Deciding Correctness with Fairness for Simple Transducer Networks

Tom J. Ameloot*
Hasselt University &
Transnational University of Limburg
Diepenbeek, Belgium
tom.ameloot@uhasselt.be

ABSTRACT

Ensuring the correctness of a distributed system is an important challenge. Previously, two interpretations of correctness have been proposed: the first interpretation is about determinism, saying that all infinite fair computation traces produce the same output; and, the second interpretation is a confluence notion, saying that all finite computation traces can still be extended to produce the same output. A decidability result for the confluence notion was previously obtained for so-called simple transducer networks, a model from the field of declarative networking. In the current paper, we also present a decidability result for simple transducer networks, but this time for the first interpretation of correctness, with infinite fair computation traces. We also compare the expressivity of simple transducer networks under both interpretations.

Categories and Subject Descriptors

H.2 [**Database Management**]: Languages; H.2 [**Database Management**]: Systems—*Distributed databases*; F.1 [**Computation by Abstract Devices**]: Models of Computation

General Terms

languages, theory

Keywords

distributed database, relational transducer, consistency, decidability, expressive power, cloud programming

1. INTRODUCTION

Cloud environments have emerged as a modern way to store and manipulate data [26, 12]. Essentially, a cloud is a distributed system that should produce output as the result of some computation. A big challenge in distributed systems is dealing with the nondeterminism that is caused by the parallel execution of the compute nodes and the underlying network infrastructure. A typical example of such nondeterminism is the unpredictable delay on messages, called *asynchronous communication*, and the permutation on message sequences that are caused by these delays [9].

From this viewpoint, we may call a distributed system *correct* when its output is not affected by the above nondeterminism. In this paper, we will assume that the output can not be retracted once it is produced.

Useful insights have already emerged about how correctness might be obtained. One approach is to let the compute nodes run monotone programs that steadily accumulate messages [1, 4, 21, 5]. Another approach lies in the design of so-called commutative replicated datatypes, where the messages represent commutative operations, that are thus resilient to nondeterministic reorderings [22, 23, 13, 10]. If the task at hand is not monotone, however, a form of coordination has to be used to enforce correctness [27, 5]. There is ongoing theoretical research about the cost and complexity of coordination [19, 11].

A method that is complementary to the above would be to automatically check whether a distributed system is correct. That approach is the focus of the current paper. One can expect correctness to be undecidable in general, but an investigation might nevertheless shed some light on tradeoffs between expressivity and decision complexity. Moreover, such an investigation might inform us about which strategy will be the most viable in practice: do we build distributed programs by composing mechanisms that guarantee correctness, as above, or do we just allow any programs to be created that are checked for correctness afterward? First, it should be noted that different interpretations of correctness have already been studied in the literature.

In one interpretation, a system is called correct if all infinite computation traces produce the same output. Generally, only traces are considered that are "fair". As fairness conditions, we typically demand basic liveness properties: all compute nodes are infinitely often triggered, and all sent messages are eventually delivered [1, 5]. We refer to this interpretation as *consistency*, with the opposite being *inconsistency*.[1] Intuitively, if a system is consistent, it is determinate: as long as the conditions on the network are sufficiently good, i.e., fair, the system succeeds in obtaining the desired output, no matter how messages might be

*PhD Fellow of the Fund for Scientific Research, Flanders (FWO).

[1] Abiteboul et al. [1] refer to consistency as *convergence*.

delayed.

It is also possible to understand correctness as a confluence notion [6]. We call a system *confluent* if for any two *finite* computation traces on the same input, the second trace can be finitely extended to also produce the (partial) output of the first trace. The opposite of confluence is *diffluence*. Intuitively, a system is confluent whenever any two finite traces can be made to flow together, to achieve the same output. Note that this might require the arrangement of somewhat artificial conditions, like the simultaneous delivery of messages, or the delivery of a certain sequence of messages without intermittent delays. These arrangements might not be provided by an infinite fair trace!

To rephrase, the big difference between these interpretations is that a confluent system can, in principle, at any point in its execution still steer its computation towards the desired output, in some particular way. A consistent system on the other hand always has to achieve the right output when only the basic fairness conditions are provided: all nodes become active infinitely often and all messages are eventually delivered. It appears that consistency is a stronger requirement. Indeed, every consistent system is also confluent: whenever we have two finite traces, they can both always be extended to infinite fair traces that produce (in finite time) the same output by consistency; hence, the second finite trace can be finitely extended to achieve the output of the first finite trace.

We can also provide an example of a confluent system that is not consistent: we have a single compute node that sends two messages to itself repeatedly, and, the node will produce output only when both messages are delivered simultaneously. We call this simultaneous delivery a *message join*. In some fair computation traces, this arrangement never happens; these traces produce no output, whereas other fair traces do.

In this paper, we will only be concerned with consistency and confluence, as defined above. For completeness, however, we should also mention the terms *eventual consistency* and *strong consistency* [25, 18, 4, 13]. The former term is used to describe systems that achieve correctness without resorting to (heavy) coordination, whereas the latter term refers to systems that do the opposite. In this paper, we want to check whether a system is correct, regardless of how much coordination it uses. From this viewpoint, eventual and strong consistency seem to be orthogonal to our terms consistency and confluence.

Previously, we have investigated the decidability of confluence for distributed systems formalized as relational transducer networks [6, 7]. Transducer networks are a computational model for declarative networking [27, 5], building upon the established model of relational transducers for data-centric agents [3, 24, 15, 14, 16]. Transducer networks appear suitable to model and study database-oriented distributed systems. The main result from our previous work [6, 7] is that, although confluence is undecidable in general, it actually is decidable for a syntactically defined fragment of transducer networks, the so-called *simple transducer networks*, that are implemented with restricted conjunctive queries. Formally, the result is that *diffluence* for simple transducer networks is NEXPTIME-complete.

But consistency might be more practically relevant than confluence. Indeed, a distributed system should not depend on specially arranged message deliveries, as is allowed in confluence, but it should already work correctly when only the basic fairness conditions are satisfied, as in consistency. Deciding consistency, as described above, is a question left open by the previous work [6, 7]. The current paper works towards filling this gap for simple transducer networks. At this point, it should be noted that a confluent simple transducer network is not automatically consistent, because the above example of a confluent system that is not consistent can be implemented as a simple transducer network.[2] Indeed, in line with the above discussion, the author expected that deciding consistency is harder. Interestingly, the main contribution of the current paper is to show that deciding *inconsistency* for simple transducer networks is in fact also NEXPTIME-complete.

Regarding expressivity, under the *confluence semantics*, the output of a confluent simple transducer network on an input is defined as the union of the (partial) outputs that are produced by all finite computation traces on that input. With this definition, it was shown that simple transducer networks capture the distributed queries expressible by unions of conjunctive queries with negation [6, 7]. Although this indicates that simple transducer networks are too limited for general purpose distributed systems, they are not completely useless as unions of conjunctive queries with negation seem an attractive and widely used query language.

Under the *consistency semantics*, we define the output of a consistent simple transducer network on an input as the output produced by an arbitrary infinite fair computation trace on that input. This is sensible because consistency ensures that all infinite fair computation traces on the same input produce the same output. At first sight, this definition appears substantially different from the confluence semantics above. The second contribution of this paper is to confirm that the expressivity remains unchanged: consistent simple transducer networks, under the consistency semantics, also capture all distributed queries expressible by unions of conjunctive queries with negation.

This paper is organized as follows. First, Section 2 provides preliminaries on database notions and (simple) transducer networks, and, formally defines consistency and confluence. Section 3 shows example simple transducer networks. Next, Section 4 presents the NEXPTIME decidability result for consistency of simple transducer networks, along with the essential proof steps. The expressivity of consistent simple transducer networks under the consistency semantics is investigated in Section 5, also accompanied with the proof. We conclude in Section 6.

2. PRELIMINARIES

2.1 Database Basics

We first recall some basic notions from database theory [2]. A *database schema* $\mathcal{D}$ is a finite set of pairs (R, k), often written as $R^{(k)}$, where R is a *relation name* and $k \in \mathbb{N}$ its associated *arity*. A relation name occurs at most once in a database schema.

We assume some infinite universe **dom** of atomic data values. A *fact* $\boldsymbol{f}$ is a pair $(R, \bar{a})$, often written as $R(\bar{a})$, where R is a relation name, called the *predicate*, and $\bar{a}$ is a tuple of values in **dom**. We say that a fact $R(a_1, \ldots, a_k)$ is

[2]See the later Example 3.1.

over database schema $\mathcal{D}$ if $R^{(k)} \in \mathcal{D}$. A database *instance* I over $\mathcal{D}$ is a finite set of facts over $\mathcal{D}$. For a subset $\mathcal{D}' \subseteq \mathcal{D}$, we write $I|_{\mathcal{D}'}$ to denote the subset of facts in I whose predicate is a relation name in $\mathcal{D}'$. The *active domain* of I, denoted $adom(I)$, is the set of data values occurring in facts of I. For a fact $\boldsymbol{f}$, we also write $adom(\boldsymbol{f})$ to denote the set of values occurring in just $\boldsymbol{f}$.

2.2 Distributed Databases and Queries

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 node $x \in \mathcal{N}$ to an ordinary database schema. A *distributed database instance* over $\mathcal{E}$ is a function that maps each node $x \in \mathcal{N}$ to an ordinary database instance over the local schema $\eta(x)$.

A *distributed query* $\mathcal{Q}$ over an input distributed schema $\mathcal{E}_1$ and an output distributed schema $\mathcal{E}_2$, where $\mathcal{E}_1$ and $\mathcal{E}_2$ share the same network, is a function that maps instances over $\mathcal{E}_1$ to instances over $\mathcal{E}_2$.

For each instance H over a distributed schema $\mathcal{E} = (\mathcal{N}, \eta)$, we define $adom(H) = \bigcup_{x \in \mathcal{N}} adom(H(x))$.

2.3 Unions of Conjunctive Queries

We recall the query language *unions of conjunctive queries with (safe) negation*, abbreviated UCQ$^\neg$. It will be convenient to use a slightly unconventional formalization of conjunctive queries.

Let **var** be a universe of *variables*, disjoint from **dom**. An *atom* is a pair $(R, \overline{u})$, often written as $R(\overline{u})$, where R is a relation name, called the *predicate*, and $\overline{u}$ is a tuple of variables in **var**. A *literal* is an atom, or an atom with "$\neg$" prepended; these literals are respectively called *positive* and *negative*.

A *conjunctive query* φ, or simply *rule*, is a 4-tuple

$$(head^\varphi, pos^\varphi, neg^\varphi, non^\varphi)$$

where $head^\varphi$ is an atom; pos^φ and neg^φ are sets of atoms; non^φ is a set of nonequalities of the form $x \neq y$ with $x, y \in$ **var**; and, all variables of φ occur in pos^φ. Note that neg^φ contains atoms, not negative literals. The components $head^\varphi$, pos^φ and neg^φ are called respectively the *head*, the *positive body atoms* and the *negative body atoms*.

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

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

Let $\mathcal{D}$ be a database schema. A rule φ is said to be *over* $\mathcal{D}$ if for each atom $R(u_1, \ldots, u_k) \in pos^\varphi \cup neg^\varphi$ we have $R^{(k)} \in \mathcal{D}$. Suppose φ is over $\mathcal{D}$. A *valuation* for φ is a total function V from the variables of φ to **dom**. The *application* of V to an atom $R(u_1, \ldots, u_k)$ of φ, denoted $V(R(u_1, \ldots, u_k))$, is defined as the *fact* $R(a_1, \ldots, a_k)$ where for $i = 1, \ldots, k$ we have $a_i = V(u_i)$. This notation is naturally extended to a set of atoms, which results in a set of facts. Now, let I be an instance over $\mathcal{D}$. Valuation V is said to be *satisfying for* φ *on* I if $V(pos^\varphi) \subseteq I$, $V(neg^\varphi) \cap I = \emptyset$, and $V(x) \neq V(y)$ for each $(x \neq y) \in non^\varphi$. If this is so, we call (φ, V) a *derivation pair*, that *derives* the fact $V(head^\varphi)$. The *result of* φ *applied to* I, denoted $\varphi(I)$, is the set of facts derived by all satisfying valuations for φ on I.

A *union of conjunctive queries with negation* over $\mathcal{D}$ is a finite set Φ of rules over $\mathcal{D}$ that all share the same head predicate T and head arity k; we call $T^{(k)}$ the *target relation*. We also call Φ a *UCQ$^\neg$ program*, and the language of such programs is denoted UCQ$^\neg$. Let I be an instance over $\mathcal{D}$. The *result of* Φ *applied to* I, denoted $\Phi(I)$, is defined as $\bigcup_{\varphi \in \Phi} \varphi(I)$. Note that if $\Phi = \emptyset$ then $\Phi(I) = \emptyset$.

2.4 Transducers

We recall (relational) transducers [3, 24, 15, 14, 16]. 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 with disjoint relation names, with the restriction that $\Upsilon_{\text{sys}} = \{Id^{(1)}, All^{(1)}\}$. We call these database schemas respectively "input", "output", "message", "memory" and "system". A *transducer state* for Υ is a database instance over $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$. Intuitively, in such a state, relation *Id* will provide at each node of a network the identity of this node, and relation *All* will provide the identities of all the nodes in this network. This will be made more concrete in the next subsection.

We now formalize the transducer model. For technical simplicity, we immediately restrict attention to transducers implemented with UCQ$^\neg$ programs.[3] Formally, a *transducer* Π over Υ is a collection of UCQ$^\neg$ programs, each over 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 UCQ$^\neg$ program Φ^R_{out} having target relation $R^{(k)}$;
- for each $R^{(k)} \in \Upsilon_{\text{mem}}$ there are UCQ$^\neg$ programs Φ^R_{ins} and Φ^R_{del} both having target relation $R^{(k)}$;
- for each $R^{(k)} \in \Upsilon_{\text{msg}}$ there is a UCQ$^\neg$ program Φ^R_{snd} having target relation $R^{(k+1)}$.

Note the extra component for message relations; it serves to indicate the addressee of each message, which is by convention the first component (see Section 2.5).

The above programs form the mechanism to update local storage and to send messages. Formally, 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: abbreviating $I' = I \cup I_{\text{rcv}}$,

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

where, following the presentation in [27],

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

Intuitively, on receipt of I_{rcv}, a local transition updates the old state I to the new state J and sends the facts in J_{snd}. Compared to I, in J potentially more output facts

[3] General transducers are studied in [27, 5].

are produced; and, the update semantics for each memory relation R adds the facts produced by *insertion* program Φ^R_{ins} and removes the facts produced by *deletion* program Φ^R_{del}.[4] Output facts can only grow. 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 $J_{\text{snd}} = J'_{\text{snd}}$.

2.5 Transducer Networks

A *transducer network* $\boldsymbol{\mathcal{N}}$ is a triple $(\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ where $\mathcal{N}$ is a network, $\boldsymbol{\Upsilon}$ is a function mapping each node $x \in \mathcal{N}$ to a transducer schema, and $\boldsymbol{\Pi}$ is a function mapping each node $x \in \mathcal{N}$ to a transducer over schema $\boldsymbol{\Upsilon}(x)$. For technical convenience, we assume that all transducer schemas use the same message relations (with the same arities). This is not really a restriction because the transducers are not obliged to use all message relations.

We define the distributed database schema $in^{\boldsymbol{\mathcal{N}}}$ that maps each $x \in \mathcal{N}$ to $\boldsymbol{\Upsilon}(x)_{\text{in}}$. Any distributed database instance H over $in^{\boldsymbol{\mathcal{N}}}$ can be given as input to $\boldsymbol{\mathcal{N}}$. A *configuration of* $\boldsymbol{\mathcal{N}}$ *on* H is a pair $\rho = (s, b)$ of functions s and b where for each $x \in \mathcal{N}$,

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

Intuitively, a configuration describes a snapshot of the network at some moment. We call s the *state* function and b the *(message) buffer* function. For each $x \in \mathcal{N}$, inside $s(x)$, the subinstance $H(x)$ provides the local input of x, and the system relations *Id* and *All* provide respectively the identity of x and the identities of all nodes. Next, $b(x)$ is a multiset of message facts, with the intuition that these are the messages sent to x but that are not yet delivered. A multiset allows us to represent duplicates of the same message (sent at different times).

The *start configuration*, denoted $start(\boldsymbol{\mathcal{N}}, H)$, is the unique configuration $\rho = (s, b)$ of $\boldsymbol{\mathcal{N}}$ on H 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 is a submultiset of $b_1(x)$, and letting m' be m collapsed to a set, there exists a J_{snd} such that
$$s_1(x),\ m' \to s_2(x),\ J_{\text{snd}},$$
is a local transition of transducer $\boldsymbol{\Pi}(x)$;
- for each $y \in \mathcal{N} \setminus \{x\}$ we have $s_2(y) = s_1(y)$;
- regarding the message buffers, letting $J^{\to z}_{\text{snd}} = \{R(\bar{a}) \mid R(z, \bar{a}) \in J_{\text{snd}}\}$ for each $z \in \mathcal{N}$, we have
  (i) $b_2(x) = (b_1(x) \setminus m) \cup J^{\to x}_{\text{snd}}$; and,
  (ii) for each $y \in \mathcal{N} \setminus \{x\}$ we have $b_2(y) = b_1(y) \cup J^{\to y}_{\text{snd}}$,

  where we use multiset difference and union.

We call x the *active node*, m the *delivered messages*, ρ_1 the *source* configuration, and ρ_2 the *target* configuration. Intuitively, we select an arbitrary node x and allow it to receive some arbitrary submultiset m from its message buffer, collapsed to a set. Next, x performs a local transition, in which the local output and memory is updated, and possibly new messages are sent. The first component of each fact in J_{snd} is regarded as the addressee, and this component is projected away during the transfer of the message to the buffer of that addressee. Messages having an addressee outside $\mathcal{N}$ are lost. If $m = \emptyset$, the global transition is called a *heartbeat*.

A *run* $\mathcal{R}$ of $\boldsymbol{\mathcal{N}}$ on H is a sequence of global transitions, where the source configuration of the first global transition is $start(\boldsymbol{\mathcal{N}}, H)$, and the target configuration of each global transition is the source configuration of the next global transition. Runs can be finite or infinite, and we will always indicate which option is taken. Transition ordinals start at one. Note that there is no bound on how long a message has to wait in a buffer, giving rise to asynchronous communication. We will often refer to global transitions simply as "transitions".

2.5.1 Fairness

When infinite runs are considered, in the literature on process models it is customary to require certain "fairness" conditions [17, 8, 20]. Let $\boldsymbol{\mathcal{N}}$ be a transducer network. An infinite run of $\boldsymbol{\mathcal{N}}$ on some input instance is called *fair* if the run satisfies the following conditions: *(i)* each node of $\boldsymbol{\mathcal{N}}$ is made active infinitely often; and, *(ii)* if a message $\boldsymbol{f}$ appears infinitely often in the buffer of a node x then there are an infinite number of transitions in which $\boldsymbol{f}$ is delivered to x. The second fairness condition intuitively means that every sent message is eventually delivered.[5]

Note that every transducer network has an infinite fair run for every input because heartbeats are still possible when the message buffers are empty.

2.5.2 Output of Runs

Let $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a transducer network. In addition to the distributed schema $in^{\boldsymbol{\mathcal{N}}}$ (defined above), we also define the distributed schema $out^{\boldsymbol{\mathcal{N}}}$ that maps each node $x \in \mathcal{N}$ to $\boldsymbol{\Upsilon}(x)_{\text{out}}$.

Let H be an instance over $in^{\boldsymbol{\mathcal{N}}}$. Let $\mathcal{R}$ be a finite or infinite run of $\boldsymbol{\mathcal{N}}$ on H. The *output* of $\mathcal{R}$, denoted $out(\mathcal{R})$, is defined as the instance J over $out^{\boldsymbol{\mathcal{N}}}$ where, for each $x \in \mathcal{N}$, the set $J(x)$ consists of all output facts produced at x during $\mathcal{R}$. Note that $J(x)$ is always finite for each $x \in \mathcal{N}$ because $\boldsymbol{\mathcal{N}}$ can not invent new values.

2.5.3 Simple Transducer Networks

We recall some syntactical restrictions on individual transducers and on transducer networks as a whole [6, 7]. Let Π be a transducer over a schema Υ. For an individual rule φ of Π, we consider the following restrictions:

[4] We follow the convention that there is no-op semantics in case a fact is both inserted and deleted at the same time [24].

[5] This condition even ensures that messages are eventually delivered that are sent only a finite number of times: if they would not get delivered, they would appear infinitely often in buffers *without* being infinitely often delivered (violating the condition).

- We say that φ is *message-positive* if there are no message atoms in neg^{φ}.[6]
- We say that φ is *static* if φ does not use output or memory relations in its body.
- We say that φ is *message-bounded* if every bound variable (i.e., not occurring in the head) occurs in a positive message atom of the body, and does not occur in output or memory atoms of the body (positive or negative). This is an application of the more general notion of "input-boundedness" [24, 16, 15].

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 the body of a rule in Π with head predicate R.[7]
- We say that Π is *inflationary* if the deletion programs of memory relations are empty. This means that Π can not delete memory facts once they are produced.

Transducer Π is called *simple* if

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

Let $\boldsymbol{\mathcal{N}}$ be a transducer network. The transducer network $\boldsymbol{\mathcal{N}}$ is called *globally recursion-free* if there are no cycles in the *positive message dependency graph* of $\boldsymbol{\mathcal{N}}$, which is the graph having as vertices the (shared) message relations of $\boldsymbol{\mathcal{N}}$ and there is an edge from relation R to relation S if S occurs positively in the body of a rule with head predicate R in some transducer of $\boldsymbol{\mathcal{N}}$.

Lastly, transducer network $\boldsymbol{\mathcal{N}}$ is called *simple* if

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

2.6 Consistency and Confluence

We formalize the different interpretations of correctness that we have mentioned in the introduction: consistency and confluence. Let $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a transducer network.

We call $\boldsymbol{\mathcal{N}}$ *consistent* if for each input H, for any two infinite fair runs $\mathcal{R}_1$ and $\mathcal{R}_2$ on H, we have $out(\mathcal{R}_1) = out(\mathcal{R}_2)$. The opposite of consistency is *inconsistency*.

We call $\boldsymbol{\mathcal{N}}$ *confluent* if for each input H, for any two finite runs $\mathcal{R}_1$ and $\mathcal{R}_2$ on H, if there is a node $x \in \mathcal{N}$ and a fact $\boldsymbol{f} \in out(\mathcal{R}_1)(x)$ then we can extend $\mathcal{R}_2$ to a finite run $\mathcal{R}_2'$ such that $\boldsymbol{f} \in out(\mathcal{R}_2')(x)$. The opposite of confluence is *diffluence*.

[6]This appears to be a natural constraint because message delivery is asynchronous: typically, it seems that one would make sure that every message of interest has arrived before applying negation (amounting to negation on a memory relation instead).

[7]Intuitively, focusing on the positive dependency graph will limit the length of the derivation history of each output, memory, and message fact. For messages we also enforce this globally; see below.

3. EXAMPLE PROGRAMS

We give two examples of simple transducer networks.

EXAMPLE 3.1. *We give an inconsistent simple transducer network. We have a single node x, for which the transducer schema Υ is as follows: $\Upsilon_{\text{in}} = \{R^{(1)}, S^{(1)}\}$, $\Upsilon_{\text{out}} = \{T^{(1)}\}$, $\Upsilon_{\text{msg}} = \{A^{(1)}_{\text{msg}}, B^{(1)}_{\text{msg}}\}$, and $\Upsilon_{\text{mem}} = \emptyset$.[8] The transducer rules at x are as follows:*

$$A_{\text{msg}}(n, u) \leftarrow R(u),\ Id(n).$$
$$B_{\text{msg}}(n, u) \leftarrow S(u),\ Id(n).$$
$$T(u) \leftarrow A_{\text{msg}}(u),\ B_{\text{msg}}(u).$$

This single-node simple transducer network is confluent, but not consistent. Indeed, some infinite fair runs might jointly deliver A_{msg}- and B_{msg}-facts, whereas other infinite fair runs might not. □

EXAMPLE 3.2. *We give a consistent simple transducer network. We have three nodes x, y, and z. The input schemas at x and y consist respectively of a single relation $R^{(1)}$ and a single relation $S^{(1)}$. Node z has an output relation $T^{(1)}$. The shared message relations are $A^{(1)}_{\text{msg}}$ and $B^{(1)}_{\text{msg}}$.*

We give x the following sending rule:

$$A_{\text{msg}}(n, u) \leftarrow R(u),\ All(n).$$

We give y the following sending rule:

$$B_{\text{msg}}(n, u) \leftarrow A_{\text{msg}}(u),\ S(u),\ All(n).$$

Lastly, we give z the following output rule:

$$T(u) \leftarrow B_{\text{msg}}(u).$$

Note that this simple transducer network is consistent: if R and S are nonempty at respectively node x and y, and R and S share a value a, then in each run, x will send $A_{\text{msg}}(a)$ to (at least) y, and y will subsequently send $B_{\text{msg}}(a)$ to (at least) z, and z outputs $T(a)$. □

We note that the syntactical presence of message joins by itself is not sufficient to cause inconsistency: rules with a message join might have no satisfiable valuation on any input, hence, have no effect on the computation. Moreover, even in the absence of message joins, inconsistency (and diffluence) can arise [6, 7]: for example, we can output received message-facts of a first type as long as a certain memory fact is absent, but the memory fact is created upon receiving a message-fact of a second type; so, depending on the relative delivery orders of these two message types, some finite runs can not be extended anymore to produce the outputs of other runs.

4. CONSISTENCY DECIDABILITY

We recall the following result:

PROPOSITION 4.1. *([6, 7]) Deciding diffluence for simple transducer networks is* NEXPTIME*-complete.*

One of the difficulties of the diffluence decision problem is that a property of an infinite state system needs to be verified. Indeed, there are infinitely many inputs and even for a

[8]Recall that always $\Upsilon_{\text{sys}} = \{Id^{(1)}, All^{(1)}\}$.

fixed input there are infinitely many configurations because message buffers have no size limit. Naturally, these difficulties are carried over when we want to decide (in)consistency. The main contribution of the current paper is that the following decidability result also holds:

THEOREM 4.2. *Deciding inconsistency for simple transducer networks is* NEXPTIME*-complete.*

The NEXPTIME lower bound is briefly discussed in Section 4.1. The next subsections will sketch the essential proof ideas of the upper bound: Section 4.2 discusses a small model property, and, Section 4.3 gives the concrete NEXPTIME decision procedure with correctness proof. The major technically difficult result of our paper is Claim 4.4, discussed in Section 4.3.2.

4.1 Lower Bound

The NEXPTIME lower bound follows from the construction already provided in [6, 7], of which we repeat the main idea. Each problem A from NEXPTIME is reduced to the inconsistency decision problem. Concretely, fixing a NEXPTIME Turing machine M for A, for each input string w, we transform w into a simple single-node transducer network $\mathcal{N}$ that checks whether its own input describes an accepting trace of M on w. If so, the single node sends the message $accept()$ to itself. Next, when $accept()$ is received, $\mathcal{N}$ does something inconsistent, as in Example 3.1.

4.2 Small Model Property

The small model property of [6, 7], for simple transducer networks with only one node, can be generalized to simple transducer networks with multiple nodes and infinite fair runs.[9] This is formalized next.

Let $\mathcal{N} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a simple transducer network, where $|\mathcal{N}| \geq 1$. Suppose $\mathcal{N}$ is inconsistent. We show that $\mathcal{N}$ is inconsistent on an input instance over $in^{\mathcal{N}}$ whose active domain size is bounded by an expression over the syntactical properties of $\mathcal{N}$. For this purpose, consider the following syntactically defined quantities about $\mathcal{N}$:

- the length $\mathtt{P}$ of the longest path in the positive message dependency graph of $\mathcal{N}$ (defined in Section 2.5.3), where the length of a path is measured here as the number of vertices on this path;
- the largest number $\mathtt{B}$ of positive body atoms in any rule, over all transducers;
- the largest arity $\mathtt{I}$ among input relations, over all transducers;
- the largest arity $\mathtt{O}$ among output relations, over all transducers;
- the number $\mathtt{C}$ of different output and memory facts that can be made with values in A, where $A \subseteq \mathbf{dom}$ is an arbitrary set with $|A| = \mathtt{O}$, and output and memory relations are taken over all transducers.

[9]This generalization is technically more convenient for the current paper, whereas the previous decidability result [6, 7] used a simulation of a multi-node network on a single-node network.

We write $|\mathcal{N}|$ to denote the size of a reasonable encoding of $\mathcal{N}$.[10] Note that all quantities except $\mathtt{C}$ are linear in $|\mathcal{N}|$. Also, $\mathtt{C} \leq k\mathtt{O}^a$, where k is the total number of output and memory relations in $\mathcal{N}$ and a is the largest arity among output and memory relations. We define $sizeDom(\mathcal{N}) = 2\mathtt{ICB}^{(\mathtt{P}+2)}$. Note that both $\mathtt{C}$ and $sizeDom(\mathcal{N})$ are single-exponential in $|\mathcal{N}|$.

We have the following small model property:

LEMMA 4.3. *If $\mathcal{N}$ is inconsistent, then $\mathcal{N}$ is inconsistent on an input J over $in^{\mathcal{N}}$ for which $|adom(J)| \leq sizeDom(\mathcal{N})$.*

The proof is based on a straightforward extension of the technique in [6, 7].

4.3 Decision Procedure

Let $\mathcal{N} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a simple transducer network. In this section, we provide a NEXPTIME procedure to decide if $\mathcal{N}$ is inconsistent. Concretely, this procedure will accept on at least one computation branch if and only if $\mathcal{N}$ is inconsistent.

We first define auxiliary expressions over syntactical properties of $\mathcal{N}$. Let $\mathtt{P}$, $\mathtt{B}$, $\mathtt{C}$, and $sizeDom(\mathcal{N})$ be as defined in Section 4.2. We define $\mathbf{numMsg} = |\mathcal{N}| \cdot r \cdot sizeDom(\mathcal{N})^a$, where r is the number of message relations and a is the largest message arity; this is an upper bound on the number of messages that a node can send during each transition when an input distributed database instance H is given with $|adom(H)| = sizeDom(\mathcal{N})$.[11] We also define $\mathbf{runLen} = \mathtt{CB}^{(\mathtt{P}+1)}\mathbf{numMsg}^{\mathtt{P}}$. Note that both **numMsg** and **runLen** are single-exponential in $|\mathcal{N}|$.

We also need some additional terminology. Letting $C \subseteq \mathbf{dom}$, we call a fact $\boldsymbol{f}$ a *C-fact* if $adom(\boldsymbol{f}) \subseteq C$.

The decision procedure does the following steps:

1. Guess an input instance H over $in^{\mathcal{N}}$ that satisfies $|adom(H)| \leq sizeDom(\mathcal{N})$.
2. Guess two finite runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\mathcal{N}$ on H, that do at most **runLen** transitions.
3. Choose a *focus* node $x \in \mathcal{N}$ and an output fact $\boldsymbol{f}$ such that $\boldsymbol{f} \in out(\mathcal{S}_1)(x)$ and $\boldsymbol{f} \notin out(\mathcal{S}_2)(x)$. Reject if no such node and fact can be chosen.
4. We extend $\mathcal{S}_2$ with $\mathtt{P} + 1$ *rounds* as follows. In each round, do the following for each node y: do one heartbeat, and, letting b denote the buffer of y at the beginning of the round, deliver to y all unique messages in b one after the other, jointly delivered with their duplicates in b (if any). To rephrase, different messages are never delivered simultaneously (see also Section 4.3.3).
5. Letting $C = adom(\boldsymbol{f})$, accept if no new output or memory C-fact is created at x in the above extension of $\mathcal{S}_2$.[12] Otherwise reject.

[10]For example, $\mathcal{N}$ can be encoded as a sequence of nodes, transducer schemas, and transducers whose rules are written in full; and, where all input and output relations of a transducer schema are effectively used in the rules of the corresponding transducer [7].

[11]In the expression of **numMsg**, the number of addressees is represented by $|\mathcal{N}|$.

[12]In particular, note that $\boldsymbol{f}$ was not created in the extension.

We remark that the overall procedure can be implemented in nondeterministic procedural code, so that each computation branch takes single-exponential time. We give the correctness proof in Sections 4.3.1 and 4.3.2. In Section 4.3.3, we highlight some important differences between the design of the inconsistency decision procedure of the current paper and the diffluence decision procedure of our previous work [6, 7].

4.3.1 Correctness 1

We start with the technically simplest direction: suppose that the procedure accepts. We show that $\boldsymbol{\mathcal{N}}$ is inconsistent.

We first observe what happened on an accepting computation branch of the procedure on $\boldsymbol{\mathcal{N}}$. To start, the procedure has guessed an input H over $in^{\mathcal{N}}$. Next, it has guessed two finite runs $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\boldsymbol{\mathcal{N}}$ on H such that there is a node $x \in \mathcal{N}$ and a fact $\boldsymbol{f}$ that satisfy $\boldsymbol{f} \in out(\mathcal{S}_1)(x)$ and $\boldsymbol{f} \notin out(\mathcal{S}_2)(x)$. Denote $C = adom(\boldsymbol{f})$. Lastly, the procedure has extended $\mathcal{S}_2$ with $\mathrm{P}+1$ rounds, of the form described in the decision procedure, to obtain a finite run $\mathcal{S}_2'$. The procedure has observed that no new output or memory C-facts were created at x in $\mathcal{S}_2'$ after $\mathcal{S}_2$.

We use the guessed finite runs to show that $\boldsymbol{\mathcal{N}}$ is inconsistent on H. Note that $\mathcal{S}_1$ can surely be extended to an infinite fair run $\mathcal{R}_1$ of $\boldsymbol{\mathcal{N}}$ on H, in which $\boldsymbol{f}$ is thus output at x. Next, let $\mathcal{R}_2$ denote an extension of $\mathcal{S}_2'$ obtained by doing an infinite number of rounds of the form described in the procedure. Run $\mathcal{R}_2$ is fair because in each round, all nodes become active at least once, and all messages at the beginning of the round are delivered. We show that $\boldsymbol{f}$ is never created at x during $\mathcal{R}_2$, demonstrating that $\boldsymbol{\mathcal{N}}$ is indeed inconsistent. We show more concretely that x produces no new output or memory C-facts in $\mathcal{R}_2$ after $\mathcal{S}_2'$. Hence, using the above observation of the decision procedure, $\boldsymbol{f} \notin out(\mathcal{R}_2)(x)$.

Towards a contradiction, suppose that x produces a new output or memory C-fact $\boldsymbol{g}$ during some transition k in $\mathcal{R}_2$ after $\mathcal{S}_2'$, by means of a derivation pair (φ, V) (defined in Section 2.3). We assume that $\boldsymbol{g}$ is the *first* new C-fact that x creates after $\mathcal{S}_2'$. We show that (φ, V) also (newly) derives $\boldsymbol{g}$ during some transition of x in $\mathcal{S}_2'$ after $\mathcal{S}_2$, which contradicts the above observation of the decision procedure.

We consider the body literals of φ in turn, and argue that they are satisfied under valuation V during some transition of x in $\mathcal{S}_2'$ after $\mathcal{S}_2$. First, because $\mathcal{S}_2'$ and $\mathcal{R}_2$ operate on the same network and the same input, the system and input literals of φ are satisfied under V during every transition of x in $\mathcal{S}_2'$ after $\mathcal{S}_2$. In those transitions, the output and memory literals of φ are also satisfied under V because they refer to C-facts by message-boundedness ($\boldsymbol{g}$ is a C-fact), and those transitions see exactly the same output and memory C-facts as transition k of $\mathcal{R}_2$: we use the above observation of the decision procedure and the assumption that $\boldsymbol{g}$ is the first new C-fact at x in $\mathcal{R}_2$ after $\mathcal{S}_2'$.

We are left to show that any messages needed by (φ, V) are delivered to x in some round of $\mathcal{S}_2'$ after $\mathcal{S}_2$, which, combined with the argumentation for the other body literals above, gives rise to a particular transition of x during which all body literals of φ are satisfied under V. Recall that message literals in φ are positive because $\boldsymbol{\mathcal{N}}$ is simple. Moreover, (φ, V) can require at most one message by design of $\mathcal{R}_2$. If no message is required, (φ, V) will already derive $\boldsymbol{g}$ in the very first round of $\mathcal{S}_2'$. If (φ, V) requires one message $\boldsymbol{h}$ to be delivered, then, starting at transition k of $\mathcal{R}_2$, we can follow the derivation history of $\boldsymbol{h}$ backwards, and collect derivation pairs for recursively needed messages (if any). We stop going backward when we encounter a needed message that does not require other messages, or when we encounter a needed message delivered to a node z of which a copy also occurs in the last buffer of z in $\mathcal{S}_2$. This second condition causes us to stop tracing the derivation history (the latest) when we reach the tail of $\mathcal{S}_2$. Hence, each encountered message in the derivation history requires at most one other message by design of the rounds after $\mathcal{S}_2$, i.e., the derivation history of $\boldsymbol{h}$ is a *chain*. This backward chain is of length at most P by recursion-freeness. Next, using that sending rules are message-positive and static, we can show by forward induction on this derivation history that $\boldsymbol{h}$ is delivered to x the latest in round $\mathrm{P}+1$ of $\mathcal{S}_2'$; this induction starts at the first round of $\mathcal{S}_2'$ after $\mathcal{S}_2$.

4.3.2 Correctness 2

Let $\boldsymbol{\mathcal{N}}$ be as above. Suppose $\boldsymbol{\mathcal{N}}$ is inconsistent. We go sequentially over the steps of the decision procedure, and show that at least one computation branch accepts.

Step 1.

By the small model property (Lemma 4.3), there is an input H over $in^{\mathcal{N}}$ with $|adom(H)| \leq sizeDom(\boldsymbol{\mathcal{N}})$ on which $\boldsymbol{\mathcal{N}}$ is inconsistent: there are two infinite fair runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\boldsymbol{\mathcal{N}}$ on H such that there is a node $x \in \mathcal{N}$ and a fact $\boldsymbol{f}$ that satisfy $\boldsymbol{f} \in out(\mathcal{R}_1)(x)$ and $\boldsymbol{f} \notin out(\mathcal{R}_2)(x)$. Denote $C = adom(\boldsymbol{f})$.

The procedure can guess an instance isomorphic to H, but for technical simplicity we assume that the procedure guesses exactly H.

Step 2.

Next, we guess two finite runs of at most **runLen** transitions. First, for an infinite fair run $\mathcal{R}$, we define the function $fin^{\mathcal{R}}$ that maps each node $y \in \mathcal{N}$ to the set of message facts $\boldsymbol{g}$ for which there are only a finite number of transitions in $\mathcal{R}$ during which $\boldsymbol{g}$ is sent to y. We call $fin^{\mathcal{R}}(y)$ the *finite messages* of y during $\mathcal{R}$. Also, for a multiset m and a set A, we write $m|_A$ to denote the multiset m' containing all elements of m that also occur in A, with the same cardinalities they had in m.

Letting x and C be as above, consider the following claim:

CLAIM 4.4. *Let $\mathcal{R}$ be an infinite fair run of $\boldsymbol{\mathcal{N}}$ on H. For every finite prefix $\mathcal{R}'$ of $\mathcal{R}$, there exists a finite run $\mathcal{S}$ of $\boldsymbol{\mathcal{N}}$ on H, such that*

- *$\mathcal{S}$ does at most* **runLen** *transitions;*
- *$s_1(x)$ and $s_2(x)$ contain the same output and memory C-facts, where $s_1(x)$ and $s_2(x)$ are the states of x at the end of respectively $\mathcal{S}$ and $\mathcal{R}'$; and,*
- *for each $y \in \mathcal{N}$, abbreviating $F_y = fin^{\mathcal{R}}(y)$, we have that $b_1(y)|_{F_y}$ is contained in $b_2(y)|_{F_y}$, where $b_1(y)$ and $b_2(y)$ are the message buffers of y at the end of respectively $\mathcal{S}$ and $\mathcal{R}'$.*

PROOF. We sketch the proof. The first condition of the claim is an effect of the way we construct $\mathcal{S}$ to satisfy the other two conditions. The main idea in this construction is as follows.

To satisfy the condition on output and memory C-facts at x, we first collect the derivation history of messages in $\mathcal{R}'$ that are needed to create the output and memory C-facts of x during $\mathcal{R}'$: we "mark" the transitions in which the needed messages are sent, and what derivation pairs they use. Messages might recursively depend on each other, and their derivation history may span multiple nodes. By recursion-freeness, at most $\mathbf{cb}^{(\mathbf{P}+1)}$ transitions are marked in this way. If we only had to satisfy the condition on C-facts, we could execute these marked transitions and deliver just the needed messages. But, the claim also demands an additional condition on finite messages.

To satisfy the condition on finite messages, we continuously detect the sending of finite messages that accidentally originate in the marked transitions as a side-effect of generating the needed messages; and, we deliver these accidental messages in additional transitions, to imitate the way that $\mathcal{R}'$ delivers them. Each of those marked transitions from above can send at most **numMsg** finite messages, each of which might require its own additional delivery transition, in each of which again at most **numMsg** finite messages are sent, etc. We can show that at most $\mathbf{numMsg}^{\mathbf{P}}$ additional delivery transitions should be provided for each of the individual $\mathbf{cb}^{(\mathbf{P}+1)}$ marked transitions. In total, $\mathcal{S}$ contains at most **runLen** transitions. □

Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be as above. For $i \in \{1,2\}$, letting A_i denote the set of all output and memory C-facts created at x during the entire run $\mathcal{R}_i$, let $\mathcal{R}'_i$ be a prefix of $\mathcal{R}_i$ that already creates all of A_i at x.[13] Next, for $i \in \{1,2\}$, let $\mathcal{S}_i$ be a finite run corresponding to $\mathcal{R}'_i$ as implied by Claim 4.4. Note that each $\mathcal{S}_i$ does at most **runLen** transitions, so the procedure can guess them.

Step 3.

For $i \in \{1,2\}$, by Claim 4.4, run $\mathcal{S}_i$ creates precisely the same output and memory C-facts at x as in $\mathcal{R}'_i$, and by extension $\mathcal{R}_i$. Hence, letting $\boldsymbol{f}$ be as above, we have $\boldsymbol{f} \in \mathit{out}(\mathcal{S}_1)(x)$ and $\boldsymbol{f} \notin \mathit{out}(\mathcal{S}_2)(x)$. So, the procedure can choose node x and fact $\boldsymbol{f}$ to focus on in step 3.

Steps 4 and 5.

Let $\mathcal{S}'_2$ be an extension of $\mathcal{S}_2$ as performed by the rounds of step 4. We show that no new output or memory C-facts are created at x in this extension after $\mathcal{S}_2$, causing the procedure to accept, as desired.

Towards a proof by contradiction, suppose that a new output or memory C-fact $\boldsymbol{g}$ is created at x, during some new transition k in $\mathcal{S}'_2$, by means of a derivation pair (φ, V). We assume that $\boldsymbol{g}$ is the *first* newly created C-fact at x in the extension. We show that V is also satisfying for φ during a transition of x in the suffix of $\mathcal{R}_2$, i.e., after $\mathcal{R}'_2$, causing $\boldsymbol{g}$ to be derived there as well, which is impossible by choice of the prefix $\mathcal{R}'_2$.

First, we show there is a transition j of x in the suffix of $\mathcal{R}_2$ in which the messages required by (φ, V) are delivered. By construction of transition k, the pair (φ, V) actually requires at most one message. If no message is required, transition j can be any transition of x in the suffix of $\mathcal{R}_2$, which surely exists by fairness of $\mathcal{R}_2$. And if (φ, V) requires a message $\boldsymbol{h}$ to be delivered, Claim 4.5 (below) also guarantees the existence of transition j.

We now show that (φ, V) is satisfying during transition j of $\mathcal{R}_2$, giving the contradiction mentioned above. First, during j, the system and input literals of φ are satisfied under V because $\mathcal{S}'_2$ and $\mathcal{R}_2$ operate on the same network and the same input. The messages required by (φ, V) (at most one) are delivered in transition j by choice of j. Next, any output or memory facts positively required by (φ, V), which are C-facts by message-boundedness, must be in the last configuration of $\mathcal{S}_2$ because $\boldsymbol{g}$ is by assumption the first new C-fact created at x in the extension. Hence, by construction of $\mathcal{S}_2$ (Claim 4.4), these facts are in the last configuration of $\mathcal{R}'_2$, making them available during transition j. Lastly, any output or memory facts whose absence is required by (φ, V), again C-facts by message-boundedness, must, by inflationarity, be absent from the last configuration of $\mathcal{S}_2$. So, again by construction of $\mathcal{S}_2$, they are absent from the last configuration of $\mathcal{R}'_2$. Now, using the assumption that no more output and memory C-facts are created at x in $\mathcal{R}_2$ after $\mathcal{R}'_2$, they are also absent during transition j.

Claim 4.5. *For each $y \in \mathcal{N}$, every message delivered to y in the extension $\mathcal{S}'_2$ after $\mathcal{S}_2$ is also delivered to y at least once in the suffix of $\mathcal{R}_2$.*

Proof. We show by induction on the rounds i of $\mathcal{S}'_2$ that whenever a message is delivered to a node $y \in \mathcal{N}$ in round i then this message is also delivered to y at least once in the suffix of $\mathcal{R}_2$.

For the first round, if a message $\boldsymbol{g}$ is delivered to a node y, this message is in the last buffer of y in $\mathcal{S}_2$. If $\boldsymbol{g}$ is a finite message of y during $\mathcal{R}_2$ then by construction of $\mathcal{S}_2$ (Claim 4.4), message $\boldsymbol{g}$ is also in the last buffer of y in $\mathcal{R}'_2$. Hence, by fairness, $\boldsymbol{g}$ will be delivered to y in the suffix of $\mathcal{R}_2$. If $\boldsymbol{g}$ is not a finite message, i.e., $\boldsymbol{g}$ is sent to y an infinite number of times during $\mathcal{R}_2$, then again by fairness, $\boldsymbol{g}$ will be delivered (infinitely often) in the suffix of $\mathcal{R}_2$.

For the inductive step, suppose we deliver a message $\boldsymbol{g}$ to a node y in round i, with $i \geq 2$. Because the buffer of y at the beginning of round $i-1$ was completely delivered during round $i-1$, message $\boldsymbol{g}$ must have been sent during round $i-1$ by some node z, with possibly $z = y$. Let (φ, V) be a derivation pair that made z send $\boldsymbol{g}$ to y during round $i-1$. We show that V is also satisfying for φ in the suffix of $\mathcal{R}_2$, making z also send $\boldsymbol{g}$ to y in this suffix, which, by fairness, causes $\boldsymbol{g}$ to be delivered to y in the suffix.

Because $\mathcal{S}'_2$ and $\mathcal{R}_2$ operate on the same network and the same input, the system and input literals of φ are satisfied under V during each transition of z in the suffix of $\mathcal{R}_2$, and such transitions surely exist by fairness of $\mathcal{R}_2$. Now, by construction of round $i-1$, the pair (φ, V) can require at most one message. If no message is required, then z will send $\boldsymbol{g}$ to y during each transition of z in the suffix of $\mathcal{R}_2$. And if one message is required, this message was delivered to z in round $i-1$, and hence this message is delivered to z in the suffix of $\mathcal{R}_2$ by applying the induction hypothesis. □

4.3.3 Comparison with Previous Work

We contrast the inconsistency decision procedure of the current paper with the diffluence decision procedure in our previous work [6, 7]. We first remark that if the diffluence decision procedure has an accepting computation branch then the input simple transducer network is also inconsistent (because consistency implies confluence). Although the

[13] In each infinite fair run, the output and memory C-facts of x are always created in a finite prefix, because there are only a finite number of facts that can be created.

overall outline of the diffluence decision procedure is very much like the inconsistency decision procedure of the current paper, step 4 is substantially different: the rounds of the diffluence decision procedure deliver all possible messages simultaneously, in effect providing all possible opportunities to derive output facts. Intuitively, using the notations of Section 4.3, if the output fact $\boldsymbol{f}$ is not created at x this way, then no finite extension of $\mathcal{S}_2$ can do this either (giving diffluence). However, the simultaneous delivery of all possible messages is an unsuitable strategy to detect inconsistent simple transducer networks that are still confluent (like Example 3.1): we have to look for infinite fair runs that avoid creating $\boldsymbol{f}$ at x, even though any finite run can be finitely extended to produce this fact (using some right message deliveries). For this reason, we have modified the diffluence decision procedure to obtain the inconsistency decision procedure, as follows.

First, we have used the new design of rounds in step 4 of the procedure. In particular, within the setting of Section 4.3.2, the rounds now only do *necessary* message deliveries: the deliveries that will surely occur in the suffix of $\mathcal{R}_2$ after $\mathcal{R}'_2$. This property is formalized by Claim 4.5. Intuitively, because the suffix of $\mathcal{R}_2$ does not succeed in producing $\boldsymbol{f}$ at x, the newly designed rounds will not do this either because they generate no additional opportunities to derive $\boldsymbol{f}$ at x with respect to $\mathcal{R}_2$. Technically, this is achieved by never delivering messages simultaneously, so that any newly sent messages during the rounds also naturally arise in the suffix of $\mathcal{R}_2$. But Claim 4.5 depends on appropriately handling the so-called *finite* messages of $\mathcal{R}_2$, which are the messages that are only finitely often sent in $\mathcal{R}_2$. This issue does not appear in the previous work because only finite runs were considered there. Now, such finite messages are either (accidentally) caused by jointly delivering at least two messages, or by delivering another finite message. It could be that some opportunities to create $\boldsymbol{f}$ at x were wasted during the prefix $\mathcal{R}'_2$ by delivering finite messages in the wrong order or too soon. This has to be reflected in the last configuration of $\mathcal{S}_2$, because otherwise the rounds of step 4 can still use these finite messages to produce $\boldsymbol{f}$ at x. For this reason, we have modified how the finite run $\mathcal{S}_2$ is constructed from the prefix $\mathcal{R}'_2$, to specifically deal with finite messages (Claim 4.4). This claim is the biggest technical challenge with respect to the previous work.

Of course, the above modifications still have to ensure that if the inconsistency decision procedure has an accepting computation branch then the input simple transducer network is really inconsistent. This is the direction discussed in Section 4.3.1. To achieve this, the new design of the rounds also guarantees that if we repeat the rounds forever then we obtain an infinite fair run (that avoids creating the output $\boldsymbol{f}$ at x). Indeed, during each round, each node becomes active at least once and all messages at the beginning of the round are delivered.

5. EXPRESSIVITY

Here we study the expressivity of consistent simple transducer networks under the infinite fair run semantics. First, Section 5.1 formalizes the queries computed by (general) transducer networks, for both the consistency and confluence semantics. Section 5.2 defines how UCQ$^\neg$ can directly express distributed queries. The expressivity result is presented in Section 5.3, along with proofs for the upper and lower bound.

5.1 Computing Distributed Queries

Let $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a transducer network, not necessarily simple. Recall the distributed schemas $in^{\boldsymbol{\mathcal{N}}}$ and $out^{\boldsymbol{\mathcal{N}}}$ from Section 2.5.

If $\boldsymbol{\mathcal{N}}$ is consistent, we say that $\boldsymbol{\mathcal{N}}$ computes the following distributed query under the *consistency semantics*, with input schema $in^{\boldsymbol{\mathcal{N}}}$ and output schema $out^{\boldsymbol{\mathcal{N}}}$: an input H over $in^{\boldsymbol{\mathcal{N}}}$ is mapped to the instance J over $out^{\boldsymbol{\mathcal{N}}}$ defined as $J = out(\mathcal{R})$, where $\mathcal{R}$ is an arbitrary infinite fair run of $\boldsymbol{\mathcal{N}}$ on H. This mapping is well-defined because $\boldsymbol{\mathcal{N}}$ is consistent: all runs of $\boldsymbol{\mathcal{N}}$ on input H produce the same output.

If $\boldsymbol{\mathcal{N}}$ is confluent, we say that $\boldsymbol{\mathcal{N}}$ computes the following distributed query under the *confluence semantics* [6, 7], also with input schema $in^{\boldsymbol{\mathcal{N}}}$ and output schema $out^{\boldsymbol{\mathcal{N}}}$: an input H over $in^{\boldsymbol{\mathcal{N}}}$ is mapped to the instance J over $out^{\boldsymbol{\mathcal{N}}}$ where $J(x)$ for each $x \in \mathcal{N}$ is the union, over all finite runs of $\boldsymbol{\mathcal{N}}$ on H, of the output facts produced at x in those runs. Note that when $\boldsymbol{\mathcal{N}}$ is confluent, all finite runs on H can be extended to obtain J.

5.2 Expressing with UCQ$^\neg$

To study the expressivity of simple transducer networks, we formalize how a distributed query can be expressed directly in UCQ$^\neg$. First, for a distributed schema $\mathcal{E} = (\mathcal{N}, \eta)$, we define the ordinary database schema

$$\langle \mathcal{E} \rangle = \{x.R^{(k)} \mid x \in \mathcal{N},\, R^{(k)} \in \eta(x)\} \cup \{x.Id^{(1)} \mid x \in \mathcal{N}\} \cup \{node^{(1)}\}.$$

Intuitively, we write the node name in front of the local relation names, and we provide relations to contain node identifiers. Similarly, for an instance H over $\mathcal{E}$, we define the following ordinary instance over $\langle \mathcal{E} \rangle$:

$$\langle H \rangle = \{x.R(\bar{a}) \mid x \in \mathcal{N},\, R(\bar{a}) \in H(x)\} \cup \{x.Id(x),\, node(x) \mid x \in \mathcal{N}\}.$$

Now, we say that a distributed query $\mathcal{Q}$ over input schema $\mathcal{E}_1 = (\mathcal{N}, \eta_1)$ and output schema $\mathcal{E}_2 = (\mathcal{N}, \eta_2)$ is *expressible in UCQ$^\neg$* if for each $x \in \mathcal{N}$ and each $R^{(k)} \in \eta_2(x)$, we can give an UCQ$^\neg$ program $\Phi_{x,R}$ over input schema $\langle \mathcal{E}_1 \rangle$ with target relation $R^{(k)}$ such that for all instances H over $\mathcal{E}_1$ we have

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

Intuitively, we give an UCQ$^\neg$ program to compute local output relation R at node x. This program is given access to all facts on the network, and all node identifiers.

5.3 Expressivity Result

Our previous work [6, 7] has shown that the expressivity of confluent simple transducer networks, under the confluence semantics, coincides with the distributed queries expressible in UCQ$^\neg$. It is not obvious whether the expressivity would remain unchanged when we replace confluence by consistency, because, as remarked in the Introduction, confluence might require message deliveries not provided by infinite fair runs. But we can indeed confirm the following:

THEOREM 5.1. *Consistent simple transducer networks, under the consistency semantics, capture the distributed queries expressible in UCQ$^\neg$.*

For the upper bound, we can actually use the previously mentioned confluence expressivity result, as will be explained in Section 5.3.1. The lower bound requires a new technical construction, demonstrated in Section 5.3.2.

5.3.1 Upper Bound

Let $\boldsymbol{\mathcal{N}} = (\mathcal{N}, \boldsymbol{\Upsilon}, \boldsymbol{\Pi})$ be a consistent simple transducer network, that expresses a distributed query $\mathcal{Q}$ under the consistency semantics. We show that $\mathcal{Q}$ is expressible in UCQ$^\neg$. It suffices to show that $\boldsymbol{\mathcal{N}}$ is confluent and also computes $\mathcal{Q}$ under the confluence semantics, because then we can apply the previous result [6, 7] to know that $\mathcal{Q}$ is expressible in UCQ$^\neg$.

To show that $\boldsymbol{\mathcal{N}}$ is confluent, we formally repeat the intuitive argument mentioned in the Introduction.[14] Consider an input H for $\boldsymbol{\mathcal{N}}$. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two finite runs of $\boldsymbol{\mathcal{N}}$ on H. Suppose there is a node $x \in \mathcal{N}$ and a fact $\boldsymbol{f} \in \mathit{out}(\mathcal{R}_1)(x)$. If $\mathcal{R}_2$ has not already output $\boldsymbol{f}$ at x, we have to show that $\mathcal{R}_2$ can still be extended to do so. To start, $\mathcal{R}_1$ and $\mathcal{R}_2$ can clearly be extended to infinite fair runs, that we will denote as $\mathcal{R}_1'$ and $\mathcal{R}_2'$ respectively. Note that $\boldsymbol{f} \in \mathit{out}(\mathcal{R}_1')(x)$. Now, by consistency of $\boldsymbol{\mathcal{N}}$, fact $\boldsymbol{f}$ is also produced at x in $\mathcal{R}_2'$, in some finite prefix. Hence, $\mathcal{R}_2$ can be finitely extended to output $\boldsymbol{f}$ at x.

Now we show that $\boldsymbol{\mathcal{N}}$ computes $\mathcal{Q}$ under the confluence semantics. Let H be an input for $\boldsymbol{\mathcal{N}}$. Let J be the output of $\boldsymbol{\mathcal{N}}$ on H under the confluence semantics. We show that $J = \mathcal{Q}(H)$. Let $x \in \mathcal{N}$.

- Let $\boldsymbol{f} \in \mathcal{Q}(H)(x)$. By definition of $\mathcal{Q}(H)$, fact $\boldsymbol{f}$ is produced at x in some arbitrary infinite fair run of $\boldsymbol{\mathcal{N}}$ on H. So, $\boldsymbol{f}$ is produced in some finite prefix of this run. Hence, $\boldsymbol{f} \in J(x)$.
- Let $\boldsymbol{f} \in J(x)$. By definition of J, fact $\boldsymbol{f}$ is produced at x in some finite run of $\boldsymbol{\mathcal{N}}$ on H. This finite run can be extended to an infinite fair run. By consistency of $\boldsymbol{\mathcal{N}}$, all infinite fair runs produce the same output. Hence, $\boldsymbol{f} \in \mathcal{Q}(H)(x)$.

5.3.2 Lower Bound

Let $\mathcal{Q}$ be a distributed query over input distributed schema $\mathcal{E}_1 = (\mathcal{N}, \eta_1)$ and output distributed schema $\mathcal{E}_2 = (\mathcal{N}, \eta_2)$. Suppose $\mathcal{Q}$ is expressible in UCQ$^\neg$. More concretely, for some $x \in \mathcal{N}$, suppose we have a rule φ over $\langle \mathcal{E}_1 \rangle$ describing (part of) a local output relation T in $\eta_2(x)$. Rule φ can be written in the following canonical form:

$$T(\overline{u}) \leftarrow \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}$$

where

$$\mathbf{A} = y_1.R_1(\overline{v_1}), \ldots, y_m.R_m(\overline{v_m});$$

$$\mathbf{B} = \mathit{node}(a_1), \ldots, \mathit{node}(a_n);$$

$$\mathbf{C} = \neg z_1.S_1(\overline{w_1}), \ldots, \neg z_p.S_p(\overline{w_p});$$

$$\mathbf{D} = \neg\mathit{node}(b_1), \ldots, \neg\mathit{node}(b_q); \text{ and,}$$

$\mathbf{E}$ denotes a sequence of nonequalities. Below, we describe a transducer network $\boldsymbol{\mathcal{N}}$ over $\mathcal{N}$ to produce at node x, in its output relation T, the facts generated by φ. In $\boldsymbol{\mathcal{N}}$, we give each node $x \in \mathcal{N}$ its local input schema and local output schema as prescribed by $\eta_1(x)$ and $\eta_2(x)$ respectively. There will be no memory relations; we will only use sending rules and output rules.

If the description of $\mathcal{Q}$ in UCQ$^\neg$ consists of multiple rules, like more rules for output relation T at x, or rules to describe other output relations (also at other nodes), then the construction below can be repeated and the partial transducer networks thus obtained can be united, as long as new names are used for the auxiliary message relations, to avoid unwanted name clashes.

Rules. For technical simplicity we assume that body parts $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ are nonempty; the construction can be easily adapted to cases where one or more of these parts is empty. The main idea is that the body literals of φ, except those with predicate *node*, are sequentially evaluated on respectively the nodes $y_1, \ldots, y_m, z_1, \ldots, z_p$. This sequential behavior is obtained by chaining messages.

First, assuming some arbitrary total order on **var**, for each $i \in \{1, \ldots, m\}$, we define $\overline{t_i}$ to be an ordering of all unique variables in $\{a_1, \ldots, a_n\}$ united with all unique variables in $\overline{v_1}$ up to and including $\overline{v_i}$. Note that $\overline{t_m}$ contains all variables of φ because all variables of φ occur in positive body atoms (see Section 2.3).

We now add sending rules to our transducer network under construction. We use the base name '$x.T$' for all message relations to indicate that we are making messages for output relation T at x.[15] Each message is broadcasted, but only the intended addressee will have a sending rule to appropriately react. Let $a \in \mathbf{var}$ be a variable not yet used in φ. To start, we put the following sending rule on node y_1:

$$x.T_1^+(a, \overline{t_1}) \leftarrow \mathbf{B}', R_1(\overline{v_1}), \mathit{All}(a),$$

where $\mathbf{B}'$ is $\mathbf{B}$ but now with relation *node* replaced by relation *All*. The symbol '+' in the head indicates that we are evaluating the positive body literals of φ. Next, for $i = 2, \ldots, m$, we put the following sending rule on node y_i:

$$x.T_i^+(a, \overline{t_i}) \leftarrow x.T_{i-1}^+(\overline{t_{i-1}}), R_i(\overline{v_i}), \mathit{All}(a).$$

Note that the set of head variables grows monotonously with each new sending rule, and all variables of φ are present in the head of the rule for relation $x.T_m^+$.

Next, we will evaluate the negative literals of φ in a distributed fashion. To start, we put the following sending on node z_1:

$$x.T_1^-(a, \overline{t_m}) \leftarrow x.T_m^+(\overline{t_m}), \mathbf{D}', \neg S_1(\overline{w_1}), \mathit{All}(a),$$

where $\mathbf{D}'$ is $\mathbf{D}$ but now with relation *node* replaced by relation *All*. Next, for $i = 2, \ldots, p$, we put the following sending rule on node z_i:

$$x.T_i^-(a, \overline{t_m}) \leftarrow x.T_{i-1}^-(\overline{t_m}), \neg S_i(\overline{w_i}), \mathit{All}(a).$$

Lastly, we put the following output rule on node x:

$$T(\overline{u}) \leftarrow x.T_p^-(\overline{t_m}), \mathbf{E}.$$

In this last rule, we check all nonequalities and we project variables in the same way as φ.

[14] This argument works in general, i.e., also for the case that $\boldsymbol{\mathcal{N}}$ is not simple.

[15] Any auxiliary message relations we use are assumed to be added to the transducer schemas of the nodes.

Discussion. It can be verified that all the above sending rules are message-positive and static. The output rule is message-positive and message-bounded. The transducers are inflationary because there are no memory relations. Moreover, because of the message chain, the transducers of $\mathcal{N}$ are locally recursion-free and the entire network is also globally recursion-free. We conclude that $\mathcal{N}$ is simple.

Lastly, $\mathcal{N}$ is consistent: on every input H over $in^{\mathcal{N}}$, if the original rule φ has a satisfying valuation V on $\langle H \rangle$, fairness will allow the corresponding chain of messages to execute successfully in every run on H because the chain does not require the simultaneous delivery of messages.

6. CONCLUSION

Our previous work [6, 7] presented decidability of confluence for simple transducer networks. But consistency, as formalized here with infinite fair runs, might be preferred over confluence: a distributed system should already work correctly when only the basic fairness conditions are satisfied, instead of waiting for specially arranged message deliveries as is allowed in confluence. The current paper complements the previous result, by showing decidability of consistency for simple transducer networks. Moreover, the expressivity of simple transducer networks under both correctness notions turns out to be the same.

In further work, one could try to relax some of the syntactic restrictions of simple transducer networks, for both confluence and consistency. We might speculate that, because deciding confluence and deciding consistency have the same complexity for simple transducer networks, and yet consistency appears to be a stronger condition, it might be possible to more easily relax the syntactic restrictions for the confluence case.

One might also investigate in more detail how consistency and confluence relate. As already mentioned in the Introduction, consistency implies confluence. But there are examples of confluent programs that are not consistent. For such examples, typically the program is confluent because there exists some right order of message deliveries to achieve the output, and this order does not always occur in infinite fair computation traces. However, by gradually increasing the "fairness" assumptions, possibly these message deliveries will start to occur in every infinite fair computation trace. For instance, an additional fairness assumption could be that whenever two messages occur infinitely often together in the message buffer of a node then they are infinitely often delivered simultaneously to that node. Under this assumption, the example from the Introduction that is confluent but not consistent no longer holds. Yet, this assumption still allows for other examples that are confluent but not consistent.[16] One might try to increase the fairness assumptions to a point where confluence implies consistency, or prove that this will never be possible. Perhaps simple relational transducer networks could provide an initial setting.

A pragmatic lesson also seems possible. When we combine the previous work [6, 7] with the current, we might have a concrete indication that automatically deciding the correctness of a distributed system is not a useful strategy in practice. Indeed, it seems we have to severely restrict expressivity to obtain decidability, and yet the time complexity remains prohibitively high. What might be more promising though, is achieving correctness through practical mechanisms and protocols [22, 23, 10, 13], design guidelines [12], and insights about monotonicity [18, 4, 27, 5].

[16] Indeed, this time the program could require that a message B is delivered immediately after a message A is delivered, and this program only sends B when A is delivered (so A and B do not occur together in the message buffer).

Acknowledgments

The author thanks Jan Van den Bussche for his thoughtful comments on an earlier draft of this paper.

7. REFERENCES

[1] S. Abiteboul, M. Bienvenu, A. Galland, et al. A rule-based language for Web data management. In *Proceedings 30th ACM Symposium on Principles of Database Systems*, pages 293–304. ACM Press, 2011.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Abiteboul, V. Vianu, et al. Relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 61(2):236–269, 2000.

[4] P. Alvaro, N. Conway, J. Hellerstein, and W.R. Marczak. Consistency analysis in Bloom: A CALM and collected approach. In *Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260. www.cidrdb.org, 2011.

[5] T.J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *Journal of the ACM*, 60(2):15:1–15:38, 2013.

[6] T.J. Ameloot and J. Van den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. In *Proceedings of the 15th International Conference on Database Theory*, pages 86–98. ACM Press, 2012.

[7] T.J. Ameloot and J. Van den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. Hasselt University, Technical report (submitted to a journal), `http://hdl.handle.net/1942/14571`, 2013.

[8] K.R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.

[9] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, 2004.

[10] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), 2013.

[11] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd Symposium on Principles of Database Systems*, pages 273–284. ACM Press, 2013.

[12] M. Cavage. There's just no getting around it: You're building a distributed system. *ACM Queue*, 11(4), 2013.

[13] N. Conway, W.R. Marczak, P. Alvaro, J.M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1:1–1:14. ACM Press, 2012.

[14] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *Proceedings 12th International Conference on Database Theory*, 2009.

[15] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven Web applications. *Journal of Computer and System Sciences*, 73(3):442–474, 2007.

[16] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven Web services. In *Proceedings 25th ACM Symposium on Principles of Database Systems*, pages 90–99. ACM Press, 2006.

[17] N. Francez. *Fairness*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[18] J.M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[19] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of the 30th ACM symposium on Principles of Database Systems*, pages 223–234. ACM Press, 2011.

[20] L. Lamport. Fairness and hyperfairness. *Distributed Computing*, 13:239–245, November 2000.

[21] V. Nigam, L. Jia, B.T. Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. *Computer Languages, Systems & Structures*, 38(2):158–180, 2012.

[22] M. Shapiro, N.M. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, 2011.

[23] M. Shapiro, N.M. Preguiça, C. Baquero, and M. Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011.

[24] M. Spielmann. Verification of relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 66(1):40–65, 2003.

[25] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[26] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010.

[27] D. Zinn, T.J. Green, and B. Ludaescher. Win-move is coordination-free (sometimes). In *Proceedings of the 15th International Conference on Database Theory*, pages 99–113. ACM Press, 2012.