Explanations and Transparency in Collaborative Workflows

Serge Abiteboul, Pierre Bourhis, Victor Vianu

Explanations and Transparency in Collaborative Workflows

Serge Abiteboul
Inria-Paris & ENS Paris
fname.lname@inria.fr

Pierre Bourhis
CNRS & University Lille & Inria Lille
fname.lname@univ-lille.fr

Victor Vianu
UC San Diego & Inria-Paris
lname@cs.ucsd.edu

ABSTRACT

We pursue an investigation of data-driven collaborative workflows. In the model, peers can access and update local data, causing side-effects on other peers' data. In this paper, we study means of explaining to a peer her local view of a global run, both at runtime and statically. We consider the notion of "scenario for a given peer" that is a subrun observationally equivalent to the original run for that peer. Because such a scenario can sometimes differ significantly from what happens in the actual run, thus providing a misleading explanation, we introduce and study a faithfulness requirement that ensures closer adherence to the global run. We show that there is a unique minimal faithful scenario, that explains what is happening in the global run by extracting only the portion relevant to the peer.

With regard to static explanations, we consider the problem of synthesizing, for each peer, a "view program" whose runs generate exactly the peer's observations of the global runs. Assuming some conditions desirable in their own right, namely transparency and boundedness, we show that such a view program exists and can be synthesized. As an added benefit, the view program rules provide provenance information for the updates observed by the peer.

Keywords

data-centric workflows; collaboration; views; explanations

1. INTRODUCTION

Consider peers participating in a collaborative workflow. Such peers are typically willing to publicly share some data and actions, but keep others private or disclose them only to selected participants. During a run of the workflow, a peer observes side effects of other peers' actions, but may wish to be provided with a more informative explanation of the workflow. At runtime, one would like to explain to each peer the side effects she observes, in terms of the unfolding run. Statically, one would like to provide the peer with a program specifying all the transitions she may observe. In this paper, we consider the problem of providing peers with such runtime and static explanations. In particular, we identify two natural properties of workflows, transparency and boundedness, that are of interest in their own right and greatly facilitate the explanation task.

We use the data-driven collaborative workflow model of [6]. In contrast to process-centric workflows, data-driven workflows treat data as first-class citizens [26]. In our collaborative workflow model, each peer sees a view of a global database, that hides some relations, columns of other relations (projection), and tuples (selection). The workflow is specified by datalog-style rules, with positive and negative conditions in rule bodies, and insertions/deletions in rule heads. An *event* in the system is an instantiation by some peer of a rule in the workflow's program.

Consider a run of a workflow and a particular peer, say p. To be able to understand a run from p's perspective, it is useful to isolate the portion of the run relevant to p from other computations that may be occurring in the system. Towards this goal, we introduce the notion of *scenario for* p, which is a subrun that is observationally equivalent for p to the original run. Such a scenario includes events visible at p, but also events initiated by other peers, that have no immediate side-effects visible at p, but eventually enable events with visible side-effects. Among possible scenarios, minimal ones are desirable because they exclude redundant or useless (from the peer's viewpoint) events.

We show that computing minimal scenarios for a peer is generally hard (coNP-complete). Moreover, despite being observationally equivalent to the original run, scenarios can be misleading by differing considerably from what occurs in the actual run. To overcome these issues, we consider an additional property of scenarios, called *faithfulness*, that guarantees tighter consistency between the scenario and the actual run. Moreover, faithful scenarios turn out to be particularly well behaved. They form a semiring with respect to natural operators, which enables efficient computation of minimal faithful scenarios, as well as their incremental maintenance. We show that every run has a unique minimal faithful scenario for each peer, that can be computed efficiently. We use minimal faithful scenarios as the natural semantic and computational basis for explaining runs.

We then turn to the ambitious goal of providing *static* specifications of the runs as seen from a peer's perspective, which we call *view programs* for that peer. While such programs cannot generally exist for information-theoretic reasons, we consider some natural properties of workflows allowing to construct view programs that

define precisely a peer's observations of the workflow runs.

The properties we consider, *transparency* and *boundedness*, are often desirable in practice, for technical and even ethical reasons. In layman terms, an algorithm is transparent (for a specific purpose) if it discloses its motivation and actions. In workflows involving human participants (often the case for collaborative workflows), one might want to require transparency for particular users; indeed, one may be compelled by law to do so in certain settings. Intuitively, a workflow is *transparent* for a peer if the data that the peer sees at each point in a run is sufficient to determine all possible future transitions visible at that peer. For example, if a CEO vetoes the hiring of Alice, and as a consequence it becomes certain that she cannot be hired in the future, this information must be disclosed to her in the next transition she sees.

Boundedness is a more technical condition. For an integer h, *h-boundedness* of a run for a peer p limits to h the number of consecutive events invisible but relevant to p that the other peers can perform. (This does not prevent them from performing arbitrarily long sequences of events irrelevant to that particular peer.)

We show that one can decide whether, for a given h, a workflow program only produces transparent and h-bounded runs for a particular peer. Furthermore, when this is the case, one can construct a "view program" for a peer that specifies exactly the transitions that peer may see. The rules of the program also provide the peer with *provenance* information, consisting of the facts visible to that peer that have led to the transition. Because of boundedness, the provenance always involves a bounded number of tuples, which allows their static specification in the bodies of rules of the view program.

The synthesis of view-programs described in Section 5 is related in spirit with partner synthesis in services modeled as Petri Nets [30, 24, 28].

With practical considerations in mind, we lastly present some design guidelines for producing transparent and h-bounded programs for a specified peer. We also show that, for a large class of programs, one can force transparency and h-boundedness by rewriting each program so that, modulo minor differences, it has the same transparent and h-bounded runs as the original and filters out runs violating these properties.

The article is organized as follows. The model is described in Section 2. Scenarios are considered in Section 3, faithfulness in Section 4, transparency and view programs in Section 5, and design methodology for transparent and bounded programs in Section 6. Related work and conclusions are considered in two last sections. Some of the proofs are relegated to an appendix.

2. COLLABORATIVE WORKFLOW

In this section, we recall the collaborative workflow model of [6], introducing minor extensions.

We start with some basic terminology. We assume an infinite data domain *dom* with a distinguished element $\perp$ (intuitively denoting an undefined value), and including an infinite set $\mathcal{P}_\infty$ of *peers*. We also assume an infinite domain of variables *var* disjoint from *dom*. A *relation schema* is a relation symbol together with a sequence of distinct attributes. We denote the sequence of attributes of R by $att(R)$. A *database schema* is a finite set of relation schemas. A *tuple* over R is a mapping from $att(R)$ to *dom*. An *instance* of a database schema $\mathcal{D}$ is a mapping I associating to each $R \in \mathcal{D}$ a finite set of tuples over R, i.e., a *relation* over R. We denote by $Inst(\mathcal{D})$ the set of instances of $\mathcal{D}$.

We assume that each relation schema R is equipped with a unique key, consisting, for simplicity, of a single attribute K (the same for all relations). An instance $I \in Inst(\mathcal{D})$ is *valid* (for the key constraints) if, for each $R \in \mathcal{D}$, no tuple in $I(R)$ has value $\perp$ for attribute K, and there are no distinct tuples u, v in $I(R)$ with the same key. We denote by $Inst_K(\mathcal{D})$ the set of valid instances of $\mathcal{D}$.

We can apply to each instance $I \in Inst(\mathcal{D})$ the following chase step up to a fixpoint J denoted $chase_K(I)$:

> *for some R, some A, and distinct u, v in $I(R)$, with $u(K) = v(K)$, $u(A) \neq \perp$, and $v(A) = \perp$, replace v by v' identical to v except that $v'(A) = u(A)$.*

Note that the chase turns some invalid instances into valid ones, while for others it terminates with invalid instances. More precisely, an instance is turned into a valid one iff it contains no two tuples with the same key and distinct non-null values for the same attribute. In this case, the result of the chase is unique.

For technical reasons, we associate to each relation R (with attribute K), a unary relational view Key_R, that consists of the projection of R on K, i.e., for each R and instance I, $I(Key_R) = \pi_K(I(R))$.

We recall the notion of *full conjunctive query with negation* (FCQ$^\neg$ query for short). It is adapted to our context to take into account the view relations for keys. A *term* is a variable or a constant. A *literal* is of the form $(\neg)\ R(\bar{x})$, $(\neg)\ Key_R(y)$, $x = y$, or $x \neq y$, where $\bar{x}$ is a sequence of terms of appropriate arity, x is a variable, and y a term. A FCQ$^\neg$ query is an expression $A_1 \wedge ... \wedge A_n$ (for $n \geq 0$) where each A_i is a literal and such that each variable occurs in some positive literal $R(u)$ (a safety condition).

Observe that the use of a literal $Key_R(k)$ is syntactic sugar, since it can be replaced by $R(k, z_1, ...)$ where the z_i are new. On the other hand, the use of $\neg Key_R(k)$ is not.

For attributes A, B, and a in *dom* (possibly $\perp$) , $A = a$ and $A = B$ are *elementary conditions*. A *condition* is a Boolean combination of elementary conditions. Peer views of the database will be defined using projections and selections. Note that, in order to enable powerful static analysis, the views in [6] did not use selections. The more powerful views used here better capture realistic peer views.

Collaborative schema. We now define collaborative workflow schemas, extending the definition in [6]. Starting from a *global database schema* $\mathcal{D}$ and a finite set $\mathcal{P}$ of

peers participating in the workflow, the collaborative schema specifies, for each peer p, a view of the database. The view consists of selection-projection views of a subset of the relations in $\mathcal{D}$. A view of $R \in \mathcal{D}$ for a peer p (if provided) is denoted $R@p$. The view allows p to see only some of the attributes of R (projection on $att(R@p)$) and only some of the tuples (specified by a selection denoted $\gamma(R@p)$). The view of a global instance I of $\mathcal{D}$ at p is denoted $I@p$. Before providing the formal definition, we introduce the following notation. For a relation R and an instance J over a subset $att(J)$ of $att(R)$, $J^{\perp}$ denotes the instance of R obtained by padding all tuples of J with the value $\perp$ on all attributes in $att(R)-att(J)$.

DEFINITION 2.1. *For a global database schema $\mathcal{D}$, a* collaborative schema $\mathcal{S}$ *consists of a finite set $\mathcal{P}$ of peers, and for each $p \in \mathcal{P}$, a* view schema $\mathcal{D}@p$ *such that:*

- *each relation in $\mathcal{D}@p$ is of the form $R@p$ for R in $\mathcal{D}$,*
- *for each $R@p$, $K \in att(R@p) \subseteq att(R)$.*

To each relation $R@p \in \mathcal{D}@p$ is associated a selection condition $\gamma(R@p)$ over $att(R)$. For an instance I over $\mathcal{D}$, the view instance *of I at peer p, denoted $I@p$, is the instance over $\mathcal{D}@p$ defined by: for each $R@p$ in $\mathcal{D}@p$,*

- $I@p(R@p) = \pi_{att(R@p)}(\sigma_{\gamma(R@p)}(I(R)))$.

Furthermore, we impose the following (losslessness) condition: For each $I \in \mathrm{Inst}_K(\mathcal{D})$ and $R \in \mathcal{D}$,

$$I(R) = \mathrm{chase}_K\left(\bigcup\{(I@p(R@p))^{\perp} \mid p \in \mathcal{P}, R@p \in \mathcal{D}@p\}\right)$$

The losslessness property guarantees that for each tuple in $I(R)$, and each A in R, the value of the tuple for A is visible at some peer. The global instance can therefore be recovered from its peer views using the chase. One can effectively check whether a collaborative schema has the losslessness property.

Let $\mathcal{S}$ be a collaborative schema, with global schema $\mathcal{D}$ and set $\mathcal{P}$ of peers. A peer p can perform two kinds of *updates* on a valid instance I:

- A *deletion* of the form $-Key_{R@p}(k)$, where $R@p \in \mathcal{D}@p$ and k is a key value in $I@p(R@p)$. The deletion results in removing from $I(R)$ the tuple with key k.
- An *insertion* of the form $+R@p(u)$, where $R@p \in \mathcal{D}@p$ and u is a tuple over $att(R@p)$ such that:
  - (i) $J = chase_K(I \cup \{R(u^{\perp})\})$ is valid, and
  - (ii) u is subsumed by some tuple v in $J@p(R@p)$.

  Then J is the result of the insertion.

Note that the semantics of an update requested by some peer is specified on the global instance. This circumvents the view update problem. Observe also that a peer can delete only a tuple the peer sees, and that, if an insertion succeeds, the tuple the peer inserted is part of its view after the update (by (ii)).

Some subtleties of updates are illustrated next.

EXAMPLE 2.2. *Suppose the database consists of a single relation R over KAB, and we have two peers p, q, $att(R@p) = KAB$, $att(R@q) = KA$, $\gamma(R@p)$ is $A = \perp$, and $\gamma(R@q)$ is* true. *The losslessness condition is not satisfied by this schema. For example, consider the global instance I obtained using the sequence of inserts: $+R@p(k, \perp, c)$; $+R@q(k, a)$. It consists of a single tuple, $R(k, a, c)$. Note that as a result of the second insertion, the tuple with key k disappears from the view of p. Moreover, I cannot be reconstructed from the collective views of the peers (and the value "c" is lost). The losslessness condition prevents such anomalies. Moreover, it allows treating the global instance as a virtual rather than materialized database, represented in a distributed fashion by the views of the peers.*

Collaborative workflow. A *collaborative workflow specification* (in short *workflow spec*) $\mathcal{W}$ consists of a collaborative schema $\mathcal{S}$ and a *workflow program* for $\mathcal{W}$, i.e., a finite set of "update rules" for each peer p of $\mathcal{W}$.

The rules are defined using the auxiliary notion of "update atom" as follows. An *update atom* at p is an insertion atom or a deletion atom. An *insertion atom* at p is of the form $+R@p(\bar{x})$ where $R@p \in \mathcal{D}@p$ and $\bar{x}$ is a tuple of variables and constants, of appropriate arity. A *deletion atom* p is an expression $-Key_{R@p}(x)$, where $R@p \in \mathcal{D}@p$ and x is a variable or constant.

A *rule* at peer p is an expression *Update* :- *Cond* where:

- *Cond* is a $\mathrm{FCQ}^{\neg}$ query over $\mathcal{D}@p$, and
- *Update* is a sequence of update atoms at p such that: If the sequence includes two updates of the same relation R of tuples with keys x, x', respectively, then x and x' are not both the same constant and the body includes a condition $x \neq x'$.

In the previous definition, the conditions $x \neq x'$ impose that no two updates in the same rule affect the same tuple. As a consequence, the order of the update atoms in a rule is irrelevant. An example of rule is the following, where $Assign(x, y)$ says that employee x is assigned to project y, and HR is the Human Relations peer:

$$-Key_{Assign@HR}(x), \quad +Assign@HR(x', y) :- \\ Assign@HR(x, y), \quad Replace@HR(x, x'), \quad x \neq x'$$

The rule allows the HR peer to replace employee x by employee x' on project y.

If P is the program of a workflow spec $\mathcal{W}$ for a collaborative schema $\mathcal{S}$, we speak simply of the (workflow) program P when $\mathcal{S}$ and $\mathcal{W}$ are understood.

Let P be a program. To simplify, we assume that a run of P starts from the empty instance (note that an arbitrary "initial" instance can be constructed by the peers due to losslessness). The global instance then evolves under transitions caused by updates in rule instantiations, defined next. Let

$$\alpha = Update(\bar{y}) \text{ :- } Cond(\bar{x})$$

be a rule of P at some peer p where $\bar{x}$ are the variables occurring in *Cond* and $\bar{y}$ are the variables in *Update*. A

valuation ν of α for a global instance I is a mapping from $\bar{y} \cup \bar{x}$ to *dom* such that $I@p \models Cond(\nu(\bar{x}))$. For a valuation ν of a rule α at some p for some I, the instantiation $\nu\alpha$ is called an *event*; p the peer of this event, denoted $peer(\nu\alpha)$, and α its rule.

We define the transition relation $\vdash_e$ among valid global instances of $\mathcal{D}$ as follows. For I, J, and some event e as above, $I \vdash_e J$ if all insertions and deletions in $Update(\nu(\bar{y}))$ are applicable, and J is obtained from I by applying them in any order.

A *run* of P is a finite sequence $\rho = \{(e_i, I_i)\}_{0 \leq i \leq n}$, such that $\emptyset \vdash_{e_0} I_0$, where $\emptyset$ is the empty instance; and for each $0 < i \leq n$, $I_{i-1} \vdash_{e_i} I_i$. Additionally, we require: for each event $e_i = \nu\alpha$, if x is a variable occurring in the head of α but not in its body, then x must be instantiated to a globally fresh value, i.e. $\nu(x)$ does not occur in $const(P)$ or in $I_1 \ldots I_{i-1}$. We denote by $Runs(P)$ the set of runs of P.

Let $\rho = \{(e_i, I_i)\}_{0 \leq i \leq n}$ be a run of P. Note that the event sequence $(e_1 ... e_n)$, denoted $e(\rho)$, uniquely determines the run ρ. By slight abuse, we sometimes call *run* a sequence $(e_1 ... e_n)$ of events that yields a run.

It is occasionally useful to consider runs starting from an arbitrary initial instance I rather than $\emptyset$. A run on initial instance I is defined in the obvious manner, by replacing $\emptyset$ with I in the previous definition.

Normal-form programs. We show a normal form for workflow programs that will be particularly useful in the next sections. A workflow program P is in *normal form* if (i) each rule whose head contains a deletion $-Key_{R@q}(x)$ also contains a literal $R@q(x, u)$ in its body, and (ii) rule bodies do not contain negative literals of the form $\neg R@q(x, u)$ or positive literals of the form $Key_{R@q}(x)$. Intuitively, (i) simply makes explicit that deletion updates are effective. (Recall that this is imposed by the definition of deletion.) As for (ii), it allows distinguishing between the cases when a fact $R@q(k, u)$ is false because no tuple in $R@q$ has key k, or because $R@q(k, v)$ is a fact for some $v \neq u$. Thus, rules in normal form provide more refined information. We next show that this does not limit the expressivity of the model.

PROPOSITION 2.3. *For each workflow program P, one can construct a normal-form program P^{nf}, and a function θ from the rules of P^{nf} to rules of P, such that $\rho = \{(e_i, I_i)\}_{0 \leq i \leq n}$ is a run of P iff $\rho^{nf} = \{(f_i, I_i)\}_{0 \leq i \leq n}$ is a run of P^{nf} for some $\{f_i\}_{0 \leq i \leq n}$ such that $peer(e_i) = peer(f_i)$ and $rule(e_i) = \theta(rule(f_i))$.*

PROOF. Informally, P^{nf} is constructed as follows. For each rule containing a deletion $-Key_{R@q}(x)$ in its head, a literal $R@q(x, u)$ is added to the body, where u consists of distinct new variables. This guarantees (i). For (ii), a literal $Key_{R@q}(x)$ occurring in the body of some rule of P can be replaced by a literal $R@q(x, u)$, where u consists of distinct new variables. Now suppose a literal $\neg R@q(x, u)$ occurs in the body of some rule r of P. First, an instantiation ν of this rule may hold because $\neg Key_{R@q}(\nu(x))$ holds. Then this can be captured by a rule obtained from r by replacing $\neg R@q(x, u)$ by $\neg Key_{R@q}(x)$. Next, an instantiation ν of r may hold because $R@q(\nu(x), v)$ holds for some v and $\nu(u(A)) \neq v(A)$ for some attribute $A \neq K$ of $R@q$. Then this can be captured by a rule obtained from r by replacing $\neg R@q(x, u)$ by $R@q(x, z)$ where z is a tuple of distinct new variables, and adding the condition $u(A) \neq z(A)$. Note that the previous construction replaces r with a set $Rules(r)$ of rules in P^{nf}, corresponding to the different cases. The mapping θ is defined by $\theta(r') = r$ for each $r' \in Rules(r)$. Rules r of P that are not modified are included as such in P^{nf}, and $\theta(r) = r$. □

3. VIEWS AND SCENARIOS

In this section, we isolate the portions of a run that are relevant to a particular peer and accurately mirror what is actually occurring in the workflow from that peers viewpoint. Towards this goal of filtering out irrelevant events, we introduce the notions of subruns and scenarios. In the next section, we further consider scenarios called *faithful* that adhere more closely to the actual run and also turn out to have desirable properties from a semantic and computational viewpoint.

We first define the p-view of a run, for a peer p. Intuitively, this is the most basic view of a run, consisting essentially of the observations of peer p. Let $\rho = \{(e_i, I_i)\}_{0 \leq i \leq n}$ be a run of P. An event e_i is *visible at* p if $peer(e_i) = p$ or $peer(e_i) \neq p$ and $I_{i-1}@p \neq I_i@p$. Otherwise, e_i is invisible (or silent) at p. The p-view of an event e, denoted $e@p$, is e if $peer(e) = p$ and ω if $peer(e) \neq p$, where ω is a new symbol (standing for "world").

DEFINITION 3.1. *Let $\rho = \{(e_i, I_i)\}_{0 \leq i \leq n}$ be a run of P, p a peer, and $\rho' = \{(e_i@p, I_i@p)\}_{0 \leq i \leq n}$. The sequence obtained from ρ' by deleting all $(e_i@p, I_i@p)$ such that e_i is invisible at p, is called the* view *of ρ at p, and denoted $\rho@p$. We denote* $\text{Runs}@p(P) = \{\rho@p \mid \rho \in \text{Runs}(P)\}$.

Thus, $\rho@p$ consists of all transitions caused by p marked with $e_i@p$, as well as all transitions of events visible at p caused by other peers, marked with ω.

We next consider subruns of a given run, with the goal of isolating the portion of the run that is relevant to p, and which can be used as a sound basis for providing the peer with additional information. For example, in Section 5 we discuss richer views that include provenance information for the updates observed by a peer, extracted from subruns relevant to the peer.

A *subrun* $\widehat{\rho}$ of ρ is a run such that $e(\widehat{\rho})$ is a subsequence of $e(\rho)$. In a subrun, only some of the events of ρ are retained, in the same order as they occur in ρ. Observe that the instances in $\widehat{\rho}$ are typically different than those in ρ. Of course, not all subsequences of $e(\rho)$ yield subruns. If a subsequence α of $e(\rho)$ does yield a subrun, it is denoted $run(\alpha)$ (or α by slight abuse).

We are interested in those subruns of a run that are compatible with p's observations of the run. This is captured by the notion of "scenario". We will, in particular, be interested by "minimal" such scenarios that,

in some sense, explain what can be observed by p in a non-redundant manner.

DEFINITION 3.2. *Let P be a workflow program, p a peer of P, and $\rho = \{(e_i, I_i)\}_{0 \le i \le n}$ a run of P. A* scenario *of ρ at p is a subrun $\widehat{\rho}$ of ρ such that $\rho@p = \widehat{\rho}@p$.*

A scenario $\widehat{\rho}$ of ρ at p is minimal *if there is no scenario $\bar{\rho}$ of ρ at p for which $e(\bar{\rho})$ is a strict subsequence of $e(\widehat{\rho})$. A minimal scenario is* minimum *if there is no shorter scenario of ρ at p.*

Clearly, ρ itself is a scenario for ρ at p, but is likely to include portions that are irrelevant to p's observations, which motivates the study of minimal and minimum scenarios.

We next show that finding a minimum scenario is hard. (We will see that the problem is hard also for minimal scenarios.) In the proof, and throughout the paper whenever convenient, we use propositions in workflow programs as syntactic sugar. A proposition x can be simulated by a unary relation R^x (with key K), a literal $(\neg)x$ by $(\neg)R^x(0)$, an insertion or deletion of this literal by $+R^x(0)$ or $-Key_{R^x}(0)$.

THEOREM 3.3. *It is* NP*-complete, given a workflow program P, a run ρ of P, a peer p and an integer N, whether there exists a scenario of ρ at p, of length at most N. Moreover, this holds even for workflows with ground positive rules and no deletions.*

PROOF. Membership in NP is obvious. For hardness, we use a reduction from the hitting set problem. An instance of Hitting Set consists of a finite set $V = \{v_1, \ldots, v_n\}$, a set $\{c_1, \ldots, c_k\}$ of subsets of V, and an integer $M < n$. It is NP-complete whether there exists $W \subset V$ of size at most M such that $W \cap c_j \neq \emptyset$ for $1 \le j \le k$ [17]. From an instance of Hitting Set, we construct a program and a run ρ as follows. The schema consists of propositions V_i $(1 \le i \le n)$, C_j $(1 \le j \le k)$, and OK. There are two peers p, q. Peer q sees all propositions and p only OK. The rules are the following:

(a) $+V_i@q$:- , for each $i \in [1..n]$,
(b) $+C_j@q$:- $V_i@q$, for each $i \in [1..n]$, $j \in [1..k]$, $v_i \in c_j$,
(c) $+OK@q$:- $C_1@q, \ldots, C_k@q$

Observe that the workflow program uses only propositions.

Note that all rules belong to peer q but p can observe the value of OK. Intuitively, firing a subset of the (a)-rules designates a subset W of V. With W chosen, the (b)-rules designate the sets c_j hit by some element in W. If all sets are hit, rule (c) is enabled. Consider firing first all (a)-rules, followed by all (b) rules, and ending with rule (c). It is clear that this yields a run; call it ρ. Intuitively, ρ corresponds to picking the trivial hitting set $W = V$. It is easy to check that there exists a hitting set W of size at most M iff there exists a scenario of ρ for p, of length at most $M + k + 1$. □

For the specific runs ρ used in the proof, *some* minimal scenario can always be found efficiently in a greedy manner. (Start with ρ. First, remove one (a)-rule at a time together with the (b) rules depending on it, and check if the remaining sequence is still a scenario. Then when no (a) rule can be further removed, keep only one (b) rule for each j. The resulting scenario is minimal.) However, this cannot be done for arbitrary runs. Indeed, the next result shows that testing whether a scenario is minimal is hard (see Appendix for proof).

THEOREM 3.4. *It is* coNP*-complete, given a workflow program P, a run ρ of P, and a peer p, whether ρ is a minimal scenario of ρ at p. Moreover, this holds even for workflows with ground positive rules and no deletions.*

The lack of a unique minimal scenario of runs for a given peer is problematic when richer views need to be defined starting from several candidate minimal scenarios. Moreover, as seen in the next section, even minimal scenarios can provide misleading explanations about what occurs in the global run. We will propose a natural restriction called *faithfulness*, that overcomes the problems of unrestricted scenarios.

4. FAITHFUL SCENARIOS

A scenario $\widehat{\rho}$ of a run ρ at peer p produces the same observations as ρ at p, but is allowed to achieve this by means that differ considerably from what occurred in the original run. This can be misleading, as illustrated further in Examples 4.1 and 4.2. We therefore consider an additional property of subruns, called *faithfulness*, that guarantees tighter consistency between what happens in the subrun and in the actual run. The idea, that we will pursue in the next section, is that the workflow system wants to be more transparent for particular individual peers. Furthermore, as we shall see, faithful scenarios turn out to be particularly well behaved: each run has a unique minimal faithful scenario for p, computable in polynomial time, that explains what happens at p in a concise, non-redundant way. We also demonstrate useful properties of faithful scenarios, in particular that they are closed under intersection and union.

Before defining faithful scenarios, we illustrate some of the discrepancies that may arise between arbitrary scenarios and actual runs.

EXAMPLE 4.1. *Consider again the worlflow used in the proof of Theorem 3.3. Suppose that p also sees the propositions $C_i, i \in [1, k]$. Consider a run that derives OK. Suppose the run starts with*

$V_1@q$:- ; $C_5@q$:- $V_1@q$; $V_2@q$:- ; $C_5@q$:- $V_2@q$.

Then a scenario could ignore $C_5@q$:- $V_1@q$ although this is the event that actually derived $C_5@q$.

EXAMPLE 4.2. *Consider a workflow with peers* cto, ceo, assistant*, and* applicant *and propositions* ok *and* approval*. The peers* cto, ceo*, and* assistant *all see* ok *and* approval*, and* applicant *sees only* approval*. Consider the run ρ consisting of the following events:*

e:	+ok@cto	:-	
f:	−ok@cto	:-	
g:	+ok@ceo	:-	
h:	+approval@assistant	:-	ok@assistant

The subrun e h is a scenario of ρ at peer applicant*. It indicates that the applicant's request was approved because it was ok'd by the cto. This subrun is misleading, since in the actual workflow, the cto retracted its* ok *and the request was approved by the ceo. This arises because the subrun ignores the deletion of* ok@cto*.*

Let p be a fixed peer of some workflow program P and ρ a run. We next introduce the notion of "p-faithful subrun of ρ", that prevents the kinds of anomalies previously illustrated. First, the definition is driven by the intuition that tuples in a given relation with a fixed key k represent evolving objects in the workflow. Objects identified by k can go through several *lifecycles*, occurring between the creation and deletion of a tuple with key k. Faithfulness requires that boundaries of lifecycles of events in the subrun be the same as those in the run, eliminating anomalies such as Example 4.2. It also requires that the events affecting relevant attributes of object k in the same lifecycle be the same in the subrun as in the actual run, eliminating alternative subruns as in Example 4.1.

To define faithful subruns formally, we use the following auxiliary definitions. We assume wlog that all programs are in normal form. Let P be a program and $\rho = \{(e_i, I_i)\}_{0\leq i\leq n}$ a run of P. We say that k occurs as a key of R in an event e_i of peer q, if it occurs in a literal $R@q(k, \bar{u})$ or $\neg Key_{R@q}(k)$ in the body of e_i, or as an update $+R@q(k, \bar{u})$ or $-Key_{R@q}(k)$ in the head of e_i. We say that k occurs as a key of R in a sequence α of events if it occurs as a key of R in some event of α. We denote the set of such keys by $\mathcal{K}(R, \alpha)$.

Let $k \in \mathcal{K}(R, e(\rho))$ where $\rho = \{(e_i, I_i)\}_{0\leq i\leq n}$. A *closed R-lifecycle* of k in ρ is an interval $[i_1, i_2] \subseteq [0, n]$ such that e_{i_1} inserts in R a new tuple with key k, this tuple is not deleted between i_1 and i_2, and e_{i_2} deletes it. An *open R-lifecycle* of k in ρ is an interval $[i_1, \infty)$ such that e_{i_1} inserts in R a new tuple with key k and this tuple is not deleted later on in ρ. In both cases, we say that e_{i_1} is the left boundary event of the lifecycle, and we say that e_{i_2} is the right boundary event of the closed lifecycle $[i_1, i_2]$.

We now define the ingredient of faithfulness concerning
boundaries of lifecycles. For a subrun $\widehat{\rho}$ of ρ, the requirement applies to $e(\widehat{\rho})$ alone. In fact, it will be useful to define this notion for arbitrary subsequences of $e(\rho)$.

Definition 4.3. *Let $\rho = \{(e_i, I_i)\}_{0\leq i\leq n}$ be a run of some workflow program P. A subsequence $\alpha = \{e_{i_j}\}_{0\leq j\leq m}$ of $e(\rho)$ is* boundary faithful *if for every e_{i_j} and $k \in \mathcal{K}(R, e_{i_j})$ such that i_j belongs to an R-lifecycle[1] of k in ρ, the left boundary event of the R-lifecycle belongs to α, and the right boundary event of the R-lifecycle also belongs to α, if the R-lifecycle is closed.*

[1]Observe that $k \in \mathcal{K}(R, e_{i_j})$ need not belong to an R-lifecycle containing i_j, because k may occur in a negative literal $\neg Key_{R@q}(k)$.

Observe that boundaries of R-lifecycles of k in ρ that do not contain events in α need not be included in α.

We now consider the last ingredient of faithfulness. Its definition relies on the auxiliary concept of the set of attributes of R that are "relevant" to peer q, that is denoted $att(R, q)$. Specifically, the values on $att(R, q)$ determine whether a given tuple is seen by q, and provides the visible values. Formally, for a peer q with selection condition $\gamma(R@q)$, we define $att(R, q) = att(R@q) \cup att(\gamma(R@q))$, where $att(\gamma(R@q))$ is the subset of attributes of R used in $\gamma(R@q)$.

The last ingredient of faithfulness focuses on events that modify existing tuples with a given key. Intuitively, modification faithfulness for a peer p requires that, within a lifecycle of key k, all updates of attributes relevant to p must be included in the subsequence. It also requires that updates of attributes relevant to other peers participating in the subsequence be included as well.

Definition 4.4. *Let $\rho = \{(e_i, I_i)\}_{0\leq i\leq n}$ be a run of P and p a fixed peer. A subsequence $\alpha = \{e_{i_j}\}_{0\leq j\leq m}$ of $e(\rho)$ is* modification faithful for p *if the following holds for each e_{i_j}, for each R and each $k \in \mathcal{K}(R, e_{i_j})$: if e_i belongs to the same R-lifecycle of k in ρ as e_{i_j}, $i < i_j$, $\text{peer}(e_{i_j}) = q$, and e_i contains an insertion that turns in ρ some attribute in $\text{att}(R, q) \cup \text{att}(R, p)$ of an existing tuple with key k from $\perp$ to some other value, then e_i also belongs to α.*

Observe that boundary faithfulness is independent of the fixed peer p but modification faithfulness is dependent on p. We can now define the notion of faithful subsequence and subrun of a run.

Definition 4.5. *A subsequence α of $e(\rho)$ is* p-faithful *if it contains all events of ρ that are visible at p, is boundary faithful, and modification faithful for p. A subrun $\widehat{\rho}$ of ρ is p-faithful if $e(\widehat{\rho})$ is p-faithful.*

Observe that faithfulness rules out the counterintuitive scenarios in Examples 4.1 and 4.2. For instance, gh is an *applicant*-faithful subsequence of the run in Example 4.2, whereas eh is not. Moreover, note that gh is a subrun. This is not coincidental. We next show the following key fact: p-faithful subsequences of $e(\rho)$ always yield scenarios of ρ for p. The proof is provided in the appendix.

Lemma 4.6. *Let ρ be a run of $\mathcal{W}$, p a peer, and α a p-faithful subsequence of $e(\rho)$. Then (i) α yields a subrun of ρ, and (ii)* $\text{run}(\alpha)$ *is a scenario of ρ for p.*

We next show the existence of a unique minimal p-faithful scenario of ρ for p, that can be computed in PTIME. To this end, we define an operator $T_p(\rho, \cdot)$ on subsequences of $e(\rho)$. For each subsequence α of $e(\rho)$, $T_p(\rho, \alpha)$ consists of the subsequence of $e(\rho)$ obtained by adding to α all events of ρ whose presence is required by

boundary and modification p-faithfulness due to events *that are already* in α.

Observe that (almost by definition) a subsequence α of ρ is boundary and modification p-faithful iff it is a fixed-point of $T_p(\rho, \cdot)$, i.e., $T_p(\rho, \alpha) = \alpha$. Let $\ll$ be the subsequence relation on sequences of events. Note that $T_p(\rho, \cdot)$ is monotone with respect to $\ll$, i.e., for $\alpha \ll \beta$, $T_p(\rho, \alpha) \ll T_p(\rho, \beta)$.

Consider the increasing sequence $\alpha_0 = \alpha$, $\alpha_1 = T_p(\rho, \alpha_0)$, $\alpha_2 = T_p(\rho, \alpha_1)$, ... and let $T_p^\omega(\rho, \alpha) = \alpha_n$ where n is the minimum integer for which $\alpha_n = \alpha_{n+1}$. Now we have:

THEOREM 4.7. *Let P be a program schema. For each run ρ of P there is a unique minimal p-faithful scenario $\hat{\rho}$ of ρ. Moreover, $\hat{\rho}$ equals* $\mathrm{run}(T_p^\omega(\rho, \alpha))$*, where α consists of all events of ρ visible at p, and can be computed from ρ in polynomial time.*

PROOF. Clearly, $T_p^\omega(\rho, \alpha)$ is p-faithful, because it is a fixed-point of $T_p(\rho, \cdot)$ and contains all events of ρ visible at p. By Lemma 4.6, it also yields a scenario for p. Consider any p-faithful scenario $\bar{\rho}$ of ρ for p. By definition of p-faithfulness, $\bar{\rho}$ must include the events in α. From the fact that $\alpha \ll e(\bar{\rho})$, $T_p(\rho, \cdot)$ is monotone, and $e(\bar{\rho})$ is a fixed-point of $T_p(\rho, \cdot)$, it follows that $T_p^\omega(\rho, \alpha) \ll e(\bar{\rho})$. Thus, $\hat{\rho} = T_p^\omega(\rho, \alpha)$ is the unique minimal p-faithful scenario of ρ. Clearly, $T_p^\omega(\rho, \alpha)$ can be computed in polynomial time from ρ. □

We now consider two natural ways of combining subsequences of a run ρ, that are useful in many practical situations: "multiplying" them by taking the intersection of their events, and "adding" them by taking the union of their events. We will show that p-faithful subsequences of a run ρ are closed under both operations, and form a semiring.

Formally, let α_1 and α_2 be subsequences of $e(\rho)$ for a run ρ. Their *product*, denoted $\alpha_1 * \alpha_2$ is the subsequence consisting of the events in ρ occurring in both α_1 and α_2. Their *addition*, denoted $\alpha_1 + \alpha_2$, is the subsequence consisting of the events of ρ in α_1 or α_2. Clearly, if ρ_1 and ρ_2 are subruns of ρ, $e(\rho_1) + e(\rho_2)$ and $e(\rho_1) * e(\rho_2)$ are not guaranteed to yield subruns of ρ.

Addition and multiplication of sequences are both of interest in practice. Closure under intersection is the core reason for the existence of a unique minimal p-faithful scenario for a peer. As we will see, addition is useful in incremental maintenance of minimal p-faithful scenarios.

We next show the following (see Appendix for proof).

THEOREM 4.8. *Let ρ be a run of $\mathcal{W}$. The set of p-faithful scenarios of ρ is closed under addition and multiplication, and forms a semiring.*

Incremental evaluation To conclude the section, we discuss how to maintain incrementally the minimum p-faithful scenario of a run, leveraging the closure under addition of p-faithful scenarios. More specifically, for a run ρ, we wish to maintain incrementally $T_p^\omega(\rho, \alpha)$, where α consists of the events of ρ visible at p. To do so, we additionally maintain some auxiliary information, consisting of $T_p^\omega(\rho, f)$ for each event f in ρ. Intuitively, $T_p^\omega(\rho, f)$ represents an "explanation" of the individual event f by a minimal boundary and modification p-faithful subrun of ρ containing f. Note that f need not be visible at p.

Suppose the current run is ρ and we have computed $T_p^\omega(\rho, \alpha)$ and $T_p^\omega(\rho, f)$ for each event f in ρ. Now suppose that a new event e arrives. We need to compute the following (where $\rho.e$ denotes the concatenation of ρ and e):

(i) $T_p^\omega(\rho.e, f)$, for each event f in $\rho.e$.
(ii) $T_p^\omega(\rho.e, \alpha')$, where $\alpha' = \alpha.e$ if e is visible at p and $\alpha' = \alpha$ otherwise.

Consider (i). For $f = e$, $T_p^\omega(\rho.e, e)$ consists of e together with all events in $T_p^\omega(\rho, g)$ for some $g \in T_p(\rho.e, e)$. For $f \neq e$, there are two cases, depending on whether e is the right-boundary of an open lifecyle of a key occurring in $T_p^\omega(\rho, f)$. If this is the case, meaning that e is in $T_p(\rho.e, T_p^\omega(\rho, f))$, then $T_p^\omega(\rho.e, f)$ consists of e together with the events in $T_p^\omega(\rho, f)$ and those in $T_p^\omega(\rho, g)$ for some $g \neq e$ in $T_p(\rho.e, e)$. Otherwise, $T_p^\omega(\rho.e, f) = T_p^\omega(\rho, f)$.

Now consider (ii). Suppose e is visible at p, or is the right-boundary of an open lifecycle of a key in $T_p^\omega(\rho, \alpha)$, meaning that e is in $T_p(\rho.e, T_p^\omega(\rho, \alpha))$. Then $T_p^\omega(\rho.e, \alpha')$ consists of e together with the events in $T_p^\omega(\rho, \alpha)$ and in $T_p^\omega(\rho, g)$ for some $g \neq e$ in $T_p(\rho.e, e)$. Otherwise, $T_p^\omega(\rho.e, \alpha') = T_p^\omega(\rho, \alpha)$.

Observe that the incremental maintenance algorithm outlined above requires only single applications of the operator $T_p(\rho, \cdot)$, avoiding computations of its least fixpoint from scratch. This is similar in spirit to results on incremental maintenance of recursive views by non-recursive queries (e.g., see [16]).

5. TRANSPARENCY & VIEW-PROGRAMS

The goal. Let P be a workflow program, and p a peer. We have defined, for each run ρ of P, the view $\rho@p$ of the run as seen by peer p. As a next step, we would like to also provide p with a "view-program" P' whose runs are precisely the views $\mathit{Runs}(P)@p$. Intuitively, we wish such a program to provide p with an "explanation" of the global workflow in terms it can understand and access. The view-program will use the same schema $\mathcal{D}@p$ as p, and the fictitious peer ω (world) that represents the environment. The rules of P' consist of the rules of p in P, together with new rules for peer ω, that define the transitions caused by other peers with visible side-effects at p.

More precisely, a program P' is a *view-program* for P at p if:

- P' is over global schema $\mathcal{D}@p$ and uses two peers p and ω, both with schema $\mathcal{D}@p$, and all selection conditions *true*.
- the rules of peer p are the same in P and P'.

- (completeness) for each run ρ of P, there exists a run ρ' of P' such that $\rho@p$ is obtained from ρ' by replacing all ω-events with ω.
- (soundness) for each run ρ' of P', there exists a run ρ of P, such that $\rho@p$ is obtained from ρ' by replacing all ω-events with ω.

We illustrate the notion of view-program with an example.

EXAMPLE 5.1. *Consider a program P with peers* hr, ceo, cfo, *and* Sue, *using the following rules:*

```
+Cleared@hr(x)      :-
+cfoOK@cfo(x)       :-
+Approved@ceo(x)    :-  Cleared@ceo(x),
                        cfoOK@ceo(x)
+Hire@hr(x)         :-  Approved@hr(x)
```

Note that there are no rules for Sue. *Suppose* hr, cfo, *and* ceo *see all relations, but* Sue *sees only relations* Cleared *and* Hire. *A view-program P' for* Sue *operates on the schema* Cleared *and* Hire *and has the following rules:*

```
      +Cleared@ω(x)   :-
(†)   +Hire@ω(x)      :-  Cleared@ω(x)
```

It is easy to check that P' is sound and complete for P and peer Sue, *and so it is a view-program for P at* Sue.

REMARK 5.2. *Observe that soundness and completeness of view-programs amounts to equivalence of P' and P with respect to the views of their linear runs. However, consider the following subtlety in the above example. By soundness of P' and rule (†), if Sue sees* Cleared(x), *then there* exists *a run of P in which Sue sees* Hire(x) *inserted in the next transition visible to her. However, it is not the case that this is possible in* every *run of P, because the transition is also dependent on relation* cfoOK, *invisible to Sue. Enforcing this stronger property requires a more stringent notion of equivalence based on the* trees of runs *rather than just linear runs. We later show how this can be done by forbidding the use of information invisible to the given peer that affects the view of the peer. Intuitively, this makes the collaborative workflow more transparent to the peer. We formally introduce the notion of transparency further.*

Note that one can trivially define a sound program for P at p (by keeping only the rules of P at peer p) and a complete one (by adding to the rules of p all rules at ω that insert or delete up to M arbitrary tuples in relations of $\mathcal{D}@p$, where M is the maximum number of updates in the head of a rule in P). Clearly, these are of little interest. Ideally, one would like a program that is both sound and complete. Unfortunately, as shown next, it is not generally possible to construct such a program.

PROPOSITION 5.3. *There exists a program P and a peer p, such that there exists no view-program for P at p.*

PROOF. (sketch) The program P uses three binary relations R, S, T that a peer q sees, whereas peer p sees only R, T. Suppose p has a rule inserting an arbitrary tuple in R (so p can construct arbitrary instances over R). Peer q has two rules to add to S pairs in the transitive closure of R. Finally, q has a rule to transfer a tuple $(0, 1)$ from S to T. No view-program can exist for P at p because the insertion of the tuple $(0, 1)$ in $T@p$ is conditioned by the existence of a directed path of arbitrary length from 0 to 1 in $R@p$. This cannot be expressed by any rule with a bounded number of literals in its body. □

THEOREM 5.4. *It is undecidable, given a program P and a peer p, whether there exists a view-program for P at p.*

PROOF. (sketch) We use the following observation:

($\star$) It is undecidable, given a program Q whose schema includes a unary relation U, whether there exists a run of Q leading to an instance where U is non-empty.

The proof of ($\star$) is by a straightforward reduction from the Post Correspondence Problem, known to be undecidable [27]. Intuitively, Q attempts to construct a solution to an instance of the PCP. If it succeeds, a fact is inserted in U (details omitted).

Consider the program P in the proof of Proposition 5.3, modified so that the rule to transfer the tuple $S(0, 1)$ to $T(0, 1)$ is controlled by non-emptiness of U. Now add to P the rules of an arbitrary program Q whose schema contains U (and no other relations of P). Then a view-program at p exists for the resulting program iff there is no run of Q leading to some non-empty U. This is undecidable by ($\star$). □

Transparency and Boundedness. Clearly, an obstacle towards obtaining a view-program for P at p is that updates visible at p may depend on information unavailable to p. To overcome this difficulty, we consider a property of programs called "transparency". Intuitively, transparency of a program P for peer p guarantees that the possible updates of a view instance $I@p$ caused by other peers are determined by $I@p$. Put differently, the other peers must disclose to p all information that they use in order to modify p's view of the data. The only action that can depend on hidden information is the creation of new values, which is constrained by the global history.

It turns out that transparency of a program P for p does not alone guarantee the existence of a view-program of P for p. This is because the other peers can still perform arbitrarily complex computations hidden from p. For instance, the program used in the proof of Proposition 5.3 is transparent for p but does not have a view-program for p. To control the complexity of computations affecting p, we introduce a notion of "boundedness" of P with respect to p, that limits the number of steps invisible but relevant to p that are carried out by other peers. As we will see, transparency together with

boundedness guarantee the existence of a view-program and its effective construction.

We next define transparency, then turn to boundedness. To formalize the notion of transparency, we first define "fresh" instances, obtained as the results of events visible at p. We use the following notation. If α is a sequence of events of P yielding a run on initial instance I, we say that α is applicable at I and denote by $\alpha(I)$ the last instance in the run.

DEFINITION 5.5. *Let P be a program and p a peer. An instance I is* p-fresh *if $I = \emptyset$ or there exists an instance I' and an event e of P that is applicable to I' and visible at p, such that $e(I') = I$.*

We can now define transparency. We will use the notion of minimum p-faithful run, defined as follows. A run α on initial instance I is a minimum p-faithful run if $\alpha = T_p^\omega(\alpha, \bar{v})$, where $\bar{v}$ consists of the events of α that are visible at p. In other words, α is its own minimum p-faithful scenario for p.

To deal with new values, we will need the following. For a sequence α, let $adom(\alpha)$ consist of all values occurring in α, and $new(\alpha)$ consist of all values a for which there is an event e in α such that a occurs in the head but not in the body of e. Let $const(P)$ denote the set of constants used in program P, together with $\perp$.

DEFINITION 5.6. *A program P is* transparent *for p if for all p-fresh instances I, J such that $I@p = J@p$ the following holds. For every sequence α of events such that* $\text{adom}(J) \cap \text{new}(\alpha) = \emptyset$*, if α is a minimum p-faithful run on I such that all its events but the last are silent at p, then the same holds on J, and $\alpha(I)@p = \alpha(J)@p$.*

Intuitively, transparency implies that the computation as seen from p depends only on what p sees, except for the specific choice of new values. The definition is illustrated by the following example.

EXAMPLE 5.7. *Consider again Example 5.1. It is easy to see that the program in the example is* not *transparent for Sue. Intuitively, this is because the relation* cfoOK *carries information that Sue does not see, yet it impacts Sue's view of the workflow. Now consider the following program (obtained by eliminating* cfoOK*):*

```
+Cleared@hr(x)      :-
+Approved@ceo(x)    :-  Cleared@ceo(x)
+Hire@hr(x)         :-  Approved@hr(x)
```

At first glance, the program may now appear to be transparent for Sue. However, this is not the case. Indeed, consider the instances I, J containing the following facts:

I: Cleared(Sue); Approved(Sue)
J: Cleared(Sue)

Clearly, I and J are Sue-fresh since both can be obtained by an application of the Sue-visible event +Cleared@hr(Sue). In addition, $I@Sue = J@Sue = \{$Cleared@Sue(Sue)$\}$. The sequence consisting of the single event

+Hire@hr(Sue) :- Approved@hr(Sue)

is a minimum Sue-faithful run on instance I. Transparency requires it to also be applicable on J, which is not the case. Intuitively, in order for transparency to hold, Sue-freshness must ensure that pre-existing information invisible to Sue, such as the fact Approved(Sue) *in instance I, cannot be used in later events leading to transitions visible by Sue. This can be achieved in various ways. We illustrate one approach, that is also adopted in the design methodology for transparent programs in Section 6. We introduce an additional binary relation* Stage *visible by all peers, that inhibits the use of any information computed prior to the latest Sue-visible update. Relation* Stage *is either empty or contains a single tuple* Stage$(0, s)$ *where s is a value refreshed by peer Sue prior to events by other peers. The invisible relation* Approved *is extended with an extra column holding the current value of s. The program is the following:*

```
+Stage@Sue(0,s) :- ¬ Key_Stage@Sue(0)
+Cleared@hr(x), −Key_Stage@Sue(0) :- Stage@hr(0,s)
+Approved@ceo(x,s) :- Cleared@ceo(x), Stage@ceo(0,s)
+Hire@hr(x), −Key_Stage@hr(0) :- Approved@hr(x,s), Stage@hr(0,s)
```

Observe that Stage(0,s) *is deleted by each event visible at Sue, which forces its re-initialization with a fresh s before any event using invisible relations can proceed, preventing the use of previous invisible facts. It is easy to check that the program is now transparent for Sue.*

We next introduce the boundedness property.

DEFINITION 5.8. *Let P be a program, p a peer, and h an integer. P is* h-bounded *for p if for each instance I and sequence α of events applicable to I that yields a minimum p-faithful run at p such that all its events but the last are invisible at p, $|\alpha| \leq h$.*

Intuitively, this bounds the number of consecutive events that are silent but relevant to p. Note that the bound applies only to minimum p-faithful subruns. Thus, the other peers are still allowed to carry out arbitrarily long silent computations that do not affect p.

We next consider the decidability of transparency and boundedness. It is easy to show:

THEOREM 5.9. *It is undecidable, given a program P and peer p, (i) whether P is transparent for p, and (ii) whether there exists h such that P is h-bounded for p.*

PROOF. Straightforward, using ($\star$) in the proof of Theorem 5.4. □

On the other hand, as shown next, it is decidable if a program is h-bounded for some *given* h. Moreover, for programs that are h-bounded, transparency is decidable. In particular, it is decidable whether a program is simultaneously h-bounded and transparent. Indeed, we will see that the two together guarantee the existence of a view program that can be effectively constructed.

We first show that h-boundedness is decidable (see Appendix for proof). Intuitively, this holds because violations of h-boun-dedness are witnessed by minimum

p-faithful runs of bounded length, on initial instances of bounded size.

THEOREM 5.10. *It is decidable in* PSPACE*, given a program P, peer p, and integer h, whether P is h-bounded for p.*

Next, we show that transparency is decidable for h-bounded programs.

THEOREM 5.11. *The following are decidable in* PSPACE*:*

(i) given a program P that is h-bounded for peer p, whether P is transparent for p, and
(ii) given a program P, a peer p and an integer h, whether P is h-bounded and transparent for p.

The proof is provided in the appendix. Clearly, (ii) follows from (i) and Theorem 5.10. The proof of (i) relies on the existence of short counterexamples for violations of transparency by h-bounded programs.

REMARK 5.12. *Observe that the p-fresh instances used in the definition of transparency need not be reachable in actual runs of P. Thus, the definition has a "uniform" flavor, reminiscent of uniform containment for Datalog. Limiting transparency to reachable instances would yield a much weaker requirement, leading to undecidability of p-transparency even for h-bounded programs.*

Previously, we assumed that the boundedness parameter h is given. There are various ways to obtain h. One approach is heuristic: by examining traces of runs, one can "guess" h and then test h-boundedness using Theorem 5.10. Another possibility, briefly considered in Section 6, is to provide syntactic restrictions on the program P that ensure h-boundedness for some h computable from P. Alternatively, we introduce in Section 6 the means of ensuring *by design* transparency and h-boundedness of a program, for a given peer and desired h.

View-programs and provenance. We show that for each program P and peer p such that P is h-bounded and transparent for p, one can construct a view-program of P for p. As discussed earlier, the view program uses peers p and ω (for world). It contains the rules for p, and additional rules for ω that define the side-effects observed by p as a result of actions by other peers. We describe the construction of the rules for ω. Intuitively, the rules specify, for each instance $I@p$ visible at p, the possible updates to $I@p$ caused by minimal p-faithful runs of length up to h starting from I. The body of each rule specifies the tuples of $I@p$ causing the update, so intuitively provides the *provenance* of the update in terms of the data visible at p.

The view program $P@p$. We outline informally the construction of the view-program of P for p, that we denote $P@p$.

Let $C_{h+1} = \{a_1, ..., a_m\}$, where C_{h+1} is the set of constants (polynomial in P) defined in the proof of Theorem 5.10. For each $i \in [1, m]$, let x_i be a new distinct variable. Let ν be the mapping defined by $\nu(a_i) = x_i$ if $a_i \notin const(P)$ and $\nu(a_i) = a_i$ otherwise. Consider a p-fresh instance I and a sequence of events α of P, both over C_{h+1}, such that the tuples in $I(R)$ use only keys in $\mathcal{K}(R, \alpha)$ for each relation R, and α is a minimal p-faithful run of P on I in which all events but the last are invisible at p. Let $J = \alpha(I)$. Observe that by boundedness of P for p, $|\alpha| \leq h$, and so there are finitely many triples (I, α, J) as above. For each such triple (I, α, J), the view program $P@p$ contains a rule for peer ω constructed as follows. For each relation R:

- (positive body) for each t in $I@p(R)$, add $R@\omega(\nu(t))$ to the body.
- (negative body) for each a_i in $\mathcal{K}(R, \alpha)$ that is not a key value in $I@p(R)$, add $\neg Key_{R@\omega}(\nu(a_i))$ to the body.
- (inequalities) for all a_i, a_j where $i \neq j$, the body contains the inequality $\nu(a_i) \neq \nu(a_j)$.
- (head insertions) For each t in $J@p(R) - I@p(R)$, add $+R@\omega(\nu(t))$ to the head.
- (head deletions) For each a_i in $I@p(Key_{R@p}) - J@p(Key_{R@p})$, add $-Key_{R@\omega}(\nu(a_i))$ to the head.

Observe that $P@p$ is a syntactically valid program with schema $\mathcal{D}@p$ and peers p and ω. We next show that the construction is correct (see Appendix for proof).

THEOREM 5.13. *Given a program P that is h-bounded and transparent for peer p, the program $P@p$ is a view-program of P for p.*

Moreover, as suggested in Remark 5.2, the view-program $P@p$ constructed above is sound and complete not only with respect to the linear runs of P as viewed by p, required by the definition, but also with respect to its *tree* of runs as viewed by p. We omit the formal development.

6. TRANSPARENT PROGRAM DESIGN

Transparency and h-boundedness for a given peer may be desirable goals for some applications. There are various ways to achieve them. It is of course possible to first design the workflow program and then test it *a posteriori* for transparency and h-boundedness. However, it may be preferable in practice to specify directly view programs that are transparent and h-bounded *by design*. We begin the section by showing how this can be done by following some simple design guidelines. We then show that a large class of programs can be transformed so as to make them transparent and h-bounded, by filtering out the runs that violate these properties while preserving the runs that satisfy them (modulo some minor differences).

Before proceeding, we introduce some notions used throughout the section. Consider a run ρ, and a subsequence $e.\alpha.e'$ of consecutive events of ρ, of which the only events visible at p are e and e'. Then $\alpha.e'$ is a p-stage. In the following, each non-trivial stage ($\alpha \neq \epsilon$) will be equipped with a unique id, in order to identify the facts produced during that stage. Intuitively,

transparency is obtained by controlling the provenance of facts produced in that stage that lead to the visible event e'. Generating the stage ids can be done using a binary relation *Stage*, visible by all peers. *Stage* is either empty or contains a single tuple with key 0. A fact $Stage(0, s)$ indicates that the current stage id is s. When a peer q carries out an event visible at p, it deletes the current fact $Stage(0, s)$ (if such exists). A special rule, that can be performed by any peer q, inserts a new tuple $Stage(0, s')$ with a fresh value s'. Specifically, the rule is $+Stage@q(0, z) :- \neg\, Key_{Stage@q}(0)$. All rules generating events invisible at p are guarded by an atom $Stage(0, x)$ (so all p-invisible events of the stage are preceded by the event creating a new stage id).

Note that the above assumes that each peer q can tell whether its updates are visible at p. This is not always the case, but holds under certain conditions, such as (C1) further.

Transparency and boundedness by design. We introduce design guidelines to guarantee transparency and h-boundedness for a designated p. It turns out to be rather subtle to guarantee transparency while allowing other peers to perform arbitrary computations that do not impact p.

In order to ensure transparency of a program P for a peer p, we impose the following restrictions on program specifications, to be followed in the design process:

(C1) Each peer that sees a relation R visible at p (including p) sees it fully. Formally, for each relation $R@p \in \mathcal{D}@p$, if $R@q \in \mathcal{D}@q$ then $att(R@q) = att(R)$ and $\gamma(R@q) = true$.

(C2) The program maintains the *Stage* relation as previously described. Note that, because of (C1), every non-trivial update of a relation in $\mathcal{D}@p$, caused by any peer, is also visible at p. Thus, every peer can tell when it performs an update visible at p. As noted previously, this enables the maintenance of relation *Stage*.

(C3) The relations in $\mathcal{D}$ are separated in two disjoint classes: p-transparent and p-opaque. The relations that p sees are all p-transparent. The p-transparent relations that p does not see include an attribute, *StageID*, that contains the id of the stage in which the tuple was created.

(C4) If an event modifies some p-transparent relation, (i) only positive facts from p-transparent relations with the current stage id can occur in its body, and (ii) all the updates in the head are either updates of p-visible relations, creations of tuples with new keys in a p-transparent relation, or modifications of tuples in such a relation that have been created during the same stage and are visible by the peer performing the event.

When p is understood, we simply speak of transparent and opaque relations/facts. It is straightforward to specify syntactic criteria to guarantee (C1-C2-C3-C4). For instance, (C4)(ii) can be ensured as follows: if $+R@q(x, u)$ occurs in the head of a rule for a transparent p-invisible relation R, then either x is a variable that does not occur in the body (so it generates a new key) or $R@q(x, v)$ occurs in the body, where $v(StageID)$ is the current stage id provided by relation *Stage*.

Condition (C1) is natural in many applications where peers are doing some computations *about* a peer p, for which transparency is desired. For instance, p may be a customer, a job applicant, a participant in a crowdsourcing application, etc. Intuitively, (C1) prevents a peer from unknowingly performing some update that is visible at p.

We briefly elaborate on the motivation of (C4). Clearly, the use of opaque relations in rule bodies may lead to non-transparent computations. The restriction disallowing deletions from p-invisible transparent relations in heads of rules simplifies the presentation, but such deletions could be allowed at the cost of a more complex construction. The following example illustrates the motivation for prohibiting simultaneous updates of transparent and opaque relations in rule heads.

EXAMPLE 6.1. *Consider a workflow program P with a peer p, a p-visible relation R, and a p-invisible opaque relation T. Note that there is no transparent invisible relation. Suppose that P includes the following rules:*

+R@q(Sue, hire), +T@q(Sue, hire) :-
+R@q(Sue, reject), +T@q(Sue, reject) :-

The other peers may have silently computed for an arbitrarily long time, and derived $T(Sue, reject)$. This precludes application of the first rule. Intuitively, they have ruled out a possible future event for Sue without letting her know, thus violating transparency.

It is straightforward to see that a program satisfying (C1-C2-C3-C4) is transparent for p. Note that other peers may perform arbitrary computations as long as they do not affect what p sees. Observe that the transparent program shown in Example 5.7 follows the previous design guidelines.

We next show how to guarantee h-boundedness within a stage $\alpha.e'$ immediately following an event e. We wish to ensure that the minimum p-faithful subrun of $\alpha.e'$ contains no more than h events, leading to the activation of the visible event e'. This could be easily done by limiting the length of the entire stage to $h + 1$ using a propositional counter. However, this brute-force solution would be overly restrictive, because one often wishes to allow within the same stage an unbounded number of events that affect only p-opaque relations, or p-transparent relations in events not leading to e'.

Achieving this requires a more careful approach, in which the "steps" in each stage are identified by ids consisting of fresh values. More precisely, each event within the stage $\alpha.e'$ is called a *step*, and is identified by a step id. We will use the notion of *step-provenance* of a fact, i.e. the set of step ids in that stage that contribute to deriving the fact. In more detail, we equip each p-transparent relation invisible at p, say Q, with h additional columns $B_1 \cdots B_h$ that are used to record the step-provenance of each fact in the relation (ids of the steps contributing to its creation). When an update $+Q(u, b_1, ..., b_h)$ is performed, its set of non-$\perp$ B_i's

is set to the concatenation of all distinct non-$\perp$ values of the B_i's in the body of the event, augmented with a new id for the current step (shared by all insertions in the head). Recall that by (C4), there is no key deletion from Q. (In some sense, all the facts in Q are logically deleted when a new stage is entered.) Thus, a p-transparent event can be activated only if there is "enough room", i.e., if a sequence of at most h events of the stage are sufficient to enable this event. In particular, the last update of the stage has at most h non-$\perp$ B_i's. The events of the corresponding steps provide a p-equivalent sequence of length at most h, so the minimal p-faithful scenario for that stage has length at most h. Thus, the resulting run is h-bounded for p.

In summary, we can show:

THEOREM 6.2. *Each program obtained using the aforementioned guidelines is transparent and h-bounded for p.*

Boundedness by acyclicity. We next show how to guarantee boundedness for a certain class of programs using an acyclicity condition. We consider programs with single updates in heads of rules (which we call *linear-head* programs), satisfying (C1). For such a program, we define the p-graph of P as follows. Its nodes are the relations in $\mathcal{D}$, and there is an edge (intuitively "depends on") from R to Q if Q is invisible at p and there is a rule at some peer q whose head is $+R@q(u)$ or $-Key_{R@q}(h)$ and its body contains $Q@q(v)$ or $\neg Key_{Q@q}(k)$. The program P is p-acyclic if for each $R@p \in \mathcal{D}@p$, the subgraph of its p-graph induced by the nodes reachable from R is acyclic.

We can show the following (see Appendix for proof).

THEOREM 6.3. *Let P be a linear-head program over schema $\mathcal{D}$ satisfying (C1). Let b be the maximum number of facts in a rule body of P, $d = |\mathcal{D}|$, and a the maximum arity of a relation in $\mathcal{D}$ plus one. If P is p-acyclic then it is h-bounded for p, where $h = (ab+1)^d$.*

Enforcing transparency and boundedness. We next show, given a program P and peer p, how to rewrite P into a transparent and h-bounded program for p that has essentially the same behaviour as P except that it filters out the runs that are either not transparent or not h-bounded for p. We already defined these properties for programs. We need to define them for runs.

DEFINITION 6.4. *Let P be a program and p a peer. A run ρ of P is transparent for p if for each stage $\alpha.e'$ of ρ, the minimum p-faithful subrun $\alpha'.e'$ of $\alpha.e'$ has the following property. For any p-fresh instance J such that $I@p = J@p$, and $\mathrm{adom}(J) \cap \mathrm{new}(\alpha'.e') = \emptyset$, $\alpha'.e'$ is a minimum p-faithful run on J, all its events but e' are silent at p, and $\alpha'.e'(I)@p = \alpha'.e'(J)@p$. We say that ρ is h-bounded for p if $|\alpha'.e'| \leq h$ for every $\alpha'.e'$ as above. The set of transparent and h-bounded runs of P for p is denoted $tRuns_{p,h}(P)$.*

Our rewriting technique applies to the programs satisfying certain conditions, that we call *transparency-form*. Unlike the conditions used in the design guidelines, transparency-form does not require separating transparent and opaque relations at the schema level, instead allowing to make a more refined distinction at the fact level. In this more permissive setting, runs are no longer guaranteed to be transparent and h-bounded. However, we show how the violating runs can be filtered out.

DEFINITION 6.5. *A normal-form program is in* transparency-form (*TF for short*) *for p if it satisfies (C1-C2) and:*

- *(C3') For each rule of a peer $q \neq p$, if its head contains an update $+R@q(x, \bar{y})$ for some R that p does not see, either x is a variable that does not appear in the body (key creation) or the body contains an atom $R@q(x, \bar{z})$.*
- *(C4') For each relation R that p does not see, and each peer q such that $R@q \in \mathcal{D}@q$, the selection $\gamma(R@q)$ uses only attributes in $att(R@q)$.*

As discussed earlier, condition (C1) is meant to guarantee that a peer knows when it is performing an event visible at p, which enables maintaining the relation *Stage*.

(C3') is a natural condition that essentially comes down to preventing the "reuse" of a key after it has been deleted. The motivation for (C4') is more subtle. The presence of a fact in the view of some peer q may depend (because of selections) on some values that q does not see and that have been derived in a non-transparent manner. This may lead to violations of transparency that cannot be filtered out.

We will show the main result of this section for programs satisfying (C1-C2-C3'-C4'). We first need to introduce the notion of "run projection".

DEFINITION 6.6. *(run projection) Let P be a program over schema $\mathcal{D}$. Let Π be a schema consisting of a subset of the relations in $\mathcal{D}$, each having a subset of its attributes in $\mathcal{D}$ (always containing the key). Π induces a projection function on runs, defined as follows. The* projection $\Pi(\rho)$ *of a run ρ of P is obtained from ρ by removing facts and updates over relations not in Π, projecting out the missing attributes in facts and updates over relations in Π, and removing events with resulting empty heads. We say that Π is the identity for peer p if for every run of ρ of P, $\Pi(\rho)@p = \rho@p$.*

We extend this definition to a set *Runs* of runs and denote the result $\Pi(Runs)$. We now state the main result of the section:

THEOREM 6.7. *Let P be a TF program, p a peer, and h an integer. One can construct a program P^t that is transparent and h-bounded for p, and a projection function Π that is the identity for p, such that the runs of P that are transparent and h-bounded for p are exactly the projections of the runs of P^t, i.e., $tRuns_{p,h}(P) = \Pi(Runs(P^t))$.*

The construction of the program P^t is outlined further. From Theorems 5.13 and 6.7 it also follows that,

for an arbitrary TF program P, we can obtain a view program that specifies precisely the views at p of the runs of P that are transparent and h-bounded for p.

COROLLARY 6.8. *Let P be a TF program, p a peer, and h an integer. One can construct a view program P^t_p for P^t and p such that $Runs(P^t_p) = \{\rho@p \mid \rho \in tRuns_{p,h}(P)\}$.*

REMARK 6.9. *Note that, if the peers attempt to perform a non-transparent computation, the transformed program P^t will prevent carrying out the run and the computation may block. In practice, one might want to let the computation proceed and simply send an alert. Alternatively, one might wish to perform some "recovery", e.g., roll back to the state at the beginning of the stage. It is possible to modify P^t to implement such an alert or a roll-back.*

Program construction. We next outline the construction of the program P^t from the given TF program P. Intuitively, we need to identify, in each stage of a run of P, the "transparent facts" that have been obtained in a transparent manner within that stage. Transparent facts can only be created or updated by "transparent events", in which all the facts used in the body are transparent. More precisely, a positive fact is transparent at a particular time within a stage if it is p-visible, or if all events that participate in defining that fact up to that time, within that stage, are transparent. A negative fact is transparent if it concerns a p-visible relation or if its key was transparently created and transparently deleted in the same stage (recall that by (C3') keys cannot be reused after being deleted).

We next enrich the schema of P in order to keep track of the transparent facts. There are some subtleties in the process: (i) a p-invisible tuple may have portions that are transparent and portions that are not, (ii) step-provenance has to be recorded at the level of attributes rather than of the entire fact, and (iii) the system remembers which are the keys that were created and then deleted transparently during that stage.

The schema is modified as follows. Each relation R of P has a corresponding relation R^t in P^t. Tuples in R^t will use the same keys as in R; intuitively, the tuple of key k of R^t will hold information about the tuple of key k in R. For each attribute A of R, R^t has an attribute tA. For each q, tA has the same visibility as A, i.e. $tA \in att(R^t@q)$ iff $A \in att(R@q)$. For the key k, besides the attribute tK, the relation R^t includes an attribute dK with the same visibility as K. Intuitively, the attribute tA indicates if the value of the corresponding attribute was produced transparently (its value is $\perp$) or not (it has the particular value 1). Each p-visible fact is transparent by definition. The attribute dK is turned to 1 when the tuple is deleted transparently. Finally, for each $A \in att(R)$, R^t has attributes $A^s_1, ..., A^s_h$ in R^t, where A^s_1 holds the step id of the event that defined this attribute, and the others provide the step-provenance of that event (the list of step ids that lead to it).

A rule r in R at q is transformed into a set of transparent rules in P^t by adding new atoms and updates as follows. For each atom $R@q(k, u)$ in the body, we add an atom $R^t@q(k, ...)$ to the body to record extra information for k, and for each atom $\neg Key_{R@q}(k)$, an atom $R^t@q(k, ...)$ to the body to record extra information about k, including the fact that the deletion of k was transparent. For each update $+R@q(k, u)$, there exists an update $+R^t@q(k, ...)$, and for each update $-Key_{R@q}(k)$ an update $+R^t@q(k, ...)$. The information included in $+R^t@q$ is explained further.

Consider a fact $R@q(k, u)$ in a p-invisible relation. Suppose that the tuple with key k in $R^t@q$ satisfies: for each A in $att(R@q)$, the value of tA is $\perp$, the tuple stage (as provided by K^s_1) is the current stage id, and dK is $\perp$. Then $R@q(k, u)$ holds transparently. Now, consider $\neg Key_{R@q}(k)$ holds. Suppose that the tuple with key k in $R^t@q$ satisfies: tK is $\perp$, the tuple stage (as provided by K^s_1) is the current stage id, and dK is 1. Then $\neg Key_{R@q}(k)$ holds transparently.

To detail the use of the A^s_i attributes, consider the firing of a transparent event and let H be the set of step-IDs occurring in its body augmented with the current step-ID. Then:

- $\mid H \mid \leq h$.
- If the event modifies a non-key attribute A (there is a single step in the minimum p-faithful subrun of a stage that may do that), the set of non-$\perp$ values in the A^s_i attributes of the resulting tuple is H.
- If the event creates a tuple with a new key, the set of non-$\perp$ values of the K^s_i attributes of the created tuple equals H.
- If the event deletes a tuple already recording a set H_0 of step-IDs in its attributes K^s_i, the values in $H - H_0$ are added in still-available places in the K^s_i.

Note that this imposes that runs can only progress transparently as long as there is enough space available to record h step ids, which guarantees h-boundedness. It is also important to observe that, in transparent events, all updates are effective. This is guaranteed because the program is in TF.

The program also allows non-transparent events. These may come from the use of some non-transparent fact in the body. They may also come from the use of only transparent facts in the body, but such that the number of step ids occurring in them plus one is larger than h. When an event is not transparent, it is not allowed to modify a visible relation; it can only update other relations in an opaque manner.

The program P can be modified to incorporate the above information, allowing to trace transparency status and step ids. All the necessary information can be maintained as outlined above. Each rule of P yields at most exponentially many new rules resulting from a case analysis on the transparency status of the attributes, and the number of steps constributing to their generation.

One can show correctness of the construction, which yields Theorem 6.7 and Corollary 6.8.

7. RELATED WORK

A survey on data-centric business process management is provided in [22], and surveys on formal analysis of data centric workflows are presented in [11, 14].

Although not focused explicitly on workflows, Dedalus [7, 19] and Webdamlog [4, 2] are systems supporting distributed data processing based on condition/action rules. Local-as-view approaches are considered in a number of P2P data management systems, e.g., Piazza [29] that also consider richer mappings to specify views. Update propagation between views is considered in a number of systems, e.g., based on ECA rules in Hyperion [8].

Finite-state workflows with multiple peers have been formalized and extensively studied using communicating finite-state systems (called CFSMs in [1, 10], and *e-compositions* in the context of Web services, as surveyed in [20, 21]). Formal research on infinite-state, data-driven collaborative workflows is still in an early stage. The business artifact model [26] has pioneered data-driven workflows, but formal studies have focused on the single-user scenario. Compositions of data-driven web services are studied in [15], focusing on automatic verification. Active XML [3] provides distributed data-driven workflows manipulating XML data. A collaborative system for distributed data sharing geared towards life sciences applications is provided by the Orchestra project [18, 23].

Our model is an extension of the collaborative data-driven workflow of [6]. The results in [6] focus on the ability of peers to reason about temporal properties of global runs based on their local observations, and are orthogonal to the present investigation. To enable static analysis, the model of [6] uses more restricted views that those considered here.

Attaching provenance to facts derived in a rule-based language is considered in, e.g., [25, 5]. The paper [9] studies a notion of explanation in a model of data-centric workflows with a single user, no views, and no abstraction. They consider a notion of explanation that has some similarities to our faithful explanations, but is much simpler. Their results do not apply here.

There has been extensive work on causality and explanations in logic and AI. More specific to data-centric workflows, the relationship between provenance, explanations, and causality is considered in [12]. The focus is on provenance of data resulting from complex processes, such as scientific workflows.

The synthesis of view-programs described in Section 5 is related in spirit to partner synthesis in services modeled as Petri Nets [30, 24, 28].

The issue of transparency of algorithms is gaining increased attention, see e.g., the Data Transparency Lab (datatransparencylab.org/) in the US, and the Transalgo Lab starting in France. Data transparency has been studied in different contexts. For instance, the causality of machine-learning-based decisions is studied in [13].

Workflow transparency sometimes refers to the ability of considering a business process independently of the workflow implementing it, an aspect not considered here. Data transparency has also been considered in the context of workflows in [31], where an architecture for providing transparency in human-centric eGovernment workflows is proposed.

8. CONCLUSION

In this paper, we formally studied the problem of providing explanations of data-driven collaborative workflows to peers participating in the workflows, exploring semantic and computational issues.

We identified faithful scenarios for a peer p as a particularly appealing basis for explanations from a semantic viewpoint. In a first contribution, we show that faithful scenarios form a semiring with respect to union and intersection, implying the existence of a unique minimal faithful scenario for each peer, computable in polynomial time, and enabling incremental maintenance of scenarios. In a second contribution, we identified desirable properties of workflows, namely transparency and boundedness, that guarantee the existence of a view program for a peer p, and showed how such a program can be constructed.

Finally, we studied how programs satisfying transparency and h-boundedness for some peer p can be designed. We also show how, under certain restrictions, runs violating transparency or h-boundedness can be filtered out. A remaining open question is whether it is possible to perform such a filtering for arbitrary workflows.

It is possible to implement workflow programs by relying on a master server that has access to all the information, receives the updates, propagates them to appropriate peers, and controls transparency and boundedness for certain peers. Blockchain technology provides an alternative to such a central authority. It is, in spirit, an excellent match with collaborative workflows. A blockchain-based implementation of collaborative workflows is therefore a promising research direction with challenging technical issues, notably with respect to performance and access control.

Acknowledgment.

Serge Abiteboul and Pierre Bourhis are supported by the ANR Project Headwork, ANR-16-CE23-0015, 2016-2021. Victor Vianu is supported in part by the National Science Foundation under award IIS-1422375.

9. REFERENCES

[1] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Inf. and Comp.*, 127(2), 1996.

[2] Serge Abiteboul, Emilien Antoine, and Julia Stoyanovich. Viewing the web as a distributed knowledge base. In *ICDE*, 2012.

[3] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The Active XML project: an overview. *VLDB J.*, 17(5), 2008.

[4] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Emilien Antoine. A rule-based

language for web data management. In *PODS*, pages 293–304, 2011.
[5] Serge Abiteboul, Pierre Bourhis, and Victor Vianu. A formal study of collaborative access control in distributed datalog. In *ICDT*, 2016.
[6] Serge Abiteboul and Victor Vianu. Collaborative data-driven workflows: think global, act local. In *PODS*, 2013.
[7] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In *Datalog*, pages 262–281, 2010.
[8] Marcelo Arenas, Vasiliki Kantere, Anastasios Kementsietsidis, Iluju Kiringa, Renée J. Miller, and John Mylopoulos. The hyperion project: from data integration to data coordination. *SIGMOD Record*, 32(3):53–58, 2003.
[9] Pierre Bourhis, Daniel Deutch, and Yuval Moskovitch. Analyzing data-centric applications: Why, what-if, and how-to. In *ICDE*, 2016.
[10] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *JACM*, 30(2), 1983.
[11] Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. Foundations of data-aware process analysis: a database theory perspective. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, 2013.
[12] James Cheney. Causality and the semantics of provenance. In *Proceedings Sixth Workshop on Developments in Computational Models: Causality, Computation, and Physics, DCM 2010, Edinburgh, Scotland, 9-10th July 2010.*, pages 63–74, 2010.
[13] Anupam Datta, Shayak Sen, and Yair Zick. Algorithmic transparency via quantitative input influence: Theory and experiments with learning systems. In *IEEE Symposium on Security and Privacy, San Jose, CA, 2016*, pages 598–617, 2016.
[14] A. Deutsch, R. Hull, and V. Vianu. Automatic verification of database-centric systems. *SIGMOD Record*, 43(3):5–17, 2014.
[15] Alin Deutsch, Liying Sui, Victor Vianu, and Dayou Zhou. Verification of communicating data-driven web services. In *PODS*, 2006.
[16] Guozhu Dong and Jianwen Su. Incremental maintenance of recursive views using relational calculus/sql. *SIGMOD Record*, 29(1):44–51, 2000.
[17] M. R. Garey and D. S. Johnson. *Computers and Intractibilitiy: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.
[18] Todd J. Green, Gregory Karvounarakis, Nicholas E. Taylor, Olivier Biton, Zachary G. Ives, and Val Tannen. Orchestra: facilitating collaborative data sharing. In *SIGMOD Conference*, 2007.
[19] Joseph M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1), 2010.
[20] Richard Hull. Web services composition: A story of models, automata, and logics. In *ICSOC*, 2005.
[21] Richard Hull and Jianwen Su. Tools for composite web services: a short overview. *SIGMOD Record*, 34(2):86–95, 2005.
[22] Richard Hull, Jianwen Su, and Roman Vaculín. Data management perspectives on business process management: tutorial overview. In *SIGMOD Conference*, pages 943–948, 2013.
[23] Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. The orchestra collaborative data sharing system. *SIGMOD Record*, 37(3), 2008.
[24] Niels Lohmann and Daniela Weinberg. Wendy: A tool to synthesize partners for services. In *Applications and Theory of Petri Nets*, pages 297–307. Springer Berlin Heidelberg, 2010.
[25] Vera Zaychik Moffitt, Julia Stoyanovich, Serge Abiteboul, and Gerome Miklau. Collaborative access control in Webdamlog. In *SIGMOD Conference*, 2015.
[26] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.
[27] E. L. Post. Recursive unsolvability of a problem of Thue. *J. of Symbolic Logic*, 12:1–11, 1947.
[28] Jan Sürmeli and Marvin Triebel. Synthesizing cost-minimal partners for services. In *Service-Oriented Computing*, pages 584–591, 2013.
[29] Igor Tatarinov, Zachary G. Ives, Jayant Madhavan, Alon Y. Halevy, Dan Suciu, Nilesh N. Dalvi, Xin Dong, Yana Kadiyska, Gerome Miklau, and Peter Mork. The piazza peer data management project. *SIGMOD Record*, 32(3):47–52, 2003.
[30] Karsten Wolf. Does my service have partners? *Trans. Petri Nets and Other Models of Concurrency*, 2:152–171, 2009.
[31] Christian Wolter, Henrik Plate, and Cedric Hebert. Collaborative workflow management for egovernment. In *International Workshop on Database and Expert Systems Applications (DEXA)*, pages 845–849, 2007.

APPENDIX

A. APPENDIX

We provide proof outlines for several results of the paper.

Proof of Theorem 3.4 Membership in coNP is immediate. For hardness, we reduce the problem of testing unsatisfiability of a Boolean formula to testing whether a scenario for some peer p is minimal. Let φ be a formula over some set $X = \{x_1, ..., x_n\}$ of variables. We assume without loss of generality that (*) φ does not

hold for the valuation mapping all variables to true. Let R be a relation of arity $n+2$, with key K, attributes A_x for each $x \in X$, and a last attribute A_q. For each variable $x \in X$, there is a peer p_x that sees the projection of R over K, A_x. There is a peer q that sees K, A_q. In addition, there is a peer p that sees the projection of R on K with the selection $\sigma_p = (A_q = 1) \wedge (\gamma \vee \gamma_\varphi)$ where $\gamma = \wedge_{x \in X}(A_x = 1)$, and γ_φ checks that the formula φ is true for the valuation ν of X such that $\nu(x) = true$ iff $A_x = 1$.

The program consists of all ground rules of the form

$$\begin{array}{lll} r_{x_i}: & +R@p_{x_i}(0,1) & :- \quad (\text{for each } x_i \in X) \\ e: & +R@q(0,1) & :- \end{array}$$

Consider the run ρ consisting of $r_{x_1} \ldots r_{x_n} e$. Observe that p sees $R@p(0)$ only after the last event, because of the condition $A_q = 1$ in its selection condition on R.

We prove that φ is not satisfiable iff ρ is a minimal scenario of ρ at p. First suppose that φ is satisfiable. Let ν be a valuation satisfying φ. Consider the subsequence of $r_{x_{i_1}} \ldots r_{x_{i_k}} e$ obtained by keeping only the events $r_{x_{i_j}}$ such that $\nu(x_{i_j})$ is true. By (*), it is shorter than the original sequence. Let ρ_ν be the corresponding run. It is easy to see that ρ_ν is a strict subrun of ρ and that $\rho_\nu@p = \rho@p$. Thus ρ is not minimal at p.

Now suppose that φ is not satisfiable. Let ρ' be a scenario of ρ at p. Because φ is not satisfiable, γ_φ can never be satisfied, so in order for p to see $R@p(0)$, it is necessary that γ hold, so ρ' must contain all events in ρ. Thus, ρ is minimal.

Proof of Lemma 4.6 Let $\rho = \{(e_i, I_i)\}_{0 \le i \le n}$ and $\alpha = \{e_{i_j}\}_{0 \le j \le m}$. We prove by induction on h $(0 \le h \le m)$ that

(†) $\alpha|_h = \{e_{i_j}\}_{0 \le j \le h}$ yields a subrun $\{(e_{i_j}, I'_j)\}_{0 \le j \le h}$ of ρ and $I'_h@p = I_{i_h}@p$.

Suppose (†) holds. Then α yields a subrun of ρ, establishing (i). Additionally, $I'_j@p = I_{i_j}@p$ for $0 \le j \le m$. This together with the fact that α includes all events in ρ visible at p implies that $run(\alpha)@p = \rho@p$. Thus, $run(\alpha)$ is a scenario of ρ for p, establishing (ii)

We now prove (†). For the basis, let $h = 0$. We need to show that (a) e_{i_0} is applicable to the empty instance and (b) $I'_0@p = I_{i_0}@p$. For (a), suppose the body of e_{i_0} contains a literal $R@q(k,u)$. Then i_0 belongs to an R-lifecycle of k in ρ, whose left boundary must be included in α, a contradiction. Thus the body of e_{i_0} contains no literal of the form $R@q(k,u)$ and is applicable to the empty instance, proving (a). Now consider (b). Consider $t \in I'_0@p(R@p)$ with key k. Then e_{i_0} must insert in R a tuple t' with the same key, such that $t'@p = t$. Thus, i_0 belongs to the R-lifecycle of k in ρ, and in fact it must be its left boundary (otherwise, by boundary faithfulness, e_{i_0} must be preceded by another event in α, a contradiction). It follows that I'_0 and I_{i_0} both contain t', so $t \in I_{i_0}@p(R@p)$. Thus, $I'_0@p \subseteq I_{i_0}@p$. Conversely, let $t \in I_{i_0}@p(R@p)$ with key k. Let h' $(h' \le i_0)$ be the minimum in the same R-lifecycle of k in ρ, for which a tuple with key k is visible by p in $I_{h'}@p(R@p)$. It follows that $e_{h'}$ is visible at p so it is included in α, so $h' = i_0$ and e_{i_0} must insert a tuple t' with key k. Moreover, i_0 must also be the left boundary of the R-lifecycle of k (or else, that left boundary would have to be included in α prior to e_{i_0}). It follows that $t'@p = t$. Thus, $t \in I'_0(R@p)$, and $I_{i_0} \subseteq I'_0$. This completes the basis.

For the induction step, suppose $\{e_{i_j}\}_{0 \le j \le h}$ yields a subrun $\{(e_{i_j}, I'_j)\}_{0 \le j \le h}$ of ρ where $I'_h@p = I_{i_h}@p$, for $h < m$, and consider $e_{i_{h+1}}$. For (a), we need to show that $e_{i_{h+1}}$ is applicable in I'_h. Let $R@q(k,u)$ occur in the body of $e_{i_{h+1}}$. Then i_h belongs to an R-lifecycle of k in ρ, and, by modification faithfulness, all prior events of the R-lifecycle that affect attributes in $att(R,q)$ of tuples with key k are included in α. It follows that $I'_h(R)$ and $I_{i_h}(R)$ both contain tuples with key k, that agree on $att(R,q)$. Thus, since $R@q(k,u)$ holds in I_{i_h}, it also holds in I'_h. Next, suppose $\neg Key_R@q(k)$ occurs in the body of $e_{i_{h+1}}$. Suppose $k \in I_{i_h}(Key_R)$. Then, similarly to the previous case, $I'_h(R)$ and $I_{i_h}(R)$ both contain tuples with key k, that agree on $att(R,q)$, so $\neg Key_R@q(k)$ holds in I'_h. Now suppose that $k \notin I_{i_h}(Key_R)$ but $k \in I'_h(Key_R)$. Let $v < h$ be the left boundary of the R-lifecycle in $run(\alpha|_h)$ to which h belongs. It follows that i_v belongs to an R-lifecycle of k in ρ but i_h does not, so the R-lifecycle has a right boundary in ρ occurring before i_h, which by boundary faithfulness must also belong to α. This contradicts the fact that h is in an R-lifecyle of k in $run(\alpha|_h)$. So, $k \notin I'_h(Key_R)$ and $\neg Key_R@q(k)$ holds in I'_h. In summary, $e_{i_{h+1}}$ is applicable in I'_h.

Now consider (b). Let $t \in I'_{h+1}@p(R@p)$ where t has key k. Thus, $I'_{h+1}(R)$ contains a tuple t' with key k such that $t'@p = t$. From boundary and modification faithfulness it easily follows that $I_{i_{h+1}}(R)$ contains a tuple t'' with key k that agrees with t' on $att(R,p)$, so $t''@p = t'@p = t$ and $t \in I_{i_{h+1}}@p(R@p)$. Thus, $I'_{h+1}@p \subseteq I_{i_{h+1}}@p$. Conversely, let $t \in I_{i_{h+1}}@p(R@p)$ with key k. Similarly to the base case, let h' be the minimum in the same R-lifecycle of k in ρ, for which a tuple with key k is visible by p in $I_{h'}@p(R@p)$. It follows that $e_{h'}$ is visible at p so is included in α. Clearly, $e_{h'}$ must contain an insertion of a tuple with key k into R. From boundary and modification faithfulness it follows that $I_{i_{h+1}}(R)$ and $I'_{h+1}(R)$ contain tuples with key k that agree on $att(R,p)$, so $t \in I'_{h+1}@p(R@p)$. Thus, $I_{i_{h+1}}@p \subseteq I'_{h+1}@p$. This completes the induction and the proof of (†). □

Proof of Theorem 4.8 We first note the following useful fact, that follows immediately from the definition of $T_p(\rho, \cdot)$.

LEMMA A.1. *The operator $T_p(\rho, \cdot)$ is additive: for all subsequences α_1, α_2 of $e(\rho)$, $T_p(\rho, \alpha_1 + \alpha_2) = T_p(\rho, \alpha_1) + T_p(\rho, \alpha_2)$.*

We now turn to the proof of the theorem. Let ρ_1 an ρ_2 be p-faithful scenarios of ρ. Consider first $e(\rho_1) + e(\rho_2)$. By definition, $e(\rho_1) + e(\rho_2)$ contains all events of ρ visible at p. By additivity of $T_p(\rho, \cdot)$, $T_p(\rho, e(\rho_1) +$

$e(\rho_2)) = T_p(\rho, e(\rho_1)) + T_p(\rho, e(\rho_2)) = e(\rho_1) + e(\rho_2)$. Thus, $e(\rho_1) + e(\rho_2)$ is also a fixed-point of $T_p(\rho, \cdot)$ and so it is a p-faithful scenario of ρ.

Now consider $e(\rho_1) * e(\rho_2)$. Since ρ_1 and ρ_2 are p-faithful scenarios, $e(\rho_1)$ and $e(\rho_2)$ are fixed-points of $T_p(\rho, \cdot)$ and contain all events of ρ visible at p. Since $e(\rho_1) * e(\rho_2) \ll e(\rho_1)$ and $e(\rho_1) * e(\rho_2) \ll e(\rho_2)$, it follows that $T_p^\omega(\rho, e(\rho_1) * e(\rho_2)) \ll e(\rho_1)$ and $T_p^\omega(\rho, e(\rho_1) * e(\rho_2)) \ll e(\rho_2)$, so

$$T_p^\omega(\rho, e(\rho_1) * e(\rho_2)) = e(\rho_1) * e(\rho_2)$$

It follows that $e(\rho_1) * e(\rho_2)$ is a fixed-point of $T_p(\rho, \cdot)$. Since $e(\rho_1) * e(\rho_2)$ also contains all events visible at p, by Lemma 4.6, $e(\rho_1) * e(\rho_2)$ yields a p-faithful scenario of ρ.

Finally, observe that multiplication distributes over addition, ϵ is the additive identity and ρ the multiplicative identity. □

Proof of Theorem 5.10 We begin with two technical lemmas. The first essentially says that various properties of sequences of events are invariant under isomorphism.

LEMMA A.2. *Let I be an instance and $\alpha = e_1 \dots e_n$ a sequence of events applicable at I. Let f be a bijection on dom that is the identity on const(P). We denote by $f(\alpha)$ the sequence of events obtained by applying f to every value occurring in α. Then the following hold:*

(i) $f(\alpha)$ yields a run on $f(I)$, and $f(\alpha(I)) = f(\alpha)(f(I))$,
(ii) α is a (minimum) p-faithful run on I iff $f(\alpha)$ is a (minimum) p-faithful run on $f(I)$, and the events visible at p are the same in the two runs.

PROOF. Straightforward induction on the length of α. □

We also need the following. Recall that, for each relation R, $\mathcal{K}(R, \alpha)$ denotes the set of values occurring as keys of relation R in some event of α. For an instance I, we denote by $I|\mathcal{K}(\alpha)$ the instance retaining, for each relation R, only the tuples in $I(R)$ with keys in $\mathcal{K}(R, \alpha)$.

LEMMA A.3. *Let I be an instance, α a sequence of events, and $I|\mathcal{K}(\alpha) \subseteq J \subseteq I$. The following hold:*

(i) if α is a (minimum) p-faithful run on I then it is also a (minimum) p-faithful run on J, and the events visible at p are the same in the two runs.
(ii) if α is a (minimum) p-faithful run on J such that $\mathrm{adom}(I) \cap \mathrm{new}(\alpha) = \emptyset$ then α is also a (minimum) p-faithful run on I, and the events visible at p are the same in the two runs.

PROOF. Also by induction on the length of α. The only subtlety concerns new values. For (i), note that, if an event of α creates a new value on I, that value is also new in the run on J, since $J \subseteq I$. For (ii), the converse holds because $adom(I) \cap new(\alpha) = \emptyset$ ensuring that the new values created in α do not occur in I. □

We can now prove Theorem 5.10. By definition, P is not h-bounded iff (‡) there is an instance I and sequence α of events, of length $h+1$, that yields a minimim p-faithful run on initial instance I, such that all of its events but the last are silent at p. Let c_m be the maximum number of values occurring in a sequence α of events of length at most m and an instance I such that the tuples in $I(R)$ use only keys in $\mathcal{K}(R, \alpha)$ for each relation R. Let $\bar{c}_m = |const(P)| + c_m$ and C_m consist of $const(P)$ together with c_m additional distinct constants (so $|C_m| = \bar{c}_m$). By Lemmas A.2 and A.3 (i), it is sufficient to check (‡) for sequences α of events of length at most $h+1$ and instances I, both using only values in C_{h+1}. This establishes decidability. The PSPACE upper bound follows from the fact that $\bar{c}_{h+1}$ is polynomial in h and P, which yields a non-deterministic PSPACE test. This completes the proof. □

Proof of Theorem 5.11 Clearly, (ii) follows from (i) and Theorem 5.10. Consider (i). Let P be a program that is h-bounded for p. By a slight reformulation of the definition of transparency, P is transparent for p iff the following holds.

(†) For all instances I_1, I_2 and events e_1, e_2 such that e_i is applicable at I_i and visible at p $(i = 1, 2)$ and $e_1(I_1)@p = e_2(I_2)@p$, and for each sequence α of events such that $adom(e_2(I_2)) \cap new(\alpha) = \emptyset$, if α is a minimum p-faithful run on $e_1(I_1)$ such that all its events but the last are silent at p, then the same holds on $e_2(I_2)$, and $\alpha(e_1(I_1))@p = \alpha(e_2(I_2))@p$.

We show that (†′): (†) holds iff it holds for all instances I_1', I_2' such that, for each relation R, $I_i'(R)$ contains at most $\bar{c}_{|\alpha|+2}$ tuples. For suppose this holds. Since P is h-bounded, it is sufficient to check (†) for instances I_1 and I_2 with at most $\bar{c}_{h+2}$ tuples in each relation. The existence of counterexamples can be easily checked by a nondeterministic PSPACE algorithm. This completes the proof.

We now show (†′). The "only if" part is trivial. Consider the "if" part. Suppose (†) holds for all instances I_1', I_2' such that, for each relation R, $I_i'(R)$ contains at most $\bar{c}_{|\alpha|+2}$ tuples. Let I_1, I_2 be arbitrary instances, e_1, e_2 events such that e_i is applicable at I_i and visible at p $(i = 1, 2)$, $e_1(I_1)@p = e_2(I_2)@p$, α is a minimum p-faithful run on $e_1(I_1)$ such that all but its last event are silent at p, and $adom(e_2(I_2)) \cap new(\alpha) = \emptyset$.

We can assume without loss of generality that $adom(I_2) \cap new(\alpha) = \emptyset$; otherwise, we can rename the values in I_2 and e_2 that occur in the intersection by a bijection that is the identity on $const(P) \cup adom(e_2(I_2)) \cup adom(\alpha)$ and use Lemma A.2.

Let $\mathcal{K}_{1,2} = \mathcal{K}(e_1) \cup \mathcal{K}(e_2) \cup \mathcal{K}(\alpha)$, $I_1' = I_1|\mathcal{K}_{1,2}$ and $I_2' = I_2|\mathcal{K}_{1,2}$. Note that each relation in I_1' and I_2' contains at most $\bar{c}_{|\alpha|+2}$ tuples. We next show that I_1', I_2' satisfy the conditions of (†).

By Lemma A.3 (i), e_i is applicable to I_i' and is visible at p $(i = 1, 2)$. Moreover, α is a minimum p-faithful run on $e_1(I_1')$ and all but its last event are silent at p. We show that $e_1(I_1')@p = e_2(I_2')@p$. Let $t \in e_1(I_1')@p(R@p)$ with key k for some R. Observe

that $k \in \mathcal{K}_{1,2}$. Since $e_1(I_1') \subseteq e_1(I_1)$ and $e_1(I_1)@p = e_2(I_2)@p$, $t \in e_2(I_2)@p(R@p)$ and there is $t' \in e_2(I_2)(R)$, with the same key k as t, such that $t = t'@p$. Suppose there is no tuple with key k in I_2, so t' is created by e_2. Then t' is also in $e_2(I_2')$ and $t \in e_2(I_2')@p(R@p)$. Now suppose there is a tuple t'' with key k in $I_2(R)$. Since $k \in \mathcal{K}_{1,2}$, $t'' \in I_2'(R)$ and so $t' \in e_2(I_2')(R)$ and $t \in e_2(I_2')@p(R@p)$. We have shown that $e_1(I_1')@p \subseteq e_2(I_2')@p$. The converse holds by symmetry, so $e_1(I_1')@p = e_2(I_2')@p$. Also, $adom(e_2(I_2')) \cap new(\alpha) = \emptyset$.

Since I_1' and I_2' satisfy the condition of (†), it follows that α is a minimum p-faithful run on $e_2(I_2')$ such that all but its last event are silent at p, and $\alpha(e_1(I_1'))@p = \alpha(e_2(I_2'))@p$. By Lemma A.3 (ii), α is also a minimum p-faithful run on $e_2(I_2)$ such that all but its last event are silent at p.

It remains to show that $\alpha(e_1(I_1))@p = \alpha(e_2(I_2))@p$. Let $t \in \alpha(e_1(I_1))@p(R@p)$ with key k for some R. Thus, there exists a tuple $t' \in \alpha(e_1(I_1))(R)$, with key k, such that $t = t'@p$. First suppose $k \notin \mathcal{K}_{1,2}$, then t' also belongs to $e_1(I_1)$. Since $e_1(I_1)@p = e_2(I_2)@p$, $t \in e_2(I_2)@p(R@p)$. Thus, there is $t'' \in e_2(I_2)(R)$ with key k, such that $t''@p = t$. Since $k \notin \mathcal{K}_{1,2}$, α does not affect t'', so $t'' \in \alpha(e_2(I_2))(R)$ and $t \in \alpha(e_2(I_2))@p(R@p)$. Now suppose $k \in \mathcal{K}_{1,2}$. If there is no tuple in $e_1(I_1)(R)$ with key k, then α creates t' on any initial instance on which it is applicable, so $t' \in \alpha(e_2(I_2))(R)$ and $t \in \alpha(e_2(I_2))@p(R@p)$. Suppose there is a tuple t'' in $e_1(I_1)(R)$ with key k. Since $k \in \mathcal{K}_{1,2}$, $t'' \in e_1(I_1')(R)$. It follows that $t' \in \alpha(e_1(I_1'))(R)$ and $t \in \alpha(e_1(I_1'))@p(R@p)$. Since $\alpha(e_1(I_1'))@p = \alpha(e_2(I_2'))@p$, $t \in \alpha(e_2(I_2'))@p$. Since α is applicable to $e_2(I_2)$ and $I_2' \subseteq I_2$, it follows that $\alpha(e_2(I_2'))@p \subseteq \alpha(e_2(I_2))@p$, and $t \in \alpha(e_2(I_2))@p(R@p)$. In both cases, $t \in \alpha(e_2(I_2))@p(R@p)$, thus, $\alpha(e_1(I_1))@p \subseteq \alpha(e_2(I_2))@p$. The converse holds by symmetry. Hence, $\alpha(e_1(I_1))@p = \alpha(e_2(I_2))@p$, which concludes the proof. □

Proof of Theorem 5.13 We need to show that $P@p$ is sound and complete for P and p.

Consider completeness. For runs ρ of P and ρ' of $P@p$, we denote by $\rho@p \sim \rho'$ that fact that $\rho@p$ is obtained from ρ' by replacing all ω-events with ω. Let $\rho = \{(e_i, I_i)\}_{0 \le i \le n}$ be a run of P. We can write $e(\rho)$ as $\alpha_1.e_{i_1}.\alpha_2.e_{i_2} \ldots \alpha_n.e_{i_n}.\alpha_{n+1}$ where e_{i_j} are the events visible at p ($1 \le j \le n$) and α_j are sequences of events invisible at p ($1 \le j \le n+1$). We define a sequence of events $E_1 \ldots E_n$ yielding a run of $P@p$, such that $\rho@p \sim run(E_1 \ldots E_n)$. If e_{i_j} is an event of p, then $E_j = e_{i_j}$. Consider a fixed $j > 1$ for which e_{i_j} is not en event of p (the case $j = 1$ is a straightforward variation). Let $e = e_{i_j}$, $e' = e_{i_{(j-1)}}$, $\alpha = \alpha_j$, $I = I_{i_{(j-1)}}$, $I' = I_{i_{(j-1)}-1}$, and $J = I_{i_j}$. Let $\bar{\alpha}$ be the subsequence of α such that $\bar{\alpha}.e$ is the unique minimum p-faithful subrun of $\alpha.e$ on initial instance I. Since P is h-bounded for p, $|\bar{\alpha}.e| \le h$. Let $\bar{I} = I|\mathcal{K}(e'.\bar{\alpha}.e)$. By Lemma A.3 (i), $\bar{\alpha}.e$ is a minimum p-faithful run of P on $\bar{I}$, all events of $\bar{\alpha}$ are invisible at p, and e is visible at p. Also, $\bar{I}$ is a p-fresh instance, since it is easily seen that $\bar{I} = e'(I'|\mathcal{K}(e'.\alpha.e))$ and e' is visible at p. Let $\bar{J} = \bar{\alpha}.e(\bar{I})$. Observe that $|\bar{I}| \le c_{h+1}$. By Lemma A.2 we can assume without loss of generality that $\bar{I}$ and $\bar{\alpha}.e$ use only constants in C_{h+1}. Consider the rule of $P@p$ corresponding to the triple $(\bar{I}, \bar{\alpha}.e, \bar{J})$. Consider the event E_j obtained by applying to the variables of the rule the valuation ν^{-1}. Clearly, the event is applicable to $\bar{I}@p$ and $E_j(\bar{I}@p) = \bar{J}@p$. It remains to show that $E_j(I@p) = J@p$. Let $\bar{I}^c = I - \bar{I}$. By definition, $\bar{I}^c$ contains no tuple affected by $\bar{\alpha}.e$, so $J = \bar{J} \cup \bar{I}^c$. Similarly, no tuple of $\bar{I}^c$ $@p$ is affected by E_j. It follows that $E_j(I@p) = E_j(\bar{I}@p) \cup \bar{I}^c$ $@p = \bar{J}@p \cup \bar{I}^c$ $@p = J@p$. This establishes completeness of $P@p$.

Now consider the soundness of $P@p$. Let $\rho_p = \{(E_i, I_i)\}_{0 \le i \le n}$ be a run of $P@p$. We show that there exists a run ρ of P such that $\rho@p \sim \rho_p$. Let $\rho_p|_j = \{(E_i, I_i)\}_{0 \le i \le j}$. We prove the statement by induction on j. Consider $j = 0$. Thus, $E_0(\emptyset) = I_0$. If E_0 is an event of p then the statment holds. Otherwise, by definition, there exists a p-fresh instance I and a minimum p-faithful run $\alpha.e$ of P on I, such that the tuples in $I(R)$ use keys in $\mathcal{K}(R, \alpha)$ for each relation R, and $\alpha.e(I)@p = E_0(I@p)$. By construction, since E_0 has no positive atoms in its body, $I@p = \emptyset$. By transparency of P, $\alpha.e$ is also a minimum p-faithful run of P on $\emptyset$ in which all events but e are invisible at p, and $\alpha.e(\emptyset)@p = \alpha.e(I)@p = E_0(\emptyset)$. This completes the base of the induction.

For the induction step, let $0 < j < n$ suppose there is a run ρ_j of P such that $\rho_j@p \sim \rho_p|_j$. Let $e(\rho_j) = \alpha_j$ and $\alpha_j(\emptyset) = J$. So J is a p-fresh instance and $J@p = I_j$. From the definiton of E_{j+1}, it can be shown similarly to the base case that there exists a p-fresh instance I over $\mathcal{D}$, and a minimum p-faithful run $\alpha.e$ of P on initial instance I, in which all events but e are invisible at p, such that $I@p = I_j$ and $\alpha.e(I)@p = I_{j+1}$. We would like to obtain a run corresponding to $\rho_p|_{j+1}$ by concatenating α_j with $\alpha.e$. However, it could be that $new(\alpha.e) \cap adom(\alpha_j) \neq \emptyset$. Observe that $new(\alpha.e) \cap adom(I_j) = \emptyset$ since $adom(I_j) \subseteq adom(I)$. Thus, there are two cases to handle:

(i) $new(\alpha.e)$ contains values in $adom(\rho_p|_{j-1})$,
(ii) $new(\alpha.e)$ contains values in $adom(\alpha_j) - adom(\rho_p|_{j-1})$

Case (i) can be handled by applying to $\alpha.e$ a bijection f on dom that is the identity on $const(P) \cup adom(I) \cup adom(I_{j+1})$ and such that $adom(f(new(\alpha.e))) \cap adom(\rho_p|_{j-1}) = \emptyset$, and using Lemma A.2. Case (ii) can then be avoided by applying to α_j a bijection g on dom that is the identity on $const(P) \cup adom(\rho_p|_j)$ and such that $adom(g(\alpha_j)) \cap new(\alpha.e) = \emptyset$, and using again Lemma A.2.

Thus, we can assume that $new(\alpha.e) \cap adom(\alpha_j) = \emptyset$. By transparency of P, since I and J are p-fresh, $I@p = J@p$, and $adom(J) \cap new(\alpha.e) = \emptyset$, it follows that $\alpha.e$ is a run of P on J and $\alpha.e(J)@p = \alpha.e(I)@p = I_{j+1}$. Thus, $\alpha_j.\alpha.e$ yields a run of P and $run(\alpha_j.\alpha.e) \sim \rho_p|_{j+1}$. This completes the induction and the proof of soundness of $P@p$. □

Proof of Theorem 6.3 Consider an instance I and a sequence $\alpha.e$ of events applicable to I that yields a minimum p-faithful run at p, such that all its events but e are invisible at p. Observe that no event of α

has a relation visible at p in the head. Since $\alpha.e$ is a minimum p-faithful run, $\alpha.e = T_p^\omega(\alpha.e, e)$. Let R be the relation occurring in the head of e. Since e is visible at p, $R@p \in \mathcal{D}@p$. It can be shown that $T_p^\omega(\alpha.e, e) = T_p^g(\alpha.e, e)$, where g is the maximum length of a path in the p-graph of P, starting from relation R. Intuitively, this is because every productive application of $T_p(\alpha.e, \cdot)$ to an event corresponds to traversing at least one edge in the p-graph (from the relation in the head to some in the body). It follows that $T_p(\alpha.e, e)$ can only be applied g times before converging. Moreover, given a set E of events, $T_p(\alpha.e, E)$ adds to E at most $b \cdot |E|$ lifecycles of keys, each containing at most a events. It follows that $|\alpha.e| \leq (ab+1)^g \leq (ab+1)^d$. □

Proof of Theorem 6.7 (sketch) Let P be a TF program, p a peer, and h an integer. Let P^t be the program constructed from P as previously described. We show that

(*) P^t is transparent and h-bounded for p, and

(**) each run of P that is transparent and h-bounded for p is the projection of a run of P^t.

Towards (*), consider a p-fresh instance I and a minimal p-faithful sequence α of events of P^t such that only the last one is visible for p. Transparency is satisfied by construction. For h-boundedness, observe that the subrun consisting of the events corresponding to the stepIDs occurring in the B_i attributes of the last event of α is observationally equivalent to α for p. Therefore h-boundedness holds as well.

Towards (**), first observe that the projection simply removes from the runs of P^t, the relations R^t for each R. Consider a transparent and h-bounded run ρ of P. Let its stages be α_i for $i \in [1..n]$, and I_i be the instance reached after α_i for each i. We construct a corresponding run ρ' of P^t. For this, for each i, consider the minimal faithful subrun α_i' of α_i on input I_{i-1}. It is transparent and, by h-boundedness, its length is less or equal to h. We can therefore extend the events of P to transparent events of P^t to capture the events in α_i'. Because its length is less that h, we dispose of enough space in the B_i. For the other events, it is irrelevant whether we extend them using transparent or non transparent events. We thus obtain a run ρ' such that $\rho = \Pi(\rho')$, which concludes the proof. □