulation is busy, in which case the temporary output is discarded. The restart can also happen after a simulation was already successfully ended. Since the query $\mathcal{Q}$ is monotone, no incorrect facts were output by previous simulations.

Eventually, every node will have accumulated all input facts, so the simulation can surely run to completion on all input facts. We also do not need relations `Id` and `All` to simulate the While-program. Hence, the transducer is oblivious. □

Note that the converse of Theorem 3.27, to the effect that every query distributedly computed by an oblivious FO-transducer is monotone and expressible in While, holds by combining Theorems 3.21 and 3.26 that give respectively the monotonicity of the query and the expressibility in While.

For our next result, we will use that FO is equivalent to NrDatalog$^\neg$ [2]. Basically, a program in NrDatalog$^\neg$ is a sequence of UCQ$^\neg$ statements. The following proposition shows that transducers can simulate this sequential composition of simpler statements:

Proposition 3.28.

(i) Every query that can be distributedly computed by an FO-transducer can be distributedly computed by an UCQ$^\neg$-transducer.

(ii) Every *monotone* query that can be distributedly computed by an FO-transducer can be distributedly computed by an *oblivious* UCQ$^\neg$-transducer.

Proof. First, we make a general observation. For every query $\mathcal{Q}$ that is distributedly computed by an FO-transducer, we can apply Theorem 3.26 to know that $\mathcal{Q}$ is expressible with a While-program P. Moreover, since the language FO is equivalent to NrDatalog$^\neg$ [2], every FO-statement in P can be replaced by a sequence of UCQ$^\neg$-statements, to obtain a new program P'. Then, it is clear that program P' can be simulated by an UCQ$^\neg$-transducer on a single-node network using iterated heartbeats, very similar to the proof of the only-if direction for Lemma 3.25.

For result *(i)*, we let each node first collect a local copy of the entire input by using the protocol of Lemma 3.8, which can be done with a UCQ$^\neg$-transducer. After collecting the input, each node can simulate the program P' is isolation.

For result *(ii)*, where $\mathcal{Q}$ is monotone, we use instead Lemma 3.7 to let each node gradually accumulate all input, and we restart the simulation of P' when new inputs arrive. □

3.6.2 Datalog versus NrDatalog

What if we are only interested in Datalog? Between the languages Datalog and NrDatalog, a similar relation exists as between While and FO:

Theorem 3.29. A query is expressible in Datalog if and only if it can be distributedly computed by an inflationary NrDatalog-transducer.

Proof. First we consider the only-if direction. We construct an oblivious, inflationary transducer to simulate a Datalog program. The basic idea is the same as in the proof of Theorem 3.10. The input tuples are sent out and accumulated on every node. During every transition, we apply the immediate consequence operator of the Datalog program [2], that can be expressed by NrDatalog. The relations `Id` and `All` are not needed, and the transducer can be made oblivious. Also, by the monotone nature of Datalog evaluation, deletions are never needed, and the transducer can be made inflationary.

Now we consider the if-direction. Let $\mathcal{Q}$ be a query distributedly computed by an inflationary NrDatalog-transducer Π over a schema Υ. We show that $\mathcal{Q}$ can be expressed in Datalog. Because of network-independence, it is sufficient to look at the behavior of Π on a single-node network. We simulate this behavior with a Datalog program P as follows. We assume that the logical "and" and the universal quantifier are not core primitives of FO, since these can be simulated by negation together with respectively the logical "or" and the existential quantifier. We call an FO-formula *positive* if each atom and existential quantifier occurs under an even number of negation symbols. Under this interpretation, the language NrDatalog is equivalent to positive FO. So, Π is just an inflationary FO-transducer, in which the internal FO-queries are positive. Now, the same transformation as in the proof of the if-direction for Lemma 3.25 can be applied to transform Π into a new FO-transducer Π' that computes $\mathcal{Q}$ without reading relations `Id` and `All`. Moreover, this transformation preserves the positivity of the formula. Hence, Π' can be immediately seen as an inflationary NrDatalog transducer that does not read `Id` and `All`. Next, we unite all NrDatalog rules of Π' in a Datalog program P. Because P by the nature of Datalog can only accumulate its generated facts, P has at least the opportunities of Π' to join facts, and P outputs at least the output of Π'. Moreover, because Π' is inflationary itself, Π' eventually has the same opportunities to join facts as P. In conclusion, P computes exactly the original query $\mathcal{Q}$. $\square$

It remains open if we can drop the word "inflationary" from Theorem 3.29.

3.6.3 Restrict Delivery

It is well-known that providing an order on the active domain increases the expressiveness of a query language [2]. This result transfers nicely to our transducer model. By guaranteeing that only one message is delivered during every global transition, referred to as 1-*delivery semantics* (cf. Section 2.5.4), an order can be established on each node:

Proposition 3.30. Under 1-delivery semantics, every PSPACE query can be computed by an FO-transducer network with at least two nodes.

Proof. In a network with at least two nodes, under 1-delivery semantics, each node can establish a linear order on the active domain by cooperating with the other nodes as follows. When a node has collected all inputs of the network (by means of Lemma 3.8), it sends out the elements of the active domain, that get forwarded by other nodes. Eventually, all these elements arrive back at the node, and the order can be established because at most one value is received at once. Then, each node can simulate a While-program on the collected input, that uses the established order.

The transducer involved is not truly network-independent, as this only works when there are at least two nodes. □

3.6.4 Specialized CALM Properties

Using our previous results about expressivity, we obtain the following variants of Theorem 3.21. Especially, the second variant, which deals with Datalog, may come closest to the CALM conjecture as originally imagined by Hellerstein [37].

Corollary 3.31. Within each of the following two groups, the statements are equivalent, for any query $\mathcal{Q}$:

1. (a) $\mathcal{Q}$ can be distributedly computed by a coordination-free FO-transducer.
   (b) $\mathcal{Q}$ can be distributedly computed by an oblivious FO-transducer.
   (c) $\mathcal{Q}$ is monotone and expressible in the language While.

2. (a) $\mathcal{Q}$ can be distributedly computed by a coordination-free, inflationary NrDatalog-transducer.
   (b) $\mathcal{Q}$ can be distributedly computed by an oblivious, inflationary NrDatalog-transducer.
   (c) $\mathcal{Q}$ is expressible in Datalog.

Proof. Regarding *(1)*, for (c) ⇒ (b) use Theorem 3.27; for (b) ⇒ (a) use Proposition 3.18; for (a) ⇒ (c) use Theorems 3.20 and 3.26 to obtain respectively the properties of "$\mathcal{Q}$ is monotone" and "expressible in the language While".

Regarding *(2)*, for (c) ⇒ (b) use (proof of if-direction in) Theorem 3.29; for (b) ⇒ (a) use Proposition 3.18; for (a) ⇒ (c) use Theorem 3.29. □

3.7 Addressing Transducers

Recall the *addressing transducer model*, defined in Section 2.4.2. For a homogeneous transducer network $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \Upsilon, \Pi)$ where Π is an addressing transducer, the operational semantics is also given by Section 2.5.2. As a special case, if $\mathcal{N}$ forms a complete graph, every node can send a message to every individual other node.

We now indicate which results obtained for epidemic transducers are transferable to addressing transducers, and which results are not. First, our previous results that do not explicitly restrict the use of `Id` or `All` still hold for addressing transducers because then addressing transducers and epidemic transducers are equivalent in terms of what queries they can compute. Indeed, the epidemic model can simulate the addressing model by manually adding an addressee-component to every message relation in the transducer schema, and by comparing for each received message the addressee component with the value in the local relation `Id`. The other direction is also possible, namely, that an addressing transducer can simulate an epidemic one: it suffices for the addressing transducer to send each message explicitly to *every* neighbor.

Interestingly, a notion of obliviousness can also be defined for addressing transducers. Formally, we say that an addressing transducer is *oblivious* if the relations `Id` and `All` are only used in the message sending queries.[4]

Now, most of our results involving oblivious epidemic transducers also hold for oblivious *addressing* transducers, because of the following reasons. First, the proof techniques frequently use that every node sends out its local input facts, and these are forwarded so that eventually all nodes accumulate all inputs. This can be done with an oblivious addressing transducer as well. Second, these results are mostly about *network-independent* transducers, and a frequently occurring idea in those proofs is that we only focus on the behavior of a single node: an oblivious epidemic transducer can not distinguish between a single-node network and a multi-node network unless it receives a message, so on a single-node network it should exhibit predictable behavior if it wants to be network-independent. This trick is also applicable to oblivious addressing transducers, because they too can not distinguish between single-node and multi-node networks unless they receive a message. We now explicitly give the results that are not transferable to addressing transducers, and why this is the case.

First, Proposition 3.2 does not hold for oblivious addressing transducer networks, because of the following reasons. The result talks about a concrete transducer network, in which the transducer may know how many nodes there are. If there are multiple nodes, the transducer may assume messages are eventually delivered. So, it is possible to construct a multi-node transducer network in which the oblivious addressing transducer smuggles node identifiers in the sent messages (by reading `All`), and when these arrive, it is possible to only output the input facts whose active domain is contained in the set of node identifiers. This would prevent the transducer network from computing a generic query.

Although not purely about obliviousness, the result of Theorem 3.23 is also not transferable to addressing transducers, as illustrated by the following example, where relation `All` is used to make the nodes dependent on message arrival.

Example 3.32. We give an addressing transducer that reads only relation `All` and that computes the nonmonotone emptiness query on a nullary input relation R (see also Example 3.15). Reading relation `All` in output or memory queries, a node can know from the start if it is alone or not. If the node is alone, then it can immediately output the desired result by looking at the local relation R. But if there are multiple nodes, every node x sends each local fact `All`(y) as a message $A(y)$ to node y. Although the operational semantics drops the message when y is not a neighbor of x, because each network is connected, y has at least one neighbor from which it will receive $A(y)$. This way, each node can establish its own identity. Next, the same protocol as in Example 3.15 is followed. □

[4]Note, we can not completely forbid the use of relations `Id` or `All` because we need to somehow indicate addressees.

Chapter 4

Deciding Eventual Consistency

4.1 Outline

In this chapter we investigate the decidability of eventual consistency for transducer networks. First, Section 4.2 presents related work, and Section 4.3 gives technical remarks specific to this chapter. Section 4.4 provides some additional definitions. Section 4.5 formalizes eventual consistency for networks as *confluence*, along with syntactic restrictions leading to so-called “simple” networks; related (un)decidability results are also presented. Section 4.6 shows that confluence of a simple network with multiple nodes can be reduced to confluence of a simple *single-node* network. Next, Section 4.7 establishes a small model property for simple single-node networks. This is used in Section 4.8 to give a procedure for deciding whether a simple single-node network is confluent, along with a NEXPTIME-completeness result for the complexity. The expressiveness of simple networks, not necessarily single-node, is analyzed in Section 4.9. Variations of the transducer network model are discussed in Section 4.10.

4.2 Related work

The work most closely related to ours is that by Deutsch et al. on verification of communicating data-driven Web services [30]. The main differences between our works are the following. *(i)* In their setting, message buffers are ordered queues; in our setting, message buffers are unordered multisets. Unordered buffers model the asynchronous communication typical in cloud computing [37] where messages can be delivered out of order. *(ii)* In their setting, to obtain decidability, message buffers are bounded and lossy; in our setting, they are unbounded and not lossy. *(iii)* In their setting, transducers are less severely restricted than in our setting. *(iv)* In their setting, clusters of transducers are verified for properties expressed in (first-order) linear temporal logic;[1] in our setting, we are really focusing on the property of

[1]Deutsch et al. can also verify branching-time temporal properties, but only when transducer states are propositional.

eventual consistency. It is actually not obvious whether eventual consistency (in the way we define it formally) is a linear-time temporal property, and if it is, whether it is expressible in first-order linear temporal logic.

This chapter is the extended version of our conference paper [17]. Some proof details are not included in this chapter, but can be found in the technical report [19].

4.3 General Remarks

The transducer model of Chapter 3 was more narrowly focused on the CALM conjecture: transducer programs are considered to be network-independent, and nodes communicate in an epidemic manner by spreading messages to their neighbors. In this chapter, we will use a more general model [1, 37], where different nodes can run different programs (non-homogeneous transducer networks), and nodes can directly address their messages to specified nodes (addressing transducers). So, we will again use the general transducer network model presented in Section 2.5, and additionally assume each transducer is addressing. Other differences with Chapter 3 are highlighted below. See also Section 4.10 for a discussion.

Distributed queries In Chapter 3, we did not consider the problem of deciding eventual consistency; we simply assumed eventual consistency and were focusing on expressiveness issues. In particular, we were focusing on standard queries to databases, computed in a distributed fashion by distributing the database in an arbitrary way over the nodes of the network. This only seems useful when each node uses the same input schema, which is an assumption we no longer make in this chapter. Therefore, in the present model, we directly consider distributed queries, i.e., the input to the query is a distributed database, and different distributions of the same dataset (only applicable if nodes have compatible schemas) may yield different answers to the query.

Sending To make some constructions technically less involved, we assume the semantics of Section 2.5.2 also allows a node to send messages to itself, instead of only to its neighbors. We also assume for convenience that each network $\mathcal{N}$ is a complete graph, i.e., each node can send to any other node. For this reason, we regard $\mathcal{N}$ as simply a *set* of nodes.

Rule-based We immediately restrict attention to transducers whose queries are specified with $\mathrm{UCQ}^{\neg}$.[2] This results in a rule-based formalism to express the computations, following the idea behind declarative networking [45]. Also, rules appear quite suitable for imposing syntactic restrictions, as we will do here. For simplicity, we assume all rules are *constant-free.*

Besides positive and negative body atoms, we allow rules to additionally contain a set of nonequalities of the form $(u \neq v)$ where $u, v \in \mathbf{var}$. A satisfying valuation has to assign different values to the variables u and v.

[2]The language $\mathrm{UCQ}^{\neg}$ is formalized in Section 2.6.

Runs In this chapter, a run $\mathcal{R}$ is just a *finite* sequence of global transitions. We write $\mathit{last}(\mathcal{R})$ to denote the last configuration reached by $\mathcal{R}$. The reason for considering finite runs is to simplify the problem setting. See also Chapter 7 for a discussion.

4.4 Additional Definitions

4.4.1 Distributed Queries

Let $\mathcal{E}$ be a distributed database schema, as defined in Section 2.3. Let $\mathcal{F}$ be another distributed database schema over the *same* network as $\mathcal{E}$. A *distributed query $\mathcal{Q}$ over input schema $\mathcal{E}$ and output schema $\mathcal{F}$* is a function that maps distributed database instances over $\mathcal{E}$ to distributed database instances over $\mathcal{F}$.

Like before, we will refer to ordinary database queries (as defined in Section 2.1) just as “queries”.

4.4.2 Derivation Trees

We want to formally describe *how* a fact is derived by a transducer, i.e., we want to make visible what rules and valuations are used. To explain a fact, in some cases it suffices to give a so-called *derivation pair* (φ, V), consisting of a rule φ and a satisfying valuation V. In other cases, we want to explain all facts that are recursively needed by the satisfying valuation, i.e., the facts $V(\mathit{pos}_\varphi)$. For this purpose, we use *derivation trees*, and this is formalized below.

Let Π be a transducer over a schema Υ. A *(full) derivation tree $\mathcal{T}$* of Π is a tuple $(\mathit{nodes}^\mathcal{T}, \mathit{edges}^\mathcal{T}, \mathit{rule}^\mathcal{T}, \mathit{val}^\mathcal{T}, \mathit{lit}^\mathcal{T})$ where

- $\mathit{nodes}^\mathcal{T}$ and $\mathit{edges}^\mathcal{T}$ are respectively the nodes and parent-child edges that together form a tree;
- $\mathit{rule}^\mathcal{T}$ is a function that maps each internal node $x \in \mathit{nodes}^\mathcal{T}$ to a rule $\mathit{rule}^\mathcal{T}(x)$ of Π;
- $\mathit{val}^\mathcal{T}$ is a function that maps each internal node $x \in \mathit{nodes}^\mathcal{T}$ to a valuation $\mathit{val}^\mathcal{T}(x)$ for $\mathit{rule}^\mathcal{T}(x)$ such that the nonequalities are satisfied; and,
- $\mathit{lit}^\mathcal{T}$ is a function that maps each non-root node $x \in \mathit{nodes}^\mathcal{T}$ to a literal $\mathit{lit}^\mathcal{T}(x)$ in the body of $\mathit{rule}^\mathcal{T}(y)$ where y is the parent of x,

subject to these additional constraints:

- for each internal node $x \in \mathit{nodes}^\mathcal{T}$, for each literal $\boldsymbol{l}$ in the body of rule $\mathit{rule}^\mathcal{T}(x)$, there is precisely one child y of x such that $\mathit{lit}^\mathcal{T}(y) = \boldsymbol{l}$;
- for each non-root node $x \in \mathit{nodes}^\mathcal{T}$, if $\mathit{lit}^\mathcal{T}(x)$ is a database literal, or if $\mathit{lit}^\mathcal{T}(x)$ is negative, then x must be a leaf; and,
- for all non-root internal nodes $x \in \mathit{nodes}^\mathcal{T}$, having a parent y, applying valuation $\mathit{val}^\mathcal{T}(x)$ to the head of rule $\mathit{rule}^\mathcal{T}(x)$ results in the same fact as applying the parent valuation $\mathit{val}^\mathcal{T}(y)$ to the (positive) atom inside literal $\mathit{lit}^\mathcal{T}(x)$.

For each internal node x of $\mathcal{T}$, we write $\mathit{fact}^{\mathcal{T}}(x)$ to denote the fact $\mathit{val}^{\mathcal{T}}(x)(\boldsymbol{a})$, where $\boldsymbol{a}$ is the head of $\mathit{rule}^{\mathcal{T}}(x)$. For a leaf node y with parent x, we write $\mathit{fact}^{\mathcal{T}}(y)$ to denote the fact $\mathit{val}^{\mathcal{T}}(x)(\boldsymbol{a})$, where $\boldsymbol{a}$ is the atom inside the literal $\mathit{lit}^{\mathcal{T}}(y)$. We write $\mathit{int}^{\mathcal{T}}$ to denote the set of internal nodes of $\mathcal{T}$.

From $\mathit{nodes}^{\mathcal{T}}$ and $\mathit{edges}^{\mathcal{T}}$ we can always uniquely identify the root node of $\mathcal{T}$, which we denote as $\mathit{root}^{\mathcal{T}}$. Let $\boldsymbol{f}$ be a fact over a relation $R^{(k)} \in \Upsilon_{\text{out}} \cup \Upsilon_{\text{msg}} \cup \Upsilon_{\text{mem}}$. A derivation tree $\mathcal{T}$ is said to be *for* fact $\boldsymbol{f}$ if applying valuation $\mathit{val}^{\mathcal{T}}(\mathit{root}^{\mathcal{T}})$ to the head of rule $\mathit{rule}^{\mathcal{T}}(\mathit{root}^{\mathcal{T}})$ results in the fact $\boldsymbol{f}$.

4.4.2.1 Schedulings

We now relate derivation trees to runs. Formally, a *scheduling* for a derivation tree $\mathcal{T}$ is a function κ that assigns to each internal node x of $\mathcal{T}$ a nonzero natural number $\kappa(x)$, subject to the constraint that nodes always get strictly lower numbers than their ancestors. Intuitively, $\kappa(x)$ represents the transition number of a run in which $\mathit{rule}^{\mathcal{T}}(x)$ should fire under valuation $\mathit{val}^{\mathcal{T}}(x)$.

The *canonical scheduling* of $\mathcal{T}$, denoted $\kappa^{\mathcal{T}}$, is the (unique) scheduling for which there is at least one internal node x such that $\kappa^{\mathcal{T}}(x) = 1$, and for all parent-child edges (x, y) we have $\kappa^{\mathcal{T}}(x) = \kappa^{\mathcal{T}}(y) + 1$. Intuitively, the canonical scheduling executes the derivations of $\mathcal{T}$ as tightly as possible at the beginning of a run.

4.4.3 Encoding

We specify how a transducer network can be given as input to a decision procedure. The encoding of a transducer network $\boldsymbol{\mathcal{N}}$ is a sequence of transducers (and their schemas), one for each node of $\boldsymbol{\mathcal{N}}$, as follows. For each node, *(i)* the transducer schema is represented by a sequence of (relname,type)-pairs, where relname is a relation name and the type indicates whether the relation is input, output, etc; and, *(ii)* the transducer itself is given by a sequence of rules that are written in full, like in the later Example 4.1.[3] We assume that the transducer schema only mentions relations effectively used by the rules. To represent the relation names and variables, binary numbers must be used, so that the number of bits is logarithmic in the total number of relations and variables respectively. Moreover, some small fixed alphabet of auxiliary characters needs to be used, to represent the type of relations in the transducer schema, and to separate the different components (schemas, transducers, rules, etc).

We write $|\boldsymbol{\mathcal{N}}|$ to denote the size of the encoding of $\boldsymbol{\mathcal{N}}$.

4.5 Confluence

In this chapter, we formalize "eventual consistency" [57, 37] as a confluence notion. Let $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a transducer network. Let H be an input distributed database instance for $\boldsymbol{\mathcal{N}}$. By the asynchronous nature of message delivery, different (finite) runs of $\boldsymbol{\mathcal{N}}$ on H can deliver messages in different orders. So, in some run $\mathcal{R}$, it is possible that the transducer at some node $x \in \mathcal{N}$ applies negation too quickly, without having seen some crucial messages. This could accidentally prevent some outputs obtained in

[3]It is reasonable to write the components of body atoms in full, because we need to describe which variables are used, and how they are potentially shared between atoms.

other runs from ever being produced in any extension of $\mathcal{R}$. By contrast, transducer networks where such problems are not possible are called *confluent.*

Formally, we call $\boldsymbol{\mathcal{N}}$ *confluent on H* if for any two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\boldsymbol{\mathcal{N}}$ on H, for every node $x \in \mathcal{N}$, for every output fact $\boldsymbol{f}$ available at x in the last configuration of $\mathcal{R}_1$, there exists an extension $\mathcal{R}_2'$ of $\mathcal{R}_2$ such that $\boldsymbol{f}$ is available at x in the last configuration of $\mathcal{R}_2'$. To rephrase, if during one run some node can produce an output fact, then for any run there exists an extension in which that fact can be produced on that node too. Naturally, we call $\boldsymbol{\mathcal{N}}$ *confluent* if $\boldsymbol{\mathcal{N}}$ is confluent on all input distributed database instances. If $\boldsymbol{\mathcal{N}}$ is not confluent, we say that $\boldsymbol{\mathcal{N}}$ is *diffluent.* Confluence is one way of formalizing eventual consistency; but see also Chapter 7 for a discussion.

Consider now the following example of a confluent transducer network:

Example 4.1. Let $\mathcal{N} = \{x, y\}$ be a network of two nodes. We define a transducer network $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$. There are no memory relations in this example.

First, define $\boldsymbol{\Upsilon}(x)_{\text{in}} = \{A^{(1)}\}$, $\boldsymbol{\Upsilon}(x)_{\text{out}} = \{T^{(1)}\}$, $\boldsymbol{\Upsilon}(x)_{\text{msg}} = \{A^{(1)}_{\text{msg}}, B^{(2)}_{\text{msg}}\}$, and $\boldsymbol{\Upsilon}(x)_{\text{mem}} = \emptyset$. Transducer $\boldsymbol{\Pi}(x)$ is given as

$$A_{\text{msg}}(\mathtt{y},\mathtt{u}) \leftarrow A(\mathtt{u}),\ \mathtt{All(y)},\ \neg\mathtt{Id(y)}.$$
$$T(\mathtt{u}) \leftarrow B_{\text{msg}}(\mathtt{x},\mathtt{u}),\ \mathtt{Id(x)}.$$

Next, define $\boldsymbol{\Upsilon}(y)_{\text{in}} = \{B^{(2)}\}$, $\boldsymbol{\Upsilon}(y)_{\text{out}} = \{T^{(1)}\}$, $\boldsymbol{\Upsilon}(y)_{\text{msg}} = \boldsymbol{\Upsilon}(x)_{\text{msg}}$ (shared messages), and $\boldsymbol{\Upsilon}(y)_{\text{mem}} = \emptyset$. Transducer $\boldsymbol{\Pi}(y)$ is given as

$$B_{\text{msg}}(\mathtt{y},\mathtt{u},\mathtt{v}) \leftarrow B(\mathtt{u},\mathtt{v}),\ \mathtt{All(y)},\ \neg\mathtt{Id(y)}.$$
$$T(\mathtt{u}) \leftarrow A_{\text{msg}}(\mathtt{u}).$$

On any input distributed database instance H for $\boldsymbol{\mathcal{N}}$, node x sends its local A-facts as A_{msg}-facts to y. Similarly, y sends its local B-facts as B_{msg}-facts to x. For a received B_{msg}-fact, node x outputs the second component in relation T if the first component is its identifier. Node y simply outputs all received A_{msg}-facts.

The above transducer network is confluent. Indeed, say, node x outputs a fact $T(a)$ during a run. This means that x has received $B_{\text{msg}}(x, a)$, which was sent by node y based on an input fact $B(x, a)$. On the same input distributed database instance, consider now any run where x has not yet output $T(a)$. We can extend this run as follows. We do a global transition with active node y, so that y sends its input B-facts as B_{msg}-facts to x. One of these messages is $B_{\text{msg}}(x, a)$. Then, in a following global transition, we deliver $B_{\text{msg}}(x, a)$ to x, and x again outputs $T(a)$. Similarly, we can argue that if the node y outputs a T-fact in one run, then any other run on the same input can be extended to make y output this fact. □

By contrast, consider the following example of a transducer network that is diffluent.

Example 4.2. Let $\mathcal{N} = \{x, y\}$ be a network. We define a transducer network $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ as follows. In this example, we do no deletions on memory relations, and we will only explicitly specify the insertions.

First, define $\boldsymbol{\Upsilon}(x)_{\text{in}} = \{A^{(1)}, B^{(1)}\}$, $\boldsymbol{\Upsilon}(x)_{\text{out}} = \emptyset$, $\boldsymbol{\Upsilon}(x)_{\text{msg}} = \{A^{(1)}_{\text{msg}}, B^{(1)}_{\text{msg}}\}$, and $\boldsymbol{\Upsilon}(x)_{\text{mem}} = \emptyset$. The node x sends its local A- and B-facts to the other node y. Transducer $\boldsymbol{\Pi}(x)$ is given as

$$A_{\text{msg}}(\mathtt{y},\mathtt{u}) \leftarrow A(\mathtt{u}),\ \mathtt{All}(\mathtt{y}),\ \neg\mathtt{Id}(\mathtt{y}).$$
$$B_{\text{msg}}(\mathtt{y},\mathtt{u}) \leftarrow B(\mathtt{u}),\ \mathtt{All}(\mathtt{y}),\ \neg\mathtt{Id}(\mathtt{y}).$$

Next, define $\boldsymbol{\Upsilon}(y)_{\text{in}} = \emptyset$, $\boldsymbol{\Upsilon}(y)_{\text{out}} = \{T^{(1)}\}$, $\boldsymbol{\Upsilon}(y)_{\text{msg}} = \boldsymbol{\Upsilon}(x)_{\text{msg}}$ (shared messages), and $\boldsymbol{\Upsilon}(y)_{\text{mem}} = \{B^{(1)}\}$. Transducer $\boldsymbol{\Pi}(y)$ is given as:

$$B(\mathtt{u}) \leftarrow B_{\text{msg}}(\mathtt{u}).$$
$$T(\mathtt{u}) \leftarrow A_{\text{msg}}(\mathtt{u}),\ \neg B(\mathtt{u}).$$

Now we show why $\boldsymbol{\mathcal{N}}$ is diffluent. Let H be the following instance over $in^{\boldsymbol{\mathcal{N}}}$: $H(x) = \{A(1),\ B(1)\}$ and $H(y) = \emptyset$. There are two quite different runs possible, that we describe next. Suppose that both runs start with a global transition with active node x. This causes x to send both $A_{\text{msg}}(1)$ and $B_{\text{msg}}(1)$ to y. For the first run, in the second transition we deliver only $A_{\text{msg}}(1)$ to y, which causes y to output $T(1)$. For the second run, in the second transition we deliver only $B_{\text{msg}}(1)$ to y, which causes y to only create the memory fact $B(1)$. Now, the output fact $T(1)$ can not be created in any extension of the second run because each time we deliver $A_{\text{msg}}(1)$ to y, the presence of $B(1)$ prevents $T(1)$ from being created. □

4.5.1 Decision Problem

In a diffluent transducer network, the output can vary strongly depending on how messages are delivered. It thus seems useful to know if a transducer network could be diffluent. Formally, we have the following *diffluence decision problem*: given a transducer network $\boldsymbol{\mathcal{N}}$, decide if $\boldsymbol{\mathcal{N}}$ is diffluent (for some input). One can expect this problem to be undecidable in general. For this reason, we consider possible syntactical restrictions on transducer networks in Section 4.5.2, and Section 4.5.3 investigates their effect on decidability.

4.5.2 Syntactical Restrictions

We introduce several syntactical restrictions on individual transducers and on transducer networks as a whole. Let Π be a transducer over a schema Υ. For an individual rule φ of Π, we consider the following possible restrictions:

- We say that φ is *message-positive* if there are no message atoms in neg_φ. Note, this seems to be a natural constraint in our model because message delivery is asynchronous.
- We say that φ is *static* if pos_φ and neg_φ do not contain output or memory atoms, i.e., φ does not use output or memory relations in its body.

- We say that φ is *message-bounded* if $V \subseteq A$ and $V \cap B = \emptyset$, where V is the set of bound variables of φ (i.e., not occurring in the head); A is the set of variables of φ occurring in positive message atoms; and, B is the set of variables of φ occurring in output or memory atoms. In words: every bound variable occurs in a positive message atom, and does not occur in output or memory atoms (positive or negative). This is an application of the more general notion of "input-boundedness" [54, 30, 29].[4]

We consider the following restrictions for transducer Π:

- We say that Π is *recursion-free* if there are no cycles in the *positive dependency graph* of Π, which is the graph having as vertices the relations of $\Upsilon_{\text{out}} \cup \Upsilon_{\text{msg}} \cup \Upsilon_{\text{mem}}$ and there is an edge from relation R to relation S if S occurs positively in a rule for R in Π.
- We say that Π is *inflationary* if there are no rules for the deletion queries of memory relations. This means that Π can not delete memory facts once they are produced.

We call Π *simple* (for lack of a better name) if

- Π is recursion-free and inflationary;
- all send rules are message-positive and static;[5] and,
- all insertion rules for output and memory relations are message-positive and message-bounded.

Because input facts are never changed, note that static send rules always produce the same result on receipt of the same messages, independently of what output or memory facts might have been derived. Also, if Π is inflationary, memory and output relations basically behave in the same way. However, we preserve the difference between these two kinds of relations to retain the connection to the unrestricted transducer model and because memory relations are useful as a separate construct, namely, as relations used for computation but that don't belong to the final result.

Let $\mathcal{N}$ be a transducer network. We present a restriction that we can impose on $\mathcal{N}$ as a whole. Note that messages are the only way to introduce a dependency between different nodes of $\mathcal{N}$. Now, we say that $\mathcal{N}$ is *globally recursion-free* if there are no cycles in the *positive message dependency graph* of $\mathcal{N}$, which is the graph having as vertices the (shared) message relations of $\mathcal{N}$ and there is an edge from relation R to relation S if S occurs positively in a rule for R in some transducer of $\mathcal{N}$.

We call $\mathcal{N}$ *simple* if

- all transducers of $\mathcal{N}$ are simple; and,
- $\mathcal{N}$ is globally recursion-free.

The Examples 4.1 and 4.2 are simple transducer networks.

[4]We have replaced the term "input-boundedness" by "message-boundedness" because the word "input" has a different meaning in our text, namely, as the input that a transducer is locally initialized with.

[5]The restrictions considered by Deutsch et al. [29] for "input-rules", which are closely related to our send rules, are a bit less restrictive. Roughly speaking, they still allow the use of nullary output and memory facts. It seems plausible that our results can be similarly extended.

4.5.3 Results on Decidability

One of the difficulties of the diffluence decision problem is that we need to verify a property of an infinite state system. Intuitively, there are infinitely many inputs and even for a fixed input there are infinitely many configurations because there is no bound on the size of the message buffer. As the following two propositions show, diffluence for transducer networks is undecidable, even under several restrictions:

Proposition 4.3. Diffluence is undecidable for transducer networks that are simple, except that send rules do not have to be static.

Proof. Inspired by the proof technique of Deutsch et al. [30], we reduce the the finite implication problem for functional and inclusion dependencies [26] to the diffluence decision problem. We sketch the proof; the full details are in [19]. An instance of the finite implication problem is a triple $(\mathcal{D}, \Sigma, \sigma)$, where $\mathcal{D}$ is a database schema, Σ is a set of functional and inclusion dependencies over $\mathcal{D}$, and σ is a functional or inclusion dependency over $\mathcal{D}$. We call $(\mathcal{D}, \Sigma, \sigma)$ *valid* if $I \models \Sigma$ implies $I \models \sigma$ for each instance I over $\mathcal{D}$.[6] We have to check validity of $(\mathcal{D}, \Sigma, \sigma)$.

For the instance $(\mathcal{D}, \Sigma, \sigma)$, we construct a single-node transducer network $\boldsymbol{\mathcal{N}}$ that is simple except that send rules are not static, and so that $\boldsymbol{\mathcal{N}}$ is diffluent iff $(\mathcal{D}, \Sigma, \sigma)$ is not valid. Let Π denote the single transducer of $\boldsymbol{\mathcal{N}}$. We let the input schema of Π contain $\mathcal{D}$. Transducer Π sends a special marker message to itself, and when the marker is received, Π checks whether the input over $\mathcal{D}$ satisfies Σ and σ. Non-static send rules are needed for checking the inclusion dependencies. For each violated dependency $\tau \in \Sigma \cup \{\sigma\}$, transducer Π sends a $\mathtt{viol}_\tau(\,)$-message to itself.

Upon receiving $\mathtt{viol}_\sigma(\,)$, the transducer does something diffluent, by blocking a rule for output relation T as was done in Example 4.2, so that an incoming $A_{\text{msg}}(a)$-fact is ignored when memory fact $B(a)$ was previously created. But when some $\mathtt{viol}_\tau(\,)$ message with $\tau \in \Sigma$ is received, we repair the diffluent behavior: we fill a nullary memory relation `repair`, that is tested positively in another output rule for relation T. This second rule for T can henceforth output all received A_{msg}-facts.

Now, if $(\mathcal{D}, \Sigma, \sigma)$ is not valid, there is an instance I over $\mathcal{D}$ such that $I \models \Sigma$ and $I \not\models \sigma$. Instance I can be extended to an input J for $\boldsymbol{\mathcal{N}}$, and we make two runs as follows. In the first run, an output $T(a)$ is produced by first delivering some fact $A_{\text{msg}}(a)$ and by postponing the marker message (to postpone the dependency checking). In the second run, we do the converse, i.e., we deliver the marker first. Then, dependency σ turns out to be violated, and upon delivery of $\mathtt{viol}_\sigma(\,)$, we can block the output. No repairs are possible because only σ is violated.

Conversely, if $\boldsymbol{\mathcal{N}}$ is diffluent on an input J, this can only be explained by σ being violated and no dependency of Σ, so that the input of $\boldsymbol{\mathcal{N}}$ gives rise to an instance I over $\mathcal{D}$ for which $I \models \Sigma$ and $I \not\models \sigma$. Hence, $(\mathcal{D}, \Sigma, \sigma)$ is not valid. □

Proposition 4.4. Diffluence is undecidable for transducer networks that are simple, except that messages may participate in cycles in the local positive dependency graphs of individual transducers.

[6]We write $I \models \sigma$ to denote that σ holds in I. We write $I \models \Sigma$ to denote that $I \models \sigma$ for each $\sigma \in \Sigma$.

Proof. Inspired by the proof technique of Deutsch et al. [30], we reduce the Post correspondence problem [51] to the diffluence decision problem. We sketch the proof; the full details are in [19]. An instance of the Post correspondence problem is a pair (U, V) where $U = u_1, \ldots, u_n$ and $V = v_1, \ldots, v_n$ are two nonempty equal-length sequences of nonempty words over some alphabet with at least two symbols. A *match* for U and V is a finite sequence $E = e_1, \ldots, e_m$ of indices in $\{1, \ldots, n\}$ such that the words $u_{e_1} \ldots u_{e_m}$ and $v_{e_1} \ldots v_{e_m}$ are equal. Sequence E may contain the same index multiple times. The problem is to check whether a match exists.

For the instance (U, V), we construct a single-node transducer network $\boldsymbol{\mathcal{N}}$ that is simple except that messages can have recursive dependencies, and so that $\boldsymbol{\mathcal{N}}$ is diffluent iff (U, V) has a match. Let Π denote the single transducer of $\boldsymbol{\mathcal{N}}$. First, we provide Π with input relations to encode a word-structure: a binary relation R represents a chain, and a binary relation L assigns a label to each element of the chain.

The idea is to use messages to align the words of U and V to the input word-structure, to discover a match for (U, V). Concretely, we use messages of the form $\mathtt{align}[i, k, l](a, b)$, with $i \in \{1, \ldots, n\}$, $k \in \{1, \ldots, |u_i|\}$ and $l \in \{1, \ldots, |v_i|\}$, expressing that we have already successfully aligned a sequence of (u_j, v_j)-pairs with $j \in \{1, \ldots, n\}$ to the word-structure, where (u_i, v_i) is the last pair tried, and the alignment of u_i and v_i has progressed partially up to respectively symbols k and l, arriving at respectively elements a and b of the word-structure. After a message $\mathtt{align}[i, |u_i|, |v_i|](a, b)$ is sent, indicating that (u_i, v_i) is fully aligned, we have sending rules to align a next pair (u_j, v_j), by sending message $\mathtt{align}[j, 1, 1](a', b')$, where a' and b' are the successor-elements of respectively a and b on the word-structure. Adding unrestricted message recursion adds some notion of "iteration" to the transducer model: because message relations are allowed to participate in cycles, the alignment to the word-structure can repeatedly use the *same* pair (u_i, v_i), allowing us to consider all candidate sequences E like above (but restricted to the input word structure).

If there is indeed a match for (U, V) then we can encode the resulting word as an input word-structure for $\boldsymbol{\mathcal{N}}$. So, the above alignment process can eventually send a message of the form $\mathtt{align}[j, |u_j|, |v_j|](a, a)$, i.e., we can align a sequence of (u_i, v_i)-pairs fully to the word-structure, where the implied concatenation of U-words ends at the same element of the word-structure as the implied concatenation of V-words. Then we do something diffluent, like in Example 4.2.

For the other direction, when $\boldsymbol{\mathcal{N}}$ is diffluent on some input, we can attribute that to the sending of a message $\mathtt{align}[j, |u_j|, |v_j|](a, a)$, whose derivation history reveals a match for (U, V) against a valid word-structure contained in the input of $\boldsymbol{\mathcal{N}}$. □

By disallowing the syntactical liberties of the previous two propositions, we obtain decidability:

Theorem 4.5. Diffluence for simple transducer networks is decidable in NEXPTIME; in fact, the problem is NEXPTIME-complete.

Theorem 4.5 is proven in Sections 4.6, 4.7, and 4.8.

4.6 Simulation on Single Node

Let $\mathcal{N}$ be a simple transducer network. We construct a simple *single-node* transducer network $\mathcal{M}$ that simulates $\mathcal{N}$, and so that $\mathcal{M}$ is confluent iff $\mathcal{N}$ is confluent. This will be made more precise below. The transformation can be done in PTIME for reasonable encodings of a transducer network, and so $|\mathcal{M}|$ is polynomial in $|\mathcal{N}|$ (cf. Section 4.4.3). The merit of this section lies in reducing the technical complexity for the decidability result (Sections 4.7 and 4.8) and the expressivity analysis (Section 4.9).

First, Section 4.6.1 gives syntactical simplifications for single-node networks. Next, Section 4.6.2 formalizes the notion of simulation and formulates the result. The sections thereafter show the result: Sections 4.6.3 and 4.6.4 respectively define the transducer schema and transducer of $\mathcal{M}$, and Section 4.6.5 shows that $\mathcal{M}$ satisfies the desired properties.

4.6.1 Syntactical Simplifications

For a single-node transducer network $\mathcal{M}$, we use the following syntactical simplifications. It will be sufficient to view $\mathcal{M}$ as consisting of only a transducer schema Υ and a transducer Π over Υ; the actual node of $\mathcal{M}$ is immaterial. The schemas $in^{\mathcal{M}}$, $out^{\mathcal{M}}$ and $mem^{\mathcal{M}}$ (Section 2.5.1) are regarded as ordinary (non-distributed) database schemas. Accordingly, an input for $\mathcal{M}$ is an ordinary database instance I. A configuration of $\mathcal{M}$ on I is a pair (s, b) where s is a transducer state of Π and b is a multiset of facts over Υ_{msg}. Because there is only a single node, sending rules of Π have no explicit addressee variable in the head. Hence, schema Υ_{sys} will not be used.

4.6.2 Simulation Concept and Result

To formalize the notion of "simulation", we introduce some auxiliary notations. Let $\mathcal{N}$ denote the network of $\mathcal{N}$. For a distributed database schema $\mathcal{E}$ over $\mathcal{N}$, we view each node $x \in \mathcal{N}$ as a namespace containing the relations $\mathcal{E}(x)$: we use symbol "$x.R$" to denote relation R at x. Let $\langle\mathcal{E}\rangle$ denote the (ordinary) database schema

$$\{x.R^{(k)} \mid x \in \mathcal{N},\ R^{(k)} \in \mathcal{E}(x)\}.$$

For each distributed database instance H over $\mathcal{E}$, let $\langle H\rangle$ be the following ordinary database instance over $\langle\mathcal{E}\rangle$:

$$\{x.R(\bar{a}) \mid x \in \mathcal{N},\ R(\bar{a}) \in H(x)\}.$$

Let $sch^{\mathcal{N}}$ denote the database schema $\{x.\texttt{Id}^{(1)} \mid x \in \mathcal{N}\} \cup \{\texttt{Node}^{(1)}\}$. Let $inst^{\mathcal{N}}$ be the following instance over $sch^{\mathcal{N}}$:

$$\{x.\texttt{Id}(x),\ \texttt{Node}(x) \mid x \in \mathcal{N}\}.$$

We abbreviate $\langle\mathcal{E}\rangle^{\mathcal{N}} = \langle\mathcal{E}\rangle \cup sch^{\mathcal{N}}$ and $\langle H\rangle^{\mathcal{N}} = \langle H\rangle \cup inst^{\mathcal{N}}$. We say that an instance I over $\langle\mathcal{E}\rangle^{\mathcal{N}}$ is *well-formed* if I is isomorphic to an instance J over $\langle\mathcal{E}\rangle^{\mathcal{N}}$ for which $J|_{sch^{\mathcal{N}}} = inst^{\mathcal{N}}$.[7] An instance that is not well-formed is called *ill-formed.*

[7] I is isomorphic to J if there is an injective function $f : \mathbf{dom} \to \mathbf{dom}$ such that $f(I) = J$.

For a configuration $\rho = (s, b)$ of $\boldsymbol{\mathcal{N}}$, we write $out(\rho)$ to denote the following distributed instance H' over $out^{\boldsymbol{\mathcal{N}}}$: for each $x \in \mathcal{N}$, instance $H'(x)$ consists of all output facts in $s(x)$. If $\boldsymbol{\mathcal{N}}$ is a single-node network, we consider $out(\rho)$ to be an ordinary database instance.

Now, we say that a single-node transducer network $\boldsymbol{\mathcal{M}}$ *simulates* $\boldsymbol{\mathcal{N}}$ if *(i)* $in^{\boldsymbol{\mathcal{M}}} = \langle in^{\boldsymbol{\mathcal{N}}} \rangle^{\mathcal{N}}$; *(ii)* $out^{\boldsymbol{\mathcal{M}}} = \langle out^{\boldsymbol{\mathcal{N}}} \rangle$; and, *(iii)* for each input H for $\boldsymbol{\mathcal{N}}$, the following holds:

- for every run $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on H, there is a run $\mathcal{S}$ of $\boldsymbol{\mathcal{M}}$ on $\langle H \rangle^{\mathcal{N}}$ such that $\langle out(last(\mathcal{R})) \rangle = out(last(\mathcal{S}))$,
- for every run $\mathcal{S}$ of $\boldsymbol{\mathcal{M}}$ on $\langle H \rangle^{\mathcal{N}}$, there is a run $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on H such that $\langle out(last(\mathcal{R})) \rangle = out(last(\mathcal{S}))$.

We set $in^{\boldsymbol{\mathcal{M}}} = \langle in^{\boldsymbol{\mathcal{N}}} \rangle^{\mathcal{N}}$ instead of $in^{\boldsymbol{\mathcal{M}}} = \langle in^{\boldsymbol{\mathcal{N}}} \rangle$ because $\boldsymbol{\mathcal{M}}$ needs the identifiers of the nodes to simulate message sending and the nodes' comparisons of their identifier to input values, and because we disallow constants in rules (Section 4.3).

Now we are ready to present the result:

Proposition 4.6. For each simple transducer network $\boldsymbol{\mathcal{N}}$, there exists a simple single-node transducer network $\boldsymbol{\mathcal{M}}$ such that *(i)* $\boldsymbol{\mathcal{M}}$ simulates $\boldsymbol{\mathcal{N}}$, and *(ii)* $\boldsymbol{\mathcal{M}}$ is confluent iff $\boldsymbol{\mathcal{N}}$ is confluent.

Note, the simulation property says nothing about confluence and vice versa. The following subsections define $\boldsymbol{\mathcal{M}}$ so that the desired properties are satisfied.

4.6.3 Transducer Schema

We define the single transducer schema Υ of $\boldsymbol{\mathcal{M}}$. Denote $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$. Let $\mathcal{D}^{\boldsymbol{\mathcal{N}}}_{\text{msg}}$ denote the shared message schema of $\boldsymbol{\mathcal{N}}$. We define Υ as follows:

- $\Upsilon_{\text{in}} = \langle in^{\boldsymbol{\mathcal{N}}} \rangle^{\mathcal{N}}$; $\Upsilon_{\text{out}} = \langle out^{\boldsymbol{\mathcal{N}}} \rangle$; $\Upsilon_{\text{mem}} = \langle mem^{\boldsymbol{\mathcal{N}}} \rangle$; and,
- Υ_{msg} consists of *(i)* the relations $R^{(k+1)}_{\to x}$ for which $x \in \mathcal{N}$ and $R^{(k)} \in \mathcal{D}^{\boldsymbol{\mathcal{N}}}_{\text{msg}}$, *(ii)* a relation $\texttt{do}^{(0)}_x$ for each $x \in \mathcal{N}$, *(iii)* relation $\texttt{error}^{(0)}$, and *(iv)* relation $\texttt{adom}^{(1)}$.

Relations of the form $\texttt{do}_x$ allow us to explicitly simulate a transition of node x. Next, a relation $R_{\to x}$ is used to send R-facts specifically to node x. The latter relations have an incremented arity when compared to $\mathcal{D}^{\boldsymbol{\mathcal{N}}}_{\text{msg}}$, for the following reason. Each transition of the transducer Π in $\boldsymbol{\mathcal{M}}$ can simulate multiple nodes simultaneously, and these simulated nodes could send the same message to the same addressee. But the transition of Π can only send a *set* of messages. So, by letting Π additionally put the simulated sender node in each simulated message, we can avoid that these distinct simulated sending events would all be collapsed. Lastly, the relations `error` and `adom` allow Π to be confluent on ill-formed inputs; see below.

4.6.4 Transducer Rules

We now describe the single transducer Π of $\boldsymbol{\mathcal{M}}$. Essentially, the UCQ$^\neg$ queries of Π are unions of modified UCQ$^\neg$ queries of the original transducers in $\boldsymbol{\mathcal{N}}$. Some extra rules deal with ill-formed inputs.

4.6.4.1 Output and Memory

We do the following for each node $x \in \mathcal{N}$. Let $T^{(k)}$ be an output or memory relation in $\boldsymbol{\Upsilon}(x)$. All rules for relation T in $\boldsymbol{\Pi}(x)$ are message-positive and message-bounded. An insertion rule φ for relation T in transducer $\boldsymbol{\Pi}(x)$ is modified to insertion rule φ' for relation $x.T$ in Π as follows:

- input, output and memory atoms $R(\bar{\mathtt{u}})$ in φ become $x.R(\bar{\mathtt{u}})$ in φ', including the head;
- atoms of the form $\mathtt{Id}(\mathtt{u})$ and $\mathtt{All}(\mathtt{u})$ in φ become respectively $x.\mathtt{Id}(\mathtt{u})$ and $\mathtt{Node}(\mathtt{u})$ in φ';
- (positive) message atoms $R(\bar{\mathtt{u}})$ in φ become $R_{\rightarrow x}(\mathtt{z}, \bar{\mathtt{u}})$ in φ' where $\mathtt{z}$ is a new variable that is unique per message atom;
- the nonequalities in φ are the nonequalities in φ';
- φ' additionally contains the positive body atom $\mathtt{do}_x(\,)$.

Intuitively, because relation `All` always contains $\mathcal{N}$ on every node of $\boldsymbol{\mathcal{N}}$, it is replaced by the shared relation `Node` in $\boldsymbol{\mathcal{M}}$. For a message atom $R_{\rightarrow x}(\mathtt{z}, \bar{\mathtt{u}})$, the new variable $\mathtt{z}$ represents the extra sender-component (cf. Section 4.6.3). This component is not used elsewhere in the rule and is basically projected away.

The resulting output and memory insertion rules are message-positive and message-bounded. Because $\boldsymbol{\Pi}(x)$ is simple, there are no deletion rules for memory relations, so we don't have to translate these.

4.6.4.2 Messages

We do the following for each node $x \in \mathcal{N}$. Let $T^{(k)}$ be a shared message relation of $\boldsymbol{\mathcal{N}}$. All rules for relation T in $\boldsymbol{\Pi}(x)$ are message-positive and static. To let simulated node x send messages in $\boldsymbol{\mathcal{M}}$, we add to Π all rules φ'_y obtained by combining a sending rule φ for T in $\boldsymbol{\Pi}(x)$ and a node $y \in \mathcal{N}$. Intuitively, rule φ'_y models the sending of T-messages by x to the specific addressee y. Denote $\mathit{head}_\varphi = T(\mathtt{n}_0, \bar{\mathtt{u}})$, where $\mathtt{n}_0$ is the addressee variable. Let $\mathtt{n}_1$ be a new variable. Rule φ'_y is obtained as follows:

- the head $T(\mathtt{n}_0, \bar{\mathtt{u}})$ of φ becomes the head $T_{\rightarrow y}(\mathtt{n}_1, \bar{\mathtt{u}})$ in φ'_y;
- φ'_y contains positive body atoms $y.\mathtt{Id}(\mathtt{n}_0)$ and $x.\mathtt{Id}(\mathtt{n}_1)$;
- input atoms $R(\bar{\mathtt{u}})$ in φ become $x.R(\bar{\mathtt{u}})$ in φ'_y;
- atoms of the form $\mathtt{Id}(\mathtt{u})$ and $\mathtt{All}(\mathtt{u})$, and message atoms, are transformed as in the output and memory rules above;
- the nonequalities of φ are the nonequalities of φ'_y;
- φ'_y additionally contains the positive body atom $\mathtt{do}_x(\,)$.

Variable $\mathtt{n_0}$ is not removed because it might occur on several places in φ, and by adding the atom $y.\mathtt{Id}(\mathtt{n_0})$, we fix the addressee y. Variable $\mathtt{n_1}$ represents the sender x by addition of the body atom $x.\mathtt{Id}(\mathtt{n_1})$, and $\mathtt{n_1}$ replaces $\mathtt{n_0}$ in the head.

Denote $\mathcal{N} = \{x_1, \ldots, x_n\}$. For each $x \in \mathcal{N}$, we also add the following rule to Π, to send simulation messages for x:

$$\mathtt{do}_x(\,) \leftarrow x_1.\mathtt{Id}(\mathtt{u_1}), \ldots, x_n.\mathtt{Id}(\mathtt{u_n}).$$

The above rule has the effect that a message $\mathtt{do}_y(\,)$ for any $y \in \mathcal{N}$ can only be sent if *all* relations $z.\mathtt{Id}$ with $z \in \mathcal{N}$ are nonempty. And because the simulated output, memory, and sending rules are guarded by message atoms of the form $\mathtt{do}_y(\,)$, the *entire* simulation requires that these relations $z.\mathtt{Id}$ are nonempty.

The above message rules of Π are all message-positive and static.

4.6.4.3 Ill-formed Inputs

We indicate how $\boldsymbol{\mathcal{M}}$ can be made confluent on ill-formed input instances. First, using message-positive and static send rules, it is possible to send a message $\mathtt{error}(\,)$ if the following constraints are violated: *(i)* some relation $x.\mathtt{Id}$ contains two different values; *(ii)* two relations $x.\mathtt{Id}$ and $y.\mathtt{Id}$ with $x \neq y$ share a value; and, *(iii)* relation $\mathtt{Node}$ is not the union of all $x.\mathtt{Id}$ relations.

We also add new output rules that on receipt of $\mathtt{error}(\,)$ can produce all possible output facts in Υ_{out}. Technically, this is done by adding rules to send all values a from the input active domain as an $\mathtt{adom}(a)$-message, and the additional output rules combine these values upon delivery when $\mathtt{error}(\,)$ is also jointly delivered.

4.6.4.4 Check Simple

We verify that Π is simple: *(i)* Π is inflationary by construction; *(ii)* Π is recursion-free because the transducers of $\boldsymbol{\mathcal{N}}$ are recursion-free and because there are no cycles in the positive message dependency graph of $\boldsymbol{\mathcal{N}}$ (global recursion-freeness); and, *(iii)* the desired constraints on output, memory and sending rules hold, as remarked above. Moreover, because Π is the only transducer of $\boldsymbol{\mathcal{M}}$ and Π is recursion-free, there are no cycles in the positive message dependency graph of $\boldsymbol{\mathcal{M}}$, and thus $\boldsymbol{\mathcal{M}}$ is simple.

4.6.5 Simulation and Confluence Equivalence

We now show that *(i)* $\boldsymbol{\mathcal{M}}$ simulates $\boldsymbol{\mathcal{N}}$ and *(ii)* $\boldsymbol{\mathcal{M}}$ is confluent iff $\boldsymbol{\mathcal{N}}$ is confluent. First we need some additional concepts and notations. Let $\rho = (s, b)$ be a configuration of $\boldsymbol{\mathcal{N}}$ on input H and let $\sigma = (s', b')$ be a configuration of $\boldsymbol{\mathcal{M}}$ on input $\langle H \rangle^{\mathcal{N}}$. We say that σ and ρ are *output-equivalent* if for each $x \in \mathcal{N}$ and each output relation R at x, we have $R(\bar{a}) \in s(x)$ iff $x.R(\bar{a}) \in s'$. The notions of *input-*, *memory-*, and *system-equivalence* can be similarly defined, where the latter demands that the relations $\mathtt{Id}$ and $\mathtt{All}$ of each original node in $\boldsymbol{\mathcal{N}}$ are represented exactly by the relations $x.\mathtt{Id}$ with $x \in \mathcal{N}$ and $\mathtt{Node}$ in $\boldsymbol{\mathcal{M}}$. The definition of $\langle H \rangle^{\mathcal{N}}$ implies that configurations σ and ρ are always input- and system-equivalent.

We say that σ and ρ are *message-equivalent* if for each $x \in \mathcal{N}$, for each fact $R(\bar{a})$, the cardinality of $R(\bar{a})$ in $b(x)$ equals the number of messages of the form $R_{\to x}(z, \bar{a})$

in b' (each may have a different sender component). Similarly, we say that σ has its *messages included* in ρ when for each $x \in \mathcal{N}$ the number of messages of the form $R_{\to x}(z, \bar{a})$ in b' is less than or equal to the cardinality of $R(\bar{a})$ in $b(x)$.

Claims 4.7 and 4.8 show that $\boldsymbol{\mathcal{M}}$ simulates $\boldsymbol{\mathcal{N}}$, but they are phrased slightly more general for later use in the confluence equivalence:

Claim 4.7. Every run $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on an input H can be converted to a run $\mathcal{S}$ of $\boldsymbol{\mathcal{M}}$ on $\langle H \rangle^{\mathcal{N}}$ such that $\mathit{last}(\mathcal{R})$ and $\mathit{last}(\mathcal{S})$ are output-, memory-, and message-equivalent.

Proof. Let n be the number of transitions in $\mathcal{R}$, and let $x_1, \ldots, x_n$ be the active nodes in order. Run $\mathcal{S}$ will consist of $n+1$ transitions: for each $i = 1, \ldots, n$, we deliver $\mathtt{do}_{x_i}(\,)$ in transition $i+1$ of $\mathcal{S}$ (and no other $\mathtt{do}_y$-messages). We start $\mathcal{S}$ by doing one heartbeat transition, so that at least $\mathtt{do}_{x_1}(\,)$ is sent. This message is delivered in the second transition of $\mathcal{S}$, to simulate the behavior of node x_1. By input- and system-equivalence of the second configuration of $\mathcal{S}$ and the first configuration of $\mathcal{R}$, the third configuration of $\mathcal{S}$ and the second configuration of $\mathcal{R}$ are output-, memory-, and message-equivalent. We can now repeat the same for nodes x_2, x_3, etc. Moreover, the message-equivalence allows us to deliver k messages of the form $R_{\to x}(z, \bar{a})$ in a transition of $\mathcal{S}$ when the corresponding transition in $\mathcal{R}$ would deliver k instances of (the same) message $R(\bar{a})$ to an active node x. $\square$

Claim 4.8. Let H be an input for $\boldsymbol{\mathcal{N}}$. Every run $\mathcal{S}$ of $\boldsymbol{\mathcal{M}}$ on $\langle H \rangle^{\mathcal{N}}$ can be converted to a run $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on H such that $\mathit{last}(\mathcal{R})$ and $\mathit{last}(\mathcal{S})$ are output- and memory-equivalent, and $\mathit{last}(\mathcal{S})$ has its messages included in $\mathit{last}(\mathcal{R})$.

Proof. First, some transitions of $\mathcal{S}$ might deliver a message of the form $R_{\to x}(z, \bar{a})$ without jointly delivering $\mathtt{do}_x(\,)$. Because node x is only simulated when $\mathtt{do}_x(\,)$ is delivered, message $R_{\to x}(z, \bar{a})$ is effectively lost. So, we can refrain from delivering $R_{\to x}(z, \bar{a})$ in this case, without compromising future message deliveries. After doing this modification for all deliveries of $\mathcal{S}$, we also drop any resulting (or preexisting) heartbeat transitions except the first transition, because they do not simulate nodes.[8] This results in a new run $\mathcal{S}'$ such that $\mathit{last}(\mathcal{S})$ and $\mathit{last}(\mathcal{S}')$ have the same output and memory facts, and such that the buffer of $\mathit{last}(\mathcal{S})$ is included in the buffer of $\mathit{last}(\mathcal{S}')$ when ignoring the $\mathtt{do}_x$-messages.

Next, some transitions i of $\mathcal{S}'$ might deliver two messages $\mathtt{do}_x(\,)$ and $\mathtt{do}_y(\,)$ with $x \neq y$. Such a transition i simulates multiple nodes in parallel. But in $\boldsymbol{\mathcal{M}}$, the simulated rules of each node x are guarded by $\mathtt{do}_x(\,)$, and these rules can only access relations of x itself. Hence, transition i can be converted to a sequence of transitions in which only one node is simulated at a time (in some arbitrary order), and in which each node receives the same messages that it received in i. This results in a new run $\mathcal{S}''$, where $\mathit{last}(\mathcal{S}')$ and $\mathit{last}(\mathcal{S}'')$ are exactly the same when ignoring the $\mathtt{do}_x$-messages.

Starting from the second transition, run $\mathcal{S}''$ simulates precisely one node in each transition. In the opposite fashion as in Claim 4.7, we can now convert $\mathcal{S}''$ to a run $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on input H so that $\mathit{last}(\mathcal{S}'')$ and $\mathit{last}(\mathcal{R})$ are output-, memory-, and message-equivalent. Note, $\mathit{last}(\mathcal{S})$ and $\mathit{last}(\mathcal{R})$ are output- and memory-equivalent, and $\mathit{last}(\mathcal{S})$ has its messages included in $\mathit{last}(\mathcal{R})$. $\square$

[8]This does not compromise the supply of $\mathtt{do}_x$-messages because they are sent in each transition.

Now we are ready for the actual confluence equivalence between $\boldsymbol{\mathcal{N}}$ and $\boldsymbol{\mathcal{M}}$, where each direction is shown in a separate claim:

Claim 4.9. If $\boldsymbol{\mathcal{M}}$ is confluent then $\boldsymbol{\mathcal{N}}$ is confluent.

Proof. Let H be an input for $\boldsymbol{\mathcal{N}}$. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two runs of $\boldsymbol{\mathcal{N}}$ on H, where $\mathit{last}(\mathcal{R}_1)$ contains an output fact $R(\bar{a})$ at some node $x \in \mathcal{N}$. We have to show that $\mathcal{R}_2$ can be extended to a run $\mathcal{R}_2'$ such that $\mathit{last}(\mathcal{R}_2')$ also contains fact $R(\bar{a})$ at x. Using Claim 4.7, we can make two runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\boldsymbol{\mathcal{M}}$ on $\langle H\rangle^{\mathcal{N}}$ such that for $i \in \{1, 2\}$, configurations $\mathit{last}(\mathcal{S}_i)$ and $\mathit{last}(\mathcal{R}_i)$ are output-, memory-, and message-equivalent. In particular, $\mathit{last}(\mathcal{S}_1)$ contains output fact $x.R(\bar{a})$. By confluence of $\boldsymbol{\mathcal{M}}$, run $\mathcal{S}_2$ can be extended to a run $\mathcal{S}_2'$ such that $\mathit{last}(\mathcal{S}_2)$ also contains $x.R(\bar{a})$. Lastly, extension $\mathcal{S}_2'$ gives rise to an extension $\mathcal{R}_2'$ of $\mathcal{R}_2$ such that $\mathit{last}(\mathcal{R}_2')$ is output- and memory-equivalent to $\mathit{last}(\mathcal{S}_2')$, and so $\mathit{last}(\mathcal{R}_2')$ contains $R(\bar{a})$ at x: the proof is similar to that of Claim 4.8, with the exception that the configurations in $\mathcal{S}_2'$ have their messages included in the corresponding configurations of $\mathcal{R}_2'$. This is sufficient to guarantee that $\mathcal{R}_2'$ can mimic the behavior of $\mathcal{S}_2'$. $\square$

Claim 4.10. If $\boldsymbol{\mathcal{N}}$ is confluent then $\boldsymbol{\mathcal{M}}$ is confluent.

Proof. Let I be an input for $\boldsymbol{\mathcal{M}}$. We have to show that $\boldsymbol{\mathcal{M}}$ is confluent on I.

First, suppose that I is ill-formed. If I does not contain a value for each relation $x.\texttt{Id}$ with $x \in \mathcal{N}$ then no output can ever be produced. Indeed, no message $\texttt{do}_x(\,)$ for any $x \in \mathcal{N}$ can be sent (and delivered), so no diffluence could arise because the nodes are not simulated. Otherwise, if I contains a value for each relation $x.\texttt{Id}$, because I is still ill-formed, it will be possible to send `error( )` (see Section 4.6.4.3). Then any run can be extended to produce all possible output facts, so potential diffluent behavior can always be corrected.

Now suppose that I is well-formed, which means there is an instance J isomorphic to I with $J|_{sch^{\mathcal{N}}} = \mathit{inst}^{\mathcal{N}}$ (see Section 4.6.2). Because transducer rules of $\boldsymbol{\mathcal{M}}$ only express generic queries, it is sufficient to show that $\boldsymbol{\mathcal{M}}$ is confluent on J. Let H be the (unique) input for $\boldsymbol{\mathcal{N}}$ for which $\langle H\rangle^{\mathcal{N}} = J$. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two runs of $\boldsymbol{\mathcal{M}}$ on J, where $\mathit{last}(\mathcal{S}_1)$ contains an output fact $x.R(\bar{a})$. We have to show that there is an extension of $\mathcal{S}_2$ for which the last configuration also contains $x.R(\bar{a})$.

First, applying Claim 4.8 to run $\mathcal{S}_1$, we can construct a run $\mathcal{R}_1$ of $\boldsymbol{\mathcal{N}}$ on input H such that $\mathit{last}(\mathcal{S}_1)$ and $\mathit{last}(\mathcal{R}_1)$ are output- and memory-equivalent. In particular, output fact $R(\bar{a})$ is at node x in $\mathit{last}(\mathcal{R}_1)$.

Next, suppose we can construct an extension $\mathcal{S}_2''$ of $\mathcal{S}_2$ and a run $\mathcal{R}_2''$ of $\boldsymbol{\mathcal{N}}$ on input H such that $\mathit{last}(\mathcal{S}_2'')$ and $\mathit{last}(\mathcal{R}_2'')$ are output-, memory-, *and* message-equivalent. If by chance $\mathit{last}(\mathcal{S}_2'')$ already contains $x.R(\bar{a})$ then we are ready. Otherwise, by output-equivalence of $\mathit{last}(\mathcal{S}_2'')$ and $\mathit{last}(\mathcal{R}_2'')$, fact $R(\bar{a})$ will not be at x in $\mathit{last}(\mathcal{R}_2'')$. But, by confluence of $\boldsymbol{\mathcal{N}}$, because $R(\bar{a})$ can be derived at x in $\mathcal{R}_1$ (see above), there is an extension of $\mathcal{R}_2''$ to derive $R(\bar{a})$ at x. By message-equivalence of $\mathit{last}(\mathcal{S}_2'')$ and $\mathit{last}(\mathcal{R}_2'')$, this extension can be simulated at the end of $\mathcal{S}_2''$ to derive $x.R(\bar{a})$, in a similar vein as in the proof of Claim 4.7.

We are left to construct the runs $\mathcal{S}_2''$ and $\mathcal{R}_2''$.

Message saturation Because transducer Π of $\boldsymbol{\mathcal{M}}$ is recursion-free (Π is simple), we can consider the maximum height n amongst derivation trees of Π, where the height is the largest number of edges on any path from a leaf to the root. Now, we extend $\mathcal{S}_2$ to a run $\mathcal{S}_2'$ by doing n additional transitions: each transition delivers the *entire* message buffer, and thus simulates all nodes in parallel where each node receives its entire (simulated) message buffer.[9] Because the sending rules of Π are message-positive and static, the message buffer of $\mathcal{M}$ — degenerated to a set — will monotonously grow. Because n is the maximum height of a derivation tree, $\mathit{last}(\mathcal{S}_2')$ contains all messages that could possibly be sent on input J.

Run of $\mathcal{N}$ Applying Claim 4.8 to $\mathcal{S}_2'$ (not to $\mathcal{S}_2$), we can construct a run $\mathcal{R}_2'$ of $\mathcal{N}$ on input H such that $\mathit{last}(\mathcal{S}_2')$ and $\mathit{last}(\mathcal{R}_2')$ are output- and memory-equivalent, and such that the messages of $\mathit{last}(\mathcal{S}_2')$ are included in $\mathit{last}(\mathcal{R}_2')$. We now show that actually all messages in the buffers of $\mathit{last}(\mathcal{R}_2')$ are simulated in the (single) buffer of $\mathit{last}(\mathcal{S}_2')$, except for maybe their precise cardinalities.

Let $S(\bar{b})$ be a message in the buffer of some node y in $\mathit{last}(\mathcal{R}_2')$. We can extract from $\mathcal{R}_2'$ a "global" derivation tree $\mathcal{T}$ to explain how $S(\bar{b})$ was sent to y: this is like a normal derivation tree, except that we also say at which node a message was derived. Letting Π be the single transducer of $\mathcal{M}$, and letting x be the node in the root of $\mathcal{T}$ (i.e., x sends $S(\bar{b})$ to y), the natural correspondence between Π and $\mathcal{N}$ allows us to convert $\mathcal{T}$ into a derivation tree $\mathcal{T}'$ of Π, to explain how to send the message $S_{\to y}(x, \bar{b})$. Because sending rules are message-positive and static, this tree $\mathcal{T}'$ is successfully executed in the last n transitions of $\mathcal{S}_2'$, so that $S_{\to y}(x, \bar{b})$ is in the message buffer of $\mathit{last}(\mathcal{S}_2')$, as desired.

Obtain message-equivalence Consider the extension $\mathcal{R}_2''$ of $\mathcal{R}_2'$ that is obtained by letting each node, in some arbitrary order, receive its entire message buffer from configuration $\mathit{last}(\mathcal{R}_2')$. Similarly, consider the extension $\mathcal{S}_2''$ of $\mathcal{S}_2'$ obtained by letting each simulated node, in the same order as in $\mathcal{R}_2''$, receive its entire message buffer as it is simulated by configuration $\mathit{last}(\mathcal{S}_2')$.

As we have seen above, $\mathit{last}(\mathcal{R}_2')$ and $\mathit{last}(\mathcal{S}_2')$ essentially represent the same messages in the buffer of each node, except that the cardinalities might be different. But since duplicate messages are collapsed upon delivery, the nodes do not observe the difference in cardinalities when the above two extensions are performed. Hence, configurations $\mathit{last}(\mathcal{R}_2'')$ and $\mathit{last}(\mathcal{S}_2'')$ are output- and memory-equivalent. But they are also message-equivalent as we now explain. First, for a node $y \in \mathcal{N}$, the extensions deliver equivalent message sets to y. Hence, in both extensions, node y in turn sends equivalent message sets. And because node y has its entire message buffer (of configurations $\mathit{last}(\mathcal{R}_2')$ and $\mathit{last}(\mathcal{S}_2')$) emptied during the delivery, the cardinalities of messages in $\mathit{last}(\mathcal{R}_2'')$ and $\mathit{last}(\mathcal{S}_2'')$ are the same. $\square$

4.7 Small Model Property

Let $\mathcal{N}$ be a simple single-node transducer network. We establish a small model property: if $\mathcal{N}$ is diffluent, then $\mathcal{N}$ is diffluent on an input whose active domain size

[9]We assume run $\mathcal{S}_2$ contains at least one transition, so that all $\mathtt{do}_x$-messages are available in the buffer of $\mathit{last}(\mathcal{S}_2)$.

is upper bounded by an expression purely over syntactical properties of $\mathcal{N}$. For this result, we use all syntactical restrictions of simple transducer networks.

Let Π and Υ denote respectively the single transducer and its schema in $\mathcal{N}$. Like in Section 4.6.1, an input for $\mathcal{N}$ is an instance I over Υ_{in}, and a configuration of $\mathcal{N}$ is a pair (s, b) where s is a transducer state and b is a multiset of facts over Υ_{msg}. Moreover, the sending rules have no explicit addressee variable in their head, and Υ_{sys} will not be used in any rule. Such a network can always be obtained by applying the simulation in Section 4.6.

4.7.1 Syntactical Quantities

Consider the following syntactically defined quantities about $\mathcal{N}$:

- the length $\textsc{p}$ of the longest path in the positive dependency graph of Π (defined in Section 4.5.2), where the length of a path is measured as the number of edges on this path;
- the largest number $\textsc{b}$ of positive body atoms in any rule of Π;
- the largest arity $\textsc{i}$ among input relations;
- the largest arity $\textsc{o}$ among output relations;
- the number $\textsc{c}$ of different output and memory facts that can be made with values in A, where $A \subseteq \mathbf{dom}$ is an arbitrary set with $|A| = \textsc{o}$.

Now, let *sizeDom*($\mathcal{N}$) abbreviate the expression $2\textsc{icb}^{\textsc{p}}$. We have the following small model property:

Proposition 4.11. If $\mathcal{N}$ is diffluent, then $\mathcal{N}$ is diffluent on an instance J over Υ_{in} for which $|adom(J)| \leq sizeDom(\mathcal{N})$.

The rest of this section is devoted to showing this result.

4.7.2 Proof Outline

Here we sketch the proof of Proposition 4.11. The details are provided by the following subsections. The proof technique is inspired by *pseudoruns* from Deutsch et al. [29], although it was adapted to deal with the diffluence problem and to deal with message buffers (multisets). Let $\mathcal{N}$, Π and Υ be as above, and recall the syntactical quantities of $\mathcal{N}$ from Section 4.7.1.

First we give some additional terminology and notations. Let $A \subseteq \mathbf{dom}$. We call a fact $\boldsymbol{g}$ an *A-fact* if the values in $\boldsymbol{g}$ are a subset of A. For a set of facts H, we write $H^{[A]}$ to denote the subset of all A-facts in H. Note, nullary facts of H are always in $H^{[A]}$.

Let I be an input for $\mathcal{N}$. Suppose $\mathcal{N}$ is diffluent on I, i.e., there are two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\mathcal{N}$ on I such that $last(\mathcal{R}_1)$ contains an output fact $\boldsymbol{f}$ that is not in $last(\mathcal{R}_2)$, and there is no extension $\mathcal{R}_2'$ of $\mathcal{R}_2$ such that $last(\mathcal{R}_2')$ contains $\boldsymbol{f}$. Abbreviate $C = adom(\boldsymbol{f})$, the set of values in $\boldsymbol{f}$. Note, $|C| \leq \textsc{o}$.

In Section 4.7.3, for $i = 1, 2$, we will select a subset of input facts $K_i \subseteq I$ that are needed to make all output and memory C-facts of run $\mathcal{R}_i$, with the property

$|K_i| \leq \mathbf{cb}^{\mathbf{P}}$. This gives the instances K_1 and K_2. Note, $C \subseteq \mathit{adom}(K_1)$ because $\boldsymbol{f}$ is created in $\mathcal{R}_1$. Define

$$J = I^{[\mathit{adom}(K_1) \cup \mathit{adom}(K_2)]}.$$

Note, $|\mathit{adom}(J)| \leq 2\mathbf{icb}^{\mathbf{P}} = \mathit{sizeDom}(\boldsymbol{\mathcal{N}})$.

Next, in Section 4.7.4, for $i = 1, 2$, we will construct a run $\mathcal{S}_i$ on input J with the following properties:

- $\mathit{last}(\mathcal{S}_i)$ and $\mathit{last}(\mathcal{R}_i)$ contain precisely the same output and memory C-facts;
- every extension $\mathcal{S}_i'$ of $\mathcal{S}_i$ gives rise to an extension $\mathcal{R}_i'$ of $\mathcal{R}_i$ such that $\mathit{last}(\mathcal{S}_i')$ and $\mathit{last}(\mathcal{R}_i')$ again contain precisely the same output and memory C-facts.

This gives the runs $\mathcal{S}_1$ and $\mathcal{S}_2$ on J. The focus on output and memory C-facts is mainly the result of the message-boundedness constraint. Since $\boldsymbol{f}$ is an output C-fact, the first property above tells us that $\mathit{last}(\mathcal{S}_1)$ contains $\boldsymbol{f}$ and $\mathit{last}(\mathcal{S}_2)$ does not. Moreover, if $\mathcal{S}_2$ can be extended to a run $\mathcal{S}_2'$ such that $\mathit{last}(\mathcal{S}_2')$ contains $\boldsymbol{f}$, then the second property above would tell us that $\mathcal{R}_2$ can be extended to a run $\mathcal{R}_2'$ such that $\mathit{last}(\mathcal{R}_2')$ also contains $\boldsymbol{f}$. But the latter is not possible by assumption on $\mathcal{R}_2$. Hence, $\mathcal{S}_2'$ does not exist, and $\boldsymbol{\mathcal{N}}$ is diffluent on the instance J, whose active domain size is upper bounded by $\mathit{sizeDom}(\boldsymbol{\mathcal{N}})$, as desired.

4.7.3 Input Selection

Consider the symbols defined in Sections 4.7.1 and 4.7.2. Let $\mathcal{R}$ be either $\mathcal{R}_1$ or $\mathcal{R}_2$. In this section, we select an instance $K \subseteq I$ that is needed to make all output and memory C-facts of $\mathcal{R}$, and such that $|K| \leq \mathbf{cb}^{\mathbf{P}}$.

We construct a derivation history of each output and memory C-fact in $\mathcal{R}$: this includes the rules and valuations that derive the C-facts, and it also includes the derivation histories of messages recursively needed to make those C-facts.

4.7.3.1 Derivation History

Let $\boldsymbol{g}$ be an output or memory C-fact derived during $\mathcal{R}$. By inflationarity of Π, the derivation of $\boldsymbol{g}$ happens in some unique transition i. We choose *one* pair (φ, V) of a rule φ and satisfying valuation V such that $\boldsymbol{g}$ is derived during transition i by applying V to φ. Let us call (φ, V) a *derivation pair*. If φ contains a (positive) body message atom $\boldsymbol{a}$, the message $\boldsymbol{h} = V(\boldsymbol{a})$ is required by (φ, V) to derive $\boldsymbol{g}$. Similarly as we did for $\boldsymbol{g}$, we can go to a transition in which $\boldsymbol{h}$ was derived and select there also *one* pair (φ', V') to derive $\boldsymbol{h}$. We can again recursively repeat the selection of derivation pairs for any message facts needed by (φ', V').

Formally, after the selection of derivation pairs, we obtain a function $\mathit{hist}_{\mathcal{R}}$ that maps each pair $(i, \boldsymbol{g})$ to a derivation pair for $\boldsymbol{g}$, where $\boldsymbol{g}$ is an output or memory C-fact or a recursively needed message derived in transition i. We also have a set $\mathit{msg}_{\mathcal{R}}$ containing triples $(k, \boldsymbol{h}, l)$ to indicate that a valuation in transition l needs the message $\boldsymbol{h}$ to arrive, and that $\boldsymbol{h}$ itself is sent in (an earlier) transition k. These triples indicate the timing of the needed messages.

Now, let K denote the subset of all input facts $\boldsymbol{h} \in I$ for which there exists a pair $(i, \boldsymbol{g})$ in the domain of $\mathit{hist}_{\mathcal{R}}$, denoting $\mathit{hist}_{\mathcal{R}}(i, \boldsymbol{g}) = (\varphi, V)$, such that $\boldsymbol{h} \in V(\mathit{pos}_{\varphi})$.

In words: K contains the (positive) input facts needed by the derivation history of all output and memory C-facts in $\mathcal{R}$ (and any needed messages). We now show $|K| \leq \textsc{cb}^{\mathbf{P}}$. First, let us fix one output or memory C-fact $\boldsymbol{g}$. Any chain of messages recursively needed by $\boldsymbol{g}$ has length at most $\textsc{p}$ by recursion-freeness of Π. Moreover, in the worst case, each message recursively requires $\textsc{b}$ other messages. Therefore, the number of input facts needed by $\boldsymbol{g}$ alone is bounded by $\textsc{b}^{\mathbf{P}}$. And since at most $\textsc{c}$ different output and memory C-facts are created in $\mathcal{R}$, we overall have that $|K| \leq \textsc{cb}^{\mathbf{P}}$, as desired.

4.7.3.2 Natural Properties

Section 4.7.3.1 allows much liberty in which $hist_{\mathcal{R}}$ and $msg_{\mathcal{R}}$ may be chosen. We now demand that some natural properties hold on $msg_{\mathcal{R}}$, upon which the construction in Section 4.7.4 crucially depends.

First, based on $msg_{\mathcal{R}}$, for each transition i of $\mathcal{R}$, we define the message multisets β_i, γ_i, and $\mathcal{E}_i$ as follows, with the intuition provided below:

- the multiplicity of a message $\boldsymbol{h}$ in β_i is the number of triples $(k, \boldsymbol{h}, l) \in msg_{\mathcal{R}}$ for which $l = i$;
- the multiplicity of a message $\boldsymbol{h}$ in γ_i is the number of triples $(k, \boldsymbol{h}, l) \in msg_{\mathcal{R}}$ for which $k < i$ and $i \leq l$;
- the multiplicity of a message $\boldsymbol{h}$ in $\mathcal{E}_i$ is the number of triples $(k, \boldsymbol{h}, l) \in msg_{\mathcal{R}}$ for which $k = i$.

Let $\rho_1, \ldots, \rho_n, \rho_{n+1}$ denote the sequence of configurations of $\mathcal{R}$, where n is the number of transitions. Intuitively, β_i contains the messages needed in transition i; γ_i contains the needed messages that are sent before configuration ρ_i and that travel through configuration ρ_i to be delivered in transition i (when $l = i$) or later (when $i < l$); and, $\mathcal{E}_i$ contains the needed messages that should be sent in transition i.

In the technical report [19], we show that $hist_{\mathcal{R}}$ and $msg_{\mathcal{R}}$ can be chosen so that the following properties are satisfied, with the intuition provided below:

1. $\gamma_i \sqsubseteq b_i^{\mathcal{R}}$ for each transition i of $\mathcal{R}$, where $\rho_i = (s_i^{\mathcal{R}}, b_i^{\mathcal{R}})$;
2. β_i is a set for each transition i of $\mathcal{R}$, i.e., for each $(k, \boldsymbol{h}, i)$ and $(k', \boldsymbol{h}, i)$ in $msg_{\mathcal{R}}$, we have $k = k'$;
3. $\mathcal{E}_i = \gamma_{i+1} \cap \delta_i^{\mathcal{R}}$, where $\delta_i^{\mathcal{R}}$ is the set of messages sent in transition i of $\mathcal{R}$.

Intuitively, property 1 means that all needed messages whose transmission overlaps in time, also jointly occur in the message buffer, with the correct cardinalities. Property 2 means that if multiple derivation pairs in the same transition need the same message, the same origin of this message is used. Lastly, property 3 implies that for each needed message, its origin transition is chosen as late as possible: whenever for some needed message $\boldsymbol{h} \in \gamma_{i+1}$ we have the opportunity to explain its origin in transition i (i.e., $\boldsymbol{h} \in \delta_i^{\mathcal{R}}$), we take this opportunity (i.e., $\boldsymbol{h} \in \mathcal{E}_i$).

4.7.4 Run Projection

Consider the symbols defined in Section 4.7.2. Let $\mathcal{R}$ be either $\mathcal{R}_1$ or $\mathcal{R}_2$. We construct a run $\mathcal{S}$ on input J with the following properties:

- $\mathit{last}(\mathcal{S})$ and $\mathit{last}(\mathcal{R})$ contain the same output and memory C-facts;
- every extension $\mathcal{S}'$ of $\mathcal{S}$ gives rise to an extension $\mathcal{R}'$ of $\mathcal{R}$ such that $\mathit{last}(\mathcal{S}')$ and $\mathit{last}(\mathcal{R}')$ again contain precisely the same output and memory C-facts.

To reduce the technical complexity of this section, we sketch the important ideas; the full details are in [19]. First, it can be shown that the second property above holds when the first property holds *and* when the message buffer of $\mathit{last}(\mathcal{S})$ is included in the message buffer of $\mathit{last}(\mathcal{R})$. Intuitively, this inclusion allows every extension $\mathcal{S}'$ of $\mathcal{S}$ to be converted to an extension $\mathcal{R}'$ of $\mathcal{R}$ so that the buffer of $\mathcal{S}'$ *remains* included in the buffer of $\mathcal{R}'$, allowing $\mathcal{R}'$ to make precisely the same message deliveries as $\mathcal{S}'$.

Now we sketch the main idea in the construction of $\mathcal{S}$. For run $\mathcal{R}$, let $\mathit{hist}_{\mathcal{R}}$, $\mathit{msg}_{\mathcal{R}}$, β_i, γ_i, and $\mathcal{E}_i$ be as defined in Section 4.7.3. We assume that $\mathit{msg}_{\mathcal{R}}$ satisfies the properties given in Section 4.7.3.2. Run $\mathcal{S}$ will be a projected version of $\mathcal{R}$: we do the same number of transitions as $\mathcal{R}$, and perform the message deliveries selected by $\mathit{msg}_{\mathcal{R}}$, so that the output and memory C-facts of $\mathcal{R}$ are faithfully created. One caveat, however, is that some transitions of $\mathcal{S}$ should sometimes deliver more messages than just those of $\mathit{msg}_{\mathcal{R}}$ because we want the message buffer of $\mathcal{S}$ to be included in the corresponding message buffer of $\mathcal{R}$ (see above). Let n be the number of transitions in $\mathcal{R}$. For each $i \in \{1, \dots, n+1\}$, we denote the i^{th} configuration of $\mathcal{R}$ and $\mathcal{S}$ respectively as $\rho_i = (s_i^{\mathcal{R}}, b_i^{\mathcal{R}})$ and $\sigma_i = (s_i^{\mathcal{S}}, b_i^{\mathcal{S}})$. We want to inductively specify the message deliveries of $\mathcal{S}$ so that the following properties are satisfied for each $i \in \{1, \dots, n+1\}$:

1. $s_i^{\mathcal{S}}$ and $s_i^{\mathcal{R}}$ have the same output and memory C-facts;
2. message buffer $b_i^{\mathcal{S}}$ a submultiset of message buffer $b_i^{\mathcal{R}}$; and,
3. γ_i is a submultiset of the message buffer $b_i^{\mathcal{S}}$.

The need for the first two properties was already explained above, and property 3 intuitively says that all messages required by $\mathit{msg}_{\mathcal{R}}$ are in flight when they should be. For the base case ($i = 1$), properties 1 and 2 are satisfied because ρ_1 and σ_1 are start configurations, in which there are no output or memory facts and the message buffers are empty; and, property 3 is satisfied because $\gamma_1 = \emptyset$ (no needed messages can be sent before transition 1). For the induction hypothesis, we assume that the properties are satisfied for ρ_i and σ_i. For the inductive step, we have to satisfy them for ρ_{i+1} and σ_{i+1}. In transition i of $\mathcal{S}$, which transforms σ_i into σ_{i+1}, we deliver the following message *multiset*:

$$m_i^{\mathcal{S}} = \big(b_i^{\mathcal{S}} \setminus (\gamma_i \setminus \beta_i)\big) \cap m_i^{\mathcal{R}},$$

where $m_i^{\mathcal{R}}$ denotes the message multiset delivered in transition i of $\mathcal{R}$, and where we use multiset difference and intersection. Intuitively, the set β_i of messages needed in transition i, is delivered, but we have to protect the messages in $\gamma_i \setminus \beta_i$, because they are needed *after* transition i. All remaining facts can be delivered, on condition that they are delivered in $\mathcal{R}$. It can be shown that under this definition of $m_i^{\mathcal{S}}$, the properties 1, 2, and 3 are satisfied [19].

4.8 Decidability

Note, Proposition 4.11 does not immediately give decidability of diffluence for simple transducer networks because even on a fixed input instance, we still have an infinite state system since the message buffers have no size limit. In this section we show that diffluence of simple single-node transducer networks is decidable. In Section 4.8.1, we give a nondeterministic exponential time (NEXPTIME) decision procedure. In Section 4.8.2, we give a NEXPTIME lower bound, thus making the problem NEXPTIME-complete. This also makes diffluence for multi-node networks NEXPTIME-complete: *(i)* the NEXPTIME upper bound follows from the PTIME reduction to a single-node network (Section 4.6), and *(ii)* the NEXPTIME lower bound is because single-node networks are a special case of multi-node networks.

4.8.1 Decision Procedure

In Section 4.8.1.1 we give the description of the decision procedure. Next, Sections 4.8.1.2 and 4.8.1.3 show the correctness, and Section 4.8.1.4 investigates the complexity.

Let $\boldsymbol{\mathcal{N}}$ be a simple single-node transducer network. Let Π and Υ respectively denote the transducer and transducer schema of $\boldsymbol{\mathcal{N}}$. We use the syntactical simplifications for single-node networks from Section 4.6.1.

4.8.1.1 Procedure

We give a nondeterministic procedure for checking whether $\boldsymbol{\mathcal{N}}$ is diffluent. We say that the procedure *accepts* $\boldsymbol{\mathcal{N}}$ if at least one computation branch has found evidence that $\boldsymbol{\mathcal{N}}$ is diffluent, in which case that branch executes the *accept*-statement. A branch can also stop early by executing *reject*.

Let P, B, C, and $\mathit{sizeDom}(\boldsymbol{\mathcal{N}})$ be as defined in Section 4.7.1. Consider the expression $\mathbf{runLen} = \mathrm{C}\mathrm{B}^{\mathbf{P}} + \mathrm{C}$. For $A \subseteq \mathbf{dom}$, we say that a fact $\boldsymbol{f}$ is an A-fact if $\mathit{adom}(\boldsymbol{f}) \subseteq A$. The procedure does the following steps, in order:

1. [Input] Guess an input instance I for $\boldsymbol{\mathcal{N}}$ with $|\mathit{adom}(I)| \leq \mathit{sizeDom}(\boldsymbol{\mathcal{N}})$.

2. [Two runs] Guess two runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\boldsymbol{\mathcal{N}}$ on input I, such that both runs do at most **runLen** transitions. Concretely, such a run is guessed by first choosing how much transitions are done ($\leq$ **runLen**), and by choosing for each transition which submultiset of the message buffer should be delivered. For simulating these runs, it is sufficient to continuously store only the last configuration, and not all previous configurations.

3. [Output] Choose an output fact $\boldsymbol{f}$ in $\mathit{last}(\mathcal{S}_1)$ that is not in $\mathit{last}(\mathcal{S}_2)$. If no such fact can be chosen, then *reject*.

4. [Extension] Denote $C = \mathit{adom}(\boldsymbol{f})$. We extend $\mathcal{S}_2$ by doing $\mathrm{P}+1$ more transitions, and in each transition we deliver the entire message buffer. If no output or memory C-fact is created in this extension, then *accept* and else *reject*.

4.8.1.2 Correctness Part 1

Suppose $\boldsymbol{\mathcal{N}}$ is diffluent. We show that the procedure accepts. First, by the small model property (Proposition 4.11), there is an input I for $\boldsymbol{\mathcal{N}}$ such that $|adom(I)| \leq sizeDom(\boldsymbol{\mathcal{N}})$ and $\boldsymbol{\mathcal{N}}$ is diffluent on input I. Thus, there are two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\boldsymbol{\mathcal{N}}$ on input I such that $last(\mathcal{R}_1)$ contains an output fact $\boldsymbol{f}$ that is not in $last(\mathcal{R}_2)$, and there is no extension of $\mathcal{R}_2$ in which $\boldsymbol{f}$ can be output. The procedure can guess an instance I' that is isomorphic to I, but for notational simplicity we may assume that simply $I' = I$.

Denote $C = adom(\boldsymbol{f})$. By inflationarity of Π, we can always extend $\mathcal{R}_2$ to a run $\mathcal{R}_2'$ such that no more output or memory C-facts can be created in any extension of $\mathcal{R}_2'$. By assumption on $\mathcal{R}_2$, configuration $last(\mathcal{R}_2')$ does not contain $\boldsymbol{f}$. Now, there exists two runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\boldsymbol{\mathcal{N}}$ on input I with at most **runLen** transitions such that $last(\mathcal{S}_1)$ and $last(\mathcal{S}_2)$ contain exactly the same output and memory C-facts as respectively $last(\mathcal{R}_1)$ and $last(\mathcal{R}_2')$; these details are in [19].[10] So, the procedure can guess $\mathcal{S}_1$ and $\mathcal{S}_2$, and can choose $\boldsymbol{f}$ as an output that is in $last(\mathcal{S}_1)$ but not in $last(\mathcal{S}_2)$.

Next, let $\mathcal{S}_2'$ denote the extension of $\mathcal{S}_2$ as performed by the procedure: we do $\mathbf{P}+1$ additional transitions, in each of which we deliver the entire message buffer. We show that no more output or memory C-facts are created in this extension, so that the procedure accepts, as desired. Towards a proof by contradiction, suppose that there is some new transition $i \in \{1, \ldots, \mathbf{P}+1\}$ that derives an output or memory C-fact $\boldsymbol{g}$, with the assumption that i is the *first* such transition. Let (φ, V) be a derivation pair for $\boldsymbol{g}$ in transition i. We show that $\mathcal{R}_2'$ can be extended to output $\boldsymbol{g}$ as well, giving the desired contradiction.

Extend $\mathcal{R}_2'$ to a run $\mathcal{R}_2''$ by doing $\mathbf{P}+1$ more transitions in each of which we also deliver the entire message buffer. We show that V is satisfying for φ in the last transition of $\mathcal{R}_2''$. We consider the different body components of φ:

- The input literals of φ are satisfied under V in the last transition of $\mathcal{R}_2''$ because $\mathcal{S}_2'$ and $\mathcal{R}_2''$ have the same input I.

- Let $\boldsymbol{h} \in V(pos_\varphi)|_{\Upsilon_{\text{msg}}}$. Because V is satisfying for φ in $\mathcal{S}_2'$, message $\boldsymbol{h}$ can be sent. Then a derivation tree for $\boldsymbol{h}$ can be extracted from $\mathcal{S}_2'$, and by using that sending rules are message-positive and static, it can be shown [19] that this tree also sends $\boldsymbol{h}$ in the second-to-last transition of $\mathcal{R}_2''$, causing $\boldsymbol{h}$ to be delivered in the last transition.

- Let $\boldsymbol{h} \in V(pos_\varphi)|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$. We have to show that $\boldsymbol{h}$ is available in the last transition of $\mathcal{R}_2''$. First, because $\boldsymbol{g}$ is a C-fact, the message-boundedness of φ implies that $\boldsymbol{h}$ is a C-fact. Because $\boldsymbol{g}$ is assumed to be the first output or memory C-fact to be created in the extension of $\mathcal{S}_2$, fact $\boldsymbol{h}$ is in $last(\mathcal{S}_2)$. Thus $\boldsymbol{h}$ is in $last(\mathcal{R}_2')$ by construction of $\mathcal{S}_2$, so $\boldsymbol{h}$ can be read in the last transition of $\mathcal{R}_2''$.

- Let $\boldsymbol{h} \in V(neg_\varphi)|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$. We have to show that $\boldsymbol{h}$ is not read in the last transition of $\mathcal{R}_2''$. Like in the previous case, $\boldsymbol{h}$ is a C-fact. It is sufficient to show

[10]For $i = 1, 2$, the intuition about $\mathcal{S}_i$ is again to mark in $\mathcal{R}_i$ all messages recursively needed to create the C-facts (like in Section 4.7.3.1), and then to simply omit all transitions that do not contribute C-facts nor needed messages, leaving at most **runLen** transitions behind.

that $\boldsymbol{h}$ is not in $\mathit{last}(\mathcal{R}'_2)$ because no output or memory C-fact can be created in an extension of $\mathcal{R}'_2$, including $\mathcal{R}''_2$. Now, because V is satisfying for φ in $\mathcal{S}'_2$, the inflationarity of transducer Π implies that $\boldsymbol{h}$ is not in $\mathit{last}(\mathcal{S}_2)$. Thus $\boldsymbol{h}$ is not in $\mathit{last}(\mathcal{R}'_2)$ by construction of $\mathcal{S}_2$.

- Also, the nonequalities of φ are satisfied under V in $\mathcal{R}''_2$ because they are satisfied in $\mathcal{S}'_2$.

4.8.1.3 Correctness Part 2

Suppose that the procedure accepts. We show that $\boldsymbol{\mathcal{N}}$ is diffluent. Because the procedure accepts, there is a computation branch that has done the following. The branch has guessed an input instance I for $\boldsymbol{\mathcal{N}}$ such that $|\mathit{adom}(I)| \leq \mathit{sizeDom}(\boldsymbol{\mathcal{N}})$. Next, the branch has guessed two runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\boldsymbol{\mathcal{N}}$ on input I, and has been able to choose an output fact $\boldsymbol{f}$ in $\mathit{last}(\mathcal{S}_1)$ that is not in $\mathit{last}(\mathcal{S}_2)$. Denote $C = \mathit{adom}(\boldsymbol{f})$. Lastly, the branch has extended $\mathcal{S}_2$ to a run $\mathcal{S}'_2$ by doing $\textsc{p}+1$ additional transitions in which the entire message buffer is delivered each time, and the procedure has observed that no output or memory C-facts were created in this extension, including $\boldsymbol{f}$.

To show that $\boldsymbol{\mathcal{N}}$ is diffluent, it is sufficient to show that no output or memory C-facts (including $\boldsymbol{f}$) can be created in any extension of $\mathcal{S}'_2$. Towards a proof by contradiction, suppose that an output or memory C-fact $\boldsymbol{g}$ can be created in an extension $\mathcal{S}''_2$ of $\mathcal{S}'_2$, by means of a derivation pair (φ, V). Let us assume that $\boldsymbol{g}$ is the *first* such output or memory C-fact. We show that V is satisfying for φ in the last transition of $\mathcal{S}'_2$ itself, so that $\boldsymbol{g}$ would already have been created in $\mathcal{S}'_2$, which is the desired contradiction. To show that V is satisfying in $\mathcal{S}'_2$, we proceed similarly as in the first correctness proof above. We note the differences:

- Let $\boldsymbol{h} \in V(\mathit{pos}_\varphi)|_{\Upsilon_{\mathrm{out}} \cup \Upsilon_{\mathrm{mem}}}$. We have to show that $\boldsymbol{h}$ is available in the last transition of $\mathcal{S}'_2$. Like before, $\boldsymbol{h}$ is a C-fact by message-boundedness. Because $\boldsymbol{g}$ is assumed to be the first output or memory C-fact to be created in the extension of $\mathcal{S}'_2$, it must be that $\boldsymbol{h}$ is in $\mathit{last}(\mathcal{S}'_2)$. Moreover, because the decision procedure has not observed the creation of an output or memory C-fact in the transitions of $\mathcal{S}'_2$ after $\mathit{last}(\mathcal{S}_2)$, fact $\boldsymbol{h}$ is in $\mathit{last}(\mathcal{S}_2)$. Hence, $\boldsymbol{h}$ can be read in the last transition of $\mathcal{S}'_2$.

- Let $\boldsymbol{h} \in V(\mathit{neg}_\varphi)|_{\Upsilon_{\mathrm{out}} \cup \Upsilon_{\mathrm{mem}}}$. We have to show that $\boldsymbol{h}$ is not present in the last transition of $\mathcal{S}'_2$. Because V is satisfying for φ in $\mathcal{S}''_2$, fact $\boldsymbol{h}$ must be absent there. Hence, by inflationarity, $\boldsymbol{h}$ is not in $\mathit{last}(\mathcal{S}_2)$, and thus $\boldsymbol{h}$ is not read in the last transition of $\mathcal{S}'_2$.

4.8.1.4 Time Complexity

Here we analyze the time complexity of each computation branch of the decision procedure. We sketch how the procedure might be implemented in an imperative programming language where blocks of code can be guarded by a nondeterministic choice, that could either execute the corresponding block or skip it. In this framework, we show that each branch uses at most single-exponential time, making the decision procedure be in NEXPTIME.

Encoding First we make some remarks about the encoding. We use the encoding of transducer neworks from Section 4.4.3. Let $|\mathcal{N}|$ denote the input size. Now, consider the syntactical quantities defined in Section 4.7.1. The quantities $\textsc{i}$ and $\textsc{o}$ are upper bounded by $|\mathcal{N}|$ because all input and output relations are used in rules (whose atoms are written in full). The quantities $\textsc{b}$ and $\textsc{p}$ are also upper bounded by $|\mathcal{N}|$. Letting n denote the number of different transducer relations, again upper bounded by $|\mathcal{N}|$, the number $\textsc{c}$ is upper bounded by $n\textsc{o}^{\textsc{o}} = n2^{\textsc{o}\log\textsc{o}}$, which is single-exponential in $|\mathcal{N}|$. Hence, $\mathit{sizeDom}(\mathcal{N})$ is single-exponential in $|\mathcal{N}|$.

Over all relations of $\mathcal{N}$, let **numFc** denote the total number of different facts that can be made with $\mathit{sizeDom}(\mathcal{N})$ unique domain values. Note, **numFc** is single-exponential in $|\mathcal{N}|$.

Input For each input instance I' for $\mathcal{N}$ with $|adom(I')| \leq \mathit{sizeDom}(\mathcal{N})$, the procedure can guess an isomorphic instance I. Because $\mathit{sizeDom}(\mathcal{N})$ is single-exponential in $|\mathcal{N}|$, an active domain value of I can be represented as a number encoded by p bits, where p is polynomial in $|\mathcal{N}|$. We omit the algorithmic details to guess I.

Two runs Next, the procedure needs to guess two runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\mathcal{N}$ on I, such that each run does at most **runLen** transitions. We describe how to guess one run $\mathcal{S} \in \{\mathcal{S}_1, \mathcal{S}_2\}$; the other run can be guessed similarly after the first one.

To guess $\mathcal{S}$, we do a for-loop with **runLen** iterations in which we incrementally modify a configuration, starting with the start configuration. Note that **runLen** is single-exponential in $|\mathcal{N}|$. In each iteration, we choose whether or not we do a transition. To do a transition, we select a submultiset m of the message buffer to deliver. The size of the message buffer is at most **runLen** $\cdot$ **numFc**, so this selection can be done in single-exponential time. We are left to show that simulating the subsequent local transition can be done in single-exponential time. Let J denote the transducer state in the last configuration obtained. Now, for all transducer rules φ, for all valuations V for φ, if V is satisfying for φ with respect to $J \cup \mathit{set}(m)$ then derive $\boldsymbol{g} = V(\mathit{head}_\varphi)$. The number of rules is linear in $|\mathcal{N}|$. For one rule, the number of variables is also linear in $|\mathcal{N}|$. Hence, the number of valuations for one rule, using values in $adom(I)$, is single-exponential in $|\mathcal{N}|$. Finally, checking whether a valuation V is satisfying for a rule φ is done by *(i)* checking that the nonequalities are satisfied, which can be done in polynomial time; and, *(ii)* going over all body literals $\boldsymbol{l}$ of φ, applying V, and checking whether $J \cup \mathit{set}(m) \models V(\boldsymbol{l})$, which can be done in single-exponential time because $|J \cup \mathit{set}(m)| \leq$ **numFc**.

Output The procedure then selects an output fact $\boldsymbol{f}$ in $\mathit{last}(\mathcal{S}_1)$ that is not in $\mathit{last}(\mathcal{S}_2)$. Because the number of output facts in either configuration is at most **numFc**, we can select $\boldsymbol{f}$ in single-exponential time. Possibly $\mathit{last}(\mathcal{S}_2)$ has at least the output facts of $\mathit{last}(\mathcal{S}_1)$, in which case the procedure does *reject*. Otherwise, we continue.

Extension In the last step, the procedure extends $\mathcal{S}_2$ with $\textsc{p}+1$ transitions, in each of which we deliver the entire message buffer. The message buffer in $\mathit{last}(\mathcal{S}_2)$ contains at most **runLen**$\cdot$**numFc** facts, and all the subsequent buffers in the extension contain at most **numFc** facts because the buffer has degenerated to a set. Hence, we can apply the same time complexity analysis for simulating the local transitions as above.

Letting $C = adom(\boldsymbol{f})$, checking whether a newly derived output or memory fact is a C-fact can be done in polynomial time. Overall, simulating the additional $\textsc{p} + 1$ transitions can be done in single-exponential time.

4.8.2 Complexity Lower Bound

In Section 4.8.1 we gave a NEXPTIME upper bound on the time complexity for deciding diffluence for simple single-node transducer networks. In this section, we complement this result by giving a NEXPTIME lower bound, making the decision problem NEXPTIME-complete. Concretely, we show that any problem in NEXPTIME is polynomial time reducible to this decision problem.

Let A be a problem from NEXPTIME. Formally, A is a set of words over some alphabet Σ, and there exists a nondeterministic Turing machine M such that *(i)* for each word w over Σ, M accepts w iff $w \in A$; and, *(ii)* every computation trace of M on an input w over Σ eventually halts and uses at most $O(2^{|w|^k})$ steps, where k is a constant specific to M [53].

Fix some word w over Σ. We construct a simple single-node transducer network $\boldsymbol{\mathcal{N}}$ for w such that $\boldsymbol{\mathcal{N}}$ is diffluent iff M accepts w. We use the syntactical simplifications of single-node networks (Section 4.6.1).

4.8.2.1 Turing Machine

First, following the conventions in Sipser [53], the Turing machine M is given as a tuple

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

where Q is the set of states, Σ is the alphabet of the language A, Γ is the tape-alphabet (satisfying $\Sigma \subseteq \Gamma$), δ is the transition function, $q_0 \in Q$ is the start state, $q_{\text{accept}} \in Q$ is the accept state, and $q_{\text{reject}} \in Q$ is the reject state. Function δ has the signature $Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$, where L and R indicate whether the tape head moves left or right after performing a transition.

4.8.2.2 Construction

We now define the single transducer schema Υ and transducer Π of $\boldsymbol{\mathcal{N}}$. The main idea is as follows. We provide Υ with input relations to encode a computation trace of the Turing machine M on input w. By simulating the Turing machine M, transducer Π checks that the input contains a valid and accepting computation trace. If so, Π sends a special message `accept( )` to itself, whose delivery is a trigger for diffluent behavior. On a more technical note, the sending rules might sometimes send `accept( )` when the trace is actually partially incorrect. To solve this, like in Section 4.6, we also check explicitly for errors in the input: when an error is detected, a message `error( )` is sent, and this acts as a signal to correct any diffluent behavior.

Diffluence Independently of w or M, we add the following relations to Υ: input relation $A^{(1)}$; memory relation $B^{(1)}$; output relation $T^{(1)}$; and, message relations

Relation	Purpose
$\texttt{state}^{(2)}$	configuration state
$\texttt{head}^{(1+n^k)}$	configuration head position
$\texttt{tape}^{(1+n^k+1)}$	configuration tape cell contents

Table 4.1: Relations for computation trace

$A_{\text{msg}}^{(1)}$, $B_{\text{msg}}^{(1)}$, $\texttt{accept}^{(0)}$ and $\texttt{error}^{(0)}$. The following rules implement the basic idea of making Π diffluent when $\texttt{accept}(\,)$ is received. Indeed, we can vary the delivery order of A_{msg}-facts and B_{msg}-facts. The purpose of relation `error` was explained above.

$$A_{\text{msg}}(\texttt{u}) \leftarrow A(\texttt{u}),\ \texttt{accept}(\,).$$
$$B_{\text{msg}}(\texttt{u}) \leftarrow A(\texttt{u}),\ \texttt{accept}(\,).$$
$$B(\texttt{u}) \leftarrow B_{\text{msg}}(\texttt{u}).$$
$$T(\texttt{u}) \leftarrow A_{\text{msg}}(\texttt{u}),\ \neg B(\texttt{u}).$$
$$T(\texttt{u}) \leftarrow A_{\text{msg}}(\texttt{u}),\ \texttt{error}(\,).$$

Computation trace We represent a computation trace of M on w with new input relations. Henceforth we write n to denote the length of w. We can select a $k \in \mathbb{N}$ such that for each string w' over Σ, if M accepts w' then M has an accepting computation trace on w' with at most 2^{n^k} transitions. Note, k is a constant in the construction of transducer Π.

A number a in the interval $[0, 2^{n^k}]$ indicates a (zero-based) configuration ordinal in the trace. Moreover, since time usage upper bounds space usage, a can also be used to indicate an individual tape cell. The number a has a binary representation with n^k bits, which is polynomial in n. Now, Table 4.1 gives the input relations, with their precise arities, to represent a computation trace. The first component in relations `state`, `head`, and `tape`, is an identifier of a Turing machine configuration. This identifier only serves to join the different aspects of one configuration across all three relations: relation `state` gives the current state symbol; relation `head` gives the head position; and, relation `tape` gives the contents of each tape cell.

Sending `accept` We now provide rules to send $\texttt{accept}(\,)$. Newly mentioned relations are assumed to be added to Υ_{msg}. The idea is as follows: in the relations of Table 4.1, we look for a path of length at most 2^{n^k} configurations that connects the start configuration to an accepting configuration, and such that each pair of subsequent configurations is allowed by a valid transition of M.

Suppose we could send a message of the form $\texttt{reach}_0(i, j)$ to say that configuration j can be reached from configuration i by a valid transition of M. The subscript 0 indicates that the distance between i and j is $2^0 = 1$. Since the desired path is of length at most 2^{n^k}, the following *recursion-free* rules can consider all such paths:[11]

[11]We use that any length between 0 and 2^{n^k} can be represented by a sum of unique powers of two.

$$\texttt{reach}_m(\texttt{i},\texttt{j}) \leftarrow \texttt{reach}_{m-1}(\texttt{i},\texttt{l}),\, \texttt{reach}_p(\texttt{l},\texttt{j})$$

for each $m = 1, \ldots, n^k$, and each $p = 0, \ldots, m-1$.

Suppose too we could send a message of the form $\texttt{start}(i)$ to say that configuration i satisfies the properties of the start configuration of M on w. We send $\texttt{accept}(\,)$ with these rules:

$$\texttt{accept}(\,) \leftarrow \texttt{start}(\texttt{i}),\, \texttt{reach}_m(\texttt{i},\texttt{j}),\, \texttt{state}(\texttt{j},\texttt{q}),\, q_{\text{accept}}(\texttt{q})$$

for each $m = 0, \ldots, n^k$.

Here, $q_{\text{accept}}^{(1)}$ is an extra input relation containing the symbol of the start state.

Note, the number and size of the above sending rules is polynomial in n. The remaining details, regarding the messages $\texttt{reach}_0$, $\texttt{start}$, $\texttt{error}$, and the proof of correctness, can be found in [19].

4.9 Expressivity

We investigate the expressivity of simple transducer networks. These networks may consist of multiple nodes.

First we define how a transducer network can compute a distributed query. We consider only *confluent* transducer networks because otherwise the output might vary depending on the run. Let $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a confluent transducer network, not necessarily simple. Let $in^{\boldsymbol{\mathcal{N}}}$ and $out^{\boldsymbol{\mathcal{N}}}$ be the distributed schemas for $\boldsymbol{\mathcal{N}}$ as defined in Section 2.5.1. We say that $\boldsymbol{\mathcal{N}}$ *computes* the following distributed query $\mathcal{Q}$, that is over input schema $in^{\boldsymbol{\mathcal{N}}}$ and output schema $out^{\boldsymbol{\mathcal{N}}}$: $\mathcal{Q}$ maps each instance H over $in^{\boldsymbol{\mathcal{N}}}$ to the instance $\mathcal{Q}(H) = J$ over $out^{\boldsymbol{\mathcal{N}}}$ such that $J(x)$ for each $x \in \mathcal{N}$ is the set of all output facts that can be produced at x during any run of $\boldsymbol{\mathcal{N}}$ on H. The instance $\mathcal{Q}(H)$ is well-defined even if $\boldsymbol{\mathcal{N}}$ is diffluent, but when $\boldsymbol{\mathcal{N}}$ is confluent, all runs on H can be extended to obtain $\mathcal{Q}(H)$. We call $\mathcal{Q}(H)$ the *output* of $\boldsymbol{\mathcal{N}}$ on input H.

We now define how UCQ$^\neg$ can express distributed queries in a more direct way, i.e., without transducer networks. This will provide insight in the expressivity of simple transducer networks. Let $\mathcal{E}$ be a distributed database schema over a network $\mathcal{N}$, and let H be an instance over $\mathcal{E}$. Let $\langle\mathcal{E}\rangle^{\mathcal{N}}$ and $\langle H\rangle^{\mathcal{N}}$ be as defined in Section 4.6.2. Intuitively, a UCQ$^\neg$-program over $\langle\mathcal{E}\rangle^{\mathcal{N}}$ can directly access all relations of all nodes. To make such a program generic, node identifiers are provided in the relations $x.\texttt{Id}$ with $x \in \mathcal{N}$ and $\texttt{Node}$. Let $\mathcal{Q}$ be a distributed query over input schema $\mathcal{E}$ and an output schema $\mathcal{F}$ (also over $\mathcal{N}$). We say that $\mathcal{Q}$ is *expressible* in UCQ$^\neg$ if for each pair $x \in \mathcal{N}$ and $R^{(k)} \in \mathcal{F}(x)$ we can give a UCQ$^\neg$-program $\Phi_{x,R}$ over input schema $\langle\mathcal{E}\rangle^{\mathcal{N}}$ and output schema $\{R^{(k)}\}$ such that for all instances H over $\mathcal{E}$ we have

$$\mathcal{Q}(H)(x)|_R = \Phi_{x,R}(\langle H\rangle^{\mathcal{N}}).$$

Now we can present the expressivity result:

Theorem 4.12. Confluent simple transducer networks capture the distributed queries expressible in UCQ$^\neg$.

This result requires showing a lower and upper bound on the expressivity of simple transducer networks, given in respectively Section 4.9.1 and Section 4.9.2. Currently, this result depends on our addition of nonequalities to UCQ$^{\neg}$ (Section 4.3). In particular, for showing the upper bound, we do a nontrivial simulation of runs of transducer networks with UCQ$^{\neg}$, and there we depend on the availability of nonequalities. It remains open whether the result really needs this feature.

4.9.1 Lower Bound

Let $\mathcal{Q}$ be a distributed query over input distributed schema $\mathcal{E}$ and output distributed schema $\mathcal{F}$, that is expressible in UCQ$^{\neg}$. Let $\mathcal{N}$ be the network of $\mathcal{E}$ and $\mathcal{F}$. Over $\mathcal{N}$, we define a simple transducer network $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ to compute $\mathcal{Q}$. For technical convenience, we assume for each $x \in \mathcal{N}$ that $\mathcal{E}(x)$ and $\mathcal{F}(x)$ have disjoint relation names and that $\mathcal{E}(x)$ and $\mathcal{F}(x)$ do not contain Id or All. Any conflicts can always be resolved with appropriate renamings.

4.9.1.1 Transducer Schemas

First, we give the shared message relations of $\boldsymbol{\mathcal{N}}$, where relation names containing "$\neg$" are meant to indicate the absence of a fact:

- the relations $x.R^{(k)}$ and $x.R_{\neg}^{(k)}$ for each $x \in \mathcal{N}$ and $R^{(k)} \in \mathcal{E}(x)$, to broadcast local inputs;
- the relations $x.\mathtt{Id}^{(1)}$ and $x.\mathtt{Id}_{\neg}^{(k)}$ for each $x \in \mathcal{N}$, to broadcast identifiers;
- the relations $x.T^{(k)}$ for each $x \in \mathcal{N}$ and $T^{(k)} \in \mathcal{F}(x)$, to compute local outputs; and,
- the relation $\mathtt{Adom}^{(1)}$, to share active domain values.

For each $x \in \mathcal{N}$, we define $\boldsymbol{\Upsilon}(x)_{\text{in}} = \mathcal{E}(x)$; $\boldsymbol{\Upsilon}(x)_{\text{out}} = \mathcal{F}(x)$; $\boldsymbol{\Upsilon}(x)_{\text{mem}} = \emptyset$; and, $\boldsymbol{\Upsilon}(x)_{\text{msg}}$ is the above set of message relations.

4.9.1.2 Transducer Rules

Let $x \in \mathcal{N}$. We incrementally specify the rules of $\boldsymbol{\Pi}(x)$. First, to broadcast all active domain values, for each $R^{(k)} \in \boldsymbol{\Upsilon}(x)_{\text{in}} \cup \{\mathtt{Id}^{(1)}\}$ and each $i \in \{1, \ldots, k\}$, we add the following rule:

$$\mathtt{Adom}(\mathtt{n}, \mathtt{u_i}) \leftarrow \mathtt{All}(\mathtt{n}),\ R(\mathtt{u_1}, \ldots, \mathtt{u_i}, \ldots, \mathtt{u_k}).$$

Also, for each $R^{(k)} \in \boldsymbol{\Upsilon}(x)_{\text{in}} \cup \{\mathtt{Id}^{(1)}\}$, we add the following rules to send the presence or absence of local R-facts at x:

$$x.R(\mathtt{n}, \mathtt{u_1}, \ldots, \mathtt{u_k}) \leftarrow \mathtt{All}(\mathtt{n}),\ R(\mathtt{u_1}, \ldots, \mathtt{u_k}).$$

$$x.R_{\neg}(\mathtt{n}, \mathtt{u_1}, \ldots, \mathtt{u_k}) \leftarrow \mathtt{All}(\mathtt{n}),\ \mathtt{Adom}(\mathtt{u_1}),\ \ldots,\ \mathtt{Adom}(\mathtt{u_k}),\ \neg R(\mathtt{u_1}, \ldots, \mathtt{u_k}).$$

Now we let $\boldsymbol{\Pi}(x)$ produce output. Let $T^{(k)} \in \boldsymbol{\Upsilon}(x)_{\text{out}}$. To satisfy the message-boundedness restriction for the output rules, we add sending rules for message relation

$x.T^{(k)}$ and copy any received $x.T$-messages to output relation T at x. Because $\mathcal{Q}$ is expressible in UCQ$^\neg$, there is a UCQ$^\neg$ program Φ over $\langle\mathcal{E}\rangle^{\mathcal{N}}$ that expresses the T-facts at x. For each $\varphi \in \Phi$, we transform φ into a sending rule φ' for relation $x.T^{(k)}$, as follows:

- the head $T(\mathtt{u_1}, \ldots, \mathtt{u_k})$ of φ becomes the head $x.T(\mathtt{n}, \mathtt{u_1}, \ldots, \mathtt{u_k})$ of φ', where $\mathtt{n}$ is a new variable;
- the positive body atoms of φ' are *(i)* $\mathtt{Id}(\mathtt{n})$, with $\mathtt{n}$ as defined previously; *(ii)* the atoms $\mathtt{All}(\mathtt{m})$ for which $\mathtt{Node}(\mathtt{m}) \in pos_\varphi$; *(iii)* the atoms $y.R(\mathtt{v_1}, \ldots, \mathtt{v_l}) \in pos_\varphi$, which are now messages; *(iv)* the (positive) message atoms $y.R_\neg(\mathtt{v_1}, \ldots, \mathtt{v_l})$ for which $y.R(\mathtt{v_1}, \ldots, \mathtt{v_l}) \in neg_\varphi$;
- the negative body atoms of φ' are the atoms $\mathtt{All}(\mathtt{m})$ for which $\mathtt{Node}(\mathtt{m}) \in neg_\varphi$; and,
- the nonequalities of φ' are those of φ.

The positive body atom $\mathtt{Id}(\mathtt{n})$ has the effect that $x.T$-messages are sent only to x. Now, the final output for $T^{(k)}$ is created by adding this rule:

$$T(\mathtt{u_1}, \ldots, \mathtt{u_k}) \leftarrow x.T(\mathtt{u_1}, \ldots, \mathtt{u_k}).$$

This completes the specification of $\mathbf{\Pi}(x)$. Note that transducer $\mathbf{\Pi}(x)$ is simple: all message rules are message-positive and static; all output rules are message-positive and message-bounded; $\mathbf{\Pi}(x)$ is inflationary (there are no memory relations); and, $\mathbf{\Pi}(x)$ is recursion-free.

Following the above instructions, we can build the transducer at each node of $\boldsymbol{\mathcal{N}}$. There are also no cycles through message relations in $\boldsymbol{\mathcal{N}}$. Hence, $\boldsymbol{\mathcal{N}}$ is simple.

4.9.1.3 Example

The following example illustrates the construction of the transducer network.

Example 4.13. Let $\mathcal{N} = \{x, y\}$. Consider the following distributed schemas $\mathcal{E}$ and $\mathcal{F}$, both over $\mathcal{N}$: $\mathcal{E}(x) = \{A^{(2)}\}$, $\mathcal{E}(y) = \{B^{(1)}\}$, $\mathcal{F}(x) = \{S^{(1)}\}$ and $\mathcal{F}(y) = \{T^{(1)}\}$. Consider the following distributed query $\mathcal{Q}$ with input schema $\mathcal{E}$ and output schema $\mathcal{F}$, expressed in UCQ$^\neg$:

$$S(\mathtt{u}) \leftarrow x.A(\mathtt{u}, \mathtt{v}),\ \neg y.B(\mathtt{u}),\ \mathtt{u} \neq \mathtt{v}.$$

$$T(\mathtt{u}) \leftarrow x.A(\mathtt{u}, \mathtt{v}),\ x.\mathtt{Id}(\mathtt{u}).$$

Each rule corresponds to one of the output relations.

We construct a transducer network $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \mathbf{\Upsilon}, \mathbf{\Pi})$ to compute $\mathcal{Q}$. To save space, we will not literally follow the general construction from above, but instead restrict attention to the relations and rules that affect the output.

First, the shared message relations of $\boldsymbol{\mathcal{N}}$ are: $x.A^{(2)}$, $x.\mathtt{Id}^{(1)}$, $y.B_\neg^{(1)}$ and $\mathtt{Adom}^{(1)}$. The sending rules for $\mathtt{Adom}$ are clear, so we do not explicitly give them.

For node x, we define $\mathbf{\Upsilon}(x)_{\mathrm{in}} = \{A^{(2)}\}$, $\mathbf{\Upsilon}(x)_{\mathrm{out}} = \{S^{(1)}\}$, and $\mathbf{\Upsilon}(x)_{\mathrm{mem}} = \emptyset$. Transducer $\mathbf{\Pi}(x)$ contains the rules:

$$x.A(\mathtt{n},\mathtt{u},\mathtt{v}) \leftarrow \mathtt{All}(\mathtt{n}),\, A(\mathtt{u},\mathtt{v}).$$
$$x.\mathtt{Id}(\mathtt{n},\mathtt{u}) \leftarrow \mathtt{All}(\mathtt{n}),\, \mathtt{Id}(\mathtt{u}).$$
$$x.S(\mathtt{n},\mathtt{u}) \leftarrow \mathtt{Id}(\mathtt{n}),\, x.A(\mathtt{u},\mathtt{v}),\, y.B_{\neg}(\mathtt{u}),\, \mathtt{u} \neq \mathtt{v}.$$
$$S(\mathtt{u}) \leftarrow x.S(\mathtt{u}).$$

For node y, we define $\boldsymbol{\Upsilon}(y)_{\text{in}} = \{B^{(1)}\}$, $\boldsymbol{\Upsilon}(y)_{\text{out}} = \{T^{(1)}\}$, and $\boldsymbol{\Upsilon}(y)_{\text{mem}} = \emptyset$. Transducer y contains the rules:

$$y.B_{\neg}(\mathtt{n},\mathtt{u}) \leftarrow \mathtt{All}(\mathtt{n}),\, \mathtt{Adom}(\mathtt{u}),\, \neg B(\mathtt{u}).$$
$$y.T(\mathtt{n},\mathtt{u}) \leftarrow x.A(\mathtt{u},\mathtt{v}),\, x.\mathtt{Id}(\mathtt{u}).$$
$$T(\mathtt{u}) \leftarrow y.T(\mathtt{u}).$$

This completes the network $\boldsymbol{\mathcal{N}}$. □

4.9.2 Upper Bound

Let $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a confluent simple transducer network. Let $\mathcal{Q}$ denote the distributed query computed by $\boldsymbol{\mathcal{N}}$. Let $x \in \mathcal{N}$ and let $R^{(k)}$ be a local output relation of x. We have to construct a UCQ$^{\neg}$-program Φ over input schema $\langle in^{\boldsymbol{\mathcal{N}}}\rangle^{\mathcal{N}}$ and output schema $\{R^{(k)}\}$, such that $\mathcal{Q}(H)(x)|_R = \Phi(\langle H\rangle^{\mathcal{N}})$ for each input distributed database instance H over $in^{\boldsymbol{\mathcal{N}}}$.

The basic idea is to describe the computation of $\boldsymbol{\mathcal{N}}$ with UCQ$^{\neg}$-program Φ, for output relation R at x. To make this technically easier, we first convert $\boldsymbol{\mathcal{N}}$ to a single-node network in Section 4.9.2.1. Some common notations are introduced in Section 4.9.2.2, and program Φ is described in Section 4.9.2.3. The correctness is shown in [19].

4.9.2.1 Reduction to Single-node

Consider the concepts from Section 4.6.2. Using Proposition 4.6, let $\boldsymbol{\mathcal{M}}$ be the simple single-node transducer network that simulates $\boldsymbol{\mathcal{N}}$, and that is confluent because $\boldsymbol{\mathcal{N}}$ is confluent. We regard the query $\mathcal{Q}'$ computed by $\boldsymbol{\mathcal{M}}$ as an ordinary database query over input schema $\langle in^{\boldsymbol{\mathcal{N}}}\rangle^{\mathcal{N}}$ and output schema $\langle out^{\boldsymbol{\mathcal{N}}}\rangle$. If for every input H for $\boldsymbol{\mathcal{N}}$ we would know that $\mathcal{Q}'(\langle H\rangle^{\mathcal{N}}) = \langle \mathcal{Q}(H)\rangle$, because relation $x.R$ is in $\langle out^{\boldsymbol{\mathcal{N}}}\rangle$, it will be sufficient to construct the UCQ$^{\neg}$-program Φ as a description of the computation of $\boldsymbol{\mathcal{M}}$ for relation $x.R$. To keep the notation simpler, we may assume without loss of generality that output relation R only occurs at x. So, we will write "R" instead of "$x.R$".

We are left to show $\mathcal{Q}'(\langle H\rangle^{\mathcal{N}}) = \langle \mathcal{Q}(H)\rangle$ for every input H over $in^{\boldsymbol{\mathcal{N}}}$. Let the output of a configuration ρ, denoted $out(\rho)$, be as defined in Section 4.6.2. Abbreviate $J = \mathcal{Q}'(\langle H\rangle^{\mathcal{N}})$. We show $J \subseteq \langle \mathcal{Q}(H)\rangle$. By confluence of $\boldsymbol{\mathcal{M}}$, there is a run $\mathcal{S}$ of $\boldsymbol{\mathcal{M}}$ on $\langle H\rangle^{\mathcal{N}}$ such that $out(last(\mathcal{S})) = J$. Next, because $\boldsymbol{\mathcal{M}}$ simulates $\boldsymbol{\mathcal{N}}$, there is a run $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on H such that $\langle out(last(\mathcal{R}))\rangle = out(last(\mathcal{S}))$. So, $J = \langle out(last(\mathcal{R}))\rangle \subseteq \langle \mathcal{Q}(H)\rangle$.

Now we show $\langle \mathcal{Q}(H)\rangle \subseteq J$. By confluence of $\boldsymbol{\mathcal{N}}$, there exists a run $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on H such that $\mathcal{Q}(H) = out(last(\mathcal{R}))$. Because $\boldsymbol{\mathcal{M}}$ simulates $\boldsymbol{\mathcal{N}}$, there exists a run $\mathcal{S}$ of $\boldsymbol{\mathcal{M}}$

on $\langle H \rangle^{\mathcal{N}}$ such that $out(last(\mathcal{S})) = \langle out(last(\mathcal{R})) \rangle$. Hence, $\langle \mathcal{Q}(H) \rangle = out(last(\mathcal{S})) \subseteq J$.

4.9.2.2 Common Concepts and Notations

A *ground literal* is a fact or a fact with "¬" prepended. For a database instance I and a ground literal $\boldsymbol{l}$, we write $I \models \boldsymbol{l}$ to mean $\boldsymbol{l} \in I$ if $\boldsymbol{l}$ is a fact and otherwise we mean $\boldsymbol{f} \notin I$, where $\boldsymbol{l} = \neg \boldsymbol{f}$. For a derivation tree $\mathcal{T}$, for each internal node x, we write $body^{\mathcal{T}}(x)$ to denote the set of ground literals obtained by applying $val^{\mathcal{T}}(x)$ to the body literals of $rule^{\mathcal{T}}(x)$.

Two derivation trees $\mathcal{T}$ and $\mathcal{S}$ are said to be *structurally equivalent* if *(i)* the trees $(nodes^{\mathcal{T}}, edges^{\mathcal{T}})$ and $(nodes^{\mathcal{S}}, edges^{\mathcal{S}})$ are isomorphic under a node bijection $b : nodes^{\mathcal{T}} \to nodes^{\mathcal{S}}$; and, *(ii)* for every edge $(x, y) \in edges^{\mathcal{T}}$, we have $rule^{\mathcal{T}}(x) = rule^{\mathcal{S}}(b(x))$ and $lit^{\mathcal{T}}(y) = lit^{\mathcal{S}}(b(y))$. We call b the *structural bijection.*

4.9.2.3 Building the UCQ$^{\neg}$-Program

In this section, we construct the required UCQ$^{\neg}$-program Φ. We gradually build up the different parts of this program, and introduce auxiliary definitions and notations along the way. Using the equivalence between UCQ$^{\neg}$ and existential FO with nonequalities, abbreviated $\exists$FO, some parts are specified in $\exists$FO for technical convenience.

Let $\boldsymbol{\mathcal{M}}$ be the simple single-node network from above, and let Υ and Π respectively denote the transducer schema and transducer of $\boldsymbol{\mathcal{M}}$.

General derivation trees Let $\mathcal{T}$ be a derivation tree of Π. We define the *active domain* of $\mathcal{T}$ to be the set of all values assigned by valuations in $\mathcal{T}$. We say that $\mathcal{T}$ is *general* if there is no structurally equivalent derivation tree $\mathcal{S}$ with a strictly larger active domain. Intuitively, a general derivation tree assigns a different value to each variable of a rule if possible.

All output strategies Let $forest_R$ be a maximal set of general derivation trees of transducer Π for output relation R, such that no two trees are structurally equivalent, and such that no two trees have an overlap of their active domains. Because Π is recursion-free, there are only a finite number of structurally different trees, and thus $forest_R$ is finite. Intuitively, $forest_R$ represents all possible strategies of Π to derive facts over R, using as much different values as possible. For each subset $G \subseteq forest_R$, we write $adom(G)$ to denote the union of all active domains of trees in G.

Canonical runs Intuitively, for any particular input for Π, we can make a selection $G \subseteq forest_R$ of all trees that "work" on that input, i.e., for all trees $\mathcal{T} \in G$ there is a substitution of the values in $\mathcal{T}$ by values in the input so that the new valuations are true. If we regard values in $adom(G)$ as variables (as we will do later), this substitution of values looks very much like a valuation. Next, for G, we can formally define a *canonical run* $\mathcal{R}^G$. The idea is that in $\mathcal{R}^G$ we execute all trees of G concurrently, with as few transitions as possible, i.e., by using their canonical schedulings. The run

$\mathcal{R}^G$ will do n transitions, where n is the largest height of a tree in G.[12] Hence, the length of $\mathcal{R}^G$ is bounded by the syntactical properties of Π.

Note, for an internal node x of a derivation tree $\mathcal{T}$, by message-positivity, the set $\mathit{body}^{\mathcal{T}}(x)|_{\Upsilon_{\mathrm{msg}}}$ of ground literals contains only facts. Now, for each transition $i \in \{1, \ldots, n\}$ of $\mathcal{R}^G$, we (want to) deliver the following message set

$$M_i^G = \bigcup_{\mathcal{T} \in G} \bigcup_{\substack{x \in \mathit{int}^{\mathcal{T}}, \\ \kappa^{\mathcal{T}}(x) = i}} \mathit{body}^{\mathcal{T}}(x)|_{\Upsilon_{\mathrm{msg}}}.$$

In words: for each transition i, set M_i^G is the union across all trees of G of the messages needed by rules scheduled at transition i. We now make an $\exists$FO-formula sndMsg_G to express that these message sets can be sent. For notational simplicity, the symbols of $\mathit{adom}(G)$ represent variables. For a derivation tree $\mathcal{T} \in G$, let $\mathit{msg}^{\mathcal{T}} \subseteq \mathit{int}^{\mathcal{T}}$ denote the set of internal nodes x where $\mathit{lit}^{\mathcal{T}}(x)$ is over a message relation. Because sending rules are message-positive and static, it suffices to demand that all involved input literals are satisfied (both positive and negative):

$$\mathit{sndMsg}_G := \bigwedge_{\mathcal{T} \in G} \bigwedge_{x \in \mathit{msg}^{\mathcal{T}}} \mathit{body}^{\mathcal{T}}(x)|_{\Upsilon_{\mathrm{in}}}.$$

This is a quantifier-free formula, where we write sets of literals in the conjunction, with the understanding that such a set is written using some arbitrary ordering on its elements.

Canonical runs: output succeeds Let G be as above. Fix some $\mathcal{T} \in G$. In the following, we specify an $\exists$FO-formula to express that $\mathcal{T}$ succeeds in deriving its root fact in $\mathcal{R}^G$. Here, a possible "danger", is that the concurrent execution of $\mathcal{T}$ with another tree $\mathcal{S}$ might make certain valuations in $\mathcal{T}$ become unsatisfying. This could for instance happen when $\mathcal{S}$ derives a memory fact that $\mathcal{T}$ later tests for absence. We formalize this below.

The *alpha* nodes of $\mathcal{T}$, denoted $\alpha^{\mathcal{T}}$, are all internal nodes x of $\mathcal{T}$ for which $\mathit{lit}^{\mathcal{T}}(x)$ is a (positive) output or memory literal.[13] Note, $\mathit{root}^{\mathcal{T}} \in \alpha^{\mathcal{T}}$. The valuations of these alpha nodes have to be satisfiable to make $\mathcal{T}$ succeed. For each $x \in \alpha^{\mathcal{T}}$, the *beta* nodes of x, denoted $\beta^{\mathcal{T}}(x)$, are the child-nodes y of x for which $\mathit{lit}^{\mathcal{T}}(y)$ is a negative output or memory literal. By definition of derivation tree, $\beta^{\mathcal{T}}(x)$ contains only leafs. For each $x \in \alpha^{\mathcal{T}}$, a node $y \in \beta^{\mathcal{T}}(x)$ is a potential danger: if the fact inside the ground literal $\mathit{val}^{\mathcal{T}}(x)(\mathit{lit}^{\mathcal{T}}(y))$, henceforth referred to as "beta fact", is accidentally derived before transition $\kappa^{\mathcal{T}}(x)$, then $\mathit{val}^{\mathcal{T}}(x)$ is not satisfying in transition $\kappa^{\mathcal{T}}(x)$ (by inflationarity of Π). The derivation of beta facts could happen when the message deliveries of $\mathcal{R}^G$ accidentally trigger some rules of Π.

To represent these unwanted derivations, we consider *truncated derivation trees* that are like normal derivation trees, except that message nodes are also leafs. We only consider truncated derivation trees for deriving output and memory facts. We

[12]The height of a derivation tree is the largest number of edges on any path from a leaf to the root.

[13]This literal is always positive because x is an internal node.

say that a truncated derivation tree $\mathcal{S}$ can be *aligned* to $\mathcal{R}^G$ if there is a scheduling $\lambda : \mathit{int}^{\mathcal{S}} \to \{1, \ldots, n\}$ such that for each $x \in \mathit{int}^{\mathcal{S}}$, message set $M^G_{\lambda(x)}$ contains $\mathit{body}^{\mathcal{S}}(x)|_{\Upsilon_{\text{msg}}}$, i.e., for each valuation in $\mathcal{S}$, the necessary messages occur in some well-chosen transitions. Possibly multiple alignments exist for $\mathcal{S}$. For an output or memory fact $\boldsymbol{f}$, we write $\mathit{align}^G(\boldsymbol{f})$ to denote the set of all pairs $(\mathcal{S}, \lambda)$ where $\mathcal{S}$ is a truncated derivation tree for $\boldsymbol{f}$ having alignment λ to $\mathcal{R}^G$, and such that no two pairs in $\mathit{align}^G(\boldsymbol{f})$ differ only in the values for representing tree-nodes. This set is finite, as we now argue. First, because Π is recursion-free, there are only a finite number of structurally different (truncated) derivation trees for $\boldsymbol{f}$. Second, only a finite number of valuations can be used in the rules of such trees: because these rules are output or memory rules, by message-boundedness, assigned values must either be in $\boldsymbol{f}$ or must occur in a message, and $\mathcal{R}^G$ contains only a finite number of messages.

Now we specify the formula to express that a derivation tree $\mathcal{T}$ derives its root fact in $\mathcal{R}^G$. To obtain a general construction for later use, we take $\mathcal{T}$ to be a *truncated* derivation tree for an output or memory relation, that has an alignment κ to $\mathcal{R}^G$. Note, $\alpha^{\mathcal{T}} = \mathit{int}^{\mathcal{T}}$. The formula is as follows:

$$\mathit{succeed}_{G,\mathcal{T},\kappa} := \mathit{succeed}^{\text{in}}_{G,\mathcal{T},\kappa} \wedge \mathit{succeed}^{\text{deny}}_{G,\mathcal{T},\kappa}$$

with

$$\mathit{succeed}^{\text{in}}_{G,\mathcal{T},\kappa} := \bigwedge_{x \in \alpha^{\mathcal{T}}} \mathit{body}^{\mathcal{T}}(x)|_{\Upsilon_{\text{in}}}; \text{ and,}$$

$$\mathit{succeed}^{\text{deny}}_{G,\mathcal{T},\kappa} := \bigwedge_{x \in \alpha^{\mathcal{T}}} \quad \bigwedge_{\substack{y \in \beta^{\mathcal{T}}(x), \\ \text{let } \boldsymbol{f} = \mathit{fact}^{\mathcal{T}}(y)}} \quad \bigwedge_{\substack{(\mathcal{S},\lambda) \in \mathit{align}^G(\boldsymbol{f}), \\ \lambda(\mathit{root}^{\mathcal{S}}) < \kappa(x)}} \neg \mathit{succeed}_{G,\mathcal{S},\lambda}.$$

Intuitively, for each $x \in \alpha^{\mathcal{T}}$, we express *(i)* that the input literals in $\mathit{body}^{\mathcal{T}}(x)$ are satisfied; and, *(ii)* we consider all possible truncated derivation trees for beta facts, and demand that their alignments fail to derive the root (beta) fact. The second requirement is expressed with a recursive construction through negation: intuitively, to protect the alpha facts, we must deny the beta facts, which in turn (recursively) requires letting the alpha facts of trees for these beta facts fail, and so on. This recursion ends because each time we pass a truncated derivation tree to the recursive step, the root of this tree is scheduled strictly closer to the beginning of $\mathcal{R}^G$. The final formula $\mathit{succeed}_{G,\mathcal{T},\kappa}$ is quantifier-free, with variables in $\mathit{adom}(G)$.

Combining everything Let $G \subseteq \mathit{forest}_R$ and $\mathcal{T} \in G$ be as above. We write $T^{\downarrow}$ to denote the truncated version of $\mathcal{T}$, by making the nodes that derive messages into leaf nodes. Note, the canonical scheduling $\kappa^{\mathcal{T}}$, when restricted to the internal nodes of $\mathcal{T}^{\downarrow}$, is an alignment of $\mathcal{T}^{\downarrow}$ to $\mathcal{R}^G$. We can combine our previous formulas to express that the messages of $\mathcal{R}^G$ can be sent and that $\mathcal{T}^{\downarrow}$ successfully derives its root fact when its internal nodes are scheduled by $\kappa^{\mathcal{T}}$:

$$\mathit{derive}_{G,\mathcal{T}} := \exists \bar{z} \left(\mathit{diffVal}_G \wedge \mathit{sndMsg}_G \wedge \mathit{succeed}_{G,\mathcal{T}^{\downarrow},\kappa^{\mathcal{T}}} \right),$$

where $\bar{z}$ is an arbitrary ordering of the values in $adom(G)$ that do not occur in the root fact of $\mathcal{T}$, and where

$$diffVal_G = \bigwedge_{\substack{a,b \in adom(G),\\ a \neq b}} (a \neq b).$$

The subformula $diffVal_G$ demands that a valuation for the quantifier-free part of the above formula is injective, which we need in the correctness proof to convert concrete derivation trees to abstract ones (i.e., to features of formula $derive_{G,\mathcal{T}}$). By the equivalence of $\exists$FO and UCQ$^\neg$, we may consider $derive_{G,\mathcal{T}}$ to be a UCQ$^\neg$-program, having as free variables the tuple $\bar{x}$ in the root fact of $\mathcal{T}$.[14] We can create such a UCQ$^\neg$-program for every $G \subseteq forest_R$ and $\mathcal{T} \in G$.

Before we can give the final UCQ$^\neg$-program Φ, we need to consider the following. Although $derive_{G,\mathcal{T}}$ considers alignments of beta facts, an input for Π possibly has not as many different values as $adom(G)$. For this reason, we might overlook some alignments that could occur on a real input. For example, an undesirable beta fact might be derivable by a rule $S(x,x) \leftarrow A_{\text{msg}}(x,x)$ where $A^{(2)}_{\text{msg}} \in \Upsilon_{\text{msg}}$. But because G contains general trees, in run $\mathcal{R}^G$ we might deliver only (abstract) A_{msg}-facts with two different components, preventing an alignment of this rule. To solve this problem, we consider equivalence relations E on $adom(G)$. Assuming a total order on **dom**, we can replace each value $a \in adom(G)$ by the smallest value in its equivalence class under E, giving a set of derivation trees $E(G)$ with a smaller active domain. Using $E(G)$ instead of G, and a tree $\mathcal{T} \in E(G)$, the variables in UCQ$^\neg$-program $derive_{E(G),\mathcal{T}}$ can represent more specific inputs. We write $Eq(G)$ to denote all equivalence relations of $adom(G)$ under which the nonequalities of rules in G are still satisfied.

Now, we define the final program Φ as

$$\Phi := \bigcup_{G \subseteq forest_R} \bigcup_{E \in Eq(G)} \bigcup_{\mathcal{T} \in E(G)} derive_{E(G),\mathcal{T}}.$$

The correctness of Φ is shown in [19].

4.10 Model Variations

We now relate the obtained results to the two transducer network models of Chapter 3. First we recall these models. In the first model, *(i)* we allow incomplete network graphs, i.e., not every two nodes are directly connected; *(ii)* the network is homogeneous, i.e. each node runs the same transducer; and, *(iii)* the (single) transducer is epidemic. The second model is basically the same, except it uses an addressing transducer. Let us refer to these models as A and B respectively.

To better establish the connection to this chapter, we will only apply these models to *fixed* networks, and we allow nodes to send messages to themselves.[15] Note that

[14] A variable may occur multiple times in $\bar{x}$.

[15] We assume that self-sending happens automatically for epidemic transducers whenever they send something, but that self-sending requires explicit self-addressing from addressing transducers.

the definition of simple transducer networks of Section 4.5.2 can be applied without change to transducer networks under models A and B.

As a small remark, under model A and B, the encoding of a transducer network from Section 4.4.3 has to be extended to also incorporate the edges of the network graph.

4.10.1 Diffluence Decidability

We discuss how the diffluence decidability result of Section 4.8 relates to simple transducer networks under models A and B. We will be reducing to and from the diffluence decision problem for simple single-node networks.

Model B Consider first model B. First we establish a NEXPTIME upper bound on diffluence decidability. Using a technique very similar to Section 4.6, a simple transducer network under model B can be transformed into a simple single-node network, with one difference: for a simulated sender node x, we only allow sending rules for an addressee y if y is a neighbor of x in the now incomplete network graph. The notion of homogeneousness does not require additional modifications. This transformation can be done in PTIME, so diffluence of simple transducer networks under model B can be decided in NEXPTIME (Section 4.8).

To establish the NEXPTIME lower bound, we note that a simple single-node transducer network as defined in Section 4.6 is a special case of a simple transducer network under model B; indeed, the notions of incomplete networks and homogeneousness do not manifest themselves on single-node networks.

Model A Now consider model A. To establish a NEXPTIME upper bound on diffluence decidability, we observe that an epidemic transducer can be replaced by an addressing transducer, by sending each message explicitly to every neighbor; this can be done with a PTIME syntactical translation.[16] Hence, each simple transducer network under model A can be translated to a simple transducer network under model B. So diffluence of simple transducer networks under model A can be decided in NEXPTIME.

For the NEXPTIME lower bound, we can again see that simple single-node transducer networks are a special case of simple transducer networks under model A.

4.10.2 Expressivity

We now discuss how the expressivity result of Section 4.9 relates to simple transducer networks under models A and B. We only consider confluent transducer networks.

Model B Consider first model B. To establish the upper bound on expressivity, the technique from Section 4.9.2 first does a translation of a multi-node network to single-node, which, as we have seen above, can also be done in PTIME for simple transducer networks under model B. After this step, the technique for the upper bound can be

[16]To do this, the addressee-variable is grounded in relation `All`. As before, messages addressed to non-neighbors are lost by the operational semantics.

applied unmodified, to describe the canonical runs of the single-node network with UCQ$^\neg$.

For the lower bound, some modifications to the technique in Section 4.9.1 are required. First, the specification of the distributed query in UCQ$^\neg$ can refer to specific nodes and their input relations, and it is allowed to specify different outputs for each node (although the same output relations occur on each node). By the homogeneousness of model B, we may only construct one transducer that is replicated on each node. Specifying different behavior for each node seems only achievable when constants are allowed in the transducer rules, to specify which rules should be executed by which node. Specifically, we add a guard $\mathtt{Id}(x)$, where x is a node identifier, to each rule that should only be executable on node x. Secondly, the technique of Section 4.9.1 relies on nodes sending active domain values to each other, and the presence and absence of input facts. But because model B allows incomplete networks, we need message forwarding. To do the message forwarding with only recursion-free rules, each message relation used in Section 4.9.1 can be given additional "versions", to indicate how many times it was forwarded. To illustrate, for the `Adom`-relation and an input relation $R^{(k)}$, one would have the following rules, where d is the diameter of the network that the distributed query is over:

$$\begin{aligned}
&\mathtt{Adom}_1(\mathtt{n},\mathtt{u_i}) \leftarrow \mathtt{All}(\mathtt{n}),\ R(\mathtt{u_1},\dots,\mathtt{u_i},\dots,\mathtt{u_k}) \quad \forall i \in \{1,\dots,k\}\\
&\mathtt{Adom}_2(\mathtt{n},\mathtt{u}) \leftarrow \mathtt{All}(\mathtt{n}),\ \mathtt{Adom}_1(\mathtt{u}).\\
&\vdots\\
&\mathtt{Adom}_d(\mathtt{n},\mathtt{u}) \leftarrow \mathtt{All}(\mathtt{n}),\ \mathtt{Adom}_{d-1}(\mathtt{u}).
\end{aligned}$$

Rules that need relation `Adom`, possibly in multiple body atoms, are translated into multiple rules where the different combinations of these version numbers are considered. This is similar for the other message relations.

Model A For model A, the expressivity upper bound can be transferred from model B, by first doing a PTIME translation of a simple transducer network under model A to an equivalent simple transducer network under model B (see above).

For the lower bound, if again constants are permitted in the transducer rules, we can follow the same idea as for model B above, where message relations get different versions. But because now we use an epidemic transducer, and hence a message arrives at *all* neighbors automatically, we add to each output message (see Section 4.9.1) an additional component to explicitly say which node is responsible for outputting it.

Chapter 5

Declarative Semantics for Dedalus

5.1 Outline

In this chapter we present a declarative semantics for Dedalus. First, Section 5.2 gives related work, and Section 5.3 gives technical remarks specific to this chapter. Section 5.4 gives some example Dedalus programs. Next, Section 5.5 gives the declarative semantics. Section 5.6 states the main result relating the operational and declarative semantics, the proof of which is given in two parts, namely, in Sections 5.7 and 5.8.

5.2 Related Work

An area of artificial intelligence that is closely related to declarative networking is that of programming multi-agent systems in declarative languages. The knowledge of an agent can be expressed by a logic program, which also allows for non-monotone reasoning, and agents update their knowledge by modifying the rules in these logic programs [43, 50, 42]. The language LUPS [9] was designed to specify such dynamic updates to logic programs, and LUPS is also a declarative language itself. After applying a sequence of updates specified in LUPS, the semantics of the resulting logic program can be defined in an inductive way. But an interesting connection to this current work, is that the semantics can also be given by first syntactically translating the original program and its updates into a single normal logic program, after which the stable model semantics is applied [9]. This has also lead to a practical implementation of LUPS. It should be noted however that in this second semantics, there is no modeling of causality or the sending of messages.

This chapter is based on joint work with Alvaro, Hellerstein, and Marczak [10]. Some proof details are not included in this chapter, but can be found in the technical report [11].

5.3 General Remarks

Recall the language Dedalus and its operational semantics from Section 2.8. If a run of a Dedalus program is clear from the context, we will typically refer to the transitions of this run by their ordinal, and these ordinals start at 0 for technical convenience.

5.4 Example Programs

In this section we give some example Dedalus programs that help illustrate the language. In this section, we assume that every node always has at least the local unary input relations `Id` and `Node`, that contain respectively the identifier of the local node and the identifiers of all nodes (including the local node). Additional input relations will use a different name, and for the sake of simplicity, we will assume that the relations `Id` and `Node` are automatically initialized correctly when we define the inputs for the Dedalus programs below.

We also briefly mention that it is possible to define the output of Dedalus programs based on so-called *ultimate facts* [46], which are the facts on every node that are eventually derived during every step (by the deductive subprogram). The examples below follow this principle, and this will also be used in Chapter 6.

Example 5.1. Suppose that each node has a binary input relation R that represents a graph. We want to compute at each node the transitive closure of the global graph that is obtained by uniting the local input graphs of all nodes. This output should be stored in a relation $T^{(2)}$ at each node.

For any network $\mathcal{N}$, for any distributed database instance over $\mathcal{N}$ and relation $R^{(2)}$, the following Dedalus program computes the required output at each node, in a well-known way [45]:

$$\begin{aligned}
&T(\mathtt{u},\mathtt{v}) \mid \mathtt{y} \leftarrow R(\mathtt{u},\mathtt{v}),\ \mathtt{Node}(\mathtt{y}).\\
&T(\mathtt{u},\mathtt{v}) \mid \mathtt{y} \leftarrow R(\mathtt{u},\mathtt{w}),\ T(\mathtt{w},\mathtt{v}),\ \mathtt{Node}(\mathtt{y}).\\
&T(\mathtt{u},\mathtt{v})\bullet \leftarrow T(\mathtt{u},\mathtt{v}).
\end{aligned}$$

The first asynchronous rule lets each node broadcast all of its local input R-facts as T-facts to every node, including itself. The second asynchronous rule lets each node take an incoming T-fact, join it with local R-facts, and broadcast the resulting transitive edges again to every node. The last rule is inductive, and it continuously persists all received T-facts to the next step, so that the relation T steadily grows at each node. After a while, every node will have accumulated all original graph edges and the transitive edges in relation T. □

The previous example showed how a recursive, monotone computation can be expressed. The next example illustrates how Dedalus can also be used to do a non-monotone computation in a distributed setting.

Example 5.2. In this example, nullary relations are used as booleans: *true* is represented by the nonempty relation and *false* by the empty relation. Suppose that each node has a nullary input relation R. We want to compute at each node a nullary fact $T(\,)$ if and only if the relation R is empty at *all* nodes (cf. Example 3.15). This is a nonmonotone computation. Indeed, if all nodes have an empty relation R then we produce $T(\,)$ on all nodes, and if at least one node has a nonempty relation R then we should not output $T(\,)$ on any node.

For every network $\mathcal{N}$, the following Dedalus program computes the desired output at each node:

$$\begin{aligned}
&\texttt{empty(x)} \mid \texttt{y} \leftarrow \neg R(\,),\ \texttt{Id(x)},\ \texttt{Node(y)}.\\
&\texttt{empty(x)}\bullet \leftarrow \texttt{empty(x)}.\\
&\texttt{missing(\,)} \leftarrow \texttt{Node(x)},\ \neg\texttt{empty(x)}.\\
&T(\,) \leftarrow \neg\texttt{missing(\,)}.
\end{aligned}$$

The first asynchronous rule lets a node broadcast its own identifier to every node (including itself) if its local input relation R is empty. The second rule is inductive and it persists the received node identifiers. The third rule is deductive, and it checks whether the identifiers of all nodes have been received (`missing` is false) or not (`missing` is true). Because the rule is deductive, the relation `missing` is recomputed during every local step of a node. The last rule, which is also deductive, produces a $T(\,)$-fact at every node that has received all node identifiers.

If indeed every node has an empty R-relation, after a while, all nodes have received all node identifiers, and from that moment onwards, all nodes produce $T(\,)$ in every step. In the other case, when at least one of the nodes does not have an empty R-relation, no node will receive the identifier of that node and thus no node will ever produce $T(\,)$. $\square$

5.5 Declarative Semantics

Let $\mathcal{P}$ be a Dedalus program. Throughout this section, we fix $\mathcal{P}$ and give a declarative semantics for this program. In this semantics, we want to abstract away details that are specific to the operational semantics. First, Sections 5.5.1 and 5.5.2 will provide additional notations and definitions about runs. These will be used in Section 5.5.3 to investigate an abstraction of the operational semantics and some of the properties involved. Next, the declarative semantics is given by the stable model semantics applied to a pure Datalog$^\neg$ program *pure*$(\mathcal{P})$ that is obtained from Dedalus program $\mathcal{P}$. In Sections 5.5.4 up to 5.5.8, we describe how to construct *pure*$(\mathcal{P})$ and we define its semantics. Intuitively, this new program will simulate the entire distributed computation (of all nodes together) and its construction is centered around the insights obtained in Section 5.5.3.

5.5.1 Timestamps

We will assign local time values to the steps of a node in the operational semantics. Let $\mathcal{R}$ be a run of $\mathcal{P}$ on some input H, over a network $\mathcal{N}$. For each transition $i \in \mathbb{N}$ of $\mathcal{R}$, having active node x_i, we define $loc_{\mathcal{R}}(i)$ to be the number of transitions in $\mathcal{R}$ also with active node x_i that come *strictly* before transition i. Note, $loc_{\mathcal{R}}(i)$ is the (zero-based) ordinal of the local step of x_i during transition i. We call such step ordinals the *timestamps* of that node, and these can be regarded as local clock values. We would like to stress that timestamps are relative to each node. For instance, timestamp 0 for a node x indicates the first step of x, and timestamp 0 for another node y indicates the first step of y.

As a counterpart to function $loc_{\mathcal{R}}(\cdot)$, for each $(x, s) \in \mathcal{N} \times \mathbb{N}$ we define $glob_{\mathcal{R}}(x, s)$ to be the transition ordinal i of $\mathcal{R}$ such that $x_i = x$ and $loc_{\mathcal{R}}(i) = s$. In words: we find the transition i in which node x does its local computation step with timestamp s. It follows from the definition of $loc_{\mathcal{R}}(\cdot)$ that $glob_{\mathcal{R}}(x, s)$ is uniquely defined.

5.5.2 Extended Schema and Trace

We want to associate a *location specifier* and a timestamp to facts over $sch(\mathcal{P})$. Let $R^{(k)} \in sch(\mathcal{P})$. The intuition of a fact $R(x, s, a_1, \ldots, a_k)$ with location specifier x and timestamp s will be that the fact $R(a_1, \ldots, a_k)$ is present at node x during its local step with timestamp s, after the program $deduc_{\mathcal{P}}$ is executed (defined in Section 2.8.2.3). For using timestamps in facts, we require that $\mathbb{N} \subseteq \mathbf{dom}$.

Formally, we write $sch(\mathcal{P})^{\mathrm{LT}}$ to denote the database schema obtained from $sch(\mathcal{P})$ by incrementing the arity of every relation by two. The two extra components will contain the location specifier and timestamp, which are by convention the first and second components of a fact.[1]

For a database instance I over $sch(\mathcal{P})$, $x \in \mathbf{dom}$ and $s \in \mathbb{N}$, we write $I^{\Uparrow x,s}$ to denote the facts over $sch(\mathcal{P})^{\mathrm{LT}}$ that are obtained by prepending location specifier x and timestamp s to every fact of I. For the reverse operation, for an instance J over $sch(\mathcal{P})^{\mathrm{LT}}$, we write $J^{\Downarrow}$ to denote the facts over $sch(\mathcal{P})$ obtained by removing the location specifier and timestamp from every fact of J. Lastly, we write $J|^{x,s}$ to denote the facts of J that have location specifier x and timestamp s, without removing the location specifier and timestamp.

When I and J are sets of *atoms* over schemas $sch(\mathcal{P})$ and $sch(\mathcal{P})^{\mathrm{LT}}$ respectively, and $\mathtt{x}, \mathtt{s} \in \mathbf{var}$, we will also apply the notations $I^{\Uparrow \mathtt{x},\mathtt{s}}$ and $J^{\Downarrow}$, with the same meaning as for facts (except that now the location specifiers and timestamps are variables). Also, if L is a sequence of literals over schema $sch(\mathcal{P})$, and $\mathtt{x}, \mathtt{s} \in \mathbf{var}$, we write $L^{\Uparrow \mathtt{x},\mathtt{s}}$ to denote the sequence of literals over schema $sch(\mathcal{P})^{\mathrm{LT}}$ that is obtained by adding location specifier $\mathtt{x}$ and timestamp $\mathtt{s}$ to the literals in L (negative literals stay negative).

We will now capture the computed data during a run as a set of facts that we call the *trace*. Let $\mathcal{R}$ be a run of $\mathcal{P}$ on some input H, over a network $\mathcal{N}$. For each transition $i \in \mathbb{N}$, let x_i denote the active node, and let D_i denote the output of subprogram $deduc_{\mathcal{P}}$ during i. Let $loc_{\mathcal{R}}(i)$ be as defined in Section 5.5.1. The operational semantics implies that D_i consists of *(i)* the input *edb*-facts at x_i; *(ii)* the inductively derived facts during the previous step of x_i (if $loc_{\mathcal{R}}(i) \geq 1$); *(iii)* the messages delivered during transition i; and, *(iv)* all facts deductively derived from the previous ones. Intuitively, D_i contains *all* local facts over $sch(\mathcal{P})$ that x_i has during transition i. Now, the *trace* of $\mathcal{R}$ is the following instance over $sch(\mathcal{P})^{\mathrm{LT}}$:

$$trace(\mathcal{R}) = \bigcup_{i \in \mathbb{N}} D_i^{\Uparrow x_i,\, loc_{\mathcal{R}}(i)}.$$

In words: the trace represents all locally computed facts during each transition, additionally carrying the location specifier and timestamp of the active node. The trace

[1]The abbreviation "LT" stands for "location specifier and timestamp".

shows in detail what happens in the run, in terms of what facts are available on the nodes during which of their steps.

5.5.3 Messages and Causality

In the declarative semantics, we want to represent the same computations as in the operational semantics. We believe the trace of a run represents in detail the computation of that run (see Section 5.5.2). So, our goal will be to represent in the declarative semantics exactly the traces of runs. In the operational semantics, we order the actions of the nodes on a fine-grained global time axis, by ordering the transitions in the runs. For representing the trace, we will see below that it is actually sufficient to focus on the direct and indirect relationships between just the local steps of the nodes, ignoring the global ordering of the transitions. This forms the main ingredient for the declarative semantics.

Let $\mathcal{R}$ be a run of $\mathcal{P}$, and let $\alpha_{\mathcal{R}}$ be the arrival function of $\mathcal{R}$, as defined in Section 2.8.2.4. From $\mathcal{R}$, we extract the message sending and receiving events, or simply called the "message events". Formally, we define $\mathit{mesg}(\mathcal{R})$ to be the set of all tuples $(x, s, y, t, \boldsymbol{f})$, with $\boldsymbol{f} = R(\bar{a})$ a fact, and abbreviating $i = \mathit{glob}_{\mathcal{R}}(x, s)$ and $j = \mathit{glob}_{\mathcal{R}}(y, t)$, such that $\alpha_{\mathcal{R}}(i, y, \boldsymbol{f}) = j$, i.e., node x during step s sends message $\boldsymbol{f}$ to y that arrives at the step t of y, with possibly $x = y$. Note that in $\mathit{mesg}(\mathcal{R})$ there is no mention of transitions since it only contains relationships between local steps. The following lemma says $\mathit{mesg}(\mathcal{R})$ is sufficient for representing the trace:

Lemma 5.3. Let H be an input for $\mathcal{P}$. For any two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\mathcal{P}$ on H, if $\mathit{mesg}(\mathcal{R}_1) = \mathit{mesg}(\mathcal{R}_2)$ then $\mathit{trace}(\mathcal{R}_1) = \mathit{trace}(\mathcal{R}_2)$.

Proof. This result is perhaps not really surprising, and we will only give a proof sketch. Let x be a node of the network that H is over. We will see by inductive reasoning on the local steps of x that x will produce during each step exactly the same deductive facts in $\mathcal{R}_1$ as in $\mathcal{R}_2$. In the first step of x, the previous state of x in both $\mathcal{R}_1$ and $\mathcal{R}_2$ consists of just the local input $H(x)$, since there was no previous step of x to derive inductive facts. Also, the given assumption $\mathit{mesg}(\mathcal{R}_1) = \mathit{mesg}(\mathcal{R}_2)$ implies that x receives exactly the same messages during its first step in $\mathcal{R}_1$ and $\mathcal{R}_2$. Once the previous state and the received messages are known, the execution of the subprograms $\mathit{deduc}_{\mathcal{P}}$ and $\mathit{induc}_{\mathcal{P}}$ during the first step of x is completely determined. Hence, in runs $\mathcal{R}_1$ and $\mathcal{R}_2$ the node x produces exactly the same deductive and inductive facts during the first step. This implies that also the second stored state of x is the same in both runs. Our reasoning can now be repeated for the second step of x, the third step, etc. Generalized to all nodes, we see $\mathit{trace}(\mathcal{R}_1) = \mathit{trace}(\mathcal{R}_2)$. □

Consider now the following example, that illustrates how the pairs in $\mathcal{N} \times \mathbb{N}$ might be related by a run.

Example 5.4. Suppose we have a run $\mathcal{R}$, in which the following events take place, where we assume that x, y and z are three distinct nodes:

1. Node x during step s sends a message A to node z.
2. This message A arrives at node z during step u.

3. The node z during step $u + 5$ sends a message B to node y.

4. This message B arrives at node y during step t.

There is a chain of events that connects (x, s) to (y, t):

$$(x, s) \xrightarrow{\text{send } A} (z, u) \xrightarrow{\text{step}} (z, u + 1) \ldots \xrightarrow{\text{step}} (z, u + 5) \xrightarrow{\text{send } B} (y, t).$$

We say that the step s of node x *(causally) happens before* the step t of node y because we can follow a forward chain of local steps and sent messages to connect step s of x to step t of y. And such a path can make a “detour” to other nodes and some of their steps as well; in this case node z and its steps u up to $u + 5$. □

For a run $\mathcal{R}$, the intuition of Example 5.4 can be formalized with the *happens-before* relation [23] on the set $\mathcal{N} \times \mathbb{N}$, which is defined as the smallest relation $\prec_{\mathcal{R}}$ on $\mathcal{N} \times \mathbb{N}$ that satisfies the following three conditions:

- for each $(x, s) \in \mathcal{N} \times \mathbb{N}$, we have $(x, s) \prec_{\mathcal{R}} (x, s + 1)$;
- $(x, s) \prec_{\mathcal{R}} (y, t)$ whenever for some fact $\boldsymbol{f}$ we have $(x, s, y, t, \boldsymbol{f}) \in \mathit{mesg}(\mathcal{R})$;
- $\prec_{\mathcal{R}}$ is transitive, i.e., $(x, s) \prec_{\mathcal{R}} (z, u) \prec_{\mathcal{R}} (y, t)$ implies $(x, s) \prec_{\mathcal{R}} (y, t)$.

We call these three cases respectively *local* edges, *message* edges and *transitive* edges. Naturally, the first two cases express a direct relationship, whereas the third case is more indirect. Note that the relation $\prec_{\mathcal{R}}$ does not say how the messages are used. For instance, in Example 5.4, we cannot say for sure if z at step u reads the data in message A, or that it is “necessary” for z to first receive A before it can send B. Also, even if two runs on the same input have the same happens-before relation, it is not guaranteed that they have the same trace. This is because the happens-before relation does not talk about the specific messages that arrive at the nodes (whereas Lemma 5.3 does).

Consider the following property:

Lemma 5.5. For every run $\mathcal{R}$, the happens-before relation $\prec_{\mathcal{R}}$ contains no cycles.

Proof. If there would be a cycle in $\prec_{\mathcal{R}}$ that contains transitive edges, then we can substitute each transitive edge in this cycle with a path consisting of non-transitive edges. Therefore it is sufficient to show the absence of cycles consisting of only non-transitive edges. We show this with a proof by contradiction. So, suppose that there is a chain in $\mathcal{N} \times \mathbb{N}$ without transitive edges

$$(x_1, s_1) \prec_{\mathcal{R}} (x_2, s_2) \prec_{\mathcal{R}} \ldots \prec_{\mathcal{R}} (x_n, s_n)$$

with $n \geq 2$ and $(x_1, s_1) = (x_n, s_n)$. Because there are no transitive edges, for each $i \in \{1, \ldots, n - 1\}$, the edge $(x_i, s_i) \prec_{\mathcal{R}} (x_{i+1}, s_{i+1})$ falls into one of the following two cases:

- $x_i = x_{i+1}$ and $s_{i+1} = s_i + 1$ (local edge);
- x_i during step s_i sends a message to x_{i+1} that arrives in step s_{i+1} of x_{i+1} (message edge).

In the first case, it follows from the definition of $loc_{\mathcal{R}}(\cdot)$ that

$$glob_{\mathcal{R}}(x_i, s_i) < glob_{\mathcal{R}}(x_{i+1}, s_{i+1}).$$

For the second case, by our operational semantics, every message is always delivered in a later transition than the one in which it was sent. So, again we have

$$glob_{\mathcal{R}}(x_i, s_i) < glob_{\mathcal{R}}(x_{i+1}, s_{i+1}).$$

Since this property holds for all above edges, by transitivity we have $glob_{\mathcal{R}}(x_1, s_1) < glob_{\mathcal{R}}(x_n, s_n)$. But this is a contradiction because $(x_1, s_1) = (x_n, s_n)$ and thus $glob_{\mathcal{R}}(x_1, s_1) = glob_{\mathcal{R}}(x_n, s_n)$. □

Corollary 5.6. For every run $\mathcal{R}$, the relation $\prec_{\mathcal{R}}$ is a strict partial order on $\mathcal{N} \times \mathbb{N}$.

Proof. From its definition, we immediately have that $\prec_{\mathcal{R}}$ is transitive. Secondly, irreflexivity for $\prec_{\mathcal{R}}$ follows from Lemma 5.5. □

In Example 5.4, the happens-before relation is indeed partial because (x, s) does not happen before $(y, t-1)$ and $(y, t-1)$ does not happen before (x, s). So, (x, s) and $(y, t-1)$ are incomparable with $\prec_{\mathcal{R}}$. On a similar note, it is in general possible that the directed graph with vertices $\mathcal{N} \times \mathbb{N}$ and edges $\prec_{\mathcal{R}}$ is not even weakly connected. This occurs for instance when there are no message edges, in which case the graph consists of only isolated chains of local edges (and their transitive closure).

Consider the following property:

Corollary 5.7. For every run $\mathcal{R}$, for each $(x, s, y, t, \boldsymbol{f}) \in mesg(\mathcal{R})$ we have $(y, t) \not\prec_{\mathcal{R}} (x, s)$.

Proof. First, $(x, s, y, t, \boldsymbol{f}) \in mesg(\mathcal{R})$ implies $(x, s) \prec_{\mathcal{R}} (y, t)$ by definition of $\prec_{\mathcal{R}}$. So, if $(y, t) \prec_{\mathcal{R}} (x, s)$ then $\prec_{\mathcal{R}}$ contains a cycle, which not possible by Lemma 5.5. □

This last property is equivalent to saying that if $(y, t) \not\prec_{\mathcal{R}} (x, s)$ then it is possible that x during step s sends a message to y that arrives in step t of y. We will concretely use this in our declarative semantics: we only allow x during step s to send a message to y at step t if $(y, t) \not\prec_{\mathcal{R}} (x, s)$ (cf. Section 5.5.7.3).

5.5.4 Additional Relation Names

In the following subsections, we will start with the construction of the pure Datalog$^{\neg}$ program obtained from $\mathcal{P}$, denoted $pure(\mathcal{P})$. For this purpose, we need some new relation names not yet used in $sch(\mathcal{P})$.[2] These are listed in Table 5.1. The concrete purpose of these relations will become clear in the following subsections.

[2]In practice, this can always be arranged through a namespace mechanism.

New relation names	Represents
`all`	network
`time`, `tsucc`, $<$, $\neq$	timestamps
`before`	happens-before relation
$\texttt{cand}_R$, $\texttt{chosen}_R$, $\texttt{other}_R$, for each relation name R in *idb*($\mathcal{P}$)	messages
`hasSender`, `isSmaller`, `hasMax`, `rcvInf`	delivery of only a finite number of messages to each step of a node

Table 5.1: Relation names not in *sch*($\mathcal{P}$).

5.5.5 Network and Time Relations

In *pure*($\mathcal{P}$), we will use unary relation `all` to represent the whole network of interest. For example, a fact $\texttt{all}(x)$ will express that x is a node of the network. A small remark: if the original rules of $\mathcal{P}$ need access to node identifiers, then those identifiers must be explicitly provided in extra input relations different from `all` (like relation `Node` in Section 5.4) or they must be received from other nodes by means of messages.

In *pure*($\mathcal{P}$), we will also explicitly reason about timestamps, using the relations of the following database schema:

$$\mathcal{D}_{\text{time}} = \{\texttt{time}^{(1)}, \texttt{tsucc}^{(2)}, <^{(2)}, \neq^{(2)}\}.$$

The relations '$<$' and '$\neq$' will be written in infix notation in rules. We consider only the following instance over $\mathcal{D}_{\text{time}}$:

$$\begin{aligned} I_{\text{time}} \quad = \quad & \{\texttt{time}(s), \texttt{tsucc}(s, s+1) \mid s \in \mathbb{N}\} \\ & \cup \{(s < t) \mid s, t \in \mathbb{N} : \ s < t\} \\ & \cup \{(s \neq t) \mid s, t \in \mathbb{N} : \ s \neq t\}. \end{aligned}$$

Intuitively, the instance I_{time} provides timestamps together with relations to compare them.

5.5.6 Representing Causality

We now explain how causality will be represented in *pure*($\mathcal{P}$). To express that $(x, s) \in \mathcal{N} \times \mathbb{N}$ causally happens before $(y, t) \in \mathcal{N} \times \mathbb{N}$, we use a fact of the form $\texttt{before}(x, s, y, t)$. We add to *pure*($\mathcal{P}$) the following rules:

$$\texttt{before}(\texttt{x}, \texttt{s}, \texttt{x}, \texttt{t}) \leftarrow \texttt{all}(\texttt{x}), \texttt{tsucc}(\texttt{s}, \texttt{t}). \tag{5.1}$$

$$\texttt{before}(\texttt{x}, \texttt{s}, \texttt{y}, \texttt{t}) \leftarrow \texttt{before}(\texttt{x}, \texttt{s}, \texttt{z}, \texttt{u}), \texttt{before}(\texttt{z}, \texttt{u}, \texttt{y}, \texttt{t}). \tag{5.2}$$

The rule (5.1) expresses that on every node, a step causally comes before the next step. The rule (5.2) computes the transitive closure on the `before` relation.

The above rules are added to *pure*($\mathcal{P}$) independently of what rules $\mathcal{P}$ contains. But in the following subsection we will add rules to *pure*($\mathcal{P}$) that are obtained by transforming the original rules of $\mathcal{P}$. In particular, the sending of messages will also have an impact on the happens-before relation.

5.5.7 Rule Transformation

For each type of rule in $\mathcal{P}$ we specify what corresponding rules should be added to *pure*($\mathcal{P}$). Because $\mathcal{P}$ is constant-free, all rules we add are constant-free. For technical convenience, we will assume that rules of $\mathcal{P}$ always contain at least one positive body atom. This assumption allows us to enforce more elegantly that the variables in the head atoms of *pure*($\mathcal{P}$) also occur in at least one positive body atom. This assumption is not really a restriction, since a nullary positive body atom is already sufficient.

First, let $\mathtt{x}, \mathtt{s}, \mathtt{t}, \mathtt{t}' \in \mathbf{var}$ be distinct variables that do not yet occur in the rules of $\mathcal{P}$. In *pure*($\mathcal{P}$), the variables $\mathtt{x}$ and $\mathtt{s}$ will be used as location specifier and timestamp respectively. The variables $\mathtt{t}$ and $\mathtt{t}'$ will also be used for timestamps. We write $\mathbf{B}\{\bar{\mathtt{v}}\}$, where $\bar{\mathtt{v}}$ is a tuple of variables, to denote any sequence β of literals over database schema *sch*($\mathcal{P}$), such that the variables in β are precisely those in the tuple $\bar{\mathtt{v}}$.

5.5.7.1 Deductive Rules

For each deductive rule

$$R(\bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}$$

in $\mathcal{P}$, we add to *pure*($\mathcal{P}$) the following rule:

$$R(\mathtt{x}, \mathtt{s}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}^{\Uparrow \mathtt{x}, \mathtt{s}}. \tag{5.3}$$

This rule expresses that the facts deductively derived at some node x during step s are (immediately) visible within step s of x.

5.5.7.2 Inductive Rules

For each inductive rule

$$R(\bar{\mathtt{u}})\bullet \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}$$

in $\mathcal{P}$, we add to *pure*($\mathcal{P}$) the following rule:

$$R(\mathtt{x}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}^{\Uparrow \mathtt{x}, \mathtt{s}}, \ \mathtt{tsucc}(\mathtt{s}, \mathtt{t}). \tag{5.4}$$

Intuitively, this rule derives a fact that becomes visible in the *next* step of the *same* node.

5.5.7.3 Asynchronous Rules

For the situation in which a node x at its step s sends a message $R(\bar{a})$ to a node y, we use a fact $\mathtt{cand}_R(x, s, y, t, \bar{a})$ to say that t could be the arrival timestamp of this message at y.[3] We use a fact $\mathtt{chosen}_R(x, s, y, t, \bar{a})$ to say that t is the *effective* arrival timestamp of this message at y. Lastly, a fact $\mathtt{other}_R(x, s, y, t, \bar{a})$ means that t is *not* the arrival timestamp of the message.

Now, for each asynchronous rule

$$R(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}$$

[3]Here, '$\mathtt{cand}$' abbreviates "candidate".

in $\mathcal{P}$, letting $\bar{\mathtt{w}}$ be a tuple of new and distinct variables with $|\bar{\mathtt{w}}| = |\bar{\mathtt{u}}|$, we add to *pure*($\mathcal{P}$) the following rules, for which the intuition is given below:

$$\begin{aligned}\mathtt{cand}_R(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t},\bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}},\bar{\mathtt{v}},\mathtt{y}\}^{\Uparrow \mathtt{x},\mathtt{s}},\ \mathtt{all}(\mathtt{y}),\ \mathtt{time}(\mathtt{t}),\\ \neg\mathtt{before}(\mathtt{y},\mathtt{t},\mathtt{x},\mathtt{s}).\end{aligned} \tag{5.5}$$

$$\mathtt{chosen}_R(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t},\bar{\mathtt{w}}) \leftarrow \mathtt{cand}_R(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t},\bar{\mathtt{w}}),\ \neg\mathtt{other}_R(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t},\bar{\mathtt{w}}). \tag{5.6}$$

$$\mathtt{other}_R(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t},\bar{\mathtt{w}}) \leftarrow \mathtt{cand}_R(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t},\bar{\mathtt{w}}),\ \mathtt{chosen}_R(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t}',\bar{\mathtt{w}}),\ \mathtt{t} \neq \mathtt{t}'. \tag{5.7}$$

$$R(\mathtt{y},\mathtt{t},\bar{\mathtt{w}}) \leftarrow \mathtt{chosen}_R(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t},\bar{\mathtt{w}}). \tag{5.8}$$

$$\mathtt{before}(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t}) \leftarrow \mathtt{chosen}_R(\mathtt{x},\mathtt{s},\mathtt{y},\mathtt{t},\bar{\mathtt{w}}). \tag{5.9}$$

Rule (5.5) represents the messages that are sent. It evaluates the body of the original asynchronous rule, verifies that the addressee is within the network by using relation `all`, and it generates all possible candidate arrival timestamps that are not restricted by relation `before`. This last restriction comes from Corollary 5.7, and it will prevent cycles from occurring in relation `before`.

Now remains the matter of actually choosing *one* arrival timestamp amongst all these candidates. Intuitively, rule (5.6) selects an arrival timestamp for a message with the condition that this timestamp is not yet ignored, as expressed with relation $\mathtt{other}_R$. Also, looking at rule (5.7), a possible arrival timestamp t becomes ignored if there is already a chosen arrival timestamp t' with $t \neq t'$. Together, both rules have the effect that exactly one arrival timestamp will be chosen under the stable model semantics. This technical construction is due to Saccà and Zaniolo [52], who show how to express dynamic choice under the stable model semantics.

Rule (5.8) represents the actual arrival of an R-message with the chosen arrival timestamp: the data-tuple in the message becomes part of the addressee's state for relation R. When the addressee reads relation R, it thus transparently reads the arrived R-messages.

The rule (5.9) adds the causal restriction that the local step of the sender happens before the arrival step of the addressee. Together with the previously introduced rules (5.1) and (5.2), this will make sure that when the addressee later *causally* replies to the sender, the reply — as generated by a rule of the form (5.5) — will arrive after this first send-step of the sender.

Note, if multiple asynchronous rules in $\mathcal{P}$ have the same head predicate R, only new $\mathtt{cand}_R$-rules have to be added because the rules (5.6)–(5.9) are general for all R-messages.

Note, if there are asynchronous rules in $\mathcal{P}$, the program *pure*($\mathcal{P}$) will not be syntactically stratifiable because relation `before` negatively depends on itself through rules of the following forms, in order: (5.5), (5.6) and (5.9). Moreover, *pure*($\mathcal{P}$) is not locally stratifiable [21] because on a network with a least two nodes x and y, the fact $\mathtt{before}(x, s, y, t)$ with $s, t \in \mathbb{N}$ can negatively depend on itself by means of ground versions of these same rules.

5.5.7.4 Fairness and Finite Messages

We now relate the fairness conditions of Section 2.8.2.4 to *pure*($\mathcal{P}$). The fairness condition that every node does an infinite number of transitions does not require explicit modeling, because our previous transformations of deductive, inductive and asynchronous rules implicitly look at all possible pairs in $\mathcal{N} \times \mathbb{N}$, i.e., all possible steps for all nodes. The other fairness condition demands that every sent message is eventually delivered. This too is already satisfied by our transformation of the asynchronous rules, because for every sent message we choose precisely one arrival timestamp.

But there is one more thing that requires special attention. Our program *pure*$(\mathcal{P})$ so far allows an infinite number of messages to arrive at any step of a node. This does not happen in our operational semantics, or in any real-world distributed system for that matter: indeed, no node has to process an infinite number of messages at any given moment. We consider this to be an additional fairness restriction that must be explicitly enforced in *pure*$(\mathcal{P})$.

We will approach this problem as follows. Suppose there are an infinite number of messages that arrive at some node y during its step t. Since in a network there are only a finite number of nodes and a node can only send a finite number of messages during each step (the active domain is finite), there must be at least one node x that sends messages to step t of y during an infinite number of steps of x. Hence there is no maximum value amongst the corresponding send-timestamps of x. Thus, in order to prevent the arrival of an infinite number of messages at step t of y, it will be sufficient to demand that there always *is* such a maximum send-timestamp for every sender. Below, we will implement this strategy with some concrete rules in *pure*$(\mathcal{P})$.

We use a fact $\texttt{rcvInf}(y, t)$ to express that node y receives an infinite number of messages during its step t. Below we add new rules, and their intuition is that they are relative to an addressee and a step of this addressee, represented by the variables `y` and `t` respectively. First, we add the following rule to *pure*$(\mathcal{P})$ for *each* relation $\texttt{chosen}_R$ that results from the transformation of asynchronous rules, where `x`, `s`, `y`, and `t` are variables and $\bar{\texttt{w}}$ is a tuple of distinct variables disjoint from the previous ones with $|\bar{\texttt{w}}|$ the arity of relation R in *sch*$(\mathcal{P})$:

$$\texttt{hasSender}(\texttt{y}, \texttt{t}, \texttt{x}, \texttt{s}) \leftarrow \texttt{chosen}_R(\texttt{x}, \texttt{s}, \texttt{y}, \texttt{t}, \bar{\texttt{w}}), \neg\texttt{rcvInf}(\texttt{y}, \texttt{t}). \tag{5.10}$$

This rule intuitively means that as long as addressee `y` has not received an infinite number of messages during its step `t`, we register the senders and their send-timestamps. Recall that $<^{(2)} \in \mathcal{D}_{\text{time}}$. Next, we add to *pure*$(\mathcal{P})$ the following rules, for which the intuition is provided below:

$$\begin{aligned}\texttt{isSmaller}(\texttt{y}, \texttt{t}, \texttt{x}, \texttt{s}) \leftarrow\ & \texttt{hasSender}(\texttt{y}, \texttt{t}, \texttt{x}, \texttt{s}), \texttt{hasSender}(\texttt{y}, \texttt{t}, \texttt{x}, \texttt{s}'),\\ & \texttt{s} < \texttt{s}'.\end{aligned} \tag{5.11}$$

$$\texttt{hasMax}(\texttt{y}, \texttt{t}, \texttt{x}) \leftarrow \texttt{hasSender}(\texttt{y}, \texttt{t}, \texttt{x}, \texttt{s}), \neg\texttt{isSmaller}(\texttt{y}, \texttt{t}, \texttt{x}, \texttt{s}). \tag{5.12}$$

$$\texttt{rcvInf}(\texttt{y}, \texttt{t}) \leftarrow \texttt{hasSender}(\texttt{y}, \texttt{t}, \texttt{x}, \texttt{s}), \neg\texttt{hasMax}(\texttt{y}, \texttt{t}, \texttt{x}). \tag{5.13}$$

The rule (5.11) checks for each sender and each of its send-timestamps whether there is a later send-timestamp of that same sender. The rule (5.12) tries to find a maximum send-timestamp. Finally, the rule (5.13) derives a `rcvInf`-fact if no maximum send-timestamp was found for at least one sender.

We will show in Section 5.8.1 that in any stable model, the above rules make sure that every node receives only a finite number of messages at every step.

5.5.8 Input and Stable Models

Now we define the actual declarative semantics for $\mathcal{P}$. Let H be an input distributed database instance for $\mathcal{P}$, over a network $\mathcal{N}$. Let *pure*($\mathcal{P}$) be as previously constructed. We define *decl*(H) to be the following database instance over the schema $edb(\mathcal{P})^{\mathrm{LT}} \cup \{\mathtt{all}^{(1)}\} \cup \mathcal{D}_{\mathrm{time}}$:

$$\begin{aligned} decl(H) \quad = \quad & \{R(x, s, \bar{a}) \mid x \in \mathcal{N},\ s \in \mathbb{N},\ R(\bar{a}) \in H(x)\} \\ & \cup \{\mathtt{all}(x) \mid x \in \mathcal{N}\} \cup I_{\mathrm{time}}. \end{aligned}$$

In words: we make for each node its input facts available at all timestamps; we provide the set of all nodes; and, I_{time} provides the timestamps with comparison relations (see Section 5.5.5). Note, instance *decl*(H) is infinite because $\mathbb{N}$ is infinite.

Recall the *stable model semantics* for Datalog$^{\neg}$ programs, as reviewed in Section 2.7.3. We now define the declarative semantics for $\mathcal{P}$ on input H:

Definition 5.8. Any stable model of *pure*($\mathcal{P}$) on input *decl*(H) is called a *model* of $\mathcal{P}$ on input H.

Importantly, we are using stable models of *pure*($\mathcal{P}$), not of $\mathcal{P}$.

5.6 Main Result

Recall from Section 5.5.2 the definition of the trace of a run, representing in detail the computation of that run. Our main result shows that our declarative semantics of Dedalus expresses exactly the same computations as our operational semantics:

Theorem 5.9. Let $\mathcal{P}$ be a Dedalus program and let H be an input distributed database instance for $\mathcal{P}$. On input H,

(i) for every fair run $\mathcal{R}$ of $\mathcal{P}$ there is a model M of $\mathcal{P}$ such that $trace(\mathcal{R}) = M|_{sch(\mathcal{P})^{\mathrm{LT}}}$, and

(ii) for every model M of $\mathcal{P}$ there is a fair run $\mathcal{R}$ of $\mathcal{P}$ such that $trace(\mathcal{R}) = M|_{sch(\mathcal{P})^{\mathrm{LT}}}$.

□

The proof of item *(i)* of the theorem is described in Section 5.7. The proof of item *(ii)*, which is the most difficult, is described in Section 5.8. We only describe the crucial reasoning steps of the proofs; the intricate technical details can be found in [11].

5.7 Run to Model

Let $\mathcal{P}$ be a Dedalus program and let H be an input for $\mathcal{P}$, over some network $\mathcal{N}$. Let $\mathcal{R}$ be a fair run of $\mathcal{P}$ input H. In this section, we show that there is a model M of $\mathcal{P}$ on input H such that $trace(\mathcal{R}) = M|_{sch(\mathcal{P})^{\mathrm{LT}}}$. The main idea is that we translate the transitions of $\mathcal{R}$ to facts over the schema of *pure*($\mathcal{P}$).

5.7.1 Construction

In this section we construct M. We define

$$M = \mathit{decl}(H) \cup \bigcup_{i\in\mathbb{N}} \mathrm{trans}^{[i]}_{\mathcal{R}},$$

where $\mathrm{trans}^{[i]}_{\mathcal{R}}$ for each $i \in \mathbb{N}$ is an instance over the schema of $\mathit{pure}(\mathcal{P})$ that describes the transition i, which is detailed below. First, the reason for including $\mathit{decl}(H)$ is because if we want M to be a stable model, it must always contain the input $\mathit{decl}(H)$ (see Section 2.7.3). Let $i \in \mathbb{N}$. We define $\mathrm{trans}^{[i]}_{\mathcal{R}}$ as

$$\mathrm{trans}^{[i]}_{\mathcal{R}} = \mathrm{caus}^{[i]}_{\mathcal{R}} \cup \mathrm{fin}^{[i]}_{\mathcal{R}} \cup \mathrm{duc}^{[i]}_{\mathcal{R}} \cup \mathrm{snd}^{[i]}_{\mathcal{R}},$$

where each of these new sets focuses on different aspects of transition i, and they are defined next, together with their intuition with respect to $\mathit{pure}(\mathcal{P})$.

Before we continue, we recall and define some symbols. Let $\alpha_{\mathcal{R}}$ denote the arrival function of $\mathcal{R}$, as defined in Section 2.8.2.4. For the run $\mathcal{R}$, let $\mathit{loc}_{\mathcal{R}}(\cdot)$ and $\mathit{glob}_{\mathcal{R}}(\cdot)$ be as defined in Section 5.5.1, and let $\prec_{\mathcal{R}}$ be the happens-before relation as defined in Section 5.5.3. Let x_i denote the active node of transition i, and abbreviate $s_i = \mathit{loc}_{\mathcal{R}}(i)$.

Causality The set $\mathrm{caus}^{[i]}_{\mathcal{R}}$ represents the pairs $(x, s) \in \mathcal{N} \times \mathbb{N}$ that causally happen before (x_i, s_i). We define $\mathrm{caus}^{[i]}_{\mathcal{R}}$ to consist of all facts $\texttt{before}(x, s, x_i, s_i)$ for which $(x, s) \in \mathcal{N} \times \mathbb{N}$ and $(x, s) \prec_{\mathcal{R}} (x_i, s_i)$. This represents the joint result of rules (5.1), (5.2), and (5.9), corresponding to respectively the local edges, transitive edges, and message edges of $\prec_{\mathcal{R}}$.

Finite Messages The set $\mathrm{fin}^{[i]}_{\mathcal{R}}$ represents that only a finite number of messages are delivered in transition i, thus at step s_i of node x_i. First, let $\mathrm{senders}^{[i]}_{\mathcal{R}}$ denote all pairs $(x, s) \in \mathcal{N} \times \mathbb{N}$ such that, denoting $j = \mathit{glob}_{\mathcal{R}}(x, s)$, for some fact $\boldsymbol{f}$ we have $\alpha_{\mathcal{R}}(j, x_i, \boldsymbol{f}) = i$, i.e., the node x during its step s sends a message to x_i with arrival timestamp s_i. It follows from the operational semantics that for each $(x, s) \in \mathrm{senders}^{[i]}_{\mathcal{R}}$ we have $\mathit{glob}_{\mathcal{R}}(x, s) < i$.

We define $\mathrm{fin}^{[i]}_{\mathcal{R}}$ to consist of the following facts:

- the fact $\texttt{hasSender}(x_i, s_i, x, s)$ for each $(x, s) \in \mathrm{senders}^{[i]}_{\mathcal{R}}$, representing the result of rule (5.10);
- the fact $\texttt{isSmaller}(x_i, s_i, x, s)$ for each $(x, s) \in \mathrm{senders}^{[i]}_{\mathcal{R}}$ and $(x, s') \in \mathrm{senders}^{[i]}_{\mathcal{R}}$ with $s < s'$, representing the result of rule (5.11);
- the fact $\texttt{hasMax}(x_i, s_i, x)$ for each sender-node x mentioned in $\mathrm{senders}^{[i]}_{\mathcal{R}}$, representing the result of rule (5.12).

We know that in $\mathcal{R}$ only a finite number of messages arrive at step s_i of x_i. Hence we add no fact $\texttt{rcvInf}(x_i, s_i)$ to $\mathrm{fin}^{[i]}_{\mathcal{R}}$. This also explains why the specification of the $\texttt{hasMax}$-facts above is relatively simple: there is always a maximum send-timestamp for each sender-node.

Deductive The set $\mathrm{duc}^{[i]}_{\mathcal{R}}$ represents all facts over $sch(\mathcal{P})$ that are available at x_i during transition i, thus during step s_i of x_i. Let D_i denote the output of subprogram $deduc_{\mathcal{P}}$ during transition i. We define $\mathrm{duc}^{[i]}_{\mathcal{R}}$ to consist of the facts $D_i^{\Uparrow x_i, s_i}$. This represents the result of rules in $pure(\mathcal{P})$ of the form (5.3), (5.4) and (5.8).

Sending The set $\mathrm{snd}^{[i]}_{\mathcal{R}}$ represents the sending of messages during transition i. Let $async_{\mathcal{P}}$ be the subprogram of $\mathcal{P}$ as defined in Section 2.8.2.2, and let $\mathrm{mesg}^{[i]}_{\mathcal{R}}$ denote the output of $async_{\mathcal{P}}$ during transition i, restricted to the facts having their addressee-component in the network.

We define $\mathrm{snd}^{[i]}_{\mathcal{R}}$ to consist of the following facts:

- the fact $\texttt{cand}_R(x_i, s_i, y, t, \bar{a})$ for each $R(y, \bar{a}) \in \mathrm{mesg}^{[i]}_{\mathcal{R}}$ and $t \in \mathbb{N}$ such that $(y, t) \not\prec_{\mathcal{R}} (x_i, s_i)$, representing the result of rule (5.5);
- all facts $\texttt{chosen}_R(x_i, s_i, y, t, \bar{a})$ and $\texttt{other}_R(x_i, s_i, y, u, \bar{a})$ for which $R(y, \bar{a}) \in \mathrm{mesg}^{[i]}_{\mathcal{R}}$, $t = loc_{\mathcal{R}}(j)$ with $j = \alpha_{\mathcal{R}}(i, y, R(\bar{a}))$, $u \in \mathbb{N}$, $(y, u) \not\prec_{\mathcal{R}} (x_i, s_i)$ and $u \neq t$. This represents the choice of an arrival timestamp for the messages, as performed by rules (5.6) and (5.7).

5.7.2 Final Steps

It can be shown that M is a model of $\mathcal{P}$ on input H [11]. By construction of M, we have, as desired:

$$M|_{sch(\mathcal{P})^{\mathrm{LT}}} = \bigcup_{i \in \mathbb{N}} \mathrm{duc}^{[i]}_{\mathcal{R}} = \bigcup_{i \in \mathbb{N}} D_i^{\Uparrow x_i, s_i} = trace(\mathcal{R}).$$

5.8 Model to Run

Let $\mathcal{P}$ be a Dedalus program and let H be an input for $\mathcal{P}$, over some network $\mathcal{N}$. Let M be a model of $\mathcal{P}$ on input H. In this section we show there is a fair run $\mathcal{R}$ of $\mathcal{P}$ on H such that $trace(\mathcal{R}) = M|_{sch(\mathcal{P})^{\mathrm{LT}}}$.

The direction shown in Section 5.7 is perhaps the most intuitive direction because we only have to show that a concrete set of facts is actually a stable model. In this section we do not yet understand what M contains. So, a first important step is to show that M has some desirable properties which allow us to construct a run from it.

First, it is important to know that in M we find location specifiers where we expect location specifiers and we find timestamps where we expect timestamps. Formally, we call M *well-formed* if:

- for each $R(x, s, \bar{a}) \in M|_{sch(\mathcal{P})^{\mathrm{LT}}}$ we have $x \in \mathcal{N}$ and $s \in \mathbb{N}$;
- for each $\texttt{before}(x, s, y, t) \in M$, we have $x, y \in \mathcal{N}$ and $s, t \in \mathbb{N}$;
- for each fact $\texttt{cand}_R(x, s, y, t, \bar{a})$, $\texttt{chosen}_R(x, s, y, t, \bar{a})$ and $\texttt{other}_R(x, s, y, t, \bar{a})$ in M, we have $x, y \in \mathcal{N}$ and $s, t \in \mathbb{N}$;
- for each fact $\texttt{hasSender}(x, s, y, t)$, $\texttt{isSmaller}(x, s, y, t)$, $\texttt{hasMax}(x, s, y)$ and $\texttt{rcvInf}(x, s)$ in M, we have $x, y \in \mathcal{N}$ and $s, t \in \mathbb{N}$.

Using the notation from Section 2.7.3, let $gr_M^{\mathcal{P},H}$ abbreviate the ground program

$$ground_M(pure(\mathcal{P}), decl(H)).$$

By definition of M as a stable model, we have $M = gr_M^{\mathcal{P},H}(decl(H))$. It can be shown by induction on the fixpoint computation of $gr_M^{\mathcal{P},H}$ on input $decl(H)$ that M is always well-formed. We omit the details.

5.8.1 Partial Order

Based on the relation `before` in M, in this subsection we define a strict partial order $\prec_M$ on $\mathcal{N} \times \mathbb{N}$. This forms a crucial insight in the causality information represented by M. In the following subsection, we use this partial order to establish a total order on $\mathcal{N} \times \mathbb{N}$, around which we can build a run. The idea is that this total order tells us which are the active nodes in the transitions of the constructed run.

We define $\prec_M$ as follows. For each $(x, s) \in \mathcal{N} \times \mathbb{N}$ and $(y, t) \in \mathcal{N} \times \mathbb{N}$, we write $(x, s) \prec_M (y, t)$ if and only if $\texttt{before}(x, s, y, t) \in M$.

Let $gr_M^{\mathcal{P},H}$ be as above. A ground rule $\psi \in gr_M^{\mathcal{P},H}$ is called *active* if $pos_\psi \subseteq M$, which implies that $head_\psi \in M$ because M is stable. Now, an edge $(x, s) \prec_M (y, t)$ is called a *local edge*, a *message edge* or a *transitive edge* if the fact $\texttt{before}(x, s, y, t) \in M$ is the head of an active ground rule in $gr_M^{\mathcal{P},H}$ that is of respectively the form (5.1), the form (5.9), or the form (5.2). It is possible that an edge is of two or even three types at the same time.

The rest of this section is dedicated to showing that $\prec_M$ has certain desirable properties, so that we can later derive a total order with desirable properties. First, consider the following claim:

Claim 5.10. Relation $\prec_M$ is a strict partial order on $\mathcal{N} \times \mathbb{N}$.

Proof. We show that $\prec_M$ is transitive and irreflexive.

Transitive First, we show that $\prec_M$ is transitive. Suppose we have $(x, s) \prec_M (z, u)$ and $(z, u) \prec_M (y, t)$. We have to show that $(x, s) \prec_M (y, t)$. By definition of $\prec_M$, we have $\texttt{before}(x, s, z, u) \in M$ and $\texttt{before}(z, u, y, t) \in M$. Because the rule (5.2) is positive, we have the following ground rule in $gr_M^{\mathcal{P},H}$:

$$\texttt{before}(x, s, y, t) \leftarrow \texttt{before}(x, s, z, u),\ \texttt{before}(z, u, y, t).$$

Because M is a stable model and the body of the previous ground rule is in M, we obtain $\texttt{before}(x, s, y, t) \in M$. Hence, $(x, s) \prec_M (y, t)$, as desired.

Irreflexive Because an edge $(x, s) \prec_M (x, s)$ for any $(x, s) \in \mathcal{N} \times \mathbb{N}$ would form a cycle of length one, it is sufficient to show that there are no cycles in $\prec_M$ at all. This gives us irreflexivity, as desired.

First, let $\prec'_M$ denote the restriction of $\prec_M$ to the edges that are local or message edges. Note that this definition allows some edges in $\prec'_M$ to also be transitive. The edges that are missing from $\prec'_M$ with respect to $\prec_M$ are only derivable by ground rules of the form (5.2); we call these the *pure* transitive edges. We start by showing that $\prec'_M$ contains no cycles. We show this with a proof by contradiction. So, suppose that there is a cycle in $\mathcal{N} \times \mathbb{N}$ through the edges of $\prec'_M$:

$$(x_1, s_1) \prec_M (x_2, s_2) \prec_M \ldots \prec_M (x_n, s_n)$$

with $n \geq 2$ and $(x_1, s_1) = (x_n, s_n)$. We have $\texttt{before}(x_i, s_i, x_{i+1}, s_{i+1}) \in M$ for each $i \in \{1, \ldots, n-1\}$. Moreover, based on these previous $\texttt{before}$-facts, ground rules in $gr_M^{\mathcal{P},H}$ of the form (5.2) will have derived $\texttt{before}(x_i, s_i, x_j, s_j) \in M$ for each $i, j \in \{1, \ldots, n\}$. If each edge on the above cycle would be only local, then for each $i, j \in \{1, \ldots, n\}$ with $i < j$ we have $x_i = x_j$ and $s_i < s_j$, and hence $s_1 \neq s_n$, which is false. So, there has to be some $k \in \{1, \ldots, n-1\}$ such that $(x_k, s_k) \prec_M (x_{k+1}, s_{k+1})$ is a message edge, derived by a ground rule of the form (5.9):

$$\texttt{before}(x_k, s_k, x_{k+1}, s_{k+1}) \leftarrow \texttt{chosen}_R(x_k, s_k, x_{k+1}, s_{k+1}, \bar{a}).$$

Therefore $\texttt{chosen}_R(x_k, s_k, x_{k+1}, s_{k+1}, \bar{a}) \in M$. This $\texttt{chosen}_R$-fact must be derived by a ground rule of the form (5.6) in $gr_M^{\mathcal{P},H}$, which implies that

$$\texttt{cand}_R(x_k, s_k, x_{k+1}, s_{k+1}, \bar{a}) \in M.$$

This $\texttt{cand}_R$-fact must in turn be derived by a ground rule ψ of the form (5.5). Because rules of the form (5.5) in *pure*($\mathcal{P}$) contain a negative $\texttt{before}$-atom in their body, the presence of ψ in $gr_M^{\mathcal{P},H}$ requires that $\texttt{before}(x_{k+1}, s_{k+1}, x_k, s_k) \notin M$. But that is a contradiction, because $\texttt{before}(x_i, s_i, x_j, s_j) \in M$ for each $i, j \in \{1, \ldots, n\}$ (see above).

Now we show there are no cycles in the entire relation $\prec_M$. Using that $M = gr_M^{\mathcal{P},H}(decl(H))$, we have $M = \bigcup_{i \in \mathbb{N}} M_i$ where $M_0 = decl(H)$ and $M_i = T(M_{i-1})$ for each $i \geq 1$ where T is the immediate consequence operator of $gr_M^{\mathcal{P},H}$. By induction on i, we show that an edge $\texttt{before}(x, s, y, t) \in M_i$ either is a local or message edge, or it can be replaced by a path of local or message edges in M_i. Then any cycle in $\prec_M$ would imply there is a cycle in $\prec'_M$, which is impossible (shown above). So, $\prec_M$ can not contain cycles. Now, this induction property is satisfied for the base case because M_0 does not contain $\texttt{before}$-facts. For the induction hypothesis, assume the property holds for M_{i-1}, where $i \geq 1$. For the inductive step, let $\texttt{before}(x, s, y, t) \in M_i \setminus M_{i-1}$. If this fact is derived by a ground rule of the form (5.1) or (5.9) then the property is satisfied. Now suppose the fact is derived by a ground rule of the form (5.2):

$$\texttt{before}(x, s, y, t) \leftarrow \texttt{before}(x, s, z, u),\ \texttt{before}(z, u, y, t).$$

Both body facts are in M_{i-1}, so the induction hypothesis implies that M_{i-1} contains a path of local or message edges from (x, s) to (z, u) and from (z, u) to (y, t). Hence, using $M_{i-1} \subseteq M_i$, the edge $\texttt{before}(x, s, y, t) \in M_i$ can be replaced by a path of local or message edges in M_i. $\square$

In Section 5.5.7.4 we have added extra rules to *pure*($\mathcal{P}$) to enforce that every node only receives a finite number of messages during each step. We now verify that this works correctly:

Claim 5.11. For each $(y, t) \in \mathcal{N} \times \mathbb{N}$ there are only a finite number of pairs $(x, s) \in \mathcal{N} \times \mathbb{N}$ such that $(x, s) \prec_M (y, t)$ is a message edge.

Proof. We start by noting that M does not contain the fact $\texttt{rcvInf}(y,t)$. Indeed, in order to derive this fact, we need a ground rule in $gr_M^{\mathcal{P},H}$ of the form (5.13), which has a body fact of the form $\texttt{hasSender}(y,t,x,s)$. Such hasSender-facts must be generated by ground rules in $gr_M^{\mathcal{P},H}$ of the form (5.10). The rule (5.10) negatively depends on relation rcvInf. Thus, specifically, if we want a ground rule in $gr_M^{\mathcal{P},H}$ that can derive $\texttt{hasSender}(y,t,x,s)$, we should require the absence of $\texttt{rcvInf}(y,t)$ from M. So $\texttt{rcvInf}(y,t) \in M$ requires $\texttt{rcvInf}(y,t) \notin M$, which is impossible.

The rest of the proof works towards a contradiction. So, suppose that (y,t) has an infinite number of incoming message edges. Because there are only a finite number of nodes in $\mathcal{N}$, there has to be a node x that has an infinite number of timestamps s for which there exists some relation R in *idb*$(\mathcal{P})$ and a tuple $\bar{a}$ such that $\texttt{chosen}_R(x,s,y,t,\bar{a}) \in M$. Because $\texttt{rcvInf}(y,t) \notin M$ (see above), for *each* of these $\texttt{chosen}_R$-facts, there is a ground rule of the form (5.10) in M that derives $\texttt{hasSender}(y,t,x,s) \in M$.

Rule (5.13) has a negative hasMax-atom in its body. If we can show $\texttt{hasMax}(y,t,x) \notin M$, then there is a ground rule in $gr_M^{\mathcal{P},H}$ of the form (5.13), where $\texttt{hasSender}(y,t,x,s) \in M$:

$$\texttt{rcvInf}(y,t) \leftarrow \texttt{hasSender}(y,t,x,s).$$

This then causes $\texttt{rcvInf}(y,t) \in M$, giving the desired contradiction.

Also towards a proof by contradiction, suppose that $\texttt{hasMax}(y,t,x) \in M$. This means that there is a ground rule ψ in $gr_M^{\mathcal{P},H}$ of the form (5.12):

$$\texttt{hasMax}(y,t,x) \leftarrow \texttt{hasSender}(y,t,x,s).$$

Because the rule (5.12) contains a negative isSmaller-atom in the body, and because $\psi \in gr_M^{\mathcal{P},H}$, we know that $\texttt{isSmaller}(y,t,x,s) \notin M$. But because there are infinitely many facts of the form $\texttt{hasSender}(y,t,x,s') \in M$, there is at least one fact $\texttt{hasSender}(y,t,x,s') \in M$ with $s < s'$. Moreover, the rule (5.11) is positive, and therefore the following ground rule is always in $gr_M^{\mathcal{P},H}$:

$$\texttt{isSmaller}(y,t,x,s) \leftarrow \texttt{hasSender}(y,t,x,s),\ \texttt{hasSender}(y,t,x,s'),\ s < s'.$$

Since the body of this ground rule is in M, the rule derives $\texttt{isSmaller}(y,t,x,s) \in M$, which gives the desired contradiction. ☐

An ordering $\prec$ on a set A is called *well-founded* if for each $a \in A$, there are only a finite number of elements $b \in A$ such that $b \prec a$. We now use Claim 5.11 to show:

Claim 5.12. Relation $\prec_M$ on $\mathcal{N} \times \mathbb{N}$ is well-founded.

Proof. Let $(x,s) \in \mathcal{N} \times \mathbb{N}$. We have to show that there are only a finite number of pairs $(y,t) \in \mathcal{N} \times \mathbb{N}$ such that $(y,t) \prec_M (x,s)$. Technically, we can limit our attention to paths in $\prec_M$ consisting of local edges and message edges, because if we can show that there are only a finite number of predecessors of (x,s) on such paths, then there are only a finite number of predecessors when we include the transitive edges as well.

First we show that every pair $(y,t) \in \mathcal{N} \times \mathbb{N}$ has only a finite number of incoming local and message edges. If $t > 0$, we can immediately see that (y,t) has precisely one incoming local edge, as created by a ground rule of the form (5.1), and if $t = 0$ then

(y, t) has no incoming local edge. Also, Claim 5.11 tells us that (y, t) has only a finite number of incoming message edges. So, the number of incoming local and message edges in (y, t) is finite.

Let $(y, t) \in \mathcal{N} \times \mathbb{N}$ be a pair such that $(y, t) \prec_M (x, s)$ is a local edge or a message edge. Starting in (x, s), we can follow this edge backwards so that we reach (y, t). If (y, t) itself has incoming local or message edges, from (y, t) we can again follow an edge backwards. This way we can incrementally construct backward paths starting from (x, s). Because at each pair of $\mathcal{N} \times \mathbb{N}$ there are only a finite number of incoming local or message edges (shown above), if (x, s) would have an infinite number of predecessors, we must be able to construct a backward path of infinite length. We now show that the existence of such an infinite path leads to a contradiction. So, suppose that there is a backward path of infinite length. Because there are only a finite number of nodes in the network $\mathcal{N}$, there must be a node y that occurs infinitely often on this path. We will now show that, as we progress further along the backward path, we must see the local timestamps of y strictly decrease. Hence, we must eventually reach timestamp 0 of y, after which we cannot decrement the timestamps of y anymore, and thus it is impossible that y occurs infinitely often along the path. Suppose that the timestamps of y do not strictly decrease. There are two cases. First, if the same pair (y, t) would occur twice on the path, we would have a cycle in $\prec_M$, which is not possible by Claim 5.10. Secondly, suppose that there are two timestamps t and t' of y such that $t < t'$ and (y, t) occurs before (y, t') on the backward path, meaning that (y, t) lies closer to (x, s). Because the edges were followed in reverse, we have

$$(y, t') \prec_M \ldots \prec_M (y, t).$$

But since $t < t'$, by means of local edges, we always have

$$(y, t) \prec_M (y, t + 1) \prec_M \ldots \prec_M (y, t').$$

Combining these two sets of edges leads to a cycle in $\prec_M$, which is impossible by Claim 5.10. ☐

5.8.2 Construction of Run

Let $\prec_M$ be the well-founded strict partial order on $\mathcal{N} \times \mathbb{N}$ as defined in the preceding subsection. The relation $\prec_M$ has the intuition of a happens-before relation of a run (see Section 5.5.3), but the novelty is that it comes from a purely declarative model M. We will now use $\prec_M$ to construct a run $\mathcal{R}$ such that $\mathit{trace}(\mathcal{R}) = M|_{\mathit{sch}(\mathcal{P})^{\mathrm{LT}}}$.

Total order It is well-known that a well-founded strict partial order can be extended to a well-founded strict total order. So, let $<_M$ be a well-founded strict total order on $\mathcal{N} \times \mathbb{N}$ that extends $\prec_M$, i.e., for each $(x, s) \in \mathcal{N} \times \mathbb{N}$ and $(y, t) \in \mathcal{N} \times \mathbb{N}$, if $(x, s) \prec_M (y, t)$ then $(x, s) <_M (y, t)$, but the reverse does not have to hold.

Intuitively, ordering the set $\mathcal{N} \times \mathbb{N}$ according to $<_M$ gives us a sequence of pairs that will form the transitions in the constructed run $\mathcal{R}$. Concretely, we obtain a sequence of nodes by taking the node-component from each pair. This will form our sequence of active nodes. Similarly, by taking the timestamp-component from each pair of $\mathcal{N} \times \mathbb{N}$, we obtain a sequence of timestamps. These are the local clocks of the active nodes during their transitions.

We introduce some extra notations to help us reason about the ordering of time that is implied by $<_M$. For each $(x, s) \in \mathcal{N} \times \mathbb{N}$, let $glob_M(x, s) \in \mathbb{N}$ denote the ordinal of (x, s) as implied by $<_M$, which is well-defined because $<_M$ is well-founded. For technical convenience, we let ordinals start at 0. Note, $glob_M(\cdot)$ is an injective function. For any $i \in \mathbb{N}$, we define (x_i, s_i) to be the unique pair in $\mathcal{N} \times \mathbb{N}$ such that $glob_M(x_i, s_i) = i$.

As a counterpart to function $glob_M(\cdot)$, for $i \in \mathbb{N}$ and $x \in \mathcal{N}$, let $loc_M(i, x)$ denote the *size* of the set

$$\{s \in \mathbb{N} \mid glob_M(x, s) < i\}.$$

Intuitively, if i is regarded to be the ordinal of a transition in a run, $loc_M(i, x)$ is the number of local steps of x that came before transition i, i.e., the number of transitions before i in which x was the active node. If $x = x_i$ (the active node) then $loc_M(i, x)$ is effectively the timestamp of x *during* transition i, and if $x \neq x_i$ then $loc_M(i, x)$ is the next timestamp of x that still has to come *after* transition i. Note, the functions $glob_M(\cdot)$ and $loc_M(\cdot)$ closely resemble the functions $glob_{\mathcal{R}}(\cdot)$ and $loc_{\mathcal{R}}(\cdot)$ of Section 5.5.1.

Configurations We will now define the desired run $\mathcal{R}$ of $\mathcal{P}$ on H. First we define the (infinite) sequence of configurations ρ_0, ρ_1, ρ_2, etc. In a second step we will connect each pair of subsequent configurations by a transition.

Recall from Section 2.8.2.1 that a configuration describes for each node what facts it has stored locally (state), and also what messages have been sent to this node but that are not yet received (message buffer). The facts that are stored on a node are either input *edb*-facts, or facts derived by inductive rules in a previous step of the node. The first kind of facts can be easily obtained from M by keeping only the facts over schema $edb(\mathcal{P})^{\mathrm{LT}}$, which gives a subset of $decl(H)$. For the second kind of facts, we look at the inductively derived facts in M, which is detailed next. The rules in $pure(\mathcal{P})$ that represent inductive rules of $\mathcal{P}$ are easily recognizable: they are of the form (5.4), meaning that they have a head atom over $sch(\mathcal{P})^{\mathrm{LT}}$ and they have a positive body `tsucc`-atom. No other kind of rule in $pure(\mathcal{P})$ has this form. Hence, the ground rules in $gr_M^{\mathcal{P},H}$ that are based on rules of the form (5.4) are also easily recognizable, and we will call these *inductive ground rules*. A ground rule $\psi \in gr_M^{\mathcal{P},H}$ is called *active* on M if $pos_\psi \subseteq M$, which implies that $head_\psi \in M$ because M is stable. Let M^{ind} denote all head atoms of inductive ground rules in $gr_M^{\mathcal{P},H}$ that are active on M. Note, $M^{\mathrm{ind}} \subseteq M$. Now, for each $i \in \mathbb{N}$, for each node $x \in \mathcal{N}$, in configuration $\rho_i = (s_i, b_i)$, the state $s_i(x)$ is defined as

$$\left((M|_{edb(\mathcal{P})^{\mathrm{LT}}})|^{x,s} \cup M^{\mathrm{ind}}|^{x,s}\right)^{\Downarrow},$$

where $s = loc_M(i, x)$, and where we use notations from Section 5.5.2. Note, we remove the location specifier and timestamp because we have to obtain facts over the schema of $\mathcal{P}$, not over the schema of $pure(\mathcal{P})$.

Now we define the message buffers in the configurations. Recall that the message buffer of a node always contains pairs of the form $(j, \boldsymbol{f})$, where $j \in \mathbb{N}$ is the transition in which the fact $\boldsymbol{f}$ over $idb(\mathcal{P})$ was sent. For each $i \in \mathbb{N}$, for each node $x \in \mathcal{N}$, in configuration $\rho_i = (s_i, b_i)$, the message buffer $b_i(x)$ is defined as

$$\{(\mathit{glob}_M(y,t),\ R(\bar{a})) \mid \\ \exists u : \texttt{chosen}_R(y,t,x,u,\bar{a}) \in M,\ \mathit{glob}_M(y,t) < i \leq \mathit{glob}_M(x,u)\}.$$

Note the use of addressee x in the definition of $b_i(x)$. The definition of $b_i(x)$ reflects the operational semantics, in that the messages in the buffer of node x must be sent in a previous transition, as expressed by the constraint $\mathit{glob}_M(y,t) < i$. Moreover, the constraint $i \leq \mathit{glob}_M(x,u)$ says that $b_i(x)$ contains only messages that will be delivered in transitions of x that come after configuration ρ_i. Possibly $i = \mathit{glob}_M(x,u)$, and in that case the message will be delivered in the transition immediately after configuration ρ_i (see also below).

Transitions So far we have obtained a sequence of configurations ρ_0, ρ_1, ρ_2, etc. Now we define a sequence of tuples, one tuple per ordinal $i \in \mathbb{N}$, that represents the transition i. Let $i \in \mathbb{N}$. The tuple τ_i is defined as $(\rho_i, x_i, m_i, i, \rho_{i+1})$, also denoted as $\rho_i \xrightarrow[i]{x_i,\, m_i} \rho_{i+1}$, where

$$m_i = \{(\mathit{glob}_M(y,t),\ R(\bar{a})) \mid \texttt{chosen}_R(y,t,z,u,\bar{a}) \in M,\ \mathit{glob}_M(z,u) = i\}.$$

Intuitively, in m_i, we select all messages that arrive in transition i. And since $\mathit{glob}_M(z,u) = i$ implies $z = x_i$ and $u = s_i$, we thus select all messages destined for step s_i of node x_i.

It can be shown that the sequence $\mathcal{R}$ is indeed a legal run of $\mathcal{P}$ on input H such that $\mathit{trace}(\mathcal{R}) = M|_{\mathit{sch}(\mathcal{P})^{\mathrm{LT}}}$ [11]. In the following subsection we show that $\mathcal{R}$ is also fair.

5.8.3 Fair Run

In this subsection we show that run $\mathcal{R}$ is fair. For each $i \in \mathbb{N}$, let $\rho_i = (s_i, b_i)$ denote the source configuration of transition i. Recall from Section 2.8.2.4 that we have to check two fairness conditions:

1. every node is the active node in an infinite number of transitions; and,
2. for every transition $i \in \mathbb{N}$, for every node $y \in \mathcal{N}$, for every $(j, \boldsymbol{f}) \in b_i(y)$, there is a transition k with $i \leq k$ in which $(j, \boldsymbol{f})$ is delivered (to y).

We show that the first fairness condition is satisfied by $\mathcal{R}$. Let $x \in \mathcal{N}$ be a node, and let $s \in \mathbb{N}$ be a timestamp of x. Consider transition $i = \mathit{glob}_M(x,s)$. This transition has active node $x_i = x$. We can find such a transition with active node x for every timestamp $s \in \mathbb{N}$ of x, and these transitions are all unique because function $\mathit{glob}_M(\cdot)$ is injective. So, there are an infinite number of transitions in $\mathcal{R}$ with active node x, and the first fairness condition is satisfied.

Now we show that the second fairness condition is satisfied. Let $i \in \mathbb{N}$, $y \in \mathcal{N}$, and $(j, \boldsymbol{f}) \in b_i(y)$. Denote $\boldsymbol{f} = R(\bar{a})$. By definition of $b_i(y)$, pair $(j, \boldsymbol{f})$ implies that there are values $x \in \mathcal{N}$, $s \in \mathbb{N}$ and $t \in \mathbb{N}$ such that $\texttt{chosen}_R(x,s,y,t,\bar{a}) \in M$ and $j = \mathit{glob}_M(x,s) < i \leq \mathit{glob}_M(y,t)$. Denote $k = \mathit{glob}_M(y,t)$. Hence, $i \leq k$ and $(j, \boldsymbol{f}) \in m_k$ by definition of m_k. Thus $(j, \boldsymbol{f})$ is delivered to $x_k = y$ in transition k, as desired.

Chapter 6

The CRON Conjecture

6.1 Outline

In this chapter we use Dedalus to investigate the CRON conjecture by Hellerstein. First, Section 6.2 gives related work, and Section 6.3 gives technical remarks specific to this chapter. Next, Section 6.4 formalizes the output of a Dedalus program. Section 6.5 states the CRON conjecture and gives a formalization of non-causality. Section 6.6 contains the results. Because the output of a Dedalus program plays an important role in this chapter, Section 6.7 also shortly discusses the expressivity of Dedalus.

6.2 Related Work

This chapter is the extended version of our conference paper [18]. Some proof details are not included in this chapter, but can be found in the technical report [20].

6.3 General Remarks

Recall the language Dedalus and its operational semantics from Section 2.8. In this chapter, ordinals of transitions and configurations in a run start at 0 for technical convenience.

Also recall from Section 5.5.1 the definition of *timestamps* in the operational semantics. For example, suppose we have the following sequence of active nodes in a run: x, y, y, x, x, etc. If we would write the timestamps next to the nodes, we get this sequence: $(x, 0)$, $(y, 0)$, $(y, 1)$, $(x, 1)$, $(x, 2)$, etc.

Some constructions of Chapter 5 are reused in this chapter. For this reason, we will sometimes refer to Datalog$^\neg$ rules from Chapter 5.

6.4 Output

Let $\mathcal{P}$ be a Dedalus program. We formalize the output of a run. Assume a subset $out(\mathcal{P}) \subseteq idb(\mathcal{P})$, called the *output schema*, is selected: the relation names in $out(\mathcal{P})$ designate the intended output of the program. Following Marczak et al. [46], we define this output based on *ultimate* facts. In a run $\mathcal{R}$ of $\mathcal{P}$, we say that a fact $\boldsymbol{f}$ over schema $out(\mathcal{P})$ is *ultimate* at some node x if there is some transition of $\mathcal{R}$ after which $\boldsymbol{f}$ is output by $deduc_{\mathcal{P}}$ during every transition of x. Thus, $\boldsymbol{f}$ is eventually always present at x. The output of $\mathcal{R}$, denoted $output(\mathcal{R})$, is the union of the ultimate facts across all nodes. Note, we ignore what node is responsible for what piece of the output, following the intuition of cloud computing.

Because the operational semantics is nondeterministic, different runs can produce different outputs. Now, program $\mathcal{P}$ is called *consistent* if individually for every input H, every run produces the *same* output, which we denote as $outInst(\mathcal{P}, H)$. Guaranteeing or deciding consistency in special cases is an important research topic; see e.g. [1, 46], and Chapter 4.

6.5 CRON Conjecture and Non-Causality

We recall the CRON conjecture (Causality Required Only for Non-monotonicity), which was informally stated as follows [37]:

CRON Conjecture (Informal). Program semantics require causal message ordering if and only if the messages participate in non-monotonic derivations.

The CRON conjecture talks about an intuitive notion of “causality” on messages. As mentioned in Section 1.4, causality here stands for the physical constraint that an effect can only happen after its cause. Our operational semantics respects causality because a message can only be delivered after it was sent. When the delivery of one message causes another one to be sent, the second message is delivered in a later transition.

In order to obtain a conjecture that can be formally proved or disproved, we need a formal definition of “requiring causal message ordering”. To this end, we next introduce an alternative semantics for Dedalus that allows non-causality (sending messages “into the past”). This is then used in Section 6.6 to formally investigate the CRON conjecture.

6.5.1 Modeling Non-Causality

In Chapter 5, we have shown that the operational semantics of Dedalus is equivalent to a declarative semantics based on stable models. In Section 5.5, we have described a transformation to convert a Dedalus program $\mathcal{P}$ to a pure Datalog$^{\neg}$ program $pure(\mathcal{P})$ that contains extra rules to enforce causality on message sending in every stable model. In this chapter, we remove these causality rules and explain how stable models can now represent non-causal message sending.

6.5.1.1 Transformation

Let $\mathcal{P}$ be a Dedalus program. To model non-causality, we describe the *SZ-transformation* to transform $\mathcal{P}$ into $pure_{\mathrm{SZ}}(\mathcal{P})$, which is obtained exactly like $pure(\mathcal{P})$, with the only difference that the use of relation `before` is completely removed (including its rules). This is detailed below.

Again assuming that each rule of $\mathcal{P}$ has at least one positive body atom, the deductive and inductive rules of $\mathcal{P}$ are translated just as in $pure(\mathcal{P})$. For each asynchronous rule '$R(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}$' in $\mathcal{P}$, letting $\mathtt{x}$, $\mathtt{s}$ and $\mathtt{t}$ be variables not yet used in this rule and letting $\bar{\mathtt{w}}$ be a tuple of new and distinct variables with $|\bar{\mathtt{w}}| = |\bar{\mathtt{u}}|$, the previous rule transformation (5.5) is modified to become:

$$\mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}^{\Uparrow \mathtt{x}, \mathtt{s}}, \ \mathtt{all}(\mathtt{y}), \ \mathtt{time}(\mathtt{t}). \tag{6.1}$$

Note, we simply omit relation `before`. The rule transformations (5.6) to (5.8) are retained, and rule transformation (5.9) is omitted. As in $pure(\mathcal{P})$, a fact of the form $\mathtt{all}(x)$ means that x is a node of the network. Also, if multiple asynchronous rules in $\mathcal{P}$ have the same head predicate R, only new $\mathtt{cand}_R$-rules have to be added because the rules (5.6)–(5.8) are sufficiently general.

After transforming the rules of $\mathcal{P}$, we also include in $pure_{\mathrm{SZ}}(\mathcal{P})$ the rules of $pure(\mathcal{P})$ that enforce finite message delivery (see Section 5.5.7.4). This concludes the specification of program $pure_{\mathrm{SZ}}(\mathcal{P})$.

6.5.1.2 Semantics

The semantics for $pure_{\mathrm{SZ}}(\mathcal{P})$ is the same as for $pure(\mathcal{P})$, but we repeat this for clarity. Let H be an input for $\mathcal{P}$, over a network $\mathcal{N}$. We give $pure_{\mathrm{SZ}}(\mathcal{P})$ the input $decl(H)$, as defined in Section 5.5.8. We call any stable model M of $pure_{\mathrm{SZ}}(\mathcal{P})$ on $decl(H)$ an *SZ-model* of $\mathcal{P}$ on input H.

Note, program $pure_{\mathrm{SZ}}(\mathcal{P})$ does not enforce causality on the messages in M because the arrival timestamps can be chosen arbitrarily, even into the past. But causality could be respected in some models. In fact, $\mathcal{P}$ has at least one such "causal" SZ-model on every input. This is because $\mathcal{P}$ has at least one run on every input (possibly with only heartbeats), and because each run can be naturally encoded into an SZ-model (see Section 5.7, but omitting relation `before`).

Again, using the definition of stable model, it can be shown that M is always well-formed (cf. Section 5.8).

6.5.1.3 Output and Tolerating Non-Causality

The *output* of an SZ-model M, denoted $output(M)$, is defined with ultimate facts like in the operational semantics (Section 6.4):

$$output(M) = \bigcup_{R^{(k)} \in out(\mathcal{P})} \{R(\bar{a}) \mid \exists x \in \mathcal{N},\ \exists s \in \mathbb{N},\ \forall t \in \mathbb{N} : \ t \geq s \Rightarrow R(x, t, \bar{a}) \in M\}.$$

Now, we say that an already consistent Dedalus program $\mathcal{P}$ *tolerates non-causality* if individually for every input H, every SZ-model M yields the output $outInst(\mathcal{P}, H)$. Intuitively, if a consistent program tolerates non-causality, then it also computes the same result when messages can be sent into the past.

Algorithm 6.1 Program for emptiness query

```
$\texttt{empty(x)} \mid \texttt{y} \leftarrow \neg R(\,),\ \texttt{Id(x)},\ \texttt{Node(y)}.$
$\texttt{empty(x)}\bullet \leftarrow \texttt{empty(x)}.$
$\texttt{missing(\,)} \leftarrow \texttt{Node(x)},\ \neg\texttt{empty(x)}.$
$T(\,) \leftarrow \texttt{Id(x)},\ \neg\texttt{missing(\,)}.$
```

6.6 Results

We have considered a semantical and syntactical interpretation of the CRON conjecture, for which we present the results below.

6.6.1 Semantical Interpretation

We have first formalized the CRON conjecture purely on the semantical level, by relating causality to the monotonicity of the queries computed by Dedalus programs.

Like in Section 2.1, a *query* $\mathcal{Q}$ is a function from database instances over an input schema $\mathcal{D}_1$ to database instances over an output schema $\mathcal{D}_2$. Query $\mathcal{Q}$ is *monotone* if for all instances I and J over $\mathcal{D}_1$, $I \subseteq J$ implies $\mathcal{Q}(I) \subseteq \mathcal{Q}(J)$. Relating to the distributed setting, an instance I over a database schema $\mathcal{D}$ can be *partitioned* over a network $\mathcal{N}$ by putting each fact of I on at least one node, resulting in a distributed database instance over $\mathcal{N}$ and $\mathcal{D}$. Now, we say that a Dedalus program $\mathcal{P}$, for which *edb*($\mathcal{P}$) is the input schema of $\mathcal{Q}$, *(distributedly) computes* $\mathcal{Q}$ if $\mathcal{P}$ is consistent and for every input instance I for $\mathcal{Q}$, for every network $\mathcal{N}$, for every partition H of I over $\mathcal{N}$, we have $\mathit{outInst}(\mathcal{P}, H) = \mathcal{Q}(I)$. To compute non-monotone queries, every node needs its own identifier and the identifiers of the other nodes, or equivalent information (cf. Chapter 3). Therefore, we restrict attention to Dedalus programs $\mathcal{P}$ for which $\{\texttt{Id}^{(1)}, \texttt{Node}^{(1)}\} \subseteq \mathit{edb}(\mathcal{P})$, and each input H over a network $\mathcal{N}$ includes for each $x \in \mathcal{N}$ the facts $\{\texttt{Id}(x)\} \cup \{\texttt{Node}(y) \mid y \in \mathcal{N}\}$, which are treated just like any other *edb*-fact.

In this context, we have looked at the following formalization of the CRON conjecture:

CRON Conjecture (Semantical). A Dedalus program computes a monotone query if and only if it tolerates non-causality.

Both directions of this conjecture can be refuted by counterexamples, as we do in the following two subsections. So, contrary to the CALM conjecture (Chapter 3), a formalization of the CRON conjecture that is situated purely on the semantical level does not seem promising.

6.6.1.1 If Direction

To refute the if-direction of the semantical CRON conjecture, we give a Dedalus program tolerating non-causality that computes a non-monotone query.

Algorithm 6.2 Program for non-emptiness query

```
$A(\,) \mid \mathtt{x} \leftarrow R(\,),\ \mathtt{Id}(\mathtt{x}).$
$A(\,)\bullet \leftarrow A(\,).$
$B(\,) \mid \mathtt{x} \leftarrow A(\,),\ \neg\mathtt{sent}_B(\,),\ \mathtt{Id}(\mathtt{x}).$
$\mathtt{sent}_B(\,)\bullet \leftarrow A(\,).$
$T(\,) \leftarrow A(\,),\ B(\,).$
$T(\,)\bullet \leftarrow T(\,).$
```

Algorithm 6.1 repeats Example 5.2.[1] This Dedalus program computes the non-monotone emptiness query on a nullary relation R, that is, output "true" (encoded by a nullary relation T) if and only if R is empty (at all nodes). The asynchronous rule lets each node broadcast its own identifier if its relation R is empty. The inductive rule lets a node remember all received node identifiers. The deductive rules let a node output $T(\,)$ starting at the moment that it has all identifiers (including its own). This program is consistent.

Now we consider the tolerance to non-causality. Intuitively, in an SZ-model for this program, even if messages are sent into the past, the inductive rule persists any received identifier towards the future. If S is empty on all nodes, each node still has a timestamp after which it has all node identifiers. Thus every SZ-model yields the output $T(\,)$ if and only if all nodes have an empty relation S. So, the program tolerates non-causality. The proof details can be found in [20].

6.6.1.2 Only-If Direction

To refute the only-if direction of the semantical CRON conjecture, we give a Dedalus program computing a monotone query and that does not tolerate non-causality.

Algorithm 6.2 gives a (contrived) Dedalus program to compute the monotone non-emptiness query on a nullary relation R, that is, output "true" if and only if R is not empty (on at least one node). In the program, a node with nonempty relation R sends $A(\,)$ to itself. On receipt of $A(\,)$, the node stores $A(\,)$ and sends $B(\,)$ to itself if it has not previously done so. Thus, if a node sends $A(\,)$ then it sends $B(\,)$ precisely once. When the $B(\,)$ is later received, it is paired with the stored $A(\,)$, producing the fact $T(\,)$ that is stored by the inductive rule. This program is consistent.

However, the program does not tolerate non-causality, which we now explain. Let H be the input over singleton network $\{z\}$ with $H(z) = \{R(\,)\}$. On input H, we can exhibit an SZ-model M in which $A(\,)$-facts arrive at node z starting at timestamp 1, which implies that $\mathtt{sent}_B(\,)$ will exist starting at timestamp 2. This implies that $B(\,)$ is sent precisely once in M, namely, at timestamp 1. Now, the trick is to violate the causal dependency between relations A and B, by letting $B(\,)$ arrive in the past, at timestamp 0 of z, which is before any $A(\,)$ is received. Then the arriving $B(\,)$ cannot pair with any stored or arriving $A(\,)$. Since $B(\,)$ itself is not stored, we have thus

[1]With the only difference that we have now added the atom $\mathtt{Id}(\mathtt{x})$ in the last rule to have at least one positive body atom (see the assumption in Section 6.5.1.1).

Algorithm 6.3 Positive but not consistent

```
$A(\,) \mid \mathtt{x} \leftarrow \mathtt{Id(x)}.$
$B(\,) \mid \mathtt{x} \leftarrow \mathtt{Id(x)}.$
$T(\,) \leftarrow A(\,),\, B(\,).$
$T(\,)\bullet \leftarrow T(\,).$
```

erased the single chance of producing $T(\,)$. Hence $\mathit{output}(M) = \emptyset$, and the program does not tolerate non-causality. The proof details can be found in [20].

6.6.2 Syntactical Interpretation

Now we look at the CRON conjecture from a syntactical point of view. A Dedalus program without negation is called *positive.* Our main result now is that the following does hold:

Theorem 6.1. Every positive consistent Dedalus program tolerates non-causality.

The converse direction of Theorem 6.1, to the effect that every consistent Dedalus program tolerating non-causality is equivalent to a positive program, cannot hold by our counterexample for the if-direction of the semantical CRON conjecture (see Section 6.6.1.1).

The following subsections prove Theorem 6.1. In particular, we have to show for each positive consistent Dedalus program $\mathcal{P}$, and each input H, that every SZ-model of $\mathcal{P}$ on H produces *(i)* at least $\mathit{outInst}(\mathcal{P}, H)$ and *(ii)* at most $\mathit{outInst}(\mathcal{P}, H)$, respectively shown in Sections 6.6.2.1 and 6.6.2.2.

We remark that a positive program is not automatically consistent; Algorithm 6.3 gives a simple example, where the output $T(\,)$ can only be created when $A(\,)$ and $B(\,)$ are delivered simultaneously, which does not happen in every fair run.

6.6.2.1 At Least All Operational Outputs

Let $\mathcal{P}$ be a positive and consistent Dedalus program. Let H be an input for $\mathcal{P}$, over a network $\mathcal{N}$, and let M be an SZ-model of $\mathcal{P}$ on H. We have to show $\mathit{outInst}(\mathcal{P}, H) \subseteq \mathit{output}(M)$. We construct a fair run $\mathcal{R}$ of $\mathcal{P}$ on H such that $\mathit{output}(\mathcal{R}) \subseteq \mathit{output}(M)$. Then, since $\mathit{output}(\mathcal{R}) = \mathit{outInst}(\mathcal{P}, H)$ by consistency of $\mathcal{P}$, we have $\mathit{outInst}(\mathcal{P}, H) \subseteq \mathit{output}(M)$, as desired.

Notations We need some auxiliary notations. For each $(x, s) \in \mathcal{N} \times \mathbb{N}$, let $\mathit{all}_M(x, s)$ be the set of all facts $R(\bar{a})$ for which $R(x, s, \bar{a}) \in M|_{\mathit{sch}(\mathcal{P})}$, i.e., the set of all facts over $\mathit{sch}(\mathcal{P})$ in M at node x on timestamp s.

For each $(x, s) \in \mathcal{N} \times \mathbb{N}$, let $\mathit{rcv}_M(x, s)$ be the set of all facts $R(\bar{a})$ for which there is some y and t such that $\mathtt{chosen}_R(y, t, x, s, \bar{a}) \in M$, i.e., the set of all messages arriving at (x, s) in M. Note, $\mathit{rcv}_M(x, s) \subseteq \mathit{all}_M(x, s)$ by rules of the form (5.8) in $\mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$.

For each $x \in \mathcal{N}$, let $snd_M(x)$ be the set of all pairs $(y, R(\bar{a}))$ for which there is some s and t such that $\texttt{chosen}_R(x, s, y, t, \bar{a}) \in M$, i.e., the set of all messages (with addressee) that x ever sends in M.

We define $sndFin_M(x) \subseteq snd_M(x)$ to be the subset of pairs $(y, R(\bar{a}))$ for which there are only a finite number of times s such that $\texttt{chosen}_R(x, s, y, t, \bar{a}) \in M$ for some $t \in \mathbb{N}$, i.e., there are only a finite number of times s on which x sends $R(\bar{a})$ to y in M. Now, for each $x \in \mathcal{N}$, we define $start_M(x) = 0$ if $sndFin_M(x) = \emptyset$ and otherwise we define $start_M(x)$ to be 1 plus the largest timestamp on which x sends a pair of $sndFin_M(x)$ in M. Intuitively, $start_M(x)$ is the first local timestamp of x at which x no longer sends messages in $sndFin_M(x)$, so the messages that x sends starting from $start_M(x)$ are sent infinitely often.

Main idea We inductively define the transitions of $\mathcal{R}$. More specifically, for each $i = 0, 1, \ldots$, we define the (partial) *arrival function* $\alpha_{\mathcal{R}}^{(i)}$ that contains for each transition $j \leq i$ mappings of the form $(j, y, R(\bar{a})) \mapsto k$, where $R(\bar{a})$ is a message with addressee y sent in transition j, to say that $R(\bar{a})$ is delivered to y in transition k (with $j < k$).[2] The arrival function is merely a technical aid; it helps us make explicit how messages are delivered. We also write a mapping $(j, y, R(\bar{a})) \mapsto k$ simply as $(j, y, R(\bar{a}), k)$.

Assuming some arbitrary order on $\mathcal{N}$, consider the following (co-lexical) total order $\leq$ on $\mathcal{N} \times \mathbb{N}$:

$$(x, s) \leq (y, t) \iff s < t \text{ or } (s = t \text{ and } x \leq y).$$

For each $(x, s) \in \mathcal{N} \times \mathbb{N}$, let $ord(x, s)$ denote the ordinal of (x, s) in this total order. We define the active node in transition i of $\mathcal{R}$ to be the unique $x \in \mathcal{N}$ satisfying $ord(x, s) = i$ for some $s \in \mathbb{N}$. For each $i \in \mathbb{N}$, we write D_i, x_i and s_i to denote respectively the deductive fixpoint, active node and timestamp (of the active node) during transition i. For each $i \in \mathbb{N}$, we want the following induction properties to be satisfied, for which the intuition is provided below:

$$D_i \subseteq all_M(x_i, s_i) \tag{6.2}$$

$$\forall (j, y, \boldsymbol{f}, k) \in \alpha_{\mathcal{R}}^{(i)} : \boldsymbol{f} \in rcv_M(x_k, s_k) \tag{6.3}$$

$$\forall (j, y, \boldsymbol{f}, k) \in \alpha_{\mathcal{R}}^{(i)} : s_k \geq start_M(y) \tag{6.4}$$

Property (6.2) ensures that all ultimate facts of $\mathcal{R}$ are ultimate facts of M, resulting in $output(\mathcal{R}) \subseteq output(M)$, as desired. Property (6.3) ensures we do not have more opportunities in $\mathcal{R}$ for messages to arrive "together" when compared to M, so that induction property (6.2) can be satisfied. To explain property (6.4), note that some messages in M are sent only a finite number of times, even into the past. Such messages are the result of a coincidence, like the coincident arrival of messages, and because such messages can not be sent into the past in $\mathcal{R}$, we would have to deliver them somewhere in the future, risking a violation of induction property (6.3). Now, induction property (6.4) will ensure that we only send messages in $\mathcal{R}$ that are sent an infinite number of times in M, and this can be used to satisfy induction property (6.3).

[2]To make sure that each message is eventually delivered, all messages sent in transitions $j \leq i$ will get a mapping in $\alpha_{\mathcal{R}}^{(i)}$.

Inductive construction For uniformity, we start with $i = -1$, and define $\alpha_{\mathcal{R}}^{(-1)} = \emptyset$ and $D_{-1} = \emptyset$. So, properties (6.2) through (6.4) are trivially satisfied for $i = -1$. For the induction hypothesis, assume $\mathcal{R}$ has been partially constructed up to and including transition $i-1$, where $i \geq 0$, and assume the properties hold for all transitions $j = -1$, 0, ..., $i-1$. For the inductive step, we show that property (6.2) is satisfied for i, and we show how to extend $\alpha_{\mathcal{R}}^{(i-1)}$ to $\alpha_{\mathcal{R}}^{(i)}$ such that properties (6.3) and (6.4) are satisfied. The set m_i of (tagged) messages to be delivered in transition i consists of all pairs $(j, \boldsymbol{f})$ for which $\alpha_{\mathcal{R}}^{(i-1)}$ contains $(j, y, \boldsymbol{f}, i)$.[3] Henceforth, we will omit technical details and give only the most important steps of the proof; the omitted details are in [20].

Property (6.2) We have to show $D_i \subseteq \mathit{all}_M(x_i, s_i)$. Using the definition $D_i = \mathit{deduc}_{\mathcal{P}}(s_i(x_i) \cup \mathit{untag}(m_i))$ with $\rho_i = (s_i, b_i)$ the source-configuration of transition i, the problem can be reduced to showing $s_i(x_i) \cup \mathit{untag}(m_i) \subseteq \mathit{all}_M(x_i, s_i)$ [20]. First, by applying the induction hypothesis for property (6.3) to $\alpha_{\mathcal{R}}^{(i-1)}$, we know $\mathit{untag}(m_i) \subseteq \mathit{rcv}_M(x_i, s_i) \subseteq \mathit{all}_M(x_i, s_i)$.

We are left to show $s_i(x_i) \subseteq \mathit{all}_M(x_i, s_i)$. We distinguish between facts over $\mathit{edb}(\mathcal{P})$ and $\mathit{idb}(\mathcal{P})$. We have $s_i(x_i)|_{\mathit{edb}(\mathcal{P})} \subseteq \mathit{all}_M(x_i, s_i)$ because $s_i(x_i)|_{\mathit{edb}(\mathcal{P})} = H(x_i)$ by the operational semantics and $H(x_i)^{\Uparrow x_i, s_i} \subseteq \mathit{decl}(H) \subseteq M$ by definition of M. Next, if i is the first transition of x_i, we have $s_i(x_i)|_{\mathit{idb}(\mathcal{P})} = \emptyset \subseteq \mathit{all}_M(x_i, s_i)$. Otherwise, we consider the last transition j before i in which x_i was also the active node. By the operational semantics, $s_i(x_i)|_{\mathit{idb}(\mathcal{P})} = \mathit{induc}_{\mathcal{P}}\langle D_j\rangle$. Using $D_j \subseteq \mathit{all}_M(x_i, s_j)$ by the induction hypothesis for property (6.2), we can show that $\mathit{induc}_{\mathcal{P}}\langle D_j\rangle \subseteq \mathit{all}_M(x_i, s_j + 1) = \mathit{all}_M(x_i, s_i)$, as desired [20].

Properties (6.3) and (6.4) We have to extend $\alpha_{\mathcal{R}}^{(i-1)}$ to $\alpha_{\mathcal{R}}^{(i)}$ so that properties (6.3) and (6.4) are satisfied. Suppose transition i sends a message $R(\bar{a})$ to an addressee $y \in \mathcal{N}$. We have to choose a transition k with $i < k$ in which to deliver $R(\bar{a})$ to y. We start by showing there are an infinite number of timestamps s on which x_i sends $R(\bar{a})$ to y in M. We differentiate between two cases.

First, suppose $s_i < \mathit{start}_M(x_i)$. The induction hypothesis for property (6.4) implies x_i has only done heartbeats up to and including transition i, i.e., no messages have been delivered to x_i yet. Then it is intuitively clear that node x_i sends $R(\bar{a})$ to y on an infinite number of timestamps in M; indeed, by positivity of $\mathcal{P}$, if x_i can start sending messages by using just local facts, then after a while these messages are generated continuously [20].

Now suppose $s_i \geq \mathit{start}_M(x_i)$. Using $D_i \subseteq \mathit{all}_M(x_i, s_i)$ (see above), $R(y, \bar{a}) \in \mathit{async}_{\mathcal{P}}\langle D_i\rangle$, and $y \in \mathcal{N}$, we can show there is a local timestamp t of y for which $\mathtt{chosen}_R(x_i, s_i, y, t, \bar{a}) \in M$ [20]. So, in M, node x_i sends $R(\bar{a})$ to y on a timestamp at least $\mathit{start}_M(x_i)$, which by definition of $\mathit{start}_M(x_i)$ implies that node x_i sends $R(\bar{a})$ to y on an infinite number of timestamps in M.

Now, because x_i sends $R(\bar{a})$ to y on an infinite number of timestamps in M, and y receives only a finite number of messages on each timestamp (enforced in Section 6.5.1.1), there must be an infinite number of timestamps $t \in \mathbb{N}$ on which y receives $R(\bar{a})$ in M. Among these, we can surely choose some arrival timestamp $t \in \mathbb{N}$ for

[3]This implies $y = x_i$.

which $ord(y,t) > i$ and $t \geq start_M(y)$. Then we extend $\alpha_{\mathcal{R}}^{(i-1)}$ by adding the mapping $(i, y, R(\bar{a}), k)$ where $k = ord(y,t)$. Note, this mapping satisfies properties (6.3) and (6.4).

6.6.2.2 No Wrong Outputs

Let $\mathcal{P}$ be a positive and consistent Dedalus program. Let H be an input for $\mathcal{P}$, and let M be an SZ-model of $\mathcal{P}$ on H. We have to show $output(M) \subseteq outInst(\mathcal{P}, H)$. We construct a fair run $\mathcal{R}$ such that $output(M) \subseteq output(\mathcal{R})$. Then, using $output(\mathcal{R}) = outInst(\mathcal{P}, H)$ by consistency of $\mathcal{P}$, we get $output(M) \subseteq outInst(\mathcal{P}, H)$, as desired.

Run $\mathcal{R}$ proceeds in rounds: in each round we let each node become active precisely once to receive its entire buffer at the beginning of the round. Messages sent in each round are accumulated and are delivered only during the next round. The number of rounds is infinite. Because $\mathcal{P}$ is positive, the programs $deduc_{\mathcal{P}}$, $induc_{\mathcal{P}}$, and $async_{\mathcal{P}}$ are monotone. Then, since always the entire buffer is delivered to each node, the sets of deductively derived facts monotonically increase on each node.

For each transition i of $\mathcal{R}$, let D_i denote the output of $deduc_{\mathcal{P}}$ during i. For each fact $R(x, s, \bar{a}) \in M|_{sch(\mathcal{P})}$ we show there is a transition i of x in $\mathcal{R}$ with $R(\bar{a}) \in D_i$. This gives $output(M) \subseteq output(\mathcal{R})$ because for each ultimate fact $R(\bar{a})$ in M at some node x, surely $R(x, s, \bar{a}) \in M$ for some $s \in \mathbb{N}$, and so if $R(\bar{a}) \in D_i$ for some transition i of x then $R(\bar{a}) \in D_j$ for all subsequent transitions j of x by the monotonous nature of $\mathcal{R}$.

Abbreviate $G_M(\mathcal{P}) = ground_M(\mathcal{P}', I)$ where $\mathcal{P}' = pure_{\mathrm{SZ}}(\mathcal{P})$ and $I = decl(H)$. Because $M = G_M(\mathcal{P})(I)$ by definition of stable model, we can consider the infinite sequence M_0, M_1, M_2, …, such that $M = \bigcup_l M_l$; $M_0 = I$; and, for each $l \geq 1$ the instance M_l is obtained from M_{l-1} by applying the immediate consequence operator of $G_M(\mathcal{P})$. This implies $M_{l-1} \subseteq M_l$ for each $l \geq 1$. By induction on l, we show that for each $R(x, s, \bar{a}) \in M_l|_{sch(\mathcal{P})}$ there is a transition i of x in $\mathcal{R}$ with $R(\bar{a}) \in D_i$.

For the base case, $R(x, s, \bar{a}) \in M_0|_{sch(\mathcal{P})}$ implies $R(\bar{a}) \in H(x)$. Then $R(\bar{a}) \in D_i$ for any transition i of x because each state of x contains $H(x)$ by the operational semantics. For the induction hypothesis, assume the property holds for M_{l-1} where $l \geq 1$. Now, let $R(x, s, \bar{a}) \in M_l|_{sch(\mathcal{P})} \setminus M_{l-1}$. Let $\psi \in G_M(\mathcal{P})$ be a ground rule responsible for deriving this fact, i.e., $pos_\psi \subseteq M_{l-1}$ and $head_\psi = R(x, s, \bar{a})$. Rule ψ must have one of the following three forms: the deductive form (5.3), the inductive form (5.4), or the delivery form (5.8). We handle each case in turn.

Deductive Let $\varphi \in pure_{\mathrm{SZ}}(\mathcal{P})$ be the rule corresponding to ψ, so φ is of the form (5.3). Let V be the valuation for φ such that ψ results from applying V to φ. In turn, let $\varphi' \in \mathcal{P}$ be the original deductive rule on which φ is based. Note, $\varphi' \in deduc_{\mathcal{P}}$. By the syntactical correspondence between φ and φ', we can apply V to φ'. Now, it suffices to show $V(pos_{\varphi'}) \subseteq D_i$ for some transition i of x in $\mathcal{R}$, resulting in $V(head_{\varphi'}) = R(\bar{a}) \in D_i$ by the fixpoint semantics of $deduc_{\mathcal{P}}$, as desired. So, let $S(\bar{b}) \in V(pos_{\varphi'})$. By the syntactical correspondence between φ' and φ, we have $S(x, s, \bar{b}) \in V(pos_\varphi) = pos_\psi$. Using $pos_\psi \subseteq M_{l-1}$ gives $S(x, s, \bar{b}) \in M_{l-1}|_{sch(\mathcal{P})}$. Then the induction hypothesis implies there is a transition j of x in $\mathcal{R}$ satisfying $S(\bar{b}) \in D_j$. And because deductive facts monotonously grow at x in $\mathcal{R}$, there is a transition i of x such that $S(\bar{b}) \in D_i$ for each $S(\bar{b}) \in V(pos_{\varphi'})$.

Inductive Let φ and V be like in the deductive case, but now φ is of the form (5.4). Let $\varphi' \in \mathit{induc}_{\mathcal{P}}$ be the rule corresponding to φ. Again, we can apply V to φ'. Again, it suffices to show $V(\mathit{pos}_{\varphi'}) \subseteq D_i$ for some transition i of x in $\mathcal{R}$, causing $V(\mathit{head}_{\varphi'}) = R(\bar{a})$ to be stored in the next state of x. Then, with j being the first transition of x after i, we get $R(\bar{a}) \in D_j$ by the operational semantics, as desired. The existence of i is established like in the deductive case.

Delivery Rule ψ is of the form (5.8), with body fact $\texttt{chosen}_R(y, t, x, s, \bar{a}) \in M_{l-1}$. We show there is a transition i of y in $\mathcal{R}$, in which y sends $R(\bar{a})$ to x. Then, in the next round of $\mathcal{R}$ following i, we deliver $R(\bar{a})$ to x in some transition j. Then $R(\bar{a}) \in D_j$ by the operational semantics, as desired.

Now, $\texttt{chosen}_R(y, t, x, s, \bar{a}) \in M_{l-1}$ implies $\texttt{cand}_R(y, t, x, s, \bar{a}) \in M_{l-1}$. There is some $k \in \mathbb{N}$ with $0 < k < l-1$ such that $\texttt{cand}_R(y, t, x, s, \bar{a}) \in M_k \setminus M_{k-1}$. Let $\psi' \in G_M(\mathcal{P})$ be a rule responsible for deriving the $\texttt{cand}_R$-fact. Let $\varphi' \in \mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$ be the rule corresponding to ψ', and let V' be the valuation for φ' giving rise to ψ'. In turn, let $\varphi'' \in \mathit{async}_{\mathcal{P}}$ be the rule corresponding to φ'. By the syntactical correspondence between φ' and φ'', we can apply V' to φ''. Note, $V'(\mathit{head}_{\varphi''}) = R(x, \bar{a})$. To make y send $R(\bar{a})$ to x in some transition i, we need $V'(\mathit{pos}_{\varphi''}) \subseteq D_i$. The existence of transition i is again established like in the deductive case.

6.7 Expressivity

We now discuss the expressivity of Dedalus. Let $\mathcal{Q}$ be a query. Like in Section 6.6.1, we say that a Dedalus program $\mathcal{P}$, with $\mathit{edb}(\mathcal{P})$ the input schema of $\mathcal{Q}$, *computes* $\mathcal{Q}$ if $\mathcal{P}$ is consistent and for every input instance I for $\mathcal{Q}$, for every network $\mathcal{N}$, for every partition H of I over $\mathcal{N}$, we have $\mathit{outInst}(\mathcal{P}, H) = \mathcal{Q}(I)$. We now argue that Dedalus captures the queries expressible in the language While [2].

6.7.1 Upper Bound

First we argue the upper bound. Let $\mathcal{P}$ be a Dedalus program that computes a query $\mathcal{Q}$. We immediately focus on a single-node network $\mathcal{N}$, where, by assumption, $\mathcal{P}$ also correctly computes $\mathcal{Q}$. Let I be an input for $\mathcal{Q}$. The single node of $\mathcal{N}$ is given the entire instance I. Now, consider the run $\mathcal{R}$ in which we deliver the entire message buffer in each transition. By consistency of $\mathcal{P}$, run $\mathcal{R}$ produces $\mathcal{Q}(I)$. Note, because only a set of messages is sent in each transition, the message buffer degenerates to a set in $\mathcal{R}$. So, the message buffer could just be simulated with inductive rules, giving a modified program $\mathcal{P}'$ that only works correctly on single-node networks.

Next, we can simulate the behavior of $\mathcal{P}'$ by a single-node *transducer* network, whose single transducer Π has its queries implemented with Datalog$^\neg$ under the stratified semantics: each inductively computed relation R of $\mathcal{P}'$ becomes a memory relation of Π, whose insertion query contains $\mathit{deduc}_{\mathcal{P}'}$ and the inductive rules for R (these determine the answer), and whose deletion query always deletes the previous contents of R.[4] There are no message relations. Note, a stratified Datalog$^\neg$ program can be simulated by a sequence of fixpoint-loops in the language While (one fixpoint

[4]This strategy always keeps precisely the inductively derived facts.

for each stratum), and hence by a single While-program. So, Π can be regarded as a While-transducer. Then, using the technique for the only-if direction in Lemma 3.25, transducer Π can be simulated by a transducer Π' whose queries are implemented in FO, still on a single-node network. Lastly, applying the if-direction of Lemma 3.25 to Π' gives that $\mathcal{Q}$ is in While, as desired.

6.7.2 Lower Bound

Now we argue the lower bound. Let $\mathcal{Q}$ be a query expressible with a While-program P. We may assume that P consists of a single while-loop (with no nested while-loops) after which a final sequence of FO-statements sets the contents of the output relations; P can always be rewritten into this form. We construct a Dedalus program $\mathcal{P}$ to compute $\mathcal{Q}$ as follows. First, we implement in $\mathcal{P}$ the protocol from Lemma 3.8 to let each node accumulate all inputs across the network. At each node, the end of this protocol is signaled by the derivation of a nullary fact `ready( )`, that we persist by inductive rules. After obtaining `ready( )`, every node acts as if it was alone on the network, and it simulates program P as follows. One iteration of the while-loop is simulated with one local step of $\mathcal{P}$, where the deductive rules simulate the FO-expressions of P, and the inductive rules simulate the changes to the temporary relations of P. Once the (simulated) condition of the while-loop in P becomes false, $\mathcal{P}$ follows the specification of P to set the final contents of the output relations, that are persisted with inductive rules.

Chapter 7

Conclusion

Below, we give conclusions for our work and mention possible future work.

Relational transducers for declarative networking In Chapter 3, we have tried to formalize and prove the CALM Conjecture of Hellerstein [36, 37]. Although we could not confirm the original formulation that mentions Datalog, we have shown a more general correspondence between coordination-freeness and monotonicity of distributed computations. We have also identified a more syntactical counterpart to these notions, namely, obliviousness. Our approach uses the relational transducer model, grounding the results firmly in previous database theory practice. An expressivity analysis also demonstrates that the transducer model is quite natural: basically, it only introduces a notion of iteration to the local query language of the transducers.

In future work, it might be interesting to consider other notions of coordination-freeness, like a variant where little communication is still allowed before outputting the result. It might also be interesting to quantify the amount of coordination (see e.g. [12]). Lastly, it might be useful to design data initialization strategies for a network to reduce the need for communication, and hence more easily obtain coordination-freeness in the way we have defined it (see e.g. [59]).

Deciding eventual consistency In Chapter 4, we have formalized eventual consistency as *confluence* (with the opposite being *diffluence*). We have shown decidability in NEXPTIME of diffluence for relational transducer networks implemented with unions of conjunctive queries with negation and that are "simple" (for lack of a better name). The problem turns out to be complete for NEXPTIME. These simple transducer networks satisfy five restrictions: recursion-freeness, inflationarity, message-positivity, static message sending, and message-boundedness. We have also shown that simple transducer networks capture a natural class of distributed queries based on unions of conjunctive queries with negation.

As already mentioned in the Introduction (Section 1.2), a topic for further work is to investigate whether decidability can be retained while (slightly) relaxing the restrictions of recursion-freeness, inflationarity, and message-positivity. Also, we have only considered concrete transducer networks, i.e., networks with a particular set of nodes. It might be interesting to decide if for a given transducer Π, all transducer networks are confluent where Π is replicated on all nodes. This is related to the notion of network-independence from Chapter 3. It might also be interesting to transfer the restrictions proposed in Chapter 4 to other rule-based languages, like Dedalus, and also try to transfer the decidability result.

Regarding expressivity, the techniques of the upper bound can transform a given confluent simple transducer network to a query-description in $\mathrm{UCQ}^{\neg}$. When the techniques of the lower bound are applied to this query-description, we obtain a simple transducer network that does not use memory relations anymore, but still expresses the same query as the original network. This could be considered as some normal form. It might be interesting to describe the smallest size that the normal form could have in relationship to the original network. Perhaps in some situations this could demonstrate a pragmatic benefit in using memory relations, namely, because otherwise there is an exponential blowup in the transducer descriptions.

There seem to be several reasonable ways to formalize the intuitive notion of eventual consistency. In contrast to the confluence formalization of Chapter 4 (with finite runs), a stronger view of eventual consistency is to require that on every input, all infinite "fair" runs produce the same set of output facts, as in Chapter 3 and [1]. When a transducer network is eventually consistent in this stronger sense, it is also in the confluence sense, but the other implication is not obvious. Indeed, our confluence interpretation of eventual consistency only guarantees that outputs can still be produced when messages are delivered in the "right" way. For example, we might have to deliver two messages simultaneously. But this might never happen in some particular fair run. It deserves further research to better understand the relationship between eventual consistency and fairness requirements. In this context, it might be possible to consider other fairness conditions, besides the ones we considered.

There also seems to be a pragmatic lesson in Chapter 4: although eventual consistency is an interesting property to guarantee for a network, the cost of automatically deciding it might be too high. Indeed, we have to severely restrict the expressiveness of the language and still the resulting decision problem has high intrinsic complexity. For this reason, other approaches might be more viable, such as providing sufficient syntactic guarantees on eventual consistency without unduly limiting the expressive power (e.g. [47]) and without strongly increasing the distributed coordination (e.g. [59], and the notion of "obliviousness" from Chapter 3).

Declarative semantics for Dedalus In Chapter 5, we have shown for the language Dedalus that distributed computations expressed in an operational way can also be described purely declaratively, using stable models. We believe this could provide an interesting alternative viewpoint on distributed computations.

Regarding future work, we have probably not yet explored the full power of stable models. We therefore expect that our result can be extended to languages incorporating more powerful constructs, such as dynamic choice [40], aggregation [45], or constructs that allow for reasoning about different time-scales on which events occur [34]. It might also be possible to remove the syntactic stratification condition on the deductive rules of Dedalus.

More related to multi-agent systems [43, 50, 42], it might be interesting to allow logic programs used in declarative networking to dynamically modify their rules. The question would be how (and if) this can be represented in a declarative semantics.

Lastly, we can think about the output of Dedalus programs. Marczak et al. [46] define the output of Dedalus programs with *ultimate* facts, which are facts that will eventually always be present on the network. This was also used in Chapter 6, where the output of the Dedalus operational semantics (and stable models) was defined. Then, a *consistent* Dedalus program is required to produce, for each input individually, the same output in every run. For consistent programs, for each input, the corresponding output can thus be defined as the output of any run. Then it might be interesting to find finite representations of the stable models for each input, for example by only keeping the ultimate facts.[1] This could serve as a more intuitive programmer abstraction, or it could perhaps be used to more efficiently simulate the behavior of the network for testing purposes.

Finding the ultimate facts of a program for a certain input can be reduced to the following output decision problem: for a consistent Dedalus program, an input for that program, and a fact, decide if this fact is output by the program on that input. But because there is no bound on the message buffers of Dedalus programs (Section 2.8.2), we are dealing with an infinite state system like in Chapter 4. So, we expect that this problem can not be solved in general. But it might be interesting to find solutions in particular (syntactically defined) cases.

The CRON conjecture In Chapter 6, we have tried to formalize and prove the CRON conjecture of Hellerstein [36, 37] in the context of Dedalus. We have explored a semantical formalization with database queries, as we did for the CALM conjecture in Chapter 3, but it was refuted by counterexamples. Yet, we have been able to rescue some intuition of the original conjecture, and have shown that positive Dedalus programs do not require causal message delivery.

In future work, the spectrum of causality needs to be better understood. Generalizing positive programs, perhaps richer classes of programs can tolerate some relaxations of causality as well. Conversely, perhaps for some classes of programs (or problems) causality is always strictly needed. Lastly, the CRON conjecture could be more concretely linked to crash recovery applications, and the design of recovery mechanisms.

[1]This gives a finite set because the input-domain is finite and Dedalus programs do not create new values.

Bibliography

[1] S. Abiteboul, M. Bienvenu, A. Galland, et al. A rule-based language for Web data management. In *Proceedings 30th ACM Symposium on Principles of Database Systems*, pages 293–304. ACM Press, 2011.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[3] S. Abiteboul and E. Simon. Fundamental properties of deterministic and nondeterministic extensions of Datalog. *Theoretical Computer Science*, 78:137–158, 1991.

[4] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[5] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings 23rd ACM Symposium on the Theory of Computing*, pages 209–219, 1991.

[6] S. Abiteboul and V. Vianu. Computing with first-order logic. *Journal of Computer and System Sciences*, 50(2):309–335, 1995.

[7] S. Abiteboul, V. Vianu, et al. Relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 61(2):236–269, 2000.

[8] F.N. Afrati, S.C. Cosmadakis, and M. Yannakakis. On Datalog vs polynomial time. *Journal of Computer and System Sciences*, 51(2):177–196, 1995.

[9] J.J. Alferes, L.M. Pereira, H. Przymusinska, and T.C. Przymusinski. LUPS—a language for updating logic programs. *Artificial Intelligence*, 138(1–2):87–116, 2002.

[10] P. Alvaro, T.J. Ameloot, J.M. Hellerstein, W.R. Marczak, and J. Van den Bussche. A declarative semantics for Dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley, Nov 2011.

[11] P. Alvaro, T.J. Ameloot, J.M. Hellerstein, W.R. Marczak, and J. Van den Bussche. A declarative semantics for Dedalus. Hasselt University, Technical report, `http://hdl.handle.net/1942/14572`, 2013.

[12] P. Alvaro, N. Conway, J. Hellerstein, and W.R. Marczak. Consistency analysis in Bloom: A CALM and collected approach. In *Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260. www.cidrdb.org, 2011.

[13] P. Alvaro, W.R. Marczak, et al. Dedalus: Datalog in time and space. Technical Report EECS-2009-173, University of California, Berkeley, 2009.

[14] P. Alvaro, W.R. Marczak, et al. Dedalus: Datalog in time and space. In de Moor et al. [27], pages 262–281.

[15] T.J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *Proceedings 30th ACM Symposium on Principles of Database Systems*, pages 283–292. ACM Press, 2011.

[16] T.J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. Hasselt University, Technical report, `http://hdl.handle.net/1942/14570`, 2013.

[17] T.J. Ameloot and J. Van den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. In *Proceedings of the 15th International Conference on Database Theory*, pages 86–98. ACM Press, 2012.

[18] T.J. Ameloot and J. Van den Bussche. On the CRON conjecture. In Barceló and Pichler [24], pages 44–55.

[19] T.J. Ameloot and J. Van den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. Hasselt University, Technical report, `http://hdl.handle.net/1942/14571`, 2013.

[20] T.J. Ameloot and J. Van den Bussche. On the CRON conjecture. Hasselt University, Technical report, `http://hdl.handle.net/1942/14567`, 2013.

[21] K.R. Apt and R.N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19-20, Supplement 1(0):9–71, 1994.

[22] K.R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.

[23] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, 2004.

[24] P. Barceló and R. Pichler, editors. *Datalog in Academia and Industry*, volume 7494 of *Lecture Notes in Computer Science*. Springer, 2012.

[25] A. Blass, Y. Gurevich, and J. Van den Bussche. Abstract state machines and computationally complete query languages. *Information and Computation*, 174(1):20–36, 2002.

[26] A.K. Chandra and M.Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

[27] O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors. *Datalog Reloaded: First International Workshop, Datalog 2010*, volume 6702 of *Lecture Notes in Computer Science*, 2011.

[28] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *Proceedings 12th International Conference on Database Theory*, 2009.

[29] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven Web applications. *Journal of Computer and System Sciences*, 73(3):442–474, 2007.

[30] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven Web services. In *Proceedings 25th ACM Symposium on Principles of Database Systems*, pages 90–99. ACM Press, 2006.

[31] N. Francez. *Fairness.* Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[32] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[33] J. Gray. Notes on data base operating systems. In M.J. Flynn et al., editors, *Operating Systems—An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.

[34] G. Greco, A. Guzzo, D. Saccà, and F. Scarcello. Event choice datalog: a logic programming language for reasoning in multiple dimensions. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP, pages 238–249. ACM Press, 2004.

[35] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In M. Carro and R. Peña, editors, *Proceedings 12th International Symposium on Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 88–103, 2010.

[36] J.M. Hellerstein. Datalog redux: experience and conjecture. Video available (under the title "The Declarative Imperative") from `http://db.cs.berkeley.edu/jmh/`, 2010. PODS 2010 keynote.

[37] J.M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[38] S.S. Huang, T.J. Green, and B.T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on the Management of Data*, SIGMOD '11, pages 1213–1216. ACM, 2011.

[39] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP, pages 106–115. IEEE Computer Society, 2001.

[40] R Krishnamurthy and S.A. Naqvi. Non-deterministic choice in Datalog. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pages 416–424, 1988.

[41] L. Lamport. Fairness and hyperfairness. *Distributed Computing*, 13:239–245, November 2000.

[42] J. Leite and L. Soares. Adding evolving abilities to a multi-agent system. In *Proceedings of the 7th International Conference on Computational Logic in Multi-agent Systems*, CLIMA VII'06, pages 246–265. Springer-Verlag, 2007.

[43] J.A. Leite, J.J. Alferes, and L.M. Pereira. Minerva – a dynamic logic programming agent architecture. In *Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, ATAL, pages 141–157. Springer-Verlag, 2002.

[44] B.T. Loo, T. Condie, et al. Declarative networking: language, execution and optimization. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *SIGMOD Conference*, pages 97–108. ACM, 2006.

[45] B.T. Loo et al. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[46] W.R. Marczak, P. Alvaro, N. Conway, J.M. Hellerstein, and D. Maier. Confluence analysis for distributed programs: A model-theoretic approach. Technical Report UCB/EECS-2011-154, EECS Department, University of California, Berkeley, Dec 2011.

[47] W.R. Marczak, P. Alvaro, N. Conway, J.M. Hellerstein, and D. Maier. Confluence analysis for distributed programs: A model-theoretic approach. In Barceló and Pichler [24], pages 135–147.

[48] J.A. Navarro and A. Rybalchenko. Operational semantics for declarative networking. In A. Gill and T. Swift, editors, *Proceedings 11th International Symposium on Practical Aspects of Declarative Languages*, volume 5419 of *Lecture Notes in Computer Science*, pages 76–90, 2009.

[49] V. Nigam, L. Jia, B.T. Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. *Computer Languages, Systems & Structures*, 38(2):158–180, 2012.

[50] V. Nigam and J. Leite. A dynamic logic programming based system for agents with declarative goals. In *Proceedings of the 4th International Conference on Declarative Agent Languages and Technologies*, DALT, pages 174–190. Springer-Verlag, 2006.

[51] E.L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[52] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 205–217. ACM Press, 1990.

[53] M. Sipser. *Introduction to the Theory of Computation, Second Edition, International Edition.* Thomson Course Technology, Boston, Massachusetss, USA, 2006.

[54] M. Spielmann. Verification of relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 66(1):40–65, 2003.

[55] J. Van den Bussche and L. Cabibbo. Converting untyped formulas into typed ones. *Acta Informatica*, 35(8):637–643, 1998.

[56] M. Vardi. The complexity of relational query languages. In *Proceedings 14th ACM Symposium on the Theory of Computing*, pages 137–146, 1982.

[57] W. Vogels. Eventual consistency. *Communications of the ACM*, 52(1):40–44, 2009.

[58] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010.

[59] D. Zinn, T.J. Green, and B. Ludaescher. Win-move is coordination-free. In *Proceedings of the 15th International Conference on Database Theory*, pages 99–113. ACM Press, 2012.