Automating Provenance Capture in Software Engineering with UML2PROV

Carlos Sáenz-Adán[1]*, Luc Moreau[2], Beatriz Pérez[1], Simon Miles[2], and Francisco J. García-Izquierdo[1]

[1] Dept. of Mathematics and Computer Science, Univ. of La Rioja, La Rioja, Spain,
{carlos.saenz,beatriz.perez,francisco.garcia}@unirioja.es
[2] Dept. of Informatics, King's College London, London, UK,
{luc.moreau,simon.miles}@kcl.ac.uk

Abstract. UML2PROV is an approach to address the gap between application design, through UML diagrams, and provenance design, using PROV-Template. Its original design (i) provides a mapping strategy from UML behavioural diagrams to templates, (ii) defines a code generation technique based on Proxy pattern to deploy suitable artefacts for provenance generation in an application, (iii) is implemented in Java, using XSLT as a first attempt to implement our mapping patterns. In this paper, we complement and improve this original design in three different ways, providing a more complete and accurate solution for provenance generation. First, UML2PROV now supports UML structural diagrams (Class Diagrams), defining a mapping strategy from such diagrams to templates. Second, the UML2PROV prototype is improved by using a Model Driven Development-based approach which not only implements the overall mapping patterns, but also provides a fully automatic way to generate the artefacts for provenance collection, based on Aspect Oriented Programming as a more expressive and compact technique for capturing provenance than the Proxy pattern. Finally, there is an analysis of the potential benefits of our overall approach.

Keywords: Provenance data modeling and capture · PROV-Template · UML

1 Introduction

The diversity of provenance models used by existing software products (such as PASS [1], PERM [2], or Taverna [3]) to capture provenance has motivated the creation of PROV [4], an extensible provenance model created to exchange and integrate provenance captured among different provenance models. By giving support to PROV, these tools facilitate the software engineer's task of creating, storing, reading and exchanging provenance; however, they do not help decide which provenance data should be included, nor how software should be designed to allow its capture. In this context, the ability to consider the intended use of provenance during software development has become crucial, especially in the design phase, to support software designers in making provenance-aware systems.

Several design methodologies have been proposed to shorten the development time of software products. In particular, the Unified Modeling Language (UML) [5] has become a standard notation for OO software design. However, it does not offer support for provenance. In fact, our experience in developing software applications enhanced with support for provenance is that including provenance within the design phase can entail significant changes to an application design [6]. Against this background, PROV-Template [7] has been proposed as a declarative approach that enables software engineers to develop programs that generate provenance compatible with the PROV standard. Provenance *templates* are provenance documents expressed in a PROV-compatible format and containing placeholders (referred as *variables*), for values. PROV-Template includes an *expansion algorithm* by means of which, given a template and a set of *bindings* (associating variables to values), replaces the placeholders by the concrete values, generating a provenance record in one of the standardized PROV representations. Although this approach reduces the development and maintenance effort, it still requires designers to have provenance knowledge.

To overcome these challenges, we introduced *UML2PROV* [8], an approach to address the gap between application design, through UML behavioural diagrams, and provenance design, using PROV-Template. Briefly speaking, we (i) provided a mapping strategy from UML State Machine and Sequence diagrams to *templates*, (ii) defined a code generation technique based on the Proxy pattern to deploy suitable artefacts for provenance generation in an application, and (iii) developed a first prototype of UML2PROV in Java, using XSLT as a first attempt to implement our mapping patterns. In this paper, we complement and improve our previous approach by providing a more complete and accurate solution for provenance generation. First, we mainly give support to UML structural diagrams (UML Class Diagrams), by establishing a mapping strategy from such type of diagrams to templates. Our approach for capturing provenance data included on a system's class diagram provides a mean of storing lower level factors from objects' internal structure, factors not given by the previously considered behavioural diagrams. Overall, we provide an effective mechanism that integrates provenance data regarding both structural and behavioural aspects of a system, allowing for more realistic software designs to be supported. Second, we improve our first prototype by using a Model Driven Development (MDD)-based approach which implements the overall mapping patterns, and provides a fully automatic way to generate the artefacts for provenance collection based on Aspect Oriented Programming (AOP). Finally, we analyse the potential benefits of our overall approach in terms of time it takes to generate the templates, run-time overhead given by bindings collection, development and maintenance.

This paper is organized as follows: Section 2 gives an overview of UML2PROV. Section 3 describes our overall approach to translate UML Class diagrams to templates. A detailed description of the new implementation we propose for our first UML2PROV prototype is described in Section 4. We analyse our overall approach in Section 5, while Section 6 discusses related work. Finally, conclusions and further work are set out in Section 7.

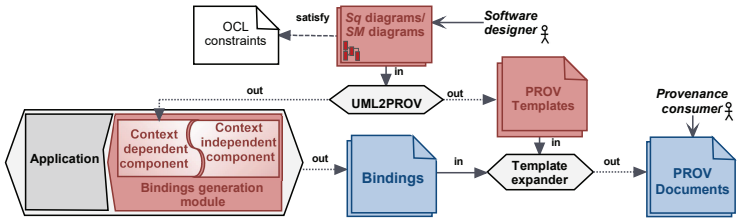


Fig. 1. The UML2PROV approach. The red and blue colours are used to refer to *design time* and *runtime* documents of the approach, respectively.

2 Overview: The UML2PROV approach

To lay the foundation for a more in-depth understanding of the following sections, we provide an overview of the UML2PROV architecture presented in [8]. We illustrate our explanations using Figure 1 which identifies the key facets of our proposal together with the different stakeholders involved on the process. The overall process consists of both *design time* (red) and *runtime* (blue) elements.

Design time facets. They correspond to the *UML diagrams* modelling the system, the associated *PROV templates* generated from those diagrams, and the *bindings generation module*. In particular, this module is composed by: a *context-independent component*, which contains the bindings' generation code that is common to all applications, and a *context-dependent component*, which is generated from the system's UML diagrams and includes the bindings' generation code specific to the concrete application. The starting point of the overall process corresponds to the UML system design, created by the *software designers* as stated by the concrete domain's requirements. Among the two major categories of UML diagrams (*structural* and *behavioural*) [5], in [8] we focused on these latter ones given the strong relation that provenance bears with all behavioural data taking part in producing a final item. Having defined the UML diagrams, and before applying our UML2PROV proposal, the diagrams are checked against a set of OCL [9] constraints we have defined to ensure that they are consistent with each other (see [10] for details about these constraints). Then, the UML2PROV proposal takes as input the UML diagrams and automatically generates: (1) the PROV *templates* with the design of the provenance to be generated, relying on the information extracted from such diagrams, and (2) the *context-dependent component* aimed at capturing provenance according to the PROV templates.

Runtime execution facets. They consist of the values logged by the application, in the form of *bindings*, and the *PROV documents*. As far as the process is concerned, taking as source both the *templates* and the *bindings* previously created, the *provenance consumer* uses the provenance *template expander* included in the PROV Template proposal to generate the final *PROV documents* (see Figure 1).

3 From Class Diagrams to Templates

Our class diagrams to templates mapping takes *operations* as cornerstone elements. Translating data implicit on *operations* provides us with a complete background including not only the internal structure of the object before and

Table 1. Extension of the taxonomy of methods' stereotypes given in [11].

Stereotype category	Stereotype name	Description
Structural *Accessor*	get	Returns a data member.
	*get-collection	Returns an element from a data member collection.
	predicate	Returns a Boolean value which is not a data member.
	property	Returns information about data members.
	void-accessor	Returns information through a parameter.
Structural *Mutator*	set	Sets a data member.
	*set-add-collection	Adds an element within a data member collection.
	*set-remove-collection	Removes an element within a data member collection.
	command	Perform a complex change to the object's state.
	non-void-command	
Creational	constructor/destructor	Creates/Destroys objects.
Collaborational	*collaborator*	*Works with objects (parameter, local variable and return object).*
	controller	*Changes an external object's state.*
Degenerate	*incidental*	*Does not read/change the object's state.*
	empty	*Has no statements.*

after the execution (values of the *attributes*), but also information showing the internal changes (e.g. setting a new attribute, adding/removing an element in a collection). This represents a significant new capability since we were not able to extract these lower-level aspects from Sequence/State Machine Diagrams in [8].

Aimed at defining concrete operation transformation patterns, their different nature must be taken into account if we want to provide meaningful provenance which explains the nuances of each type of operation's execution. For instance, the key factors involved in the execution of an operation such as *getName* (which would return information about a data member) are different from the ones related to a *setName* operation (which would set a data member). Thus, the provenance data to be generated in both cases would be expected to be different. For this reason, we have first established a taxonomy of UML Class Diagrams' operations (Subsection 3.1) to identify the different types of operations. Second, based on such a classification, we have defined different transformation mappings (Subsection 3.2) depending on each type of operation.

3.1 A taxonomy of operations stereotypes

More than a nuance in terminology, the distinction between *operation* and *method* is important to lay the foundations of this section. *Operations* are characterized by their declaration, including name or parameters [5]. *Methods* are made up of the declaration (given by the *operation*) as well as the behaviour. From now on, we use the term *operation* and *method* interchangeably, always referring to the behaviour. In particular, we refer to the low-level behaviour related to the internal structure of the object's class to which the operation belongs.

In order to establish a taxonomy of operations that allows us to identify the different transformation patterns, we have undertaken a literature search looking for different categorizations of operations. Among the different works, the presented by Dragan et al. [11] stands out for being one of the most complete. Such a taxonomy is showed in Table 1 where, as we explain later, we have also included additional stereotypes needed in our proposal (marked with an asterisk). Their taxonomy establishes five categories of methods by defining stereotypes for their

categorization, three of which have been included in our proposal (*Structural Accessor*, *Structural Mutator* and *Creational*). An explanation of these categories together with their specific transformation will be presented in Subsection 3.2.

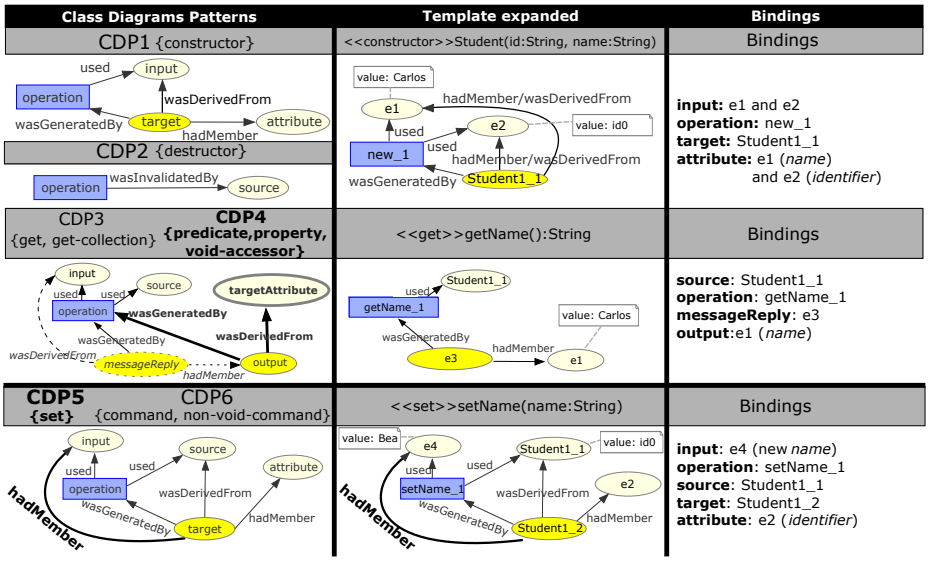
Whilst this taxonomy covers a wide range of behaviours, it lacks specific stereotypes for methods that manage collections of data members (e.g. search, addition or removal). Aimed at identifying this kind of methods on class diagrams to generate concrete provenance data, we have enriched the previous taxonomy with the additional stereotypes *get-collection*, *set-add-collection* and *set-remove-collection* (marked with an asterisk in Table 1). On the other hand, some stereotypes denote behaviours that cannot be faced without checking the source code (*empty*), or behaviours already provided by Sequence/State Machine Diagrams. In particular, Sequence Diagrams allow us to know if an *operation* works with objects (*collaborator*), and State Machine Diagrams provide us with information regarding external (*controller*) and internal (*incidental*) state changes. Thus, we have not considered *Collaborational* and *Degenerate* categories.

3.2 Class Diagrams to templates transformation patterns

Our transformations are focused on *operations* customized by stereotypes so that, depending on the stereotype applied to an operation, they translate such an *operation* into the corresponding PROV template representing the *object*'s state. We define the *state* of an object as its internal structure, consisting of the object's properties (attributes and relationships) together with the values of those properties. The set of mappings comprises 8 transformation patterns identified *CDP1-8*, referred to as *C*lass *D*iagram *P*attern. Table 2 shows patterns *CDP1-6*, while patterns referring to collections, *CDP7* (*set-remove-collection*) and *CDP8* (*set-add-collection*), are presented in [10] due to space reasons. Table 2 has three columns: the first one shows each pattern together with the corresponding provenance template; the second and third columns depict the provenance document generated after expansion, and the provenance information collected during the operation's execution (*bindings*), respectively. The information shown in these two last columns corresponds to the case study we use in [8] referring to a system that manages the enrolment and attendance of students to seminars of a University course. We have used the *Student*'s class constructor and the self-explained *getName* and *setName* operations to exemplify *CDP1*, *CDP3*, and *CDP5*. In Table 2: (1) the stereotypes (i.e. the types of operations) tackled by each pattern are showed between curly brackets, and (2) the `prov:Entities` created as a result of the operation's execution are in dark yellow, while `prov:Entities` assumed to exist before the operation's invocation are in light yellow.

All patterns share common transformations. First, all the *operations* are translated into a `prov:Activity` identified by `var:operation`. Second, when applicable, the object's initial state is given by a `prov:Entity` identified by `var:source`. Third, each *input operation*'s *argument* is mapped to a `prov:Entity` named `var:input`. Finally, when applicable, two `prov:used` relationships link `var:operation` with `var:source` and `var:input` to represent that the operation "uses" an initial state of the object (`var:source`), and a set of *input arguments* (`var:input`).

Table 2. Patterns *CDP1-CDP6* including the proposed provenance templates, together with the expanded template and the values of the variables (*bindings*).



Creational. The operations included in this category, which are *constructor* and *destructor*, are addressed by *CDP1* and *CDP2*, respectively. Following *CDP1*, a *constructor* operation (identified by `var:operation`) creates a new object using (or not) *input arguments* (identified by `var:input`). Such a new object is translated into a `prov:Entity` identified by `var:target`, together with its set of data members, represented by the `prov:Entity` named `var:attribute`. Additionally, to show that the new object (`var:target`) has been generated using the *input arguments* (`var:input`), we define a `prov:wasDerivedFrom` relationship between them. In turn, `var:target` is related to `var:operation` through `prov:wasGeneratedBy` to show that the new object (`var:target`) has been generated by the *constructor* operation (`var:operation`). Following *CDP2*, a *destructor* operation (identified by `var:operation`) destroys an object (identified by `var:source`), fact represented by the relationship `prov:wasInvalidatedBy` between `var:source` and `var:operation`.

Structural Accessors. The operations that do not change the state of an object (internal structure) are translated by *CDP3* and *CDP4* (see Table 2). In particular, these operations are used for retrieving information, represented by the `prov:Entity` identified by `var:output`. While the operations *get* and *get-collection* tackled by *CDP3* return the data member directly, the operations *predicate*, *property* and *void-accessor* addressed by *CDP4* generate new information based on the data member(s). To represent the return of information (not the generation of information) in *CDP3*, we use a `prov:Entity` identified by `var:messageReply`, which is created by the operation (`var:operation`), and encapsulates the retrieved information (`var:output`). These elements, highlighted in italic and with dashed lines in *CDP3* of Table 2, are related to `var:operation` by the relationship `prov:wasGeneratedBy`. The relationship `prov:hadMember` is also used to link them (`var:messageReply` as source and `var:output` as target). On the contrary, the information retrieved by the operations tackled in *CDP4* is generated by such operations, involving a data member which is rep-

resented by an `prov:Entity` identified by `var:targetAttribute`. These additional aspects, highlighted in bold in *CDP4* of Table 2, are represented by the relationships: `prov:wasGeneratedBy`, between `var:operation` and `var:output`, and `prov:wasDerivedFrom`, between `var:output` and `var:targetAttribute`.

Structural Mutators. For operations that change the state of an object, we distinguish (i) those that set a specific data member –*set* methods– together with those whose behaviour performs a complex change –*command* and *non-void-command* methods– (tackled by *CDP5* and *CDP6*); from (ii) those that manage data member collections –*set-remove-collection* and *set-add-collection* methods– (tackled by *CDP7* and *CDP8*, presented in [10]).

In addition to the set of transformations shared by all patterns as explained before, *CDP5* and *CDP6* also have a set of common transformations. The operations tackled by these patterns change the object's state (internal structure) through the modification of some of its data member(s). Hence, the new state of the object is represented by a `prov:Entity` identified by `var:target`, while each object's data member is translated using a `prov:Entity` identified by `var:attribute`. To represent that such attributes (`var:attribute`) belong to the new state of the object (`var:target`), we use the relationship `prov:hadMember` between them. In turn, `var:target` is also related to the operation (`var:operation`) through `prov:wasGeneratedBy`, representing that the new object's state has been generated by such an operation. Additionally, `var:target` is linked, by means of `prov:wasDerivedFrom`, with a `prov:Entity` identified by `var:source`, which represents the previous object's state. In addition to these elements, the *CDP5* pattern, which tackles *set* operations, includes the `prov:hadMember` relationship between `var:target` and `var:input` to show that the input parameter is set as a new data member (see the highlighted `prov:hadMember` relationship in Table 2).

4 Implementation

Here, we discuss our proposal for enhancing our first UML2PROV approach [8], which is mainly characterized by: (1) the implementation of our transformation patterns from UML Diagrams to *provenance templates files*, and (2) the generation of artefacts for provenance collection. Although both aspects were reasonably tackled in our prototype, they were subject to improvement. Next, we explain why and how we have enhanced our prototype leaning on Figure 2.

4.1 Implementation of the mapping patterns

Given the wide range of contexts of application, a manual translation of the UML Diagrams of a system to *templates* constitutes a time-consuming, error-prone and not cost-effective task. To overcome these challenges, we originally developed an XSLT-based prototype as first attempt to implement our mapping patterns [8]. Although being a powerful solution, the usage of XSLT for implementing mapping rules is no longer the best option, given the availability of mapping and transformation languages created by the MDD community which have better properties in terms of maintenance, reusability, and support to software development processes [12]. For this reason, in this paper, we propose to use an MDD approach [13], focusing on models rather than computer programs, so that the

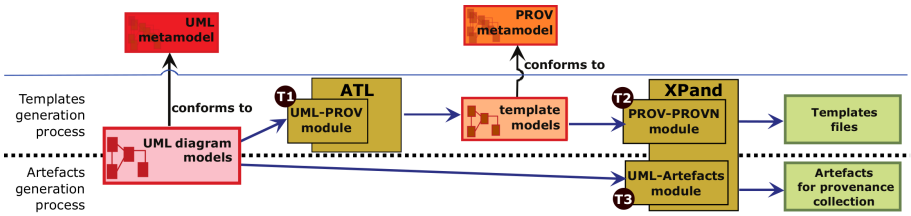


Fig. 2. MDD-based implementation proposal.

templates files are automatically generated using a refinement process from the UML Diagrams (see the top of Figure 2). Our solution for template's generation follows an MDD-based tool chain, comprising transformations *T1* and *T2*.

First, *T1* performs a model–to–model (M2M) transformation, taking as source the *UML diagram models* of the system (which conform to the UML metamodel) and generating the corresponding provenance *template models* (which conform to the PROV metamodel (PROV-DM [14]). Among the different MDD-based tools in the literature, we have implemented this transformation by means of the ATL Eclipse plug–in [15]. We have defined an ATL module named *UML-PROV* which automatically translates each diagram model (sequence, state machine and class diagram) into the corresponding provenance *template models*. Second, *T2* carries out a model–to–text (M2T) transformation, taking the provenance *template models* resulted previously, and generating the final *templates files* serialized in PROV-N notation. *T2* has been implemented in the XPand tool [16] by means of a one-to-one transformation module named *PROV-PROVN*. This module takes the previously generated models and returns the *template files* in PROV-N.

By using the transformations defined in these two MDD–based tools, we are able to automatically generate, starting from the UML Diagrams of a system, the corresponding provenance *template files*. It is worth noting that the ATL and Xpand transformations can be applied to UML Diagrams (Sequence, State Machine, and Class Diagrams) in any context.

4.2 Generation of artefacts

Having generated the template files, we need suitable code artefacts to create the bindings containing the pairs template variables–values. Programming the creation of bindings typically involves manually adding many lines of code repeated along the whole application's base code (obtaining the well-known *scattering* code), with its consequent loss of time on development and maintenance. Additionally, performing a manual creation of bindings requires the programmer to have a deep understanding of the design of both the application and the provenance to be generated. In [8] we faced this issue by following a Proxy pattern [17] approach as a first attempt to generate bindings with a minor programming intervention. Whilst the Proxy pattern approach facilitates such a generation by wrapping each object to extend its behaviour with extra lines of code, this solution still requires to manually modify the application's source code. In order to provide a fully automatic way for bindings generation, we instead propose to use the Aspect Oriented Programming (AOP) [18] paradigm. AOP aims at improving the modularity of software systems, by capturing inherently scattered functionality, often called *cross-cutting concerns* (thus, data provenance can be considered as a cross-cutting concern). Our solution exploits

AOP to seamlessly integrate cross-cutting concerns into existing software applications without interference with the original system. The core of AOP is the *aspect*, which constitutes a separate module that describes the new functionality that should be executed at precise locations as the original program runs.

Taking this into account, we have followed an MDD-based approach for generating, starting from the source UML Diagrams, a context-dependent *aspect* in AspectJ (an AOP extension created for Java) together with other auxiliary components in Java, constituting what we have called *artefacts for provenance collection*. This new transformation *T3* has been implemented as an Xpand module named *UML-Artefacts* (see the bottom of Figure 2) which, starting from the *UML diagram models* which represent the system design, directly generates the *artefacts for provenance collection* (Section 4 of online appendix [10] contains an example). The generated AOP *aspect* implements the behaviour that is to be executed to generate the bindings at specific points in the concrete application code. We note that, although the new functionality to be executed for bindings generation is common to all applications, such points are specific to the concrete application. With our proposal, the programmer just needs to include the resulted *artefacts* into the application, so that it will become automatically provenance-aware without requiring any other intervention.

5 Analysis and Discussion

We first analyses the strengths and weaknesses of UML2PROV taking into account (i) the automatically generation of *templates*, focusing on the time it takes to generate the *templates* and how much elements are included on the *templates*; and (ii) the collection of bindings during the execution of the application, discussing its run–time overhead. Finally, we highlight development and maintenance benefits of using UML2PROV.

As for the generation of the *templates*, since it is carried out during the design phase, it does not interfere in any way with the overall application performance. Regarding the amount of generated *templates*' elements, each *template* defines a fixed number of elements; thus, there is a linear association between the number of elements and the number of *templates* generated. Thus, in case of a huge amount of input/output arguments, and attributes, the number of elements after the expansion process grows proportionally to the length of these elements.

Another issue that may concern the users of UML2PROV is the run–time overhead. As a way of example, in Table 3 we provide a benchmark of seven execution experiments (identified from 1 to 7) using the *Stack* case study presented in [10]. In particular, it depicts the execution times with and without UML2PROV (see columns 2 and 3, respectively). We note that all experiments use retrieved information from a database. Based on the benchmarks showed in this table, as it would be expected, recording the provenance using our approach increases the original processing time by ∼14.5%. We can consider worthwhile this increment, taking into account that the approach herein captures provenance from all the elements modelled in the UML Diagrams with a high level of detail. In this line, an interesting aspect of future work would be to provide the

Table 3. Results obtained from seven experiments using the *Stack* case study [10].

ID	Without UML2PROV (ms)	With UML2PROV (ms)	Increment (%)	Number variables	Description
1	48	56	16,67	2260	25 push operations from Stack
2	84	97	15,48	4510	50 push operations from Stack
3	161	182	13,04	9010	100 push operations from Stack
4	45	53	17,78	3160	25 pop operations from Stack
5	82	93	13,41	4710	50 pop operations from Stack
6	153	175	14,38	7810	100 pop operations from Stack
7	300	332	10,67	19952	Turn down a stack with size 100

Legend:
-ID: Experiment identifier.
-Without UML2PROV: Average time taken by 50 executions without UML2PROV.
-With UML2PROV: Average time taken by 50 executions generating bindings with UML2PROV.
-Increment: Percentage of time increased by applying UML2PROV.
-Number variables: Total number of variables captured.
-Description: Brief explanation of the experiment

* The experiments were run on a personal computer, Intel(R) Core(TM) i7 CPU, 2.8 GHz, running Windows 10 Enterprise. This computer runs Oracle JDK 1.8 together with MySQL 5.5

UML designer with a mechanism to specify both the (i) the specific elements in the UML Diagrams to be traced, and (ii) the level of detail of the captured provenance for each selected element.

As said previously, UML2PROV makes the development and maintenance of provenance-aware systems a simple task, by automatically generating provenance templates and artefacts for provenance collection. In particular, the automation of template's generation entails direct benefits in terms of compatibility between the design of the application and the design of the provenance to be generated. Every time the design of the application changes, provenance design is updated automatically. As a consequence, since the artefacts for provenance collection –which create bindings– are also automatically generated from the design of the application (as well as the templates), there are no problems with regard to incompatibility between templates and bindings. In fact, since these artefacts contain all the instructions to generate the *bindings*, programmers do not need to traverse the overall application's code, and include suitable instructions. Specifically, for each variable in a provenance template, a method call is needed to assign a value to it; thus, a programmer would need to write one line of code per each variable in a template. Although Table 2 shows that the templates are relatively small (e.g. *CDP4* –which is the biggest– comprises 6 nodes), we note that an application may encompass thousands of methods. Thus, our approach makes the collection of bindings a straightforward task.

6 Related Work

There is a huge amount of scientific literature about provenance, which has been collected and analysed by several surveys among different fields (see a complete review in [19]). Additionally, there are several works which particularly undertake the development of provenance-aware systems. For example, PASS [20], which is a storage systems supporting the collection and maintenance of provenance; PERM [2], which is a provenance-aware database middleware; or Taverna [3], Vistrails [21] and Kepler [22] which include provenance into workflow systems. Whilst these applications show efficacy in their research areas, they manually weave provenance generation instructions into programs, making the code maintenance a cumbersome task. In contrast to this strategy, some mechanisms for automatically provenance capture have been proposed in the literature. Among the systems in which the developers do not need to manually manipulate the code, Tariq et al. [23], noWorkflow [24] and Brauer et al. [25] stand out. Tariq et al. [23] automatically weave provenance capture instructions within the application before and after each function call during the compilation process. The noWorkflow tool [24] is registered as a listener in the Python profiling API, so that the profiler notifies when the functions have been activated in the source code. Brauer et al. [25] use AOP aspects for generating provenance. Our ap-

proach is similar in spirit with all these works, since UML2PROV transparently captures provenance in a non-intrusive way. Unlike these approaches which rely on the source code of the application, UML2PROV constitutes a generic solution based on the application's design. It identifies the design of the provenance to be generated (*templates*) and creates the context-dependent artefacts for provenance collection using the application design given by UML Diagrams. This fact unlinks the provenance capture with the specific implementation of the application, providing a generic solution for developing provenance-aware applications.

Finally, we note PrIMe [6] which, although being considered the first provenance-focused methodology, is standalone and is not integrated with existing software engineering methodologies. UML2PROV complements PrIMe, since it integrates the design of provenance by means of PROV-Templates enriched with UML.

7 Conclusions and Future Work

We have defined a comprehensive approach UML2PROV. First, we complete it by giving support to Class Diagrams, establishing a mapping strategy from such diagrams to templates. Second, we improve our first prototype by using an MDD-based approach which not only implements the overall mapping patterns, but also generates the AOP artefacts for provenance collection. Finally, there is an analysis of the potential benefits of our overall approach.

In addition to the future work advanced previously, another line of future work is the application of UML2PROV in a distributed system. We plan to tackle this goal by automatically generating an artefact for provenance collection able to capture provenance not only in a fully-in-memory system (as until now), but also in a system comprising distributed components. Finally, we may use some PROV attributes (e.g. `prov:type`, `prov:role`...) in the templates, in order to specialize concrete elements. With such specializations, we aim to improve the provenance consumption by creating less complex queries with higher accuracy, reducing the noise levels in the retrieved provenance information.

Acknowledgements. This work was partially supported by the spanish MINECO project EDU2016-79838-P, and by the U. of La Rioja (grant FPI-UR-2015).

References

1. Holland, D., Braun, U., Maclean, D., Muniswamy-Reddy, K.K., Seltzer, M.I.: Choosing a data model and query language for provenance. In: Proceedings of IPAW'08. (2008) 98–115
2. Glavic, B., Alonso, G.: Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In: Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE'09). (2009) 174–185
3. Wolstencroft, K., et al.: The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. Nucleic acids research **41** (2013) 557–561
4. Groth P., Moreau L. (eds.): PROV-Overview. An Overview of the PROV Family of Documents. W3C Working Group Note prov-overview-20130430 (2013) `http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/`.

5. OMG. Unified Modeling Language (UML). Version 2.5: (2015) formal/15-03-01, http://www.omg.org/spec/UML/2.5/. Last visited on March 2018.
6. Miles, S., Groth, P.T., Munroe, S., Moreau, L.: Prime: A methodology for developing provenance-aware applications. ACM Trans. Softw. Eng. Methodol. **20**(3) (2011) 8:1–8:42
7. Moreau, L., Batlajery, B.V., Huynh, T.D., Michaelides, D., Packer, H.: A Templating System to Generate Provenance. IEEE Transactions on Software Engineering (2017) http://eprints.soton.ac.uk/405025/.
8. Sáenz-Adán, C., Pérez, B., Huynh, T.D., Moreau, L.: UML2PROV: automating provenance capture in software engineering. In: Proc. of Sofsem'18. (2018) 667–681
9. OMG: Object Constraint Language, Version 2.4 (2014) formal/2014-02-03 http://www.omg.org/spec/OCL/2.4/PDF.
10. Supplementary material of UML2PROV (2018): https://uml2prov.github.io/.
11. Reverse Engineering Method Stereotypes. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance. (2006)
12. Catalina Martínez Costa, Marcos Menárguez-Tortosa, J.T.F.B.: Clinical data interoperability based on archetype transformation. Journal of Biomedical Informatics **44**(5) (2011) 869–880
13. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software **20**(5) (2003) 19–25
14. Moreau, L., et al.: PROV-DM: The PROV Data Model. W3C Recommendation REC-prov-dm-20130430, World Wide Web Consortium (2013) http://www.w3.org/TR/2013/REC-prov-dm-20130430/.
15. ATL - a model transformation technology, version 3.8: (May 2017) Available at http://www.eclipse.org/atl/. Last visited on March 2018.
16. XPand: Eclipse platform (2018) https://wiki.eclipse.org/Xpand, Last visited on March 2018.
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison Wesley (1995)
18. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proc. of the European Conference on Object-Oriented Programming (ECOOP'97), Berlin, Heidelberg (1997) 220–242
19. Pérez, B., Sáenz-Adán, C., Rubio, J.: A systematic review of provenance systems. Knowl. Inf. Syst. (2018)
20. Glavic, B., Dittrich, K.R.: Data Provenance: A Categorization of Existing Approaches. In: Proceedings of Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'07). (2007) 227–241
21. Silva, C.T., Anderson, E., Santos, E., Freire, J.: Using vistrails and provenance for teaching scientific visualization. Computer Graphics Forum **30**(1) (2011) 75–84
22. Altintas, I., Barney, O., Jaeger-Frank, E. In: Provenance Collection Support in the Kepler Scientific Workflow System. (2006) 118–132
23. Tariq, D., Ali, M., Gehani, A.: Towards automated collection of application-level data provenance. In: Proceedings of TaPP'12. (2012)
24. Pimentel, J.F., Murta, L., Braganholo, V., Freire, J.: noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts. In: Proceedings of VLDB'17. Volume 10. (2017) 1841–1844
25. Brauer, P.C., Fittkau, F., Hasselbring, W.: The aspect-oriented architecture of the caps framework for capturing, analyzing and archiving provenance data. In: Proceedings of IPAW'14, Springer (2014) 223–225