# Toward provenance capturing as cross-cutting concern

Martin Schäler, Sandro Schulze, and Gunter Saake
*School of Computer Science, University of Magdeburg, Germany*
{*schaeler, sanschul, saake*}*@iti.cs.uni-magdeburg.de*

## Abstract

Although provenance gained much attention, solutions to capture provenance do not meet all the requirements. For instance, most solution currently assume a closed world and are explicitly designed to capture provenance. Thus, they fail in integrating the provenance concern into existing environments. Hence, we argue that provenance should be considered as cross-cutting concern that can easily be integrated into existing systems and aims at establishing a universe of provenance. In this paper, we propose a solution concept, introduce different types of provenance systems, adequate software engineering techniques, and report our experiences from a first prototype.

## 1 Introduction

As provenance gained much attention in the recent years, there exist several solutions that are explicitly designed to capture provenance. For instance, a prominent example to capture provenance for scientific workflows is the Kepler system [6]. Moreover, formal approaches for database systems and respective prototypes as Orchestra for the semiring model [4] are proposed in the literature [3]. However, we argue that in practice capturing provenance efficiently is still one of the major challenges. This becomes even worse when considering complex IT-landscapes, including systems with limited knowledge about their implementation (e.g., third party services). Furthermore, human interactions or non-computational steps may introduce highly different levels of granularity and the necessity for different ways to collect provenance. Finally, different user interests and changing privacy policies may influence provenance even at capturing level (despite respective query models). As a result, designing, linking, and adapting solutions that are explicitly designed to capture provenance is costly, inflexible, and in some cases even impossible. To overcome these limitations, we argue that, especially for complex IT-landscapes, provenance should be considered as cross-cutting concern that can seamlessly be integrated into existing systems and thus is able to gather provenance in a flexible and efficient way. To achieve this vision:

- we propose a solution concept addressing provenance on different granularity levels, where we change black-box into more white-box computation,
- we describe different types of systems based on implementation details and the way the desired provenance information is spread across these systems,
- we show how to use techniques, known from software engineering, to integrate the provenance concern depending on the system type and report experiences from a first prototype we implemented.

## 2 Solution concept

To realize our vision of ubiquitous and flexible provenance capturing, we analyzed ways to link coarse-grained forms of provenance to more fine-grained forms in a systematic way. In fact, our results indicate that several approaches proposed in the literature form a conceptual framework indicating *what* to capture at different levels of granularity [10]. However, this conceptual framework does not specify *how* to capture the desired data. In contrast, with our proposed concept we can apply a flexible and formal approach (if there is one) that indicates what data to capture. To this end, we use adequate software engineering techniques to collect the provenance data.

### 2.1 Overview on our Solution

A major goal of our solution is that it is generally usable. This means that we want to be able to exchange the collected provenance data among different systems heading toward a universe of provenance. Consequently, we use an internal representation that is based on the Open Provenance Model (OPM) [8, 10], a commonly accepted model to exchange provenance data. However, our internal representation contains two major extensions compared to the OPM. First, we introduce complex artifacts (having internal structure), which are similar to *collections* in the OPM,

allowing for instance to link parts of the input to parts of the output. In contrast to collections that are mainly associated to arrays or queues, complex artifacts may also represent objects in object-oriented programming or a database containing several tables, which again have tuples etc. The second extension are complex processes, which are mainly inspired by user-views [2]. However, in combination with complex artifacts and implicit dependencies and due to the advantage of considering the zooming-problem already when collecting the data, we are more flexible in visualizing different levels of granularity in the provenance graph [10].

The basic term for our provenance capturing is the execution of a certain *function* and its respective in- and output (equivalent to OPM processes), encapsulating a certain computation in a black box manner. This procedure is similar to the one used by Amsterdamer et al. for the operations of the pig latin language [1]. However, they map the operations of the pig latin language to a nested relational calculus to apply the semiring model allowing very detailed provenance capturing. In contrast, for *user defined functions* they keep the black box representation. Since we are interested in complex IT landscapes, the problems we face are similar to those of user defined functions. Hence, we additionally refine these functions if and only if (1) additional implementation details or a formal approach are known and (2) this particular function is of user interest. Note that this allows to flexibly change the implementation, for example, if there are advances in formalizing provenance capturing or changes of the user interest for instance due to privacy policy changes. Summarily, the provenance information for the execution of a certain *function* we are looking for are: (1) input and output, (2) (optional) call hierarchy (for zooming), (3) input classification into exogenous and endogenous input in spirit of [7], (4) (optional) linking input parts to parts of the output, and (5) (currently only for database operations) value origin.

## 2.2 Provenance capturing for object-oriented programs

As we assume that many programs in a heterogeneous IT-environment are object-oriented programs, we need to specify how to capture provenance within these systems. To the best of our knowledge, there is currently no commonly accepted formalism that allows for provenance collection, if the information we are looking for is represented by the structure of the program itself. Thus, we currently map the elements of such a program to the information we are looking for with the help of our own approach. In the following, we will briefly explain this approach with he help of the example in Figure 1. Due to space limitations, we refer the reader to Appendix A for a more formal and detailed description.

```
1  class Foo{
2     private int memOne = 3;
3     private Foo2 memTwo;//never used
4
5     RetType[] somefunc(int arg1/*=2*/){
6        RetType[] ret = new  RetType[arg1];
7        memOne++;
8        for (int i=0; i<arg1; i++)
9           ret[i] = new RetType(memOne);
10       return ret;
11 }  }
```

Figure 1: Provenance for object-oriented programs

**Points of interest**

To initiate a specific provenance collection event, we need distinct points in the structure of a program, called *points of interest (PoI)*. As our basic term for provenance capturing is a *function*, we refer to `method` and `constructor` calls. For instance, in Figure 1 there are two possible PoIs: (1) call of the method `Foo.somefunc(int)` in Line 5 and (2) a constructor call `RetType.RetType(int)` in Line 9. Hence, a PoI contains a *return type*, a *function name*, and a *list of argument types* (from the function call). Note that we currently do not consider array constructors (Line 6), as they mainly allocate the required memory. The basic idea to achieve flexibility is that a user defines the PoIs and we collect provenance only at these distinct points. The provenance capturing procedure at each point is the same as we will explain in the following.

## 2.3 Two phases for black box provenance

When reaching a PoI, we need two phases to collect *in- and output* of a function (black box). This is due to the fact that, when entering the function, we do not know the return values (output). Furthermore, when leaving the function, possibly input values may have changed (or are already destroyed). To explain *how* to determine in- and output of a function, we assume that there is only one PoI in our example (c.f. Figure 2 Line 14): `Foo.somefunc(int)` and the method is called with $arg1 = 2$. Furthermore, we assume that class member *memOne* is equal to three. We will explain how we capture provenance with the help of a reference approach. This approach is furthermore used to compare this approach to alternative ones. The necessary source-code modification to capture provenance for the example in Figure 1 are depicted in Figure 2. The basic idea is to use if-condition to decide whether a PoI is active.

**Entering a function: Pre-phase.** The pre-phase of each PoI creates the function entry (including time-stamps) and determines the respective input. For `Foo.somefunc(int)` the pre-phase is executed when entering the function (Figure 2 Line 2-4). First, all arguments of the method call are part of the input. For the

```
1  RetType[] somefunc(int arg1/*=2*/){
2     if(ActivePOI.SOMEFUNC){
3        Object input ={arg1,memOne,memTwo};
4        ProvenanceHandler.prePhase(input);}
5     RetType[] ret = new  RetType[arg1];
6     memOne++;
7     for (int i=0; i<arg1; i++)
8        ret[i] = new RetType(memOne);
9     if(ActivePOI.SOMEFUNC)
10       ProvenanceHandler.postPhase(ret);
11    return ret;
12 }
```

```
13 interface ActivePOI{
14    SOMEFUNC = true;
15    REF_CONSTRUCTOR = false;}
```

Figure 2: Source-code modification to capture provenance using the reference approach

example, we therefore add `arg1`. Second, we need to add all class members to the input, as they can be used within the method code. Thus, `memOne` and `memTwo` are added. Finally, for all arguments passed by reference, we state the current values. This is because any modification of these arguments may be visible for different parts of the program and thus we have to add this input artifact to the list of output artifacts if there was any modification. Note that we currently do not consider changes to the internal state (members), because we may not see *every* modification, which is the main difference to [1].

**Leaving a function: Post-phase.** The post-phase determines the output of the function and persistently saves the provenance records when leaving the function (Figure 2 Line 9-10). To this end, we first add the return value of the method or constructor and then every argument (i.e., no class members) that is (1) called by reference and (2) was modified within this function. As a result, we know in- and output, but we do not have knowledge how parts of the input are mapped to parts of the output. For instance, the return value in our example is an array containing two object of type `RetType`. To solve this problem, we connect different fragments (cf. Section 2.4) when the required details are known and additional PoIs are defined within the execution of this particular function.

## 2.4 From black box to white box

To turn a black box computation step into a more white box one, we use call hierarchies of the functions, which is highly related to granularity. In Figure 3, we depict the linked provenance data, captured at both PoIs in our example, in graph notation. The call hierarchies indicate which additional functions were called while executing a specific function. For instance, we know that function `Foo.somefunc(int)` called two additional functions in Figure 1 Line 9 (in this case constructors). Now we want to link parts of the input to parts of the output. In particular, we want to know which artifacts are used to

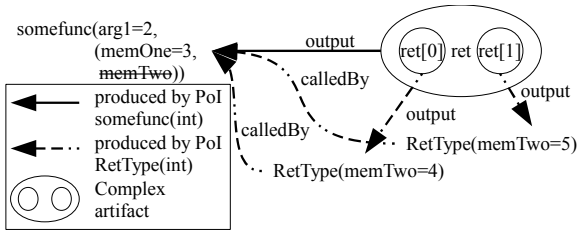


Figure 3: Toward white box computation

create the single parts of array `ret`. To this end, we need a second PoI `RetType(int)` that collects the provenance data whenever this constructor is executed. As a result, we know that each `RetType` was created using an `integer` from the calling function. Furthermore, because each artifact has a unique id we know that `memOne` is used to create each part of `ret` and its current value, because it is a simple artifact directly linked to a value.

# 3 Toward efficient provenance capturing

Since we want to collect the provenance information efficiently, we need criteria to evaluate different software engineering techniques that allow to implement provenance capturing. Furthermore, we need a reference technique to qualify alternative solutions.

## 3.1 Evaluation criteria

The criteria we consider are: (1) Invasiveness, (2) Manual implementation overhead, and (3) Runtime overhead. In the following, we will briefly introduce these criteria and motivate their significance.

**Invasiveness** determines the amount of changes to the original program and the ability to automatically remove the code implementing a specific point of interest if it is no longer of interest.

**Manual implementation effort** states the effort to implement the provenance collection. It is introduced to give credit to solutions that (semi)-automatically inject the additionally required source code.

**Runtime overhead** includes the additional amount of time as well as additional memory consumption for provenance capturing.

**Properties of the reference approach.** For the *dynamic-if* approach (cf. Section 2.3), which is the reference approach, we have to *manually* introduce if-conditions for every PoI. As a result, this approach is maximally invasive as we permanently modify the source of the program and cannot automatically remove the modification. Furthermore, this technique requires maximum implementation effort, as we have to manually locate and implement the modification for each PoI.

## 3.2 Groups of provenance systems

The basic idea behind introducing different groups of provenance systems is that we want to identify software

engineering techniques that are beneficial for a certain group, rather than to consider every system in isolation. Furthermore, we want to show the general feasibility of our approach based on the groups and find properties in the program structure that are either beneficial for integrating the provenance concern or impose a specific drawback. For instance, implementation effort highly depends on the number and distribution of the PoIs in the source code of the system. Thus, specific techniques that introduce high performance overhead but require low implementation effort may be feasible for one system while a different system requires very low performance overhead (e.g., databases).

**Group 1: No source code access.** For third party services such as APIs and respective libraries or proprietary database systems, we have no access to the source code and thus cannot modify the implementation. However, as we want to collect provenance for these system as well, we need to get as close as possible. For databases, we may use modified drivers to capture provenance (i.e., query and respective result), or link the provenance collection event to certain functions of an API. Generally, we assume that there is only a limited number of PoIs (because we cannot capture fine-grained provenance). As a result, using techniques that have high implementation or performance overhead seem to be possible for these systems.

```
1  aspect ProvenanceImplementation{
2  //PoI specific
3  pointcut SOMEFUNC() : execution(int Foo.somefunc(..));
4  before() : SOMEFUNC () { getInput(thisJoinPoint); }
5  after() returning (Object o) : SOMEFUNC(){ getOutput, o);}
6  //works for all PoI
7  void getInput(JoinPoint jp){
8      Object[] inputArgs = joinPoint.getArgs();
9      //get Class members, requires reflection
10     ProvenanceHandler.prePhase(inputArgs, classMembers);}
```

Figure 4: Source-code modification to capture provenance using AspectJ

**Group 2: Congruent PoIs.** Having congruent PoIs means that the PoIs are represented by the structure of the program as described in Section 2.2. As a result, our *function* is linked to the execution of *one* particular method. This is, for instance, the case when debugging programs that will possibly introduce a high number of PoIs. An example for congruent PoIs is given in Figure 2. Hence, we hypothesize that using aspects (e.g., ApsectJ [5]) for these systems is beneficial.

**Group 3: Scattered PoIs.** The basic characteristic of these systems is that the *functions* of interest are not necessarily congruent to the *methods* but scattered over multiple methods of the system. Furthermore, we are typically interested only into small parts of the system. For instance, for database-like provenance, query processing is only a part of the overall system. Moreover, the specific implementation of a relational algebra operator may cut across multiple methods. Consequently, we need to assemble the provenance and locate the respective source code locations. As a result, we need more flexible and fine-grained techniques than for congruent PoIs.

To sum up, these groups give a first hint what techniques may be beneficial for a certain system. However, in the remainder we will show that not the system itself, but the user interest is the decisive criterion that maps a system to one of these groups.

## 4 First prototype and experience report

In this section, we introduce software engineering techniques to integrate the provenance concern. Moreover, we report experiences of applying specific techniques for a certain group of systems with the help of an implementation prototype. Furthermore, we want to emphasize that the provenance we collect highly depends on the user interest. We choose one system that can, dependent on the user interest, belong to each of the groups. In the remainder of this Section, we use the open-source Java DBMS HyperSQL as example system and give respective use cases when the system belongs to which group.

**Group 1: No source code access.** When a user is not interested into the details of the computation or we have no source code access as for proprietary DBMS, the simplest way to capture provenance is to relate the queries to the query response. Hence, we do not refine the black box characteristic of the function. For the prototype we modified the JDBC driver, using aspects and two PoIs. Particularly, the benefits of using AspectJ is that we can use the implementation for every JDBC driver, because the PoI is directly defined on the methods of the interface that each JDBC driver has to implement. Hence, we conclude that when there is some communication infrastructure, such as a JDBC driver, or a service infrastructure, using aspects is beneficial, because we simply link an aspect to the particular class implementing this commonly used part of the infrastructure.

**Group 2: Congruent PoIs.** In a current research project, we modify the HyperSQL table manager to store tables column-wise (instead of row-wise) to analyze the benefit of column stores in general and different materialization techniques in particular. We used provenance for debugging the system, to visualize the changes, and to keep reference to the original implementation. As our modifications are in fact changes to the program structure, we have congruent PoIs and can use techniques such as AspectJ to keep track of the changed program structure.

To explain how aspects work and why they are beneficial for this group of systems, we use the already known example in Figure 1. In Figure 4, we depict source code

that allows to capture provenance with the help of aspects. First, we have to define a *pointcut* (Line 3), which is our PoI. Furthermore, we define that *before* and *after* this pointcut the methods `getInput` and `getOutput` are executed respectively. To specify new PoIs we simply need new pointcuts, that means three lines of code per PoI. Furthermore, we can create one pointcut for a whole packages using wildcards in the pointcut definition, which is highly useful to flexibly adapt the provenance collection process. Finally, as we do not permanently modify the original source code, we can totally remove the additional source code that collects the provenance, for instance when benchmarking different variants of the modified DBMS. However, our first results indicate a significant performance overhead using aspects. The main reason therefore is not the aspect itself but the reflective[1] determination of class members (e.g. to get tuple IDs etc.). To sum up, one of the lessons learned is, when there are frequently executed functions (e.g., for every row in a database table) our current approach needs improvement.

**Group 3: Scattered PoIs.** When collecting provenance for tuples, we have scattered PoIs. This is because only small parts of the DBMS are used for the relational operations and different parts such as transaction management are of minor interest. Furthermore, tuple operations are scattered over multiple methods, some operations are mixed (e.g., projection and selection), and some information such as tuple identifiers are not visible at all in these methods. As a result, we do not use aspects here for two reasons. First, we discovered a significant performance overhead using aspects, which is undesirable for a DBMS. Second, we need more flexible techniques as some of the desired information are only visible in certain blocks of a method (i.e., not when entering or leaving a method). Moreover, as the main challenge is to identify the source code locations implementing a certain PoI, which is currently done manually, automatic code injection *currently* introduces no benefit. Thus, we use a preprocessor-based technique. Hence, we can manually introduce source code at any location within the system (even to modify existing statements). As this technique furthermore does not wrap around methods, our first measurements indicate a smaller runtime overhead than using aspects. Finally, it is also possible to remove unnecessary source code automatically if not needed (e.g., if we are only interested into why-provenance, but not in where-provenance).

## 5 Conclusion and future work

This paper addresses the challenge of capturing provenance efficiently in existing complex IT-environments. While in previous work we focused on how to exchange provenance annotated data with the help of invertible watermarks in a reliable way [9], this paper paper deals with efficient provenance capturing in one system. We present a first approach that allows for provenance capturing in object-oriented programs beyond black-box assumption. Furthermore, we introduce evaluation criteria and a reference approach to qualify software engineering technique that can be used to efficiently integrate the provenance. Based on the distribution of the provenance concern, we introduce different classes of systems that act differently w.r.t. to the evaluation criteria. For instance, if the provenance concern is represented by the structure of the program (classes and methods) and the performance is not a major challenge using aspects (e.g. AspectJ) is beneficial. In contrast, if performance is a major challenges using aspects is problematic mainly due to reflection, which has a significant drawback on performance.

For future work, we intend to examine how the provenance concern is distributed in existing systems and for different producer consumer relationships. Based on these results and our system classes we will work out a catalog that shall help programmers to choose an adequate engineering technique to integrate the provenance concern. Additionally, we will state properties of a program that allow for easy provenance integration. Furthermore, we will determine how programming conventions, such as always use `get() set()` methods to access class members, simplify our provenance capturing. Finally, we need to examine technical limits and their practical relevance of our approach.

[1] http://java.sun.com/developer/technicalArticles/ALT/Reflection/

## References


[1] AMSTERDAMER, Y., ET AL. Putting lipstick on pig: Enabling database-style workflow provenance. *Proc. VLDB Endow. 5*, 4 (2011), 346–357.

[2] BITON, O., ET AL. Querying and managing provenance through user views in scientific workflows. In *ICDE* (2008), pp. 1072–1081.

[3] GREEN, T. J., ET AL. Orchestra: facilitating collaborative data sharing. In *SIGMOD Demonstration* (2007), pp. 1131–1133.

[4] GREEN, T. J., ET AL. Provenance semirings. In *PODS* (2007), pp. 31–40.

[5] KICZALES, G., ET AL. An overview of AspectJ. In *ECOOP* (2001), pp. 327–353.

[6] LUDÄSCHER, B., ET AL. Scientific workflow management and the kepler system. *Concurr. Comput. : Pract. & Exper. 18*, 10 (2006), 1039–1065.

[7] MELIOU, A., ET AL. Causality in databases. *IEEE Data Eng. Bull. 33*, 3 (2010), 59–67.

[8] MOREAU, L., ET AL. The open provenance model core specification (v1.1). *Fut. Gen. Comp. Sys. 27*, 6 (2010), 743–756.

[9] SCHÄLER, M., ET AL. Reliable provenance information for multimedia data using invertible fragile watermarks. In *BNCOD* (2011), vol. 7051 of *LNCS*, pp. 3–17.

[10] SCHÄLER, M., ET AL. A hierarchical framework for provenance based on fragmentation and uncertainty. Tech. Rep. FIN-01-2012, School of Computer Science, Univ. of Magdeburg, 2012.

# A Terminology

Table 1: This table lists the terms used to collect the provenance data from Section 2.2 in a more formal way.

| Term | Definition | Description |
|---|---|---|
| Point of interest | id | Unique id of the PoI. |
| | name | Fully qualified name of the PoI. |
| | return type | Type of the return value. |
| | arguments | List of argument types. |
| Function execution | id | Unique id of the function execution. |
| | id | Reference to the PoI that triggered the provenance collection event. |
| | Input | List of input artifacts for this function execution. |
| | output | List of output artifacts for this function execution. |
| | begin time | Timestamp when execution of this function starts. |
| | stop time | Timestamp when execution of this function is finished. |
| Artifact | - | Each artifact either is a complex or a simple artifact. |
| Simple artifact | id | ID, unique for all artifacts. |
| | name | Optional name. |
| | type | Type or role: E.g. for Java (int, double, etc.). |
| | value | Value of this simple artifact, interpretation depends on the type. |
| Complex artifact | id | ID, unique for all artifacts. |
| | name | Optional name. |
| | type | Type or role: E.g. for Java everything that is called by reference. |
| | simple children | Contained simple artifact. |
| | complex children | List of IDs pointing to contained complex artifacts. |
| Call | invoker | ID of the invoking function. |
| | called function | ID of the called function. |
| Current values | function | Reference to the executing function. |
| | complex artifacts | Sublist of IDs of input artifacts called by reference. |
| | hash | Hash value representing current state. |
| Exogenous input | function | Reference to the executing function. |
| | artifacts | Sublist of IDs of input artifacts that can be omitted in computation without changing the result. |

Table 2: This table lists functions used to collect the provenance data.

| Name | Definition | Description |
|---|---|---|
| identity | $identity:$ $artifact \times artifact \rightarrow \{true \mid false\}$ | Returns true if both artifact have the same ID, else false. |
| weak_identity | $weak\_identity:$ $artifact \times artifact \rightarrow \{true \mid false\}$ | Returns true in case of the same structure and all contained simple artifacts have the same value. |
| value | $value:$ $artifact \rightarrow N$ | Returns a hash value that represents the current state of the contained simple artifacts. |