Theory Comput Syst
DOI 10.1007/s00224-015-9624-6

Deciding Confluence for a Simple Class of Relational Transducer Networks

Tom J. Ameloot[1] · **Jan Van den Bussche**[1]

Abstract Networks of relational transducers can serve as a formal model for declarative networking, focusing on distributed database querying applications. In declarative networking, a crucial property is eventual consistency, meaning that the final output does not depend on the message delays and reorderings caused by the network. Here, we formalize eventual consistency as a confluence notion, meaning that finite executions of the system can always be extended to yield the desired output. We show that confluence is decidable when the transducers satisfy some syntactic restrictions, some of which have also been considered in earlier work on automated verification of relational transducers. This simple class of transducer networks computes exactly all distributed queries expressible by unions of conjunctive queries with negation.

Keywords Relational transducer · Eventual consistency · Confluence · Decidability · Conjunctive query

T. J. Ameloot is a Postdoctoral Fellow of the Research Foundation – Flanders.

✉ Tom J. Ameloot
tom.ameloot@uhasselt.be

Jan Van den Bussche
jan.vandenbussche@uhasselt.be

[1] Hasselt University and Transnational University of Limburg, Hasselt, Belgium

Published online: 15 May 2015

1 Introduction

Declarative networking [18] is an approach by which distributed computations and networking protocols, as occurring in cloud computing, are modeled and programmed using formalisms based on Datalog. Recently, declarative networking formalisms are enjoying attention from the database theory community, so that now a number of models and languages are available with a formally defined semantics and initial investigations on their expressive power [1, 5, 8, 15, 20].

A major hurdle in using declarative methods for cloud computing is the nondeterminism inherent to such systems. This nondeterminism is typically due to the asynchronous communication between the compute nodes in a cluster or 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* [4, 16, 17, 24]. Of course, eventual consistency is undecidable in general, and there is much recent interest in finding ways to guarantee it [1, 4].

In the present paper, we 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).

In this paper, we consider clusters of compute nodes modeled as *relational transducers*, an established formal model for data-centric agents [3, 12–14, 23]. In particular, we consider relational transducers where the rules used by the nodes to send messages, to update their state relations, and to produce output, are unions of conjunctive queries with negation. This setting yields a clear model of declarative networking, given the affinity between conjunctive queries and Datalog. We thus believe our results also apply to other declarative networking formalisms, although in this paper 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 [23] and was investigated further by Deutsch et al. [13, 14].
– Finally, the message-sending rules must be "static" in the sense that they cannot 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 our paper 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.

Related Work The work most closely related to ours is that by Deutsch et al. on verification of communicating data-driven Web services [14]. 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 [17] 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 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 paper extends our conference paper [9] by detailing all proofs, and by fully characterizing the computational complexity of the decision problem. There is also follow-up work investigating another formalization of eventual consistency [6]; more discussion is provided in Section 8.

Organization We start in Section 2 by giving preliminaries about common database constructs, relational transducers, and their networks. Section 3 formalizes confluence for networks, along with syntactic restrictions leading to so-called "simple" networks; related (un)decidability results are also presented. Section 4 shows that

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

confluence of a simple network with multiple nodes can be reduced to confluence of a simple *single-node* network. Next, Section 5 establishes a small model property for simple single-node networks. This is used in Section 6 to give a procedure for deciding whether a simple single-node network is not confluent, along with a NEXPTIME-completeness result for the complexity.

The expressiveness of simple networks, not necessarily single-node, is analyzed in Section 7. We conclude in Section 8.

2 Preliminaries

2.1 Database Concepts

We first recall some basic notions from database theory [2]. A *database schema* is a finite set $\mathcal{D}$ of pairs (R, k) where R is a *relation name* and $k \in \mathbb{N}$ is the associated *arity* of R. A relation name is allowed to occur only once in a database schema. We often write a pair $(R, k) \in \mathcal{D}$ as $R^{(k)}$. An arity of zero is also called *nullary*.

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—also called *predicate* – and $\bar{a}$ is a tuple of values over **dom**. A *database instance* I *over a database schema* $\mathcal{D}$ is a finite set of facts such that for each $R(a_1, \ldots, a_k) \in I$ we have $R^{(k)} \in \mathcal{D}$. Let Z be a subset of relation names in $\mathcal{D}$. We write $I|_Z$ to denote the restriction of I to the facts whose predicate is a relation name in Z. For a function h : **dom** $\rightarrow$ **dom** we define $h(I) = \{R(h(a_1), \ldots, h(a_k)) | R(a_1, \ldots, a_k) \in I\}$. The *active domain* of I, denoted $adom(I) \subseteq$ **dom**, is the set of atomic data values that occur in I. We also use this notation for facts.

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.

2.2 Multisets

A *multiset* m over a universe $\mathcal{U}$ is a function that maps each element e of $\mathcal{U}$ 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 write *set(m)* to denote the collapse of m to a set in which we put only the elements of $\mathcal{U}$ with multiplicity at least 1. Lastly, when m is given by a more complicated expression, we will write *num(e, m)* to denote the count of e in m.

2.3 Unions of Conjunctive Queries

We now recall the query language *unions of conjunctive queries with (safe) negation*, abbreviated 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.

We assume an infinite universe **var** of variables. We will use typewriter font for variables. An *atom* is of the form $R(\mathtt{u}_1, \ldots, \mathtt{u}_k)$ where $\mathtt{u}_i \in \mathbf{var}$ for each $i \in \{1, \ldots, k\}$. A *literal* is an atom, or an atom with "¬" prepended; these literals are respectively called *positive* and *negative*.

A *conjunctive query* (or simply *rule*) φ is a four-tuple $(head^\varphi, pos^\varphi, neg^\varphi, non^\varphi)$ where $head^\varphi$ is an atom, and pos^φ and neg^φ are sets of atoms, and non^φ is a set of nonequalities of the form $(\mathtt{u} \neq \mathtt{v})$ with $\mathtt{u}, \mathtt{v} \in \mathbf{var}$. Note that neg^φ is a set of atoms, and not negative literals. We call $head^\varphi$, pos^φ, and neg^φ respectively the "head atom", the "positive body atoms", and the "negative body atoms". Let $var(\varphi)$ denote all variables that occur in φ. Let $free(\varphi)$ denote all free variables of φ (occurring in the head), and let us abbreviate $bound(\varphi) = var(\varphi) \setminus free(\varphi)$. Bound variables can be thought of as being existentially quantified. As a safety restriction, we require that all variables of $head^\varphi$, $bneg^\varphi$ and non^φ occur in pos^φ. Note that nonequalities can be simulated by applying negation to an equality relation $=$ that would have to be provided in every context where the rule is used, but for technical convenience we will immediately consider $\neq$ to be a primitive in our language.

A rule φ may be written in the conventional syntax. For example, if $head^\varphi = T(\mathtt{u}, \mathtt{v})$, $pos^\varphi = \{R(\mathtt{u}, \mathtt{v})\}$, $neg^\varphi = \{S(\mathtt{v})\}$, and $non^\varphi = \{\mathtt{u} \neq \mathtt{v}\}$, then we may write φ as

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

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

A rule φ is said to be *over* a database schema $\mathcal{D}$ if for each atom $R(\mathtt{u}_1, \ldots, \mathtt{u}_k) \in \{head^\varphi\} \cup pos^\varphi \cup neg^\varphi$ we have $R^{(k)} \in \mathcal{D}$. A *valuation for* φ is a total function $V : var(\varphi) \to \mathbf{dom}$. The *application* of V to an atom $R(\mathtt{u}_1, \ldots, \mathtt{u}_k)$ of φ, denoted $V(R(\mathtt{u}_1, \ldots, \mathtt{u}_k))$, results in the *fact* $R(a_1, \ldots, a_k)$ with $a_i = V(\mathtt{u}_i)$ for each $i \in \{1, \ldots, k\}$. We will also use this notation for applying V to a set of atoms, which results in a set of facts. Let I be a database instance over $\mathcal{D}$. The valuation V is said to be *satisfying for* φ *on* I if $V(pos^\varphi) \subseteq I$, $V(neg^\varphi) \cap I = \emptyset$, and $V(\mathtt{u}) \neq V(\mathtt{v})$ for each $(\mathtt{u} \neq \mathtt{v}) \in non^\varphi$. In that case, φ is said to *derive* the fact $V(head^\varphi)$. The *result of* φ *applied to* I, denoted $\varphi(I)$, is defined as the set of facts derived by all possible satisfying valuations for φ on I. Note that rules can only define generic queries.

A *union of conjunctive queries* is a finite set Φ of conjunctive queries that all have the same predicate and arity for the head atom. The resulting language is denoted as 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 $\bigcup_{\varphi \in \Phi} \varphi(I)$. If $\Phi = \emptyset$ then always $\Phi(I) = \emptyset$.

2.4 Distributed Databases and Queries

We now formalize how input data is distributed across a network and define a notion of queries over this data. A *network* $\mathcal{N}$ is a finite, nonempty set of *nodes*, which are values in **dom**. 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 $x \in \mathcal{N}$ to an ordinary database schema. A *distributed database instance H over schema* $\mathcal{E}$ is a function that assigns to each node $x \in \mathcal{N}$ an ordinary database instance over the local schema $\eta(x)$.

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 instances over $\mathcal{E}$ to instances over $\mathcal{F}$.

2.5 Transducers

The computation on a single node of a network is formalized by means of relational transducers [3, 8, 12–14, 23, 25]. First, 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 $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$.

An *relational transducer* Π over Υ is a collection of queries, where each query has the input schema $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{msg}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$:

- 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 $\{R^{(k+1)}\}$;

These queries will form the internal mechanism that a node uses to update its local storage and to send messages. The reason for the incremented arity in the message queries is that the extra component will be used to indicate the addressee, as will be explained in the next section.

Let Π be a transducer over schema Υ. 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, I_{rcv} is an instance over Υ_{msg} and J_{snd} is an instance over Υ_{msg} but where each fact has one extra component, such that (denoting $I' = I \cup I_{\text{rcv}}$):

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

where, following the presentation in [25],

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

Intuitively, on the receipt of 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 [23]. Output facts can not be removed. Note that local transitions are deterministic in the following sense: if $I, I_{\text{rcv}} \to J, J_{\text{snd}}$ and $I, I_{\text{rcv}} \to J', J'_{\text{snd}}$ then $J = J'$ and $J_{\text{snd}} = J'_{\text{snd}}$.

For the current paper, we immediately restrict attention to transducers whose queries are specified with UCQ$^\neg$. This results in a rule-based formalism to express the computations, following the idea behind declarative networking [18].

2.6 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. In other cases, we want to explain all facts that are recursively needed by the satisfying valuation, i.e., the facts $V(pos^\varphi)$. 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 $(nodes^\mathcal{T}, edges^\mathcal{T}, rule^\mathcal{T}, val^\mathcal{T}, lit^\mathcal{T})$ where

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

subject to the additional constraints:

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

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

From $nodes^{\mathcal{T}}$ and $edges^{\mathcal{T}}$ we can always uniquely identify the root node of $\mathcal{T}$, which we denote as $root^{\mathcal{T}}$. 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 $val^{\mathcal{T}}(root^{\mathcal{T}})$ to the head of rule $rule^{\mathcal{T}}(root^{\mathcal{T}})$ results in the fact $\boldsymbol{f}$.

2.6.1 Schedulings

To relate derivation trees to runs, we use the concept of schedulings. 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 the rule $rule^{\mathcal{T}}(x)$ should fire under valuation $val^{\mathcal{T}}(x)$.

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.

2.7 Transducer Networks

We now formalize a network of compute nodes. 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 that maps each node $x \in \mathcal{N}$ to a transducer schema, and $\boldsymbol{\Pi}$ is a function that maps each node $x \in \mathcal{N}$ to a transducer over the schema $\boldsymbol{\Upsilon}(x)$. For technical convenience, we assume that all transducer schemas use the same message relations. This is not really a restriction because the transducers are not obliged to use all message relations. We make no further assumptions about how names for input, output and memory relations might be shared by several nodes.

2.7.1 Distributed Schemas

Naturally, we can define the distributed input database schema $in^{\boldsymbol{\mathcal{N}}}$ for $\boldsymbol{\mathcal{N}}$ that maps each node x to the input schema of $\boldsymbol{\Upsilon}(x)$. The distributed schemas $out^{\boldsymbol{\mathcal{N}}}$ and $mem^{\boldsymbol{\mathcal{N}}}$ can be defined similarly.

2.7.2 Operational Semantics

Any distributed database instance over $in^{\boldsymbol{\mathcal{N}}}$ can be given as input to $\boldsymbol{\mathcal{N}}$. Let H be such an instance. Let Υ_{msg} denote the shared message schema of $\boldsymbol{\mathcal{N}}$. A *configuration of* $\boldsymbol{\mathcal{N}}$ *on* H is a pair $\rho = (s, b)$ of functions s and b where for each $x \in \mathcal{N}$,

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

We call s the *state function* and b the *buffer function*. Intuitively, the instance H is used to initialize each node, and for each $x \in \mathcal{N}$, the system relations `Id` and `All` provide the local transducer $\boldsymbol{\Pi}(x)$ the identity of the node x it is running on and the identities of the other nodes. Next, the buffer function maps each $x \in \mathcal{N}$ to the multiset of messages that have been sent to x but that have not yet been delivered to x. A multiset allows us to represent duplicates of the same message (sent at different times).

The *start configuration of* $\mathcal{N}$ *on* H, denoted $start(\boldsymbol{\mathcal{N}}, H)$, is the unique configuration $\rho = (s, b)$ where for each $x \in \mathcal{N}$, 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 $x \in \mathcal{N}$, 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

$$s_1(x), set(m) \to s_2(x), J_{\text{snd}}$$

  is a local transition of transducer $\boldsymbol{\Pi}(x)$;
- for each $y \in \mathcal{N}\backslash\{x\}$ we have $s_1(y) = s_2(y)$;
- for $y \in \mathcal{N}\backslash\{x\}$ we have $b_2(y) = b_1(y) \cup J_{\text{snd}}^{\rightarrow y}$ (multiset union) and for x we have $b_2(x) = (b_1(x)\backslash m) \cup J_{\text{snd}}^{\rightarrow x}$ (multiset union and difference) where $J_{\text{snd}}^{\rightarrow z} = \{R(\bar{a}) | R(z, \bar{a}) \in J_{\text{snd}}\}$ for each $z \in \mathcal{N}$.

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 addressed to specific nodes (possibly itself). The first component of each message 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 outside the network are lost. If $m = \emptyset$, we call this global transition a *heartbeat* transition and otherwise we call it a *delivery* 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 a transducer network $\boldsymbol{\mathcal{N}}$ on distributed input database instance H is a *finite* sequence of global transitions $\rho_i \xrightarrow{x_i m_i} \rho_{i+1}$ for $i = 1, 2, 3, \ldots, n$, with $n \in \mathbb{N}$, where $\rho_1 = start(\boldsymbol{\mathcal{N}}, H)$, and the ith transition with $i \geq 2$ operates on the resulting configuration of the previous transition $i - 1$. We write $last(\mathcal{R})$ to denote the last configuration reached by $\mathcal{R}$.

Note that 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.

2.7.3 Example

Here we give an example transducer network.

Example 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

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

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

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

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.

2.8 Encoding

We specify how a transducer network can be given as input to a decision procedure. Let $\boldsymbol{\mathcal{N}}$ be a transducer network. The encoding is a sequence of transducers (and their schemas), one for each node of $\boldsymbol{\mathcal{N}}$. 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 Example 1.[2] 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

[2]The components of the body atoms have to be specified in full, because we need to describe which variables are used, and how they are potentially shared between atoms.

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

3 Confluence

Let $\mathcal{N} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a transducer network. Let H be an input distributed database instance for $\mathcal{N}$. By the asynchronous nature of message delivery, different runs of $\mathcal{N}$ on H can deliver messages in different orders. So, if a transducer at some node $x \in \mathcal{N}$ applies negation too quickly, without having seen some crucial messages, we could accidentally produce a wrong output. Worse, output facts can never be retracted once they are produced. Transducer networks where such problems are not possible are called confluent.

Formally, we call $\mathcal{N}$ *confluent on* H if for any two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\mathcal{N}$ 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, then for any run there exists an extension in which that fact can be produced on that node too. Naturally, we call $\mathcal{N}$ *confluent* if $\mathcal{N}$ is confluent on all input distributed database instances. If $\mathcal{N}$ is not confluent, we say that $\mathcal{N}$ is *diffluent*. Our definition of confluence is a formalization of the notion of "eventual consistency" [4, 17], but see also Section 8 for a discussion.

The transducer network given in Example 1 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 so that y outputs again this fact. Therefore the transducer network is confluent.

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

Example 2 Let $\mathcal{N} = \{x, y\}$ be a network. We define a transducer network $\mathcal{N} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ 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:

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

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. These two runs show that $\boldsymbol{\mathcal{N}}$ is not confluent.

3.1 Decision Problem

Since output facts can not be retracted once they are produced, it 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 3.2, and Section 3.3 investigates their effect on decidability.

3.2 Syntactical Restrictions

We introduce several syntactical restrictions on individual transducers and on transducer networks as a whole.

Let $\boldsymbol{\Pi}$ be a transducer over a schema $\boldsymbol{\Upsilon}$. For an individual rule φ of $\boldsymbol{\Pi}$, we consider the following possible restrictions:

- We say that φ is *message-positive* if there are no message atoms in neg^{φ}. 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.
- We say that φ is *message-bounded* if $bound(\varphi) \subseteq A$ and $bound(\varphi) \cap B = \emptyset$, where A and B are respectively the set of variables of φ occurring in positive message atoms, and 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" [13, 14, 23].[3]

[3]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.

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;[4] 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 networks of Examples 1 and 2 are simple transducer networks.

3.3 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:

[4]The restrictions considered by Deutsch et al. [13] 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.

Proposition 1 *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. [14], we reduce the the finite implication problem for functional and inclusion dependencies [10] to the diffluence decision problem. We sketch the proof; the technical details are in Appendix A.1. 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 $\mathcal{D}$.[5] We have to check validity of $(\mathcal{D}, \Sigma, \sigma)$.

For the instance $(\mathcal{D}, \Sigma, \sigma)$, we construct a single-node transducer network $\mathcal{N}$ that is simple except that send rules are not static, and so that $\mathcal{N}$ is diffluent iff $(\mathcal{D}, \Sigma, \sigma)$ is not valid. Let Π denote the single transducer of $\mathcal{N}$. 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. For each violated dependency $\tau \in \Sigma \cup \{\sigma\}$, transducer Π sends a $\texttt{viol}_\tau()$-message to itself. Non-static send rules are needed for checking the inclusion dependencies.

Upon receiving $\texttt{viol}_\sigma()$, the transducer can do something diffluent, by blocking a rule for output relation T as was done in Example 2, so that an incoming $A_{\text{msg}}(a)$-fact is ignored when memory fact $B(a)$ was previously created. But when some $\texttt{viol}_\tau()$ message with $\tau \in \Sigma$ is received, we can repair the inconsistencies. Concretely, 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 $\mathcal{N}$, 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 $\texttt{viol}_\sigma()$, we can block the output. No repairs are possible because only σ is violated.

Conversely, if $\mathcal{N}$ is not confluent on some input J, this can only be explained by σ being violated and no dependency of Σ, so that the input of $\mathcal{N}$ 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 2 *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.*

Proof Inspired by the proof technique of Deutsch et al. [14], we reduce the Post correspondence problem [21] to the diffluence decision problem. We sketch the proof;

[5] 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 $\sigma \in \Sigma$.

the technical details are in Appendix A.2. 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 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 $\mathcal{N}$ that is simple except that messages canhave cyclic dependencies, and so that $\mathcal{N}$ is diffluent iff (U, V) has a match. Let Π denote the single transducer of $\mathcal{N}$. 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 $\texttt{align}[i, k, l](a, b)$, 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 $\texttt{align}[i, |u_i|, |v_i|](a, b)$ 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 $\texttt{align}[j, 1, 1](a', b')$, 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 $\mathcal{N}$. So, the above alignment process can eventually send a message of the form $\texttt{align}[j, |u_j|, |v_j|](a, a)$, 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 Example 2.

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

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

Theorem 1 *Diffluence for simple transducer networks is decidable in* NEXPTIME*; the problem is* NEXPTIME*-complete.*

Theorem 1 is proven in Sections 4, 5, and 6.

4 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 2.8). The merit of this section lies in reducing the technical complexity for the decidability result (Sections 5 and 6) and the expressivity analysis (Section 7).

First, Section 4.1 gives syntactical simplifications for single-node networks. Next, Section 4.2 formalizes the notion of simulation and formulates the result. The sections thereafter show the result: Sections 4.3 and 4.4 respectively define the transducer schema and transducer of $\mathcal{M}$, and Section 4.5 shows that $\mathcal{M}$ satisfies the desired properties.

4.1 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 $in^{\mathcal{N}}$, $out^{\mathcal{M}}$ and $mem^{\mathcal{M}}$ (Section 2.7.1) are regarded as ordinary (non-distributed) database schemas. Accordingly, an input for $\mathcal{N}B$ 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.

4.2 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

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

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

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

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

$$\{x.\mathtt{Id}(x),\ \mathtt{Node}(x) | x \in \mathcal{N}\}.$$

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 $J|_{sch^{\mathcal{N}}} = inst^{\mathcal{N}}$.[6] An instance that is not well-formed is called *ill-formed*.

[6] I is isomorphic to J if there is an injective function f : **dom** → **dom** such that $f(I) = J$.

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

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

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

We use $in^{\mathcal{M}} = \langle in^{\mathcal{N}} \rangle^{\mathcal{N}}$ instead of $in^{\mathcal{M}} = \langle in^{\mathcal{N}} \rangle$ because $\mathcal{M}$ needs the identifiers of the nodes to simulate message sending and the nodes' comparisons of their identifier to input values, and because we do not use values from **dom** directly in rules (cf. Section 2.3).

Now we are ready to present the result:

Proposition 3 *For each simple transducer network $\mathcal{N}$, there exists a simple single-node transducer network $\mathcal{M}$ such that* (i) *$\mathcal{M}$ simulates $\mathcal{N}$, and* (ii) *$\mathcal{M}$ is confluent iff $\mathcal{N}$ is confluent.*

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

4.3 Transducer Schema

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

- $\Upsilon_{\text{in}} = \langle in^{\mathcal{N}} \rangle^{\mathcal{N}}$; $\Upsilon_{\text{out}} = \langle out^{\mathcal{N}} \rangle$; $\Upsilon_{\text{mem}} = \langle mem^{\mathcal{N}} \rangle$; and,
- Υ_{msg} consists of (i) the relations $R^{(k+1)}_{\rightarrow x}$ for which $x \in \mathcal{N}$ and $R^{(k)} \in \mathcal{D}^{\mathcal{N}}_{\text{msg}}$, (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 $R_{\rightarrow x}$ is used to send R-facts specifically to node x. The latter relations have an incremented arity when compared to $\mathcal{D}^{\mathcal{N}}_{\text{msg}}$, for the following reason. Each transition of the transducer Π in $\mathcal{M}$ 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.

4.4 Transducer Rules

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

4.4.1 Output and Memory

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

- input, output and memory atoms $R(\bar{u})$ in φ become $x.R(\bar{u})$ in φ', including the head;
- atoms of the form `Id(u)` and `All(u)` in φ become respectively x.`Id(u)` and `Node(u)` in φ';
- (positive) message atoms $R(\bar{u})$ in φ become $R_{\to x}(z, \bar{u})$ in φ' where `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 $\texttt{do}_x()$.

Intuitively, because relation `All` always contains $\mathcal{N}$ on every node of $\mathcal{N}$, it is replaced by the shared relation `Node` in $\mathcal{M}$. For a message atom $R_{\to x}(z, \bar{u})$, the new variable `z` represents the extra sender-component (cf. Section 4.3). 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 $\Pi(x)$ is simple, there are no deletion rules for memory relations, so we don't have to translate these.

4.4.2 Messages

We do the following for each node $x \in \mathcal{N}$. Let $T^{(k)}$ be a shared message relation of $\mathcal{N}$. All rules for relation T in $\Pi(x)$ are message-positive and static. To let simulated node x send messages in $\mathcal{M}$, we add to Π all rules φ'_y obtained by combining a sending rule φ for T in $\Pi(x)$ 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 $head^{\varphi} = T(n_0, \bar{u})$, where n_0 is the addressee variable. Let n_1 be a new variable. Rule φ'_y is obtained as follows:

- the head $T(n_0, \bar{u})$ of φ becomes the head $T_{\to y}(n_1, \bar{u})$ in φ'_y;
- φ'_y contains positive body atoms $y.\texttt{Id}(n_0)$ and $x.\texttt{Id}(n_1)$;
- input atoms $R(\bar{u})$ in φ become $x.R(\bar{u})$ in φ'_y;
- atoms of the form `Id(u)` and `All(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 $\texttt{do}_x()$.

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.

4.4.3 Ill-Formed Inputs

We indicate how $\mathcal{M}$ can be made confluent on ill-formed instances. First, using message-positive and static send rules, it is possible to send a message $\mathtt{error}()$ if the following constraints are violated: some relation $x.\mathtt{Id}$ contains two different values; two relations $x.\mathtt{Id}$ and $y.\mathtt{Id}$ with $x \neq y$ share a value; 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.

4.4.4 Check Simple

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

4.5 Simulation and Confluence Equivalence

We now show that *(i)* $\mathcal{M}$ simulates $\mathcal{N}$ and *(ii)* $\mathcal{M}$ is confluent iff $\mathcal{N}$ is confluent. First we need some additional concepts and notations. Let $\rho = (s, b)$ be a configuration of $\mathcal{N}$ on input H and let $\sigma = (s', b')$ be a configuration of $\mathcal{M}$ 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 is about relations $\mathtt{Id}$ and $\mathtt{All}$. By definition of $\langle H \rangle^{\mathcal{N}}$, configuration σ is always input- and system-equivalent to ρ.

We say that σ is *message-equivalent* to ρ 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 4.1 and 4.2 show that $\mathcal{M}$ simulates $\mathcal{N}$, but they are phrased slightly more general for later use in the confluence equivalence:

Claim 4.1 *Every run $\mathcal{R}$ of $\mathcal{N}$ on an input H can be converted to a run $\mathcal{S}$ of $\mathcal{M}$ on $\langle H \rangle^{\mathcal{N}}$ such that $last(\mathcal{S})$ and $last(\mathcal{R})$ 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 behaviour 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. □

Claim 4.2 *Let H be an input for $\mathcal{N}$. Every run $\mathcal{S}$ of $\mathcal{M}$ on $\langle H \rangle^{\mathcal{N}}$ can be converted to a run $\mathcal{R}$ of $\mathcal{N}$ on H such that $last(\mathcal{R})$ and $last(\mathcal{S})$ are output- and memory-equivalent, and $last(\mathcal{S})$ has its messages included in $last(\mathcal{R})$.*

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.[7] This results in a new run $\mathcal{S}'$ such that $last(\mathcal{S})$ and $last(\mathcal{S})'$ have the same output and memory facts, and such that the buffer of $last(\mathcal{S})$ is included in the buffer of $last(\mathcal{S}')$ 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 $\mathcal{M}$, 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 $last(\mathcal{S}')$ and $last(\mathcal{S}'')$ are exactly the same when ignoring the $\mathtt{do}_x$-messages.

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

Starting from the second transition, run $\mathcal{S}''$ simulates precisely one node in each transition. In the opposite fashion as in Claim 4.1, we can now convert $\mathcal{S}''$ to a run $\mathcal{R}$ of $\mathcal{N}$ on input H so that $last(\mathcal{S}'')$ and $last(\mathcal{R})$ are output-, memory-, and message-equivalent. Note that $last(\mathcal{S})$ and $last(\mathcal{R})$ are output- and memory-equivalent, and $last(\mathcal{S})$ has its messages included in $last(\mathcal{R})$. □

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

Claim 4.3 *If $\mathcal{M}$ is confluent then $\mathcal{N}$ is confluent.*

Proof Let H be an input for $\mathcal{N}$. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two runs of $\mathcal{N}$ on H, where $last(\mathcal{R}_1)$ 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 $last(\mathcal{R}_2')$ also contains fact $R(\bar{a})$ at x. Using Claim 4.1, we can make two runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\mathcal{M}$ on $\langle H \rangle^{\mathcal{N}}$ such that for each $i \in \{1, 2\}$, configurations $last(\mathcal{S}_i)$ and $last(\mathcal{R}_i)$ are output-, memory-, and message-equivalent. In particular, $last(\mathcal{S}_1)$ contains output fact $x.R(\bar{a})$. By confluence of $\mathcal{M}$, run $\mathcal{S}_2$ can be extended to a run $\mathcal{S}_2'$ such that $last(\mathcal{S}_2)$ also contains $x.R(\bar{a})$. Lastly, extension $\mathcal{S}_2'$ gives rise to an extension $\mathcal{R}_2'$ such that $last(\mathcal{R}_2')$ is output- and memory-equivalent to $last(\mathcal{S}_2')$, and so $last(\mathcal{R}_2')$ contains $R(\bar{a})$ at x: the proof is similar to that of Claim 4.2, 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 mimick the behaviour of $\mathcal{S}_2'$. □

Claim B.1 *If $\mathcal{N}$ is confluent then $\mathcal{M}$ is confluent.*

Proof Let I be an input for $\mathcal{M}$. We have to show that $\mathcal{M}$ 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 $\texttt{error}()$. Then any run can be extended to produce all possible output facts, so potential inconsistencies can always be corrected.

Now suppose that I is well-formed, which means there is an instance J isomorphic to I with $J|_{sch^{\mathcal{N}}} = inst^{\mathcal{N}}$ (cf. Section 4.3). Because transducer rules of $\mathcal{M}$ only express generic queries, it is sufficient to show that $\mathcal{M}$ is confluent on J. Let H be the (unique) input for $\mathcal{N}$ for which $\langle H \rangle^{\mathcal{N}} = J$. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two runs of $\mathcal{M}$ on J, where $last(\mathcal{S}_1)$ 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 4.2 to run $\mathcal{S}_1$, we can construct a run $\mathcal{R}_1$ of $\mathcal{N}$ on input H such that $last(\mathcal{S}_1)$ and $last(\mathcal{R}_1)$ are output- and memory-equivalent. In particular, output fact $R(\bar{a})$ is at node x in $last(\mathcal{R}_1)$.

Next, suppose we can construct an extension $\mathcal{S}_2''$ of $\mathcal{S}_2$ and a run $\mathcal{R}_2''$ of $\mathcal{N}$ on input H such that $last(\mathcal{S}_2'')$ and $last(\mathcal{R}_2'')$ are output-, memory-, *and* message-equivalent. If by chance $last(\mathcal{S}_2'')$ already contains $x.R(\bar{a})$ then we are ready. Otherwise, by

output-equivalence of $last(\mathcal{S}_2'')$ and $last(\mathcal{R}_2'')$, fact $R(\bar{a})$ will not be at x in $last(\mathcal{R}_2'')$. But, by confluence of $\mathcal{N}$, 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 $last(\mathcal{S}_2'')$ and $last(\mathcal{R}_2'')$, 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 4.1.

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

Message Saturation Because transducer Π of $\mathcal{M}$ is recursion-free, 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.[8] 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, $last(\mathcal{S}_2')$ contains all messages that could possibly be sent on input J.

Run of $\mathcal{N}$ Applying Claim 4.2 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 $last(\mathcal{S}_2')$ and $last(\mathcal{R}_2')$ are output- and memory-equivalent, and such that the messages of $last(\mathcal{S}_2')$ are included in $last(\mathcal{R}_2')$. We now show that actually all messages in the buffers of $last(\mathcal{R}_2')$ are simulated in the (single) buffer of $last(\mathcal{S}_2')$, except for maybe their precise cardinalities.

Let $S(\bar{b})$ be a message in the buffer of some node y in $last(\mathcal{R}_2')$. 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 $last(\mathcal{S}_2')$, 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 $last(\mathcal{R}_2')$. 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 $last(\mathcal{S}_2')$.

As we have seen above, $last(\mathcal{R}_2')$ and $last(\mathcal{S}_2')$ 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 $last(\mathcal{R}_2'')$ and $last(\mathcal{S}_2'')$ are output- and memory-equivalent. But they

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

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 $last(\mathcal{R}_2')$ and $last(\mathcal{S}_2')$) emptied during the delivery, the cardinalities of messages in $last(\mathcal{R}_2'')$ and $last(\mathcal{S}_2'')$ are the same. □

5 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 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 transducer and its schema in $\mathcal{N}$. Like in Section 4, 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.

5.1 Syntactical Quantities

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

- the length $\mathbf{P}$ the longest path in the positive dependency graph of Π (defined in Section 3.2), where the length of a path is measured as the number of edges on this path;
- the largest number $\mathbf{B}$ of positive body atoms in any rule of Π;
- the largest arity $\mathbf{I}$ among input relations;
- the largest arity $\mathbf{O}$ among output relations;
- the number $\mathbf{C}$ of different output and memory facts that can be made with values in A, where $A \subseteq \mathbf{dom}$ is an arbitrary set with $|A| = \mathbf{O}$.

Now, let $sizeDom(\mathcal{N})$ abbreviate the expression $2\mathbf{ICB}^{\mathbf{P}}$. We have the following small model property:

Proposition 4 *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.

5.2 Proof Outline

Here we sketch the proof of Proposition 4. The details are provided by the following subsections. The proof technique is inspired by *pseudoruns* from Deutsch et al. [13], although it was adapted to deal with the diffluence problem and to deal with

message buffers (multisets). Let $\mathcal{N}$, Π and Υ be like above, and recall the syntactical quantities of $\mathcal{N}$ from Section 5.1.

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 that 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 $\boldsymbol{f}$. Let $C \subseteq \mathbf{dom}$ be the set of values in $\boldsymbol{f}$. Note that $|C| \leq \mathbf{o}$.

In Section 5.3, 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 $|K_i| \leq \mathbf{CB^P}$. This gives the instances K_1 and K_2. Note that $C \subseteq adom(K_1)$ because $\boldsymbol{f}$ is created in $\mathcal{R}_1$. Define

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

Note that $|adom(J)| \leq 2\mathbf{ICB^P} = sizeDom(\mathcal{N})$.

Next, in Section 5.4, for $i = 1, 2$, we will construct a run $\mathcal{S}_i$ on input J with the following properties:

- $last(\mathcal{S}_i)$ and 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 $last(\mathcal{S}_i')$ and $last(\mathcal{R}_i')$ 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 $last(\mathcal{S}_1)$ contains $\boldsymbol{f}$ and $last(\mathcal{S}_2)$ does not. Moreover, if $\mathcal{S}_2$ can be extended to a run $\mathcal{S}_2'$ such that $last(\mathcal{S}_2')$ 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 $last(\mathcal{R}_2)'$ also contains $\boldsymbol{f}$. But the latter is not possible by assumption on $\mathcal{R}_2$. Hence, $\mathcal{S}_2'$ does not exist, and $\mathcal{N}$ is diffluent on the instance J, whose active domain size is upper bounded by $sizeDom(\mathcal{N})$, as desired.

5.3 Input Selection

Consider the symbols defined in Sections 5.1 and 5.2. 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 $|K| \leq \mathbf{CB^P}$.

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.

5.3.1 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 $hist_{\mathcal{R}}$ 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 $msg_{\mathcal{R}}$ 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 required 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 $hist_{\mathcal{R}}$, denoting $hist_{\mathcal{R}}(i, \boldsymbol{g}) = (\varphi, V)$, such that $\boldsymbol{h} \in V(pos^{\varphi})$. 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 $|K| \leq \mathbf{C}\mathbf{B}^{\mathbf{P}}$. 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 $\mathbf{P}$ by recursion-freeness of Π. Moreover, in the worst case, each message recursively requires $\mathbf{B}$ other messages. Therefore, the number of input facts needed by $\boldsymbol{g}$ alone is bounded by $\mathbf{B}^{\mathbf{P}}$. And since at most $\mathbf{C}$ different output and memory C-facts are created in $\mathcal{R}$, we overall have that $|K| \leq \mathbf{C}\mathbf{B}^{\mathbf{P}}$, as desired.

5.3.2 Natural Properties

Section 5.3.1 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 5.4 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 Appendix B.1, 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 $\rho_i = \left(s_i^{\mathcal{R}}, b_i^{\mathcal{R}}\right)$;

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. $\mathcal{E}_i = \gamma_{i+1} \cap \delta_i^R$, where δ_i^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., $\boldsymbol{h} \in \mathcal{E}_i$).

5.4 Run Projection

Consider the symbols defined in Section 5.2. 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:

- $last(\mathcal{S})$ and $last(\mathcal{R})$ 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 $last(\mathcal{S}')$ and $last(\mathcal{R}')$ again contain precisely the same output and memory C-facts.

To improve the readability of this section, helper claims are placed in Appendix B.2. First, Claim B.4 tells us that the second property above holds when the first property holds *and* when the message buffer of $last(\mathcal{S})$ is included in the message buffer of $last(\mathcal{R})$. 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 $\mathcal{S}'$.

We first sketch the main idea in the construction of $\mathcal{S}$. For run $\mathcal{R}$, let $hist_{\mathcal{R}}$, $msg_{\mathcal{R}}$, β_i, γ_i, and $\mathcal{E}_i$ be as defined in Section 5.3. We assume that $msg_{\mathcal{R}}$ satisfies the properties given in Section 5.3.2. 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 $msg_{\mathcal{R}}$, so that the output and memory C-facts of $\mathcal{R}$ are faithfully created. One caveat, however, is that some transitions of $\mathcal{R}B$ should sometimes deliver more messages than just those of $msg_{\mathcal{R}}$ 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 $i \in \{1, \ldots, n+1\}$, we denote the ith 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 inductively specify the message deliveries of $\mathcal{S}$ so that the following properties are satisfied for each $i \in \{1, \ldots, n+1\}$:

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 helps in proving them. 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 $\gamma_1 = \emptyset$, which follows from $b_1^{\mathcal{R}} = \emptyset$ and the property $\gamma_1 \sqsubseteq b_1^{\mathcal{R}}$ of $msg_{\mathcal{R}}$. For the induction hypothesis, we assume that the properties are satisfied for γ_i and σ_i. For the inductive step, we show that they are satisfied 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*:

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

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 $\mathcal{R}$.

The following subsections show the properties 1 to 3.

5.4.1 Property 1

We show that $s_{i+1}^{\mathcal{S}}$ and $s_{i+1}^{\mathcal{R}}$ contain the same output and memory C-facts. First, because $m_i^{\mathcal{S}} \sqsubseteq m_i^{R}$, Claim B.6 tells us that the output and memory C-facts of $s_{i+1}^{\mathcal{S}}$ are a subset of those in $s_{i+1}^{\mathcal{R}}$. For the other direction, let $\boldsymbol{g}$ be an output or memory C-fact in $s_{i+1}^{\mathcal{R}} \setminus s_i^{\mathcal{R}}$. Because $\boldsymbol{g}$ is a C-fact, the mapping $hist_{\mathcal{R}}(i, \boldsymbol{g}) = (\varphi, V)$ is defined, where valuation V is satisfying for φ during transition i of $\mathcal{R}$ and derives $\boldsymbol{g}$. We show that this is also true during transition i of $\mathcal{S}$, so that $\boldsymbol{g} \in s_{i+1}^{\mathcal{S}}$. We look at the different components in the body of φ:

- Consider the input atoms. Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\Upsilon_{\text{in}}}$. We have to show $\boldsymbol{h} \in J$. First, because V is satisfying for φ during transition i of $\mathcal{R}$, we have $\boldsymbol{h} \in I$. Moreover, because $\boldsymbol{h}$ is an input fact needed in $hist_{\mathcal{R}}$, we have $\boldsymbol{h} \in K$ (Section 5.3). Hence, $\boldsymbol{h} \in I^{[adom(K)]} \subseteq I^{[adom(K_1) \cup adom(K_2)]} = J$. Let $\boldsymbol{h} \in V(neg^{\varphi})|_{\Upsilon_{\text{in}}}$. We have to show $\boldsymbol{h} \notin J$. This follows from $\boldsymbol{h} \notin I$ (because V is satisfying in $\mathcal{R}$) and $J \subseteq I$.
- Consider the message atoms. Recall that φ is message-positive. Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\Upsilon_{\text{msg}}}$. We have to show that $\boldsymbol{h}$ is delivered in transition i of $\mathcal{S}$, i.e., $\boldsymbol{h} \in set(m_i^{\mathcal{S}})$. Because $\boldsymbol{h}$ is a message needed in $hist_{\mathcal{R}}$, there is a triple $(k, \boldsymbol{h}, i) \in msg_{\mathcal{R}}$ for some $k < i$. Hence, $\boldsymbol{h} \in \beta_i$. Finally, Claim B.3 applied to $\gamma_i \sqsubseteq b_i^{\mathcal{S}}$ (induction hypothesis) gives $\beta_i \subseteq set(m_i^{\mathcal{S}})$.
- Consider the output and memory atoms. Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$. We have to show $\boldsymbol{h} \in s_i^{\mathcal{S}}$. First, because V is satisfying in $\mathcal{R}$, we have $\boldsymbol{h} \in s_i^{\mathcal{R}}$. Moreover, because $\boldsymbol{g}$ is a C-fact, the message-boundedness of φ implies that $\boldsymbol{h}$ is a C-fact. Hence, $\boldsymbol{h} \in s_i^{\mathcal{S}}$ by the induction hypothesis. Similarly, for each $\boldsymbol{h} \in V(neg^{\varphi})|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$ we can show $\boldsymbol{h} \notin s_i^{\mathcal{S}}$.
- The nonequalities of φ are satisfied under V in $\mathcal{R}$, hence in $\mathcal{S}$ as well.

We conclude that V is satisfying for φ in transition i of $\mathcal{S}$.

5.4.2 Property 2

We show $b_{i+1}^{\mathcal{S}} \sqsubseteq b_{i+1}^{\mathcal{R}}$. By the operational semantics, $b_{i+1}^{\mathcal{S}} = \left(b_i^{\mathcal{S}} \backslash m_i^{\mathcal{S}}\right) \cup \delta_i^{\mathcal{S}}$ and $b_{i+1}^{\mathcal{R}} = \left(b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right) \cup \delta_i^{\mathcal{R}}$, where $\delta_i^{\mathcal{S}}$ and $\delta_i^{\mathcal{R}}$ denote the set of messages sent in transition i of $\mathcal{S}$ and $\mathcal{R}$ respectively. Because $\delta_i^{\mathcal{S}} \subseteq \delta_i^{\mathcal{R}}$ by Claim B.6, it is sufficient to show $b_i^{\mathcal{S}} \backslash m_i^{\mathcal{S}} \sqsubseteq b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}$. Let $\boldsymbol{g}$ be an arbitrary fact. We show $num\left(\boldsymbol{g}, b_i^{\mathcal{S}} \backslash m_i^{\mathcal{S}}\right) \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right)$.

Because $m_i^{\mathcal{S}} \sqsubseteq b_i^{\mathcal{S}}$, we have $num\left(\boldsymbol{g}, b_i^{\mathcal{S}} \backslash m_i^{\mathcal{S}}\right) = num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right) - num\left(\boldsymbol{g}, m_i^{\mathcal{S}}\right)$. Applying the definition of $m_i^{\mathcal{S}}$ further gives

$$\begin{aligned} num\left(\boldsymbol{g}, b_i^{\mathcal{S}} \backslash m_i^{\mathcal{S}}\right) &= num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right) - \min\left\{num\left(\boldsymbol{g}, b_i^{\mathcal{S}} \backslash (\gamma_i \backslash \beta_i)\right), num\left(\boldsymbol{g}, m_i^{\mathcal{R}}\right)\right\} \\ &= \max\{e_1, e_2\}, \end{aligned}$$

where

$$\begin{aligned} e_1 &= num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right) - num\left(\boldsymbol{g}, b_i^{\mathcal{S}} \backslash (\gamma_i \backslash \beta_i)\right), \text{ and} \\ e_2 &= num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right) - num\left(\boldsymbol{g}, m_i^{\mathcal{R}}\right). \end{aligned}$$

We show that both $e_1 \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right)$ and $e_2 \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right)$.

- We show $e_1 \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right)$. First, rewriting $e_1 = num\left(\boldsymbol{g}, b_i^{\mathcal{S}} \backslash \left(b_i^{\mathcal{S}} \backslash (\gamma_i \backslash \beta_i)\right)\right)$ and applying $\gamma_i \backslash \beta_i \sqsubseteq b_i^{\mathcal{S}}$ (follows from induction hypothesis $\gamma_i \sqsubseteq b_i^{\mathcal{S}}$), we obtain $e_1 = num(\boldsymbol{g}, \gamma_i \backslash \beta_i)$.

  Now, since $\gamma_{i+1} = (\gamma_i \backslash \beta_i) \cup \mathcal{E}_i$ (Claim B.2), we further have $e_1 = num(\boldsymbol{g}, \gamma_{i+1} \backslash \mathcal{E}_i)$. If we can show $num(\boldsymbol{g}, \gamma_{i+1} \backslash \mathcal{E}_i) = num\left(\boldsymbol{g}, \gamma_{i+1} \backslash \delta_i^{\mathcal{R}}\right)$ then $\gamma_{i+1} \sqsubseteq b_{i+1}^{\mathcal{R}}$ (property of $msg_{\mathcal{R}}$) implies $e_1 \leq num\left(\boldsymbol{g}, b_{i+1}^{\mathcal{R}} \backslash \delta_i^{\mathcal{R}}\right) = num\left(\boldsymbol{g}, b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right)$, as desired.

  To show $num(\boldsymbol{g}, \gamma_{i+1} \backslash \mathcal{E}_i) = num\left(\boldsymbol{g}, \gamma_{i+1} \backslash \delta_i^{\mathcal{R}}\right)$, it suffices to show that if $\boldsymbol{g} \in \gamma_{i+1}$ then $num(\boldsymbol{g}, \delta_i^{\mathcal{R}}) = num(\boldsymbol{g}, \mathcal{E}_i)$. This equality holds, because $msg_{\mathcal{R}}$ satisfies $\mathcal{E}_i = \gamma_{i+1} \cap \delta_i^{\mathcal{R}}$.
- We show $e_2 \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right)$. We have $b_i^{\mathcal{S}} \sqsubseteq b_i^{\mathcal{R}}$ by the induction hypothesis. Hence, $e_2 \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right) - num\left(\boldsymbol{g}, m_i^{\mathcal{R}}\right)$. But since $m_i^{\mathcal{R}} \sqsubseteq b_i^{\mathcal{R}}$, we may write $e_2 \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right)$, as desired.

5.4.3 Property 3

We show $\gamma_{i+1} \sqsubseteq b_{i+1}^{\mathcal{S}}$. First, Claim B.2 tells us that $\gamma_{i+1} = (\gamma_i \backslash \beta_i) \cup \mathcal{E}_i$. It is sufficient to show $\gamma_i \backslash \beta_i \sqsubseteq b_i^{\mathcal{S}} \backslash m_i^{\mathcal{S}}$ and $\mathcal{E}_i \subseteq \delta_i^{\mathcal{S}}$ because then $\gamma_{i+1} \sqsubseteq \left(b_i^{\mathcal{S}} \backslash m_i^{\mathcal{S}}\right) \cup \delta_i^{\mathcal{S}} = b_{i+1}^{\mathcal{S}}$, as desired.

We show that $\gamma_i \backslash \beta_i \sqsubseteq b_i^{\mathcal{S}} \backslash m_i^{\mathcal{S}}$. First, from the definition of $m_i^{\mathcal{S}}$, we get $m_i^{\mathcal{S}} \sqsubseteq b_i^{\mathcal{S}} \backslash (\gamma_i \backslash \beta_i)$. By adding $\gamma_i \backslash \beta_i$ to both sides of this inclusion, and using $\gamma_i \backslash \beta_i \sqsubseteq b_i^{\mathcal{S}}$ (by induction hypothesis $\gamma_i \sqsubseteq b_i^{\mathcal{S}}$), we obtain $m_i^{\mathcal{S}} \cup (\gamma_i \backslash \beta_i) \sqsubseteq b_i^{\mathcal{S}}$. Hence, $\gamma_i \backslash \beta_i \sqsubseteq b_i^{\mathcal{S}} \backslash m_i^{\mathcal{S}}$.

We show that $\mathcal{E}_i \subseteq \delta_i^{\mathcal{S}}$. Let $\boldsymbol{g} \in \mathcal{E}_i$. By definition of $\mathcal{E}_i$, there is a triple $(i, \boldsymbol{g}, l) \in msg_{\mathcal{R}}$ for some $l > i$, i.e., $\boldsymbol{g}$ is a needed message that should be sent in transition i. By construction of $hist_{\mathcal{R}}$, the mapping $hist_{\mathcal{R}}(i, \boldsymbol{g}) = (\varphi, V)$ is defined, where valuation V is satisfying for rule φ during transition i of $\mathcal{R}$ and derives $\boldsymbol{g}$. We show that V is also satisfying for φ in transition i of $\mathcal{S}$, which gives $\boldsymbol{g} \in \delta_i^{\mathcal{S}}$. This goes similarly as in property 1, where we showed that the C-facts of $s_{i+1}^{\mathcal{R}}$ are in $s_{i+1}^{\mathcal{S}}$, except that this time we only have to consider input atoms, message atoms and nonequalities of φ (because sending rules are static).

6 Decidability

Note that Proposition 4 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 6.1, we give a nondeterministic exponential time (NEXPTIME) decision procedure. In Section 6.2, 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), and (ii) the NEXPTIME lower bound is because single-node networks are a special case of multi-node networks.

6.1 Decision Procedure

In Section 6.1.1 we give the description of the decision procedure. Next, Sections 6.1.2 and 6.1.3 investigate the correctness, and Section 6.1.4 investigates the complexity.

Let $\mathcal{N}$ be a simple single-node transducer network. Let Π and Υ respectively denote the transducer and transducer schema of $\mathcal{N}$. We use the syntactical simplifications for single-node networks (Section 4.1).

6.1.1 Procedure

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

Let $\mathbf{P}$, $\mathbf{B}$, $\mathbf{C}$, and $sizeDom(\mathcal{N})$ be as defined in Section 5.1. Consider the expression $\mathbf{runLen} = \mathbf{C}\mathbf{B}^{\mathbf{P}} + \mathbf{C}$. For $A \subseteq \mathbf{dom}$, we say that a fact $\boldsymbol{f}$ is a A-fact if $adom(\boldsymbol{f}) \subseteq A$. The procedure does the following steps, in order:

1. [Input] Guess an input instance I for $\mathcal{N}$ with $|adom(I)| \leq sizeDom(\mathcal{N})$.
2. [Two runs] Guess two runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\mathcal{N}$ 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 store only the last configuration, and not all previous configurations.

3. [Output] Choose an output fact $\boldsymbol{f}$ in $last(\mathcal{S}_1)$ that is not in $last(\mathcal{S}_2)$. If no such fact can be chosen, then *reject*.
4. [Extension] Denote $C = adom(\boldsymbol{f})$. We extend $\mathcal{S}_2$ by doing $\mathbf{P}+1$ 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*.

6.1.2 Correctness Part 1

Suppose that $\mathcal{N}$ is diffluent. We show that the procedure accepts. Helper claims can be found in Appendix C.1.

First, by the small model property (Section 5), there is an input I for $\mathcal{N}$ such that $|adom(I)| \leq sizeDom(\mathcal{N})$ and $\mathcal{N}$ is diffluent on input I. Thus, there are two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\mathcal{N}$ 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 $\boldsymbol{f}$. We now convert $\mathcal{R}_1$ and $\mathcal{R}_2'$ to runs that the procedure can guess: by Claim C.1, there exists two runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\mathcal{N}$ 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 $last(\mathcal{R}_2')$. Hence, $last(\mathcal{S}_1)$ contains $\boldsymbol{f}$ and $last(\mathcal{S}_2)$ does not. So, the procedure can choose $\boldsymbol{f}$ as the output fact to focus on.

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, and then Claim C.2 can be applied to know that $\boldsymbol{h}$ is delivered in the last transition of $\mathcal{R}_2''$.
- 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 that $\boldsymbol{h}$ is not in $last(\mathcal{R}'_2)$ 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 $last(\mathcal{S}_2)$. Thus $\boldsymbol{h}$ is not in $last(\mathcal{R}'_2)$ by construction of $\mathcal{S}_2$.
- Also, the nonequalities of φ are satisfied under V in $\mathcal{R}''_2$.

6.1.3 Correctness Part 2

Suppose that the procedure accepts. We show that $\mathcal{N}$ 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 $\mathcal{N}$ such that $|adom(I)| \leq sizeDom(\mathcal{N})$. Next, the branch has guessed two runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\mathcal{N}$ on input I, and has been able to choose an output fact $\boldsymbol{f}$ in $last(\mathcal{S}_1)$ that is not in $last(\mathcal{S}_2)$. Denote $C = adom(\boldsymbol{f})$. Lastly, the branch has extended $\mathcal{S}_2$ to a run $\mathcal{S}'_2$ by doing $\mathbf{P}+1$ 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 $\mathcal{N}$ 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 $\mathcal{S}'_2$. Let us assume that $\boldsymbol{g}$ is the *first* such output or memory C-fact. Let φ and V be a rule and valuation that are responsible for deriving $\boldsymbol{g}$. 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 $\boldsymbol{h} \in V(pos^{\varphi})|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$. 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 $last(\mathcal{S}'_2)$. 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 $last(\mathcal{S}_2)$, fact $\boldsymbol{h}$ is in $last(\mathcal{S}_2)$. Hence, $\boldsymbol{h}$ can be read in the last transition of $\mathcal{S}'_2$.
- Let $\boldsymbol{h} \in V(neg^{\varphi})|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$. 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 $last(\mathcal{S}_2)$.

6.1.4 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 We use the encoding of transducer neworks from Section 2.8. Let $|\mathcal{N}|$ denote the input size. Now, consider the syntactical quantities defined in Section 5.1. The quantities $\mathbf{I}$ and $\mathbf{O}$ are upper bounded by $|\mathcal{N}|$ because all input and output relations are used in rules (whose atoms are written in full). The quantities $\mathbf{B}$ and $\mathbf{P}$ are also upper bounded by $|\mathcal{N}|$. Letting n denote the number of different transducer relations, again upper bounded by $|\mathcal{N}|$, the number $\mathbf{C}$ is upper bounded by $n\mathbf{O}^{\mathbf{O}} = n2^{\mathbf{O}\log\mathbf{O}}$, which is single-exponential in $|\mathcal{N}|$. Hence, $sizeDom(\mathcal{N})$ is single-exponential in $|\mathcal{N}|$.

Let **numFc** denote the number of different facts that can be created with $sizeDom(\mathcal{N})$ unique domain values (across all relations). Note that **numFc** is single-exponential in $|\mathcal{N}|$.

Input For each input instance I' for $\mathcal{N}$ with $|adom(I')| \leq sizeDom(\mathcal{N})$, the procedure can guess an isomorphic instance I. Because $sizeDom(\mathcal{N})$ 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**·**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 $J \cup set(m)$ then derive $\boldsymbol{g} = V(head^{\varphi})$. 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 $J \cup set(m) \models V(\boldsymbol{l})$, which can be done in single-exponential time because $|J \cup set(m)| \leq$ **numFc**.

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

Extension In the last step, the procedure extends $\mathcal{S}_2$ with $\mathbf{P}{+}1$ transitions, in each of which we deliver the entire message buffer. The message buffer in $last(\mathcal{S}_2)$ contains at most **runLen**·**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 $\mathbf{P}{+}1$ transitions can be done in single-exponential time.

6.2 Complexity Lower Bound

In Section 6.1 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 [22].

Fix some word w over Σ. We construct a simple single-node transducer network $\mathcal{N}$ for w such that $\mathcal{N}$ is diffluent iff M accepts w. We use the syntactical simplifications of single-node networks (Section 4.1).

6.2.1 Turing Machine

First, following the conventions in Sipser [22], 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 $Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\text{L}, \text{R}\})$, where L and R indicate whether the tape head moves left or right after performing a transition.

6.2.2 Construction

We define the transducer schema Υ and transducer Π of $\mathcal{N}$. 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 behaviour. 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, 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 $A^{(1)}_{\text{msg}}$, $B^{(1)}_{\text{msg}}$, $\texttt{accept}^{(0)}$ and $\texttt{error}^{(0)}$. The following rules implement the basic idea of making Π diffluent when `accept()` is received; we can vary the delivery order of A_{msg}-facts and B_{msg}-facts. The purpose of relation `error` was explained above.

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

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 that k is considered a constant in the construction of the transducer.

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 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 `accept()`. Newly mentioned relations are assumed to be added to Υ_{msg}. The idea is as follows: in the relations of Table 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

Table 1 Computation trace input 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

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:[9]

$$\texttt{reach}_m(\texttt{i}, \texttt{j}) \leftarrow \texttt{reach}_{m-1}(\texttt{i}, \texttt{l}),\ \texttt{reach}_p(\texttt{l}, \texttt{j})$$
$$\text{for each } m = 1, \ldots, n^k, \text{ and each } p = 0, \ldots, m-1.$$

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:

$$\texttt{accept}() \leftarrow \texttt{start}(\texttt{i}),\ \texttt{reach}_m(\texttt{i}, \texttt{j}),\ \texttt{state}(\texttt{j}, \texttt{q}),\ q_{\text{accept}}(\texttt{q})$$
$$\text{for each } m = 0, \ldots, n^k.$$

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

Note that the number and size of the above sending rules is polynomial in n. Appendix C.2 fills in the missing details regarding the messages $\texttt{reach}_0$, $\texttt{start}$, $\texttt{error}$, and argues the correctness.

7 Expressivity

We investigate the expressivity of simple transducer networks. 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 $\mathcal{N} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a confluent transducer network, not necessarily simple. Let $in^{\mathcal{N}}$ and $out^{\mathcal{N}}$ be the distributed schemas for $\mathcal{N}$ as defined in Section 2.7. We say that $\mathcal{N}$ *computes* the following distributed query $\mathcal{Q}$, that is over input schema $in^{\mathcal{N}}$ and output schema $out^{\mathcal{N}}$: $\mathcal{Q}$ maps each instance H over $in^{\mathcal{N}}$ to the instance $\mathcal{Q}(H) = J$ over $out^{\mathcal{N}}$ 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 $\mathcal{N}$ on H. The instance $\mathcal{Q}(H)$ could be defined even if $\mathcal{N}$ is diffluent, but when $\mathcal{N}$ *is* confluent, all runs on H can be extended to obtain $\mathcal{Q}(H)$. We call $\mathcal{Q}(H)$ the *output* of $\mathcal{N}$ 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. First, for a distributed database schema $\mathcal{E}$ over a network $\mathcal{N}$, and an instance H over $\mathcal{E}$, let $\langle\mathcal{E}\rangle^{\mathcal{N}}$ and $\langle H\rangle^{\mathcal{N}}$ be as defined in Section 4.2. 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 an input schema $\mathcal{E}$ and an output schema $\mathcal{F}$, where both schemas are over the same network $\mathcal{N}$. 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$^\neg$

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

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

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

Now we can present the expressivity result:

Theorem 2 *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. These directions are given in the following subsections. Currently, this result depends on our definition of UCQ$^\neg$ as a language with built-in nonequalities (or equivalently by having a built-in equality relation). 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.

7.1 Lower Bound

Let $\mathcal{Q}$ be a distributed query over input distributed schema $\mathcal{E}$ and output distributed schema $\mathcal{F}$, and 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 $\mathcal{Q}$. We assume $\mathcal{E}(x)$ and $\mathcal{F}(x)$ have disjoint relation names for each $x \in \mathcal{N}$; that $\mathcal{E}(x)$ and $\mathcal{F}(x)$ do not contain `Id` or `All`; and, that that any relations we add to $\boldsymbol{\mathcal{N}}$ do not yet occur in $\mathcal{E}$ and $\mathcal{F}$. Any conflicts can always be resolved with appropriate renamings.

7.1.1 Transducer Schemas

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

- the relations $x.R^{(k)}$ and $x.R^{(k)}_\neg$ for each $x \in \mathcal{N}$ and $R^{(k)} \in \mathcal{E}(x)$, to broadcast local inputs;
- the relations $x.\mathtt{Id}^{(1)}$ and $x.\mathtt{Id}^{(k)}_\neg$ 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 set of message relations from above.

7.1.2 Transducer Rules

Let $x \in \mathcal{N}$. We incrementally specify the rules of $\boldsymbol{\Pi}(x)$. First, to send the active domain of the input, 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:

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

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

$$x.R(\texttt{n}, \texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{All}(\texttt{n}),\ R(\texttt{u}_1, \ldots, \texttt{u}_k).$$
$$x.R_{\neg}(\texttt{n}, \texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow \texttt{All}(\texttt{n}),\ \texttt{adom}(\texttt{u}_1),\ \ldots,\ \texttt{adom}(\texttt{u}_k),\ \neg R(\texttt{u}_1, \ldots, \texttt{u}_k).$$

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. 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 $T(\texttt{u}_1, \ldots, \texttt{u}_k)$ of φ becomes the head $x.T(\texttt{n}, \texttt{u}_1, \ldots, \texttt{u}_k)$ of φ', where $\texttt{n}$ is a new variable;
- the positive body atoms of φ' are *(i)* $\texttt{Id}(\texttt{n})$, with $\texttt{n}$ as defined previously; *(ii)* the atoms $\texttt{All}(\texttt{m})$ for which $\texttt{Node}(\texttt{m}) \in pos^{\varphi}$; *(iii)* the atoms $y.R(\texttt{v}_1, \ldots, \texttt{v}_1) \in pos^{\varphi}$, which are now messages; *(iv)* the (positive) message atoms $y.R_{\neg}(\texttt{v}_1, \ldots, \texttt{v}_1)$ for which $y.R(\texttt{v}_1, \ldots, \texttt{v}_1) \in neg^{\varphi}$;
- the negative body atoms of φ' are the atoms $\texttt{All}(\texttt{m})$ for which $\texttt{Node}(\texttt{m}) \in neg^{\varphi}$; and,
- the nonequalities of φ' are those of φ.

The positive body atom $\texttt{Id}(\texttt{n})$ 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:

$$T(\texttt{u}_1, \ldots, \texttt{u}_k) \leftarrow x.T(\texttt{u}_1, \ldots, \texttt{u}_k).$$

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

7.1.3 Example

The following example illustrates the construction of the transducer network.

Example 3 Let $\mathcal{N} = \{x, y\}$. Consider the following distributed schemas $\mathcal{E}$ and $\mathcal{F}$, that are 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$^{\neg}$:

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

Each rule corresponds to one of the output relations.

We construct a transducer network $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ 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. Also, the sending rules for adom are clear, so we do not explicitly give them.

First, the shared message relations of $\mathcal{N}$ are: $x.A^{(2)}$, $x.\mathtt{Id}^{(1)}$, $y.B_{\neg}^{(1)}$ and $\mathtt{adom}^{(1)}$. For node x, we define $\Upsilon(x)_{\text{in}} = \{A^{(2)}\}$, $\Upsilon(x)_{\text{out}} = \{B^{(1)}\}$, and $\Upsilon(x)_{\text{mem}} = \emptyset$. Transducer $\Pi(x)$ contains the rules:

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

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

$$
\begin{aligned}
&y.B_{\neg}(\mathtt{n},\mathtt{u}) \leftarrow \mathtt{All}(\mathtt{n}),\ \mathtt{adom}(\mathtt{u}),\ \neg B(\mathtt{u}).\\
&y.T(\mathtt{n},\mathtt{u}) \leftarrow x.A(\mathtt{u},\mathtt{v}),\ x.\mathtt{Id}(\mathtt{u}).\\
&T(\mathtt{u}) \leftarrow y.T(\mathtt{u}).
\end{aligned}
$$

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

7.2 Upper Bound

Let $\mathcal{N} = (\mathcal{N}, \Upsilon, \Pi)$ be a confluent simple transducer network. Let $\mathcal{Q}$ denote the distributed query computed by $\mathcal{N}$. 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 $\langle in^{\mathcal{N}}\rangle^{\mathcal{N}}$ 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 $in^{\mathcal{N}}$.

The basic idea is to describe the computation of $\mathcal{N}$ with UCQ$^{\neg}$-program Φ, for output relation R at x. To make this technically easier, we first convert $\mathcal{N}$ to a single-node network in Section 7.2.1. Some common notations are introduced in Section 7.2.2, and program Φ is described in Section 7.2.3. The correctness is shown in Appendix D.

7.2.1 Reduction to Single-Node

Consider the concepts from Section 4.2. Using Proposition 3, let $\mathcal{M}$ be the simple single-node transducer network that simulates $\mathcal{N}$, and that is confluent because $\mathcal{N}$ is confluent. By the syntactical simplications of single-node networks (Section 4.1), the query $\mathcal{Q}'$ computed by $\mathcal{M}$ is regarded as an ordinary database query over input schema $\langle in^{\mathcal{N}}\rangle^{\mathcal{N}}$ and output schema $\langle out^{\mathcal{N}}\rangle$. If for every input H for $\mathcal{N}$ we would know that $\mathcal{Q}'(\langle H\rangle^{\mathcal{N}}) = \langle \mathcal{Q}(H)\rangle$, because $x.R$ is in $\langle out^{\mathcal{N}}\rangle$, it will be sufficient to construct the UCQ$^{\neg}$-program Φ as a description of the computation of $\mathcal{M}$ 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 "$x.R$".

Now we are left to show $\mathcal{Q}'(\langle H\rangle^{\mathcal{N}}) = \langle \mathcal{Q}(H)\rangle$ for every input H over $in^{\mathcal{N}}$. Let the output of a configuration ρ, denoted $out(\rho)$, be as defined in Section 4.2. Abbreviate $J = \mathcal{Q}'(\langle H\rangle^{\mathcal{N}})$. We show $J \subseteq \langle \mathcal{Q}(H)\rangle$. By confluence of $\mathcal{M}$, there is a run $\mathcal{S}$ of $\mathcal{M}$ on $\langle H\rangle^{\mathcal{N}}$ such that $out(last(\mathcal{S})) = J$. Next, because $\mathcal{M}$ simulates $\mathcal{N}$, there is a run $\mathcal{R}$ of $\mathcal{N}$ 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 $\mathcal{N}$, there exists a run $\mathcal{R}$ of $\mathcal{N}$ on H such that $\mathcal{Q}(H) = out(last(\mathcal{R}))$. Because $\mathcal{M}$ simulates $\mathcal{N}$, there exists a run $\mathcal{S}$ of $\mathcal{M}$ on $\langle H\rangle^{\mathcal{N}}$ such that $out(last(\mathcal{S})) = \langle out(last(\mathcal{R}))\rangle$. Hence, $\langle \mathcal{Q}(H)\rangle = out(last(\mathcal{S})) \subseteq J$.

7.2.2 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*.

7.2.3 Building the UCQ¬-Program

In this section, we construct the required UCQ¬-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¬ and existential FO with nonequalities, abbreviated ∃FO, some parts are specified in ∃FO for technical convenience.

Let Υ and Π respectively denote the transducer schema and transducer of single-node transducer network $\mathcal{M}$.

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.[10] Hence, the length of $\mathcal{R}^G$ is bounded by the syntactical properties of Π.

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

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

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 $adom(G)$ represent variables. For a derivation tree $\mathcal{T} \in G$, let $msg^{\mathcal{T}} \subseteq int^{\mathcal{T}}$ denote the set of internal nodes x where $lit^{\mathcal{T}}(x)$ 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):

$$sndMsg_G := \bigwedge_{\mathcal{T} \in G} \bigwedge_{x \in msg^{\mathcal{T}}} body^{\mathcal{T}}(x)|_{\Upsilon_{\text{in}}}.$$

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 $lit^{\mathcal{T}}(x)$ is a (positive) output or memory literal.[11] Note that $root^{\mathcal{T}} \in \alpha^{\mathcal{T}}$. 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 $lit^{\mathcal{T}}(y)$ is a negative output or memory literal. By definition of derivation tree, $\beta^{\mathcal{T}}(x)$ contains

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

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

only leafs. For each $x \in \alpha^{\mathcal{T}}$, a node $y \in \beta^{\mathcal{T}}(x)$ is a potential danger: if the fact in the ground literal $val^{\mathcal{T}}(x)(lit^{\mathcal{T}}(y))$, henceforth referred to as "beta fact", is accidentally derived before transition $\kappa^{\mathcal{T}}(x)$, then $val^{\mathcal{T}}(x)$ is unsatisfying 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 say that a truncated derivation tree $\mathcal{S}$ can be *aligned* to $\mathcal{R}^G$ if there is a scheduling $\lambda : int^{\mathcal{S}} \to \{1, \ldots, n\}$ such that for each $x \in int^{\mathcal{S}}$, message set $M^G_{\lambda(x)}$ contains $body^{\mathcal{S}}(x)|_{\gamma_{\text{msg}}}$, 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 $align^G(\boldsymbol{f})$ 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 $align^G(\boldsymbol{f})$ 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 that $\alpha^{\mathcal{T}} = int^{\mathcal{T}}$. The formula is as follows:

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

with

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

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

Intuitively, for each $x \in \alpha^{\mathcal{T}}$, we express *(i)* that the input literals in $body^{\mathcal{T}}(x)$ are satisfied; and, *(ii)* we consider all possible truncated derivation trees for beta facts, and their alignments, and demand that these 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 $succeed_{G,\mathcal{T},\kappa}$ is quantifier-free, with variables in $adom(G)$.

Combining Everything Let $G \subseteq forest_R$ 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 that 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 $\kappa^{\mathcal{T}}$:

$$derive_{G,\mathcal{T}} := \exists \bar{z} \left(diffVal_G \wedge sndMsg_G \wedge succeed_{G,\mathcal{T}^{\downarrow},\kappa^{\mathcal{T}}} \right),$$

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 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 $\mathcal{T}$.[12] 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 Appendix D.

8 Discussion and Future Work

We have shown that under five restrictions: recursion-freeness; inflationarity; message-positivity; static message sending; and message-boundedness, one obtains

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

decidability in NEXPTIME of diffluence of relational transducer networks implemented by unions of conjunctive queries with negation (and nonequalities). In fact, the problem turns out to be complete for NEXPTIME.

As already mentioned in the Introduction, 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 nodeset. It might be interesting to decide if for a given transducer Π, all transducer networks are confluent where Π is replicated on all nodes [8].

Regarding expressivity, the techniques of the upper bound can transform a given confluent simple transducer network to a query description in 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 can be considered to be some normal form. It might be interesting to describe the smallest size that the normal form could have in relationship to the original network.

There seem to be several reasonable ways to formalize the intuitive notion of eventual consistency. In contrast to our current formalization, a stronger view of eventual consistency [1, 8] is to require that on every input, all infinite "fair" runs produce the same set of output facts. Again, a number of reasonable fairness conditions could be considered here; a rather standard one would be to require that every node performs a transition infinitely often, and that every sent message is eventually delivered. When a transducer network is eventually consistent in this stronger sense, it is also in the confluence sense of this paper, but the other implication is not necessarily true. Indeed, confluence 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. Clearly, the choice of fairness notion plays an important role. Since eventual consistency is indeed meant to be a very weak guarantee [24], it deserves further research to better understand the relationship between eventual consistency and fairness requirements. Further results on this matter are provided in follow-up work [6].

There also seems to be a pragmatic lesson: although confluence is an interesting property to guarantee for a network, the cost of automatically deciding it might be too high. Automatically deciding confluence of distributed programs is, of course, not the only way of guaranteeing confluence. Other approaches guarantee confluence by syntactic limitations [4, 19], or by focusing on semantic classes of programs that are confluent without expensive coordination [7, 8, 25].

Appendix A: Undecidability Results

A.1 Proof of Proposition 1

Inspired by the work of Deutsch et al. [11, 14], we reduce the finite implication problem for functional and inclusion dependencies to the diffluence decision problem.

Section A.1.1 provides notations for dependencies. Next, Section A.1.2 contains the technical description of the reduction. The correctness is shown in Section A.1.3.

A.1.1 Dependencies

We introduce notations for dependencies. Let $\mathcal{D}$ be a database schema, and let $R^{(k)} \in \mathcal{D}$. A *functional dependency* σ *over* R is a tuple $(R, \bar{a}, b)$, where $\bar{a}$ is a subsequence of $[1, \ldots, k]$ and $b \in \{1, \ldots, k\}$. This dependency holds for a database instance I over $\mathcal{D}$ if for any pair of facts in I, if they have the same values on components $\bar{a}$ then they have the same value on component b.

Let $R^{(k)}$ and $S^{(l)}$ be relations in $\mathcal{D}$. An *inclusion dependency* σ *from* R *to* S is a tuple $(R, \bar{a}, S, \bar{b})$, where $\bar{a}$ and $\bar{b}$ are subsequences of $[1, \ldots, k]$ and $[1, \ldots, l]$ respectively, and $\bar{a}$ and $\bar{b}$ have the same length. Denoting $\bar{a} = [a_1, \ldots, a_m]$ and $\bar{b} = [b_1, \ldots, b_m]$, this dependency holds for a database instance I over $\mathcal{D}$ if

$$\{(u_{a_1}, \ldots, u_{a_m}) | R(u_1, \ldots, u_k) \in I\} \subseteq \{(v_{b_1}, \ldots, v_{b_m}) | S(v_1, \ldots, v_l) \in I\}.$$

A.1.2 Transducer Network Construction

Let $(\mathcal{D}, \Sigma, \sigma)$ be an instance of the finite implication problem. We create a single-node transducer network $\mathcal{N}$ that is simple except that send rules don't have to be static and such that $\mathcal{N}$ is diffluent iff $(\mathcal{D}, \Sigma, \sigma)$ is not valid.

The syntactical simplifications of Section 4.1 are applied.

Abbreviate $\Sigma' = \Sigma \cup \{\sigma\}$. Let Υ be the transducer schema of Π. We define $\Upsilon_{\text{in}} = \mathcal{D} \cup \{A^{(1)}\}$ where A is a new relation name not yet occurring in $\mathcal{D}$. Relation A is used to cause inconsistencies. We define $\Upsilon_{\text{out}} = \{T^{(1)}\}$. We introduce the message and memory relations of Υ while we describe the rules of Π below.

We construct Π to be recursion-free; so $\mathcal{N}$ is also globally recursion-free. Moreover, the output and memory rules will be message-bounded and all rules are message-positive. We only add rules to insert memory facts, making Π inflationary.

Send Input First, Π sends all input facts to itself. This helps satisfy the message-boundedness restriction. So, for each relation $R^{(k)} \in \mathcal{D}$, we have a rule:

$$R_{\text{msg}}(\mathtt{u}_1, \ldots, \mathtt{u}_k) \leftarrow R(\mathtt{u}_1, \ldots, \mathtt{u}_k).$$

Projecting To check violations of Σ', received input messages are projected onto auxiliary memory relations.

Let $\tau \in \Sigma'$ be a functional dependency. Denote $\tau = (R, \bar{a}, b)$. We add a memory relation $R_\tau^{(l)}$ where l is the length of $\bar{a}$ plus 1 (for b). On receipt of an R_{msg}-fact, we project components $\bar{a}$ and b to R_τ, with $\bar{a}$ placed (in order) before b. This can be done in a message-bounded manner (details omitted).

Let $\tau \in \Sigma'$ be an inclusion dependency. Denote $\tau = (R, \bar{a}, S, \bar{b})$. We add two memory relations $R_\tau^{(m)}$ and $S_\tau^{(m)}$, where m is the length of $\bar{a}$ and $\bar{b}$. On receipt of an R_{msg}- and S_{msg}-fact, we project the components $\bar{a}$ and $\bar{b}$ (in order) to the relations R_τ and S_τ respectively. Again, this can be done in a message-bounded manner.

Checking The above auxiliary memory relations depend on message delivery, but we don't know when all input facts have been delivered. For this purpose we introduce a special marker message $\texttt{datadone}^{(0)}$. We unconditionally send it in every transition, with the rule

$$\texttt{datadone}() \leftarrow .$$

On receipt of datadone(), we create a snapshot of the input facts. We check dependencies only once in this snapshot, by using the memory relation $\texttt{checkdone}^{(0)}$, which is filled by the rule

$$\texttt{checkdone}() \leftarrow \texttt{datadone}().$$

To actually check dependencies, we proceed as follows. Let $\tau \in \Sigma'$ be a functional dependency. Denote $\tau = (R, \bar{a}, b)$. We send message $\texttt{viol}_\tau()$ if τ is violated in the snapshot, where $k = |\bar{a}|$:

$$\begin{aligned}\texttt{viol}_\tau() \leftarrow\ & R_\tau(\texttt{a}_1, \ldots, \texttt{a}_k, \texttt{b}),\ R_\tau(\texttt{a}_1, \ldots, \texttt{a}_k, \texttt{b}'),\ \texttt{b} \neq \texttt{b}',\\ & \texttt{datadone}(),\ \neg\texttt{checkdone}().\end{aligned}$$

Now, let $\tau \in \Sigma'$ be an inclusion dependency. Denote $\tau = (R, \bar{a}, S, \bar{B})$. We send message $\texttt{viol}_\tau()$ if τ is violated in the snapshot, where $m = |\bar{a}| = |\bar{b}|$:

$$\begin{aligned}\texttt{viol}_\tau() \leftarrow\ & R_\tau(\texttt{a}_1, \ldots, \texttt{a}_m),\ \neg S_\tau(\texttt{a}_1, \ldots, \texttt{a}_m),\\ & \texttt{datadone}(),\ \neg\texttt{checkdone}().\end{aligned}$$

Diffluent Behavior We cause diffluent behavior if σ is violated and Σ is not. First, we (unconditionally) send A_{msg}-facts, based on the input A-facts:

$$A_{\text{msg}}(\texttt{u}) \leftarrow A(\texttt{u}).$$

Received A_{msg}-facts are copied to output relation T while new memory relation $\texttt{blocked}^{(0)}$ is empty:

$$T(\texttt{u}) \leftarrow A_{\text{msg}}(\texttt{u}),\ \neg\texttt{blocked}().$$

Blocking is triggered by the violation of σ:

$$\texttt{blocked}() \leftarrow \texttt{viol}_\sigma().$$

So, if σ is violated, diffluence can be caused by varying the delivery order of A_{msg}-facts and $\texttt{viol}_\sigma()$. But we want to remove the diffluence if any $\tau \in \Sigma$ turns out to be violated as well, by adding this output rule:

$$T(\texttt{u}) \leftarrow A_{\text{msg}}(\texttt{u}),\ \texttt{repair}().$$

Here, $\texttt{repair}^{(0)}$ is a new memory relation that becomes enabled when Σ is violated, denoting $\Sigma = \{\tau_1, \ldots, \tau_n\}$:

$$\begin{aligned}\texttt{repair}() &\leftarrow \texttt{viol}_{\tau_1}().\\ &\vdots\\ \texttt{repair}() &\leftarrow \texttt{viol}_{\tau_n}().\end{aligned}$$

A.1.3 Correctness

Let $(\mathcal{D}, \Sigma, \sigma)$ be as above. Let $\mathcal{N}$ denote the constructed transducer network.

First Direction Suppose $(\mathcal{D}, \Sigma, \sigma)$ is not valid. There is an instance I over $\mathcal{D}$ such that $I \models \Sigma$ and $I \not\models \sigma$. We give $\mathcal{N}$ the input $J = I \cup \{A(a)\}$ and we obtain diffluence as follows.

In a first run $\mathcal{R}_1$, the message $A_{\text{msg}}(a)$ is sent during the first transition, and in the second transition we deliver only this message, causing the output fact $T(a)$ to be derived.

In a second run $\mathcal{R}_2$, we do not deliver $A_{\text{msg}}(a)$. Instead, in $\mathcal{R}_2$ we send and deliver all input facts of I, after which we deliver `datadone()`. Now, message $\texttt{viol}_\sigma()$ is sent because $I \not\models \sigma$. We deliver this message, causing `blocked()` to be derived. This completes the construction of $\mathcal{R}_2$. Run $\mathcal{R}_2$ produces no output because $A_{\text{msg}}(a)$ is not delivered. Next, no extension of $\mathcal{R}_2$ can deliver $\texttt{viol}_\tau()$ for some $\tau \in \Sigma$ because $I \models \Sigma$. Hence, `repair()` can not be derived. So, `blocked()` prevents $T(a)$ from being derived whenever $A_{\text{msg}}(a)$ would be delivered.

Second Direction For the other direction, suppose that $\mathcal{N}$ is diffluent. There is an input J for $\mathcal{N}$, and two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\mathcal{N}$ on J, such that $\mathcal{R}_1$ derives an output fact $T(a)$ and $\mathcal{R}_2$ does not, and neither can $T(a)$ be derived in any extension of $\mathcal{R}_2$. We show there is a subset $I \subseteq J|_{\mathcal{D}}$ such that $I \models \Sigma$ and $I \not\models \sigma$, so that $(\mathcal{D}, \Sigma, \sigma)$ is not valid.

First, the derivation of $T(a)$ in $\mathcal{R}_1$ implies that $A_{\text{msg}}(a)$ can be sent in $\mathcal{R}_1$. Hence, $A_{\text{msg}}(a)$ can be sent in $\mathcal{R}_2$ and in extensions thereof. Therefore, what is preventing $T(a)$ from being derived in extensions of $\mathcal{R}_2$ is the presence of `blocked()` and the absence of `repair()`. The fact `blocked()` was derived by the delivery of $\texttt{viol}_\sigma()$. This delivery must have happened inside $\mathcal{R}_2$ because otherwise in some extension of $\mathcal{R}_2$ we could postpone the delivery of $\texttt{viol}_\sigma()$ until after $A_{\text{msg}}(a)$ was delivered, deriving $T(a)$, which is impossible in any extension of $\mathcal{R}_2$.

The sending of $\texttt{viol}_\sigma()$ implies that `datadone()` was delivered in some transition i of $\mathcal{R}_2$, and at moment the transducer had received a snapshot $I \subseteq J|_{\mathcal{D}}$ such that $I \not\models \sigma$. Also, because `repair()` was not derived in $\mathcal{R}_2$ and can not be derived in an extension, it must be that no $\texttt{viol}_\tau()$-fact was ever sent for any $\tau \in \Sigma$. So, in transition i of $\mathcal{R}_2$, we have $I \models \Sigma$.

A.2 Proof of Proposition 2

Let (U, V) be an instance of the Post correspondence problem. Denote $U = u_1, \ldots, u_n$ and $V = v_1, \ldots, v_n$. We construct a single-node transducer network $\mathcal{N}$ that is simple except that local message recursion is allowed, such that (U, V) has a match iff $\mathcal{N}$ is diffluent.

A.2.1 Notations

For a word w and an index $k \in \{1, \ldots, |w|\}$, we write $w[k]$ to denote the symbol of w at position k.

A.2.2 Transducer Network Construction

We now define the single transducer Π of $\mathcal{N}$ and its transducer schema Υ. The syntactical simplifications of Section 4.1 are applied.

Represent Words For each $i \in \{1, \ldots, n\}$, we add to Υ_{in} unary relations U_k^i and V_l^i with $k \in \{1, \ldots, |u_i|\}$ and $l \in \{1, \ldots, |v_i|\}$. Now, the words u_i and v_i can be encoded. To illustrate, $u_i = aba$ is represented by the facts $\{U_l^i(a), U_2^i(b), U_3^i(a)\}$.

To represent a word-structure with arbitrary length, we provide Υ_{in} with the input relations $R^{(2)}$, $L^{(2)}$ and $F^{(1)}$. Here, L and F respectively stand for "label" and "first". For instance, the word abc might be represented as the facts $\{R(1,2), R(2,3), L(1,a), L(2,b), L(3,c), F(1)\}$. The word a can be represented by $\{F(1), L(1,a)\}$.

We send `error()` whenever the previous input relations violate the following natural constraints:

- all relations U_k^i and V_l^j contain at most one symbol; for each pair u_i and v_j, and each $k \in \{1, \ldots, |u_i|\}$ and $l \in \{1, \ldots, |v_j|\}$, the relations U_k^i and V_l^j contain a different symbol iff $u_i[k] \neq v_j[l]$; similarly for pairs of two U-words or two V-words;
- relation R contains only chains; relation F designates at most one start element; each element on the chain has at most one label.

We omit the details of the rules to check these constraints.

Alignment We search a match for (U, V) by aligning (u_i, v_i)-pairs against the input word-structure. Let $i \in \{1, \ldots, n\}$. To align the single pair (u_i, v_i), we use the following binary message relations:

- relations $\texttt{align}[i, k, k]$ with $1 \leq k \leq \min(|u_i|, |v_i|)$ to represent simultaneous alignment, one character at a time;
- relations $\texttt{align}[i, k, |v_i|]$ with $|v_i| + 1 \leq k \leq |u_i|$ to continue aligning u_i when v_i has reached its end;
- relations $\texttt{align}[i, |u_i|, k]$ with $|u_i| + 1 \leq k \leq |v_i|$ to continue aligning v_i when u_i has reached its end.

Next, we have the *start rule*, to start aligning at the beginning of the word-structure:

$$\texttt{align}[i, 1, 1](\mathtt{a}, \mathtt{a}) \leftarrow F(\mathtt{a}),\ L(\mathtt{a}, \mathtt{c}),\ U_1^i(\mathtt{c}),\ V_1^i(\mathtt{c}).$$

Then we have *simultaneous continuation rules* for each k satisfying $1 \leq k \leq \min(|u_i|, |v_i|) - 1$:

$$\texttt{align}[i, k+1, k+1](\mathtt{a}', \mathtt{b}') \leftarrow \texttt{align}[i, k, k](\mathtt{a}, \mathtt{b}),\ R(\mathtt{a}, \mathtt{a}'),\ R(\mathtt{b}, \mathtt{b}'),\ L(\mathtt{a}', \mathtt{c}_1),\ L(\mathtt{b}', \mathtt{c}_2),\ U_{k+1}^i(\mathtt{c}_1),\ V_{k+1}^i(\mathtt{c}_2).$$

We have *separate continuation rules* for u_i, for each k satisfying $|v_i| \leq k \leq |u_i| - 1$:

$$\texttt{align}[i, k+1, |v_i|](\mathtt{a}', \mathtt{b}) \leftarrow \texttt{align}[i, k, |v_i|](\mathtt{a}, \mathtt{b}),\ R(\mathtt{a}, \mathtt{a}'),\ L(\mathtt{a}', \mathtt{c}),\ U_{k+1}^i(\mathtt{c}).$$

Similarly, we have *separate continuation rules* for v_i, for each k satisfying $|u_i| \leq k \leq |v_i| - 1$:

$$\texttt{align}[i, |u_i|, k{+}1](\mathtt{a}, \mathtt{b}') \leftarrow \texttt{align}[i, |u_i|, k](\mathtt{a}, \mathtt{b}), R(\mathtt{b}, \mathtt{b}'), L(\mathtt{b}', \mathtt{c}), V^i_{k+1}(\mathtt{c}).$$

Lastly, once u_i and v_i are both fully aligned, for each pair (u_j, v_j) with $j \in \{1, \ldots, n\}$ we have the *switch rule* from pair i to pair j (with possibly $i = j$):

$$\begin{aligned}\texttt{align}[j, 1, 1](\mathtt{a}', \mathtt{b}') \leftarrow\ & \texttt{align}[i, |u_i|, |v_i|](\mathtt{a}, \mathtt{b}),\ R(\mathtt{a}, \mathtt{a}'),\ R(\mathtt{b}, \mathtt{b}'),\\ & L(\mathtt{a}', \mathtt{c}_1),\ L(\mathtt{b}', \mathtt{c}_2), U^j_1(\mathtt{c}_1), V^j_1(\mathtt{c}_2).\end{aligned}$$

Diffluent Behavior Diffluence is obtained in a similar fashion as in Section A.1. We add input relation $A^{(1)}$ and message relation $A^{(1)}_{\text{msg}}$, and a sending rule:

$$A_{\text{msg}}(\mathtt{u}) \leftarrow A(\mathtt{u}).$$

We also have an output relation $T^{(1)}$ to which received A_{msg}-facts are copied while a memory relation `blocked()` is nonempty:

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

Now, whenever we receive a message of the form $\texttt{align}[i, |u_i|, |v_i|](a, a)$, we have been able to successfully align a sequence of (u_i, v_i)-pairs to the input word-structure, so that the U- and V-side end at the same position. This corresponds to a match for (U, V). For each $i \in \{1, \ldots n\}$, add the memory insertion rule:

$$\texttt{blocked}() \leftarrow \texttt{align}[i, |u_i|, |v_i|](\mathtt{a}, \mathtt{a}).$$

Note that these rules are message-bounded. So, diffluence is obtained by varying the delivery order of A_{msg}-facts and such alignment-messages. Inconsistencies are repaired when `error()` is received (together with A_{msg}-facts):

$$T(\mathtt{u}) \leftarrow A_{\text{msg}}(\mathtt{u}),\ \texttt{error}().$$

A.2.3 Correctness

Let (U, V) be an instance of the Post correspondence problem. Let $\mathcal{N}$ be the constructed transducer network.

First Direction Suppose (U, V) has a match $E = e_1, \ldots, e_m$. Diffluence of $\mathcal{N}$ is obtained as follows. Denote $w = u_{e_1} \ldots u_{e_m}$ (or equivalently $w = v_{e_1} \ldots v_{e_m}$). We can naturally encode (U, V) and w (as the word-structure) over the input relations. This results in an instance J on which `error()` can not be sent. We give $I = J \cup \{A(a)\}$ as input to $\mathcal{N}$.

In a first run $\mathcal{R}_1$ on I, we immediately send and deliver $A_{\text{msg}}(a)$, causing $T(a)$ to be derived. In a second run $\mathcal{R}_2$, we do not deliver $A_{\text{msg}}(a)$, but, following sequence E, we send messages to align pairs of (U, V) to the encoding of w. Abbreviating $z = e_m$, and assuming the chain in the word-structure consists of consecutive natural numbers starting at 1, at some point we send $\texttt{align}[z, |u_z|, |v_z|](|w|, |w|)$. Upon delivering this message in $\mathcal{R}_2$, we derive `blocked()`. Because `error()` can not be sent, $T(a)$ can not be derived in any extension of $\mathcal{R}_2$.

Second Direction Suppose that $\mathcal{N}$ is diffluent. We show that (U, V) has a match. There is an input I for $\mathcal{N}$ and two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $\mathcal{R}_1$ derives an output fact $T(a)$ that is not derived in $\mathcal{R}_2$ or any extensions thereof. The presence of $T(a)$ in $\mathcal{R}_1$ implies that $A_{\text{msg}}(a)$ can be delivered in $\mathcal{R}_1$. So, $A_{\text{msg}}(a)$ can also be delivered in extensions of $\mathcal{R}_2$. The reason why $T(a)$ can not be derived in such extensions is the presence of `blocked()` and because `error()` can never be sent. Fact `blocked()` must have been derived in $\mathcal{R}_2$ itself, by delivering a message of the form `align`$[i, |u_i|, |v_i|](a, a)$.[13]

By going over the derivation history of `align`$[i, |u_i|, |v_i|](a, a)$ in a forward manner, we obtain a sequence $E = e_1, \ldots, e_m$ of indices in $\{1, \ldots, n\}$ by looking at the used start- or switch-rules. Sequence E is a match, because the absence of `error()` implies that the alignment of the U-words "sees" the same word-structure as the alignment of the V-words. This would not be the case, for instance, when an element of the word-structure could have two labels or when the other natural constraints on the input are violated.

Appendix B: Small Model Property

B.1 Details of Section 5.3

Let $\mathcal{R}$ be a run of $\mathcal{N}$ on input I. We construct $hist_{\mathcal{R}}$ and $msg_{\mathcal{R}}$ such that the properties 1, 2, and 3 of Section 5.3 are satisfied. Let n be the number of transitions of $\mathcal{R}$. For each $i \in \{1, \ldots, n+1\}$, we denote the ith configuration of $\mathcal{R}$ as $\rho_i = (s_i^{\mathcal{R}}, b_i^{\mathcal{R}})$. For a transition i, we denote the multiset of delivered messages and the set of sent messages respectively as $m_i^{\mathcal{R}}$ and $\delta_i^{\mathcal{R}}$.

We will perform the construction backwards, starting in the last transition of $\mathcal{R}$. Inductively, for each transition $j = n, n-1, \ldots, 1$, we define $hist_j^{\mathcal{R}}$ and $msg_{\mathcal{R}}^j$, where, intuitively, $hist_{\mathcal{R}}^j$ and $msg_{\mathcal{R}}^j$ say something about the C-facts and their needed messages for transition j and later. In the end, we define $hist_{\mathcal{R}} = hist_{\mathcal{R}}^1$ and $msg_{\mathcal{R}} = msg_{\mathcal{R}}^1$. For each pair of transitions j and i, $hist_{\mathcal{R}^j}$ and $msg_{\mathcal{R}}^j$ give rise to the (multi)sets γ_i^j, β_i^j, and $\mathcal{E}_i^j$, defined as in Section 5.3.2. By induction on j, we want the following properties to be satisfied:

1. $\gamma_i^j \sqsubseteq b_i^{\mathcal{R}}$ for each transition index i;
2. β_i^j is a set for each transition index i;
3. $\mathcal{E}_i^j = \gamma_{i+1}^j \cap \delta_i^{\mathcal{R}}$ for each transition index i; and,
4. $hist_{\mathcal{R}}^j$ contains only derivation pairs for transitions j and later.

To allow for a simple base case, we start the inductive construction at $j = n+1$ and we define $hist_{\mathcal{R}}^{n+1} = \emptyset$ (no mappings) and $msg_{\mathcal{R}}^{n+1} = \emptyset$. The induction properties

[13] If `blocked()` would not be derived in $\mathcal{R}_2$ itself, we could simply extend $\mathcal{R}_2$ by delivering $A_{\text{msg}}(a)$, upon which $T(a)$ would be derived.

are satisfied for the base case. For the induction hypothesis, we assume that $hist_{\mathcal{R}}^{j+1}$ and $msg_{\mathcal{R}}^{j+1}$ are defined such that the properties are satisfied.

B.1.1 Extend Derivation History

We define $hist_{\mathcal{R}}^{j}$ to be $hist_{\mathcal{R}}^{j+1}$ extended with an assignment of a derivation pair (φ, V) to each pair $(j, \boldsymbol{g})$ where $\boldsymbol{g}$ is either (i) an output or memory C-fact created during transition j of $\mathcal{R}$, or (ii) a needed message such that $(j, \boldsymbol{g}, l) \in msg_{\mathcal{R}}^{j+1}$ for some l. Note that $hist_{\mathcal{R}}^{j}$ is a function because there are no derivation pairs for transition j in $hist_{\mathcal{R}}^{j+1}$.

Now we define $msg_{\mathcal{R}}^{j}$ as an extension of $msg_{\mathcal{R}}^{j+1}$. Let β be the set of all messages positively needed by the selected derivation pairs in $hist_{\mathcal{R}}^{j}$ for transition j. For each $\boldsymbol{g} \in \beta$, we will select an origin transition k of $\boldsymbol{g}$, and the resulting triple $(k, \boldsymbol{g}, j)$ is added to $msg_{\mathcal{R}}^{j}$. There are two cases:

- If there is no triple $(k_0, \boldsymbol{g}, l) \in msg_{\mathcal{R}}^{j+1}$ with $k_0 < j$ then we define k to be the largest transition index of $\mathcal{R}$ for which $k < j$ and $\boldsymbol{g} \in \delta_k^{\mathcal{R}}$;
- Otherwise, let k_0 be the smallest transition of $\mathcal{R}$ for which $(k_0, \boldsymbol{g}, l) \in msg_{\mathcal{R}}^{j+1}$ and $k_0 < j$. Then we can apply Claim B.1 to know $num(\boldsymbol{g}, \gamma_{k_0}^{j+1}) < num(\boldsymbol{g}, b_{k_0}^{\mathcal{R}})$. So, intuitively, we have some instance of $\boldsymbol{g}$ in $b_{k_0}^{\mathcal{R}}$ that is not yet used in $msg_{\mathcal{R}}^{j+1}$. We now define k as the largest transition index of $\mathcal{R}$ for which $k < k_0$ and $\boldsymbol{g} \in \delta_k^{\mathcal{R}}$.

B.1.2 Show Induction Properties

We show that the induction properties are satisfied. First, $hist_{\mathcal{R}}^{j}$ by construction only contains derivation pairs for transitions j and later. Now we show the properties for $msg_{\mathcal{R}}^{j}$. Because we have added triples only for facts in β to $msg_{\mathcal{R}}^{j}$ with respect to $msg_{\mathcal{R}}^{j+1}$, it is sufficient to focus on one $\boldsymbol{g} \in \beta$. Let k be the transition index such that $(k, \boldsymbol{g}, j) \in msg_{\mathcal{R}}^{j}$. Let $i \in \{1, \ldots, n\}$ be an arbitrary transition index. We consider each of the properties:

Inclusion We have to show $num(\boldsymbol{g}, \gamma_i^{j}) \leq num(\boldsymbol{g}, b_i^{\mathcal{R}})$. If $i \leq k$ then $num(\boldsymbol{g}, \gamma_i^{j}) = 0$, because index k by choice is the smallest transition index of $\mathcal{R}$ for which $(k, \boldsymbol{g}, l) \in msg_{\mathcal{R}}^{j}$ for some l. If $j < i$, then $num(\boldsymbol{g}, \gamma_i^{j}) = num(\boldsymbol{g}, \gamma_i^{j+1})$ since $(k, \boldsymbol{g}, j)$ is only a delivery for transition j; thus the property is satisfied by applying the induction hypothesis.

Lastly, we consider the case $k < i \leq j$. If there is no triple $(k_0, \boldsymbol{g}, l) \in msg_{\mathcal{R}}^{j+1}$ with $k_0 < j$ then by choice of k we have $num(\boldsymbol{g}, \gamma_i^{j}) = 1$. And because $\boldsymbol{g}$ is not sent between k and j and yet $num(\boldsymbol{g}, b_j^{\mathcal{R}}) \geq 1$ (since $\boldsymbol{g} \in \beta$), it must be $num(\boldsymbol{g}, b_i^{\mathcal{R}}) \geq 1$; hence, $num(\boldsymbol{g}, \gamma_i^{j}) \leq num(\boldsymbol{g}, b_i^{\mathcal{R}})$.

Now suppose that k_0 exists. We consider the subcases $k < i \leq k_0$ and $k_0 < i \leq j$. If $k < i \leq k_0$ then $num\left(\boldsymbol{g}, \gamma_i^j\right) = 1$, and since $\boldsymbol{g}$ is not sent between k and k_0 and yet $num\left(\boldsymbol{g}, b_{k_0}^{\mathcal{R}}\right) \geq 1$ (Claim B.1), it must be $num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right) \geq 1$; hence, $num\left(\boldsymbol{g}, \gamma_i^j\right) \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right)$. If $k_0 < i \leq j$, we have $num\left(\boldsymbol{g}, \gamma_i^j\right) = num\left(\boldsymbol{g}, \gamma_i^{j+1}\right) + 1$ because $(k, \boldsymbol{g}, j) \in msg_{\mathcal{R}}^j$ is new (and $k < k_0$) and $num\left(\boldsymbol{g}, \gamma_i^{j+1}\right) < num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right)$ (Claim B.1); hence, $num\left(\boldsymbol{g}, \gamma_i^j\right) \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right)$.

Set We have to show $num\left(\boldsymbol{g}, \beta_i^j\right) \leq 1$. If $i < j$ then $num\left(\boldsymbol{g}, \beta_i^j\right) = 0$ and if $j < i$ then $num\left(\boldsymbol{g}, \beta_i^j\right) = num\left(\boldsymbol{g}, \beta_i^{j+1}\right) \leq 1$. If $i = j$ then the property is satisfied because we have selected only one k such that $(k, \boldsymbol{g}, j) \in msg_{\mathcal{R}}^j$.

Equality We have to show $num\left(\boldsymbol{g}, \mathcal{E}_i^j\right) = num\left(\boldsymbol{g}, \gamma_{i+1}^j \cap \delta_i^{\mathcal{R}}\right)$. Let k be as defined above. If $i < k$ then $num\left(\boldsymbol{g}, \mathcal{E}_i^j\right) = 0$ and $num\left(\boldsymbol{g}, \gamma_{i+1}^j\right) = 0$ because k is the smallest origin transition of $\boldsymbol{g}$ registered in $msg_{\mathcal{R}}^j$. If $j \leq i$ then $\mathcal{E}_i^j = \mathcal{E}_i^{j+1}$ and $\gamma_{i+1}^j = \gamma_{i+1}^{j+1}$ because in $msg_{\mathcal{R}}^j \backslash msg_{\mathcal{R}}^{j+1}$ we do not register the sending of messages in j. Next, we consider the case $k \leq i < j$. A first observation is that by choice of k, we have $num\left(\boldsymbol{g}, \gamma_{i+1}^j\right) \geq 1$. Hence, it suffices to show $num\left(\boldsymbol{g}, \mathcal{E}_i^j\right) = num\left(\boldsymbol{g}, \delta_i^{\mathcal{R}}\right)$. If $i = k$ then both $num\left(\boldsymbol{g}, \delta_i^{\mathcal{R}}\right) = 1$ and $num\left(\boldsymbol{g}, \mathcal{E}_i^j = 1\right)$ hold. Now only the more specific case $k < i < j$ remains, which we divide in two subcases.

If there is no triple $(k_0, \boldsymbol{g}, l) \in msg_{\mathcal{R}}^{j+1}$ with $k_0 < j$, then because $k < i < j$, by choice of k, the message $\boldsymbol{g}$ is not sent in transition i. This gives $num\left(\boldsymbol{g}, \delta_i^{\mathcal{R}}\right) = 0$. Consequently $\boldsymbol{g}$ was never registered as being sent from transition i, giving $num\left(\boldsymbol{g}, \mathcal{E}_i^j\right) = 0$, as desired.

Now suppose that k_0 exists. If $k < i < k_0$ then, again like the previous case, we have $num\left(\boldsymbol{g}, \delta_i^{\mathcal{R}}\right) = 0$ and $num\left(\boldsymbol{g}, \mathcal{E}_i^j\right) = 0$. Suppose $k_0 \leq i < j$. We have $num\left(\boldsymbol{g}, \gamma_{i+1}^{j+1}\right) \geq 1$ because $(k_0, \boldsymbol{g}, l) \in msg_{\mathcal{R}}^{j+1}$ for some l with $j < l$. Moreover, since $num\left(\boldsymbol{g}, \mathcal{E}_i^{j+1}\right) = num\left(\boldsymbol{g}, \gamma_{i+1}^{j+1} \cap \delta_i^{\mathcal{R}}\right)$ by the induction hypothesis, we obtain $num\left(\boldsymbol{g}, \mathcal{E}_i^{j+1}\right) = num\left(\boldsymbol{g}, \delta_i^{\mathcal{R}}\right)$. Lastly, we have $num\left(\boldsymbol{g}, \mathcal{E}_i^j\right) = num\left(\boldsymbol{g}, \mathcal{E}_i^{j+1}\right)$ because $k < i$. Hence, $num\left(\boldsymbol{g}, \mathcal{E}_i^j\right) = num\left(\boldsymbol{g}, \delta_i^{\mathcal{R}}\right)$.

B.1.3 Claims

Claim B.1 *Suppose we are in transition j of the inductive construction, with* $hist_{\mathcal{R}}^{j+1}$ *and* $msg_{\mathcal{R}}^{j+1}$ *already defined, satisfying the induction properties. Let* $\boldsymbol{g} \in \beta$. *Suppose there is a transition index* k_0 *of* $\mathcal{R}$ *such that* $(k_0, \boldsymbol{g}, l) \in msg_{\mathcal{R}}^{j+1}$ *and* $k_0 < j$. *Assume*

that k_0 is the smallest such index. For each transition $i \in \{j, j-1, \ldots, k_0\}$, we have $num\left(\boldsymbol{g}, \gamma_i^{j+1}\right) < num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right)$.

Proof We show this by backward induction on $i = j, j-1, \ldots, k_0$. To increase readability, we will abbreviate $j+1$ as the prime symbol $\prime$. So, γ_i^{j+1}, $\mathcal{E}_i^{j+1}$, and $msg_{\mathcal{R}}^{j+1}$ become respectively γ_i', $\mathcal{E}_i'$, and $msg_{\mathcal{R}}'$.

Base Case For the base case, $i = j$, we have to show $num\left(\boldsymbol{g}, \gamma_i'\right) < num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right)$. If we can show $num\left(\boldsymbol{g}, \gamma_i'\right) \leq num\left(\boldsymbol{g}, \gamma_{i+1}' \backslash \delta_i^{\mathcal{R}}\right)$, then by applying the induction property $\gamma_{i+1}' \sqsubseteq b_{i+1}^{\mathcal{R}}$ on $msg_{\mathcal{R}}'$, we obtain $num\left(\boldsymbol{g}, \gamma_i'\right) \leq num\left(\boldsymbol{g}, b_{i+1}^{\mathcal{R}} \backslash \delta_i^{\mathcal{R}}\right)$. And using $b_{i+1}^{\mathcal{R}} \backslash \delta_i^{\mathcal{R}} = b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}$ (by the operational semantics), we get $num\left(\boldsymbol{g}, \gamma_i'\right) \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right)$. Lastly, because $m_i^{\mathcal{R}} \sqsubseteq b_i^{\mathcal{R}}$ and $num\left(\boldsymbol{g}, m_i^{\mathcal{R}}\right) > 1$ (indeed, $\boldsymbol{g} \in \beta \sqsubseteq m_j^{\mathcal{R}} = m_i^{\mathcal{R}}$), we obtain $num\left(\boldsymbol{g}, \gamma_i'\right), < num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right)$, as desired.

We are left to show $num\left(\boldsymbol{g}, \gamma_i'\right), \leq num\left(\boldsymbol{g}, \gamma_{i+1}' \backslash \delta_i^{\mathcal{R}}\right)$. Because in $msg_{\mathcal{R}}'$ no needed messages are registered for transition j (and smaller), it must be $num\left(\boldsymbol{g}, \gamma_i'\right), = num\left(\boldsymbol{g}, \gamma_{i+1}' \backslash \mathcal{E}_i'\right)$. If we can show $num\left(\boldsymbol{g}, \mathcal{E}_i'\right) = num\left(\boldsymbol{g}, \delta_i^{\mathcal{R}}\right)$, then we are ready. It actually suffices to show $\boldsymbol{g} \in \gamma_{i+1}'$, because then $num\left(\boldsymbol{g}, \mathcal{E}_i'\right) = num\left(\boldsymbol{g}, \delta_i^{\mathcal{R}}\right)$ follows from the induction property $\mathcal{E}_i' = \gamma_{i+1}' \cap \delta_i^{\mathcal{R}}$ of $msg_{\mathcal{R}}'$.

We show $\boldsymbol{g} \in \gamma_{i+1}'$. By definition of k_0, there is a triple $(k_0, \boldsymbol{g}, l) \in msg_{\mathcal{R}}'$ for some l. Again, because in $msg_{\mathcal{R}}'$ no needed messages are registered for transition j and smaller, it must be $j < l$ or equivalently $j+1 = i+1 \leq l$. Hence, $\boldsymbol{g} \in \gamma_{i+1}'$ by definition of γ_{i+1}'.

Inductive Step For the induction hypothesis, suppose that $num\left(\boldsymbol{g}, \gamma_{i+1}'\right) < num\left(\boldsymbol{g}, b_{i+1}^{\mathcal{R}}\right)$. We show $num\left(\boldsymbol{g}, \gamma_i'\right) < num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right)$. We proceed similarly as in the base case, but the strictness "$<$" is obtained differently.

First, by definition of k_0, we have $(k_0, \boldsymbol{g}, l) \in msg_{\mathcal{R}}^{j+1}$ for some l. Like above, we have $j < l$. Hence, $k_0 \leq i < l$ or equivalently $k_0 < i+1 \leq l$ and thus $num\left(\boldsymbol{g}, \gamma_{i+1}'\right) \geq 1$. Because $\delta_i^{\mathcal{R}}$ is a set, if we can show $num\left(\boldsymbol{g}, \gamma_i'\right) \leq num\left(\boldsymbol{g}, \gamma_{i+1}' \backslash \delta_i^{\mathcal{R}}\right)$, then the induction hypothesis gives $num\left(\boldsymbol{g}, \gamma_i'\right) < num\left(\boldsymbol{g}, b_{i+1}^{\mathcal{R}} \backslash \delta_i^{\mathcal{R}}\right)$. By the operational semantics we would further obtain $num\left(\boldsymbol{g}, \gamma_i'\right) < num\left(\boldsymbol{g}, b_i^{\mathcal{R}} \backslash m_i^{\mathcal{R}}\right) \leq num\left(\boldsymbol{g}, b_i^{\mathcal{R}}\right)$, as desired.

Showing $num\left(\boldsymbol{g}, \gamma_i'\right) \leq num\left(\boldsymbol{g}, \gamma_{i+1}' \backslash \delta_i^{\mathcal{R}}\right)$ is like in the base case. □

Claim B.2 *Let $\mathcal{R}$ be a run of $\mathcal{N}$ on I. Let $hist$ and $msg_{\mathcal{R}}$ be as defined in* Section 5.3. *Let i be a transition index of $\mathcal{R}$. We have $\gamma_{i+1} = (\gamma_i \backslash \beta_i) \cup \mathcal{E}_i$ (multiset difference and union).*

Proof Let $\boldsymbol{g}$ be a fact. We show $num(\boldsymbol{g}, \gamma_{i+1}) = num(\boldsymbol{g}, (\gamma_i \backslash \beta_i) \cup \mathcal{E}_i)$.

First, $num(\boldsymbol{g}, \gamma_{i+1})$ is, by definition of γ_{i+1}, the number of triples $(j, \boldsymbol{g}, k) \in msg_{\mathcal{R}}$ for which $j < i+1$ and $i+1 \leq k$. Hence, $num(\boldsymbol{g}, \gamma_{i+1}) = e_1 + e_2$, where

– e_1 is the number of triples $(j, \boldsymbol{g}, k) \in msg_{\mathcal{R}}$ for which $j < i$ and $i+1 \leq k$, and,

– e_2 is the number of triples $(j, \boldsymbol{g}, k) \in msg_{\mathcal{R}}$ for which $j = i$ and $i + 1 \leq k$.

Regarding e_2, since always $j < k$, the equality $j = i$ already implies $i + 1 \leq k$. So, e_2 simplifies to the number of triples $(i, \boldsymbol{g}, k) \in msg_{\mathcal{R}}$, or equivalently $e_2 = num(\boldsymbol{g}, \mathcal{E}_i)$. If we would know that $e_1 = num(\boldsymbol{g}, \gamma_i \backslash \beta_i)$ then overall we would obtain, as desired:

$$\begin{aligned} num(\boldsymbol{g}, \gamma_{i+1}) &= num(\boldsymbol{g}, \gamma_i \backslash \beta_i) + num(\boldsymbol{g}, \mathcal{E}_i) \\ &= num(\boldsymbol{g}, (\gamma_i \backslash \beta_i) \cup \mathcal{E}_i). \end{aligned}$$

Now we show $e_1 = num(\boldsymbol{g}, \gamma_i \backslash \beta_i)$. Using that $i + 1 \leq k$ is equivalent to $i < k$, we have $e_1 = f_1 - f_2$, where

– f_1 is the number of triples $(j, \boldsymbol{g}, k) \in msg_{\mathcal{R}}$ for which $j < i$ and $i \leq k$, and,
– f_2 is the number of triples $(j, \boldsymbol{g}, k) \in msg_{\mathcal{R}}$ for which $j < i$ and $i = k$ (or simply $i = k$ because always $j < k$).

By definition of γ_i and β_i, we have $f_1 = num(\boldsymbol{g}, \gamma_i)$ and $f_2 = num(\boldsymbol{g}, \beta_i)$. Lastly, because $num(\boldsymbol{g}, \beta_i) \leq num(\boldsymbol{g}, \gamma_i)$, we obtain

$$\begin{aligned} e_1 &= num(\boldsymbol{g}, \gamma_i) - num(\boldsymbol{g}, \beta_i) \\ &= num(\boldsymbol{g}, \gamma_i \backslash \beta_i). \end{aligned}$$

□

B.2 Details of Section 5.4

Claim B.3 *Let the transitions of $\mathcal{S}$ be defined up to and including transition i. If $\gamma_i \sqsubseteq b_i^{\mathcal{S}}$ then $\beta_i \subseteq \left(m_i^{\mathcal{S}}\right)$.*

Proof By definition, $m_i^{\mathcal{S}} = \left(b_i^{\mathcal{S}} \backslash (\gamma_i \backslash \beta_i)\right) \cap m_i^{\mathcal{R}}$. Let $\boldsymbol{g} \in \beta_i$. It is sufficient to show that $num\left(\boldsymbol{g}, b_i^{\mathcal{S}} \backslash (\gamma_i \backslash \beta_i)\right) \geq 1$ and $num\left(\boldsymbol{g}, m_i^{\mathcal{R}}\right) \geq 1$.

We show that $num\left(\boldsymbol{g}, b_i^{\mathcal{S}} \backslash (\gamma_i \backslash \beta_i)\right) \geq 1$. It is sufficient to show $num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right) \geq 1$ and $num(\boldsymbol{g}, \gamma_i \backslash \beta_i) < num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right)$. First, because β_i is a set (property of $msg_{\mathcal{R}}$), and $\boldsymbol{g} \in \beta_i$, we have $num(\boldsymbol{g}, \beta_i) = 1$. Also, the given assumption $\gamma_i \sqsubseteq b_i^{\mathcal{S}}$ implies $num(\boldsymbol{g}, \gamma_i) \leq num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right)$.

– We show $num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right) \geq 1$. From the definition of β_i and γ_i, we have $num(\boldsymbol{g}, \beta_i) \leq num(\boldsymbol{g}, \gamma_i)$. And since $num(\boldsymbol{g}, \beta_i) = 1$ and $num(\boldsymbol{g}, \gamma_i) \leq num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right)$, we obtain $num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right) \geq 1$.
– We show $num(\boldsymbol{g}, \gamma_i \backslash \beta_i) < num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right)$. Since $num(\boldsymbol{g}, \beta_i) = 1$ and $num(\boldsymbol{g}, \beta_i) \leq num(\boldsymbol{g}, \gamma_i)$, we have $num(\boldsymbol{g}, \gamma_i \backslash \beta_i) < num(\boldsymbol{g}, \gamma_i)$. Combined with $num(\boldsymbol{g}, \gamma_i) \leq num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right)$, we obtain $num(\boldsymbol{g}, \gamma_i \backslash \beta_i) < num\left(\boldsymbol{g}, b_i^{\mathcal{S}}\right)$.

We are left to show that $num\left(\boldsymbol{g}, m_i^{\mathcal{R}}\right) \geq 1$. By definition of $\boldsymbol{g} \in \beta_i$, there is a triple $(k, \boldsymbol{g}, l) \in msg_{\mathcal{R}}$ with $l = i$. Hence, by construction of $msg_{\mathcal{R}}$, we have $num\left(\boldsymbol{g}, m_i^{\mathcal{R}}\right) \geq 1$. □

Claim B.4 *Let $\mathcal{R}$ be a run of $\mathcal{N}$ on input I. Suppose a run $\mathcal{S}$ of $\mathcal{N}$ on J has the properties that* (i) $last(\mathcal{S})$ *and* $last(\mathcal{R})$ *contain the same output and memory C-facts, and,* (ii) *the message buffer of* $last(\mathcal{S})$ *is a submultiset of the message buffer in* $last(\mathcal{R})$*. Then, for every extension $\mathcal{S}'$ of $\mathcal{S}$, there is an extension $\mathcal{R}'$ of $\mathcal{R}$ such that* $last(\mathcal{S}')$ *and* $last(\mathcal{R}')$ *again contain precisely the same output and memory C-facts.*

Proof Let $\mathcal{S}'$ be an extension of $\mathcal{S}$ that does m new transitions after those of $\mathcal{S}$, with $m \geq 1$. The idea is to extend $\mathcal{R}$ by also doing m new transitions, in each of which we do the same message deliveries as in the corresponding transition in the extension of $\mathcal{S}$. This results in run $\mathcal{R}'$.

For each $i \in \{1, \ldots, m+1\}$, let $\rho_i = \left(s_i^{\mathcal{R}}, b_i^{\mathcal{R}}\right)$ and $\sigma_i = \left(s_i^{\mathcal{S}}, b_i^{\mathcal{S}}\right)$ denote the ith configuration in the extension of respectively $\mathcal{R}$ and $\mathcal{S}$, with $\rho_1 = last(\mathcal{R})$ and $\sigma_1 = last(\mathcal{S})$. We show by induction on $i \in \{1, \ldots, m+1\}$ that *(i)* σ_i and ρ_i contain the same output and memory C-facts, and, *(ii)* the message buffer of σ_i is a submultiset of the message buffer of ρ_i. This second property helps us deliver the same messages in the extension of $\mathcal{R}$ as done in the extension of $\mathcal{S}$.

For the base case, these properties hold because $\rho_1 = last(\mathcal{R})$ and $\sigma_1 = last(\mathcal{S})$. Assuming the properties hold for configuration i with $i \geq 1$, for the inductive step we show that they can be satisfied in configuration $i+1$. Recall that transition i is responsible for transforming configuration i into configuration $i+1$. Now, in transition i of $\mathcal{R}'$ we deliver the same message multiset as in transition i of $\mathcal{S}'$, which is possible by induction property (ii).

Output and Memory We show that σ_{i+1} and ρ_{i+1} have the same output and memory C-facts. To show that the C-facts of σ_{i+1} are a subset of those in ρ_{i+1}, we can apply Claim B.6 (property 1). To show the reverse inclusion, let $\boldsymbol{g}$ be a newly derived C-fact in transition i of $\mathcal{R}'$. We show that $\boldsymbol{g}$ is also created in transition i of $\mathcal{S}'$. Let (φ, V) be a derivation pair for $\boldsymbol{g}$ in transition i of $\mathcal{R}'$. We show that V is also satisfying for φ in transition i of $\mathcal{S}'$.

- Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\gamma_{\text{in}}}$. We have to show $\boldsymbol{h} \in J$. Suppose we would know that $adom(\boldsymbol{h}) \subseteq adom(J)$. Then, since $\boldsymbol{h} \in I$ (because V is satisfying for φ in $\mathcal{R}'$) and $J = I^{[adom(J)]}$ (Claim B.5), we have $\boldsymbol{h} \in J$, as desired.

  Now we show that $adom(\boldsymbol{h}) \subseteq adom(J)$. Let $\boldsymbol{a} \in pos^{\varphi}|_{\gamma_{\text{in}}}$ be an atom such that $V(\boldsymbol{a}) = \boldsymbol{h}$. A variable $\mathtt{u}$ in $\boldsymbol{a}$ is either free or bound. If $\mathtt{u}$ is free then $V(\mathtt{u}) \in C$ because $\boldsymbol{g}$ is a C-fact, and thus $V(\mathtt{u}) \in adom(J)$ because $C \subseteq adom(K_1) \subseteq adom(J)$. Next, if $\mathtt{u}$ is bound then by message-boundedness of φ, value $V(\mathtt{u})$ occurs in a delivered message during transition i of $\mathcal{R}'$. But this message is also delivered during transition i of $\mathcal{S}'$, and because values in messages of $\mathcal{S}'$ are restricted to $adom(J)$, value $V(\mathtt{u})$ occurs in $adom(J)$.
- Let $\boldsymbol{h} \in V(neg^{\varphi})|_{\gamma_{\text{in}}}$. We have to show $\boldsymbol{h} \notin J$. This follows from $\boldsymbol{h} \notin I$ (since V is satisfying for φ in $\mathcal{R}'$) and $J \subseteq I$.
- Recall that φ is message-positive. Because V is satisfying for φ during transition i of $\mathcal{R}'$, each message $\boldsymbol{h} \in V(pos^{\varphi})|_{\gamma_{\text{msg}}}$ is delivered during that transition. By definition of the message deliveries in $\mathcal{R}'$, these messages are also delivered in transition i of $\mathcal{S}'$.

- Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\gamma_{\text{out}} \cup \gamma_{\text{mem}}}$. We have to show that $\boldsymbol{h}$ is in σ_i. Because $\boldsymbol{g}$ is a C-fact, the message-boundedness of φ implies that $\boldsymbol{h}$ is a C-fact. And because V is satisfying for φ in $\mathcal{R}'$, $\boldsymbol{h}$ is in ρ_i. By the induction hypothesis, ρ_i and σ_i have the same output and memory C-facts. Hence, $\boldsymbol{h}$ is in σ_i. Similarly we can show for each $\boldsymbol{h} \in V(neg^{\varphi})|_{\gamma_{\text{out}} \cup \gamma_{\text{mem}}}$ that $\boldsymbol{h}$ is not in σ_i.
- Because the nonequalities of φ are satisfied under V in $\mathcal{R}'$, they are also satisfied in $\mathcal{S}'$.

We conclude that V is satisfying for φ during transition i of $\mathcal{S}'$. Hence, $\boldsymbol{g} \in \sigma_{i+1}$.

Message Buffer We show $b^{\mathcal{S}}_{i+1} \sqsubseteq b^{\mathcal{R}}_{i=1}$. Let m denote the message multiset delivered in transition i. Let $\delta^{\mathcal{R}}_i$ and $\delta^{\mathcal{S}}_i$ denote the message sets sent in new transition i of $\mathcal{R}'$ and $\mathcal{S}'$ respectively. The operational semantics implies that $b^{\mathcal{R}}_{i+1} = (b^{\mathcal{R}}_i \backslash m) \cup \delta^{\mathcal{R}}_i$ and $b^{\mathcal{S}}_{i+1} = (b^{\mathcal{S}}_i \backslash m) \cup \delta^{\mathcal{S}}_i$ (multiset difference and union). The desired inclusion $b^{\mathcal{S}}_{i+1} \sqsubseteq b^{\mathcal{R}}_{i+1}$ follows from $(b^{\mathcal{S}}_i \backslash m) \sqsubseteq (b^{\mathcal{R}}_i \backslash m)$ (by the induction hypothesis) and $\delta^{\mathcal{S}}_i \subseteq \delta^{\mathcal{R}}_i$ (by Claim B.6, property 2). □

Claim B.5 *The instance J satisfies* $J = I^{[adom(J)]}$.

Proof This is because *(i)* $J \subseteq I$ implies $J \subseteq I^{[adom(J)]}$, and *(ii)*, since $adom(J) \subseteq adom(K_1) \cup adom(K_2)$, we have

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

□

Claim B.6 *Let* $\mathcal{R}$ *be a run of* $\mathcal{N}$ *on I and let* $\mathcal{S}$ *be a run of* $\mathcal{N}$ *on J. Let i and j be a transition index of respectively* $\mathcal{R}$ *and* $\mathcal{S}$*. For transition i of* $\mathcal{R}$*, let* ρ_i*,* $m^{\mathcal{R}}_i$*, and* ρ_{i+1}*, respectively denote the begin-configuration, the delivered messages, and the end-configuration. For transition j of* $\mathcal{S}$ *we similarly define* σ_j*,* $m^{\mathcal{S}}_j$*, and* σ_{j+1}*.*

Suppose that (i) ρ_i *and* σ_j *have the same output and memory C-facts, and,* (ii) $m^{\mathcal{S}}_j \sqsubseteq m^{\mathcal{R}}_i$*. The following properties hold:*

1. *The output and memory C-facts of* σ_{j+1} *are a subset of those in* ρ_{i+1}*.*
2. *The messages sent in transition j of* $\mathcal{S}$ *are a subset of those sent in transition i of* $\mathcal{R}$*.*

Proof The two properties are shown below.

Property 1 Let $\boldsymbol{g}$ be an output or memory C-fact that is newly derived during transition j of $\mathcal{S}$, by means of a derivation pair (φ, V). We show that V is also satisfying for φ during transition i of $\mathcal{R}$.

- Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\gamma_{\text{in}}}$. We have to show $\boldsymbol{h} \in I$. This follows from $\boldsymbol{h} \in J$ (since V is satisfying for φ in $\mathcal{S}$) and $J \subseteq I$ (by construction of J).
- Let $\boldsymbol{h} \in V(neg^{\varphi})|_{\gamma_{\text{in}}}$. We have to show $\boldsymbol{h} \notin I$. Since V is satisfying for φ in $\mathcal{S}$, we have $\boldsymbol{h} \notin J$. Since V can only assign values from $adom(J)$, we have

$adom(\boldsymbol{h}) \subseteq adom(J)$. So, if $\boldsymbol{h} \in I$ then $\boldsymbol{h} \in I^{[adom(J)]} = J$ (Claim B.5), which is false. Hence, $\boldsymbol{h} \notin I$.

- Recall that φ is message-positive. Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\Upsilon_{\text{msg}}}$. We have to show that $\boldsymbol{h} \in m_i^{\mathcal{R}}$. Because V is satisfying for φ in $\mathcal{S}$, we have $\boldsymbol{h} \in m_i^{\mathcal{S}} \sqsubseteq m_i^{\mathcal{R}}$.
- Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$. We have to show that $\boldsymbol{h}$ is in ρ_i. Because $\boldsymbol{g}$ is a C-fact, the message-boundedness of φ implies that $\boldsymbol{h}$ is a C-fact. Moreover, because V is satisfying for φ, fact $\boldsymbol{h}$ is a C-fact in σ_j and thus by assumption also in ρ_i.
  We can similarly show for each $\boldsymbol{h} \in V(neg^{\varphi})|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$ that $\boldsymbol{h} \notin \rho_i$.
- Lastly, because the nonequalities of φ are satisfied under V in $\mathcal{S}$, they are also satisfied under V in $\mathcal{R}$.

We obtain that V is satisfying for φ during transition i of $\mathcal{R}$. Hence, $\boldsymbol{g}$ is in ρ_{i+1}.

Property 2 Let $\boldsymbol{g}$ be a message sent in transition j of $\mathcal{S}$, by means of a derivation pair (φ, V). We show that V is also satisfying for φ during transition i of $\mathcal{R}$. Because send rules are static, we only have to reason about input and message body atoms of φ. For these body atoms, the proof of property 1 above can actually be applied verbatim to show (i) for each $\boldsymbol{h} \in V(pos^{\varphi})|_{\Upsilon_{\text{in}}}$ and $\boldsymbol{h} \in V(neg^{\varphi})|_{\Upsilon_{\text{in}}}$ that respectively $\boldsymbol{h} \in I$ and $\boldsymbol{h} \notin I$; and (ii) for each $\boldsymbol{h} \in V(pos^{\varphi})|_{\Upsilon_{\text{msg}}}$ that $\boldsymbol{h}$ is delivered in transition i of $\mathcal{R}$. □

Appendix C: Decidability

C.1 Details of Section 6.1.2

Claim C.1 *Let $\boldsymbol{f}$ be an output fact created in some run of $\mathcal{N}$ on an input I. Denote $C = adom(\boldsymbol{f})$. Let $\mathcal{R}$ be an arbitrary run of $\mathcal{N}$ on input I. There exists a run $\mathcal{S}$ of $\mathcal{N}$ on input I with at most* **runLen** *transitions and such that $last(\mathcal{S})$ contains precisely the same output and memory C-facts as $last(\mathcal{R})$.*

Proof We start by sketching the approach. Like in Section 5.3, we can "mark" the transitions where the output and memory C-facts are created, and also the transitions where any message is sent that is recursively needed by such a C-fact. This gives us the function $hist_{\mathcal{R}}$ and the set $msg_{\mathcal{R}}$ as defined there (satisfying the properties of Section 5.3.2). Since each C-fact requires at most $\mathbf{B}^{\mathbf{P}}$ messages by recursion-freeness, at most $\mathbf{C}\mathbf{B}^{\mathbf{P}} + \mathbf{C} = \mathbf{runLen}$ transitions are marked this way. The maximum would be reached if each C-fact requires a unique set of messages. Let $\mathcal{M}$ denote the marked transition indices of $\mathcal{R}$. Intuitively, the new run $\mathcal{S}$ does only the marked transitions, so $|\mathcal{M}|$ in total.

We also need some extra notations. We write $\rho_i = (s_i^{\mathcal{R}}, b_i^{\mathcal{R}})$ and $\sigma_i = (s_i^{\mathcal{S}}, b_i^{\mathcal{S}})$ to denote the begin-configuration of transition i in $\mathcal{R}$ and $\mathcal{S}$ respectively. For transition i of $\mathcal{R}$, let γ_i be as defined in Section 5.3.2, based on $msg_{\mathcal{R}}$. Denote $n = |\mathcal{M}|$. We can order the transitions of $\mathcal{M}$ in ascending order, and we write $\mathcal{M}(i)$ to denote the transition index of $\mathcal{M}$ at ordinal i in this ordering, with $i \in \{1, \ldots, n\}$. For uniformity, we define $\mathcal{M}(n+1) = n' + 1$, with n' the last transition index of $\mathcal{R}$.

Now, by induction on the configurations, we construct $\mathcal{S}$ so that each configuration index $i \in \{1, \ldots, n+1\}$ satisfies the following properties:

- $s_i^{\mathcal{S}}$ contains the same output and memory C-facts as $s^{\mathcal{R}}_{\mathcal{M}(i)}$; and,
- $\gamma_{\mathcal{M}(i)}$ is a submultiset of $b_i^{\mathcal{S}}$.

Then, the last configuration $s^{\mathcal{S}}_{n+1}$ contains the same output and memory C-facts as $s^{\mathcal{R}}_{\mathcal{M}(n+1)} = s^{\mathcal{R}}_{n'+1}$, which is the last configuration of $\mathcal{R}$, as desired. The second induction property helps in showing the first induction property.

For the base case ($i = 1$), we have $s_1^{\mathcal{S}} = \emptyset$ because σ_1 is the start configuration of $\mathcal{S}$. Moreover, $s^{\mathcal{R}}_{\mathcal{M}(1)}$ can not contain any output and memory C-facts because $\mathcal{M}(1)$ is the first marked transition, and thus the C-facts are created *in* or *after* transition $\mathcal{M}(1)$. A similar reasoning applies to needed messages: $\gamma_{\mathcal{M}(1)} = \emptyset$, which is a submultiset of $b_1^{\mathcal{S}}$.

For the induction hypothesis, we assume that the properties hold for configuration σ_i of $\mathcal{S}$, with $i \geq 1$ (and $i \leq n$). Abbreviate $j = \mathcal{M}(i)$ and let β_j be as in Section 5.3.2. We define transition i of $\mathcal{S}$ to deliver *precisely* set β_j. Note that we can deliver β_j because $\gamma_j \sqsubseteq b_i^{\mathcal{S}}$ (induction hypothesis) and $\beta_j \sqsubseteq \gamma_j$ (follows from their definition).[14] We now show that the induction properties are satisfied for configuration σ_{i+1}.

Output and Memory Abbreviate $k = \mathcal{M}(i+1)$. We have to show that $s^{\mathcal{S}}_{i+1}$ and $s^{\mathcal{R}}_k$ contain the same output and memory C-facts. We have $j < k$ (because $\mathcal{M}(i) < \mathcal{M}(i+1)$). Also, there are no other marked transitions between j and k, so no new output and memory C-facts are created between j and k. Finally, inflationarity implies that $s^{\mathcal{R}}_{j+1}$ and $s^{\mathcal{R}}_k$ contain precisely the same output and memory C-facts. Hence, it is sufficient to show that $s^{\mathcal{S}}_{i+1}$ and $s^{\mathcal{R}}_{j+1}$ contain the same output and memory C-facts.

First, let $\boldsymbol{g}$ be an output or memory C-fact in $s^{\mathcal{S}}_{i+1}$. We show that $\boldsymbol{g} \in s^{\mathcal{R}}_{j+1}$. If $\boldsymbol{g} \in s_i^{\mathcal{S}}$ then by the induction hypothesis $\boldsymbol{g} \in s_j^{\mathcal{R}} \subseteq s^{\mathcal{R}}_{j+1}$. Now suppose $\boldsymbol{g} \in s^{\mathcal{S}}_{i+1} \setminus s_i^{\mathcal{S}}$. Let (φ, V) be a derivation pair for $\boldsymbol{g}$ in transition i of $\mathcal{S}$. We show that V is also satisfying for φ in transition j of $\mathcal{R}$.

- Since $\mathcal{S}$ and $\mathcal{R}$ are given the same input, the input literals in the body of φ are satisfied under V in transition j of $\mathcal{R}$ as well.
- Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\gamma_{\text{msg}}}$. Since V is satisfying for φ in transition i of $\mathcal{S}$, it must be $\boldsymbol{h} \in \beta_j$. By construction of $msg_{\mathcal{R}}$, the set β_j is delivered in transition j of $\mathcal{R}$, as desired.
- Since $s_i^{\mathcal{S}}$ and $s_j^{\mathcal{R}}$ contain the same output and memory C-facts (induction hypothesis), message-boundedness of φ implies that the output and memory literals of φ are satisfied under V in transition j of $\mathcal{R}$.
- Finally, the nonequalities of φ under V are also satisfied in transition j of $\mathcal{R}$ because they are satisfied in transition i of $\mathcal{S}$.

[14] We deliver no more than β_j to avoid unwanted fact derivations.

Let $\boldsymbol{g}$ be an output or memory C-fact in $s^{\mathcal{R}}_{j+1}$. Similarly to the above, if $\boldsymbol{g} \in s^{\mathcal{R}}_{j}$ then by the induction hypothesis $\boldsymbol{g} \in s^{\mathcal{S}}_{i} \subseteq s^{\mathcal{S}}_{i+1}$. Because $\boldsymbol{g}$ is an output or memory C-fact, the mapping $hist_{\mathcal{R}}(j, \boldsymbol{g}) = (\varphi, V)$ is defined. We show that V is also satisfying for φ in transition i of $\mathcal{S}$. The reasoning for nonequalities and input, output, and memory literals of φ is the same as above for the case $\boldsymbol{g} \in s^{\mathcal{S}}_{i+1} \backslash s^{\mathcal{S}}_{i}$. Let $\boldsymbol{h} \in V(pos^{\varphi})|_{\gamma_{\text{msg}}}$. Then $\boldsymbol{h}$ is a message needed by (φ, V), and thus $\boldsymbol{g} \in \beta_j$ by construction of $msg_{\mathcal{R}}$. Hence, $\boldsymbol{h}$ is delivered in transition i of $\mathcal{S}$.

Buffer We have to show $\gamma_{\mathcal{M}(i+1)} \sqsubseteq b^{\mathcal{S}}_{i+1}$. Abbreviate $j = \mathcal{M}(i)$ and $k = \mathcal{M}(i+1)$. We have $j + 1 \leq k$ because $j < k$. We start by showing $\gamma_{j+1} = \gamma_k$, so it becomes sufficient to show $\gamma_{j+1} \sqsubseteq b^{\mathcal{S}}_{i+1}$.

Let $\boldsymbol{g}$ be a fact. We show $num(\boldsymbol{g}, \gamma_{j+1}) \leq num(\boldsymbol{g}, \gamma_k)$. By definition of γ_{j+1}, expression $num(\boldsymbol{g}, \gamma_{j+1})$ is the number of triples $(a, \boldsymbol{g}, b) \in msg_{\mathcal{R}}$ for which $a < j + 1 \leq b$. Let $(a, \boldsymbol{g}, b)$ be such a triple. It is sufficient to show that $a < k \leq b$. We have $a < k$ because $a < j + 1$ and $j + 1 \leq k$. Secondly, if $b < k$ then a needed message is delivered at transition b of $\mathcal{R}$, implying $b \in \mathcal{M}$, which is impossible because $j < b < k$ and there are no marked transitions between j and k. Hence, $k \leq b$.

Let $\boldsymbol{g}$ be a fact. We show $num(\boldsymbol{g}, \gamma_k) \leq num(\boldsymbol{g}, \gamma_{j+1})$. This is similar to the previous direction, but there are also some differences. By definition of γ_k, expression $num(\boldsymbol{g}, \gamma_k)$ is the number of triples $(a, \boldsymbol{g}, b) \in msg_{\mathcal{R}}$ for which $a < k \leq b$. Let $(a, \boldsymbol{g}, b)$ be such a triple. It is sufficient to show that $a < j + 1 \leq b$. We have $j + 1 \leq b$ because $j + 1 \leq k$ and $k \leq b$. Secondly, if $j + 1 \leq a$ then a needed message would be sent at transition a of $\mathcal{R}$, implying $a \in \mathcal{M}$, which is impossible because $j < a < k$ and there are no marked transitions between j and k. Hence, $a < j + 1$.

Lastly, we show that $\gamma_{j+1} \sqsubseteq b^{\mathcal{S}}_{i+1}$. Using Claim B.2, we have $\gamma_{j+1} = (\gamma_j \backslash \beta_j) \cup \mathcal{E}_j$. Let $\delta^{\mathcal{S}}_i$ denote the set of messages sent during transition i of $\mathcal{S}$. The operational semantics implies $b^{\mathcal{S}}_{i+1} = (b^{\mathcal{S}}_i \backslash \beta_j) \cup \delta^{\mathcal{S}}_i$. It is sufficient to show $\gamma_j \backslash \beta_j \sqsubseteq b^{\mathcal{S}}_i \backslash \beta_j$ and $\mathcal{E}_j \subseteq \delta^{\mathcal{S}}_i$. The first inclusion follows from the induction hypothesis $\gamma_j \sqsubseteq b^{\mathcal{S}}_i$. Now, let $\boldsymbol{g} \in \mathcal{E}_j$. We show $\boldsymbol{g} \in \delta^{\mathcal{S}}_i$. By definition of $\mathcal{E}_j$, there is a triple $(j, \boldsymbol{g}, b) \in msg_{\mathcal{R}}$. So, $\boldsymbol{g}$ is a needed message that should be sent in transition j of $\mathcal{R}$. Hence, $hist_{\mathcal{R}}(j, \boldsymbol{g}) = (\varphi, V)$ is defined. We show that V is satisfying for φ during transition i of $\mathcal{S}$, so that $\boldsymbol{g} \in \delta^{\mathcal{S}}_i$. Because φ is static, we only consider the input and message literals, where the latter are positive by message-positivity. The input literals of φ are satisfied under V in transition i of $\mathcal{S}$, because they are satisfied in transition j of $\mathcal{R}$ and because both runs have the same input. Now, let $\boldsymbol{h} \in V(pos^{\varphi})|_{\gamma_{\text{msg}}}$. We have to show that $\boldsymbol{h}$ is delivered in transition i of $\mathcal{S}$. Because $\boldsymbol{h}$ is delivered in transition j of $\mathcal{R}$ (since V is satisfying for φ), $\boldsymbol{h}$ is a needed message for transition j; hence, $\boldsymbol{h} \in \beta_j$ and this set is delivered in transition i of $\mathcal{S}$. □

Claim C.2 *Let I be an input for $\mathcal{N}$. Let $\mathcal{R}$ be a run of $\mathcal{N}$ on I. Let $\mathcal{R}'$ be $\mathcal{R}$ extended by doing* P+1 *additional transitions in each of which we deliver the entire message buffer. Let $\boldsymbol{g}$ be a message that is sent in some run $\mathcal{S}$ of $\mathcal{N}$ on I. Message $\boldsymbol{g}$ is delivered in the last transition of $\mathcal{R}'$.*

Table 2 Input relations for M

Relation	Purpose
$s^{(1)}$ with $s \in \Gamma$	One relation for each tape symbol
$q^{(1)}$ with $q \in Q$	One relation for each state symbol
$0^{(1)}, 1^{(1)}, 01^{(1)}$	Relations providing the numbers 0 and 1

Proof Recall the definitions and notations regarding derivation trees from Section 2.6. Let $\mathcal{T}$ be a derivation tree for $\boldsymbol{g}$ extracted from $\mathcal{S}$. Let $\kappa^{\mathcal{T}}$ be the canonical scheduling of $\mathcal{T}$. Let n denote the height of $\mathcal{T}$, measured as the number of edges on the longest path from the root to a leaf. For $i \in \{1, \ldots, n\}$, define the following message set M_i:

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

Because the rules of Π are message-positive, $body^{\mathcal{T}}(x)|_{\Upsilon_{\text{msg}}}$ contains only facts. Intuitively, M_i is the union of all message facts needed by rules scheduled at transition i by $\kappa^{\mathcal{T}}$. Since $n \leq \mathbf{P}$, we can consider the transition index j of $\mathcal{R}'$ such that $j+1, \ldots, j+n, j+n+1$ are the last $n+1$ transitions of $\mathcal{R}'$. If we can show that $\boldsymbol{g}$ is sent in transition $j+n$, then $\boldsymbol{g}$ is delivered in the last transition $j+n+1$ (because the entire buffer is delivered), as desired.

Because sending rules are static and message-positive, and $\mathcal{R}'$ and $\mathcal{S}$ have the same input I, it is sufficient to show that M_n is delivered in transition $j+n$, so that the root rule and valuation of $\mathcal{T}$ derive $\boldsymbol{g}$. Specifically, we show by induction on $i \in \{1, \ldots, n\}$ that M_i is delivered in transition $j+i$ of $\mathcal{R}'$. The property holds for the base case because $M_1 = \emptyset$.[15] For the induction hypothesis, we assume that M_i can be delivered in transition $j+i$ of $\mathcal{R}'$. We now show that M_{i+1} can be delivered in transition $j+i+1$ of $\mathcal{R}'$. Let $\boldsymbol{h} \in M_{i+1}$. By definition of M_{i+1}, there is an internal node x of $\mathcal{T}$ with $\kappa^{\mathcal{T}}(x) = i+1$ and $\boldsymbol{h} \in body^{\mathcal{T}}(x)|_{\Upsilon_{\text{msg}}}$. We show that $\boldsymbol{h}$ is sent in transition $j+i$ of $\mathcal{R}'$, so that $\boldsymbol{h}$ is delivered in transition $j+i+1$. By message-positivity of $rule^{\mathcal{T}}(x)$, there is a child node $y \in int^{\mathcal{T}}$ of x such that $fact^{\mathcal{T}}(y) = \boldsymbol{h}$. By definition of $\kappa^{\mathcal{T}}$, we have $\kappa^{\mathcal{T}}(y) = i$. We show that $val^{\mathcal{T}}(y)$ is satisfying for $rule^{\mathcal{T}}(y)$ during transition $j+i$ of $\mathcal{R}'$. Like above, because sending rules are static and message-positive, and $\mathcal{R}'$ and $\mathcal{S}$ have the same input I, it is sufficient to show that M_i is delivered in transition $j+i$, which holds by the induction hypothesis. □

C.2 Complexity Lower Bound

Here we complete the specification of transducer Π over schema Υ from Section 6.2. We assume that Υ_{in} contains the additional relations of Table 2. All rules we specify below are *sending* rules.

[15] Indeed, if an internal node x needs child messages then the corresponding child nodes are scheduled earlier, making $\kappa^{\mathcal{T}}(x) > 1$.

Let w denote the input word for M under consideration, and let $n = |w|$. We can select a constant $k \in \mathbb{N}$ such that if M accepts w then M has an accepting computation trace on w with at most 2^{n^k} transitions.

C.2.1 Binary Addresses

Abbreviate $z = n^k$. Note that z is polynomial in n. Because we are only concerned with accepting computation traces of length at most 2^{n^k}, the address of a reachable tape cell can be represented as a binary number with z bits. We denote such a number as $(a_1 \dots a_z)$ where each a_i is 0 or 1 and a_z is the least significant bit. Note that z bits actually allow us to represent addresses larger than 2^{n^k}, but the accepting computation trace will never reach these tape cells, hence, we will ignore those addresses in the following.

We will use messages of the form $\texttt{succ}(a_1, \dots, a_z; b_1, \dots, b_z)$ to say that address $(b_1 \dots b_z)$ is the successor of address $(a_1 \dots a_z)$, i.e., $(b_1 \dots b_z)$ is obtained from $(a_1 \dots a_z)$ by adding 1.[16] Similarly, we use messages of the form $\texttt{less}(a_1, \dots, a_z; b_1, \dots, b_z)$ and $\texttt{diff}(a_1, \dots, a_z; b_1, \dots, b_z)$ to say respectively that $(a_1 \dots a_z)$ is smaller than $(b_1 \dots b_z)$ and that $(a_1 \dots a_z)$ and $(b_1 \dots b_z)$ are different. To specify these messages, we add the following rules for each $p = 1, \dots, z$:

$$
\begin{aligned}
&\texttt{succ}(\texttt{a}_1, \dots, \texttt{a}_{p-1}, \texttt{a}_p, \dots, \texttt{a}_z;\ \texttt{a}_1, \dots, \texttt{a}_{p-1}, \texttt{b}_p, \dots, \texttt{b}_z) \leftarrow \\
&\qquad 01(\texttt{a}_1),\ \dots,\ 01(\texttt{a}_{p-1}),\ 0(\texttt{a}_p),\ 1(\texttt{b}_p), \\
&\qquad 1(\texttt{a}_{p+1}),\ \dots,\ 1(\texttt{a}_z),\ 0(\texttt{b}_{p+1}),\ \dots,\ 0(\texttt{b}_z). \\
&\texttt{less}(\texttt{a}_1, \dots, \texttt{a}_{p-1}, \texttt{a}_p, \dots, \texttt{a}_z;\ \texttt{a}_1, \dots, \texttt{a}_{p-1}, \texttt{b}_p, \dots, \texttt{b}_z) \leftarrow \\
&\qquad 01(\texttt{a}_1),\ \dots,\ 01(\texttt{a}_{p-1}),\ 0(\texttt{a}_p),\ 1(\texttt{b}_p), \\
&\qquad 01(\texttt{a}_{p+1}),\ \dots,\ 01(\texttt{a}_z),\ 01(\texttt{b}_{p+1}),\ \dots,\ 01(\texttt{b}_z). \\
&\texttt{diff}(\texttt{a}_1, \dots, \texttt{a}_{p-1}, \texttt{a}_p, \dots, \texttt{a}_z;\ \texttt{b}_1, \dots, \texttt{b}_{p-1}, \texttt{b}_p, \dots, \texttt{b}_z) \leftarrow \\
&\qquad 01(\texttt{a}_1),\ \dots,\ 01(\texttt{a}_z),\ 01(\texttt{b}_1),\ \dots,\ 01(\texttt{b}_z),\ \texttt{a}_p \neq \texttt{b}_p.
\end{aligned}
$$

Here, if $p = 1$ then the variables $\texttt{a}_1$ to $\texttt{a}_{p-1}$ are nonexistent, and if $p = z$ then the variables $\texttt{a}_{p+1}$ to $\texttt{a}_z$ and $\texttt{b}_{p+1}$ to $\texttt{b}_z$ are nonexistent. Note that the number and size of these above rules is polynomial in n, and they have no cyclic dependencies (leads to recursion-freeness).

C.2.2 Sending `error`

The message `error` is sent when some crucial properties of the input relations are violated.

[16] The semicolon in the fact only serves to better separate the two binary numbers visually.

First, we demand that for each configuration at most one state and head position is specified, and also that each tape cell has at most one symbol:

$$\begin{aligned}\texttt{error}() \leftarrow\ & \texttt{state}(\texttt{i}, \texttt{q}_1),\ \texttt{state}(\texttt{i}, \texttt{q}_2),\ \texttt{q}_1 \neq \texttt{q}_2.\\ \leftarrow\ & \texttt{head}(\texttt{i}, \texttt{h}_1, \ldots, \texttt{h}_z),\ \texttt{head}(\texttt{i}, \texttt{k}_1, \ldots, \texttt{k}_z),\\ & \texttt{diff}(\texttt{h}_1, \ldots, \texttt{h}_z; \texttt{k}_1, \ldots, \texttt{k}_z).\\ \leftarrow\ & \texttt{tape}(\texttt{i}, \texttt{a}_1, \ldots, \texttt{a}_z, \texttt{s}_1),\ \texttt{tape}(\texttt{i}, \texttt{a}_1, \ldots, \texttt{a}_z, \texttt{s}_2),\\ & \texttt{s}_1 \neq \texttt{s}_2.\end{aligned}$$

For the relations providing the binary numbers, we demand that relations 0 and 1 are disjoint, contain at most one value, and that relation 01 is the union of 0 and 1:

$$\begin{aligned}\texttt{error}() \leftarrow\ & 0(\texttt{v}),\ 1(\texttt{v}).\\ \leftarrow\ & 0(\texttt{v}),\ 0(\texttt{w}),\ \texttt{v} \neq \texttt{w}.\\ \leftarrow\ & 1(\texttt{v}),\ 1(\texttt{w}),\ \texttt{v} \neq \texttt{w}.\\ \leftarrow\ & 0(\texttt{v}),\ \neg 01(\texttt{v}).\\ \leftarrow\ & 1(\texttt{v}),\ \neg 01(\texttt{v}).\\ \leftarrow\ & 01(\texttt{v}),\ \neg 0(\texttt{v}),\ \neg 1(\texttt{v}).\end{aligned}$$

For the relations providing symbols of Γ, we demand that they are pairwise disjoint and that each contains at most one symbol. We demand the same properties of the relations providing symbols of Q. Formally, for each $(s_1, s_2) \in (\Gamma \times \Gamma) \cup (Q \times Q)$ with $s_1 \neq s_2$, we add the rule

$$\texttt{error}() \leftarrow s_1(\texttt{v}),\ s_2(\texttt{v}).$$

And for each $s \in \Gamma \cup Q$, we add the rule

$$\texttt{error}() \leftarrow s(\texttt{v}),\ s(\texttt{w}),\ \texttt{v} \neq \texttt{w}.$$

C.2.3 Sending `accept`

We give the rules to send messages of the form $\texttt{reach}_0(i, j)$ and $\texttt{start}(i)$, where $\texttt{reach}_0(i, j)$ indicates that configuration j can be reached by a valid Turing machine transition from configuration i, and where $\texttt{start}(i)$ indicates that configuration i has the properties of the start configuration.

Sending $\texttt{reach}_0$ We will send messages of the form $\texttt{tapeCOK}(i, j, a_1, \ldots, a_z)$ to say that in configuration j, the tape cell at address $(a_1 \ldots a_z)$ can be explained by a Turing machine transition applied to configuration i.[17] To send $\texttt{reach}_0(i, j)$, we have to check that such messages can be sent for *all* tape cells. We will simultaneously enforce that the state and head position of j can follow from the state and head position of i.

To send $\texttt{tapeCOK}(i, j, a_1, \ldots, a_z)$, we consider three cases, where $(h_1 \ldots h_z)$ denotes the head position of configuration i:

[17] The name `tapeCOK` stands for "tape cell ok".

- $(a_1 \dots a_z) < (h_1 \dots h_z)$, in which case the cell contents at $(a_1 \dots a_z)$ should be unaltered in j with respect to i;
- the symmetric case $(h_1 \dots h_z) < (a_1 \dots a_z)$, with the same constraint;
- $(a_1 \dots a_z) = (h_1 \dots h_z)$, in which case a transition of Turing machine M has to explain the symbol at cell $(a_1 \dots a_z)$ in j.

The first case is implemented by the following rule:

$$\begin{aligned}\texttt{tapeCOK}(\texttt{i}, \texttt{j}, \texttt{a}_1, \dots, \texttt{a}_\texttt{z}) \leftarrow\ & \texttt{head}(\texttt{i}, \texttt{h}_1, \dots, \texttt{h}_\texttt{z}),\ \texttt{less}(\texttt{a}_1, \dots, \texttt{a}_\texttt{z};\ \texttt{h}_1, \dots, \texttt{h}_\texttt{z}),\\ & \texttt{tape}(\texttt{i}, \texttt{a}_1, \dots, \texttt{a}_\texttt{z}, \texttt{s}),\ \texttt{tape}(\texttt{j}, \texttt{a}_1, \dots, \texttt{a}_\texttt{z}, \texttt{s}).\end{aligned}$$

The second case is done with a similar rule, except that $\texttt{less}(\texttt{a}_1, \dots, \texttt{a}_\texttt{z};\ \texttt{h}_1, \dots, \texttt{h}_\texttt{z})$ is replaced by $\texttt{less}(\texttt{h}_1, \dots, \texttt{h}_\texttt{z};\ \texttt{a}_1, \dots, \texttt{a}_\texttt{z})$.

The third case is split further depending on whether the head moves left or right. Let δ denote the transition function of Turing machine M. For each mapping $(q_1, s_1 \mapsto q_2, s_2, \mathrm{L}) \in \delta$, add the rule:

$$\begin{aligned}\texttt{tapeCOK}(\texttt{i}, \texttt{j}, \texttt{h}_1, \dots, \texttt{h}_\texttt{z}) \leftarrow\ & \texttt{head}(\texttt{i}, \texttt{h}_1, \dots, \texttt{h}_\texttt{z}),\ \texttt{head}(\texttt{j}, \texttt{k}_1, \dots, \texttt{k}_\texttt{z}),\\ & \texttt{succ}(\texttt{k}_1, \dots, \texttt{k}_\texttt{z};\ \texttt{h}_1, \dots, \texttt{h}_\texttt{z}),\\ & \texttt{state}(\texttt{i}, \texttt{q}_1),\ \texttt{tape}(\texttt{i}, \texttt{h}_1, \dots, \texttt{h}_\texttt{z}, \texttt{s}_1),\\ & \texttt{state}(\texttt{j}, \texttt{q}_2),\ \texttt{tape}(\texttt{j}, \texttt{h}_1, \dots, \texttt{h}_\texttt{z}, \texttt{s}_2),\\ & q_1(\texttt{q}_1),\ s_1(\texttt{s}_1),\ q_2(\texttt{q}_2),\ s_2(\texttt{s}_2).\end{aligned}$$

Regarding relations q_1, s_1, q_2 and s_2, it does not matter what precise values they contain by genericity of the rules (as long as the conditions enforced in Section C.2.2 hold). A similar rule is added for each mapping $(q_1, s_1 \mapsto q_2, s_2, \mathrm{R}) \in \delta$, except that $\texttt{succ}(\texttt{k}_1, \dots, \texttt{k}_\texttt{z};\ \texttt{h}_1, \dots, \texttt{h}_\texttt{z})$ is replaced by $\texttt{succ}(\texttt{h}_1, \dots, \texttt{h}_\texttt{z};\ \texttt{k}_1, \dots, \texttt{k}_\texttt{z})$. Note that the nondeterminism of Turing machine M is implemented by having multiple rules in Π of these last two forms. Also, the number of rules for relation $\texttt{tapeCOK}$ is *constant* because M is fixed, but their size is polynomial in n.

Next, we send messages of the form $\texttt{tapeOK}_m(i, j, a_1, \dots, a_z;\ b_1, \dots, b_z)$, with $m = 0, \dots, z$ and $(a_1 \dots a_z) \leq (b_1 \dots b_z)$, to say that interval $[(a_1 \dots a_z), (b_1 \dots b_z)]$ contains 2^m tape cells and that the message $\texttt{tapeCOK}(i, j, c_1, \dots, c_z)$ can be sent for *all* addresses $(c_1 \dots c_z)$ in this interval. The goal is to eventually send a message $\texttt{tapeOK}_z(i, j, a_1, \dots, a_z;\ b_1, \dots, b_z)$ where $(a_1 \dots a_z)$ is the first tape cell. To start, we generate $\texttt{tapeOK}_0$-messages:

$$\texttt{tapeOK}_0(\texttt{i}, \texttt{j}, \texttt{a}_1, \dots, \texttt{a}_\texttt{z};\ \texttt{a}_1, \dots, \texttt{a}_\texttt{z}) \leftarrow \texttt{tapeCOK}(\texttt{i}, \texttt{j}, \texttt{a}_1, \dots, \texttt{a}_\texttt{z}).$$

And we add the following rule for each $m = 1, \dots, z$:

$$\begin{aligned}\texttt{tapeOK}_m(\texttt{i}, \texttt{j}, \texttt{a}_1, \dots, \texttt{a}_\texttt{z};\ \texttt{b}_1, \dots, \texttt{b}_\texttt{z}) \leftarrow\ & \\ & \texttt{tapeOK}_{m-1}(\texttt{i}, \texttt{j}, \texttt{a}_1, \dots, \texttt{a}_\texttt{z};\ \texttt{c}_1, \dots, \texttt{c}_\texttt{z}),\\ & \texttt{tapeOK}_{m-1}(\texttt{i}, \texttt{j}, \texttt{d}_1, \dots, \texttt{d}_\texttt{z};\ \texttt{b}_1, \dots, \texttt{b}_\texttt{z}),\\ & \texttt{succ}(\texttt{c}_1, \dots, \texttt{c}_\texttt{z};\ \texttt{d}_1, \dots, \texttt{d}_\texttt{z}).\end{aligned}$$

Note that the number and size of such rules is polynomial in n.

Finally, the $\texttt{reach}_0$-messages are sent with the following rule:

$$\texttt{reach}_0(\texttt{i},\texttt{j}) \leftarrow \texttt{tapeOK}_z(\texttt{i},\texttt{j},\texttt{a}_1,\ldots,\texttt{a}_z;\,\texttt{b}_1,\ldots,\texttt{b}_z),\; 0(\texttt{a}_1),\;\ldots,\;0(\texttt{a}_z).$$

Note that we constrain attention to the range $[0, 2^z]$.

Sending `start` To send a message $\texttt{start}(i)$, we have to check that configuration i has the properties of the start configuration: (i) the tape contains the input word w starting at the first tape cell, with the other tape cells blank; (ii) the state is q_0; and, (iii) the head is at tape cell 0. The last two properties are easily checked.

To check property (i), we send messages of the form $\texttt{startTapeCOK}(i, a_1, \ldots, a_z)$ to indicate that the contents of tape cell $(a_1 \ldots a_z)$ in configuration i is as required by the start configuration. We add the following rule for all addresses $a \in [0, n-1]$, where $(a_1 \ldots a_z)$ is the binary representation of a and w_a is the symbol of word w at (zero-based) index a:

$$\texttt{startTapeCOK}(\texttt{i},\texttt{a}_1,\ldots,\texttt{a}_z) \leftarrow a_1(\texttt{a}_1),\;\ldots,\;a_z(\texttt{a}_z),\;\texttt{tape}(\texttt{i},\texttt{a}_1,\ldots,\texttt{a}_z,\texttt{s}),\;w_a(\texttt{s}).$$

We also add one rule to demand that the other tape cells contain blanks, where $\sqcup \in \Gamma$ denotes the blank symbol and $(b_1 \ldots b_z)$ is the binary representation of $n-1$:

$$\texttt{startTapeCOK}(\texttt{i},\texttt{a}_1,\ldots,\texttt{a}_z) \leftarrow b_1(\texttt{b}_1),\;\ldots,\;b_z(\texttt{b}_z),\;\texttt{less}(\texttt{b}_1,\ldots,\texttt{b}_z;\,\texttt{a}_1,\ldots,\texttt{a}_z),\;\texttt{tape}(\texttt{i},\texttt{a}_1,\ldots,\texttt{a}_z,\texttt{s}),\;\sqcup(\texttt{s}).$$

Note that the number and size of rules for relation `startTapeCOK` is polynomial in n.

Next, similarly to the relations $\texttt{tapeOK}_m$ above, we send messages of the form $\texttt{startTapeOK}_m(i, a_1, \ldots, a_z;\, b_1, \ldots, b_z)$, with $m = 0, \ldots, z$ and $(a_1 \ldots a_z) \leq (b_1 \ldots b_z)$, to say that the interval $[(a_1 \ldots a_z), (b_1 \ldots b_z)]$ contains 2^m tape cells and that message $\texttt{startTapeCOK}(i, c_1, \ldots, c_z)$ can be sent for all addresses $(c_1 \ldots c_z)$ in this interval. We do not explicitly give the rules, because they are very similar to the rules of the relations $\texttt{tapeOK}_m$. The number and size of the added rules is also polynomial in n.

Finally, we can send the `start`-messages:

$$\texttt{start}(\texttt{i}) \leftarrow \texttt{startTapeOK}_z(\texttt{i},\texttt{a}_1,\ldots,\texttt{a}_z;\,\texttt{b}_1,\ldots,\texttt{b}_z),\; 0(\texttt{a}_1),\;\ldots,\;0(\texttt{a}_z),\;\texttt{head}(\texttt{i},\,\texttt{a}_1,\ldots,\texttt{a}_z),\;\texttt{state}(\texttt{i},\texttt{q}),\;q_0(\texttt{q}).$$

C.2.4 Correctness

Here we argue the correctness of the reduction.

First Direction Suppose that M has an accepting computation trace on input word w. We have to show that the transducer network $\mathcal{N}$ for w is diffluent.

The accepting computation trace of M is a sequence of configurations, and we identify each configuration by their (one-based) ordinal. We always have $i \leq 2^z$. Let I be the input instance for $\mathcal{N}$ consisting of the following facts:

- facts $\texttt{state}(i, q_i)$ and $\texttt{head}(i, h_1, \ldots, h_z)$ for each configuration i, where q_i and $(h_1 \ldots h_z)$ are respectively the state and head position of i;
- fact $\texttt{tape}(i, a_1, \ldots, a_z, s)$ for each configuration i and each address $(a_1 \ldots a_z) \in [0, 2^z]$, where $s \in \Gamma$ is the contents of cell $(a_1 \ldots a_z)$ in configuration i;
- fact $s(s)$ for each $s \in \Gamma$; fact $q(q)$ for each $q \in Q$; facts $0(0)$, $1(1)$, $01(0)$, and $01(1)$; and, fact $A(a)$.

Note that no `error`-message can be sent on this instance (cf. Section C.2.2). Hence, it is sufficient to show that `accept`() can be sent, so that input fact $A(a)$ gives rise to the messages $A_{\text{msg}}(a)$ and $B_{\text{msg}}(a)$. Then there exist two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ so that $T(a)$ is created in $\mathcal{R}_1$ and not in $\mathcal{R}_2$ or any extension thereof.

Let e denote the last configuration of the computation trace. The state of e is q_{accept}. Looking at the rules for sending `accept`-messages (Section 6.2), since I contains $\texttt{state}(e, q_{\text{accept}})$ and $q_{\text{accept}}(q_{\text{accept}})$, we are left to show that the following messages can be sent: $\texttt{start}(1)$ and $\texttt{reach}_m(1, e)$ for some $m \in [0, z]$. Because configuration 1 is the start configuration of the computation trace, and because we have accurately described this configuration in the input relations, we can see that $\texttt{start}(1)$ can be sent. Similarly, we can see that for each pair (i, j) of subsequent configurations in the trace, the message $\texttt{reach}_0(i, j)$ can be sent. And because the $\texttt{reach}_m$-rules with $m \in [0, z]$ allow us to connect configurations over arbitrary distances within $[1, 2^z]$, we can also send $\texttt{reach}_m(1, e)$ for some $m \in [0, z]$.

Second Direction Suppose that the transducer network $\mathcal{N}$ for w is diffluent. We have to show that M has an accepting computation trace on w.

First, because $\mathcal{N}$ is diffluent, there exists an input instance I for $\mathcal{N}$, and two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\mathcal{N}$ on I, such that $last(\mathcal{R}_1)$ contains an output fact $T(a)$ that is not in $last(\mathcal{R}_2)$, and $T(a)$ can not be created in any extension of $\mathcal{R}_2$.

We first show that `accept`() can be sent on input I and that `error`() can not. The presence of $T(a)$ in $last(\mathcal{R}_1)$ implies that the message $A_{\text{msg}}(a)$ can be sent. This in turn implies that `accept`() can be sent. Now, since by static send rules the message $A_{\text{msg}}(a)$ can also be sent in an extension of $\mathcal{R}_2$, the reason why $T(a)$ can not be created in that extension is that the memory fact $B(a)$ is present *and* that the message `error`() can never be delivered, and hence can never be sent.

Looking at the sending rules for relation `accept`, the sending of `accept`() in $\mathcal{R}_1$ must have been caused by the joint occurrence of the following four facts during some transition of $\mathcal{N}$: the message facts $\texttt{start}(x)$ and $\texttt{reach}_m(x, y)$ for some $x, y \in adom(I)$ and $m \in [0, z]$, and the input facts $\texttt{state}(y, q)$ and $q_{\text{accept}}(q)$. The input facts together already imply that y could describe an accepting configuration. Now we have to look at the derivation histories of the two messages to construct a full accepting computation trace.

As a general remark, because `error`() can never be sent, the input satisfies the restrictions enforced in Section C.2.2. In particular, each configuration has at most

one state and at most one head position in relations `state` and `head` respectively, and each configuration has at most one symbol for each tape cell in relation `tape`. So, the presence of the message `start`(x) implies that x not only has *precisely* one state, one head position and one symbol in each tape cell, but also that x satisfies the additional properties of a valid start configuration. Hence, x is a fully specified start configuration.

The presence of the message $\texttt{reach}_m(x, y)$ implies there is a sequence of configurations $c_1, \ldots, c_e$ in the input with $c_1 = x$ and $c_e = y$ and such that the message $\texttt{reach}_0(i, j)$ can be sent for each pair (i, j) of subsequent configurations. Again using the absence of `error`(), the presence of the message $\texttt{reach}_0(i, j)$ implies that configurations i and j each have *precisely* one state, one head position, and one symbol in each tape cell, and that there exists a valid transition rule of Turing machine M to explain how configuration j follows from configuration i. Finally, using that y is accepting (see above), we have found an accepting computation trace of M on w.

Appendix D: Expressivity Upper Bound

D.1 Correctness Part 1

Let Φ be as constructed in Section 7.2.3. Let H be an arbitrary distributed database instance over $in^{\mathcal{N}}$. Abbreviate $I = \langle H \rangle^{\mathcal{N}}$. Let $\boldsymbol{f} \in \Phi(I)$. We have to show that $\boldsymbol{f}$ is output at node x when $\mathcal{N}$ is run on H. It is sufficient to show that $\boldsymbol{f}$ is output by $\mathcal{M}$ on input I.

We remind that Section 7.2.2 contains common concepts and notations. Helper claims can be found in Section D.3.

D.1.1 Satisfying Valuation

Since $\boldsymbol{f} \in \Phi(I)$, program Φ contains a UCQ$^{\neg}$-program $derive_{G,\mathcal{T}_0}$ such that $\boldsymbol{f} \in derive_{G,\mathcal{T}_0}(I)$. Hence, there exists a subset $G_0 \subseteq forest_{\mathcal{R}}$ and an equivalence relation E on $adom(G_0)$ such that $G = E(G_0)$ and $\mathcal{T}_0 \in G$.

Like before, we regard $derive_{G,\mathcal{T}}$ as an $\exists$FO-formula, where $\mathcal{T}$ is the truncated version of $\mathcal{T}_0$ and κ is the canonical scheduling of $\mathcal{T}_0$:

$$derive_{G,\mathcal{T}_0} := \exists \bar{z} \left(diffVal_G \wedge sndMsg_G \wedge succeed_{G,\mathcal{T},\kappa} \right).$$

Here, free variables are constituted by the tuple $\bar{x}$ of values occurring in the root fact of $\mathcal{T}_0$, and $\bar{z}$ are the values in $adom(G)$ that are not in $\bar{x}$. Since $\boldsymbol{f} \in derive_{G,\mathcal{T}_0}(I)$, there exists a valuation $Val : adom(G) \to adom(I)$ that makes the following quantifier-free formula true:

$$diffVal_G \wedge sndMsg_G \wedge succeed_{G,\mathcal{T},\kappa}.$$

The part $diffVal_G$ makes Val injective.

D.1.2 Concrete Run

For each tree $\mathcal{T}' \in G$, for each internal node x of $\mathcal{T}'$, we can apply the function Val after valuation $val^{\mathcal{T}}(x)$. The resulting valuations still satisfy the nonequalities of the rules, because these nonequalities are satisfied under $val^{\mathcal{T}}(x)$ and Val is injective. Let F denote the forest of (structurally equivalent) derivation trees obtained from G in this way. Following the principle of canonical runs of Section 7.2.3, we will concurrently execute all trees in F by their canonical scheduling. This results in a run $\mathcal{R}$, whose length is the largest height of any tree in F. We now show that $\boldsymbol{f}$ is derived in $\mathcal{R}$.

Let $\mathcal{T}_0$ be as above. Let $\mathcal{S}_0 \in F$ be the structurally equivalent tree. We first show that $fact^{\mathcal{S}_0}(root^{\mathcal{S}_0}) = \boldsymbol{f}$. The tuple of values in $fact^{\mathcal{T}_0}(root^{\mathcal{T}_0})$ are the free variables of $derive_{G,\mathcal{T}_0}$. Thus $Val(fact^{\mathcal{T}_0}(root^{\mathcal{T}_0})) = \boldsymbol{f}$. And by construction of F, we have $fact^{\mathcal{S}_0}(root^{\mathcal{S}_0}) = Val(fact^{\mathcal{T}_0}(root^{\mathcal{T}_0}))$.

Henceforth, we will focus on the truncated trees $\mathcal{T}$ and $\mathcal{S}$ of $\mathcal{T}_0$ and $\mathcal{S}_0$ respectively. The canonical scheduling κ of $\mathcal{T}_0$ is also defined on $\mathcal{S}$. Now, using the order implied by κ, we show by induction on $x \in a^{\mathcal{S}}$ that $fact^{\mathcal{S}}(x)$ is derived in transition $\kappa(x)$ of $\mathcal{R}$. So, let $x \in a^{\mathcal{S}}$ be a node such that for all alpha child nodes y of x, the fact $fact^{\mathcal{S}}(y)$ is derived in transition $\kappa(y)$ of $\mathcal{R}$.[18] We show that $val^{\mathcal{S}}(x)$ is satisfying for $rule^{\mathcal{S}}(x)$ in transition $\kappa(x)$. The nonequalities of $rule^{\mathcal{S}}(x)$ are satisfied because they are satisfied under $val^{\mathcal{T}}(x)$ and because Val is injective. Next, we differentiate between the different kinds of atoms in the body of $rule^{\mathcal{S}}(x)$.

Input Let $\boldsymbol{l} \in body^{\mathcal{S}}(x)|_{\Upsilon_{\text{in}}}$. We have to show $I \models \boldsymbol{l}$. Let $\boldsymbol{l}' \in body^{\mathcal{T}}(x)|_{\Upsilon_{\text{in}}}$ be such that $\boldsymbol{l} = Val(\boldsymbol{l}')$. By construction, $\boldsymbol{l}'$ occurs in the conjunction $succeed^{\text{in}}_{G,\mathcal{T},\kappa}$, and since this formula is true under Val with respect to I, we have $I \models Val(\boldsymbol{l}')$ or equivalently $I \models \boldsymbol{l}$, as desired.

Messages Let $\boldsymbol{l} \in body^{\mathcal{S}}(x)|_{\Upsilon_{\text{msg}}}$. Abbreviate $i = \kappa(x)$. We have to show that $\boldsymbol{l}$ is delivered in transition i of $\mathcal{R}$. Because $rule^{\mathcal{S}}(x)$ is message-positive, $\boldsymbol{l}$ is a fact. Let $\boldsymbol{g} \in body^{\mathcal{T}}(x)|_{\Upsilon_{\text{msg}}}$ be such that $\boldsymbol{l} = Val(\boldsymbol{g})$. Because κ is an alignment for $\mathcal{T}$ with respect to the abstract canonical run $\mathcal{R}^G$, we have $\boldsymbol{g} \in M^G_i$. By Claim D.1, the fact $\boldsymbol{l} = Val(\boldsymbol{g})$ is delivered during transition i of $\mathcal{R}$, as desired.

Positive Output and Memory Let $\boldsymbol{l} \in body^{\mathcal{S}}(x)|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$ be such that $\boldsymbol{l}$ is positive. There is an alpha child y of x such that $fact^{\mathcal{S}}(y) = \boldsymbol{l}$. By assumption on x, $fact^{\mathcal{S}}(y)$ is derived during transition $\kappa(y)$ of $\mathcal{R}$, and thus $\boldsymbol{l}$ is available during transition $\kappa(x)$, as desired.

Negative Output and Memory Let $\boldsymbol{l} \in body^{\mathcal{S}}(x)|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$ be such that $\boldsymbol{l}$ is negative. Denote $\boldsymbol{l} = \neg \boldsymbol{g}$. We show that $\boldsymbol{g}$ is not derived before transition $\kappa(x)$ of $\mathcal{R}$. To relate back to $\mathcal{T}$, there is also a fact $\boldsymbol{h}$ such that $\boldsymbol{g} = Val(\boldsymbol{h})$ and $\neg \boldsymbol{h} \in body^{\mathcal{T}}(x)$.

[18] This property is automatically satisfied in the base case, where x has no alpha child nodes.

Towards a proof by contradiction, suppose that $\boldsymbol{g}$ is derived in some transition $j < \kappa(x)$ of $\mathcal{R}$. Then it is possible to extract a truncated derivation tree $\mathcal{S}'$ from $\mathcal{R}$ with $fact^{\mathcal{S}'}(root^{\mathcal{S}'}) = \boldsymbol{g}$, together with an alignment κ' of $\mathcal{S}'$ such that for all alpha nodes z of $\mathcal{S}'$, the fact $fact^{\mathcal{S}'}(z)$ is derived during transition $\kappa'(z)$ of $\mathcal{R}$ because $val^{\mathcal{S}'}(z)$ is satisfying for $rule^{\mathcal{S}'}(z)$. Note that val^{-1} is defined because Val is injective. Let $\mathcal{T}'$ be the truncated derivation tree obtained from $\mathcal{S}'$ by applying for each alpha node z, the function Val^{-1} after the valuation $val^{\mathcal{S}'}(z)$. The tree $\mathcal{T}'$ has root fact $val^{-1}(\boldsymbol{g}) = \boldsymbol{h}$.

There exists $y \in \beta^{\mathcal{T}}(x)$ with $fact^{\mathcal{T}}(y) = \boldsymbol{h}$. Suppose we would also know that $(\mathcal{T}', \kappa') \in align^{G}(\boldsymbol{h})$ (shown below). Then the subformula $succeed^{\text{deny}}_{G,\mathcal{T},\kappa}$ contains the subformula $\neg succeed_{G,\mathcal{T}',\kappa'}$, which is true under Val. Equivalently, $succeed_{G,\mathcal{T}'\kappa'}$ is false under Val. We will use this information to show that at least one alpha node z of $\mathcal{T}'$ exists for which valuation $Val \circ val^{\mathcal{T}'}(z)$ is not satisfying for $rule^{\mathcal{T}'}(z)$ during transition $\kappa'(z)$ of $\mathcal{R}$, or equivalently, valuation $Val \circ Val^{-1} \circ val^{\mathcal{S}'}(z) = val^{\mathcal{S}'}(z)$ is not satisfying for $rule^{\mathcal{S}'}(z)$ during transition $\kappa'(z)$. This gives the desired contradiction.

Since $succeed_{G,\mathcal{T}',\kappa'}$ is false under Val, it must be that either $succeed^{\text{in}}_{G,\mathcal{T}',\kappa'}$ is false or $succeed^{\text{deny}}_{G,\mathcal{T}',\kappa'}$ is false. In the first case, there is an alpha node z of $\mathcal{T}'$ and a literal $\boldsymbol{l} \in body^{\mathcal{T}'}(z)|_{\gamma_{\text{in}}}$ such that $I \not\models Val(\boldsymbol{l})$. This immediately gives that $Val \circ val^{\mathcal{T}'}(z)$ is not satisfying for $rule^{\mathcal{T}'}(z)$ during any transition of $\mathcal{R}$, hence, not in transition $\kappa'(z)$, as desired.

Now suppose that $succeed^{\text{deny}}_{G,\mathcal{T}',\kappa'}$ is false under Val. Thus, $succeed^{\text{deny}}_{G,\mathcal{T}',\kappa'}$ contains a subformula $\neg succeed_{G,\mathcal{T}'',\kappa''}$ where $succeed_{G,\mathcal{T}'',\kappa''}$ is true under Val. Hence, there is an alpha node z of $\mathcal{T}'$, with a beta child u, letting $\boldsymbol{i} = fact^{\mathcal{T}'}(u)$, and there is a pair $(\mathcal{T}'', \kappa'') \in align^{G}(\boldsymbol{i})$ with $\kappa''(root^{\mathcal{T}''}) < \kappa'(z)$. Let $\mathcal{S}''$ be the (truncated) derivation tree obtained from $\mathcal{T}''$ by applying Val after all valuations. Now, using the natural recursion on $succeed_{G,\mathcal{T}'',\kappa''}$, it is possible to show that $(\mathcal{S}'', \kappa'')$ derives $Val(\boldsymbol{i})$ during earlier transition $\kappa''(root^{\mathcal{T}''}) < \kappa'(z)$. This reasoning ends, because in each recursive step we come strictly closer to the beginning of $\mathcal{R}$, and eventually we only use formulas of the form $succeed^{\text{in}}_{G,\_,\_}$. Since valuation $Val \circ val^{\mathcal{T}'}(z)$ requires the absence of $Val(\boldsymbol{i})$ during $\kappa'(z)$, and $Val(\boldsymbol{i})$ is present in $\kappa'(z)$, this valuation is not satisfying during transition $\kappa'(z)$ of $\mathcal{R}$, as desired.

Let $\mathcal{T}'$ and κ' be as above. We are left to show that $(\mathcal{T}', \kappa') \in align^{G}(\boldsymbol{h})$. First, because κ' is an alignment for $\mathcal{S}'$, and because $\mathcal{T}'$ and $\mathcal{S}'$ are structurally equivalent, κ' is a scheduling for $\mathcal{T}'$. Next, let z be an internal (alpha) node of $\mathcal{T}'$. Let $\boldsymbol{l} \in body^{\mathcal{T}'}(z)|_{\gamma_{\text{msg}}}$, where $\boldsymbol{l}$ is a fact by message-positivity of $rule^{\mathcal{T}'}(z)$. We have to show that $\boldsymbol{l} \in M^{G}_{j}$ where $j = \kappa'(z)$. Since $val^{\mathcal{T}'}(z) = Val^{-1} \circ val^{\mathcal{S}'}(z)$, we can consider the fact $\boldsymbol{i} \in body^{\mathcal{S}'}(z)|_{\gamma_{\text{msg}}}$ such that $\boldsymbol{l} = Val^{-1}(\boldsymbol{i})$. Now, since κ' is an alignment for $\mathcal{S}'$ with respect to $\mathcal{R}$, we know that $\boldsymbol{i}$ is delivered in transition j of $\mathcal{R}$. Then, by Claim D.1, there is a fact $\boldsymbol{l}' \in M^{G}_{j}$ such that $Val(\boldsymbol{l}') = \boldsymbol{i}$. But by injectivity of Val, this means $\boldsymbol{l}' = Val^{-1}(\boldsymbol{i}) = \boldsymbol{l}$, as desired.

D.2 Correctness Part 2

Let H be an arbitrary input over $in^{\mathcal{N}}$. Abbreviate $I = \langle H \rangle^{\mathcal{N}}$. Let $\boldsymbol{f}$ be an R-fact output at node x when $\mathcal{N}$ is run on H. This implies that $\mathcal{M}$ outputs $\boldsymbol{f}$ on input I. We have to show that $\boldsymbol{f} \in \Phi(I)$, with Φ as constructed in Section 7.2.3.

Let Π denote the transducer of $\mathcal{M}$. We remind that Section 7.2.2 contains common concepts and notations. Additionally, for two structurally equivalent derivation trees $\mathcal{T}$ and $\mathcal{S}$, we write $map_{\mathcal{T},\mathcal{S}}$ to denote the structural bijection from nodes of $\mathcal{T}$ to nodes of $\mathcal{S}$. Lastly, helper claims can be found in Appendix D.3.

D.2.1 Collecting Trees

On input I, from each run of $\mathcal{M}$ in which $\boldsymbol{f}$ is output, we can extract a derivation tree for $\boldsymbol{f}$. Now, let F be a maximal set of derivation trees for $\boldsymbol{f}$ extracted from all possible runs of $\mathcal{M}$ on I, such that no two trees are structurally equivalent. Set F is finite because Π is recursion-free.

D.2.2 Canonical Run

Following the principle of canonical runs from Section 7.2.3, we can concurrently execute all trees of F. This results in a run $\mathcal{R}$ whose length is the height of the largest tree in F.

We now show that $\boldsymbol{f}$ is derived in $\mathcal{R}$. Because $\mathcal{M}$ outputs $\boldsymbol{f}$ on input I, confluence of $\mathcal{M}$ implies that $\mathcal{R}$ can always be extended to a run $\mathcal{R}'$ in which $\boldsymbol{f}$ is output. From $\mathcal{R}'$, we can extract a pair $(\mathcal{T}, \kappa)$ of a concrete derivation tree for $\boldsymbol{f}$ and a scheduling for this tree, such that for each $x \in int^{\mathcal{T}}$ the fact $fact^{\mathcal{T}}(x)$ is derived during transition $\kappa(x)$ of $\mathcal{R}'$ by applying $val^{\mathcal{T}}(x)$ to $rule^{\mathcal{T}}(x)$. There is some tree $\mathcal{S} \in F$ structurally equivalent to $\mathcal{T}$. Using the order implied by canonical scheduling $\kappa^{\mathcal{S}}$, we show by induction on the alpha nodes $x \in a^{\mathcal{S}}$ that $fact^{\mathcal{S}}(x)$ is derived during transition $\kappa^{\mathcal{S}}(x)$ by applying valuation $val^{\mathcal{S}}(x)$ to $rule^{\mathcal{S}}(x)$. Let $x \in a^{\mathcal{S}}$, assuming for each descendant $y \in a^{\mathcal{S}}$ of x that $fact^{\mathcal{S}}(y)$ is derived during transition $\kappa^{\mathcal{S}}(y)$.

Input Since $\mathcal{S} \in F$, the tree $\mathcal{S}$ was extracted from a run, and hence, the input literals of $rule^{\mathcal{S}}(x)$ must be satisfied under $val^{\mathcal{S}}(x)$.

Messages Moreover, because sending rules are message-positive and static, it can be shown that the messages needed by $rule^{\mathcal{S}}(x)$ under $val^{\mathcal{S}}(x)$ are delivered in $\mathcal{R}$ during transition $\kappa^{\mathcal{S}}(x)$ (details omitted).

Output and Memory Using the assumption on descendant alpha nodes of x, the positive output and memory facts required by $val^{\mathcal{S}}(x)$ are also satisfied.

As the last step, we show that the negative output and memory literals under $val^{\mathcal{S}}(x)$ are absent during transition $\kappa^{\mathcal{S}}(x)$. Let us abbreviate $n = map_{\mathcal{S},\mathcal{T}}$ (defined in Section 7.2.2). Since $\mathcal{S}$ and $\mathcal{T}$ are structurally equivalent and both derive the root fact $\boldsymbol{f}$, we can apply Claim D.2 to know that the valuations $val^{\mathcal{S}}(x)$ and $valL^{\mathcal{T}}(n(x))$ assign the same values to the free variables of $rule^{\mathcal{S}}(x)$. By selection of $(\mathcal{T}, \kappa)$,

the output and memory facts that rule $rule^{\mathcal{S}}(x)$ tests for absence under $val^{\mathcal{T}}(n(x))$, are effectively absent during transition $\kappa(n(x))$ of $\mathcal{R}'$. Now, because Π is inflationary, if we would know $\kappa^{\mathcal{S}}(x) \leq \kappa(n(x))$, then these same output and memory facts must also be absent during transition $\kappa^{\mathcal{S}}(x)$, as desired. We are left to show that $\kappa^{\mathcal{S}}(x) \leq \kappa(n(x))$. By definition of canonical scheduling $\kappa^{\mathcal{S}}$, transition $\kappa^{\mathcal{S}}(x)$ is the earliest transition of $\mathcal{R}$ in which the rule $rule^{\mathcal{S}}(x)$ can be executed if the derivation strategy represented by $\mathcal{S}$ must be followed.[19] Now, since the subtree under x in $\mathcal{S}$ is structurally equivalent to the subtree under $n(x)$ in $\mathcal{T}$, we have $\kappa^{\mathcal{S}}(x) \leq \kappa(n(x))$.

D.2.3 Create Valuation

From Section 7.2.3, recall the set $forest_R$, in which no two trees are structurally equivalent. For each tree $\mathcal{T} \in F$, there is a unique tree $\mathcal{S} \in forest_R$ that is structurally equivalent to $\mathcal{T}$. Let $G_0 \subseteq forest_R$ be all these trees. We define a function $Val_0 : adom(G_0) \to adom(F)$, giving rise to an equivalence relation on $adom(G_0)$.

First, let $\mathcal{S} \in G_0$. We can uniquely identify a *component* of a positive atom in $\mathcal{S}$ by a triple $(p, \boldsymbol{a}, i)$, where p is a path followed from the root towards an internal node x of $\mathcal{S}$; $\boldsymbol{a}$ is the head or a positive body atom of $rule^{\mathcal{S}}(x)$; and, i is a component index in $\boldsymbol{a}$. Here, p can be uniquely specified as the sequence of atoms $lit^{\mathcal{S}}(x)$ labelling the encountered internal nodes x. Two components $(p_1, \boldsymbol{a}_1, i_1)$ and $(p_2, \boldsymbol{a}_2, i_2)$ belong to the same rule if $p_1 = p_2$. Now, we define an equivalence relation over the components in a bottom-up way, as follows. Starting at an internal node x without other internal nodes as children, two components in $rule^{\mathcal{S}}(x)$ are equivalent if they contain the same variable. Going to the parent y of x, two components c_1 and c_2 in $rule^{\mathcal{S}}(y)$ are equivalent if *(i)* they contain the same variable; or *(ii)* they occur together in a positive body atom $\boldsymbol{a}$ of $rule^{\mathcal{S}}(y)$, and for the child x of y with $lit^{\mathcal{S}}(x) = \boldsymbol{a}$, the components in the head of $rule^{\mathcal{S}}(x)$ corresponding to c_1 and c_2 are equivalent. The equivalence relation on the components of $\mathcal{S}$ is unique, and its number of equivalence classes upper bounds the active domain size of $\mathcal{S}$.

Now we define function $Val_0 : adom(G_0) \to adom(F)$. Let $\mathcal{S} \in G_0$ and let $\mathcal{T} \in F$ denote the structurally equivalent tree. Because $\mathcal{S}$ and $\mathcal{T}$ are structurally equivalent, the equivalence classes on components of $\mathcal{S}$ transfer naturally to equivalence classes on the components of $\mathcal{T}$. Because $\mathcal{S}$ is general, its valuations assign a different value to each equivalence class, so we can define a function $V_{\mathcal{S}} : adom(\mathcal{S}) \to adom(\mathcal{T})$ that contains for each equivalence class e of $\mathcal{S}$ the mapping $(a \mapsto b)$, where a and b are the values assigned to e by $\mathcal{S}$ and $\mathcal{T}$ respectively. For the entire set G_0, we take the union of all mappings $V_{\mathcal{S}}$ with $\mathcal{S} \in G_0$. The result is denoted Val_0, and this is a function because each tree in G_0 has a disjoint active domain. We can now define an equivalence relation E on $adom(G_0)$: two values are equivalent if their image under Val_0 is the same. Assuming an order on **dom** (the same order as in Section 7.2.3), we can replace each value in $adom(G_0)$

[19] Indeed, for each subtree, the minimum number of transitions required to derive its root fact is precisely the height of this tree, and this is expressed in the canonical scheduling.

by the smallest value in its equivalence relation. This results in a set G of derivation trees, in which still as many structurally different trees occur as in G_0, and with $adom(G) \subseteq adom(G_0)$.[20]

Let Val denote the restriction of Val_0 to $adom(G)$; this function is injective.

D.2.4 Satisfying Valuation

Let F, G, and Val be as previously defined. For each tree $\mathcal{S} \in G$, if we would apply Val after each valuation in $\mathcal{S}$, we obtain a tree in F. So, if we would consider $adom(\mathcal{S})$ to be variable symbols, then we can see Val as an assignment to these variables. This will be used below to show that $\boldsymbol{f} \in \Phi(I)$.

As shown above, there is a derivation tree $\mathcal{T} \in F$ that derives $\boldsymbol{f}$ in $\mathcal{R}$, when executed according to its canonical scheduling. Let $\mathcal{S}_0 \in G$ be the tree that is structurally equivalent to $\mathcal{T}$. As remarked above, applying Val to $\mathcal{S}_0$ gives $\mathcal{T}$. Let $\mathcal{S}$ denote the truncated version of $\mathcal{S}_0$, and let κ denote the restriction of the canonical scheduling of $\mathcal{S}_0$ to the remaining nodes. Recalling the construction in Section 7.2.3, we have added to the UCQ$^\neg$-program Φ the UCQ$^\neg$-program $derive_{G,\mathcal{S}}$, given by the following equivalent ∃FO-formula:

$$derive_{G,\mathcal{S}} := \exists \bar{z} \left(diffVal_G \wedge sndMsg_G \wedge succeed_{G,\mathcal{S},\kappa} \right),$$

where $\bar{z}$ is an ordering of the values in $adom(G)$ not occurring in the tuple $\bar{x}$ in the root fact of $\mathcal{S}$. So, $\bar{x}$ are the free variables. Now, denoting $\boldsymbol{f} = R(\bar{a})$, to show $\boldsymbol{f} \in \Phi(I)$, it suffices to show that if $\bar{x}$ is assigned $\bar{a}$ then the resulting sentence is true with respect to I. This amounts to showing that the following quantifier-free formula is true under Val with respect to I:

$$diffVal_G \wedge sndMsg_G \wedge succeed_{G,\mathcal{S},\kappa}.$$

Diffval and sndMsg The subformula $diffVal_G$ is true because Val is injective on $adom(G)$. Next, the subformula $sndMsg_G$ is a large conjunction of input literals from the sending rules in G. Let $\boldsymbol{l}$ be such a literal. We have to show $I \models Val(\boldsymbol{l})$. There exists a tree $\mathcal{S}' \in G$ and an internal node x of $\mathcal{S}'$ such that $rule^{\mathcal{S}'}(x)$ is a sending rule and $\boldsymbol{l} \in body^{\mathcal{S}'}(x)|_{\gamma_{\text{in}}}$. Let $\mathcal{T}' \in F$ be the tree structurally equivalent to $\mathcal{S}'$, and abbreviate $n' = map_{\mathcal{S}',\mathcal{T}'}$. By construction of Val, we have $Val(\boldsymbol{l}) \in body^{\mathcal{T}'}(n'(x))$. Since $val^{\mathcal{T}'}(n'(x))$ was satisfied during some run, which follows from $\mathcal{T}' \in F$, and all runs have the same input facts, we obtain $I \models Val(\boldsymbol{l})$.

Succeed Input Now consider the subformula $succeed_{G,\mathcal{S},\kappa}$. This formula is specified as

$$succeed_{G,\mathcal{S},\kappa} := succeed^{\text{in}}_{G,\mathcal{S},\kappa} \wedge succeed^{\text{deny}}_{G,\mathcal{S},\kappa}.$$

Let $\mathcal{S}_0$ and $\mathcal{T} \in F$ be as above: $\mathcal{S}$ is the truncated version of $\mathcal{S}_0$ and $\mathcal{T}$ is structurally equivalent to $\mathcal{S}_0$. Abbreviate $n = map_{\mathcal{S}_0,\mathcal{T}}$.

[20] Nonequalities in rules of G are satisfied under their valuations because they are satisfied in F.

Similarly to $sndMsg_G$, the subformula $succeed^{\text{in}}_{G,\mathcal{S},\kappa}$ is a conjunction of input literals. Let $\boldsymbol{l}$ be such a literal. We have to show $I \models Val(\boldsymbol{l})$. There exists a node $x \in a^{\mathcal{S}}$ such that $\boldsymbol{l} \in body^{\mathcal{S}}(x)|_{\Upsilon_{\text{in}}}$. By construction of Val, we have $Val(\boldsymbol{l}) \in body^{\mathcal{T}}(n(x))$. And similarly to our reasoning for $sndMsg_G$, we can now obtain that $I \models Val(\boldsymbol{l})$.

Succeed Deny Consider the subformula $succeed^{\text{deny}}_{G,\mathcal{S},\kappa}$. Let $x \in a^{\mathcal{S}}$, $y \in \beta^{\mathcal{S}}(x)$, denoting $\boldsymbol{g} = fact^{\mathcal{S}}(y)$, and $(\mathcal{S}', \lambda) \in align^{G}(\boldsymbol{g})$ with $\lambda(root^{\mathcal{S}'}) < \kappa(x)$. We have to show that $\neg succeed_{G,\mathcal{S}',\lambda}$ is true under Val, which amounts to showing that $succeed_{G,\mathcal{S}',\lambda}$ is false under Val. The main strategy will be to use that $\mathcal{S}'$ extended with Val fails in $\mathcal{R}$ when executed according to λ. The reasons for failure make (parts of) formula $succeed_{G,\mathcal{S}',\lambda}$ false.

First, we show that the fact $Val(\boldsymbol{g})$ has to be absent during (and before) transition $\kappa(x)$ of $\mathcal{R}$. By definition of y, we have $\neg\boldsymbol{g} \in body^{\mathcal{S}}(x)$. Let $\mathcal{S}_0, \mathcal{T} \in F$, and mapping n, be as above for the case "succeed input". We have $\neg Val(\boldsymbol{g}) \in Val(body^{\mathcal{S}}(x)) = body^{\mathcal{T}}(n(x))$. Now, because valuation $val^{\mathcal{T}}(n(x))$ is satisfying during transition $\kappa^{\mathcal{T}}(n(x)) = \kappa(x)$, $Val(\boldsymbol{g})$ must be absent during $\kappa(x)$. By inflationarity of the transducer, $Val(\boldsymbol{g})$ is thus also absent before $\kappa(x)$.

Let $(\mathcal{S}', \lambda)$ be as above. There must be an alpha node z of $\mathcal{S}'$ such that fact $Val(fact^{\mathcal{S}'}(z))$ is not derived during transition $\lambda(z)$ of $\mathcal{R}$ because otherwise $Val(fact^{\mathcal{S}'}(root^{\mathcal{S}'})) = Val(\boldsymbol{g})$ would be derived in transition $\lambda(root^{\mathcal{S}'}) < \kappa(x)$, which is false. Let z be the first of such failed nodes with respect to λ. Valuation $Val \circ val^{\mathcal{S}'}(z)$ is not satisfying for $rule^{\mathcal{S}'}(z)$ during transition $\lambda(z)$ of $\mathcal{R}$, and each reason is used to show that some part of formula $succeed_{G,\mathcal{S}',\lambda}$ is false under Val. We consider the different kinds of literal in $rule^{\mathcal{S}'}(z)$:

[Input] Suppose there is a literal $\boldsymbol{l} \in body^{\mathcal{S}'}(z)|_{\Upsilon_{\text{in}}}$ such that $I \not\models Val(\boldsymbol{l})$. Then the conjunction $succeed^{\text{in}}_{G,\mathcal{S}',\lambda}$, and hence the entire formula $succeed_{G,\mathcal{S}',\lambda}$, is false under Val because $succeed_{G,\mathcal{S}',\lambda}$ contains $\boldsymbol{l}$.

[Messages] Recall that $rule^{\mathcal{S}'}(z)$ is message-positive. Suppose that there is a fact $\boldsymbol{l} \in body^{\mathcal{S}'}(z)|_{\Upsilon_{\text{msg}}}$ such that $Val(\boldsymbol{l})$ is not delivered in transition $\lambda(z)$ of $\mathcal{R}$. We argue that this is actually not possible, so this case can not occur. First, because λ is an alignment of $\mathcal{S}'$ to the abstract canonical run $\mathcal{R}^G$, fact $\boldsymbol{l}$ is delivered in transition $\lambda(z)$ of $\mathcal{R}^G$. Hence, by Claim D.1, fact $Val(\boldsymbol{l})$ is delivered in transition $\lambda(z)$ of $\mathcal{R}$.

[Positive output and memory] Suppose there is a positive literal $\boldsymbol{l} \in body^{\mathcal{S}'}(z)|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$ (i.e., $\boldsymbol{l}$ is a fact) such that $Val(\boldsymbol{l})$ is not available during transition $\lambda(z)$ of $\mathcal{R}$. We will again show that this case can not occur. The existence of $\boldsymbol{l}$ implies that z has an alpha child-node z' in $\mathcal{S}'$ with $fact^{\mathcal{S}'}(z') = \boldsymbol{l}$. This implies $\lambda(z') < \lambda(z)$. Since z is the first failed alpha node of $\mathcal{S}'$ with respect to λ, it must be that the fact $Val(fact^{\mathcal{S}'}(z')) = Val(\boldsymbol{l})$ is derived in transition $\lambda(z')$. Hence, $Val(\boldsymbol{l})$ is available in transition $\lambda(z)$ by inflationarity of Π.

[Negative output and memory] Suppose there is a negative literal $\neg\boldsymbol{i} \in body^{\mathcal{S}'}(z)|_{\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}}$, such that $\boldsymbol{h} = Val(\boldsymbol{i})$ is present during transition $\lambda(z)$ of $\mathcal{R}$. From $\mathcal{R}$, we can extract a pair $(\mathcal{T}'', \lambda'')$ with $\mathcal{T}''$ a truncated derivation tree for

$\boldsymbol{h}$, and $\mathcal{S}''$ an alignment of $\mathcal{T}''$ to $\mathcal{R}$ according to which $\mathcal{T}''$ derives $\boldsymbol{h}$. Note that Val^{-1} exists because Val is injective. Now, let $\mathcal{S}''$ denote the tree obtained from $\mathcal{T}''$ by applying for each internal node u of $\mathcal{T}''$ the function Val^{-1} after $val^{\mathcal{T}''}(u)$. Note that $adom(\mathcal{S}'') \subseteq adom(G)$.

Because in $\mathcal{S}'$ there is a beta child node z' of z with $fact^{\mathcal{S}'}(z') = \boldsymbol{i}$, if we could show $(\mathcal{S}'', \lambda'') \in align^G(\boldsymbol{i})$, then formula $succeed^{\text{deny}}_{G,\mathcal{S}',\lambda}$ contains the subformula $\neg succeed_{G,\mathcal{S}'',\lambda''}$. Then, we can recursively show that $succeed_{G,\mathcal{S}'',\lambda''}$ is true under Val, making $succeed^{\text{deny}}_{G,\mathcal{S}',\lambda}$, and by extension $succeed_{G,\mathcal{S}',\lambda}$, false under Val, as desired. This is similar to our current proof where we show that $succeed_{G,\mathcal{S},\kappa}$ is true under Val, but we would replace $(\mathcal{S}, \kappa)$ by $(\mathcal{S}'', \lambda'')$. This recursive step always ends, as we argued at the end of Section 7.2.3.

We are left to show that $(\mathcal{S}'', \lambda'') \in align^G(\boldsymbol{i})$. First, $\mathcal{S}''$ derives the fact $Val^{-1}(\boldsymbol{h}) = Val^{-1}(Val(\boldsymbol{i})) = \boldsymbol{i}$. Next, alignment λ'' for $\mathcal{S}''$ schedules nodes before their ancestors because it also does this for $\mathcal{T}''$. For the last step, let u be an internal node of $\mathcal{S}''$. We have to show that each $\boldsymbol{e} \in body^{\mathcal{S}''}(u)|_{\Upsilon_{\text{msg}}}$ is delivered during transition $\lambda''(u)$ of $\mathcal{R}^G$. By construction of $\mathcal{S}''$ from $\mathcal{T}''$, there is some $\boldsymbol{e'} \in body^{\mathcal{T}''}(u)|_{\Upsilon_{\text{msg}}}$ that is delivered in transition $\lambda''(u)$ of $\mathcal{R}$ and $\boldsymbol{e} = Val^{-1}(\boldsymbol{e'})$. But by Claim D.1, we have $\boldsymbol{e'} \in Val(M^G_j)$ with $j = \lambda''(u)$. Hence, $\boldsymbol{e} \in Val^{-1} \circ Val(M^G_j) = M^G_j$, as desired.

D.3 Claims

Claim D.2 *Consider the symbols defined in* Section 7.2.3. *Let $G \subseteq forest_R$. Let F be a set of derivation trees of Π such that* (i) *no two trees are structurally equivalent;* (ii) *for each $\mathcal{T} \in F$ there is a structurally equivalent tree $\mathcal{S} \in G$; and,* (iii) *there is an injective function $Val : adom(G) \to adom(F)$ such that when Val is applied after the valuations of a tree $\mathcal{S} \in G$, we obtain the structurally equivalent tree $\mathcal{T} \in F$. Finally, let I be an input for $\boldsymbol{\mathcal{M}}$ such that formula $sndMsg_G$ is satisfied under Val with respect to I.*

Let $\mathcal{R}^G$ and $\mathcal{R}$ denote the canonical runs based on G and F respectively, that both have the same length n. Let $i \in \{1, \ldots, n\}$ and let M^G_i denote the (abstract) message set delivered in transition i of $\mathcal{R}^G$. In transition i of $\mathcal{R}$, we deliver precisely $Val(M^G_i)$.

Proof We show this by induction on i. For the base case ($i = 1$), the property holds because $M^G_i = \emptyset$ and no messages are delivered in the first transition of $\mathcal{R}$ (as no messages were previously sent).

For the induction hypothesis, assume the property holds for transitions $j = 1, \ldots, i-1$ with $i > 1$. For the inductive step, we show that the property is satisfied for transition i. First, note that at most $Val(M^G_i)$ can be delivered in transition i of $\mathcal{R}$, because this transition only delivers the messages needed by rules in F scheduled at i, and because the trees in F are obtained from those in G by concatenating Val to their valuations.

For the second direction, let $\boldsymbol{g} \in M^G_i$ and denote $\boldsymbol{h} = Val(\boldsymbol{g})$. We show that $\boldsymbol{h}$ is delivered in transition i of $\mathcal{R}$. Since $\boldsymbol{g} \in M^G_i$, there is a tree $\mathcal{S}' \in G$, and an internal

node x of $\mathcal{S}'$, such that $\kappa^{\mathcal{S}'}(x) = i$ and $\boldsymbol{g} \in body^{\mathcal{S}'}(x)|_{\gamma_{\text{msg}}}$. By message-positivity of $rule^{\mathcal{S}'}(x)$, there is a child y of x such that $fact^{\mathcal{S}'}(y) = \boldsymbol{g}$. From the definition of the canonical scheduling, we have $\kappa^{\mathcal{S}'}(y) = \kappa^{\mathcal{S}'}(x) - 1$. Denoting $j = \kappa^{\mathcal{S}'}(y)$, we have $j = i - 1$. We show that $Val \circ val^{\mathcal{S}'}(y)$ is satisfying for $rule^{\mathcal{S}'}(y)$ during transition j, such that $Val(\boldsymbol{g}) = \boldsymbol{h}$ is sent in transition j, and can be delivered in (the next) transition i. The nonequalities of $rule^{\mathcal{S}'}(y)$ are satisfied because they are satisfied under $val^{\mathcal{S}'}(y)$ (by construction of G) and because Val is injective. Next, because $rule^{\mathcal{S}'}(y)$ is static, we only have to consider input and message atoms:

- Let $\boldsymbol{l} \in body^{\mathcal{S}'}(y)|_{\gamma_{\text{in}}}$. We have, $I \models Val(\boldsymbol{l})$, as desired, because $\boldsymbol{l}$ is added to $sndMsg_G$, which is true under Val with respect to I.
- Let $\boldsymbol{l} \in body^{\mathcal{S}'}(y)|_{\gamma_{\text{msg}}}$. Because $rule^{\mathcal{S}'}(y)$ is message-positive, $\boldsymbol{l}$ is a fact. Moreover, we have $\boldsymbol{l} \in M_j^G$. By applying the induction hypothesis to transition j, we know that $Val(\boldsymbol{l})$ is delivered during transition j, as desired.

□

Claim D.2 *Let $\mathcal{T}$ and $\mathcal{S}$ be two structurally equivalent derivation trees of Π, that derive the same output or memory fact $\boldsymbol{f}$. Abbreviate $n = map_{\mathcal{S},\mathcal{T}}$. For each $x \in a^{\mathcal{S}}$, the valuations $val^{\mathcal{S}}(x)$ and $val^{\mathcal{T}}(n(x))$ assign the same values to the free variables of the rule $rule^{\mathcal{S}}(x) = rule^{\mathcal{T}}(n(x))$.*[21]

Proof We show the property by induction on the length of the path from the root to the node $x \in a^{\mathcal{S}}$ in question. In the base case, simply $x = root^{\mathcal{S}}$ and $n(x) = root^{\mathcal{T}}$. We are given that $fact^{\mathcal{S}}(root^{\mathcal{S}}) = fact^{\mathcal{T}}(root^{\mathcal{T}})$. Hence, valuations $val^{\mathcal{S}}(root^{\mathcal{S}})$ and $val^{\mathcal{T}}(root^{\mathcal{T}})$ assign the same values to free variables. Moreover, because $\boldsymbol{f}$ is an output or memory fact, $rule^{\mathcal{S}}(root^{\mathcal{S}})$ is message-bounded, and thus any variable occurring in an output or memory literal in the body must be a free variable. Hence, for every alpha child y of $root^{\mathcal{S}}$, we have $fact^{\mathcal{S}}(y) = fact^{\mathcal{T}}(n(y))$. The reasoning can now be repeated for y. □

References

1. Abiteboul, S., Bienvenu, M., Galland, A., et al.: A rule-based language for Web data management. In: Proceedings 30th ACM Symposium on Principles of Database Systems, pp. 293–304. ACM Press (2011)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
3. Abiteboul, S., Vianu, V., et al.: Relational transducers for electronic commerce. J. Comput. Syst. Sci. **61**(2), 236–269 (2000)
4. Alvaro, P., Conway, N., Hellerstein, J., Marczak, W.R.: Consistency analysis in Bloom: a CALM and collected approach. In: Proceedings 5th Biennial Conference on Innovative Data Systems Research, pp. 249–260. www.cidrdb.org (2011)
5. Alvaro, P., Marczak, W., et al.: Dedalus: datalog in time and space. Technical Report EECS-2009-173, University of California, Berkeley (2009)

[21] Node x and $n(x)$ have the same rule by structural equivalence.

6. Ameloot, T.J.: Deciding correctness with fairness for simple transducer networks. In: Proceedings of the 17th International Conference on Database Theory, pp. 84–95. OpenProceedings.org (2014)
7. Ameloot, T.J., Ketsman, B., Neven, F., Zinn, D.: Weaker forms of monotonicity for declarative networking: a more fine-grained answer to the CALM-conjecture. In: Proceedings of the 33rd ACM Symposium on Principles of Database Systems, pp. 64–75. ACM Press (2014)
8. Ameloot, T.J., Neven, F., Van den Bussche, J.: Relational transducers for declarative networking. In: Proceedings 30th ACM Symposium on Principles of Database Systems, pp. 283–292. ACM Press (2011)
9. Ameloot, T.J., Van den Bussche, J.: Deciding eventual consistency for a simple class of relational transducer networks. In: Proceedings of the 15th International Conference on Database Theory, pp. 86–98. ACM Press (2012)
10. Chandra, A.K., Vardi, M.Y.: The implication problem for functional and inclusion dependencies is undecidable. SIAM J. Comput. **14**(3), 671–677 (1985)
11. Deutsch, A.: Personal communication (2011)
12. Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Proceedings 12th International Conference on Database Theory (2009)
13. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven Web applications. J. Comput. Syst. Sci. **73**(3), 442–474 (2007)
14. Deutsch, A., Sui, L., Vianu, V., Zhou, D.: Verification of communicating data-driven Web services. In: Proceedings 25th ACM Symposium on Principles of Database Systems, pp. 90–99. ACM Press (2006)
15. Grumbach, S., Wang, F.: Netlog, a rule-based language for distributed programming. In: Carro, M., Peña, R (eds.) Proceedings 12th International Symposium on Practical Aspects of Declarative Languages, volume 5937 of Lecture Notes in Computer Science, pp. 88–103 (2010)
16. Hellerstein, J.M.: Datalog redux: experience and conjecture. Video available (under the title "The Declarative Imperative") from http://db.cs.berkeley.edu/jmh/. PODS 2010 keynote (2010)
17. Hellerstein, J.M.: The declarative imperative: experiences and conjectures in distributed logic. SIGMOD Rec. **39**(1), 5–19 (2010)
18. Loo, B.T., et al.: Declarative networking. Commun. ACM **52**(11), 87–95 (2009)
19. Marczak, W.R., Alvaro, P., Conway, N., Hellerstein, J.M., Maier, D.: Confluence analysis for distributed programs: a model-theoretic approach. In: Barceló, P., Pichler, R. (eds.) Datalog, volume 7494 of Lecture Notes in Computer Science, pp. 135–147. Springer, Berlin (2012)
20. Navarro, J.A., Rybalchenko, A.: Operational semantics for declarative networking. In: Gill, A., Swift, T. (eds.) Proceedings 11th International Symposium on Practical Aspects of Declarative Languages, volume 5419 of Lecture Notes in Computer Science, pp. 76–90 (2009)
21. Post, E.L.: A variant of a recursively unsolvable problem. Bull. Am. Math. Soc. **52**(4), 264–268 (1946)
22. Sipser, M.: Introduction to the Theory of Computation, Second Edition, International Edition. Thomson Course Technology, Boston (2006)
23. Spielmann, M.: Verification of relational transducers for electronic commerce. J. Comput. Syst. Sci. **66**(1), 40–65 (2003)
24. Vogels, W.: Eventual consistency. Commun. ACM **52**(1), 40–44 (2009)
25. Zinn, D., Green, T.J., Ludaescher, B.: Win-move is coordination-free. In: Proceedings of the 15th International Conference on Database Theory, pp. 99–113. ACM Press (2012)