



Automated and non-intrusive provenance capture with UML2PROV

Carlos Sáenz-Adán¹ · Francisco J. García-Izquierdo¹ · Beatriz Pérez¹  · Trung Dong Huynh² · Luc Moreau²

Received: 26 February 2021 / Accepted: 2 September 2021 / Published online: 10 December 2021
© The Author(s) 2021

Abstract

Data provenance is a form of knowledge graph providing an account of what a system performs, describing the data involved, and the processes carried out over them. It is crucial to ascertaining the origin of data, validating their quality, auditing applications behaviours, and, ultimately, making them accountable. However, instrumenting applications, especially legacy ones, to track the provenance of their operations remains a significant technical hurdle, hindering the adoption of provenance technology. UML2PROV is a software-engineering methodology that facilitates the instrumentation of provenance recording in applications designed with UML diagrams. It automates the generation of (1) templates for the provenance to be recorded and (2) the code to capture values required to instantiate those templates from an application at run time, both from the application's UML diagrams. By so doing, UML2PROV frees application developers from manual instrumentation of provenance capturing while ensuring the quality of recorded provenance. In this paper, we present in detail UML2PROV's approach to generating application code for capturing provenance values via the means of Bindings Generation Module (BGM). In particular, we propose a set of requirements for BGM implementations and describe an event-based design of BGM that relies on the Aspect-Oriented Programming (AOP) paradigm to automatically weave the generated code into an application. Finally, we present three different BGM implementations following the above design and analyze their pros and cons in terms of computing/storage overheads and implications to provenance consumers.

Keywords Data provenance · Application logging · Technology audits

Mathematics Subject Classification 68p20

1 Introduction

Over the last few years, there has been a growing interest in the origin of data, in order to enable its rating, validation, and reproducibility. In this context, the term *provenance* has emerged to refer to “the information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness” [17]. This interest in provenance has led to various solutions developed to capture provenance (such as PASS [29] and SPADE [14], tools explicitly developed for provenance purposes, or the Kepler provenance system [1], an extension of the Kepler system to provide it with provenance capabilities). The need for interoperability between systems has been a driver for the creation of the PROV standard [17], a conceptual data model for provenance, aiming at the interoperable exchange of provenance information. Toolkits supporting PROV [34] have been developed to facilitate the software engineer’s task of creating, storing, reading, and exchanging provenance data; however, such toolkits do not help decide what information should be included in provenance, and how software should be designed to allow for its capture. Therefore, the ability to instrument provenance capturing into applications, especially during the design phase of software engineering, has become critically important to support software designers in building provenance-enabled systems.

PrIME [26] is the first provenance-focused methodology for adapting applications to make them provenance-aware. Although the application of PrIME has demonstrated promising results, it is standalone and does not integrate with existing software engineering methodologies, making it challenging to be adopted in practice. In contrast, design proposals have been put forth to shorten the development time of software products, and to increase their quality, avoiding developers from expending extra time and efforts during subsequent phases. Some of these proposals use the Unified Modelling Language (UML) [31] that, despite being considered as the *de facto* method for designing object-oriented software systems, offers no specific support for provenance. In fact, our experience in developing software applications augmented with support for provenance is that the inclusion of provenance within the design phase entailed significant changes to an application design [26]. This is a cumbersome task for the designers and programmers alike, since they have to be knowledgeable about provenance, deal with complex diagrams, and maintain the application’s provenance-specific code. In short, the gap between software engineering design methodologies and provenance engineering may result in applications generating provenance that is not aligned with what the application actually does, or that is not fit for purpose.

Against this background, PROV-Template [28] allows the structure of provenance to be described declaratively: a provenance *template* is a document containing placeholders (referred to as *variables*). An *expansion algorithm* instantiates a *template* with values, which are contained in *bindings* associating *variables* with concrete values. Although this approach reduces the development and maintenance effort, it still requires designers with provenance knowledge.

UML2PROV [37] is a more recent software-engineering methodology the authors proposed to facilitate the instrumentation of provenance recording in applications by addressing the gap between application design and provenance design. It automates the generation of (1) PROV templates for the provenance to be recorded in a given

application and (2) the code to capture values required to instantiate those templates at run time, both from the application's UML diagrams. The latter is produced in the form of a *Bindings Generation Module (BGM)* to be linked with the application to capture the bindings required for provenance data during its execution (Sect. 3).

UML2PROV methodology is the subject of several previous publications [36,37,39], which also include its evaluation as such methodology. However, those publications only partially described the definition of the *BGM* and provided a succinct description of its design. Here, we focus on this critical UML2PROV component to generate code for capturing provenance values. Concretely, the contributions of this paper are the proposal of: (1) a set of requirements for any *BGM* implementation to work with UML2PROV (Sect. 4), (2) an event-based design of *BGM* that relies on the Aspect-Oriented Programming (AOP) paradigm to automatically weave the generated *BGM* into an application (Sect. 5), and (3) three different *BGM* implementations following the above design. Finally, our proposal is analyzed in terms of computing and storage overheads and implications to provenance consumers (Sects. 6 and 7). The paper finishes by discussing the related work (Sect. 8) and presenting conclusions and future directions for this work (Sect. 9).

2 Background

PROV [17] is a World Wide Web Consortium standard that aims to facilitate the publication and interchange of provenance between applications. *PROV* is specified in a family of documents, including the *PROV Data Model (PROV-DM)* that forms the basis for the remainder specifications, and a human-readable notation for the provenance model (*PROV-N*). Figure 2d depicts a graphical *PROV* document with the three key elements defined by *PROV-DM*: (1) an *Entity*, which is a physical, digital, conceptual or other kind of thing with some fixed aspects (yellow oval); (2) an *Activity*, defined as an occurrence of something taking place over a period of time and acting upon or with *entities* (blue rectangle); and (3) an *Agent*, something that bears some form of responsibility for an *activity*, an *entity* or another *agent* (orange pentagon). As shown in Fig. 2d, these concepts are related to one another through relationships such as *used*, which represents an activity beginning of utilizing an entity, or *wasGeneratedBy*, a new entity was produced by an activity.

The *PROV-Template* approach [28] builds on top of *PROV* as a templating system for provenance, and helps differentiate the provenance design from the creation of provenance data. It consists of three main elements. First, *PROV templates* offer a language to design the provenance to be generated. They are provenance documents, expressed in a *PROV-compatible* form, that contain placeholders (referred to as variables) for values. Second, *bindings* are associations between a template's variables and value(s), and are usually grouped in *sets of bindings*. Finally, an *expansion algorithm* [28] replaces each variable from the templates with data values from the bindings, generating an expanded *PROV document*. Figure 2b depicts a template as a *PROV document*, where the prefix *var* identifies variables. It shows, e.g., an activity *var : operation* that *wasAssociatedWith* the agent *var : senderObject*, *used* an entity *var : starter* for its execution, and generated another entity *var : response*. Starting from this

template and the values associated to its variables given by the bindings collected during the application execution (Fig. 2c), the expansion algorithm generates the expanded PROV document (Fig. 2d).

The *Aspect Oriented Programming* (AOP) paradigm [21,22] promotes software design so that the designer focuses on the functional concerns (*core*) of a system as opposed to non-functional concerns (e.g., logging or security). Non-functional concerns tend to cut across the system rendering it difficult to understand, maintain, and modify. AOP allows a developer to modularize these crosscutting concerns into entities called *aspects*, which can then be “woven into” the core code by an *aspect weaver*, building the final system. AspectJ [23] is an AOP Java extension that expresses crosscutting mainly through *join points* and *pointcuts*. While *join points* are well-defined points in a program execution (e.g., objects creation or methods call), *pointcuts* are distinguished selections of join points that meet some specified criteria (e.g., the call of a method with a certain name or with a parameter of a concrete type). A *pointcut* expression starts with a pointcut designator, a keyword that tells AspectJ what to match (e.g., *call*, to select operation calls, or *initialization*, to select constructor invocations). AspectJ also uses *wildcards* to construct the *pointcuts* in order to capture the *join points* that share common characteristics (e.g., *** and *..*). For instance, the *pointcut* `call (* *.setName(..))` captures all calls (*call* designator) to operations with the name `setName`, regardless of the access modifier (first *** wildcard), the class to which it belongs (second *** wildcard), and number of parameters (*..* wildcard). After a pointcut captures join points, the focus is augmenting them with additional or alternative behaviour (e.g., that related to logging or security). *Advices* are method-like constructs defining such a complementary crosscutting behavior at join points. Depending on the declaration, advice bodies are executed *before* or *after* a specified join point, or they can surround (*around*) a join point. In our previous example, if we define a *before* advice associated to our *pointcut*, the advice instructions will be executed before a `setName` method starts running. Additionally, the behaviour inside the *around* advice could proceed with the actual behaviour when it considers it necessary. To do this, AspectJ defines the `proceed()` statement for carrying out the actual behaviour. Finally, *aspects* embed crosscutting logic by defining the *pointcuts* and *advices* [23].

3 UML2PROV architecture

The use of UML2PROV involves both *design time* and *runtime* scenarios (red background with a stripped texture, and blue plain background, respectively, in Fig. 1), each one identifying different key facets and stakeholders involved in the process –the *software designer* and the *developer* at the beginning, and the *provenance consumer* at the end.

Starting point. UML2PROV starts from the UML design of the application to be made provenance-aware. Such design can be the one used to guide the development of the application or, in case of legacy applications built without UML, the one obtained by means of reverse engineering [3]. Among the *UML diagrams* considered to extract data provenance, we have focused on those that not only have a strong relation with

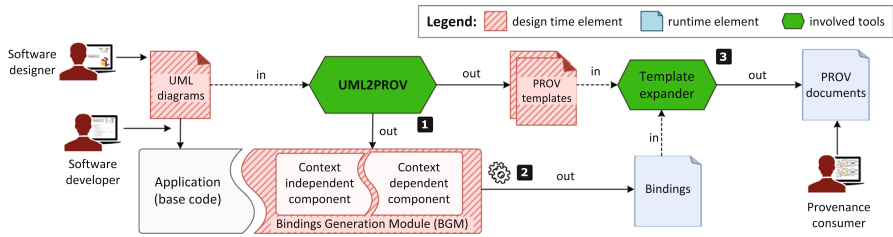






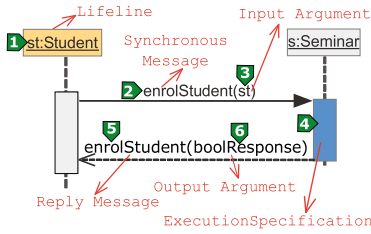


Fig. 1 The UML2PROV approach

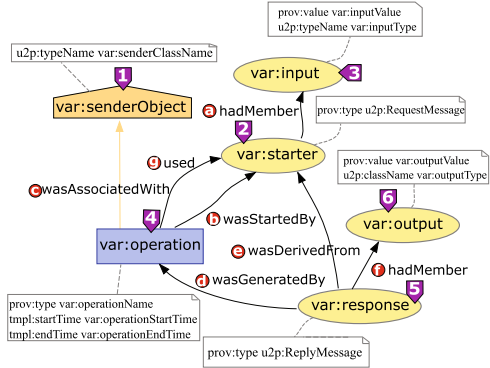
provenance, but are also mostly used by *software designers* [35]: Sequence (SqD), State Machine (CD), and Class (CD) Diagrams. Taking the *UML diagrams* as input, UML2PROV automatically generates (*Step 1* in Fig. 1): (1) the *PROV templates* with the design of the provenance to be generated, and (2) the *BGM* responsible for capturing provenance according to the *PROV templates*.

Provenance design. Our proposal for obtaining *PROV templates* from the *UML diagrams* relies on a set of 17 transformation patterns that ultimately associate *UML elements* with *PROV elements* (see [38] for full details). UML2PROV advocates designing provenance around *the executions of operations* for two main reasons: (1) the execution of concrete behaviour (operation) drives the provenance of a piece of data, and (2) the three supported types of diagrams provide elements for modeling different perspectives of the execution of an operation. Each transformation pattern identifies a concrete *situation*, usually presented in systems modelling, which can be addressed by a *UML diagram*: four of them are modelled by *SqDs*, three by *CDs*, and ten by *CDs*. Figure 2 illustrates one of our patterns presenting the translation of an excerpt of a *SqD diagram* (Fig. 2a) into *PROV Templates*. This diagram, slightly modified from [42], depicts the situation in a University application when a *Student* (sender) enrolls in a *Seminar* (recipient) by calling *enrolStudent* (operation) in the recipient; the sender waits for a response. This situation is addressed by the *Sequence Diagram Pattern 2 (SeqP2)* [38] that, as all our patterns, defines the correspondence between each *UML element* involved in the transformation (denoted by a numeric identifier inside a green label ) and a *PROV element* in the *PROV template* (labelled in purple with the same identifier ). For example, the *enrolStudent* operation execution *ExecutionSpecification*  activated by the *Lifeline*  (*Student*) of Fig. 2a is translated into a *prov:Activity* identified as *var:Operation*  in the *PROV Template* of Fig. 2b, which is associated with the *Student var:SenderObject* .

Provenance capture. The same *UML diagrams* used to obtain the provenance design are also taken as source to generate the *Bindings Generation Module (BGM)* for the application. This module is automatically and non-intrusively integrated into the application, which does not require any source code modification. The ultimate goal of the *BGM* is capturing provenance data during the application execution. When the application is running, the *BGM* generates bindings, i.e. variable-value pairs which associate concrete values collected from the execution (provenance data) with variables in a *PROV template* (*Step 2*). An example of bindings for the template of



a Excerpt of a *SqD* showing the interaction between *Student* and *Seminar*

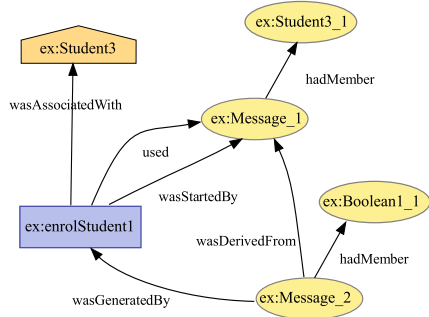


b PROV template obtained by applying *SeqP2* to the *SqD*

```

{"var":{
  "senderObject": [ { "@id": "ex:Student3" } ],
  "senderClassName": [ { "@type": "xsd:string",
    "@value": "Student" } ],
  "starter": [ { "@id": "ex:Message_1" } ],
  "input": [ { "@id": "ex:Student3_1" } ],
  "inputValue": [ { "@type": "xsd:string",
    "@value": "Carlos S." } ],
  "inputType": [ { "@type": "xsd:string",
    "@value": "Student" } ],
  "operation": [ { "@id": "ex:enrolStudent1" } ],
  "operationName": [ { "@id": "ex:enrolStudent" } ],
  "operationStartTime": [ { "@type": "xsd:dateTime",
    "@value": "..." } ],
  "operationEndTime": [ { "@type": "xsd:dateTime",
    "@value": "..." } ],
  "response": [ { "@id": "ex:Message_2" } ],
  "output": [ { "@id": "ex:Boolean1_1" } ],
  "outputValue": [ { "@type": "xsd:boolean",
    "@value": "true" } ],
  "outputType": [ { "@type": "xsd:string",
    "@value": "Boolean" } ]
},
"context":{
  "xsd": "http://www.w3.org/2001/XMLSchema#",
  "u2p": "http://uml2prov.unirioja.es/ns/u2p#",
  "ex": "http://example.com"
}}
    
```

c Possible set of bindings obtained from the execution of the *enrolStudent* operation.



d PROV document (without properties), resulting after expanding the PROV template with the bindings.

Fig. 2 UML2PROV process: **a** UML diagram; **b** corresponding PROV template; **c** set of bindings; **d** the expanded PROV document

Fig. 2b is presented in Fig. 2c, which include provenance values captured during an hypothetical execution of the University application (e.g., *var:senderObject* is related to *ex:Student3*, and the variable *var:operation* is associated with the value *ex:enrolStudent1*). This paper focuses on the strategy to define the *BGM*, as a way to enrich applications with provenance capture capabilities, distinguishing between conceptual and implementation aspects.

Provenance generation. Finally, the *expansion algorithm* [28] takes as input the PROV templates and the bindings, and generates the *PROV documents* for the executed application (*Step 3* in Fig. 1). The resulting PROV documents include high-quality provenance ready to be exploited by the *provenance consumer*. For example, Fig. 2d shows a PROV document (without properties) obtained from the template in Fig. 2b and the bindings in Fig. 2c.

4 BGM principles and requirements

The wide range of applications that may require provenance instrumentation makes it impossible to provide a single generic implementation for the above-mentioned *BGMs* that is suitable for all situations. Instead, we propose a set of principles to drive future *BGM* implementations. These principles are inspired by the findings and conclusions drawn from the taxonomy of provenance systems published by Pérez et al. in [32], concretely in its *data capture* dimension, which refers to the way in which provenance data can be captured. This dimension distinguishes four categories, *Tracing*, *Level*, *Mechanism*, and *Technique*, which embody our principles.

Principle 1 *Freedom to decide when to compute provenance.* As the *Tracing* category of the taxonomy in [32] points out, provenance systems generally employ two strategies to decide when to compute final provenance data: when it is required (usually referred to as *lazy*) or immediately (*eager*). Many readers will be familiar with these strategies, since they appear in other contexts such as Object–Relational Mapping (ORM) tools, for loading data; or in the Apache Spark framework [2], for the lazy evaluation of *transformations*, definitely computed when Spark *actions* are executed. In our context, the provenance information is computed when the PROV templates are expanded with bindings to obtain the final PROV documents. *BGMs* must allow developers to choose when to compute provenance, regardless of when and how bindings are recorded. For example, the computation of provenance could be delayed until the application finishes (the *lazy* approach), which would require storing the bindings, individually or in sets, until the provenance consumer decides to expand the corresponding PROV templates; or, these templates could be expanded as the application is being executed (the *eager* approach), thus avoiding the requirement to store bindings.

Principle 2 *Capture of provenance data from the executions of operations.* Among the literature, most provenance systems gather provenance data at database-level, workflow-level, and OS-level in the software stack (taxonomy [32], *Level* category). In our domain, a database-level approach is out of scope since these systems track provenance in database/data warehouse scenarios. On the other hand, workflow-level and OS-level were ruled out because workflow-level approaches are strongly dependent from WfMS, and OS-level systems capture very low-level information for our purposes. Taking these facts into account, and that UML2PROV advocates designing provenance for operation executions, Principle 1 states that the *BGM* must generate bindings based on data obtained from the executions of the operations of a process (i.e., at *process-level*). In this way, UML2PROV will be independent from WfMS, unlike the workflow-level, and it will also be able to capture high level meaning of the process, as opposed to the OS-level.

Principle 3 *Generation of bindings relying upon application's internal structures.* The mechanism used to capture provenance data could rely on *internal structures* or *external services* (taxonomy [32], *Mechanism* category). The *BGM* is meant to be integrated into the existing application, thus relying on *internal structures* for generating the bindings.

Table 1 Requirements established for the *BGM*

Req	Description
R1	The instrumentation of the application to add the binding generation instructions must be carried out automatically
R2	The instructions for bindings generation must be located apart from the application's source code, in an independent module, avoiding the generation of repetitive and obfuscated code
R3	The BGM has to be able to identify the specific points within the application's source code where such instructions must be included
R4	The BGM must provide software developers with mechanisms to select the configuration that best suits their needs, allowing them to decide when to compute the provenance
R5	Each binding obtained from an application's execution must be associated with at least one PROV template automatically generated from the UML diagrams
R6	The variables included in a set of bindings must correspond with the variables in their associated PROV templates

Principle 4 *Automatic annotation of existing data when generating bindings.* The *Technique* category in the taxonomy [32] specifies two techniques to capture provenance: the *inversion* approach, that relies on inversion functions to find the history of derivations of a data product; and the *annotation* approach, where metadata about the evolution of a data product are collected as *annotations* and descriptions about its source and processing. An annotation is a *name-value* pair [8], as also are bindings. We advocate for annotating data in a fully automated way, instead of in a manually or partially automated manner.

To ensure that these principles are respected, our proposal imposes a number of requirements every *BGM* implementation must meet (Table 1) in order to guarantee that provenance capture is performed in an automatic and non-intrusive manner, without modifying software designers' and developers' *modus operandi*, without affecting the maintenance of the application, and ensuring consistency between the designed provenance and the generated bindings. Consequently, the requirements are organised into the following categories:

Automatic instrumentation. The instrumentation of an application's code to generate bindings may be performed manually or automatically. A manual adaptation is a tedious, time-consuming and error-prone task for developers, who would have to work hard on traversing the whole source code, carefully analysing it to add suitable instructions to generate the bindings structures. In addition, manual code adaptation negatively impacts maintainability, since changes in the application code may affect the added provenance-specific instructions, which could need to be adapted, also manually. *Requirement 1 (R1)* in Table 1 aims to avoid this problem.

Non-intrusive instrumentation. Relying upon the *internal structures* of an application for generating bindings (*Principle 3*) could result in provenance capture code

scattered throughout the source code, since bindings must be generated at concrete points distributed over the application's code. This fact would make it difficult to maintain the application. Similarly, the need for adapting the application's design to the changes it may face over time could lead software designers to modify design elements involved in the generation of bindings. To avoid inconsistencies between the evolving application's design and the existing instructions for capturing bindings, designers should identify the concrete points in the code for bindings capture, so that they may be adapted. *Requirements R2* and *R3* have been considered to avoid these inconveniences.

Provenance computation. The *BGM* must be agnostic about when to compute provenance, that is, when to expand the PROV templates (*Principle 1*). Software developers should be able to choose between following a lazy or an eager approach. *Requirement 4 (R4)* recognises this optionality.

Consistency. As *Principles 2* and *3* claim, consistency between the PROV templates and the generated bindings must be guaranteed to ensure the production of coherent PROV documents after expansion. First, since UML2PROV focuses on the executions of operations to generate the PROV templates with the provenance design, the *BGM* must collect the provenance data from the same operation executions (*Principle 2*). To generate coherent provenance, there must be a correspondence between the provenance data captured by the *BGM* and PROV templates created by UML2PROV. This is guaranteed by *Requirements 5* and *6 (R5 and R6)*. Second, our proposal advocates for automatically annotating the existing data when generating bindings (*Principle 4*). For example, the set of bindings depicted in Fig. 2c meets *R5* and *R6*: it was generated during the execution of the `enrolStudent` operation of our example, and it links each variable of the template in Fig. 2b with the corresponding value collected during such execution.

5 An event-based design and implementation for *BGM*

An earlier implementation of the *BGM*, based on XSLT [47] and the *Proxy-pattern* [13], was presented in [36,39]. However, XSLT turned out not to be suited for model (UML diagrams) to text (code) transformations, whereas the *Proxy-pattern* requires a manual instrumentation of the code (thus not meeting requirements *R1-R3*). The proposed *BGM* design and implementation presented here not only fulfils all the requirements stated for the *BGM* (Table 1), but it also allows developers to follow different approaches (called *configurations*) to manage provenance. This approach, briefly presented in [37], is a generic *event-based* proposal developed on top of the AOP paradigm [21] to generate bindings without user intervention.

Our proposal of *BGMs* implementation is mainly based on two key elements: *events*, i.e., notable occurrences that happen while the application is running, and *listeners*, which specify the behaviour for processing the events. Thus, provenance capture is decoupled from the actual generation of provenance data, making it possible for an application to have simultaneously several listeners that manage provenance in different ways. E.g., this facilitates development in scenarios where applications composed of several modules are running. The provenance consumer may be interested in directly

obtaining provenance documents for immediate consumption, but only from a specific module of the application. For the rest of the modules, he wants to store the bindings information, perhaps in a secondary storage system, just for possible future queries. The development of two listeners would facilitate this different processing, making it modular.

Since UML2PROV advocates capturing provenance during the executions of operations (represented as PROV activities), four types of events that may occur during an operation execution have been identified, each one corresponding to a type of variable within the PROV activity element [25].

- *operationStart/operationEnd*. They refer to the start and the end of an operation execution. They are important when developers want to create and store sets of bindings associated with a concrete operation execution, instead of storing each binding independently (see Sect. 5.1).
- *newBinding*. This event type indicates the collection of a provenance value that will identify a PROV element. For instance, the collection of a value associated with `var:operation` in Fig. 2b will trigger an event of this type, since `var:operation` occurs in a mandatory identifier position.
- *newValueBinding*. It indicates the collection of a provenance value that will not identify a PROV element. For instance, obtaining values associated with `var:operationEndTime` and `var:operationName` in Fig. 2b fires *newValueBinding* events because they occur in non-identifier positions.

According to the UML2PROV architecture depicted in Fig. 1, our *BGM* implementation comprises the *context-independent* and the *context-dependent* components. The former are the elements that do not depend on the source *UML diagrams*, and are therefore common for every *BGM*. These are the *BGMEventListener*, *BGMEvent*, and *BGMEventManager* (depicted in white background in Fig. 3). The latter are the elements whose implementations depend on the source *UML diagram models*. In this implementation, the single element needed is *BGMEventInstrumenter* (in dark background in Fig. 3).

BGMEventListener. In UML2PROV, the mechanisms used to manage provenance are performed through the so-called *listeners*, which are classes that implement the *BGMEventListener* Java interface. They are registered in the *BGM* thanks to the *BGMEventManager* (see the UML2PROV User Guide [46] for more details). While the instrumented application is running, these registered listeners collect the aforementioned events generated by the *BGM*, more concretely by the *BGMEventInstrumenter*. The *BGMEventListener* Java interface defines four operations for managing each type of event (*operationStart*, *operationEnd*, *newBinding*, and *newValueBinding*). Such operations have a *BGMEvent* input parameter that encapsulates the provenance data to be processed. The implementation of these operations constitutes the mechanism used by a concrete listener to generate, manage, and store the bindings data carried by the *BGMEvents*. In the end, classes that implement the *BGMEventListener* allow developers to choose whichever suitable strategy for computing provenance (i.e., when to expand the PROV templates with the collected bindings), thus fulfilling *R4*. This way, unlike other provenance systems [32], our proposal does not couple this task to a concrete persistence infrastructure. Section 5.1 exemplifies three different imple-

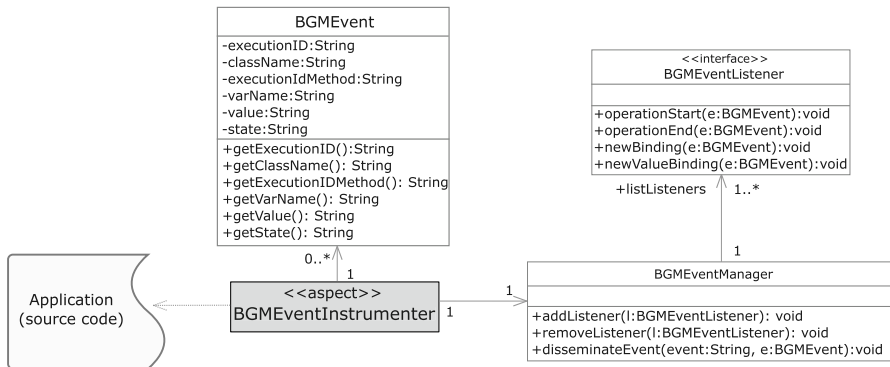


Fig. 3 UML Class diagram depicting our implementation for a *BGM*

mentations of this interface (*configurations*), which ultimately manage provenance capturing in three different ways.

BGMEvent. Objects of this class are used to carry information about the occurrence of events. They encapsulate the provenance data necessary for constructing the corresponding bindings. Attributes of the *BGMEvent* are, for instance, `varName` and `value`, for the name and value of the variable to which the binding corresponds (see Fig. 3). For example, in the case of a *newBinding* event, a *BGMEvent* object could have “`var: operation`” as the value of its attribute `varName`, and the value “`ex: enrolStudent1`” in its attribute `value` (you can see this `varName`-`value` attribute pair in the set of bindings of Fig. 2c).

BGMEventManager. Helper class that manages the list of subscribed *listeners*, and disseminates the *BGMEvent* objects among them as events occur.

BGMEventInstrumenter. With our event-based approach, instrumenting an application to include provenance capture functionality would require to identify application classes, and those places within them, where *BGMEvent* objects have to be fired. Applications’ source code should include additional instructions to construct such events with the provenance data, and to disseminate them among the *BGMEventListeners* using the *BGMEventManager*. Requirement *R1* mandates that such instrumentation tasks are performed in an automatic fashion, which prevents a developer from traversing the whole code to include such instructions. This, besides being tedious, time-consuming and error-prone, would lead to the provenance instructions being scattered throughout the application’s classes, making their maintenance a cumbersome task. The use of the AOP paradigm [21] for implementing the denominated *BGMEventInstrumenter* removes such inconveniences. Provenance capture instructions are placed apart from the application’s source code, which, thanks to AOP, automatically adopts events triggering functionality. AspectJ [45] has been used to implement the *BGMEventInstrumenter* by means of an *aspect*, which is made up of an *advice* with a specific *pointcut* (see Fig. 4). The *pointcut* identifies locations within the application code where events must be fired to collect provenance data. The *call* and the *initialization* *pointcut* designators are used to select operation calls and constructor invocations, respectively (remind that UML2PROV advocates capturing provenance from operation executions). Thus, in the *pointcut* of Fig. 4,



Fig. 4 Structure overview of the *BGMEventInstrumenter* in Aspect

the identified operation calls are denoted as “<class>.<operation>”, being <class> the full class name of the classes in the UML design, and <operation> the name of each identified operation. Likewise, invocations to constructors of classes in the UML design are denoted as <class>.new. The actual *pointcuts* generated by UML2PROV will include one `call(...)` expression per each identified operation, and one `initialization(...)` per each class in the UML design. For example, considering our University application, the defined *pointcut* would include in its definition the expression `call(* Seminar.enrolStudent(..)) || initialization (Seminar.new(..))` so that it can capture calls to the `enrolStudent` operation and invocations to the `Seminar` constructor. Since the events can occur both before and after operation calls and constructor invocations, an *around advice* has been used associated to such a *pointcut* (see Fig. 4) to specify the custom behaviour to be executed before and after the actual behaviour. More specifically, while the actual behaviour is represented by the `proceed` statement, the custom behaviour is provided by the methods `behaviourBeforeExecution` and `behaviourAfterExecution` (see Fig. 4). This custom behaviour involves precisely the aforementioned construction of *BGMEvent* objects and their dissemination to the *BGMEventListeners* (by invoking the `disseminateEvent` operation of

the `BGMEventManager`). Finally, the AspectJ *weaver* automatically integrates the behaviour from the aspect into the locations specified by the pointcut at compilation time. In this way, our AOP approach does not require a manual intervention to adapt the source code (*R1*); it collects provenance data in an automatic and transparent for software developers way (*R2* and *R3*). Also, since the `BGMEventInstrumenter` is created from the UML design, just as the PROV templates, and it considers the structure of such templates, it is ensured that the collected bindings are associated with at least one PROV template, and that the variables included in the bindings correspond to those in the templates (*R5* and *R6*).

5.1 Implementation of representative configurations

Developers may configure their *BGMs* to manage provenance in different ways by employing a different implementation of the `BGMEventListener` interface. This section presents three of such possible implementations, called *configurations*: the classes `BindingsConfiguration`, `SetBindingsConfiguration`, and `ProvenanceConfiguration`. Serving as a guide for developers, they illustrate different alternatives from the provenance capture point of view, and present different temporal and spatial overheads that should also be considered when devising the instrumentation strategy (see Sect. 6). The review of provenance systems by Pérez et al. [32] identifies several categories related to temporal and spatial overheads: *granularity* of provenance data, *technique*, *level*, and *tracing*. The first three are intrinsic aspects of the UML2PROV proposal (fine-grained data, annotations at process-level), but the tracing may be configured to be either on demand (*lazy*) or as the application is running (*eager*) just by implementing the `BGMEventListener` interface differently.

Concretely, two *lazy* (`BindingsConfiguration` and `SetBindingsConfiguration`), and one *eager* (`ProvenanceConfiguration`) configurations are presented. These are illustrated in Fig. 5, each one including two main blocks: the “Operation Execution” block represents the behaviour carried out during the execution of a tracked operation, from the start to the end of the operation execution; and the “Provenance consumer tasks” block includes the behaviour executed to exploit the provenance information, which takes place after the execution of the tracked operation.

- ***BindingsConfiguration***. This configuration expands PROV templates on demand (the *lazy* approach). Each time an `BGMEventListener` operation is executed, the data contained in the `BGMEvent` is sent to a database in the form of *binding* (“Operation Execution” block in Fig. 5a). The templates expansion may be performed later, when the final *PROV documents* are needed (“Provenance consumer tasks” block). Then, the provenance consumer has to: (1) retrieve all binding from the database, (2) create the set of bindings, and (3) expand the *PROV templates* with the set of bindings.
- ***SetBindingsConfiguration***. It also follows the *lazy* approach, but in contrast to the first configuration, bindings data received in the `BGMEvents` are stored in memory until the `operationEnd` event, signalling the end of execution of the tracked operation, is listened. Then, the accumulated set of bindings is shipped to the database (Fig. 5b). The PROV documents generation is now simpler: (1) retrieve

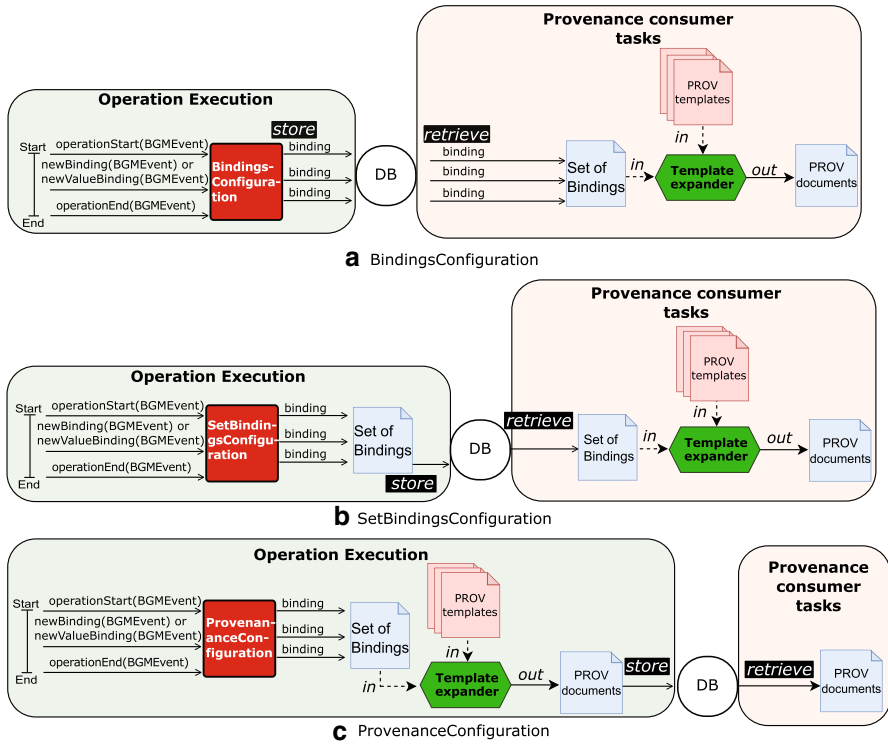


Fig. 5 Graphical representation of the three defined configurations

the set of bindings from the database, and (2) use this set of bindings to expand the PROV templates.

- **ProvenanceConfiguration.** This configuration (Fig. 5c) expands PROV templates as the application is running (the *eager* approach). Bindings data received in the events are also accumulated in memory, and when the tracked operation execution ends (*operationEnd* event), the *PROV templates* are expanded with the set of bindings then generated, and the resulting PROV documents are sent to a database. Thus, consuming provenance only requires retrieving the PROV documents from the database.

5.2 Model-driven implementation of the BGM

Our proposal advocates automatically generating the *BGMs* code using existing mapping and transformation languages created by the MDD community. They have better properties in terms of maintenance, reusability, and support to software development [43] than the previous UML2PROV implementation based on XSLT [39]. As stated in [43], *model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing*. The Xtend MDD-based tool [48] has been used to implement a model-to-text (M2T) transformation

module which takes the application's UML model as source, and automatically generates the Java and AspectJ code of the *BGM* for the modelled application. To reduce code dependencies, this Xtend based module generates the *BGM* as a single Java library (*jar* file). This library will contain both the three elements of the *context-independent component*, common for every *BGM*, and the *context-dependent component* (i.e., the *BGMEventInstrumenter* aspect) which corresponds to the concrete application's UML diagram models. Integrating the generated library into the application is straightforward by using the AspectJ compiler to weave the generated *BGM* with the original application, thus obtaining the instrumented application ready to produce provenance data (for more information, see the Supplementary Material [38] and the UML2PROV User Guide [46]).

6 Evaluation

The work in [37] provides quantitative data and qualitative arguments to show the benefits and trade-offs of applying UML2PROV, focusing mainly on how the level of detail of several UML designs of an application affect aspects such as provenance design generation, application instrumentation, maintenance of provenance capabilities, storage and run-time overheads, and quality of the generated provenance. In that evaluation only one of the configurations presented in Sect. 5.1, the *SetBindingsConfiguration*, was studied. All of them will now be compared, in order to analyze how the three representative configurations for managing the collection of provenance may impact on the application, considering aspects such as run-time and storage overhead, and provenance consumption effort.

Here, as well as in [37], a Bioinformatics application called GelJ [18] is used as case study. GelJ is a platform-independent, open-source tool that arose for analyzing DNA fingerprint *gel-images* [18]. Its main component is the *experiment wizard*, which is used to perform the analysis of gel-images through different steps that take as source a gel-image and return an *experiment* consisting of the source gel-image together with a set of detected bands on it. This evaluation has been conducted taking GelJ's UML design as starting point, and applying UML2PROV on it with the three different configurations to be compared. Each resulting version of the provenance-aware GelJ has been used to create a representative experiment (the same that is described in [37]). This use case involves the execution of about 46,000 operations in the GelJ's source code, about which provenance data is collected.

6.1 Analysis

This evaluation has been run on a personal computer, Intel(R) Core(TM) i7 CPU, 2.8GHz, with Oracle JDK 1.8 and a Windows 10 Enterprise OS running MongoDB [27] and using JSON to serialize bindings (comparing other storage systems or serialization formats is not in the scope of this paper).

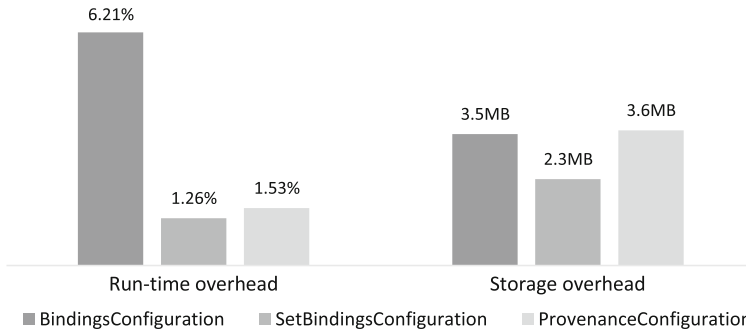


Fig. 6 Comparison of Run-time overhead (%) and Storage overhead (MB) for provenance collection with the three configurations of Sect. 5.1

Run-time overhead. Provenance capture in UML2PROV involves two time-consuming tasks that may lead to run-time overhead: (1) the execution of database operations for storing provenance data, as they imply the transmission of data and the management of database connections; and (2) the execution of the expansion algorithm for generating the final PROV documents from the PROV templates and the sets of bindings. Of the three configurations, *BindingsConfiguration* yields the highest overhead, 6.21%, as it stores the bindings one by one, each requiring a database operation. Conversely, the other two configurations require one database operation for each traced operation execution: *SetBindingsConfiguration* to store the set of bindings it collects in memory, and *ProvenanceConfiguration* to store the expanded PROV document. This reduces their run-time overheads down to 1.26% and 1.53%. What makes *ProvenanceConfiguration* to yield a higher overhead is the execution of the expansion algorithm to obtain the PROV documents (Fig. 6).

Storage overhead. *SetBindingsConfiguration* yields the most compact storage at MongoDB (2.3MB versus 3.5MB and 3.6MB corresponding to the other configurations). Note that, with the *BindingsConfiguration*, each collected binding is stored with additional data that will allow the generation of the set of bindings to which that binding belongs and, subsequently, the expansion of the associated templates (e.g., operation execution ID, its associated templates, and so on). Thus, a lot of recurrent information is stored. The *SetBindingsConfiguration* reduces the storage needs, as the additional data for expanding the templates is stored only once. In addition, and unlike *ProvenanceConfiguration*, *SetBindingsConfiguration* leverages the PROV-Template approach [28] by storing, instead of final PROV documents, sets of bindings that contain no topological information. Thus, *SetBindingsConfiguration* results in an even more compact storage.

Provenance consumption effort. *ProvenanceConfiguration* constitutes the most appropriate configuration as far as consumer effort to exploit provenance data is concerned. It directly stores the PROV documents, unlike *SetBindingsConfiguration* that stores sets of bindings to expand the templates before consumption; or, even worse, *BindingsConfiguration*, which makes it necessary a prior step to group bindings in sets to feed the template expansion algorithm. In conclusion: the more effort devoted to implementing the configuration, the less effort required for provenance consumption.

7 Discussion

Our approach advocates for an automatic and non-intrusive instrumentation of applications to make them provenance-aware. It allows the provenance capture to be decoupled from the management of the corresponding provenance data, regarding aspects such as format and data storage. Using different implementations of the *BGMEventListener* (*configurations*), bindings can be stored in different storage systems, serialized in a more or less verbose format (e.g., CSV, JSON, XML), individually or as sets of bindings, or even as expanded PROV documents (the *configurations* in Sect. 5.1 provide an example of these alternatives). Additionally, these *configurations* are reusable among all applications sharing the same bindings management policy. Our different *configurations*, together with its evaluation, provide developers with some guide to implement their own configurations.

Note that, as shown in the previous section, the chosen configuration affects not only run-time and storage overhead, but also the effort provenance consumers have to make to exploit the provenance information. Our evaluation has shown that the *SetBindingsConfiguration* has the lowest performance penalty. This strategy, based on a bulk submission of bindings, is aligned with other proposals [7,16] that lead to savings in the overhead of establishing extra database connections. This *configuration* also follows the more compact storage approach, although, it is not the most convenient option from the final provenance consumer point of view (the PROV document is not directly available). Concretely, *SetBindingsConfiguration* is oriented to those consumers that know exactly what provenance they want to retrieve. For example, the provenance from a specific operation execution that led to a set of bindings. Conversely, *ProvenanceConfiguration* is oriented to those consumers that want to explore the generated provenance as a whole; for example, executing SPARQL queries over all the provenance, and navigating the result.

Although our *BGM* implementation proposal is Java-based, the generic event-based structure on which it is built (Fig. 3) could be implemented in other programming languages. The unique requirement is to have an AOP implementation compatible with the programming language of the target application (so that the *BGMEventInstrumenter* can be implemented). This should not be considered an obstacle given the large number programming languages that implement AOP (Python, C, Ada, JavaScript, C#, and so on) [24].

Both UML2PROV provenance design and provenance capture approaches are closely related to PROV standard [17], but some of the ideas underlying the *BGM* structure proposed here may be reused, even without using UML as source. More specifically, a developer may take inspiration from *BGM* requirements *R1* to *R4*, and adapt them for defining a provenance capture artefact without resorting to PROV templates.

The UML2PROV provenance capture presented here is based on UML, so it is worth reflecting on whether the use of UML in industry justifies this choice. Empirical evidence shown by surveys investigating the adoption of UML in the software development community [19,40], together with the fact that the types of UML diagram used by UML2PROV are among the most frequently used [19], support our decision for using UML. Second, UML2PROV requires that all elements on which

provenance is to be captured are present in the UML design. Works as PrIME [26] remark that to address some provenance requirements, it is needed to adapt the application to surface some data. Nevertheless, UML2PROV advocates not changing the application, so users must avoid provenance requirements that require the adaptation of the application. Third, UML2PROV works on two main assumptions: that the UML design includes the level of detail required to capture the desired provenance, and that the implementation of the application conforms to such UML design. UML diagrams are used in industry with different purposes, such as for analysis and understanding, communication, or automatic code generation. Depending on the intended use, the level of detail of the design varies. The completeness of UML models has a direct impact on downstream development quality and productivity. More specifically, studies have shown that, when it is used for communication purposes, a lower quality of UML models led to more variety in the interpretation of the models [6]. Since one of the key purposes of UML models is to enable communication between team members [6,31], this is already a serious indication that consistency and completeness in modeling should be part of quality assurance practices. Similarly, studies shown that a lower quality of UML models correlates with a lower quality of the final source code when using UML designs as a blueprint for the implementation or for automatic code generation [6]. The second assumption is related to the degree to which applications adhere to the design specified by the UML diagrams. However, although it is not a good practice, applications that do not strictly follow the design specified by the UML diagrams are not unusual [15]. In case the underlying assumptions are not satisfied, users could leverage reverse-engineering to obtain the UML design that is faithful to the source code [37].

8 Related work

In the scope of Software Engineering, several approaches stand out for enriching the software development life-cycle of systems with provenance information in two different ways. On the one hand, approaches such as [12,41] address the development process as a whole, providing provenance information about each phase (planning, design, coding, testing, etc.). On the other hand, approaches such as [10,20] are only focused on a concrete phase in the development life-cycle: testing. UML2PROV is similar in spirit to these last works, but focusing on the design phase instead of the testing phase.

The conceptual guiding principles of the bindings generation module *BGM* are based on the taxonomy of characteristics of provenance systems presented in [32], concretely in the *Data capture* dimension defined in it. Herein, we have selected the *Level* and *Technique* categories as the two main facets of interest (we do not discuss about *Tracing* or *Mechanism* aspects since our proposal is agnostic about lazy or eager approaches, and focuses on *internal* structures).

As for the *Level* category, provenance may be collected at different points in the application software stack. The most common proposals, WfMS, follow a *workflow-level* strategy, capturing provenance while the workflow is being enacted (Kepler provenance system [1] and COMAD [4] adopt this strategy). Other works,

such as PASS [29] or SPADE [14], follow an *OS-level* strategy where provenance data is collected at the system API level, recording a low level of metadata for all executions. Despite being two low intrusive strategies, the former have a strong dependency on the WfMS, and the latter usually provides too much irrelevant information. In contrast, UML2PROV does not depend on WfMS. It drives the capture of provenance from the UML design of a system, so the generated provenance is aligned with the level of detail of the UML. UML2PROV, as systems such as noWorkflow [30], adopts a *process-level* approach, capturing provenance at the level of operations. This fact overcomes the drawbacks of the previous approaches but requires adapting pre-existing process activities to incorporate provenance capture functionalities [11], which requires modifying existing applications or scripts. This adaptation may be done from scratch [11], or, as UML2PROV proposes, automatically from the application design.

At this respect, and focusing on the *Technique* used for provenance capture, systems such as ZOOM [9] use an inversion approach that, although provides a more compact representation of the provenance [44], provide sparse information, limited to the derivation history of the data. In contrast, UML2PROV, similar to systems such as yesWorkflow [33], use annotations, giving more flexibility in the richness of provenance metadata, while avoiding computing provenance “just-in-time” like in the inversion method. Annotations can be provided both manually by users, or automatically by applications. UML2PROV relies on AOP to automatically annotate the existing data when generating bindings. To the best of our knowledge, CAPS [5] and UML2PROV are the only proposals that leverages AOP for weaving provenance collection concerns into Java applications. In contrast to UML2PROV, CAPS is restricted to a specific programming language by analysing the structure of the source code. UML2PROV is a generic solution based on the application design.

9 Conclusions and future work

In the context of UML2PROV, this paper describes our proposal to provide applications with provenance capture capabilities. On the one hand, the guiding principles of the bindings generation module *BGM*, and the requirements it must meet to respect such principles, provide a means to guarantee: 1) an automatic (*R1*) and non-intrusive (*R2* and *R3*) instrumentation of the application; 2) which is flexible enough to allow developers to choose the most suitable provenance computation strategy (*R4*); 3) all while ensuring that the generated bindings are compatible with the provenance design also generated by UML2PROV (*R5* and *R6*).

On the other hand, our generic event-based design of *BGM* relying on the Aspect-Oriented Programming paradigm, allowing for the generated code to be automatically weaved into an application. Additionally, our three different *BGM configurations* implemented following the above design may be considered as guides for developers to implement their own *BGMs*. The impact of the three *configurations* on the instrumented application have been analysed considering aspects such as computational and storage overhead, and provenance consumption effort. Configuration *SetBindingsConfiguration* was found to be the most efficient from a performance point of view, although not the best for the final provenance data consumption. Although our

approach has been established to be compatible with PROV standard, some of the ideas supporting the proposed *BGM* structure may be reused, even without using UML as source, or PROV as target. The implementation of UML2PROV in other languages (such as Python), or providing UML2PROV as a service, are lines of future work. Additionally, as commented before, the conceptual guiding principles of the BGM are based on the taxonomy of characteristics of provenance systems presented in [32]. Thus, an interesting line of further work is to compare UML2PROV to other proposals through such a taxonomy.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Altintas I, Barney O, Jaeger-Frank E (2006) Provenance collection support in the kepler scientific workflow system. In: Proceedings of the international provenance and annotation workshop (IPAW'06), pp 118–132
2. Apache SPARK: 2021 Available at <http://spark.apache.org/>. Last visited on September, 2021
3. Baxter ID, Mehlich M (2000) Reverse engineering is reverse forward engineering. *Sci Comput Programm* 36(2):131–147
4. Bowers S, McPhillips TM, Ludäscher B (2008) Provenance in collection-oriented scientific workflows. *Concurr Comput* 20(5):519–529
5. Brauer PC, Fittkau F, Hasselbring W (2014) The aspect-oriented architecture of the caps framework for capturing, analyzing and archiving provenance data. In: Proceedings of the international provenance and annotation workshop (IPAW'14), pp 223–225
6. Chaudron MRV, Heijstek W, Nugroho A (2012) How effective is UML modeling?—an empirical perspective on costs and benefits. *Softw Syst Model* 11(4):571–580
7. Chen Z, Moreau L (2008) Implementation and evaluation of a protocol for recording process documentation in the presence of failures. In: Proceedings of the 2nd International provenance and annotation workshop, pp 92–105
8. Clifford B, Foster I, Voelckler JS et al (2008) Tracking provenance in a virtual data grid. *Concurr Comput* 20(5):565–575
9. Cohen-Boulakia S, Biton O, Cohen S, Davidson S (2008) Addressing the Provenance Challenge using ZOOM. *Concurr Comput* 20(5):497–506
10. Campos Junior H de S, de Paiva CA, Braga R, Araújo MAP., David JMN, Campos F (2017) Regression tests provenance data in the continuous software engineering context. In: Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing, pp 1–6
11. da Cruz SMS, Campos MLM, Mattoso M (2009) Towards a Taxonomy of Provenance in Scientific Workflow Management Systems. In: Proceedings of the IEEE Congress on Services, Part I, SERVICES I, pp 259–266
12. Dalpra HL, Costa GCB, Sirqueira TFM, Braga RM, Campos F, Werner CML, David JMN (2015) Using ontology and data provenance to improve software processes. In: ONTOBRAS
13. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison Wesley, Boston

14. Gehani A, Tariq D (2012) SPADE: support for provenance auditing in distributed environments. In: Proceedings of the 13th International Middleware Conference, pp 101–120
15. Gorp PV, Stenten H, Mens T, Demeyer S (2003) Towards automating source-consistent UML refactorings. In: Proceedings of the 6th international conference on the unified modeling language, pp 144–159
16. Groth P (2007) The origin of data: Enabling the determination of provenance in multi-institutional scientific systems through the documentation of processes. Ph.D. thesis, University of Southampton
17. Groth P, Moreau (eds) L (2013) PROV-Overview. An Overview of the PROV Family of Documents. W3C Working Group Note NOTE-prov-overview-20130430, World Wide Web Consortium www.w3.org/TR/2013/NOTE-prov-overview-20130430/
18. Heras J, Domínguez C, Mata E, Pascual V, Lozano C, Torres C, Zarazaga M (2015) GeJ – a tool for analyzing DNA fingerprint gel images. *BMC Bioinformatics* **16**(1)
19. Hutchinson JE, Whittle J, Rouncefield M, Kristoffersen S (2011) Empirical assessment of MDE in industry. In: Proceedings of the 33rd international conference on software engineering, pp 471–480
20. Khalilian A, Azgomi MA, Fazlalizadeh Y (2012) An improved method for test case prioritization by incorporating historical test case data. *Sci Comput Programm* **78**(1):93–116
21. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J (1997) Aspect-oriented programming. In: Proceedings of the european conference on object-oriented programming (ECOOP'97), pp 220–242. Berlin, Heidelberg
22. Kiczales G, Mezini M (2005) Aspect-oriented programming and modular reasoning. In: Proceedings of the 27th international conference on software engineering (ICSE'05), pp 49–58. ACM, New York, NY, USA
23. Laddad R (2009) Aspectj in action: enterprise AOP with spring applications. Manning Publications Co, Shelter Island
24. Lilis Y, Savidis A (2019) A survey of metaprogramming languages. *ACM Comput. Surv.* **52**(6)
25. Michaelides D, Huynh TD, Moreau L (2014). PROV-TEMPLATE: A Template System for PROV Documents Available at <https://openprovenance.org/prov-template/>. Last visited on September, 2021
26. Miles S, Groth PT, Munroe S, Moreau L (2011) PrImE: a methodology for developing provenance-aware applications. *ACM Trans Softw Eng Methodol* **20**(3):8:1–8:42
27. MongoDB Inc. Version 4.0.2: (2018). Available at www.mongodb.org/. Last visited on September, 2021
28. Moreau L, Batlajery BV, Huynh TD, Michaelides D, Packer H (2018) A templating system to generate provenance. *IEEE Trans Softw Eng* **44**(2):103–121
29. Muniswamy-Reddy KK, Holland DA, Braun U, Seltzer MI (2006) Provenance-Aware Storage Systems. In: USENIX Annual Technical Conference, General Track, pp. 43–56
30. Murta L, Braganholo V, Chirigati F, Koop D, Freire J (2014) noworkflow: capturing and analyzing provenance of scripts. In: International Provenance and Annotation Workshop, pp. 71–83. Springer
31. OMG: Unified Modeling Language (UML). Version 2.5 (2015). Document formal/15-03-01, March, 2015
32. Pérez B, Sáenz-Adán C, Rubio J (2018) A systematic review of provenance systems. *Knowl Inf Syst* **57**(3):495–543. <https://doi.org/10.1007/s10115-018-1164-3>
33. Pimentel JF, Dey SC, McPhillips TM, Belhajjame K, Koop D, Murta L, Braganholo V, Ludäscher B: Yin & yang: demonstrating complementary provenance from noworkflow & yesworkflow. In: Proceedings of the International Provenance and Annotation Workshop (IPAW'16), pp. 161–165
34. ProvToolbox.: Available at <http://lucmoreau.github.io/ProvToolbox/>. Last visited on September, 2021
35. Reggio G, Leotta M, Ricca F, Clerissi D (2013) What are the used UML diagrams? A preliminary survey. In: EESSMOD@MoDELS, USA, pp. 3–12
36. Sáenz-Adán C, Moreau L, Pérez B, Miles S, García-Izquierdo FJ (2018) Automating provenance capture in software engineering with UML2PROV. In: Proceedings of the International Provenance and Annotation Workshop (IPAW'18), pp 58–70
37. Sáenz-Adán C, Pérez B, García-Izquierdo FJ (2020) Moreau L Integrating Provenance Capture and UML with UML2PROV: Principles and Experience. *IEEE Transactions on Software Engineering* <https://doi.org/10.1109/TSE.2020.2977016>. Early Access
38. Sáenz-Adán C, Pérez B, García-Izquierdo FJ, Moreau L (2020) Supplementary material of UML2PROV. Available at <http://uml2prov.unirioja.es>

39. Sáenz-Adán C, Pérez B, Huynh TD, Moreau L (2018) UML2PROV: automating provenance capture in software engineering. In: Proceedings of the 44th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'18), pp 667–681
40. Scanniello G, Gravino C, Tortora G (2010) Investigating the role of UML in the software modeling and maintenance - A preliminary industrial survey. In: Proceedings of the 12th international conference on enterprise information systems, pp 141–148
41. Schreiber A, von Kurnatowski L, de Boer C (2021) Analyzing software engineering processes with provenance-based knowledge graphs. In: 2021 IEEE Aerospace Conference, pp 1–11
42. Seidl M, Scholz M, Huemer C, Kappel G (2015) UML@Classroom: An Introduction to Object-Oriented Modeling. Springer Publishing Company, Incorporated
43. Selic B (2003) The pragmatics of model-driven development. *IEEE Softw* 20(5):19–25
44. Simmhan YL, Plale B, Gannon D (2005) A Survey of Data Provenance Techniques. Computer Science Department, Indiana University, Bloomington IN (612) Extended version of SIGMOD Record 2005. Available at: www.cs.indiana.edu/pub/techreports/TR618.pdf
45. The AspectJ Project: Available at www.eclipse.org/aspectj/. Last visited on September, 2021
46. UML2PROV User Guide: Available at <https://github.com/uml2prov/uml2prov>. Last visited on September, 2021
47. XSL Transformations (XSLT) Version 3.0: W3C Candidate Recommendation 7 February 2017. Available at www.w3.org/TR/xslt-30/. Last visited on September, 2021
48. XTend: Available at www.eclipse.org/xtend/. Last visited on September, 2021

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Carlos Sáenz-Adán¹ · Francisco J. García-Izquierdo¹ · Beatriz Pérez¹  · Trung Dong Huynh² · Luc Moreau²

✉ Beatriz Pérez
beatriz.perez@unirioja.es

Carlos Sáenz-Adán
carlos.saenz@unirioja.es

Francisco J. García-Izquierdo
francisco.garcia@unirioja.es

Trung Dong Huynh
dong.huynh@kcl.ac.uk

Luc Moreau
luc.moreau@kcl.ac.uk

¹ Department of Mathematics and Computer Science, University of La Rioja, La Rioja, Spain

² Department of Informatics, King's College London, London, UK