ProvAbs: model, policy, and tooling for abstracting PROV graphs*

Paolo Missier[1], Jeremy Bryans[1], Carl Gamble[1], Vasa Curcin[2], and Roxana Danger[2]

[1] School of Computing Science, Newcastle University
[2] Imperial College, London

Abstract. Provenance metadata can be valuable in data sharing settings, where it can be used to help data consumers form judgements regarding the reliability of the data produced by third parties. However, some parts of provenance may be sensitive, requiring access control, or they may need to be simplified for the intended audience. Both these issues can be addressed by a single mechanism for creating abstractions over provenance, coupled with a policy model to drive the abstraction. Such mechanism, which we refer to as *abstraction by grouping*, simultaneously achieves partial disclosure of provenance, and facilitates its consumption. In this paper we introduce a formal foundation for this type of abstraction, grounded in the W3C PROV model; describe the associated policy model; and briefly present its implementation, the `ProvAbs` tool for interactive experimentation with policies and abstractions.

1 Introduction

Provenance, a formal representation of the production process of data, may facilitate the assessment and improvement of the quality of data products, as well as the validation and reproducibility of scientific experimental datasets. This expectation predicates on an assumption of interoperability between mutually independent producers and consumers of provenance. The W3C PROV generic provenance model [1] is intended to facilitate such interoperability, by providing a common syntax and semantics for provenance models, and thus enable provenance-aware data sharing at Web scale.

1.1 Abstracting provenance

For provenance to be useful, it must be represented at a level of abstraction that is appropriate to the consumer. For example, system-level provenance which includes individual system calls and I/O operations may be appropriate for system auditing purposes, while a higher level description may be more appropriate to determine how a document evolved to its final version, e.g. through a series of edits involving multiple authors. In some cases, the higher abstraction can be computed from the detailed representation. One such case occurs when provenance describes the execution of a workflow or dataflow, which can itself be described at multiple levels of abstraction. Early work on *provenance views* (*Zoom*) [2] is an example. Here users specify the abstraction they require on the workflow, and that is used to compute a corresponding abstract view of the

* This work was funded in part by EPSRC UK and DSTL under grant EP/J020494/1

workflow's trace. More generally, however, a trace may represent arbitrary process executions and data derivations, and one cannot rely on a formal description of the process to specify a suitable abstraction.

The problem of abstracting over provenance in such a more general setting has been addressed in later work, notably the ProPub system [3]. Here the main goal is to ensure that sensitive elements of the trace are abstracted out, by means of a redaction process. In ProPub, users specify edit operations on a provenance graph, such as anonymizing, abstracting, and hiding certain parts of it. ProPub operates on a simplified provenance model (which pre-dates PROV) which only includes use/generation relations, and adopts an "apply–detect–repair" approach. First, user-defined abstraction rules are applied to the graph, then consistency violations that may occur in the resulting new graph are detected, and finally a set of edits are applied 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 user rules.

1.2 Contributions

Our work is motivated by the need to control the complexity of a provenance graph by increasing its level of abstraction, as well as to protect the confidentiality of parts of the graph. Our specific contributions in this paper are threefold. Firstly, we define a *Provenance Abstraction Model* (**PAM**) centred on the $Group$ abstraction operator. $Group$ replaces a set of nodes $V_{gr} \subset V$ in a valid PROV graph PG with a new abstract node, resulting in the modified graph PG'. The rewriting preserves the validity of the graph, in the sense made precise below, and it does not introduce any new relations into PG', which are not justified by existing PG relations. A formal account of this operator is given in Sec.3. A preliminary but more extended account of this work appears in our technical report [1].

Secondly, we present a simple policy model and language for controlling abstraction, based on the assumption that provenance *owners* want to control the disclosure of their provenance graphs to one or more *receivers*, with varying levels of trust (Sec.4). The model lets the owners associate a policy, pol, to a graph. Policy evaluation results in a *sensitivity* value $s(v, pol)$ being associated to each node v. Assuming, as in the Bell-Lapadula model [4], that a *clearance level* cl can be associated to each receiver, the nodes V_{gr} to be abstracted in PG according to pol are those for which $s(v, pol) > cl$.

Finally, we present the `ProvAbs` tool, which implements both $Group$ and the policy language. `ProvAbs` has been demonstrated on our confidentiality preservation use case, in the context of intelligence information exchange [5].

1.3 Related work

In addition to the Zoom and ProPub prototypes cited above, strands of research that are relevant to this work include (i) provenance-specific graph redaction, (ii) graph anonymization, and (ii) Provenance Access Control (PAC). Provenance redaction [6] employs a graph grammar technique to edit provenance that is expressed using the Open Provenance Model [7](a precursor to PROV), as well as a redaction policy language. The critical issue of ensuring that specific relationships are preserved, however, is addressed only informally in the paper, i.e., with no reference to OPM semantics.

Extensions to the relational data anynomization framework to graph data structures, specifically for social network data, have been developed [8,9,10]. The approach, involving randomly removing and adding arcs, will not work for PROV, however, as it would result in new, false dependencies. More relevantly, PAC is concerned with enforcing access control on parts of a provenance graph, in the context of secure provenance exchange. An analysis of the associated challenges [11] notes that provenance of data can be more sensitive than the data itself. In a similar setting, [12] accounts for the possibility of forgery of provenance by malicious users, and of collusion amongst users to reveal sensitive provenance to others. However, 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. Finally, our policy language is loosely related to an XACML-based policy language [13] the access control system for provenance, where path queries are used to specify target elements of the graph.

2 Essential PROV

We now introduce the PROV concepts that are required for the rest of the paper. The PROV data model[1] defines three types of sets: (i) Entities (En), i.e., data, documents; (ii) Activities (Act), which represent the execution of some process over a period of time, and (iii) Agents (Ag), i.e., humans, computing systems, software. The following set of core relations is also defined amongst these sets:

usage: $used \subseteq Act \times En$	generation: $genBy \subseteq En \times Act$
derivation: $wasDerivedFrom \subseteq En \times En$	association: $waw \subseteq Act \times Ag$
delegation: $abo \subseteq Ag \times Ag$	attribution: $wat \subseteq En \times Ag$

For simplicity and due to space constraints, in this paper we restrict our scope to just En, Act, and relations $used$ and $genBy$. The extension of this work to Agents and their relations (abo, wat), is available from our extended tech report [5]. The extension to other core relations such as $wasDerivedFrom$ is straightforward and will not be discussed here.

We denote instances of these relations as $genBy(e, a)$, $used(a, e)$, etc., where $e \in En, a \in Act$. Following common practice, we view a set I of such binary relation instances as a digraph $G = (V, E)$, where $V = En \cup Act$ and E is a set of labelled edges, and where $x \overset{r}{\leftarrow} y \in E$ iff $r(x, y) \in I$.[3] Finally, we denote the set of all such provenance graphs as $PG_{gu/ea}$, to indicate that they only contain $genBy$ and $used$ relations amongst En and Act nodes.

Fig. 1 shows an example of a $PG_{gu/ea}$ graph, where ovals and rectangles represent Entities, Activities, and Agents, respectively. The graph describes a document, `advice-report`, which was ultimately derived from twitter feeds captured at different times, through a series of query, consolidation, and analysis activities. The agents to whom the documents and activities are ascribed are omitted for simplicity. Note also that the nodes are decorated with user-defined properties, such as `Status`.

A set of formal constraints are defined on the PROV data model. These are described in the PROV-CONSTRAINTS document [14]. Two groups of constraints are relevant

[3] Conventionally, we orient these edges from right to left, to denote that the relation "points back to the past".

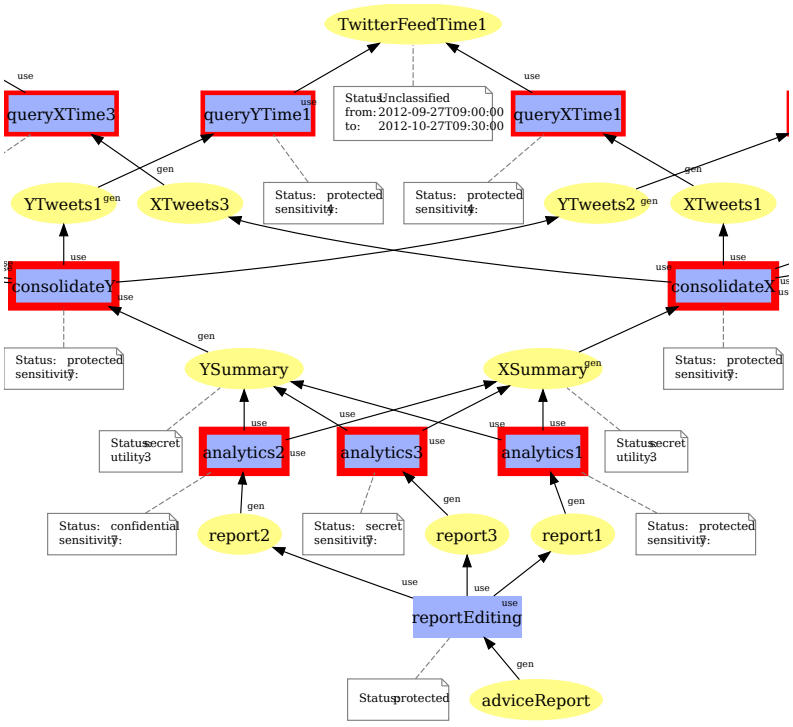

Fig. 1. Example provenance graph of a complex document production process. The `ProvAbs` model is designed to abstract some of the elements in the graph, for instance to avoid their disclosure. Coloured boxes denote `ProvAbs` sensitivity annotations, explained in Sec. 4.

here. The first (Constraint 50 — typing[4]) formalises the set-theoretical definitions of the relations given above. Additionally, Constraint 55[5] stipulates that entities and activities are disjoint: $En \cap Act = \emptyset$.

The second group concerns temporal ordering amongst events. PROV defines a set of instantaneous events which 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, such as generation and usage of an entity by an activity, attribution of an entity to an agent, and more. Optionally, events may be explicitly associated to PROV elements. In the following, we denote the start and end of an activity a by $startEv(a)$, $endEv(a)$, respectively, and the generation and usage events for an entity e and activity a with $genEv(genBy(e, a))$, $useEv(used(a, e))$, respectively (as mentioned, Agents are beyond the scope of this paper). PROV events form a preorder, which we denote $\preceq$. The relevant temporal constraints are expressed as follows.

[4] http://www.w3.org/TR/prov-constraints/#typing
[5] http://www.w3.org/TR/prov-constraints/#entity-activity-disjoint

– **C1: generation-generation-ordering (Constraint 39):** If an entity is generated by more than one activity, then the generation events must all be simultaneous. Let $e \in En, a_1, a_2 \in Act$, and let $genBy(e, a_1)$ and $genBy(e, a_2)$ hold. Then the following must hold:

$$genEv(genBy(e, a_1)) \preceq genEv(genBy(e, a_2)) \text{ and}$$
$$genEv(genBy(e, a_2)) \preceq genEv(genBy(e, a_1))$$

– **C2: generation-precedes-usage(Constraint 37):** A generation event for an entity must precede any usage event for that entity. Let $a \in Act$, $e \in En$, and let $used(a, e))$, $genBy(e, a)$ hold. Then:

$$genEv(genBy(e, a)) \preceq useEv(used(a, e))$$

– **C3: usage-within-activity (Constraint 33):** Any usage of $e \in En$ by some $a \in Act$ cannot precede the start of a and must precede the end of a. Let $used(a, e)$ hold. Then:

$$startEv(a) \preceq useEv(used(a, e)) \preceq endEv(a)$$

– **C4: generation-within-activity (Constraint 34):** The generation of e by a cannot precede the start of a and must precede the end of a. If $genBy(e, a)$, then:

$$startEv(a) \preceq genEv(genBy(e, a)) \preceq endEv(a)$$

A *valid* PROV graph is one that satisfies all the constraints defined in the PROV-CONSTR document [14]. Within our scope, a valid $PG_{gu/ea}$ graph is one that satisfies the constraints defined here.

3 Abstraction by grouping

Simple edits that can be applied to a graph to protect confidentiality of its content include removing individual nodes or edges. Alternatively, the node's identity can be changed, or the values associated to any of its properties can be removed. These straightforward edits are legal in PROV and they will not be discussed further. [6] We are instead concerned with edits that replace a group of nodes with a new abstract node.

3.1 Core concepts

To model this type of abstraction, we are going to define a *Group* operator which takes a graph $G = (V, E) \in PG_{gu/ea}$ and a subset $V_{gr} \subset V$ of its nodes, and produces a modified graph $G' = (V', E') \in PG_{gu/ea}$, where V_{gr} is replaced with a new single node. *Group* is closed under composition, thus allowing for further abstraction by repeated grouping (abstraction of abstraction). Let $v_{abs} \in V'$ be an abstract node in G'. We denote the set V_{gr} of nodes in G that it replaces by $source(v_{abs})$.

[6] Note that removing an arbitrary node may result in disconnected fragments of the graph, as in general one cannot simply add edges to reconnect the remaining nodes, unless those can be inferred from standard PROV constraints. For instance, if activity a is removed from the graph: $\{used(a, e_1), genBy(e_2, a)\}$, this results in two disconnected nodes e_1, e_2, because no relationship can be inferred between them from the original graph.

In order to understand the requirements for defining $Group$, consider the replacements in Fig. 2. On the left, nodes $V_{gr} = \{a_1, e_4, e_5\}$ are replaced with a new node e'. Simply using the original edges to connect the remaining nodes to e' leads to type constraint violations, namely for the new edges $e_1 \leftarrow e'$, $e_2 \leftarrow e'$, and thus to an invalid graph.

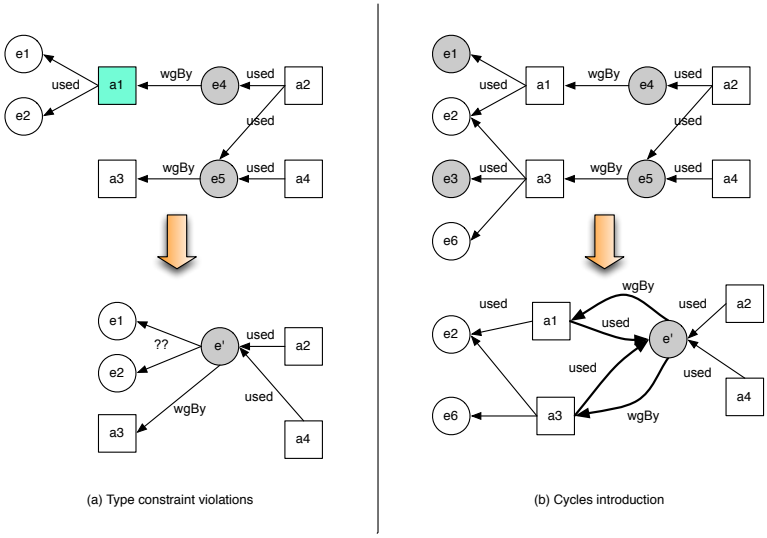

Fig. 2. Issues with naive replacement of groups of nodes.

Now consider Fig.2(b), where $V_{gr} = \{e_1, e_3, e_4, e_5\}$. In this case, the simple strategy or replacing V_{gr} with e' and reconnecting the remaining nodes leads to the two cycles: $\{genBy(e', a_1), used(a_1, e')\}$ and $\{genBy(e', a_3), used(a_3, e')\}$. Such cycles are legal, and in particular they are consistent with temporal constraints C1-C4 above. Indeed, it is easy to imagine a situation where an activity a first generates an entity e, and then makes use of e. For instance, a could be a programming artifact, i.e., an object that first instantiates a new object e, and then makes use of e. In this case, the event ordering is

$$startEv(a) \preceq genEv(e, a) \preceq useEv(a, e) \preceq endEv(a) \quad (1)$$

Yet, we argue that *introducing* new cycles during abstraction is undesirable. Intuitively, this is because cycles make stronger assumptions on the possible temporal ordering of events than those in the original graph, and thus are only representative of a restrictive class of graphs. To elaborate more precisely on this point, we first introduce new definitions of generation and usage events for an abstract node v_{abs}, from the corresponding events associated to $source(v_{abs})$. For this, consider the definition of generation and usage in [1]:

Generation is the *completion of production* of a new entity (Sec. 5.1.3).
Usage is the *beginning of utilizing* an entity (Sec. 5.1.4).

An abstract node v_{abs} can be thought of as representing the collection $source(v_{abs})$ in the new graph. Thus, its "generation" is logically defined as the completion of production of its source nodes, that is, its associated generation event should be the *latest* generation event from within its source. Note that associating a generation event to an abstract node requires the existence of a generating activity. Although this is not always provided as a result of abstraction by grouping, *Inference 7* in [1] ensures that such generating activity exists. Thus we can formally define generation for abstract nodes, as follows.

Definition 1 (Abstract node generation event). *Let $V_{gr} \in V$ and v_{abs} be a new abstract node, with $source(v_{abs}) = V_{gr}$ and generating activity a. Define:*

$$genEv(genBy(v_{abs}, a)) = \max_{e_i \in source(v_{abs})} genEv(genBy(e_i, a_i))$$

where a_i is the generating activity of e_i.

Symmetrically, we associate a usage event to v_{abs}, which is the *earliest* usage event for the nodes in $e_i \in source(v_{abs})$.

Definition 2 (Abstract node usage events). *Let $V_{gr} \in V$, $G' = (V', E')$ be the new abstract graph, and let $v_{abs} \in V'$ be a new abstract node. If there exists an activity $a \in V'$ such that $used(a, v_{abs})$ holds, then*

$$useEv(used(a, v_{abs})) = \min_{e_i \in source(v_{abs})} useEv(used(a_i, e_i))$$

where a_i is an activity that used e_i.

With these definitions in place, temporal constraint (1), which applies to simple usage-generation cycles in the graph, translates into the requirement that *every* entity $e_i \in source(v_{abs})$ be generated before *any* use of e_i. This constraint ties to each other the generation and usage time of the nodes that are abstracted. In the original graph, however, there is no such requirement: the generation of any entity is, in general, independent of that of others. This suggests that a new generation-usage cycle in the abstract graph adds constraints that are not present in the original graph, and should therefore be avoided. Note that ProPub [3] also insists on avoiding cycles, but the formal argument in support of this requirement does not appear to be clearly grounded in semantics.

To summarize, the requirements for $Group$ when G is rewritten into G' are: (i) no type constraint violations must occur in G', (ii) no new relationships that are not also present in G are introduced in G', and (iii) no new usage-generation cycles are introduced in G'.

3.2 Convexity, Closure, extensions, and replacement

Intuitively, the reason for cycles such as the one in Fig.2(b) is that set V_{gr} is 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, aimed at ensuring "convexity" and therefore acyclicity. This is defined as follows.

Definition 3 (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 is a directed path $v_i \rightsquigarrow v_j$ in G, let $V_{ij} \subset V$ be the set of all nodes in the path. The Path Closure of V_{gr} in G is*

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

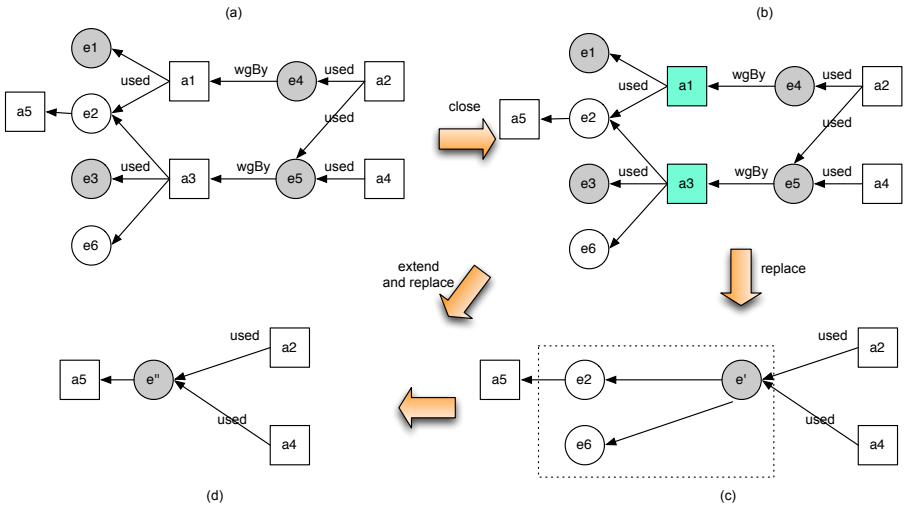

Fig. 3. Grouping by closure and extension.

Fig. 3(b) shows closure applied to the example of Fig.2, i.e. $pclos(\{e_1, e_3, e_4, e_5\}, G) = \{e_1, e_3, e_4, e_5, a_1, a_3\}$. The result of replacing this set with e' is shown in (c). However, while this solves the cycle problem, the graph still violates type constraints, namely on the new edges $e_2 \leftarrow e'$ and $e_6 \leftarrow e'$. 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 (d) is valid.

To preserve validity in the general case, we are going to first extend the closure in (b) to include e-nodes e_2, e_6, and then replace the resulting set with e'' (the "extend and replace" arrow from (b) to (d) in the figure). Following this approach, *Group* is defined as the 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 4 (*extend*). *Let* $G = (V, E) \in PG_{gu/ea}$, $t \in \{En, Act\}$.

$$\begin{aligned} extend(V_{gr}, G, t) = \quad & \{v' | (v, v') \in E \wedge v \in V_{gr} \wedge type(v') = t)\} \cup \\ & \{v | (v', v) \in E \wedge v \in V_{gr} \wedge type(v') = t)\} \cup \quad V_{gr} \end{aligned}$$

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 sink nodes in $extend(V_{gr}, G, t)$ are of type t by construction.

Replacement. Let $G = (V, E)$, $V'_{gr} \subset V$ be obtained using $extend$, and let v_{new} be a new node symbol 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, as follows. Let $\vartheta_{out}(V'_{gr})$, $\vartheta_{in}(V'_{gr})$, and $\vartheta_{int}(V'_{gr})$ denote the set of arcs of G leading out of V'_{gr}, leading into V'_{gr}, Each arc $(v', v) \in \vartheta_{out}(V'_{gr})$ is replaced with a new arc (v_{new}, v), and each arc $(v, v') \in \vartheta_{in}(V'_{gr})$ is replaced with a new arc (v, v_{new}), both of the same relation type. Arcs in $\vartheta_{int}(V'_{gr})$ are removed along with the nodes in V'_{gr}. Indeed, all sink nodes in V'_{gr} are of type t as noted above, and so is v_{new} by construction. Thus, sink nodes are replaced by a node v_{new} of the same type. Since the arcs have the same type as those they replace, it follows that $replace$ preserves type correctness. It is also easy to verify that each new edge in G' can be mapped to an existing edge in G (proof omitted).

Definition 5 **(Replace).** $replace(V_{gr}, v_{new}, G) = (V', E')$, *where:*

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

3.3 *T-grouping*

We can now define $Group$ as a composition of closure, extensions, and replacement. In general, nodes in V_{gr} can be either En or Act. It is necessary to specify the type of the replacement node, as this may lead to different results. To make this explicit, we denote the operator by **t-grouping** (i.e, **e-grouping** or **a-grouping**, respectively). In the next section, we clarify how user-defined policies are used to control the application of **t-grouping** to a provenance graph.

Definition 6 **(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:*

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

Note sink nodes in the closure are homogeneous and are replaced by a node of the same type t. This satisfies the necessary condition for $replace$ to perform correctly. Fig. 4(a-1, a-2) illustrates $Group(G, \{e_4, a_2\}, v_{new}, Act)$, while Fig. 4(e-1, e-2,e-3) shows $Group(G, \{e_4, a_2\}, v_{new}, En)$. Note that a new pattern arises in the case of *e-grouping* as shown in Fig. 4(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. 4(e-3), involving two generation events

for the new entity e_N. Although this is a valid pattern, the two generation events must be simultaneous by C1 above. The intuitive interpretation for this pattern is that each of the two activities generated one entity in $source(e_N)$, and that the abstraction makes these two events indistinguishable. Formally, nothing further needs to be done to the graph. However one can restore the more natural pattern whereby one single generation event is recorded for e_N, by propagating the grouping to the set of generating activities. In the example, this leads to the graph in Fig. 4(e-3).

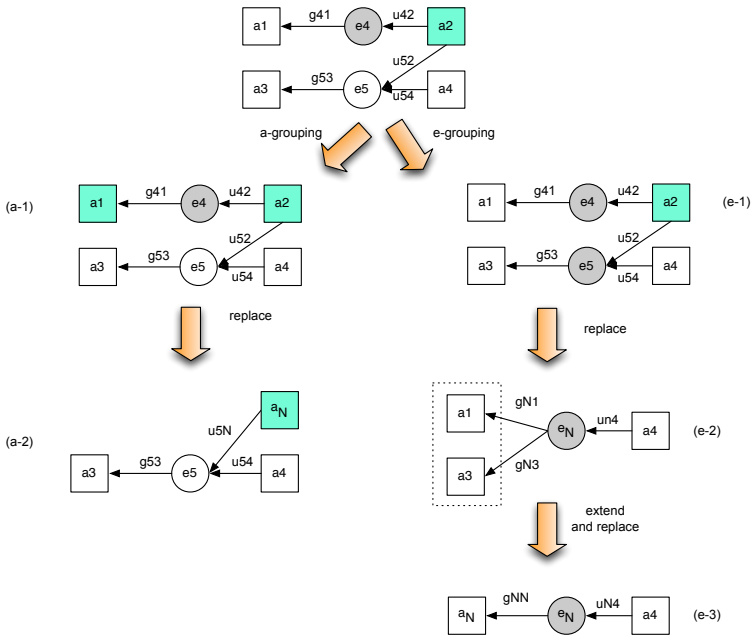

Fig. 4. e-grouping and a-grouping

4 Policy model

Having outlined the grouping operator, we now present a simple policy language to let users specify one or more grouping sets V_{gr} for abstraction. We refer to these users as Policy Setters (PS). Our approach consists of two phases. The first phase involves annotating each node n with a *sensitivity* value $s(n)$ and/or a *utility* value $u(n)$. These annotations are independent of any intended receiver of the abstracted graph. In the second phase, a grouping set V_{gr} is generated for a specific receiver r, denoted $V_{gr}(r)$ for clarity. We assume, as in Bell-Lapadula [4], that a pre-defined clearance level $cl(r)$ is associated with r. The nodes to be abstracted are simply those with sensitivity higher than $cl(r)$: $V_{gr}(r) = \{v \in V | s(n) \geq cl(r)\}$.

A policy is a sequence of rules. Each rule (i) identifies a set of nodes, and (ii) assigns a sensitivity to each of those nodes. Node selection is achieved using a simple form of

path expressions on the graph, combined with filter conditions. Keeping simplicity of use by non-expert PS in mind, we have chosen a simplified fragment of regular path expressions on graphs [15]. The example rules in Fig. 5 apply to the graph in Fig. 1:

```
list classifications  [Unclassified, Classified, Protected, Secret];
for all (act used data)
   where (data.Status >= Secret in classifications (def true)) setSensitivity(act, 7);
for all (process used data)
   where (data descendantOf d14)) setSensitivity(data, 10);
```

Fig. 5. Example Policy rules

The rules are executed in sequence. `List` declares a domain-specific *ordered* enumeration of constants, called `Classifications`. The path expression in the first command is a simple pattern where `act` and `data` are variables, and `used` is the $used$ relation. The pattern is then matched against the graph and the variables are bound to nodes. The filter condition predicates on the values of properties associated to the nodes. Here the value of `data.Status` is expected to be one of the constants in the `classification` list. This predicate selects all nodes with value *at least* `Secret` in the ordered list. The activity nodes that satisfy the conditions have their sensitivity set to 7.[7] Rather than allowing arbitrary regular path expressions in the language, we expose specific traversal operators. One example is *descendantOf*, which returns all nodes reachable from a given start node. An example of its use is the second rule above. Rule evaluation binds variables `process` and `data` to activity and entity nodes a, e, respectively, such that $used(a, e)$ holds and e is any node that is reachable from node with id `d14` (a constant value).

Utility is the counterpart to sensitivity. It denotes 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. Recall from our earlier example that grouping may remove non-selected nodes in order to preserve validity, a possibly undesirable side-effect. The utility values associated to different nodes are used to quantify such loss of utility. Let $V_{ret} = V \backslash V_{gr}$ be the set of nodes not intended for grouping, and $V'_{ret} \subset V_{ret}$ the nodes which were in fact retained after grouping. The residual utility is simply

$$RU_V = \frac{\sum_{n \in V'_{ret}} u(n)}{\sum_{n \in V_{ret}} u(n)} \tag{2}$$

which is maximized for $V'_{ret} = V_{ret}$. Policy setters who experiment with different policy rules, i.e., using a test set of provenance graphs, may use RU_V as a quantitative indicator of utility loss associated with a given policy and receiver.

4.1 `ProvAbs` tool

The Provenance Abstraction Model is implemented as part of a project involving confidentiality protection for provenance. The main purpose of the `ProvAbs` tool is to let a PS explore partial disclosure options, by experimenting with various policy settings and

[7] A default value can be specified, i.e. for the cases where a `data` node has no `Status` property, or the property has no value.

clearance level thresholds. Users may load a graph in PROV-N format [16] and either specify a policy interactively, or load a pre-defined policy file. The output consists of a graphical depiction of the graph, annotated with its sensitivity values (these are the coloured boxes in Fig. 1), as well as the final abstract version of the graph. The residual utility value (2) is also returned. Provenance graphs are stored in the Neo4J graph database (`neo4j.org`). Policy expressions are evaluated using a combination of the Neo4J Traverse API and Cypher queries. `ProvAbs` and its documentation are publicly available.[8]

5 Summary

In this paper we have presented a Provenance Abstraction Model (PAM) and its implementation, `ProvAbs`. PAM is based on a *Group* operator, which replaces a set of nodes in a PROV graph with a new abstract node while preserving the validity of the graph. A simple notion of convexity of the set of nodes to be replaced ensures that the rewriting does not introduce new cycles. Due to space limitations, the scope of this paper is limited to $PG_{gu/ea}$ graphs, which only include generation, usage relations on Activity and Entity nodes. A more comprehensive model, including its extension to Agents, can be found in our report [5]. Encouraged by this initial study, we are now developing a more comprehensive model of abstraction that accounts for larger fragments of PROV — a complex specification in its own right.

References

1. Moreau, L., Missier, P., Belhajjame, K., B'Far, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., *et al.*: PROV-DM: The PROV Data Model. Technical report, World Wide Web Consortium (2012)
2. Biton, O., Boulakia, S.C., Davidson, S.B., Hara, C.S.: Querying and Managing Provenance through User Views in Scientific Workflows. In: ICDE. (2008) 1072–1081
3. Dey, S., Zinn, D., Ludäscher, B.: ProPub: Towards a Declarative Approach for Publishing Customized, Policy-Aware Provenance. In: Procs. SSDBM. Volume 6809 of LNCS. Springer (2011) 225–243
4. Bell, D.: The bell-lapadula model. Journal of computer security **4**(2) (1996) 3
5. Missier, P., Gamble, C., Bryans, J.: Provenance graph abstraction by node grouping. Technical report, Newcastle University (2013)
6. Cadenhead, T., Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.: Transforming provenance using redaction. In: Procs. 16th ACM Symp. on Access control models and technologies. SACMAT '11, New York, NY, USA, ACM (2011) 93–102
7. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., *et al*: The Open Provenance Model — Core Specification (v1.1). Future Generation Computer Systems **7**(21) (2011) 743–756
8. Zheleva, E., Getoor, L.: Preserving the Privacy of Sensitive Relationships in Graph Data. In: Privacy, Security, and Trust in KDD. Volume 4890 of LNCS. Springer (2008) 153–171
9. Bhagat, S., Cormode, G., Krishnamurthy, B., Srivastava, D.: Class-based graph anonymization for social network data. Proc. VLDB Endow. **2**(1) (August 2009) 766–777
10. Liu, K., Terzi, E.: Towards identity anonymization on graphs. In: Procs. SIGMOD, New York, NY, USA, ACM (2008) 93–106
11. Braun, U., Shinnar, A., Seltzer, M.: Securing provenance. In: Proceedings of the 3rd conference on Hot topics in security, Berkeley, CA, USA, USENIX Association (2008) 4:1—-4:5
12. Hasan, R., Sion, R., Winslett, M.: Introducing secure provenance: problems and challenges. In: Procs 2007 ACM workshop on Storage security and survivability. StorageSS '07, New York, NY, USA, ACM (2007) 13–18
13. Cadenhead, T., Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.: A language for provenance access control. In: Procs. ACM conference on Data and application security and privacy. CODASPY '11, New York, NY, USA, ACM (2011) 133–144
14. Cheney, J., Missier, P., Moreau, L.: Constraints of the Provenance Data Model. Technical report (2012)
15. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. SIAM Journal on Computing **24**(6) (1995) 1235–1258
16. Moreau, L., Missier, P., Cheney, J., Soiland-Reyes, S.: PROV-N: The Provenance Notation. Technical report (2012)

[8] `http://bit.ly/1dxg9X1`.