Abstracting PROV provenance graphs: a validity-preserving approach

P. Missier[a,*], J. Bryans[b,*], C. Gamble[a], V. Curcin[c]

[a] *School of Computing, Newcastle University, UK*
[b] *Institute for Future Transport and Cities, Coventry University, UK*
[c] *Kings College, London, UK*

Abstract

Data provenance is a structured form of metadata designed to record the activities and datasets involved in data production, as well as their dependency relationships. The PROV data model, released by the W3C in 2013, defines a schema and constraints that together provide a structural and semantic foundation for provenance. This enables the interoperable exchange of provenance between data producers and consumers. When the provenance content is sensitive and subject to disclosure restrictions, however, a way of partially obfuscating provenance in a principled way before communicating it to certain parties is required. In this paper we present a provenance abstraction operator that achieves this goal. It maps a PROV document P_1 to a new abstract version P_2, with the properties that (i) if P_1 satisfies a certain constraint C then P_2 also satisfies C, and (ii) the dependencies that appear in P_2 are *justified* by those in P_1, i.e., no spurious dependencies are introduced in P_2. Furthermore, the operator is closed with respect to composition, making further abstraction of abstract PROV documents possible. The operator is implemented as part of a user tool, described in a separate paper, that lets owners of sensitive provenance information control the abstraction by specifying an abstraction policy.

Keywords: Provenance, Provenance metadata, provenance abstraction

1. Introduction

The provenance of data is a form of structured metadata that records the processes involved in data production. In addition to containing references to data generation or transformation processes, a *provenance trace* typically includes input or intermediate data products as well as references to *agents*, that is the humans or software systems who were responsible to enact those processes.

*Corresponding Author
Email addresses: Paolo.Missier@newcastle.ac.uk (P. Missier), Jeremy.Bryans@coventry.ac.uk (J. Bryans), Carl.Gamble@newcastle.ac.uk (C. Gamble), Vasa.Curcin@kcl.ac.uk (V. Curcin)

In multi-party collaboration settings that involve data sharing, as well as in third party auditing of data and processes, there is a broad expectation that shipping the available provenance to collaborators, or more generally publishing it along with the data, may help data consumers, including auditors, form judgements regarding the reliability of the data itself.

Offering to disclose the provenance of data as evidential basis for establishing data quality and reliability is particularly important when data products are exchanged as part of transactions that involve parties with limited mutual trust. This is the case for instance of *dynamic coalitions* [6], ad hoc collaborative partnerships that are created to pursue a common goal, in scenarios such as multi-agency emergency/threat responses, as well as the exchange of intelligence information. Despite the need to share data of a possibly sensitive nature, these coalitions are characterized by a lack of established interaction protocols and by limited trust amongst the partners. This situation creates a tension between data providers and consumers, when it comes to negotiating the level of detail of the provenance that providers are prepared to offer to consumers. On the one hand, consumers will require as much provenance detail as possible, to use as a basis for establishing data credibility. Data providers, on the other hand, will be reticent to offer detailed provenance traces, because those may contain sensitive information regarding their own internal processes as well as any proprietary data used by those processes. In fact, the provenance of a data product will typically contain more sensitive information than then data product itself.

In this paper we propose to resolve such tension by introducing an operator to achieve selective disclosure of provenance information.

1.1. Motivating scenario: provenance of intelligence information

To appreciate how such tension may arise, consider a scenario where a public agency PA wants to buy intelligence reports, say about potential threats to the public, from an intelligence provider, IP. Under assumption of limited trust, PA will want to mitigate the risk of acting upon information provided by IP, which is potentially unreliable. At the same time, IP has a business incentive to supply PA with additional evidence that facilitates PA's risk assessment and thus increases the chance of a successful transaction.

The key assumption that motivates our work is that the provenance of each intelligence report is relevant in contributing, at least in part, the required evidence. A fictional but realistic example of provenance for an intelligence report such information is shown in Fig. 1. Provenance can be visually depicted as a digraph whose nodes represent either *entities* (ovals in the figure), i.e., data, documents, etc., *activities* (rectangles), which represent the execution of some process over a period of time, or *agents* (pentagons), which represent humans or computing systems. The edges represent various types of directed relationships, the most common being "activity a used entity e", "entity e was generated by activity a", "activity a was associated with agent ag" (i.e., ag was responsible for a), and more. It is also possible to annotate each of the nodes using properties, for instance to qualify the kind of data and activities involved in the process execution. We omit annotations from our examples for

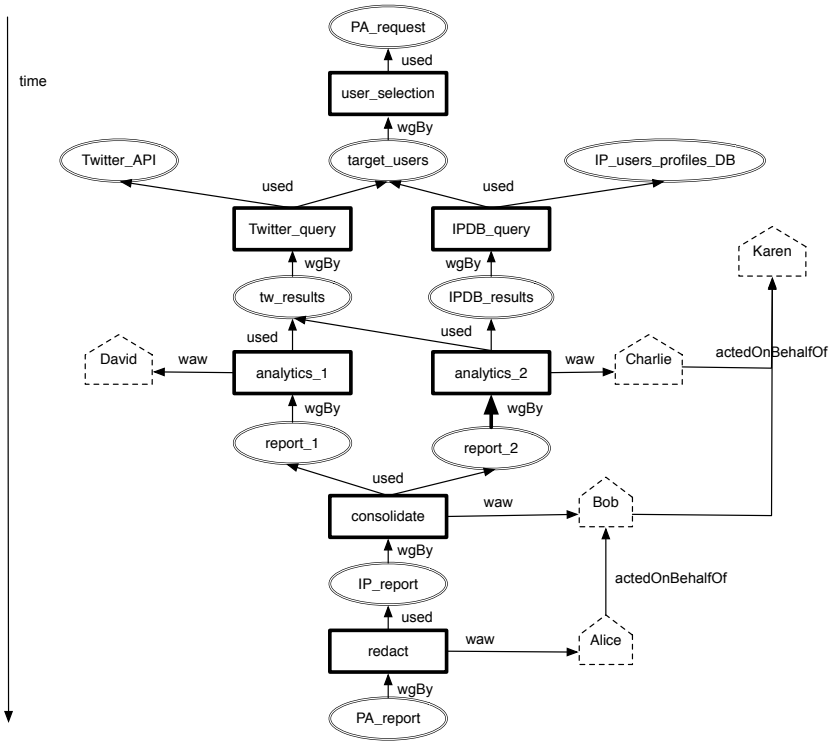

Figure 1: Example provenance graph depicting the generation of an intelligence report

readability, and because, within this work, they are not handled in any special way.

Formally, such a graph is a depiction of a PROV document, which in turn conforms to the W3C PROV data model [21], introduced in Sec. 3.1. We will use the graph representation of provenance throughout the paper, as it facilitates reasoning about the mechanisms for provenance abstraction, which are at the core of our work. The graph layout in Figs. 1, 2 and 3 is such that the process execution flows from top to bottom, i.e, the top entities represent initial inputs, while the outputs are at the bottom.

This provenance graph depicts a process of intelligence report generation, which is initiated by a request by PA. The process identifies target users from the request and acquires further information about those users, both on Twitter (`Twitter_query`) and from a proprietary database, `IP_users_profile_DB`. The results are fed to two analytics sub-processes, each of which generates a report. Note that `analytics_1` only uses Twitter data, while `analytics_2` also uses query results of the proprietary database. A `consolidate` step follows, which produces a master `IP_report`. This is checked, presumably to validate and possibly also to remove sensitive information, before the final `PA_report`

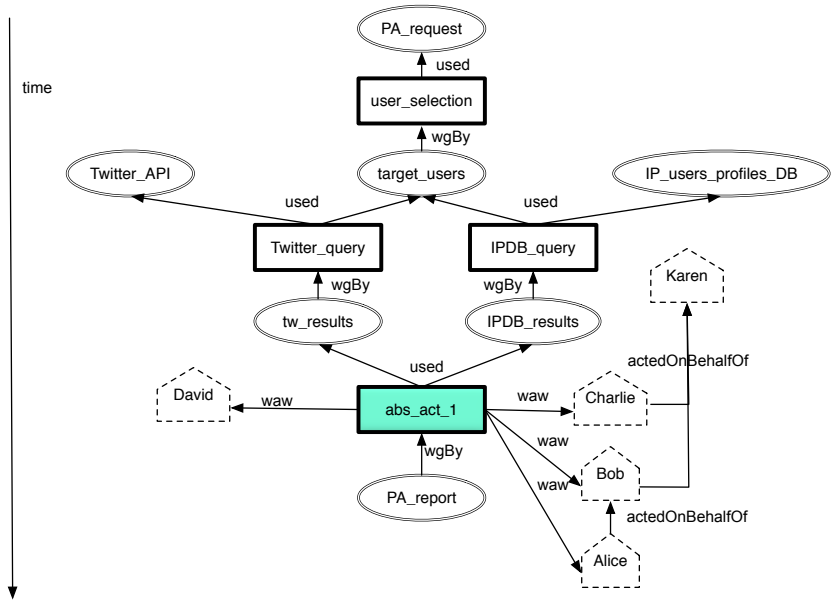

Figure 2: The result of abstracting out selected nodes `redact`, `analytics_1`, and `analytics_2` from the graph in Fig. 1.

is generated for the customer. Notice that various agents are specified as being responsible for some of the steps, along with their chain of responsibility, i.e., `Alice` acts as `Bob`'s delegate, who in turn reports to `Karen`, along with `Charlie`.

As we can see, the full-fledged provenance graph contains information about IP's internal business processes, including the use of a proprietary database, which IP may consider privileged. It is therefore realistic to imagine that IP may want to obfuscate some of those elements. Using our selective disclosure model, IP marks the sensitive elements, in this case the `redact` activity as well as the references to the two analytics processes. Note that doing so does not require any knowledge of the graph topology, rather only of the nodes (either activities, entities, or agents) that are to be abstracted out.

It is important to note that the PROV document represented in Fig. 1 is *valid*, in the sense that it satisfies all the constraints specified as part of the PROV standard [9]. As we will see in the rest of the paper, replacing the three selected nodes with a single abstract node (an activity in this case) while preserving the validity of the document, requires that all other nodes that lie on the directed paths that connect these nodes are also removed. In this example, the result of such abstraction operation is shown in Fig. 2.

By construction, this is a new valid PROV graph. It is therefore possible to further abstract out some of its nodes. For example, IP may also decide that mentioning the use of a proprietary data source is inappropriate. Nodes `IP_users_profiles_DB` and `IPDB_query` are therefore abstracted out. As these

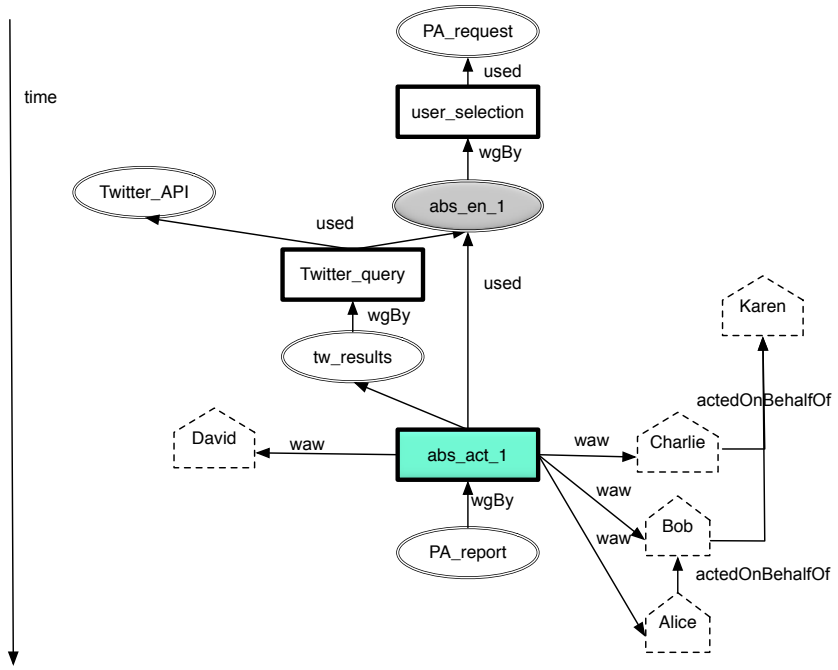

Figure 3: The result of further abstracting out `IP_users_profiles_DB` and `IPDB_query` after the first abstraction-by-gropuping step (Fig. 2).

are both entities, the new abstract node is also an entity, as shown in Fig. 3. Note that, to PA, while still informative, the report now appears as if it had been generated from its initial request using Twitter as a data source, and without reference to specific analytics algorithms.

The mechanisms by which the data and provenance owner selects the nodes to be abstracted are not discussed in this paper, however a policy-based model is described in detail in our previous work [17]. Briefly, the idea, inspired by the Bell-Lapadula model [2], is that the owner assigns a sensitivity value to each node, and nodes are selected to be abstracted out based on a specific recipient's clearance level. Thus, different recipients will potentially receive different abstract versions of the same graph. Note also that forcefully removing nodes that were not marked for abstraction has implications, too, as some of those non-sensitive nodes may have had evidential value that is now lost. To model this problem we associate a utility value to each node, and then compute the *residual utility* of the abstracted graph. The paper cited above [17] provides further details.

We observe that removal of information from a provenance graph could be achieved in a number of other ways. For example, one could simply remove the labels as well as the annotations from individual nodes and relationships, i.e., anonymize part of the graph. Doing so, however, does not hide any of the

structure of the process of data production. One could further remove nodes and relationships or indeed entire sub-graphs. The new graph will be disconnected, however, making it difficult to reconstruct the lineage of the end data product, that is, the sequence of data derivations from the initial inputs to the outcome of the process.

Instead, in our approach a sub-graph is replaced with a new abstract node, which is then "re-wired" to the remaining original graph. This has the effect of hiding parts of the process structure as it was represented in the original provenance, while maintaining connectivity. One can still query the lineage, but some of the provenance elements returned by the query will now be an abstraction of the actual data production process.

The main challenge addressed in this paper is to guarantee that abstraction produces PROV-compliant graphs, maintaining the interoperability guarantees provided from having standardized PROV and ensuring that the results can be consumed by standard PROV tools.

1.2. Contributions

In this paper we develop a model and algorithm for performing abstraction over PROV graphs, providing the theoretical underpinning to ensure that the abstraction process satisfies a number of properties. Our main contribution is the formal definition of a provenance abstraction operator that rewrites a PROV graph PG into a new graph PG', by mapping a set V_{gr} of nodes (for "vertex in a group") in PG to a new abstract node v_{new}, and then mapping each relationship involving elements of V_{gr} to a new relationship involving v_{new} in PG'. The set V_{gr} is chosen by the user of the abstraction operator as the set of nodes she wishes to obfuscate.

We prove several formal properties of PG': firstly, schema preservation: if PG is a PROV graph, that is, it conforms to the PROV data model [21], then PG' is also a PROV graph. Secondly, validity preservation: if PG is *valid*, that is, it satisfies all PROV constraints [21], then PG' is also valid. Finally, no spurious dependencies are introduced into PG': a relationship involving v_{new} is only created as a result of a mapping from an existing relationship involving elements of V_{gr}. Strictly, if a and e are not *directly* related in PG, we guarantee that they are not directly related in PG'.

Note that new indirect dependencies between two nodes in PG', manifested as new paths in the graph, may be introduced, however we show that these are always justified by the topology of the underlying graph PG.

Furthermore, by making the abstraction operator closed with respect to the set of valid PROV graphs, abstraction can be naturally composed, i.e., one can abstract PG' into some PG'' as we have shown in the earlier example. Finally, note also that PG' itself has also an associated provenance graph, that is, a record of the provenance abstraction process as it was applied to PG. PROV provides a syntactic facility to maintain the association between a provenance graph and its own provenance, namely using the "provenance of provenance" mechanism (i.e., bundles [21]).

2. Related Work

Work related to our research is broadly motivated either by the need to simplify provenance graphs to facilitate their understanding by humans, or to enforce access control over parts of the graph, or to summarise a collection of similar provenance graphs in order to reduce both space and query complexity.

2.1. Creating views over provenance

Work on provenance abstraction generally combines two elements, namely a technique or algorithm for graph editing, and a policy framework to drive the algorithm. As mentioned, in this paper we focus exclusively on the former, while the latter is described in a separate paper [18].

Work to create views over provenance graphs for the purpose of either reducing their complexity, and/or removing sensitive information, was arguably pioneered by the Zoom system [4]. In Zoom, the main assumption is that the graph is a trace that specifically represents the execution of a dataflow. This is a common occurrence in e-science, where workflows that follow the dataflow model are a popular high level programming paradigm. In this setting views over provenance are effectively a form of abstraction and are computed based on the user's indication of which workflow modules (tasks) are relevant, or perhaps based on which modules the user has access to. Thus, key to this approach is knowledge of the underlying workflow structure, which is used to specify the nodes in the graphs to be abstracted. This sets Zoom apart from our work, which instead investigates the properties of a grouping operator *independently of the origins of the trace to which it is applied.*

Also specific to workflow-generated provenance, and thus too narrow in scope for our purposes, is a strand of research that investigates the problem of preserving the privacy of functions used in workflows, when a large number of input/output pairs for those functions is revealed through the provenance traces of multiple workflow executions. This work on *module privacy* [12, 11, 10] is concerned with protecting the semantics of workflow modules. It applies anonymization techniques specifically to provenance graphs and is again centred around a workflow-specific form of provenance and is thus also peripheral to our interest.

Closer to our abstraction model, both in motivation and in its technical approach, is the ProPub system [13], which computes views over provenance graphs that are suitable for publication by meeting certain privacy requirements. In ProPub, users specify edit operations on a graph, such as anonymizing, abstracting, and hiding certain parts of it. The operations are specified as logic rules, and are interpreted natively by the Datalog-based prototype implementation. ProPub adopts an "apply–detect–repair" approach, whereby user rules are applied to the graph first, then consistency violations that may occur in the resulting new graph are detected, and a final set of edits are applied to the graph in order to repair such violations. In some cases, this causes nodes that the user wanted removed to be reintroduced, and it is not always possible to satisfy all rules. In contrast, our grouping involves more simply a set of nodes to

be abstracted (but note that anonymization is a particular case, when the group contains a single element). In return for this simplicity in the specification of the nodes to be grouped, our method always produces a valid abstract graph while ensuring that the nodes specified in the policy are removed.

Finally, *provenance redaction* [8] employs a graph grammar technique to edit provenance that is expressed using the Open Provenance Model [20] (a precursor to PROV), as well as a redaction policy language. Although the authors claim that the redaction operators ensure that specific relationships are preserved, this critical issue is not addressed formally in the paper, i.e., with reference to the OPM semantics. In contrast, the formal schema and set of constraints that come with PROV [21, 9] provide the necessary grounding for reasoning about the validity-preservation properties of the editing operations.

2.2. Provenance Access Control

Most of the work on protecting access to sensitive provenance includes policy models that extend traditional data access control frameworks (RBAC), with a distinction made between PBAC (Provenance-Based Access Control) and PAC (Provenance Access Control). PBAC is about policy to specify access rights to data objects based on their provenance. An example, from [22], is a rule of the form "only the student submitter can access the graded homework object". This rule can be enforced by looking for a dependency path in a provenance graph, whereby a given homework is attributed to a specific student (i.e., relation *IsAuthoredBy* in the Open Provenance Model). This assumes that the object's attribution is explicit in the provenance graph. It is less clear how such a rule would be evaluated when the provenance is incomplete with respect to such attribution dependency, however.

PAC, or how to enforce access control on parts of a provenance graph, is more directly relevant to our work. An analysis of some of the challenges associated with secure provenance exchange can be found in [5], where examples are presented that show how the provenance of data can be more sensitive than the data itself. Another position paper [15] describes the challenges associated with the exchange of provenance across multiple partners, in a setting where forgery of provenance by malicious users is a possibility, and where users may collude to reveal sensitive provenance to others. These are all common and complex security problems. Unfortunately, the paper stops short of providing any hints at technical solutions, and indeed it is not clear how these problems are specific to provenance, as opposed to data sharing in general.

A concrete specification of an access control system or provenance [7] consists of a XACML-based policy language, in which path queries are used to specify target elements of the graph, as well as an implementation architecture and a prototype.

2.3. Summarisation of provenance graphs

A loosely related strand of research in this area aims at summarising a collection of provenance graphs by constructing a "super-graph" that captures the

common features across a collection of similar graphs, such as those that are produced by repeating execution of a process with different inputs and parameters. This has been addressed with an aim to improve provenance queries [14], as well as to provide a compact but approximate representation of provenance at the possible cost of information loss [1]. This work is only peripherally relevant here, as our approach only operates on one graph at a time.

In [19] a mechanism is proposed to automatically construct aggregations from a single PROV graph. This relies on the concept of *provenance types*, which are fixed-length paths in the graph that occur more than once. The aggregation is defined as a mapping from provenance nodes to the provenance types, and there is a way to connect these types into a new PROV-like graph, by similarly mapping the graph edges to new weighted edges. The result is a new graph that is meant to capture the "essence" of a fine-grained set of provenance statements by observing regularities in the original graph. This is substantially different from our approach, namely (i) the choice of nodes to aggregate is driven by the discovery of provenance types, which is entirely driven by graph topology and not by a user choice, and (ii) there is no intent to generate *valid* PROV graphs, which is instead the main goal of our transformation. Thus, the approach is not suitable to support policy-driven (or other user-oriented) selective disclosure, and the aggregation operation produces a graph that may violate PROV constraints.

2.4. General graph anonymization

For completeness, we briefly mention more general techniques for graph editing, largely motivated by the need to preserve privacy in social network data. This body of work, which is not specific to provenance, extends the well-known data anynomization framework developed for relational data to graph data structures [23, 3, 16]. The main idea is to randomly remove arcs between two nodes and replace them with new ones. As arcs in PROV graphs represent relationships with a given semantics, this approach generally results in false dependencies being created in the edited graph, and is therefore not viable. The main value of this body of work in this setting, as summarised in [24], is to ensure that various forms of anonymization are provably robust to attacks from adversaries who can potentially leverage their partial information about fragments of the graph, to infer additional knowledge. In this paper we do not discuss the robustness of abstraction by grouping, indeed we do not consider any specific threats, and so the challenge of preventing the reconstruction of the abstracted fragments of provenance graphs is left for future work.

3. Background

3.1. Core PROV model

We now introduce the core elements of the PROV model, which forms the basis for the grouping operator. We maintain a dual view of provenance, both as a relational model (with binary relations) and as a graph model. Viewed

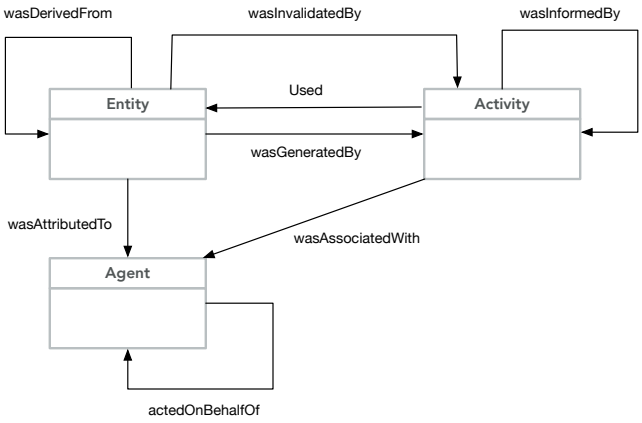

Figure 4: Core elements of the PROV model, adapted from [21]

as a relational model, PROV includes three types of elements: Entities (En), Activities (Act), and Agents (Ag), and several types of relations amongst them. In line with the description in [21] (sec. 2), PROV is defined by the following core relations, with common abbreviations in brackets.

$$\begin{aligned}
\mathit{Used}\ (\mathit{used}) &\subseteq \mathit{Act} \times \mathit{En} \\
\mathit{WasGeneratedBy}\ (\mathit{genBy}) &\subseteq \mathit{En} \times \mathit{Act} \\
\mathit{WasDerivedFrom}\ (\mathit{wasDerivedFrom}) &\subseteq \mathit{En} \times \mathit{En} \\
\mathit{WasInvalidatedBy}\ (\mathit{inval}) &\subseteq \mathit{En} \times \mathit{Act} \\
\mathit{WasAssociatedWith}\ (\mathit{waw}) &\subseteq \mathit{Act} \times \mathit{Ag} \\
\mathit{ActedOnBehalfOg}\ (\mathit{abo}) &\subseteq \mathit{Ag} \times \mathit{Ag} \\
\mathit{WasAttributedTo}\ (\mathit{wat}) &\subseteq \mathit{En} \times \mathit{Ag} \\
\mathit{WasInformedBy}\ (\mathit{wasInformedBy}) &\subseteq \mathit{Act} \times \mathit{Act}
\end{aligned}$$

These are summarized in Fig. 4. Initially, we are going to restrict ourselves to an even simpler model, consisting only of En, Act, and relations *used* and *genBy*. Agents and the relations that involve them are introduced in Sec. 6. Further extensions to the additional relations — *wasDerivedFrom* and *wasInformedBy* — are straightforward and are not considered in detail.

An instance of the model is a provenance document D, consisting of sets $en \in En$ and $act \in Act$ of symbols, and sets of relation instances $\{genBy(e, a) \mid e \in En, a \in Act\} \cup \{used(a, e) \mid e \in En, a \in Act\}$.

As these relations are binary, we view D as a digraph $G = (V, E)$, where $V = En \cup Act$, and each relation instance maps to a labelled directed edge. By convention, we orient these edges from right to left, to denote that the relation "points back to the past". Thus: $a \xleftarrow{genBy} e \in E$ iff $genBy(e, a) \in D$, and

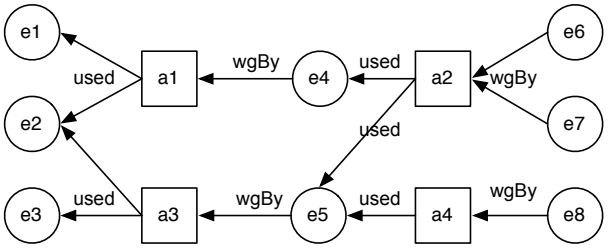

Figure 5: $PG_{gu/ea}$ provenance graph used as a running example to illustrate abstraction by grouping

$e \xleftarrow{used} a \in E$ iff $used(a, e) \in D$. We denote the label associated to edge (v_i, v_j) as $label(v_i, v_j)$.

Note that, by definition of the relations, G is a bipartite graph. We denote the set of all such graphs by $PG_{gu/ea}$, to indicate that they only contain *En* and *Act* nodes, and *genBy* and *used* edges. In Sec. 6 we are going to extend this set to include agents as well as additional relations. Fig. 5 portrays a simple $PG_{gu/ea}$ graph that we will be using as a running example.

3.2. Events in $PG_{gu/ea}$

Central to PROV is the notion that provenance is marked by events. A partial order is defined over events, so that it may or may not be possible to establish whether or not one event precedes another. Events occur instantaneously, and they mark the lifetime boundaries of Entities (generation, invalidation), Activities (start, end), and Agents (start, end), as well as some of the interactions amongst those elements. These include the generation and usage of an entity by an activity, attribution of an entity to an agent, and more. More specifically, the PROV-CONSTRAINTS document [9] defines the following types of events (quoted verbatim from Sec. 2.2):

- **An activity start event** is the instantaneous event that marks the instant an activity starts.
- **An activity end event** is the instantaneous event that marks the instant an activity ends.
- **An entity generation event** is the instantaneous event that marks the final instant of an entity's creation timespan, after which it is available for use. The entity did not exist before this event.
- **An entity usage event** is the instantaneous event that marks the first instant of an entity's consumption timespan by an activity. The described usage had not started before this instant, although the activity could potentially have used the same entity at a different time.

- **An entity invalidation event** is the instantaneous event that marks the initial instant of the destruction, invalidation, or cessation of an entity, after which the entity is no longer available for use. The entity no longer exists after this event.

We formally express events by introducing a set Ev of event symbols, with a pre-order[1] $\preceq \subset Ev \times Ev$, and the following set of partial functions that associate events to elements and relations in a provenance instance:

$$\begin{aligned} start &: Act \to Ev \\ end &: Act \to Ev \\ ev &: genBy \cup used \cup inval \to Ev \end{aligned}$$

As an example, in the graph of Fig. 5 the generation relation $genBy(e_4, a_1)$ has an associated generation event $ev(genBy(e_4, a_1))$, whilst a_1 has start and / or end events, written $start(a_1)$ and $end(a_1)$, respectively. Similarly, usage of e_4 by a_2 is marked by event $ev(used(a_2, e_4))$. Finally, if e_4 had been invalidated by some a (not in the figure), this would be represented by an invalidation event $ev(inval(e_4, a))$.

Temporal constraints involving events, and expressed by means of their pre-order relation $\preceq$, play a key role in the definition of *valid* provenance instances, as described next.

3.3. Constraints and valid $PG_{gu/ea}$ graphs

Validity of a PROV document is defined in terms of a set of constraints, as stated in the PROV-CONSTRAINTS document [9]. For instance, Constraint 55 states that the Entities and Activities are disjoint: $En \cap Act = \emptyset$. Thus, a document D that contains both statements (1) $a_1\, used\, e_1$ and (2) $e_1\, used\, a_1$ cannot be valid, because by definition(1) entails $e_1 \in En$, $a_1 \in Act$, while (2) entails $a_1 \in En$, $e_1 \in Act$, violating the constraint. Note that disjointness constraint 55 entails that $PG_{gu/ea}$ graphs are bipartite.

In this paper we are mainly concerned with temporal constraints, which apply to $PG_{gu/ea}$ instances and determine the admissible partial ordering of the event types introduced in the previous section. With reference to a graph PG, these are (including constraint (55) above, and using the original numbering in [9])

- **C1: entity-activity-disjoint (Constraint 55):**

  $$En \cap Act = \emptyset$$

- **C2: generation-generation-ordering (Constraint 39):** If an entity is generated by more than one activity, then the generation events must all be simultaneous.

[1] Recall that a pre-order is a binary relation with reflexivity and transitivity, but no symmetry or anti-symmetry.

Let $gen_1 = ev(genBy(e, a_1))$, $gen_2 = ev(genBy(e, a_2)) \in PG$. Then

$$gen_1 \preceq gen_2, \quad gen_2 \preceq gen_1$$

must hold.

- **C3: generation-precedes-usage(Constraint 37):** A generation event for an entity must precede any usage event for that entity. For any $a \in Act$ such that $used(a, e) \in PG$,

  $$ev(genBy(e, a)) \preceq ev(used(a, e))$$

  must hold.

- **C4: generation-precedes-invalidation (Constraint 36):** The generation event (or, more accurately, the set of simultaneous generation events) for an entity must precede the invalidation event.

  For any $a, a' \in Act$ such that $genBy(e, a))$, $inval(e, a') \in PG$:

  $$ev(genBy(e, a)) \preceq ev(inval(e, a'))$$

- **C5: usage-precedes-invalidation (Constraint 38):** Any usage event for an entity must precede the invalidation event. For any $a, a' \in Act$ such that $used(a, e))$, $inval(e, a') \in PG$:

  $$ev(used(a, e)) \preceq ev(inval(e, a'))$$

- **C6: usage-within-activity (Constraint 33):** Any usage of e by a cannot precede the start of a and must precede the end of a. For any $e \in En$, $a \in Act$ such that $used(a, e) \in PG$:

  $$start(a) \preceq ev(used(a, e)) \preceq end(a)$$

- **C7: generation-within-activity (Constraint 34):** The generation of e by a cannot precede the start of a and must precede the end of a. Let $genBy(e, a) \in PG$:

  $$start(a) \preceq ev(genBy(e, a)) \preceq ev(a)$$

- **C8: invalidation-invalidation-ordering (Constraint 40):** If an entity is invalidated by more than one activity, the events must all be simultaneous.

  $$\begin{aligned} &\text{if } inval(e, a_1), inval(e, a_2) \in PG \\ &\text{then } ev(inval(e, a_1)) = ev(inval(e, a_2)) \end{aligned}$$

Additional relevant constraints state that multiple start (resp. end, invalidation) events must all be simultaneous, and that the start event of an activity must precede the end event for that activity.

Definition 1 (Validity). *A graph $G \in PG_{gu/ea}$ is valid iff it satisfies constraints C1-C8.*

In the next section, we present the main result of this work, the *Group* operator. We will return to the constraints above in Section 5, where we demonstrate that application of the group operator maintains the validity of the above constraints, in a manner which we will clarify further in Section 5.

4. Grouping Provenance graph nodes

As mentioned in Sec. 1.2, our goal is to define graph editing operators that selectively remove information from a graph $G \in PG_{gu/ea}$, yielding a new graph $G' \in PG_{gu/ea}$. The first of these, namely the removal of labels or annotations associated with a node or an edge, is straightforward. Regarding the removal of a node, we note that simply reconnecting the remaining nodes generally may lead to an invalid graph. A simple example is a graph defined by: $\{used(a, e_1), genBy(e_2, a)\}$ where activity a is removed. This results simply in two disconnected nodes e_1, e_2, because no relationship can be inferred between them from the original graph.

Rather than delving into the possible consequences of such node and edge elision, we focus exclusively on the *Group* graph transformation operator as the prime way to achieve abstraction over provenance graphs. *Group* takes a graph $G = (V, E) \in PG_{gu/ea}$ and a subset $V_{gr} \subset V$ of its nodes that the user wishes to obfuscate and produces a modified graph $G' \in PG_{gu/ea}$. The nodes in V_{gr} are "grouped" together and replaced by a new single node.

$$Group : PG_{gu/ea} \times \mathbb{P}(V) \to PG_{gu/ea}$$

As the operator is closed under composition, further abstraction can be achieved by repeated grouping, either on multiple disjoint sets V_{gr}, or on sets that include abstract nodes (abstraction of abstraction).

To get a quick intuition of the problems faced in the definition of the grouping operator, consider the transformation in Fig. 6, where nodes $V_{gr} = \{e_1, e_3, e_4, e_5\}$ are simply replaced with new node e' in the example graph of Fig. 5, and all edges in and out of nodes in V_{gr} are just "rewired" in and out of e'.

This illustrative example points out two problems. Firstly, it introduces two cycles: $e' \leftrightarrow a_1$ and $e' \leftrightarrow a_3$. Furthermore, the two edges $e' \leftarrow a_1$ and $e' \leftarrow a_3$ cannot be of type *genBy*, while at the same time one cannot arbitrarily introduce *used* relations, which would be false dependencies. Thus, the resulting graph is not a valid PROV graph. Note that the former of these problems had been already pointed out in the description of the ProPub system [13], mentioned earlier.

4.1. Closure and homogeneous grouping

The example suggests that the issue is caused by nodes a_1 and a_3, which both lie on the paths between two of the nodes in V_{gr}. Intuitively, set V_{gr} is

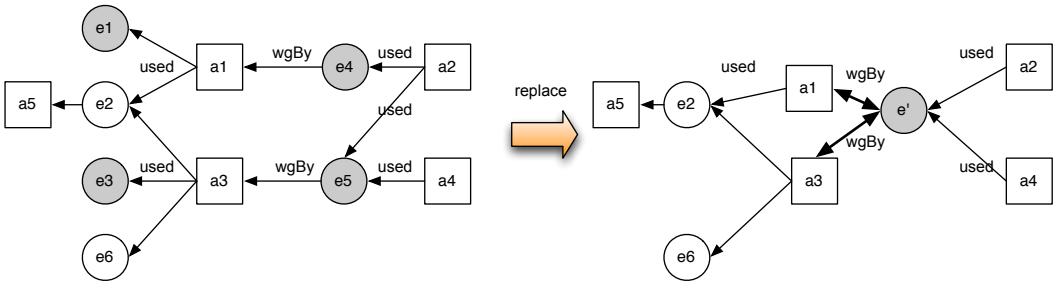

Figure 6: Example: the careless abstraction of a set of nodes may lead to a non-$PG_{gu/ea}$ graph.

not "convex", that is, there are paths in G that lead out of V_{gr} and then back in again. This observation suggests the introduction of a preliminary closure operation *pclos*(), which ensures acyclicity by capturing and including these paths. It is defined as follows.

Definition 2 (Path Closure). *Let $G = (V, E) \in PG_{gu/ea}$ be a provenance graph, and let $V_{gr} \subset V$. For each pair $v_i, v_j \in V_{gr}$ such that there one or more directed paths $v_i \rightsquigarrow v_j$ in G, let $V_{ij} \subset V$ be the set of all nodes in all paths $v_i \rightsquigarrow v_j$. The Path Closure of V_{gr} in G is*

$$pclos(V_{gr}, G) = \bigcup_{v_i, v_j \in V_{gr}} V_{ij}$$

We assume, for the moment, that the subgraph given by $pclos(V_{gr}, G)$ is a subgraph of G with the property that each source is connected to at least one of the sinks. For now, we assume there is only one such subgraph that contains all the nodes in V_{gr}.

Fig. 7 shows a continuation of the previous example. This time the replacement is performed on $pclos(\{e_1, e_3, e_4, e_5\}, G) = \{e_1, e_3, e_4, e_5, a_1, a_3\}$, (in Fig. 7(b)), resulting in graph in Fig. 7(c). However, while this solves the cycle problem, the graph is no longer bipartite, because the new edges $e' \rightarrow e_2$ and $e' \rightarrow e_6$ connect nodes of the same type. In this example, we can construct a new group of nodes, $\{e', e_2, e_6\}$, on the graph that results from the first replacement, and replace it with a new node e''. The resulting graph Fig. 7(d) is a valid $PG_{gu/ea}$ graph.

The same result can be obtained by first *extending* the closure in Fig. 7(b) to include e-nodes e_2, e_6, and then replacing the resulting set with e'' (this is indicated by the "extend and replace" arrow from Fig. 7(b) to Fig. 7(d) as shown). The *extend* operator will have the role of "gathering up" certain nodes into the set to be replaced, and the purpose of *extend* is to ensure that the eventual node replacement preserves the type-consistency of graph. We require all the set boundary nodes (nodes connected to nodes not in the set) to be of the same type, so in Fig. 7(b) we must include nodes e_2 and e_6.

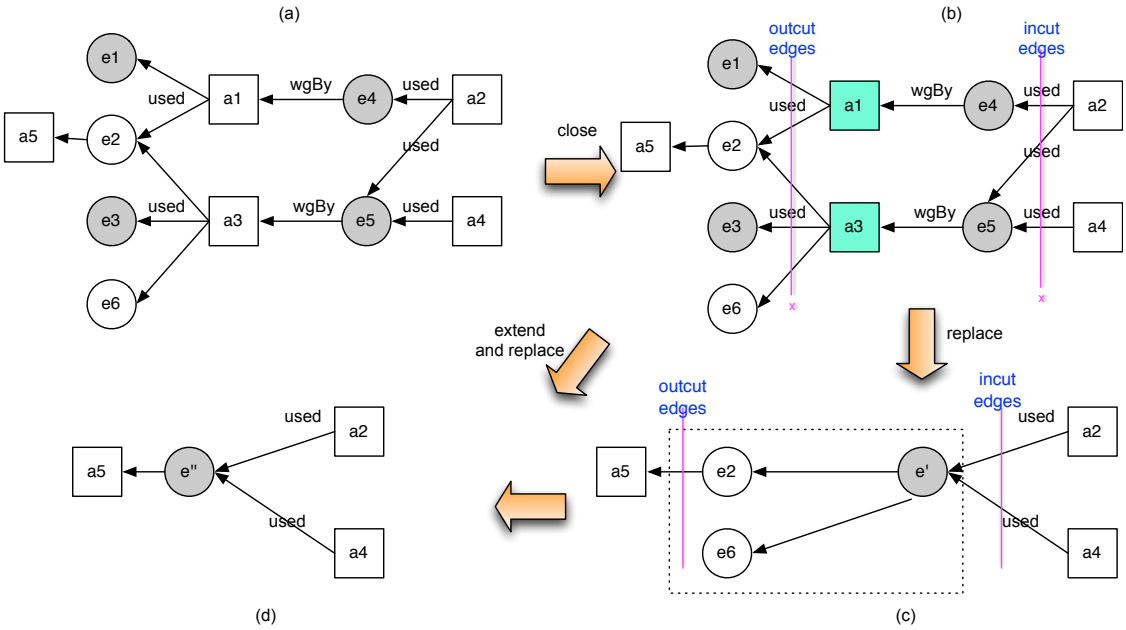

Figure 7: Path closure and replacement with extension on a set of entity nodes.

Following this approach, we are going to define grouping as a composition of three functions: *closure*, defined above, *extension*, and *replacement*, as follows.

The *extension* of a set $V_{gr} \subset V$ relative to type $t \in \{En, Act\}$ is V_{gr} augmented with all its adjacent nodes, in either direction, of type t. Formally:

Definition 3 (*extend*)**.** *Let $G = (V, E) \in PG_{gu/ea}$, $t \in \{En, Act\}$. v_s and v_d are the source and destination nodes of a relationship.*

$$\begin{aligned} &extend(V_{gr}, G, t) = \\ &\quad V_{gr} \cup \\ &\quad \{v_d \mid (v_d \leftarrow v_s) \in E \wedge v_s \in V_{gr} \wedge v_d \notin V_{gr} \wedge type(v_d) = t\} \cup \\ &\quad \{v_s \mid (v_d \leftarrow v_s) \in E \wedge v_s \notin V_{gr} \wedge v_d \in V_{gr} \wedge type(v_s) = t\} \end{aligned}$$

In our example, $extend(\{e_1, e_3, e_4, e_5, a_1, a_3\}, G, En) = \{e_1, e_3, e_4, e_5, a_1, a_3, e_2, e_6\}$. Note that all boundary nodes in $extend(V_{gr}, G, t)$ are of type t by construction.

Next, we replace the collected nodes with a new abstract node. Let $V^* \subset V$ be obtained using *pclos*() then *extend*(), as outlined above, and let v_{new} be a new node that does not appear in V. Function *replace*() replaces V^* with v_{new} in V, and connects v_{new} to the rest of the graph. To aid us in the definition, we begin by defining the *outcut*, *incut* and the *internal* nodes of V^*.

Let $\vartheta_{out}(V^*)$ denote the *outcut* of G associated with V^*, defined as:

$$\vartheta_{out}(V^*) = \{(v_d \leftarrow v_s) \mid v_s \in V^*, v_d \in V \setminus V^*\}$$

This is the set of arcs of G pointing out of V^*. Symmetrically, let $\vartheta_{in}(V^*)$ denote the *incut* of G associated with V^*, i.e., the set of arcs of G leading into V^*:

$$\vartheta_{in}(V^*) = \{(v_d \leftarrow v_s) \mid v_d \in V^*, v_s \in V \setminus V^*\}$$

Finally, let $\vartheta_{int}(V^*)$ denote internal arcs, that connect two both nodes inside V^*:

$$\vartheta_{int}(V^*) = \{(v_s \leftarrow v_d) \mid v_s, v_d \in V^*\}$$

Function *replace*() replaces each arc $(v_d \leftarrow v_s) \in \vartheta_{out}(V^*)$ with a new arc $(v_d \leftarrow v_{new})$ of the same type, and replaces each arc $(v_d \leftarrow v_s) \in \vartheta_{in}(V^*)$ with a new arc $(v_{new} \leftarrow v_d)$ of the same type. Arcs in $\vartheta_{int}(V^*)$ simply disappear along with the nodes in V^*.

The definitions of $\vartheta'_{out}(V^*)$ and $\vartheta'_{in}(V^*)$ below define the final part of the "rewiring" carried out by *replace*().

Definition 4. *Let $ty \in \{used, genBy\}$. Then:*

$$\vartheta'_{out}(V^*) = \{(v \xleftarrow{ty} v_{new}) \mid v \xleftarrow{ty} v' \in \vartheta_{out}(V^*)\}$$
$$\vartheta'_{in}(V^*) = \{(v_{new} \xleftarrow{ty} v) \mid v' \xleftarrow{ty} v \in \vartheta_{in}(V^*)\}$$

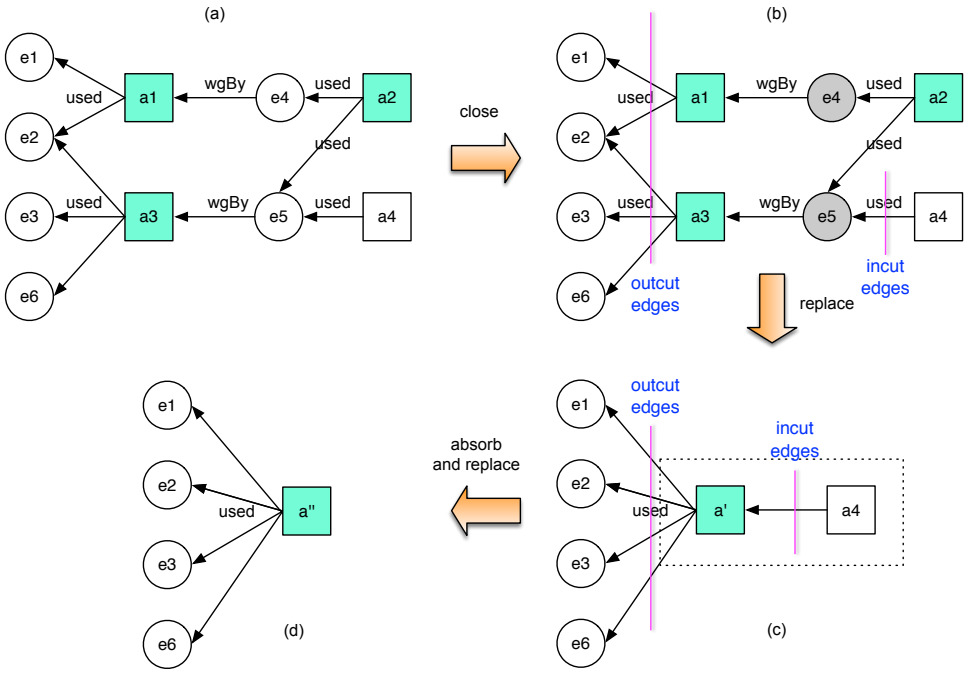

Figure 8: Grouping on a set of Activity nodes

And the full definition of *replace*() is

Definition 5 (replace).

$$replace(V^*, v_{new}, G) = (V', E'), \ \textit{where:}$$

$$\begin{aligned} V' &= V \setminus V^* \cup \{v_{new}\} \\ E' &= E \setminus (\vartheta_{out}(V^*) \cup \vartheta_{in}(V^*) \cup \vartheta_{int}(V^*)) \\ &\quad \cup \vartheta'_{out}(V^*) \cup \vartheta'_{in}(V^*) \end{aligned}$$

It is easy to verify that the resulting graph is type-correct. All boundary nodes in V^* are of the same type $t \in \{En, Act\}$, as noted above, and v_{new} is of type t by construction. Since the arcs have the same type as those they replace, it follows that *replace*() preserves type correctness.

We can now provide an initial definition of our *Group*() operator, under the simplifying assumption that all nodes in V_{gr} are of the same type before

closure. We denote this type by $type(V_{gr})$ (with a slight abuse of notation). Definitions 7 and 8 in Section 4.2 remove this assumption.

Fig. 8 shows a progression similar to that of Fig. 7, but this time $type(v) = Act$ for each $v \in V_{gr} = \{a_1, a_2, a_3\}$, and V_{gr} is replaced by another activity node, a'. Under assumption of type homogeneity, the grouping operator is a functional composition of *pclos*(), *extend*(), and *replace*() functions, defined as follows.

Definition 6 (Homogeneous Grouping). *Let $G = (V, E) \in PG_{gu/ea}$, $V_{gr} \in V$ be a type-homogeneous set, and let v_{new} be a new node with $type(v_{new}) = type(V_{gr})$.*

$$\begin{aligned} Group_{hom}(G, V_{gr}, v_{new}) = \\ & replace(\\ & extend(\\ & pclos(V_{gr}, G), V, type(V_{gr})), v_{new}, G) \end{aligned}$$

As an illustration, in our running example in Figure 7 we have:

$$\begin{aligned} V_{gr} &= \{e_1, e_3, e_4, e_5\} \\ V_{cl} &= pclos(V_{gr}, G) = \{e_1, e_3, e_4, e_5, a_1, a_3\} \\ V^* &= extend(V_{cl}, En) = V_{cl} \cup \{e_2, e_6\} \\ Group_e(G, V_{gr}, v_{new}) &= replace(V^*, v_{new}, G) \\ &= (\{a_1, a_2, a_3, e''\}, \{(a_2, e''), (a_4, e''), (e'', a_5)\}) \end{aligned}$$

4.2. Generalization to e-grouping and a-grouping

So far we have described the grouping operator in terms of the component functional parts. Up to this point we have been operating under the assumption made in Definition 2: that there is only one subgraph generated by $pclos(V_{gr}, G)$. In the case where we have two or more subgraphs, the *extend*() operator and the *replace*() operator would iterate over the set of subgraphs produced, and be applied to each subgraph separately.

Additional care must be taken if we allow V_{gr} to include nodes of mixed types. The type of the replacement node must now be specified, as it is no longer implied from the type of the nodes in V_{gr}. Indeed, the choice of such type leads to different abstracted graphs. Thus, we will now refer to grouping as *t-grouping*, where $t \in \{En, Act\}$, i.e., *e-grouping* or *a-grouping*. Fig. 9(a-1, a-2) illustrates the application of the $Group_{hom}$ operator (Def. 6), assuming *a-grouping* and $V_{gr} = \{e_4, a_2\}$. Because the boundary nodes for the set to be replaced must be of type activity, the extension incorporates activity node a_1.

Consider now the case of *e-grouping* in Fig. 9(e-1, e-2). Now the extension leads to $V_{cl} = V_{gr} \cup \{e_5\}$, which in turn leads to the pattern shown in Fig. 9(e-2), involving two generation events for the new entity e_N.

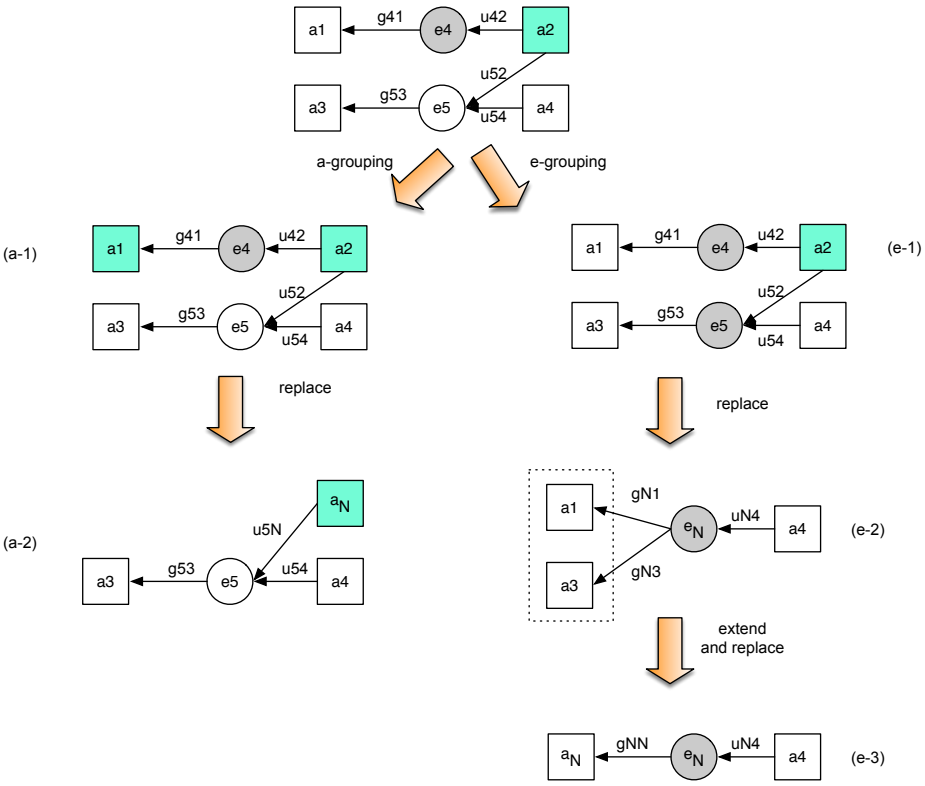

Figure 9: e-grouping and a-grouping on mixed type nodes

Although this is a valid pattern, the two generation events must be simultaneous (this is one of the temporal constraints defined in [9]):

$$ev(genBy(e_N, a_1)) \preceq ev(genBy(e_N, a_3)) \quad \wedge$$
$$ev(genBy(e_N, a_3)) \preceq ev(genBy(e_N, a_1))$$

The intuitive interpretation for this pattern is that each of the two activities generated one entity in the group represented by e_N, and that the abstraction makes these two events indistinguishable. Formally, nothing further needs to be done to the graph. We will explore the implications of the event ordering rules further in Section 5. However note that one can restore, if desired, the more natural pattern whereby one single generation event is recorded for e_N. This is achieved simply by propagating the grouping to the set of generating activities. In the example, this leads to the graph in Fig. 9(e-3).

We now formalize these considerations by introducing two definitions for *Group*. The first, which we call **t-grouping**, is agnostic of multiple generation patterns, while the second (**strict t-grouping**) applies propagation to ensure that the graph is free from multiple generation patterns.

Definition 7 (t-Grouping). *Let* $G = (V, E) \in PG_{gu/ea}$, $V_{gr} \in V$, $t \in \{En, Act\}$, *and let* v_{new} *be a new node with* $type(v_{new}) = t$. *Then:*

$$Group(G, V_{gr}, v_{new}, t) = \\ replace(extend(pclos(V_{gr}, G), V, t), v_{new}, G)$$

Note that the assumption that boundary nodes in the closure are homogeneous and are replaced by a node of the same type t, which is necessary for *replace* to perform correctly, still holds in this case.

Definition 8 (Strict t-Grouping). *Given* $G = (V, E) \in PG_{gu/ea}$, $V_{gr} \in V$, $t \in \{En, Act\}$, *and a new node* v_{new} *with* $type(v_{new}) = t$, *let*

$$G' = (V', E') = Group(G, V_{gr}, v_{new}, t).$$

Let $V_{gen} = \{a \in V' \mid a \xleftarrow{genBy} v_{new} \in E'\}$ *be the set of activity nodes that generate* v_{new} *according to* G', *and let* a_{new} *be a new activity node. Then:*

$$Group_{str}(G, V_{gr}, v_{new}, t) = \begin{cases} G' & if \mid V_{gen} \mid \leq 1 \\ replace(V_{gen}, a_{new}, G') & otherwise \end{cases}$$

It is straightforward to show that strict t-grouping reduces to normal grouping if the grouping is a homogeneous a-grouping:

$$\text{if } type(t) = Act \\ \text{then } Group_{str}(G, V_{gr}, t) = Group(G, V_{gr}, t).$$

4.3. Complexity of grouping operators

The grouping operator ensures that the validity of a PROV graph is preserved, that is, the abstracted graph that results from a valid PROV graph does not violate PROV constraints. Such guarantee, however, comes at a cost which is defined by the complexity of the *pclos*() and *extend*() operators. Here we analyse their worst-case complexity, and then argue that much better performance is expected in practical cases. Firstly, observe that the closure operator can be reduced to a special case of the node reachability problem for acyclic digraphs. Specifically, given two nodes $v_1, v_2 \in V_{gr}$ such that v_2 is reachable from v_2, we need to collect all nodes along *all* paths that connect v_1 to v_2. This can be accomplished by enumerating all nodes x that are reachable from each $v \in V_{gr}$ while keeping track of the corresponding paths. Whenever $x \in V_{gr}$, we collect all nodes along the recorded path from v to x. It is easy to see that the worst-case scenario occurs when the nodes in V_{gr} are located at the two ends of the graph, i.e., they are either source or sink nodes. In such case, the reachability algorithm needs to visit *all* nodes V and *all* edges E in the graph,

and it must additionally keep track of all edges it traverses. Using a simple BFS approach, we can solve the reachability problem in $O(\mid V \mid + \mid E \mid)$ steps, which in the worst-case is $O(\mid V \mid^2)$, with $O(\mid E \mid)$ space complexity for recording all edges. Note that the many algorithms that exist to address the problem aim to strike a balance between the cost of pre-processing the graph in order to efficiently answer multiple reachability queries, and the complexity of each individual query. In our case, however, we can dispense with pre-processing as we expect the closure over V_{gr} to be computed only once.[2] An experimental evaluation of the actual cost of computing closures in practice is beyond the scope of this paper, which is focused on the theoretical underpinnnings of the abstraction operations. However, two factors suggest that the practical complexity will be considerably less than the worst-case. Firstly, we can stop the graph traversal as soon as we have visited all V_{gr} nodes. Unless one of those is a sink node, this results in pruning part of the graph. Note also that in this case the abstraction will consist of one single abstract node that represents the entire graph, because the closure will include all nodes, which is unlikely to be a desired outcome. And secondly, in a provenance graph not all edges are allowed, in fact the graph is bipartite with respect of the nodes types (entities and activities), and furthermore, there is at most one generating activity per entity. These factors greatly reduce the expected number of edges to much less than the theoretical max $\mid V \mid^2$. Regarding the *extend*() operator, note that this requires all nodes in $pclos(V_{gr}, G)$ to be visited in order to check their type and possibly extend the closure to their immediate successors. This is a linear problem in the worst case, namely when the closure contains all nodes in the graph (in practice, we expect the closure to be much smalller than the graph).

4.4. Summary

In this section we have presented operators that abstract information in a provenance graph. We first presented *homogeneous grouping* (in Section 4.1), in which the user selects a set of nodes of the same type, and for which the new, abstract node retains that type. Section 4.2 extended this work to allow the user to select any nodes, at the expense of having to chose the type of the final, abstract, node.

It is clear that, in order to meet our initial requirement of maintaining type-correctness of the abstracted graph, in general more nodes than just the original ones selected will have to hidden. This has implications for the use of this operator, especially given that hidden information may be a critical part of the graph. We address this problem in [18] where we present a simple policy model and language for controlling abstraction, in the context where provenance owners want to control the disclosure of their provenance graphs. There, the owner defines a policy which results in a sensitivity value being associated with

[2]Note however that, when consecutive abstraction rounds are envisioned, i.e., abstraction over abstracted graphs, pre-processing may be appropriate, but that needs to be balanced against the cost of updating the data structures after each abstraction round.

nodes, which gives us a means of evaluating the "damage" to a graph caused by the abstraction operator. In [18] we do this by means of defining a property *utility* as a counterpart to sensitivity. It is used to indicate the interest of the provenance owner in ensuring that a node be retained as part of the graph, as it represents important evidence which is not sensitive. The utility values associated to different nodes are used to quantify any loss of utility as a result of the application of *group* though a measure of *residual utility*. If we write the utility of a node n as $u(n)$, and $V_{ret} = V / V_{gr}$ is the set of nodes *not* intended to be hidden, and $V'_{ret} \subset V_{ret}$ the nodes which were in fact retained after grouping, the residual utility is simply

$$RU_V = \frac{\Sigma_{n \in V'_{ret}} u(n)}{\Sigma_{n \in V_{ret}} u(n)}$$

which is a measure of the proportion of the graph utility not selected by the grouping operator.

Another approach is to break the initial set chosen by the user into multiple smaller sets, thereby reducing the amount of extra information hidden. This could be taken to its logical limit by abstracting individual nodes, but this would have to be balanced against the increasing revelation of the structure of the graph. The significance of this trade off would have to be considered in a particular context, and a full investigation is beyond the scope of this paper.

5. Abstraction over events

We now evaluate the consistency of the abstractions we propose on the ordering constraints C2-C7. The ordering constraints C2-C7 in Section 3.3 define the space of possible temporal interpretations for a $PG_{gu/ea}$ graph. This is the set of all linear orderings of the events that are consistent with the constraints. After grouping over G, new events in G' are associated with the new grouping node. For a-grouping, these are $start(a_{new})$, $end(a_{new})$, as well as $ev(used(e, a_{new}))$, $ev(genBy(a_{new}, e))$ for all e that are generated by or used by a_{new}. For e-grouping, the new events are just $ev(genBy(e_{new}, a))$ and $ev(used(e_{new}, a))$ for any a that has generated (resp. used) e_{new}.

Since $G' \in PG_{gu/ea}$, constraints C2-C7 apply. These define the validity of G'. In this section we explore the relationship between the temporal interpretations in G and those in G'. We do this by extending the notion of abstractions over elements of G, to that of abstraction over *events* of G. Specifically, in addition to introducing new abstract entities and activities, we are now going to introduce new abstract events, as well. A set of temporal interpretations for G' can then be expressed using these new abstract events. The legal interpretations amongst these are those that satisfy constraints C2-C7 on G'.

In this section we define such abstract events in terms of the events in G, in such a way that for each legal temporal interpretation in G', expressed using the abstract events, there is a corresponding legal temporal interpretation in G. We are going to consider e-grouping and a-grouping in turn.

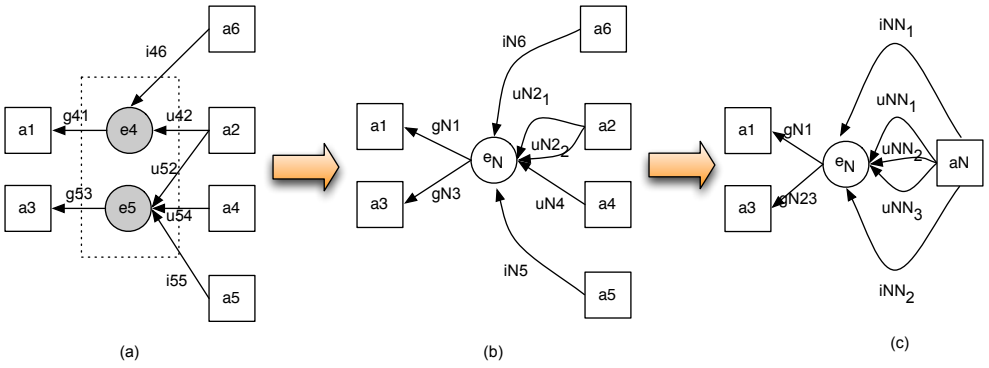

Figure 10: Abstraction over a document content, and associated abstracted events

5.1. Abstract events

Consider the scenario in Fig. 10(a), where two sections of a document are independently generated by two editing activities, and then they are independently used by four more activities. This scenario is a slight extension of the abstract pattern of Fig. 9 (which we have used to illustrate mixed type grouping), where the document sections are e_4, e_5. The e-grouping set $V_{gr} = \{e_4, e_5\}$ represents the whole document. Let G' be the result of (non-strict) e-grouping, as depicted in Fig. 10(b), where the abstract generation and usage events are given new names, namely g_{Ni} as a shorthand for $genBy(e_N, a_i)$, and u_{Ni_j} for each usage j of the form $used(a_i, e_N)$. It is easy to see that, taken in isolation, both graphs are valid: one can define the start, end, generation and usage events so that the ordering constraints are satisfied in G'.

Our goal is to go one step further, and define the new abstract events in terms of the events in G, in such a way that for each temporal interpretation that satisfies C2-C7 in G', there is at least one valid interpretation in G. We find that such a mapping of the linear orderings is possible, but that in some circumstances the range of possible orderings for events in G' is restricted.

Note: we have been using symbols like g_{41} in the figure to indicate relationships like $wgby(e_4, a_1)$. With slight abuse of notation, but in the interest of simplicity, in the following we are going to use g_{41} to also denote $ev(genBy(e_4, a_1))$ when it is clear from the context that we refer to the event rather than to the relationship itself.

The following questions provide a useful starting point for reasoning about events. Firstly, given the generation events g_{41}, g_{53} of each of the document sections, when was the entire document e_N generated? When did a_2, a_4 use the document? Secondly, suppose after the first grouping one performs two additional a-groupings, first with $V_{gr} = \{a_1, a_3\}$, and then with $V_{gr} = \{a_2, a_4, a_5, a_6\}$. This results in the abstraction depicted in Fig. 10(c), which reads simply "(abstract) document e_N was used by (abstract) activity a_N". In this abstraction,

what happens to the original generation and usage events?

Initial help in answering these questions comes from the PROV-DM recommendation document [21], namely:

- **Generation** *is the completion of production of a new entity by an activity* (Sec. 5.1.3)
- **Usage** *is the beginning of utilizing an entity by an activity* (Sec. 5.1.4)
- **Invalidation** *is the start of the destruction, cessation, or expiry of an existing entity by an activity. The entity is no longer available for use (or further invalidation) after invalidation. Any generation or usage of an entity precedes its invalidation.* (Sec. 5.1.8)

Let us consider these definitions in the context of our example. Firstly, the generation of the whole document is only complete upon generation of the last section. Thus, each of the generation events of e_N, denoted g_{N1} and g_{N3} in Fig. 10(b), cannot precede g_{41}, g_{53}. This can be written as the ordering constraints:

$$max\{g_{41}, g_{53}\} \leq g_{N1} \tag{1}$$
$$max\{g_{41}, g_{53}\} \leq g_{N3} \tag{2}$$

Furthermore, we know from constraint C2 that g_{N1} and g_{N3} must be simultaneous: $g_{N1} = g_{N3}$.

Secondly, symmetrically to generation, usage of the document $(u_{N2_1}, u_{N2_2}, u_{N4})$ in Fig. 10(b) begins with the earliest usage by any of the consuming activities:

$$u_{N2_1} \leq u_{42} \tag{3}$$
$$u_{N2_2} \leq u_{52} \tag{4}$$
$$u_{N4} \leq u_{54} \tag{5}$$

Finally, from the definition above, the rule for invalidation follows the same pattern as usage:

$$i_{N6} \leq i_{46} \tag{6}$$
$$i_{N5} \leq i_{55} \tag{7}$$

Furthermore, C3 requires each generation to precede each usage:

$$g_{N1} = g_{N13} \leq u_{N2_1} \tag{8}$$
$$g_{N1} = g_{N13} \leq u_{N2_2} \tag{9}$$
$$g_{N1} = g_{N13} \leq u_{N4} \tag{10}$$

and C4, C5 require both generation and usage to precede invalidation:

$$g_{N1} = g_{N3} \leq i_{N6} \quad g_{N1} = g_{N3} \leq i_{N5} \tag{11}$$
$$u_{N2_1} \leq i_{N6} \quad u_{N2_1} \leq i_{N5} \tag{12}$$
$$u_{N2_2} \leq i_{N6} \quad u_{N2_2} \leq i_{N5} \tag{13}$$

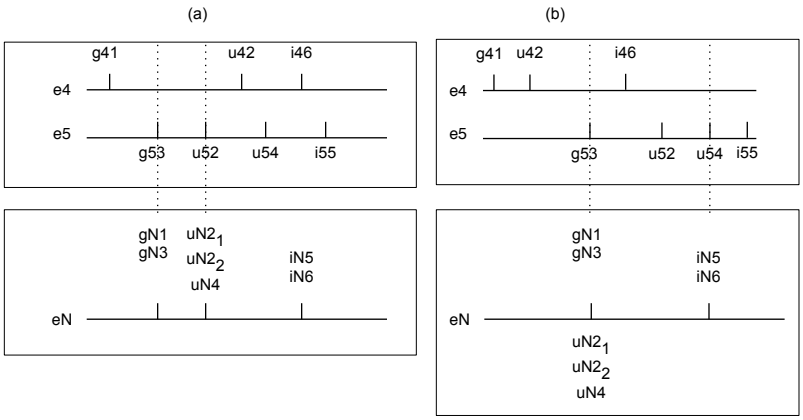

Figure 11: Two possible orderings on G (top), and corresponding orderings on G' (bottom)

In order to avoid excessive clutter in the example, start and generation constraints are only discussed in the next section, along with all general ordering constraints on G'.

Now, consider the linear orderings in G under the assumption that *every generation event precedes every usage event* and *every usage event precedes every invalidation event*, that is, there is a "generation phase" followed by a "usage phase" and by an "invalidation phase" (Fig. 11(a)). It is easy to see that, with this assumption, all these orderings are consistent with constraints (1) through (13), provided that we redefine G' events to be *simultaneous* to corresponding G events, as follows:

$$g_{N1} = g_{N3} = max\{g_{41}, g_{53}\} \tag{14}$$

$$u_{N2_1} = u_{N2_2} = u_{N4} = min\{u_{42}, u_{52}, u_{54}\} \tag{15}$$

$$i_{N5} = i_{N6} = min\{i_{46}, i_{55}\} \tag{16}$$

In this case we conclude that G' is valid by our assumption that each of its generation events precedes each of its usage events.

Consider now the more general case where generation, usage and invalidation events are interleaved for different entities in V_{gr}. Fig. 11(b) shows such an interleaving for our example. In this case, $min\{u_{42}, u_{52}, u_{54}\} = u_{42} \leq g_{53} = max\{g_{41}, g_{53}\}$. This violates (3) through (5). In other words, this more general family of interpretations over G is not represented in G' when the abstract events in G' are defined using the inequalities above.

In order to account for this general case, we modify (15) and (16) as follows:

$$u_{N2_1} = u_{N2_2} = u_{N4} = max\{g_{41}, g_{53}, min\{u_{42}, u_{52}, u_{54}\}\} \tag{17}$$

$$i_{N5} = i_{N6} = max\{g_{41}, g_{53}, u_{42}, u_{52}, u_{54}, min\{i_{46}, i_{55}\}\} \tag{18}$$

In the example of Fig. 11(b), this stricter constraint results in generation and usage events in G' to all be simultaneous to g_{53}, while the invalidation events are shifted later in the event line, to the latest usage. Note that in the special case of Fig. 11 (a), constraints (15), (17) and (16), (18) are pairwise equivalent.

The reasoning used in the examples just presented justifies the following definitions for the general inequalities which define abstract events in terms of events in G.

5.2. Abstract events for e-grouping

Let $G = (V, E) \in PG_{gu/ea}$, $V_{gr} \subset V$ be the set of nodes that are to be grouped, and $e_{new} \in En$ be the new entity node introduced through e-grouping as per Def. 7.

Abstract Generation events. Let V^* to denote $extend(pclos(V_{gr}, G), En)$, and let $genBy_{out}$ denote the set of generation relations involving entity nodes in the extension V^*, and activity nodes outside of the extension:

$$genBy_{out} = \{genBy(e, a) \mid e \in V^*, a \notin V^*\}$$

In the example of Fig. 10, $genBy_{out} = \{g_{41}, g_{53}\}$.

Correspondingly, let $genBy'_{out}$ denote the generation relations that involve e_{new}:

$$genBy'_{out} = \{genBy(e_{new}, a) \mid a \notin V^*\}$$

In the example, $genBy'_{out} = \{g_{N1}, g_{N3}\}$.

In general, we will denote values in the abstracted prov graph by primed versions of their counterparts in the original graph. The exception to this will be relationships and events involving e_{new}, since e_{new} is a new entity that does not appear in the old graph.

The following equalities, define the orderings of the events associated with the relations in $genBy'_{out}$.

Definition 9 (Abstract generation events - e-grouping). *For each* $g' \in genBy'_{out}$*:*

$$ev(g') = max\{ev(g) \mid g \in genBy_{out}\}$$

For all activities a *that participate in generating the new event* e_{new}*, we set the generation event to*

$$ev(genBy(e_{new}, a)) = max\{ev(g) \mid g \in genBy_{out}\}$$

Abstract Usage events. Usage events for e-grouping are defined similarly by generalization from (17), as follows. Let $used_{in}$ denote the set of usage relations involving entity nodes in the extension V^*, and activity nodes outside of the extension:

$$used_{in} = \{used(a, e) \mid e \in V^*, a \notin V^*\}$$

In the example of Fig. 10, $used_{in} = \{u_{42}, u_{52}, u_{54}\}$.

Correspondingly, let $used'_{in}$ denote the usage relations that involve e_{new}:

$$used'_{in} = \{used(a, e_{new}) \mid a \notin V^*\}$$

In the example, $used'_{in} = \{u_{N2_1}, u_{N2_2}, u_{N4}\}$.

The following equalities, which generalise (15), define the events associated with the relations in $used'_{in}$.

Definition 10 (Abstract usage events - e-grouping). *Let*

$$u'_{min} = min\{ev(u) \mid u \in used_{in}\}$$

and let $g'_{max} = max\{ev(g) \mid g \in genBy_{out}\}$.
For each $u' \in used'_{in}$*:*

$$ev(u') = max\{g'_{max}, u'_{min}\}$$

and so

$$ev(used(a, e_{new})) = max\{g'_{max}, u'_{min}\} \tag{19}$$

Abstract Invalidation events. The events equalities for invalidation, exemplified in (18), follow the pattern used above for usage. The only difference is that $used'_{in}$ is replaced by

$$inval'_{in} = \{inval(a, e_{new}) \mid a \notin V^*\}$$

In our example, $inval_{in} = (i_{46}, i_{55}\}$, $inval'_{in} = (i_{N6}, i_{N5}\}$. The corresponding definition is as follows.

Definition 11 (Abstract invalidation events). *Let*

$$i'_{min} = min\{ev(i) \mid i \in inval_{in}\}$$

and

$$u'_{max} = max\{ev(u) \mid u \in used_{in}\}$$

For each $i' \in inval'_{in}$*:*

$$ev(i') = max\{g'_{max}, u'_{max}, i'_{min}\}$$

and thus for each activity a that participates in the invalidation of e_{new}*, invalidation is the latest of the new generation event* g'_{max}*, the latest usage event* u'_{max} *and the minimum of the original invalidation events.*

$$inval(a, e_{new}) = max\{g'_{max}, u'_{max}, i'_{min}\} \tag{20}$$

Start events: Now that the event of usage and generation of e_{new} has been fixed, we need to ensure that the activities involved in these events continue to meet the constraints that apply to them. This includes start and end events for activities related to our new entity e_{new} by either *genBy* or *used*.

We appeal to the informal definitions of start and end in [21] to derive inequalities for the abstract start and end events for our new activity a_{new}.

- **Start** *is when an activity is deemed to have been started by an entity, known as trigger. The activity did not exist before its start. Any usage, generation, or invalidation involving an activity follows the activity's start* (See [21], Section 5.1.6)
- **End** *is when an activity is deemed to have been ended by an entity, known as trigger. The activity no longer exists after its end. Any usage, generation, or invalidation involving an activity precedes the activity's end* (See [21], Section 5.1.7)

(For simplicity we are going to leave the trigger entity implicit, and simply refer to the start and end events as $start(a)$ and $end(a)$).

Definition 12 (Start events - e-grouping). *Consider events a such that* $genBy(e_{new}, a)$. *For each such a, we set the new start event* $start'(a)$ *as the lesser of the original start event and the generation event* $ev(genBy(e_{new}, a))$. *Thus*

$$start'(a) = min\{start(a), ev(genBy(e_{new}, a))\} \tag{21}$$

End events: For all activities a that use the newly created entity, we must ensure that ensure that the end of the activity does not precede any $used(a, e_{new})$ events.

Definition 13 (End events - e-grouping). *If the set of all usage events by* a *of* e_{new} *is denoted* $\{ev(used(a, e_{new}))\}$, *we set the new end events* $end'(a)$ *to be*

$$end'(a) = max\{end(a), max\{ev(used(a, e_{new}))\}\} \tag{22}$$

A proof of that, given the definitions above, the constraints of Section 3.3 are satisfiable, is given in Appendix A.

5.3. Abstract events for a-grouping

Generation and usage abstract events follow a very similar pattern as those for e-grouping, except that the new node introduced by grouping is an activity node: $a_{new} \in Act$. As a consequence, the abstract event definitions given in the previous section also follow the same pattern, but with the roles of entities and activities reversed. They are summarized here below. We now use V^* to mean the group of nodes collected by $extend(pclos(V_{gr}, G), Act)$.

The new start (resp. end) event is taken to be the minimum (resp. maximum) relevant start (resp. end) event.

Definition 14 (Abstract start and end events - a-grouping).

$$start(a_{new}) = min\{start(a) \mid a \in V^*\}$$
$$end(a_{new}) = max\{end(a) \mid a \in V^*\}$$

Definition 15 (Abstract generation events - a-grouping). *Constraint C2 applies to the original graph, so for all activities a, b that participate in the generation of e, $ev(genBy(e, a)) = ev(genBy(e, b))$.*

The new generation event is thus given as

$$ev(genBy(e, a_{new})) = ev(genBy(e, a))$$

Definition 16 (Abstract usage events - a-grouping). *For an entity e not in V^* which is used by an activity a in V^*, the assigned usage event for a_{new} ($ev(used(a_{new}, e))$) is given by*

$$ev(used(a_{new}, e)) = max\{start(a_{new}), \\ min\{ev(used(a, e))) \mid a \in V^*\}\}$$

Definition 17 (Abstract invalidation events - a-grouping). *By Constraint C8, invalidation events from two distinct activities are simultaneous. For an entity e which is invalidated by an activity a in V^*, the event of the invalidation event remains the same when the activity is abstracted.*

$$ev(inval(a_{new}, e)) = min\{ev(inval(a, e)) \mid a \in V^*\}$$

A proof of that, given the definitions above, the constraints of Section 3.3 are satisfiable, is given in Appendix A.

6. Abstraction with Agents

Having laid the foundations for abstraction on the core $PG_{gu/ea}$ model, extending grouping to a model that also includes agents, the third pillar of the PROV model, is quite straightforward. Agents may be humans or software systems. Specifically, we now consider the node type Ag and the following additional relation types from the PROV schema of Sec. 3.1:

$$\begin{aligned} waw &\subseteq Act \times Ag \\ wat &\subseteq En \times Ag \\ abo &\subseteq Ag \times Ag \end{aligned}$$

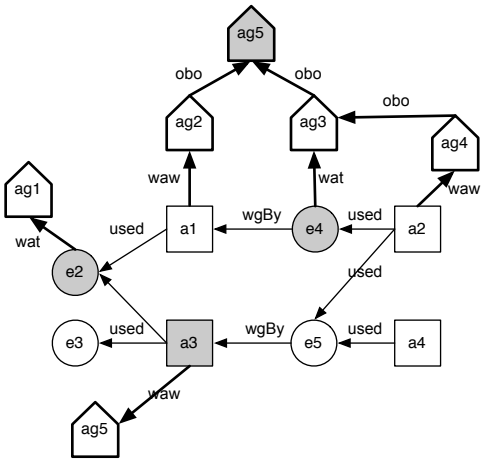

Figure 12: $PG_{gu+/eaAg}$ provenance graph. Agents lie on the outer border of the digraph. Shaded nodes show a possible grouping set in the general case.

We use the shorthand relation names *waw*, *wat* and *abo* for *wasAssociatedWith*, *wasAttributedTo* and *actedOnBehalfOf*. These denote responsibility of an agent for an activity (*waw*), responsibility of an agent for an entity (*wat*), and delegation between two agents (*abo*).

Note that PROV admits an additional optional activity element to *abo*, which is used to qualify the delegation as occurring within the scope of that activity. For simplicity, we are not going to consider this qualified version of the relation. Thus, we can still assume that these new relations are binary, and so we continue to view an instance of a provenance graph as a directed graph $G = (V, E)$, where new $V = En \cup Act \cup Ag$, and where each relation instance maps to a labelled directed edge. We denote the set of all such graphs as $PG_{gu+/eaAg}$.

The main implications of adding agents to our abstraction model are that (i) a new *ag-grouping* operator must be introduced, and (ii) the existing definitions of e-grouping and a-grouping must be modified slightly.

In order to incorporate agents into the definition of *Group*, observe that, for all three relations involving agents, the agent node is always the *target* of the directed edge. This means that agents can be viewed as part of an "outer layer" in the provenance graph. This is illustrated in Fig. 12, where the agent nodes and new relations are shown in bold lines in the outer side of the digraph.

This observation suggests we can break down the analysis of grouping with agents into the following three parts.

1. $V_{gr} \subset Ag$. This is the case for ag-grouping, which only involves the outer layer of the graph. Since agents are only related to each other through delegation: $abo(ag_1, ag_2)$, grouping in this case is akin to *homogeneous grouping* from Sec. 4.1, in the sense that no nodes of other types are ever

involved, and $v_{new} \in Ag$.

2. $V_{gr} \subset En \cup Act$ as in Sec. 4. This is the case of t-grouping (Def. 7), where the nodes involved in the abstraction are in the inner layer, but they may be related to agent nodes via *waw* and *wat* relations.
3. $V_{gr} \subset En \cup Act \cup Ag$. Here, the group set may contain any combination of nodes. However, the peripheral role played by agents relative to entities and activities suggests that it may be reasonable to restrict this case to e-grouping or a-grouping, i.e., a combination of node types may include agents, but it should not be abstracted by a new agent node.

6.1. Ag-grouping: abstracting agents

We begin with the case where abstraction is performed over a set of agents, i.e., $V_{gr} \subset Ag$. In this case, the existing definition of t-grouping (Def. 7) extends naturally to $PG_{gu+/eaAg}$ graphs. T-grouping involves three operators: *pclos*, *extend*, and *replace*. Since $pclos(V_{gr}, G)$ operates only on *abo* relations, it follows that its result is also homogeneous, i.e., $pclos(V_{gr}, G) \subset Ag$. Also, there is no need to restore type validity by extending the closure, i.e., *extend* is the identity: $extend(pclos(V_{gr}, G), V, Ag) = pclos(V_{gr}, G)$. Finally, it is easy to see that our original definition of group replace (Def. 5) is general enough to accommodate the "rewiring" of the new abstract agent node. We illustrate this informally using the patterns of Fig. 13.

In pattern (a), $V_{gr} = \{ag_1, ag_4\}$, and $pclos(V_{gr}, G) = \{ag_1, ag_2, ag_3, ag_4\}$. In this case, the closure includes all the intermediate agents in the delegation chain between ag_1 and ag_4. Replacement trivially transforms G into the single abstract agent ag_N.

In pattern (b), $V_{gr} = \{ag_2, ag_5, ag_4\}$. Note that not all agent nodes in V_{gr} are related, either directly or through a path. This is not a problem, as we have $V_{clos} = pclos(V_{gr}, G) = \{ag_2, ag_5, ag_4, ag_3\}$. Replacement applies as follows, where $\vartheta_{int}(V_{clos})$ is the set of relations beginning and ending inside V_{clos}:

$$\begin{aligned} \vartheta_{int}(V_{clos}) &= \{abo(ag_2, ag_3), abo(ag_3, ag_5)\} \\ \vartheta_{in}(V_{clos}) &= \{wat(e, ag_2), abo(ag_1, ag_4)\} \\ \vartheta_{out}(V_{clos}) &= \{abo(ag_4, ag_6)\} \end{aligned}$$

Thus, $replace(V_{clos}, ag_N, G)$ maps relations in the original graph to those in the abstracted graph as follows:

$$\begin{aligned} abo(ag_4, ag_6) &\rightarrow abo(ag_N, ag_6) \\ wat(e, ag_2) &\rightarrow wat(e, ag_N) \\ abo(ag_1, ag_4) &\rightarrow abo(ag_1, ag_N) \end{aligned}$$

In practice, replacement preserves agents ag_1 and ag_6 and restores their delegation relations relative to the new abstract agent, ag_N. It also maps relation $wat(e, ag_2)$, which involves the untouched e node, to a new relation of the same type: $wat(e, ag_N)$.

We conclude that, in this first case, Def. 6 applies without changes.

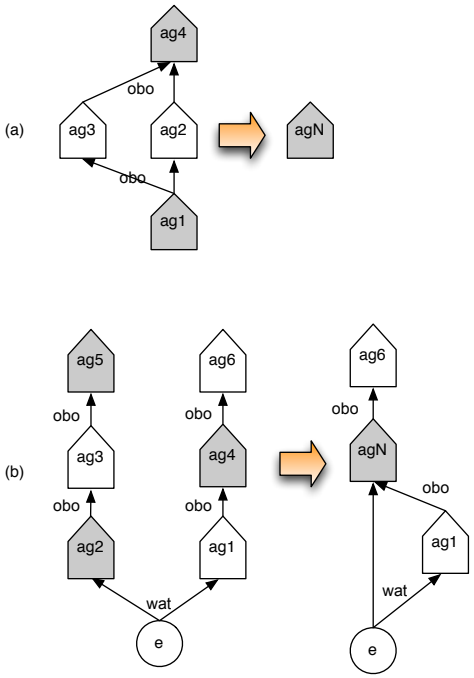

Figure 13: ag-grouping involving delegation.

6.2. a-grouping and e-grouping with agents

The second case, where $V_{gr} \subset En \cup Act$, is t-grouping with added agents relations. Here the *replace* operator (Def. 5) must now consider how edges that involve agents are mapped to new edges in the abstract graph. We have already observed that agent nodes are always the targets of directed edges. It follows that the closure of a set of nodes $V_{gr} \subset En \cup Act$ never adds agent nodes to V_{gr}, because this would require the added agent to be on a path between two nodes from $En \cup Act$, and therefore to be the source of a directed edge.

A second observation is that if $waw(a, ag)$ holds, and a is involved in a-grouping, then a is replaced by a_{new}, and thus $waw(a_{new}, ag)$ also holds (Fig. 14(a1)). Similarly for e-grouping, if $wat(e, ag)$ holds, and e is involved in e-grouping, then e is replaced by e_{new}, and $wat(e_{new}, ag)$ holds (Fig. 14(b1)). On the other hand, suppose $waw(a, ag)$ holds and e-grouping is performed. A simple case is shown in Fig. 14(c1). In this case, a is replaced by $e_{new} \in En$, therefore $waw(e_{new}, ag)$ is type-incorrect. Similarly, $wat(e, ag)$ after a-grouping would become, incorrectly, $wat(a_{new}, ag)$. These two patterns are summarised in Fig. 14(a2, b2 resp.). Note that one cannot simply replace association with attribution, i.e., replace relation $waw(a, ag)$ with $wat(e_N, ag)$, because there is no guarantee that any of the entities represented by the new e_N had been attributed to ag in the original graph. Similarly, one cannot replace $wat(e, ag)$ with $waw(a_N, ag)$. Instead, in pattern (a) we simply remove the incorrect *waw* relations following e-grouping, and similarly, in pattern (b) we remove the incorrect *wat* relations following a-grouping.

These considerations suggest that the definition of the *replace* function for grouping needs to be adapted for the case where agents are involved. To understand why, recall from Sec. 4.1 that *replace* replaces a type-homogeneous set, computed by the *extend* function, with an abstract node of the same type. An extension of type t augments the closure of a grouping set by adding all adjacent nodes of the same type to it. This ensures that replacing the nodes in the extension with an abstract node of the same type preserves the type correctness of the relations. However it should be clear from the example above (parts c1, c2 of the figure), that both e-grouping and a-grouping on the set $V_{gr} = \{a, e\}$ result in one of the two agent relations being incorrect. Intuitively, this is because the extension function fails to incorporate agents, leaving the agent relations exposed on the outcut of V_{gr} (in fact, *extend* in this example does nothing at all).

The pattern in (c2) (or its symmetric, for a-grouping), can be obtained simply by ensuring that *replace* deletes the incorrect relations. The following variation on Definition 4 ensures these deletions are enforced.

$$\begin{aligned}\vartheta'_{out}(V'_{gr}) = \{v \overset{t}{\leftarrow} v_{new} \mid v \overset{t}{\leftarrow} v' \in \vartheta_{out}(V'_{gr}) \wedge \\ ((t = wat \wedge type(v_{new}) = En) \vee \\ (t = waw \wedge type(v_{new}) = Act) \vee \\ (t \neq wat \wedge t \neq waw))\}\end{aligned}$$

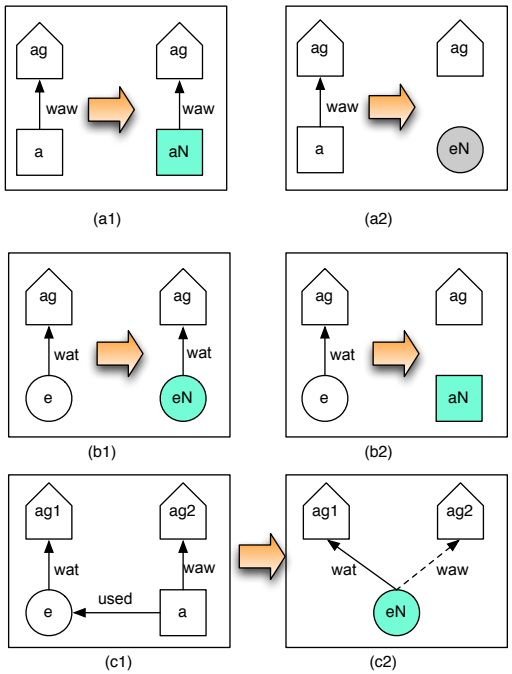

Figure 14: *waw* and *wat* edges involving nodes in $pclos(V_{gr}, G)$ may need to removed following e-grouping and a-grouping, respectively.

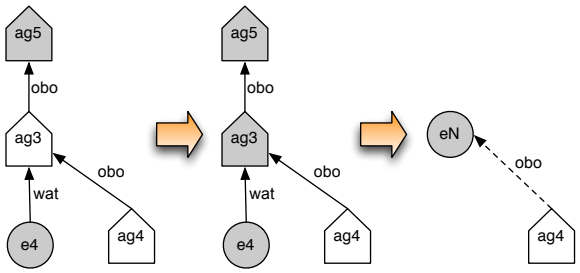

Figure 15: e-grouping leads to incorrect *abo* relation when agents are part of a closure.

6.3. The general case: grouping on any node type

The third and more general case, where $V_{gr} \subset En \cup Act \cup Ag$, presents one further difficulty. Consider performing e-grouping on the pattern of Fig. 15 (left), with $V_{gr} = \{e_4, ag_5\}$. We have $pclos(V_{gr}, G) = \{e_4, ag_5, ag_3\}$, resulting in the abstraction on the right. Clearly, the former $abo(ag_4, ag_3)$ relation should not be mapped in the final graph. This issue arises because there the extension function, which guarantees type consistency for e- and a-nodes, does not include agents.

Once again we deal with the issue by changing the definition of *replace* to ensure that the incorrect relation is not mapped. Note that a change is required to $\vartheta'_{in}(V'_{gr})$ rather than $\vartheta'_{out}(V'_{gr})$ as in the previous case. The new definition is as follows.

$$\vartheta'_{in}(V'_{gr}) = \{v_{new} \overset{t}{\leftarrow} v \mid v' \overset{t}{\leftarrow} v \in \vartheta_{in}(V'_{gr}) \\ \wedge ((t = abo \wedge type(v_{new}) = Ag) \vee t \neq abo)\}$$

Thus, a delegation relation is mapped in the abstract graph only if the target abstract node is an agent.

With the new version of the *replace* function, introduced in the previous two sections, some of the relations in the original graph G are not mapped to the abstracted graph G'. This makes it possible for some of the nodes in G' to end up disconnected from the rest of the graph. As an example, the graph in Fig. 16 shows the result of e-grouping over the shaded nodes in the graph of Fig. 12. The combination of closure, extensions and replacement results in ag_5 being isolated, or "orphaned" in the abstract graph.

As isolated agent nodes may not be significant to consumers of the abstracted graphs, for completeness we provide a simple function to optionally remove them at the end of the abstraction process, as follows.

Definition 18 (Removing isolated agents). *Let* $G = (V, E) \in PG_{gu+/eaAg}$, *and*

$$isolated(G) = \{v \in V \mid type(v) = Ag \\ \wedge \nexists v' \in V.((v', v) \in E \vee (v, v') \in E)\}$$

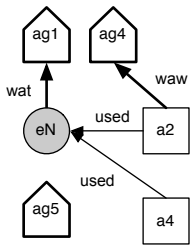

Figure 16: Abstracted version of the graph in Fig. 12 after e-grouping involving agents.

Then:

$$remIsolated((V, E)) = (V \setminus isolated(G), E)$$

7. Summary and further research

We have proposed a model for the principled obfuscation of provenance based on a formal definition of abstraction, in which sensitive elements of a provenance graph are grouped together and replaced by a single abstract node. The presentation is incremental: first we develop the model for the case when the nodes of provenance graphs are restricted to entities and activities, and then consider the case where nodes may also be agents. A guiding principle throughout is that we avoid the introduction of *false dependencies*: abstraction will reduce the information content of a provenance graph, but it will not introduce false information. The abstraction acts on and results in provenance graphs which are PROV compliant. A separate paper presents the tool implementing this model in detail.

The work described in this paper is progressing in two main directions. First, we are aware that the fragment of PROV to which this version of *Group* applies does not cover all relation types. Nevertheless, the method described in the paper for reasoning about PROV graph transformation can be used as a guideline to extend the work to the missing parts of PROV. We are going to address these in the future.

Second, so far we have ignored the implications of abstraction on the space of events that are used to characterize the semantics of PROV. As constraints over the relative ordering of events are defined in detail in the PROV-CONSTR document, there is however an obligation to extend the notion of validity of PROV graphs to include those constraints. Thus, grouping must be shown to be validity-preserving relative to those constraints as well.

Acknowledgements

The support of the ONRG and the EPSRC in funding this research is gratefully acknowledged.

References

[1] Eleanor Ainy, Pierre Bourhis, Susan B. Davidson, Daniel Deutch, and Tova Milo. Approximated summarization of data provenance. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 483–492, New York, NY, USA, 2015. ACM.

[2] D Bell. The bell-lapadula model. *Journal of computer security*, 4(2):3, 1996.

[3] Smriti Bhagat, Graham Cormode, Balachander Krishnamurthy, and Divesh Srivastava. Class-based graph anonymization for social network data. *Proc. VLDB Endow.*, 2(1):766–777, August 2009.

[4] O Biton, S Cohen Boulakia, S B Davidson, and C S Hara. Querying and Managing Provenance through User Views in Scientific Workflows. In *ICDE*, pages 1072–1081, 2008.

[5] Uri Braun, Avraham Shinnar, and Margo Seltzer. Securing provenance. In *Proceedings of the 3rd conference on Hot topics in security*, pages 4:1—-4:5, Berkeley, CA, USA, 2008. USENIX Association.

[6] Jeremy Bryans, John S. Fitzgerald, Cliff B. Jones, and Igor Mozolevsky. Formal modelling of dynamic coalitions, with an application in chemical engineering. In *ISoLA*, pages 91–98. IEEE, 2006.

[7] Tyrone Cadenhead, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham. A language for provenance access control. In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 133–144, New York, NY, USA, 2011. ACM.

[8] Tyrone Cadenhead, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham. Transforming provenance using redaction. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 93–102, New York, NY, USA, 2011. ACM.

[9] James Cheney, Paolo Missier, and Luc Moreau. Constraints of the Provenance Data Model. Technical report, 2012.

[10] Susan B Davidson, Sanjeev Khanna, Tova Milo, Debmalya Panigrahi, and Sudeepa Roy. Provenance views for module privacy. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 175–186, New York, NY, USA, 2011. ACM.

[11] Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, and Sarah Cohen Boulakia. Privacy issues in scientific workflow provenance. In Paolo Missier, Vasa Curcin, and Susan Dadvidson, editors, *First International Workshop on Workflow Approaches to New Data-centric Science (WANDS'10)*, Indianapolis, 2010. ACM.

[12] Susan B Davidson, Sanjeev Khanna, Sudeepa Roy, Julia Stoyanovich, Val Tannen, and Yi Chen. On provenance and privacy. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, pages 3–10, New York, NY, USA, 2011. ACM.

[13] Saumen Dey, Daniel Zinn, and Bertram Ludäscher. ProPub: Towards a Declarative Approach for Publishing Customized, Policy-Aware Provenance. In Judith Bayard Cushing, James French, and Shawn Bowers, editors, *Scientific and Statistical Database Management*, volume 6809 of *Lecture Notes in Computer Science*, pages 225–243. Springer Berlin / Heidelberg, 2011.

[14] Daniele El-Jaick, Marta Mattoso, and Alexandre A B Lima. SGProv: Summarization Mechanism for Multiple Provenance Graphs. *JIDM*, 5(1):16–27, 2014.

[15] Ragib Hasan, Radu Sion, and Marianne Winslett. Introducing secure provenance: problems and challenges. In *Proceedings of the 2007 ACM workshop on Storage security and survivability*, StorageSS '07, pages 13–18, New York, NY, USA, 2007. ACM.

[16] Kun Liu and Evimaria Terzi. Towards identity anonymization on graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 93–106, New York, NY, USA, 2008. ACM.

[17] Paolo Missier, Jeremy Bryans, Carl Gamble, Vasa Curcin, and Roxana Danger. ProvAbs: model, policy, and tooling for abstracting PROV graphs. In *Procs. IPAW 2014 (Provenance and Annotations)*, Koln, Germany, 2014. Springer.

[18] Paolo Missier, Jeremy Bryans, Carl Gamble, Vasa Curcin, and Roxana Dánger Mercaderes. Provabs: model, policy, and tooling for abstracting PROV graphs. *CoRR*, abs/1406.1998, 2014.

[19] Luc Moreau. Aggregation by provenance types: A technique for summarising provenance graphs. *arXiv preprint arXiv:1504.02616*, 2015.

[20] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van Den Bussche. The Open Provenance Model — Core Specification (v1.1). *Future Generation Computer Systems*, 7(21):743–756, 2011.

[21] Luc Moreau, Paolo Missier, Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. PROV-DM: The PROV Data Model. Technical report, World Wide Web Consortium, 2012.

[22] Dang Nguyen, Jaehong Park, and Ravi Sandhu. Dependency path patterns as the foundation of access control in provenance-aware systems. In *4th USENIX Workshop on the Theory and Practice of Provenance*, 2012.

[23] Elena Zheleva and Lise Getoor. Preserving the Privacy of Sensitive Relationships in Graph Data. In Francesco Bonchi, Elena Ferrari, Bradley Malin, and Yücel Saygin, editors, *Privacy, Security, and Trust in KDD*, volume 4890 of *Lecture Notes in Computer Science*, pages 153–171. Springer Berlin / Heidelberg, 2008.

[24] Bin Zhou, Jian Pei, and WoShun Luk. A brief survey on anonymization techniques for privacy preserving publishing of social network data. *SIGKDD Explor. Newsl.*, 10(2):12–22, December 2008.

Appendix A. Consistency of constraints for e-grouping

In this section we prove that the event definitions we have given for e-grouping in Section 5.2 do not render any of the constraints C1-C7 in Section 3.3 unsatisfiable.

Let $G = (V, E) \in PG_{gu/ea}$, be the original graph, $V^* \subset V$ be the set of nodes that are replaced by the *Group* operator, and let $e_{new} \in En$ be the new entity node introduced through e-grouping as per Def. 7.

In some cases the abstraction changes a timing of a node without changing the identity of the node. In these cases we refer to the original start time as $start(a)$ and the new start time as $start'(a)$.

The constraints follow.

C1: entity-activity-disjoint

The only new node introduced has type En by definition, so C1 is still satisfied: $Act \cap En = \emptyset$.

C2: generation-generation-ordering

To show this, we must show that for all activities a_1, a_2 (outside of V^*) in the original graph that generate some entities in V^*, it is the case that $genBy(e_{new}, a_1)$ and $genBy(e_{new}, a_2)$ are simultaneous. I.e.

$$\forall\, genBy(e_{new}, a_1), genBy(e_{new}, a_2) \in genBy'_{out} \bullet \\ ev(genBy'(e_{new}, a_1)) = ev(genBy'(e_{new}, a_2))$$

But by Definition 9, each for generation event $g' \in genBy'_{out}$, it is the case that $ev(g') = max\{ev(g) \mid g \in genBy_{out}\}$, and thus in particular $ev(genBy'_{out}(e_{new}, a_1)) = ev(genBy'_{out}(e_{new}, a_2))$, as required.

All other generation events in the original graph either sit wholly within V^*, and are therefore abstracted away, or wholly outside, and the generation times remain unchanged.

C3: generation-precedes-usage

For all activities a in the graph for which $genBy(e_{new}, a)$, we must show that the generation time assigned to the event $genBy(e_{new}, a)$ precedes any use made of the new entity e_{new}.

By Definition 9, $ev(genBy(e_{new}, a)) = max\{ev(g) \mid g \in genBy_{out}\}$.

By Definition 10, usage time is set to $max\{ev(genBy(e_{new}, a), min\{ev(u) \mid u \in used_{in}\}$ and thus generation precedes, or is simultaneous with, usage.

So, for any generating activity a and using activity b, $ev(genBy(e_{new}, a)) \leq ev(used(b, e_{new}))$.

C5: usage-precedes-invalidation

Demonstrating C5 first will simplify the demonstration of C4. For any a in $genBy'_{in}$, the time of usage of entity e_{new} is given by Definition 10 to be

$$ev(used(a, e_{new})) = max\{g'_{max}, u'_{min}\}$$

For any invalidating activity b, the time of invalidation is given by

$$ev(inval(b, e_{new})) = max\{g'_{max}, u'_{max}, i'_{min}\}$$

and, since $u'_{min} \leq u'_{max}$,

$$max\{g'_{max}, u'_{min}\} \leq max\{g'_{max}, u'_{max}\}$$

and since $max\{x\} \leq max\{x, y\}$,

$$max\{g'_{max}, u'_{max}\} \leq max\{g'_{max}, u'_{max}, i'_{min}\}$$

and so it follows that

$$ev(used(a, e_{new})) \leq ev(inval(b, e_{new}))$$

as required.

C4: generation-precedes-invalidation

For entity e_{new}, we have shown (C3) that for any activities a and b that generate and use e_{new} respectively,

$$ev(genBy(e_{new}, a)) \leq ev(used(b, e_{new}))$$

and (C4) that, for any activities b, c that use and invalidate entity e_{new} respectively,

$$ev(used(b, e_{new})) \leq ev(inval(c, e_{new}))$$

and so it follows that, for any activities a and c that generate and invalidate e_{new} respectively,

$$ev(wgby(e_{new}, a)) \leq ev(inval(c, e_{new}))$$

as required.

C6: usage-within-activity In e-grouping, we risk violating this constraint by moving the time at which activities use entities.

We need to show that for any a which uses e_{new} $start'(a) \leq ev(used(a, e_{new}))$ and that $ev(used(a, e_{new})) \leq end'(a)$.

First, observe (by Definition 12), that

$$start'(a) = min\{start(a), ev(genBy(e_{new}, a))\}$$

By C3 (generation-precedes-usage) we have that, for any generating activity a and using activity b, $ev(genBy(e_{new}, a)) \leq ev(used(b, e_{new}))$, and so for any using activity b,

$$start'(a) \leq min\{start(a), ev(used(b, e_{new}))\}$$

and in particular $start'(a)$ precedes any usage made by a of e_{new}:

$$start'(a) \leq min\{start(a), ev(used(a, e_{new}))\}$$

and so

$$start'(a) \leq min\{ev(used(a, e_{new}))\}$$

Second, observe from Definition 13 that the new end time is set to be the later of the original end time $end(a)$ and the latest of any uses made by a of e_{new}, given by $max\{ev(used(a, e_{new}))\}$.

$$end'(a) = max\{end(a), max\{ev(used(a, e_{new}))\}\}$$

Thus,

$$end'(a) \geq max\{ev(used(a, e_{new}))\}$$

and so

$$max\{ev(used(a, e_{new}))\} \leq end'(a)$$

Therefore, since clearly

$$min\{ev(used(a, e_{new}))\} \leq max\{ev(used(a, e_{new}))\}$$

it follows that for any a that uses e_{new},

$$\begin{array}{l} start'(a) \leq \\ \quad min\{ev(used(a, e_{new}))\} \leq \\ \quad\quad max\{ev(used(a, e_{new}))\} \leq \\ \quad\quad\quad end'(a) \end{array}$$

as required.

C7: generation-within-activity

For a generating activity a of e_{new}, the generation of e_{new} cannot precede the new start of a ($start'(a)$) and must precede the new end of a ($end'(a)$).

$$start'(a) \leq ev(genBy(e_{new}, a)) \leq end'(a)$$

By Definition 12, we have that the new start time of the a is set to be the lesser of the original start time and $ev(genBy(e_{new}, a))$, thus ensuring that

$$start'(a) \leq ev(genBy(e_{new}, a))$$

as required.

By Definition 13 we have that the new end time of a is set to the greater of the old end time $end(a)$ and all the new use times $max\{ev(used(a, e))\}$ of a by any event e.

$$end'(a) = max\{end(a), max\{ev(used(a, e))\}\}$$

and so in particular

$$end'(a) \geq max\{ev(used(a, e_{new}))\}$$

By Constraint C3, the generation of an event precedes any usage of that event, so in particular this is true for all usages of e_{new} by a.

$$ev(used(a, e_{new})) \geq max\{ev(genBy(e_{new}, a))\}$$

and so

$$end'(a) \geq max\{ev(genBy(e_{new}, a))\}$$

Appendix B. Consistency of constraints for a-grouping

In this section we prove that the event definitions we have given for a-grouping in Section 5.3 do not render any of the constraints C1-C7 in Section 3.3 unsatisfiable.

Let $G = (V, E) \in PG_{gu/ea}$, be the original graph, $V^* \subset V$ be the set of nodes that are replaced by the *Group* operator, and let $a_{new} \in Act$ be the new activity node introduced through a-grouping as per Def. 7.

The constraints follow.

C1: entity-activity-disjoint

The only new node introduced has type Act by definition, so C1 is still satisfied: $Act \cap En = \emptyset$.

C2: generation-generation-ordering

To show this, we must show that for any entity e (outside of V^*) that is generated by activity (say a) in V^* and another activity (say b) not in V^*, it is the case that $genBy(e, a)$ and $genBy(e, b)$ are simultaneous, i.e.

$$ev(genBy(e, a_{new})) = ev(genBy(e, b))$$

But, since a-ordering does not change the generation times of any entities, this follows immediately.

C3: generation-precedes-usage

For an entity e not in V^* that is generated by some activity in a in V^*,

$$ev(genBy(e, a_{new})) = ev(genBy(e, a))$$

Since in the original graph generation precedes usage, consider an activity b that uses e. Note that generation-usage cycles are excluded by a-grouping, so b is not in V^*. We know that

$$ev(genBy(e, a)) \leq ev(used(b, e))$$

and so it follows that for any activity that uses e,

$$ev(genBy(e, a_{new})) \leq ev(used(\_, e))$$

as required.

For an entity e not in V^* that is used by some activity in a in V^*, we know that in the original graph, generation of e by some activity precedes *all* usages of e, i.e.

$$ev(genBy(e, \_)) \leq min\{ev(used(b, e))) \mid b \in V^*\}$$

It follows from C3 that for arbitrary usage of e in the original graph,

$$\begin{aligned}
&ev(genBy(e, _)) \leq \{\text{by C3}\} \\
&\quad min\{ev(used(b, e))) \mid b \in V^*\} \leq \{\text{by arithmetic}\} \\
&\qquad max\{start(a_{new}), min\{ev(used(a, e))) \mid a \in V^*\}\} \\
&\qquad\quad = \{\text{by Definition 16}\} \\
&\qquad\qquad ev(used(a_{new}, e))
\end{aligned}$$

C5: usage-precedes-invalidation

For an entity e not in V^* that is invalidated by some activity a in V^*,

$$ev(inval(e, a_{new})) = ev(inval(e, a))$$

and since in the original graph usage precedes invalidation, for any activity that uses e,

$$ev(used(\_, e)) \leq ev(inval(e, a))$$

It follows that for any activity that uses e,

$$ev(used(\_, e)) \leq ev(inval(e, a_{new}))$$

as required.

C4 generation-precedes-invalidation

For entities e not in V^*, neither generation nor invalidation times have changed, so it follows that in the abstracted graph generation continues to precede invalidation.

C6 usage-within-activity

We are required to prove that for any usage events of some entity e by the new activity node a_{new} it is the case that $start(a_{new}) \leq ev(used(a_{new}, e))$ and that $ev(used(a_{new}, e)) \leq end(a_{new})$.

First, let b be a node in V^* that used e. Since

$$ev(start(a_{new})) = min\{start(a) \mid a \in V^*\}$$

by Definition 14, it follows that

$$\begin{array}{l} ev(start(a_{new})) \\ \quad \leq max\{start(b) \mid b \in V^*\} \\ \quad\quad \leq max\{max\{start(b) \mid b \in V^*\}, \\ \quad\quad\quad\quad min\{ev(used(a, e) \mid a \in V^*\}\} \\ \quad\quad\quad\quad = ev(used(a_{new}, e)) \end{array}$$

We next need to show that $ev(used(a_{new}, e)) \leq end(a_{new})$

Since by definition,

$$\begin{array}{l} ev(used(a_{new}, e)) = max\{start(a_{new}), \\ \quad\quad\quad\quad\quad\quad\quad\quad min\{ev(used(a, e))) \mid a \in V^*\}\} \end{array}$$

and since for all activities, the start time precedes the end time, it is clear that

$$min\{start(a) \mid a \in V^*\} \leq max\{end(a) \mid a \in V^*\}$$

and so

$$\begin{array}{l} ev(used(a_{new}, e)) \leq \\ \quad start(a_{new}) = \{\text{by Definition 14}\} \\ \quad\quad min\{start(a) \mid a \in V^*\} \leq \{\text{by above}\} \\ \quad\quad\quad max\{end(a) \mid a \in V^*\} = \{\text{by Definition 14}\} \\ \quad\quad\quad\quad end(a_{new}) \end{array}$$

C7 generation-within-activity

First, we show that all generation events are subsequent to the start event. Let b be a node in V^* that used e. Since

$$ev(start(a_{new})) = min\{start(a) \mid a \in V^*\}$$

it follows that

$$\begin{array}{l} ev(start(a_{new})) = \{\text{by Definition 14}\} \\ \quad min\{start(a) \mid a \in V^*\} \leq \{\text{by original graph, C7}\} \\ \quad\quad min(ev\{genBy(e,a) \mid a \in V^*\}) \\ \quad\quad\quad = \{\text{since generation is simultaneous}\} \\ \quad\quad\quad\quad ev(genBy(e,a)) = \{\text{by Definition 15}\} \\ \quad\quad\quad\quad\quad ev(genBy(e,a_{new})) \end{array}$$

Secondly, we show that all generation events precede the end of the activity. Let e be an entity generated by some activity a in V^*.

$$\begin{array}{l} ev(genBy(e,a_{new})) = \{\text{by Definition 15}\} \\ \quad ev(genBy(e,a) \leq \{\text{by C7, original graph}\} \\ \quad\quad end(a) \leq \{\text{arithmetic}\} \\ \quad\quad\quad max\{end(b) \mid b \in V^*\} = \{\text{by Definition 14}\} \\ \quad\quad\quad\quad end(e_{new}) \end{array}$$