Declaratively Processing Provenance Metadata

Scott Moore
Harvard University[*]

Ashish Gehani   Natarajan Shankar
SRI International

Abstract

Systems that gather fine-grained provenance metadata must process and store large amounts of information. Filtering this metadata as it is collected has a number of benefits, including reducing the amount of persistent storage required and simplifying subsequent provenance queries. However, writing these filters in a procedural language is verbose and error prone. We propose a simple declarative language for processing provenance metadata and evaluate it by translating filters implemented in SPADE [9], an open-source provenance collection platform.

1 Introduction

SPADE [9] is an open source software platform that supports collecting, filtering, storing, and querying provenance metadata. Computational history is reported as a stream of events that describe the relationships between agents, the processes they control, and the data artifacts produced and consumed. This stream is often voluminous and replete with information that is of limited interest. Filtering the event stream to abstract it during collection has multiple benefits. It can dramatically reduce the persistent storage needed, reduce the time to index the provenance, and simplify subsequent querying.

The specific transformations needed to abstract the provenance depend on the activity domain that is being reported. For example, a stream of individual temperature readings by a single sensor may be combined into one reading that describes the bottom and top of the observed range within a temporal interval. Consequently, SPADE provides a framework for implementing filters (that can be stacked in arbitrary order). A filter receives a stream of provenance graph vertices and edges, and can rewrite their annotations (in which domain-specific semantics are embedded). Each filter can propagate the vertices and edges to the next filter, drop the graph elements, or even synthesize new ones. However, since a filter must be implemented as procedural Java code, it can be verbose and is prone to mismatches between the user's intent and the implementation realized. To alleviate these difficulties, we propose a declarative programming language for processing streams of provenance metadata.

Declarative programming languages, in particular Datalog, have been used for a number of provenance reasoning and processing tasks, including query specification [5, 7, 14], transformation [8], and dependency inference [3]. However, all of these tasks consider offline analysis of provenance metadata. Outside of the provenance community, there has been significant research on languages and systems for event processing [4] and on declarative languages for event-based distributed programming [1, 13].

In this paper, we explore using a declarative event processing language to process provenance metadata. We propose a simple domain-specific language for event processing based on Datalog which we call Simple Event Logic (SEL). We have implemented a meta-interpreter for SEL in SWI-Prolog and used it to prototype modules implementing provenance processing tasks found in SPADE.

We introduce SEL in Section 2. Section 5 demonstrates the use of declarative programs to write provenance filters. In Section 6, we summarize our experience and describe likely avenues for future work.

2 SEL

SEL extends Datalog¬ (Datalog with negation and aggregate functions [17]) with a synchronous model of time, a single temporal operator "previously" (written '?'), and a simple module system.

SEL's temporal model is similar to synchronous programming languages like Esterel, Lustre, and Signal [2, 11, 12]. At each instant in time, an arbitrary number of

[*]while visiting SRI International

instantaneous events may occur. Instants in time have a total ordering, but physical time is not observable (unless provided explicitly as an attribute of an event). New instants are triggered when new events are introduced to the program by an external source: for example, when new provenance-related events are reported by the operating system.

Events are represented as Datalog facts and may have 0 or more attributes. Inference rules generate new events that occur simultaneously with the events that trigger them. Inference rules may also depend on the presence (or absence) of events in the immediately previous instant. Complex events can be abstracted by modules to simplify programs and facilitate code reuse.

Well-formed Datalog programs with negation and aggregate functions must be stratifiable: the dependency graph of inference rules should have no cycles with negated or aggregate goals. SEL programs must satisfy a slightly weaker constraint: *temporal stratifiability* [1]. Temporally stratifiable programs may contain cycles with negated or aggregate goals, but those goals must be preceded by a "previously" operator—such programs are stratifiable in each instant, but admit non-monotonic reasoning across instants.

We give a high level description SEL's formal semantics in the next section before demonstrating the language and its syntax with a small example.

3 Semantics

The semantics of SEL was inspired by the DEDALUS language for distributed programming [1]. DEDALUS is a declarative, Datalog-based programming language with support for mutable state and asynchronous communication. Unlike other languages that have attempted to combine imperative features with Datalog, the semantics of DEDALUS are purely declarative and are given by a transformation into a restricted sublanguage of Datalog¬, which the authors call DEDALUS_0.

Likewise, we give the semantics of SEL in terms of a restricted sublanguage of Datalog¬. Like DEDALUS_0, this core language has two syntactic restrictions:

1. *Timestamps:* The final argument of all relations must be a natural number denoting a "timestamp."

2. *Sequencing constraints:* The timestamps $T_1, \ldots, T_n$ of all subgoals in a rule must be related to the timestamp of the head of the rule, S, by either an equality constraint, $T_i = S$, or a successor constraint `successor(`T_i`, `S`)`.

Unlike DEDALUS_0, we do not require that all subgoals of a rule share the same timestamp. This allows some programs to be expressed more concisely at the expense of a slightly more complicated well-foundedness check for temporal stratification of negation and aggregation. Temporally stratifiable programs meeting these restrictions have a unique (possibly infinite) minimal model: the set of facts true at each instant in time for each relation. We take this minimal model as the semantics of the program.

The desugaring process from SEL to this core language is straightforward. A timestamp attribute is added to the schema of each relation. Within each inference rule, a constraint is added relating the timestamp of each subgoal to the timestamp of the rule head. If the subgoal is prefixed with a "previously" operator, a successor constraint is added; otherwise, an equality constraint is added.

Modules are transformed by inlining their definitions where they are imported, renaming events to unique names to avoid clashes. To allow module 'clocks' to differ, additional rules are added that relate timestamps from different modules only when an input event for an included module is triggered.

4 SEL By Example

Listing 1 gives an example SEL program that monitors a hypothetical boiler for dangerous conditions. The program comprises two modules, delta (defined in lines 1-10) and boiler (defined in lines 12-27).

The delta module takes as input a stream of `value` events, each with a single numeric attribute. The module outputs a stream of `change` events as the `value` parameter changes. The attribute of the change event is the difference between the new value and the old value. For each pair of a current value event and a value event in the previous instant, the single inference rule (lines 5-8) generates a `change` event if the attribute of a current value differs from the previous one.

The boiler module takes as input a stream of pressure readings from sensors in the boiler and switch on/off events from the boiler control panel. In line 17, sensor readings arriving in the same instance (from different sensors) are averaged using an aggregate function. The inference rule in lines 22-23 triggers an alert event if the average pressure reading exceeds the safety limits of the boiler.

The program also monitors for unexpected changes in pressure. An unexpected change in pressure is a positive change in pressure that occurs while the boiler is off. Monitoring this property requires tracking the state of the boiler. Lines 19-20 generate a `heating` event in every instant the boiler is on. The first inference rule records that the boiler is on if it has been switched on. The second inference rule denotes that the boiler remains on if it was previously on (`?heating`) and has not been turned

```
 1 module delta.
 2 input value/1.
 3 output change/1.
 4
 5 change(Change) :-
 6   value(Cur), ?value(Old),
 7   Cur != Old, Change = Cur - Old.
 8
 9 end delta.
10
11 module boiler.
12 input switch_on/0, switch_off/0, sensor/1.
13 output alert/1.
14 import delta with value=pressure.
15
16 pressure(average<P>) :- sensor(P).
17
18 heating :- switch_on.
19 heating :- ?heating, ¬switch_off.
20
21 alert('Too much pressure!') :-
22   pressure(P), P > 100.
23 alert('Shut off failed!') :-
24   ¬heating, change(D), D >= 0.
25
26 end boiler.
```

Listing 1: Monitoring a boiler for dangerous conditions.

off (`¬switch_off`). To detect changes in boiler pressure, line 15 imports the delta module using pressure events as its input stream. Finally, the inference rule in lines 24-25 triggers an event if a positive change occurs while the boiler is not in heating mode.

5 Processing Provenance Metadata

SPADE incorporates filters that apply a number of transforms to incoming streams provenance events in order to reduce storage requirements, increase performance, and simplify subsequent queries. SPADE filters are also used to abstract provenance information from detailed, low-level event streams (e.g., provenance data reported by an OS) and fuse provenance metadata received via disparate reporting mechanisms. To evaluate the applicability of declarative programing to the problem of processing streams of provenance events, we translated several SPADE filters into SEL programs.

We represent SPADE provenance metadata in SEL as streams of vertex, edge, and attribute events. Our encoding is similar to Missier and Belhajjame's [14] encoding of the PROV specification in DLV Datalog. All vertices and edges have an associated ID and type (e.g., "artifact" or "wasGeneratedBy") derived from the Open Provenance Model [15]. Attribute events associate key-value pairs with IDs.

In the remainder of this section, we give examples of two types of provenance filters in SPADE and their SEL implementations.

```
module IORuns.
input vertex/2, edge/4, attr/3.
output vertex_out/2, edge_out/4, attr_out/3.
import Aggregate.

emit(ID) :-
  vertex(ID,Type), Type != artifact.
emit(ID) :-
  edge(ID,Type,_,_),
  Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
  edge(ID,used,Process,Artifact),
  attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
  read(ID,Process,Artifact,File),
  ¬?reading(_,Process,_,File).
reading(ID,Process,Artifact,File) :-
  reading(ID,Process,Artifact,File),
  ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
  reading(ReadSeries,Process,_,File),
  read(Read,Process,_,File).
aggr(ReadArtifact,Artifact) :-
  reading(_,Process,ReadArtifact,File),
  read(_,Process,Artifact,File).

emit(Artifact) :-
  reading(_,_,Artifact,File),
  write(_,_,_,File).
emit(Read) :-
  reading(Read,_,_,File),
  write(_,_,_,File).

// ... similar rules for writes ...
end IORuns.
```

Listing 2: Aggregating runs of IO events.

Aggregation Keeping track of fine-grained provenance metadata provides a detailed view of the relationships between data object involved in a computation, but at the expense of additional storage and processing overhead. For example, SPADE includes a number of reporting interfaces that receive detailed metadata about file I/O. SPADE supports a number of filters for aggregating this metadata into provenance events [10]. Listing 2 gives a snippet of one of these filters, IORuns, translated into SEL. The IORuns filter aggregates successive file reads or writes into a single provenance artifact.

The IORuns module relies on the Aggregate module given in Listing 3. The Aggregate module provides two services: buffering events and aggregating the attributes of multiple provenance elements. Lines 6-14 retain events until they are either passed to the next module by an `emit/1` input event or dropped by a `drop/1` event. The `aggr/2` input event records relationships between provenance items. When a provenance element is emitted, it is associated with the attributes of all sub-

```
 1 module Aggregate.
 2 input vertex/2, edge/4, attr/3,
 3       aggr/2, emit/1, drop/1.
 4 output vertex_out/2, edge_out/2, attr_out/3.
 5
 6 vertex(ID,Type) :- ?vertex(ID,Type),
 7   ¬?emit_attr(_,ID), ¬?drop(ID).
 8 edge(ID,Type,Src,Dst) :-
 9   ?edge(ID,Type,Src,Dst),
10   ¬?emit_attr(_,ID), ¬?drop(ID).
11 attr(ID,Key,Value) :- ?attr(ID,Key,V.alue),
12   ¬?emit_attr(_,ID), ¬?drop(ID).
13 aggr(Super,Sub) :- ?aggr(Super,Sub),
14   ¬?emit_attr(Super,_), ¬?drop(Super),
15   ¬?emit_attr(_,Sub), ¬?drop(Sub).
16
17 emit_attr(ID,ID) :- emit(ID).
18 emit_attr(ID,Sub) :-
19   emit_attr(ID,Super), aggr(Super,Sub).
20
21 vertex_out(ID,Type) :-
22   emit(ID), vertex(ID,Type).
23 edge_out(ID,Type,Src,Dst) :-
24   emit(ID), edge(ID,Type,Src,Dst).
25 attr_out(ID,Key,Value) :-
26   emit_attr(ID,Attr), attr(Attr,Key,Value).
27
28 end Aggregate.
```

Listing 3: Buffering and aggregating provenance events.

elements specified by `aggr/2` (Lines 16-18 and 24-25).

In lines 6-10, IORuns immediately forwards non-file-related provenance elements to its output. File-related events are buffered by the Aggregate module. The `reading/4` event associates an edge ID, process, and file name with each series of successive reads of a file by a process (lines 12-21). If a process reads from a file that it is already reading, the new provenance elements are aggregated with the first read in the sequence (lines 23-28). When a series of reads is broken by a write to the same file by any process, the aggregated provenance elements are output (lines 30-35). Similar rules buffer and aggregate writes and emit pending metadata when a process ends, but we omit them here for brevity.

Our SEL implementation of IORuns, including the Aggregate module, is 74 non-comment lines of code comprising 25 inferences rules. While this is actually slightly more than the original Java implementation (Listing 6 in Appendix 7), there are three advantages to the SEL implementation. First, it is modular: the Aggregate module can be reused by other filters. Second, it is declarative: the inference rules for `emit/1` make clear what provenance data is generated by the module. In the original implementation, this required understanding the control flow of the entire filter, including the effect of imperative updates to a number of two-level hash tables. Finally, our SEL implementation aggregates the attributes of subsequent reads, whereas the original implementation simply discards them.

```
 1 module LLVMFilter.
 2 input vertex/2, edge/4, attr/3, relevant/1.
 3 output vertex_out/2, edge_out/2, attr_out/3.
 4 import Aggregate.
 5
 6 relevant_node(ID) :-
 7   attr(ID,function,Name),
 8   relevant(Name).
 9 relevant_node(Artifact) :-
10   edge(_,used,Process,Artifact),
11   relevant_node(Process).
12 relevant_node(Artifact) :-
13   edge(_,wasGeneratedBy,Artifact,Process),
14   relevant_node(Artifact).
15
16 relevant_edge(ID) :-
17   edge(ID,used,Process,Artifact),
18   relevant_node(Process).
19 relevant_edge(ID) :-
20   edge(ID,wasGeneratedBy,Artifact,Process),
21   relevant_node(Process).
22 relevant_edge(ID) :-
23   edge(ID,wasTriggeredBy,Caller,_),
24   relevant_node(Caller).
25 relevant_edge(ID) :-
26   edge(ID,wasTriggeredBy,_,Callee),
27   relevant_node(Callee).
28
29 irrelevant_edge(ID) :-
30   edge(ID,_,_,_), ¬relevant_edge(ID).
31 irrelevant_node(ID) :-
32   edge(_,_,ID,_), ¬relevant_node(ID).
33 irrelevant_node(ID) :-
34   edge(_,_,_,ID), ¬relevant_node(ID).
35
36 emit(ID) :- relevant_node(ID).
37 emit(ID) :- relevant_edge(ID).
38 drop(ID) :- irrelevant_node(ID).
39 drop(ID) :- irrelevant_edge(ID).
40
41 end LLVMFilter.
```

Listing 4: Filtering application-level provenance.

Filtering SPADE also supports the collection of fine-grained, application-specific provenance metadata by instrumenting programs to report function calls and returns [16]. To cope with the volume of metadata produced, SPADE allows users to specify a set of interesting functions. A filter drops metadata associated with irrelevant functions.

The module in Listing 4 implements this filter in SEL. The `relevant/1` input event defines a set of functions for which provenance metadata should be collected. Provenance nodes are relevant if they represent calls to relevant functions or artifacts used or generated by a relevant function (lines 6-14). An edge is relevant if it involves relevant process nodes (lines 16-27). Because whether an artifact is relevant depends on the edges it connects to (which may not have arrived yet), the module must maintain a buffer of artifact nodes. Like the original SPADE implementation, the filter assumes that artifacts share edges with either only relevant or only irrelevant process nodes. Thus, it is possible to determine when a provenance element is irrelevant (lines 29-34) and drop it from the buffer (lines 38-39).

Unlike the Java implementation (Listing 7 in Appendix 7), the declarative rules defining relevant and irrelevant provenance elements make clear the semantic conditions under which artifact nodes are buffered, dropped, or output.

6 Future Directions

As other provenance researchers have observed [3, 7, 8, 14], Datalog is a natural choice for representing, querying, and reasoning about provenance. Our experience translating SPADE filters from Java into SEL suggests that using a declarative language also makes it easier to design and implement tools for processing streams of provenance metadata.

Buffer management remains a key challenge when implementing provenance filters. Many provenance filters, including the examples in Section 5, require knowledge of past or future provenance events to determine how to process a metadata element. In both the Java and SEL implementations of our filters, this buffer is explicitly managed. Ensuring that buffers do not grow too large while still retaining relevant information is a source of significant complexity. We foresee two possible solutions to this challenge. Operationally, we could incorporate buffering primitives like sliding windows seen in event processing languages [4]. Declaratively, we plan to investigate using known query optimization techniques to attempt to infer from a filter's inference rules what current events might trigger rules in the future, and thus provide automatic buffer management.

Incorporating a query language into systems that process streams of provenance metadata opens up new avenues of research in online provenance queries. SPADE supports collecting system-wide provenance information from operating system and network reporting interfaces. Online provenance queries could allow this data to be used to detect anomalies or report on system status in real time rather than retrospectively.

For example, consider the SEL module LeakDetector in Listing 5. LeakDetector monitors a stream of provenance events to detect when sensitive data may be leaked to the network by tracking possible data flows through processes and artifacts. By actively processing system-wide provenance data, this module can detect a possible security violation as it happens and provide an alert. The original provenance data can then be audited for additional information.

```
module LeakDetector.
input vertex/2, edge/4, attr/3, secret/1.
output leak/2.

tainted(Artifact,File) :-
  attr(Artifact,location,File),
  secret(File).

tainted(Process,Secret) :-
  edge(_,used,Process,Artifact),
  tainted(Artifact,Secret).

tainted(Artifact,Secret) :-
  edge(_,wasGeneratedBy,Artifact,Process),
  tainted(Process,Secret).

leak(Secret,Connection) :-
  edge(_,wasGeneratedBy,Connection,Process),
  attr(Connection,subtype,network),
  tainted(Process,Secret).

tainted(Node,Secret) :-
  ?tainted(Node,Secret),
  ¬attr(Node,'end',_).

end LeakDetector.
```

Listing 5: Monitoring provenance for security.

In addition to the filters and queries presented in this paper, existing declarative provenance transformations could be applied to enforce privacy policies [8], integrate additional workflow information [3], or support certification with claims and evidence from formal tools [6].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CNS-0917375 and IIS-1116414, and the National Aeronautics and Space Administration under Cooperative Agreement NNX08AY53A. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the National Aeronautics and Space Administration.

References

[1] Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. Dedalus: datalog in time and space. In *Datalog'10: 1st International Conference on Datalog Reloaded*. Springer-Verlag, 2010.

[2] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. Declarative rules for inferring fine-grained data provenance from scientific workflow execution traces. In *4th International Provenance and Annotation Workshop*, pages 82–96. Springer-Verlag, 2012.

[4] Francois Bry, Michael Eckert, Opher Etzion, Jon Riecke, and Adrian Paschke. Event processing languages. *Tutorial in DEBS 2009*, 2009.

[5] Shirley Cohen, Sarah Cohen-Boulakia, and Susan Davidson. Towards a model of provenance and user views in scientific workflows. In *3rd International Conference on Data Integration in the Life Sciences*, pages 264–279, 2006.

[6] Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool integration with the evidential tool bus. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 275–294, 2013.

[7] Saumen Dey, Sven Köhler, Shawn Bowers, and Bertram Ludäscher. Datalog as a lingua franca for provenance querying and reasoning. In *4th USENIX Workshop on the Theory and Practice of Provenance*, 2012.

[8] Saumen C. Dey, Daniel Zinn, and Bertram Ludäscher. Propub: towards a declarative approach for publishing customized, policy-aware provenance. In *23rd International Conference on Scientific and Statistical Database Management*, pages 225–243, 2011.

[9] Ashish Gehani and Dawood Tariq. SPADE: Support for provenance auditing in distributed environments. In *13th ACM/IFIP/USENIX International Conference on Middleware*, pages 101–120, 2012.

[10] Ashish Gehani, Dawood Tariq, Basim Baig, and Tanu Malik. Policy-Based Integration of Provenance Metadata. In *12th IEEE International Symposium on Policies for Distributed Systems and Networks*. IEEE Computer Society, 2011.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[12] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9): 1321–1336, 1991.

[13] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[14] Paolo Missier and Khalid Belhajjame. A prov encoding for provenance analysis using deductive rules. In *4th International Provenance and Annotation Workshop*, pages 67–81, 2012.

[15] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, et al. The open provenance model core specification (v1. 1). *Future Generation Computer Systems*, 27(6):743–756, 2011.

[16] Dawood Tariq, Maisem Ali, and Ashish Gehani. Towards automated collection of application-level data provenance. In *4th USENIX Workshop on the Theory and Practice of Provenance*, 2012.

[17] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

7 SPADE filters in Java

Listings 6 and 7 give snippets from the Java implementation of the SPADE filters in Section 5. SPADE is available at `http://code.google.com/p/data-provenance` under the GPLv3.

```java
public void putVertex(AbstractVertex incomingVertex) {
  if (incomingVertex instanceof Artifact) {
    vertexBuffer.add(incomingVertex);
  } else {
    putInNextFilter(incomingVertex);
    return;
  }
  if (vertexBuffer.size() > BUFFER_SIZE)
    Logger.getLogger("IORuns").warning("*** Vertex Buffer full. Dropping! )))");
}

public void putEdge(AbstractEdge incomingEdge) {
  if (incomingEdge instanceof Used) {
    Used usedEdge = (Used) incomingEdge;
    String fileVertexHash = usedEdge.getDestinationVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(usedEdge.getSourceVertex().hashCode());
    if (!reads.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = new HashSet<String>();
      tempSet.add(processVertexHash);
      reads.put(fileVertexHash, tempSet);
    } else {
      HashSet<String> tempSet = reads.get(fileVertexHash);
      if (tempSet.contains(processVertexHash)) {
        vertexBuffer.remove(usedEdge.getDestinationVertex());
        return;
      } else { tempSet.add(processVertexHash); }
    }
    vertexBuffer.remove(usedEdge.getDestinationVertex());
    putInNextFilter(usedEdge.getDestinationVertex());
    putInNextFilter(usedEdge);
    if (writes.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = writes.get(fileVertexHash);
      tempSet.remove(processVertexHash);
    }
  } else if (incomingEdge instanceof WasGeneratedBy) {
    WasGeneratedBy wgb = (WasGeneratedBy) incomingEdge;
    String fileVertexHash = wgb.getSourceVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(wgb.getDestinationVertex().hashCode());
    if (!writes.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = new HashSet<String>();
      tempSet.add(processVertexHash);
      writes.put(fileVertexHash, tempSet);
    } else {
      HashSet<String> tempSet = writes.get(fileVertexHash);
      if (tempSet.contains(processVertexHash)) {
        vertexBuffer.remove(wgb.getSourceVertex());
        return;
      } else { tempSet.add(processVertexHash); }
    }
    vertexBuffer.remove(wgb.getSourceVertex());
    putInNextFilter(wgb.getSourceVertex());
    putInNextFilter(wgb);
    if (reads.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = reads.get(fileVertexHash);
      tempSet.remove(processVertexHash);
    }
  } else { putInNextFilter(incomingEdge); }
}
```

Listing 6: Code snippets from the Java implementation of the IORuns filter.

```java
public void putVertex(AbstractVertex incoming) {
  if (incoming instanceof Process) {
    if (methodsToMonitor.contains(incoming.getAnnotation("FunctionName"))) {
      putInNextFilter(incoming);
    }
  } else {
    String ID = incoming.getAnnotation("ID");
    artifacts.put(ID, 1);
  }
}

public void putEdge(AbstractEdge incoming) {
  if (incoming instanceof Used) {
    Artifact artifact = (Artifact) incoming.getDestinationVertex();
    Process process = (Process) incoming.getSourceVertex();
    String ArgID = artifact.getAnnotation("ID");
    if (methodsToMonitor.contains(process.getAnnotation("FunctionName"))) {
      if (artifacts.containsKey(ArgID)) // Every Artifact is used at most twice
      {
        if (artifacts.get(ArgID) == 1) // Every Artifact is used at most twice
        {
          artifacts.put(ArgID, artifacts.get(ArgID) + 1); // Increment Counter
          putInNextFilter(artifact);
        } else {
          // If Artifact seen twice remove it from the HashMap
          artifacts.remove(ArgID);
        }
        putInNextFilter(incoming);
      }
    } else {
      // If we do not want to monitor the Artifact remove it from the HashMap
      artifacts.remove(ArgID);
    }

  } else if (incoming instanceof WasGeneratedBy) {
    Process process = (Process) incoming.getDestinationVertex();
    Artifact artifact = (Artifact) incoming.getSourceVertex();
    String ArgID = artifact.getAnnotation("ID");
    if (methodsToMonitor.contains(process.getAnnotation("FunctionName"))) {
      if (artifacts.containsKey(ArgID)) {
        if (artifacts.get(ArgID) == 1) {
          artifacts.put(ArgID, artifacts.get(ArgID) + 1);
          putInNextFilter(artifact);
        } else {
          artifacts.remove(ArgID);
        }
        putInNextFilter(incoming);
      }
    } else {
      artifacts.remove(ArgID);
    }
  } else // WasTriggeredBy
  {
    AbstractVertex source = incoming.getSourceVertex();
    AbstractVertex destination = incoming.getDestinationVertex();
    if (methodsToMonitor.contains(source.getAnnotation("FunctionName"))) {
      if (methodsToMonitor.contains(destination.getAnnotation("FunctionName"))) {
        putInNextFilter(incoming);
      }
    }
  }
}
```

Listing 7: Code snippets from the Java implementation of LLVMFilter.