1 Introduction

The Open Provenance Model (OPM) [20] provides a small, extensible core for representing and exchanging provenance information in a technology-neutral manner. By design, OPM is a least common denominator, leaving aside certain aspects, including how to query provenance information. Similarly, OPM comes with a set of inference rules (e.g., for transitively closing some relations, or for stating temporal constraints that are implied by provenance assertions), but as pointed out by [15], the temporal semantics of OPM graphs is only partially defined in [20], leading to ambiguous or incompletely specified situations.

Example 1. Consider the provenance graph in Figure 1. In OPM, it shows a (data) artifact $x$ that was used by process $P$, and another artifact $y$ that was-generated-by (short: gen-by) $P$. In the terminology used in the rest of the paper, we say that the graph records a “read” and a “write” observable, denoted $x \xrightleftharpoons{\text{read}} P$ and $P \xrightleftharpoons{\text{write}} y$, respectively. Given this information, it may seem natural to ask: can we then infer that (1) $y$ was-derived-from $x$? Or that $t_{\text{read}} < t_{\text{write}}$ holds?

![Figure 1: Provenance graph showing artifacts $x$ and $y$ and process $P$.](image)

Fig. 1: Do the observables $x \xrightarrow{\text{read}} P$ and $P \xrightarrow{\text{write}} y$ imply that $y$ was-derived-from $x$? Or that $t_{\text{read}} < t_{\text{write}}$ holds?

Example 2. Consider the provenance graph in Figure 2, asserting that $z$ was-derived-from $x$. Does $t_{\text{read}2} < t_{\text{write}2}$ follow? As before, if no further information is available, we cannot infer either proposition: we simply don’t know whether the path $x.P_a.y.P_b.z$ or the path $x.P_a.z$ (or yet another one, not included in the figure) are the reason for the was-derived-from edge in Fig. 2. The OPM semantics also handles this case correctly and does not imply that $t_{\text{read}i} < t_{\text{write}j}$ (for any $i \leq j$). On the other hand, OPM also does not provide a means to specify when such inferences would indeed be correct, e.g., in the common case where the result of a write is in fact directly dependent on an earlier read.

When employing OPM as a model for provenance recorded by a scientific workflow system [8], another limitation becomes apparent: OPM only deals with retroactive provenance (the usual data lineage captured in a trace graph $T$), but not with so-called prospective provenance, i.e., workflow specifications $W$, which are recipes for future (and past) derivations [18]. These were out of scope for OPM and, apparently, are also out of scope for current W3C standardization efforts [21].

On the other hand, it is easy to see that distinguishing between traces $T$ and workflows $W$ (and then interpreting the former as instances of the latter) can provide valuable information and additional functionality for provenance applications.

---

*a* UC Davis, (scdey | skoehler | ludaesch)@ucdavis.edu  
† Gonzaga University, bowers@gonzaga.edu  
1If the computation $P$ is a function or service call, then $P$ indeed first consumes all inputs, then produces all outputs. This assumption is often correct, but a process $P$ is not necessarily limited to such strict behavior and may interleave read/write events in many ways [13, 17].

2This “near-oxymoron” captures a practically useful notion: When a scientist is asked to explain how a certain result was obtained, in the absence of runtime traces, he can point to the script/workflow that was used to generate the data products; so workflows are provenance, too.
Example 3. Consider the graphs in Figure 3: executing workflow $W$, might have produced the trace $T$. In order to validate $T$'s structure, i.e., to check whether $T$ can be an instance of $W$, we link its nodes and edges to $W$: e.g., edges $x \xrightarrow{\text{read}} a$ and $a \xrightarrow{\text{write}} y$ in $T$ (data $x$ was read and data $y$ was written by process invocation $a$), have corresponding edges $X \xrightarrow{\text{in}} A$ and $A \xrightarrow{\text{out}} Y$ in $W$, linking data containers $X$ and $Y$ to the process (or: actor) $A$. Clearly, in order to validate $T$'s structure, we have to have a representation of the workflow structure $W$ in the first place.

However, even if $T$ is structurally valid w.r.t. $W$, other (here: temporal) inconsistencies may arise: The cycle in $X$ written cern: it simply means that $W$'s structure, we have to have a representation of the workflow structure $W$ in the first place.

Consider the graphs in Figure 3: executing Example 3.

Overview and Contributions. We propose logic rules as a formal foundation for graph-based, temporal models of provenance, for querying provenance graphs (traces), and for reasoning about traces and their connection to the workflows that generated them. In particular, we argue that Datalog provides a “lingua franca” for provenance:

1. We adopt a semistructured data model, i.e., graphs with labeled edges $x \rightarrow y$, as a uniform representation of all provenance information, i.e., traces (à la OPM) and associated workflows of which they are instances. This allows us to employ regular path queries [3] as a convenient “macro-language” for concisely expressing generalized reachability queries on traces and workflows. By representing schema-level information (workflows) and instance-level information (traces) together in a single model, structural constraints can be expressed and checked easily using Datalog rules.

2. We propose to use integrity constraints, i.e., rules of the form $\text{false}_{\text{op}}(\overline{Y}) \rightarrow \overline{\text{denial}}(\overline{X})$ as a way to express provenance semantics. Workflow systems differ in their models of computation (MoCs) and thus make different assumptions about how workflow components (a.k.a. actors, processors, modules, etc.) work, i.e., whether,

And in which way, they can be stateful; how they consume their inputs, produce their outputs; and so on. As a result, different systems use different models of provenance (MoPs), with different temporal semantics. Thus, instead of “hard-wiring” a fixed temporal semantics to a particular graph-based MoP, we again use logic constraints to obtain a “customizable” temporal semantics.

3) We illustrate this concept by providing firing constraints at the workflow level, which induce temporal constraints $\leq_t$ at the level of traces (cf. Figure 4). These temporal-constraint generating rules can be chosen to conform to the temporal axioms in [15], or to accommodate a different temporal semantics, as implied by the MoC of a specific workflow or workflow system.

Due to limited space, we only illustrate some of the many ways in which logic rules and Datalog can be harnessed for provenance querying and reasoning: a detailed exposition is reserved for a long version of this paper. In the remainder, we give an overview of our approach and provide a few illustrative examples; additional examples and rules can be found in the appendix.

2 Unified Provenance Model

We would like to accommodate trace-level provenance information, schema-level information (workflow specifications), and temporal information in a single, uniform representation. To this end, we employ an underlying semistructured data model, which consists of labeled, directed graphs of the form $G = (V, E, L)$, with vertices $V$, labels $L$, and labeled edges $E \subseteq V \times L \times V$. In the following, we view workflows $W$ and traces $T$ as subgraphs of $G$. Similarly, our temporal model will consist of labeled edges (modeling one or more “before” relations).
Workflows. A workflow $W = (V_W,E_W,L_W)$ is a labeled graph whose nodes $V_W = C \cup P$ are data containers $C$ and processes $P$. Processes are computational entities (often consisting of smaller internal steps, a.k.a. invocations or firings) that can send and receive data. Containers represent structures, e.g., FIFO queues, that hold data during the communication between processes. In a workflow $W$, edges $E_W = E_{in} \cup E_{out}$ are either input edges $E_{in} \subseteq C \times \{\text{in}\} \times P$, or output edges $E_{out} \subseteq P \times \{\text{out}\} \times C$, so $L_W = \{\text{in}, \text{out}\}$. We also write $\text{in}(C,P)$ and $\text{out}(P,C)$ to denote edges $C \xrightarrow{\text{in}} P$ and $P \xrightarrow{\text{out}} C$, respectively. The former means process $P$ can read data artifacts from container $C$, while the latter means that process $P$ can write data artifacts to container $C$.

Traces. A trace $T = (V_T,E_T,L_T)$ is a labeled graph whose nodes $V_T = D \cup I$ are data artifacts $D$ or process invocations $I$; edges $E_T = E_{read} \cup E_{write} \cup E_{diff}$ are read edges $E_{read} = D \times \{\text{read}\} \times I$, write edges $E_{write} = I \times \{\text{write}\} \times D$, or derived-from edges $E_{diff} = D \times \{\text{diff}\} \times D$, so $L_T = \{\text{read},\text{write},\text{diff}\}$. The link between traces and workflows is established through homomorphisms:

**Definition 1.** Let $G = (V,E,L)$ and $G' = (V',E',L')$ be labeled graphs, and let $h = (h_1,h_2)$ be a pair of mappings $h_1 : V \rightarrow V'$ and $h_2 : L \rightarrow L'$. Then $h$ is a homomorphism from $G$ to $G'$ if

\[(x \rightarrow y) \in E \text{ implies } (h_1(x) \xrightarrow{h_2(f)} h_1(y)) \in E'.\]

The functions $h_1$ and $h_2$ map nodes and labels from $G$ to corresponding ones in $G'$. The model described here associates read and write edges in $T$ with in and out edges in $W$, so $h_2 = \{\text{read} \rightarrow \text{in}, \text{write} \rightarrow \text{out}\}$. The mapping $h_1$ is usually given as part of a trace, i.e., when testing whether $T$ is valid w.r.t. a workflow $W$, we do not have to search for $h$. Instead, traces needing validation already have corresponding workflow annotations embedded within them: We use mappings $\text{cont} : D \rightarrow C$ and $\text{proc} : I \rightarrow P$ to associate data items and process invocations with data containers and processes, respectively. The following rules derive false iff trace $T$ with workflow mappings cont and proc are not a homomorphism:

\[
\begin{align*}
\text{falseHom}(\text{read}(D,I),\text{in}(C,P)) & :\rightarrow \text{read}(D,I), \text{cont}(D,C), \text{proc}(I,P), \neg \text{in}(C,P).
\text{falseHom}(\text{write}(I,D),\text{out}(P,C)) & :\rightarrow \text{write}(I,D), \text{cont}(D,C), \text{proc}(I,P), \neg \text{out}(P,C).
\end{align*}
\]

Thus, if falseHom is empty, trace $T$ is structurally-valid w.r.t. workflow $W$. Additional structural constraints for traces can be easily defined in a similar manner:

A write-conflict occurs when a data artifact has multiple incoming write edges; a type-conflict occurs when edges link nodes of the wrong type (e.g., directly linking invocations, instead of going through data nodes). Rules for checking such constraints are listed in Appendix C.

Temporal Model. We obtain a temporal semantics on top of the graph-based model by using rules to define a “before” relation (e.g., $\preceq_a$ and $\preceq_d$ in Fig. 4) amongst events. During workflow execution, the following events are observed: When a process is executed an invocation event is recorded, when a data artifact is read by an invocation a read event is recorded, and when an invocation writes a data artifact a write event is recorded. In the temporal model we constrain the order of events: $bf(E_1,E_2)$ means that $E_1$ happened before $E_2$ and $bf(E_1,E_2)$ means that $E_1$ happened before or simultaneously with $E_2$.

A data-constraint between out-edges $P_A \xrightarrow{\text{out}} C_T$ and in-edges $C_T \xrightarrow{\text{in}} P_B$ of a data container $C_T$ at the workflow-level, implies an obvious temporal constraint at the trace-level: A write (creation) of data $y$ must come before any read of $y$ (also see Figure 4):

$bf(\text{write}(I,D), \text{read}(D,J)) :\rightarrow \text{write}(I,D), \text{read}(D,J)$.

Read, write, and derived-from edges in $T$ capture dataflow. This information gives rise to a temporal order (“flow-time”) among events. We infer this temporal order and create the corresponding temporal relations $bf$ and $dfs$ using the rules described here (and in the appendix). We make (safe) use of Skolem functions, e.g., to create unique identifiers, and to reify edges as nodes of a temporal structure: e.g., the (unspecified) execution time of an invocation is represented by a term $\text{invoc}(I)$.

Before an invocation reads data, it must have started:

$dfs(\text{invoc}(I), \text{read}(D,I)) :\rightarrow \text{read}(D,I)$.

Similarly, a data artifact could not have been written after an invocation has been completed:

$dfs(\text{write}(I,D), \text{invoc}(I)) :\rightarrow \text{write}(I,D)$.

When a data artifact is written by a process invocation and read by another invocation, the former must not have started after the latter has completed:

$dfs(\text{invoc}(I), \text{invoc}(J)) :\rightarrow \text{write}(I,D), \text{read}(D,J)$.

When a data artifact is derived from another data artifact, the latter must have been written before the former:

$bf(\text{write}(I,Y), \text{write}(I,X)) :\rightarrow \text{dfs}(X,Y), \text{write}(I,X), \text{write}(I,Y)$.

A firing-constraint between in-edges $C_x \xrightarrow{\text{in}} P_A$ and out-edges $P_A \xrightarrow{\text{out}} C_T$ of a process $P_A$ is defined via a relation $\text{fc}(C_x,P_A,C_T)$. This constraint ensures that any invocation of process $P_A$ that reads data from container $C_x$ and that writes into container $C_T$ has an associated temporal constraint: the invocation output depends on the read input.

If a user-defined constraint is provided by $\text{fc}(X,B,Z)$ as shown in Figure 4, then the $bf(\text{read}(X,b), \text{write}(b,z))$ temporal relation at the trace-level can be inferred:

$bf(\text{read}(X,J), \text{write}(I,Y)) :\rightarrow \text{fc}(C_1,P,C_2), \text{read}(X,J), \text{proc}(I,P), \text{write}(I,Y), \text{cont}(X,C_1), \text{cont}(Y,C_2)$.
3 Conclusions

The theory and practice of provenance are not always as well aligned as one may wish for.\footnote{In theory, there is no difference between theory and practice. But in practice, there is.} The DBLP (databases & programming languages) community, among others, has been advancing our understanding of fundamental principles of provenance, and notions such as Why, How, and Where provenance [2, 6], provenance semirings [11], provenance as proof-trees, dependencies, program slicing [5], relationships to causal reasoning [19, 4], etc. are slowly becoming more widely known and understood better. At the same time, the many practical applications of provenance have led practitioners and system developers to move ahead rapidly with “provenance-enabling” their systems, proposing models, languages, and even W3C standards. As a contribution to further grow the connections between theory and practice of provenance [1], we have proposed to use logic rules, and Datalog in particular for querying provenance, checking structural constraints (e.g., whether a trace is valid, i.e., homomorph to its associated workflow), and specifying temporal constraints.

Datalog seems particularly well-suited as a “lingua franca” and bridge between theory and practice: On the one hand, there is a rich body of research and formal results about the complexity and expressiveness of Datalog and numerous fragments or extensions [7]. Queries on labeled graphs are well supported, e.g., regular path queries have a direct encoding in Datalog, and theoretical results on query containment and view-based query processing [3] can be exploited in various ways. Similarly, Datalog variants such as Statelog [16] or Datalog-LITE [10] provide means to naturally express temporal queries over provenance graphs. From a practitioner’s point of view, Datalog is also attractive: powerful Datalog engines are available for experimentation with rule sets (e.g., to test and compare different provenance semantics, to query and debug programs [14] etc.), and can be used to deploy provenance querying and reasoning systems. Datalog prototypes for our approach are under development and can be demonstrated at the workshop.

Acknowledgements. Work supported in part by NSF award OCI-0722079 and DOE award DE-FC02-07ER25811.

References

A Hamming Workflow Variants

To illustrate the earlier definitions of this paper, we use two variants of a workflow to compute the Hamming numbers\(^6\) \(H = \{2^i \cdot 3^j \cdot 5^k | i, j, k \geq 0\}\) incrementally, i.e., as an ordered sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... Two workflow variants \(H_1\) and \(H_3\) are shown in Fig. 5. Note that the workflow graphs contain the same nodes (processes and containers), but are wired slightly differently (as it turns out, this makes a big difference). The data containers \(Q_i\) are queues (FIFO buffers); \(Q_8\) is the distinguished output, where the Hamming numbers will appear in the correct order. \(M_1\) and \(M_2\) are merge actors, i.e., processes which take two ordered input sequences and merge them into an ordered output sequence. If presented with the same item in both streams, the output stream will only contain one copy of the element, so duplicates are removed. The actors \(X_2, X_3, \) and \(X_5\) multiply their inputs with 2, 3, and 5, respectively. Last not least, the sample-delay actors \(S_2, S_3, S_5\) are used “to prime the pump”: initially (i.e., before reading any input), they output the number 1 to get the loop(s) going. Subsequently, they simply output whatever they received as an input. By design, the Hamming workflows \(H_1\) and \(H_3\) define an infinite output stream, i.e., the processes “run forever”.

Fig. 5: Workflow variants \(H_1, H_3\); output queue is \(Q_8\).

A.1 Structural Validity

Figure 6 shows a (partial) trace \(T_H\), i.e., for computing the Hamming numbers \(n \leq 15\).\(^7\) In order to test if a trace \(T\) is structurally-valid w.r.t. a workflow \(W\), we check if there is a homomorphism from \(T\) to \(W\). Here, a homomorphism between \(T_H\), in Figure 6 and the 1-loop variant \(H_1\) of the Hamming workflow in Figure 5(a) can be established. However, when attempting to find a homomorphism between the same trace and the 3-loop variant \(H_3\) in Figure 5(b), corresponding in-edges cannot be found for the dashed (red) read-edges in Figure 6. Note that the “bad” (missing) edges can be found automatically with the Datalog (denial) rules for false\(_{Hom}\) in Section 2.

A.2 User-Defined Provenance Queries

Our unified provenance model can easily be queried further using Datalog. For example, a user might want to know the lineage of a particular Hamming number (i.e., which other Hamming numbers “went into it”), or how many duplicates were derived in their particular workflow variant, etc. The dependencies between data items can be obtained by focusing on the read/write observables of certain processes \(P\):

\[q(D1,P,D2) = \text{read}(D1,I), \text{write}(I,D2), \text{proc}(I,P), \text{focus}(P).\]

Here, focus is a user-defined predicate to limit query answers to processes of interest, e.g., we may focus on

---

\(^6\)a.k.a. regular numbers; see [9, 12] for details

\(^7\)Fig. 7 shows user-defined trace-views for \(n \leq 1000\).
Fig. 7: User-defined provenance for Hamming numbers up to 1000: (a) for $H_1$ (“Fish”) and (b) for $H_5$ (“Sail”)

$x_2$, $x_3$, and $x_5$. Thus, tuples in the answer relation $q$ can be viewed as edges $d_1 \xrightarrow{\rho} d_2$, linking data items to each other, with the label $p$ denoting the process (multiplication factor) involved. Figure 7 shows the resulting graph structure\(^8\) for a trace containing the computation of Hamming numbers up to 1000. Edge labels are represented via colors (here: green, red, and blue are used to represent labels “$x_2$”, “$x_3$”, and “$x_5$”, respectively).

One can clearly see on the in-degree of nodes that in Fig. 7(a) many Hamming numbers are produced in multiple ways, i.e., the custom-provenance graph is a DAG. In contrast, Hamming numbers in $H_3$ are produced by one path only without unnecessary duplicates as can be seen in Fig. 7(b), i.e., this graph is a tree.

B Regular Path Queries

Our unified provenance model in Section 2 is based on a semistructured data model, i.e., all information is represented using a labeled, directed graph $G = (V, E, L)$. Edges in this graph can be represented as triples $g(x, \ell, y)$, where nodes $x, y \in V$ are connected via an $\ell$-labeled edge.

A standard method to query such graphs is using regular path queries (RPQ) [3]. Let $\Sigma = \{\ell_1, \ldots, \ell_n\}$ be a set of base labels (here: $\Sigma = L$). Regular expressions over $\Sigma$ are built in the usual way: if $R, R'$ are regular expressions, then so are $R \mid R'$ (alternation) and $R \cdot R'$ (concatenation), as well as $R^+$ (transitive closure), $R^*$ (reflexive transitive closure), etc. A number of additional operations can be defined, e.g., $R^+$ (optional edge), $\cdot$ (any label), $R^c$ (complement of $R$), etc. On labeled graphs, also $R^-$ (inverse edge direction) is common. It is easy to see that RPQs can be easily expressed in Datalog: e.g., for $R \cdot R'$, $R \mid R'$, and $R^+$, the rules are (similarly for other expressions):

\[
\begin{align*}
g(X, R_1 \text{conc} R_2 Y) & : \text{conc}(R_1 R_2, R_1 \text{conc} R_2), \\
g(X, R_1, Z). & \\
g(Z, R_2 Y). & \\
g(X, R_1 \text{or} R_2, Y) & : \text{or}(R_1, R_2 \text{or} R_2), \\
g(X, R_1 Y). & \\
g(X, R_1 \text{or} R_2 Y) & : \text{or}(R_2, R_1 \text{or} R_2), \\
g(X, R_2 Y). & \\
g(X, R + Y) : \text{plus}(R, R + Y), \\
g(X, R, Y). & \\
g(X, R + Y) : \text{plus}(R, R + Y), \\
g(X, R, Z). & \\
g(Z, R + Y). & 
\end{align*}
\]

The relations conc, or, and plus in the body can be defined, e.g., using Skolem functions as follows:

\[
\begin{align*}
\text{conc}(R_1, R_2, R_3) & : R_3 = f_\text{conc}(R_1, R_2), \\
\text{or}(R_1, R_2, R_3) & : R_3 = f_\text{or}(R_1, R_2), \\
\text{plus}(R_1, R_2) & : R_2 = f_\text{plus}(R_1).
\end{align*}
\]

\(^8\)Nodes are not intended to be readable, but if desired can be zoomed-into in the PDF version.
However, these rules are not safe under the standard bottom-up Datalog evaluation procedure\(^9\), so instead we create a finite set of facts, consisting only of the subexpressions of a given user query \(R\). For example consider the RPQ expression \(R = (x_2^+ \cdot x_3^+ \cdot x_5^+)\). The bold, red relation in Fig. 8 shows the answer to the query:
\[
\text{ans}(X,R,Y) : \neg g(X,R,Y), X \leq 30, Y \leq 30, \text{derived}(R).
\]
where derived is defined as:
\[
\text{derived}(R) : \neg \text{conc}(_-R).
\]
\[
\text{derived}(R) : \neg \text{or}(_-R).
\]
\[
\text{derived}(R) : \neg \text{plus}(_-R).
\]
and where the facts in conc and plus are obtained via the subexpressions of the top-level user query:
\[
\text{conc}((x_2^+), (x_3^+), (x_5^+)).
\]
\[
\text{conc}('x_2^+', 'x_3^+', '(x_2^+).(x_3^+)').
\]
\[
\text{plus}('x_2', 'x_2^+').
\]
\[
\text{plus}('x_3', 'x_3^+').
\]
\[
\text{plus}('x_5', 'x_5^+').
\]

**Hamming Workflow Revisited.** Consider again the Hamming workflow variants in Fig. 5 and the resulting user-defined provenance graphs in Fig. 7. It is easy to see that the “Sail” variant in Fig. 7(b) produces only paths that match the RPQ \(x_2^+ \cdot x_3^+ \cdot x_5^+\), whereas the more redundant “Fish” variant in Fig. 7(a) contains paths that match \((x_2 | x_3 | x_5)^+\). In this way, RPQs can be used to query and inspect the structure of provenance graphs.

Note that via the construction outlined above, we can view RPQs as syntactic sugar for Datalog queries and freely mix them with other Datalog rules, including temporal queries in the style of Datalog-LITE [10].

\(^9\)The usual top-down SLD-resolution of Prolog can handle such rules, however would in turn not be safe for the recursive RPQ rules.

\[C\] Further Examples

In the remainder, we present some further examples and rules, to illustrate our definitions in earlier sections. In our integrity constraint rules we employ the following auxiliary view for transitive dependencies:
\[
\text{dep}(X,Y) : \text{read}(X,Y).
\]
\[
\text{dep}(X,Y) : \text{write}(X,Y).
\]
\[
\text{tcdep}(X,Y) : \neg \text{dep}(X,Y).
\]
\[
\text{cycle}(X,Y) : \neg \text{tcdep}(X,Y), \text{tcdep}(Y,X), \neg X=Y.
\]

**C.1 Structural Integrity Constraints**

We compute the processes involved in a write-conflict using the following rule:
\[
\text{false}\_\text{wc}(X,Y) : \neg \text{write}(X,D), \text{write}(Y,D), \neg X=Y.
\]
We compute type-conflict using the rule:
\[
\text{false}\_\text{tc}(X,Y) : \neg \text{dep}(X,Y), \text{cont}(X,C1), \text{cont}(Y,C2).
\]
\[
\text{false}\_\text{tc}(X,Y) : \neg \text{dep}(X,Y), \text{proc}(X,P1), \text{proc}(Y,P2).
\]

**C.2 Temporal Integrity Constraints**

An invocation must have started in order to read a data artifact:
\[
\text{false}\_\text{rbi}(D, I) : \neg \text{bf}(\text{read}(D,I), \text{invoc}(I)).
\]
A write event must occur while the invocation is still executing:
\[
\text{false}\_\text{ibw}(I, D) : \neg \text{bf}(\text{invoc}(I), \text{write}(I,D)).
\]
A dependent data must not be written before the write event of the data artifact on which it is dependent on:
\[
\text{false}\_\text{dbd}(X,Y) :- \neg \text{df}(X,Y), \text{bf}(\text{write}(I,X), \text{write}(J,Y)).
\]
When two nodes of a read event are in cycle, then data artifact must not be read before the invocation. When two nodes of a write event are in cycle, then invocation must not be completed before the write:
\[
\text{false}\_\text{cdt}(X,Y) :- \neg \text{cycle}(X,Y), \text{bf}(\text{read}(X,Y), \text{invoc}(Y)).
\]
\[
\text{false}\_\text{cdt}(X,Y) :- \neg \text{cycle}(X,Y), \text{bf}(\text{invoc}(X), \text{write}(X,Y)).
\]