UML2PROV: Automating Provenance Capture in Software Engineering

Carlos Sáenz-Adán[1*], Beatriz Pérez[1], Trung Dong Huynh[2], and Luc Moreau[3]

[1] Department of Mathematics and Computer Science, University of La Rioja, La Rioja, Spain,
{carlos.saenz,beatriz.perez}@unirioja.es
[2] Department of Electronics & Computer Science, University of Southampton, Southampton, UK,
tdh@ecs.soton.ac.uk
[3] Department of Informatics, King's College London, London, UK,
luc.moreau@kcl.ac.uk

Abstract. In this paper we present UML2PROV, an approach addressing the gap between application design, through UML diagrams, and provenance design, using PROV-Template. PROV-Template is a declarative approach that enables software engineers to develop programs that generate provenance following the PROV standard. The main contributions of this paper are: (i) a mapping strategy from UML diagrams (UML State Machine and Sequence diagrams) to templates, (ii) a code generation technique that creates libraries, which can be deployed in an application by creating suitable artefacts for provenance generation, and (iii) a demonstration of the feasibility of UML2PROV implemented with Java, and a preliminary quantitative evaluation that shows benefits regarding aspects such as design, development and provenance capture.

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

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" [1].

This interest in provenance has led to various point solutions developed to capture provenance (such as PASS [2], PERM [3], Taverna [4], Vistrails [5] or Kepler [6]). The need for interoperability between systems has been a driver for the creation of the PROV standard [1,7,8], a conceptual data model for provenance, and its serialization to various Web technologies. Since PROV's aim is the interoperable exchange of provenance information, toolkits supporting PROV [9,10] have been facilitating the software engineer's task of creating, storing, reading

and exchanging provenance; 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 consider the use of provenance, specially during the software engineering design phase, has become critically important to support the software designer in making provenance-enabled systems. PrIMe [11], the *Provenance Incorporation Methodology*, is the first provenance-focused methodology for adapting applications to make them provenance-aware. Although the application of this methodology has demonstrated promising results, PrIMe is standalone, and does not integrate with existing software engineering methodologies, which makes it challenging to use in practice.

In contrast, design techniques have been proposed to shorten the development time of software products, as well as to increase their quality, avoiding developers from expending extra time and efforts during subsequent phases. Among such techniques, the Unified Modelling Language (*UML*) [12] is widely accepted as the de-facto method for designing object-oriented software systems. However, the *UML* design methodology offers no specific support for provenance. Specifically, *UML* does not provide the means to express elements of response to provenance questions, such as the activity that lead to a specific result, or the elements involved in its creation. In fact, our experience in developing software applications augmented with support for provenance is that the inclusion of provenance within the design phase can entail significant changes to an application design [11]. This is a cumbersome task for the designers and programmers alike, since they have to be knowledgeable about provenance, to deal with complex diagrams, and to maintain an application's provenance-specific code base. In short, the gap between software engineering design methodologies and provenance engineering can 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 [13] is a recent development allowing the structure of provenance to be described declaratively: a provenance *template* is a document containing placeholders (referred 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, separating responsibilities between software and provenance designers, it still requires designers with provenance knowledge.

The aim of this paper is to propose UML2PROV, an approach that addresses the gap between application design, through *UML* diagrams, and provenance design, by means of PROV-Template. The contributions of this paper are as follows: (i) a mapping of *UML* diagrams (*UML* State Machine and Sequence diagrams) to templates according to a set of transformation rules, (ii) a code generation technique that creates libraries, that need to be linked with the application to generate provenance, and (iii) a demonstration of the feasibility of UML2PROV by implementing it with Java, whose preliminary quantitative evaluation shows significant benefits of the approach. These benefits, which will appeal to designers in early stages of the development process, are mainly: (1) *design/development*, since we provide a way to include provenance capabilities during the design phase *without* changing the way in which software designers

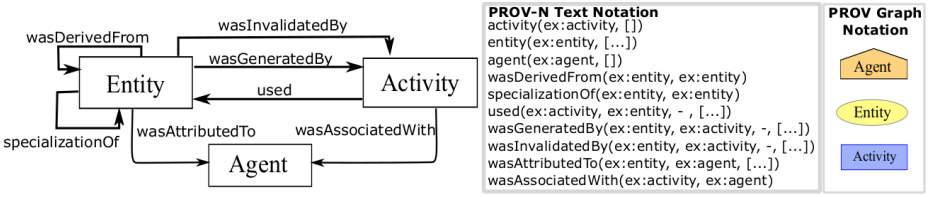

Fig. 1. PROV UML Class Diagram with graphical and textual PROV notation [7,8]

use *UML* (provenance generation is handled automatically from such *UML*), and (2) *capturing provenance*, since the provenance capture is performed automatically thanks to UML2PROV's code generation technique, which provides clear benefits over the more traditional approach of provenance capture.

This paper is organized as follows. We outline the background of this research in Section 2. In Section 3, we give an overview of UML2PROV. Sections 4 and 5 describe our approach, while Section 6 presents a complete implementation of it. A quantitative evaluation is provided in Section 7, while Section 8 discusses related work. Finally, conclusions and further work are set out in Section 9.

2 Background

In this section, we first introduce the PROV standard for provenance and provide an overview of the main insights concerning the use of PROV-Template. Second, we highlight key aspects of the *UML* diagrams used in this work.

2.1 The PROV standard and PROV-Template

PROV [1] is a World Wide Web Consortium (W3C) standard that aims to facilitate the publication and interchange of provenance among applications. PROV is fully specified in a family of documents, which cover various of its aspects such as modeling, serialization, access, interchange, translation and ways to reason over it. For the purpose of our paper, we illustrate PROV focusing on the PROV Data Model (PROV-DM) [7], which is a conceptual model that forms the basis for the remainder PROV family of specifications, and the PROV Notation (PROV-N) [8], a textual representation suitable for human consumption.

PROV is based around three concepts, together with their relationships which are depicted in the left part of Figure 1. In the right part, we also show the PROV-N representation of these concepts, together with their graphical notation. More specifically, an *Entity* is a physical, digital, conceptual or other kind of thing. An *Activity* is a set of actions that act upon or with *entities* during a specific time frame. Finally, an *Agent* refers to something which takes responsibilities of *entities* or *activities* through attribution or association, respectively.

As shown in Figure 1, these concepts are associated through relationships such as usage (*used*), which represents an activity beginning to utilize an entity, generation (*wasGeneratedBy*) used when an activity produces a new entity, derivation (*wasDerivedFrom*) which denotes an entity update, invalidation (*wasInvalidatedBy*) used when an activity starts the destruction or invalidation of an entity, association (*wasAssociatedWith*) which indicates that an agent had a role in an activity, attribution (*wasAttributedTo*) which shows an agent bearing the responsibility for an entity, and specialization (*specializationOf*) used when an entity shares the aspects of another entity, but also has more specific aspects.

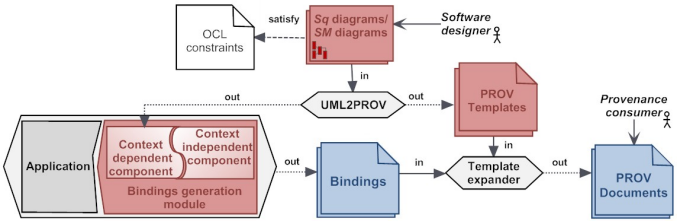

Fig. 2. The UML2PROV approach. The red and blue colours are used to refer to *design time* and *runtime* aspects of the approach, respectively.

PROV-Template [13] is a declarative approach to creating PROV compliant provenance-enabled applications. It consists of three main key elements: *provenance templates*, *bindings*, and a *provenance template expansion algorithm*. The overall process supported by PROV-Template is as follows. The *provenance templates* are firstly designed and embedded in the application's code, which logs the values in the form of *bindings* during its execution. Finally, provenance is automatically generated by *template expansion*. For further details regarding PROV and PROV-Templates, the reader is referred to [1,7,8] and [13], respectively.

2.2 UML Diagrams

UML [12] distinguishes two major categories of diagrams: *structural* diagrams are concerned with the static structure of a system, whereas *behavioural* diagrams capture the behavioural features of a system, including aspects concerning its runtime execution. This latter type of diagrams describes the dynamics between objects of a system in terms of states, interactions, collaborations, etc. Since provenance bears a strong relation with all the data taken part in producing a final item (that is, information related to involved *entities* together with the different *states* they go through over time, conducted *activities*, *interactions* among such entities, etc.), we considered *UML Sequence Diagrams* (*Sq Diagrams*) and *UML State Machine Diagrams* (*SM Diagrams*), to be the most suitable ones for our purpose. Briefly speaking, *Sq Diagrams* are used to model the *interactions* among collaborating objects in terms of *messages* exchanged from a *sender* to a *receiver's lifeline*. *SM Diagrams* specify the various states that an object goes through during its lifecycle. They mainly consist of *states*, *transitions* and other types of *vertexes* called *pseudostates*. For the sake of brevity, we do not delve into more detail regarding *Sq* and *SM Diagrams*; we refer the reader to [12].

3 Overview: Generating PROV Templates from UML

In this section, we provide an overview of the UML2PROV approach identifying its key facets, and distinguishing its different stakeholders: *software designers* and *provenance consumers*. We illustrate our explanations by means of Figure 2, where *design time* elements (red) are distinguished from *runtime* elements (blue). *Design time* facets are the *Sq/SM diagrams*, the associated *PROV templates* generated from those, and the *bindings generation module*. In particular, this module is composed by two main components: a *context-independent component*, which contains the bindings's 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's generation code specific to the concrete

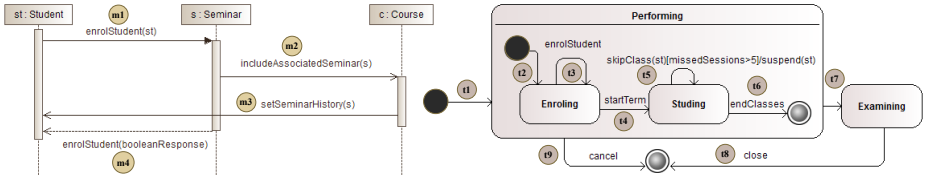

Fig. 3. On the left side, a *Sq diagram* showing the interaction between `Student`, `Seminar` and `Course`. On the right side, the *SM diagram* of the `Seminar` class.

application. The *runtime execution* facets consist of the values logged by the application, in the form of *bindings*, and the *PROV documents*.

Software designers are responsible for creating the *Sq* and *SM diagrams* based on the concrete domain's requirements (see upper part of Figure 2). Since *UML Sq* and *SM diagrams* show interconnected behavioural views of an overall system, before applying our approach, those diagrams must satisfy a set of Object Constraint Language (OCL) [14] rules we have defined to ensure that those diagrams are consistent with each other (for details about these rules, we refer to [15]). UML2PROV takes as input the *UML* diagrams satisfying such rules, and automatically generates: PROV *templates*, as defined by the *UML* to templates mapping (Section 4), and the *context-dependent component* in the *bindings generation module* (Section 5). UML2PROV determines (1) what provenance information is considered from the *Sq/SM diagrams* to be captured, and (2) how the application is wrapped with the functionality needed to allow such a capture (i.e. the functionality implemented by the *bindings generation module*).

Finally, the *provenance consumer* uses the provenance *template expander* to generate *PROV documents* from both the *templates* and the *bindings*. By distinguishing among the different stakeholders, we allow them having clearly defined roles and focusing on their specific responsibilities, avoiding task collision.

4 From UML Diagrams to provenance Templates

In this section, we present the mapping from *Sq* and *SM diagrams* satisfying our OCL constraints, to provenance templates. We have defined a set of patterns that identify commonly appearing structures on both *Sq* and *SM diagrams* and a set of translation rules that translate each single *UML* element involved in such patterns to PROV elements. We only outline the patterns due to space constraints, whereas a complete description of the rules is provided in [15]. To illustrate our explanations, we use a case study of a system that manages the enrolment and attendance of students to seminars of a University course. Figure 3 shows two *Sq* and *SM diagrams* defined for such a case study.

4.1 From Sequence Diagrams to Templates

We illustrate our translation approach by means of the *SeqP1-SeqP4* patterns presented in Figure 4, together with the template of Figure 5 which shows the translation of the *message* `m1` from the case study's *Sq diagram* in Figure 3.

For each pattern identified, the sender *object lifeline* is mapped to a `prov:Agent` (identified by `var:lifeline`) that assumes the responsibility of such an *object*

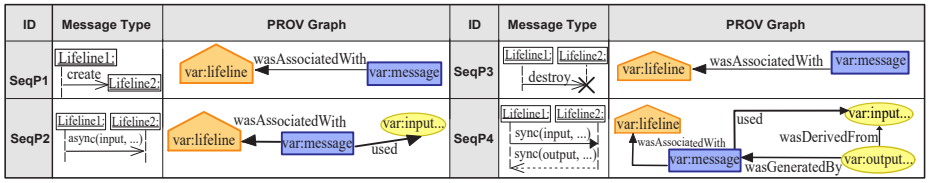

Fig. 4. Sq Diagrams' Patterns and their provenance templates

```
1 agent(var:lifeline,[prov:type='exe:Student'])
2 activity(var:message,[prov:type='exe:enrolStudent',
3       tmpl:startTime = '...', tmpl:endTime='...'])
4 entity(var:input0, [prov:value='var:input0value'])
5 entity(var:output0, [prov:value='var:output0value'])
6 wasAssociatedWith(var:message, var:lifeline,-,[])
7 used(var:message,var:input0,-,[prov:role='exe:st'])
8 wasGeneratedBy(var:output0,var:message,-,
9       [prov:role='exe:booleanResponse'])
10 wasDerivedFrom(var:output0,var:input0)
```

Fig. 5. An extract of a template generated from the case study's Sq diagram.

(e.g. in line 1 of Figure 5 we show how the *object* `Student` is translated into a `prov:Agent`). The *message* sent is modelled as a `prov:Activity` (identified by `var:message`) that represents the invocation of the message's *operation* (e.g. the *message* `enrolStudent` is mapped to the `prov:Activity` showed in lines 2-3 of Figure 5). Additionally, when an *object lifeline* sends a *message* to another *lifeline*, a new `prov:wasAssociatedWith` relationship is generated between the *message* identified by `var:message`, and the sender *lifeline* identified by `var:lifeline` (e.g. the statement in line 6 of Figure 5 shows this relationship).

Patterns *SeqP2* and *SeqP4* depict the communication between two *lifelines* through a reply *asynchronous/synchronous message* with *arguments*. Each *message's argument* is modelled as a `prov:Entity`, identified by `var:input...` Additionally, to assert that the *argument* is a parameter of the request *message*, the relationship `prov:used` links the *message* `var:message` and the *argument* `var:input...` Focusing on the *message* `m1` in Figure 3, the *argument* `st` is translated into the `prov:Entity` showed in line 4 of Figure 5, together with the link between the identifiers of both the *argument* and the *message*, shown in line 7.

SeqP4 additionally encompasses a *reply message* with an *output argument*. Additionally, the *output argument* is modelled as a `prov:Entity` (identified by `var:output...`) that was "generated" as part of the *reply*. Thus, the relationship `prov:wasGeneratedBy` is created between the *message* identified by `var:message` and the *argument* `var:output...` Regarding the reply *message* `m4` in Figure 3, the *output argument* is translated into the `prov:Entity` showed in line 5 of Figure 5, while its relation with the *message* `prov:Activity` is shown in lines 8-9.

We note that in PROV two relationships of the form (`B`, `prov:used,A`) and (`C`, `prov:wasGeneratedBy`, `B`) are usually enriched with (`C`, `prov:wasDerivedFrom,A`) to express the dependency of `C` on `A`. This structure refers to a provenance construction called *Use-generate-derive triangle* [16] which includes the three elements involved. *SeqP4* in Figure 4 depicts such a situation between the request's and the response's *arguments*: when both request and reply *messages* have *arguments*, we use the `prov:wasDerivedFrom` relationship. In line 10 of Figure 5 we reflect such a situation between the *input* and *output arguments* of `enrolStudent`.

4.2 From State Machine Diagrams to Templates

We now present the mapping from *SM diagrams* to provenance templates. Our explanation is illustrated by using the *StP1-StP6* patterns presented in Figure 6

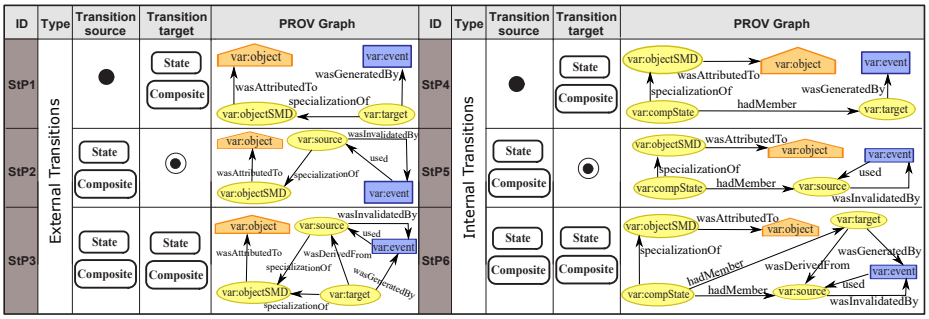

Fig. 6. Patterns identified in *SM diagrams*

and the *provenance template* showed in Figure 7, which depicts an extract of the translation resulted from the case study's *SM diagram* in Figure 3.

SM Diagrams represent the evolution of an *object* using *transitions* between *states*. In fact, among the patterns depicted in Figure 6, we can identify four common UML elements shared by all of them. (1) The *object* whose behaviour is modelled by the *SM diagram* is translated into a `prov:Agent` identified by `var:object` (e.g. in line 1 of Figure 7 the object `Seminar` whose behaviour is modelled by the *SM diagram* in Figure 3 is translated into a `prov:Agent`). (2) The *object*'s *state machine* is represented as a `prov:Entity` (identified by `var:objectSMD`). Additionally, `var:objectSMD` is related to the *object*, identified by `var:object`, using `prov:wasAttributedTo` relationship (e.g. the *object*'s *state machine* of Figure 3 is translated into the `prov:Entity` in line 2, which is associated with the corresponding *object* by means of line 11). (3) The *event* that triggers a *state* change is translated into a `prov:Activity` identified by `var:event` (e.g. the *event* `enrolStudent` is represented by the `prov:Activity` in line 3-4 of Figure 7). Finally, (4) the *state*, *simple* or *composite*, which denotes the *object*'s situation is mapped to a `prov:Entity` identified by `var:source`, `var:target` or `var:compState`. For example, the *source state*, the *target state*, and the *composite state* involved in the transition `t3` of Figure 3 are translated into the `prov:Entity` showed in lines 5-6, 7-8, 9-10 of Figure 7, respectively. To represent that the source *state* influences the outcome of a *transition*, we adopt the `prov:used` relationship between the source *state* identified by `var:source` and the *event* identified by `var:event`. Additionally, to represent that the *object* is no longer in the source *state*, the relationship `prov:wasInvalidatedBy` links the source *state* `var:source` and the *event* `var:event`. Finally, to represent that the target *state* results from the triggering of the *transition*, a `prov:wasGeneratedBy` relationship links the target *state* `var:target` and the *event* `var:event`. For instance, focusing on the *transition* `t3` in Figure 3, the source *state* `Enroling` represented by a `prov:Entity` and the *event* `enrolStudent` represented by a `prov:Activity` are linked by the relationships `prov:used` and `prov:wasInvalidatedBy` depicted in lines 12 and 13 of Figure 7. In addition, the target *state* `Enroling` represented by a `prov:Entity` is related to the *event* `enrolStudent` represented by a `prov:Agent` by means of the relationship `prov:wasGeneratedBy` shown in line 14.

Although these patterns share the previous cited aspects, the complete translation of all the elements within a *SM diagram* depends on the particular nuances such as the target/composite elements and the type of *transition* (internal or external). Whenever the *transition* is not enclosed within a *composite state* (*StP1-*

```
1  agent(var:object,[prov:type='exe:Seminar'])
2  entity(var:objectSMD,[prov:type='exe:StateMachine'])
3  activity(var:event,[prov:type='exe:enrolStudent',
4        tmpl:startTime = '...', tmpl:endTime='...'])
5  entity(var:source, [exe:state='exe:Enroling',
6        prov:type='exe:Seminar'])
7  entity(var:target, [exe:state='exe:Enroling',
8        prov:type='exe:Seminar'])
9  entity(var:performing, [exe:state='exe:Performing',
10       prov:type='exe:Seminar'])
11 wasAttributedTo(var:objectSMD, var:object,[])
12 used(var:event,var:source,-)
13 wasInvalidatedBy(var:source, var:event,-)
14 wasGeneratedBy(var:target,var:event,-)
15 wasDerivedFrom(var:target,var:source)
16 specializationOf(var:performing, var:objectSMD)
17 hadMember(var:performing,var:source)
18 hadMember(var:performing,var:target)
```

Fig. 7. An extract of a template generated from the case study's *SM diagram*.

StP3), its *source* and *target states* are related to the *state machine*, identified by `var:objectSMD`, through `prov:specializationOf`. In contrast, if the *transition* is enclosed within a *composite state* (*StP4-StP6*), its *source* and *target states* (identified by `var:source` and `var:target`, respectively) are related to the *composite state* (identified by `var:compState`) through `prov:hadMember`. Additionally, the *composite state* is related to the *state machine* using `prov:specializationOf`. For instance, since the *transition* `t3` in Figure 3 is enclosed in a *composite state*, it follows the pattern *StP6*. Thus, its source and target *states* are related to the *composite state* by the statements in lines 17 and 18 of Figure 7, while the *composite state* is linked to the *state machine* by line 16.

Finally, similarly to Section 4.1, *StP3* and *StP6* exploit the *Use-generate-derive triangle* [16] among the source *state* `var:source`, the *event* `var:event` and the target *state* `var:target`. Thus, we define a direct relationship between both the `var:source` and the `var:target` by means of the `prov:wasDerivedFrom` relationship, representing the fact that the *target* state is a consequence of the triggering of the *transition* from the *source state*. In line 15 of Figure 6 we reflect such a situation between the *source* and *target states* of transition `t3`.

5 Bindings Generation Strategy

As explained in Section 2, the PROV-Template approach takes a *provenance template* together with a set of *bindings* as input of the *template expansion* process. Such a process replaces variables in the *provenance templates* by real values in the *bindings*, producing *PROV documents*. Obtaining the *bindings* becomes a key focus of the runtime execution, requiring adaptation of existing application code. Although a manual adaptation of the source code is a valid option to extract *bindings*, software engineers would need to expend a great deal of effort on traversing the overall application's source code, and adding suitable instructions to generate the *bindings* structures. Thus, it would constitute a tedious, time-consuming and error prone process. To avoid that, PROV2UML creates *bindings* automatically by applying the *Proxy Pattern* [17], thus requiring minor modifications, without obfuscating the existing code with new statements.

Briefly speaking, the *Proxy Pattern* provides a surrogate for another object to control its behaviour. It is mainly intended to manage the access to objects' methods, allowing us to modify their behaviour. This benefit has led to a wide use of this pattern in, for example, Aspect-Oriented Programming (AOP)-based frameworks. The *Proxy Pattern* is composed of the following four elements. (1) The *Subject Interface* includes all the methods implemented by the *Real Subject*. (2) The *Real Subject* is the object whose behaviour we want to modify, must implement the *Subject Interface*. (3) The *Proxy* element also implements the

```
{"var":{
  "source":[{"@id":"exe:Seminar@57fa26b7_1"}],
  "event":[{"@id":"exe:enrolStudent_1"}],
  "lifeline":[{"@id":"exe:Student@28180122"}],
  "performing":[{"@id":"exe:Performing_1"}],
  "objectSMD":[{"@id":"exe:Seminar@57fa26b7_0"}],
  "object":[{"@id":"exe:Seminar@57fa26b7"}],
  "eventStartTime":[{"@type":"xsd:dateTime",
                     "@value":"2017-02-09T11:54:24"}],
  "target":[{"@id":"exe:enrolStudent_1"}],
  "eventEndTime":[{"@type":"xsd:dateTime",
                   "@value":"2017-02-09T11:54:24"}]},
  "vargen":{},
  "context":{"exe":"http://uml2prov.../execution#"}}
```


Fig. 8. Example of bindings collected from the method `enrolStudent` in Figure 3

Subject Interface so that it can be used in any location where the *Real Subject* can be used. The *Proxy* element maintains a reference to the *Real Subject* and executes its own code before and after the *Real Subject*'s usual execution. (4) The *Client* element is in charge of invoking the *Subject*, which allows the *Client* to interact with the *Proxy* as though it were the *Real Subject*. Thus, the *Proxy* constitutes the intermediary between the *Client* and the *Real Subject*. This pattern helps us collect suitable information to construct the *bindings* before and after the usual execution of the objects' methods. Harnessing the potential of this pattern to generate the *bindings* has two main advantages: (1) we deal with the concept of proxy independently of any programming language, and (2) this solution is suitable for both already developed applications, and applications yet to be developed. In particular, the *Proxy* element wraps the *Real Subject* allowing us to extract *provenance* information for each method defined in the *Subject Interface*. When a method is called, the *Proxy* intercepts the method invocation and gathers concrete information about the system execution (e.g. time) and specific information about the method (such as the parameters). We note that each captured value is directly related to a variable included in a provenance template (e.g. `var:message` value is given by the name of the method).

In Figure 8, we show an example of bindings in JSON format representing the bindings captured when the *transition* `t3` in Figure 3 is triggered. More specifically, it shows the *bindings* between several variables appearing in the provenance templates of Figure 7 and their corresponding values; for example, the variable `event` is associated with the concrete value `exe:enrolStudent_1`.

6 Implementation

In this section, we discuss a reference implementation of UML2PROV in Java. Regarding the translation of *UML* to provenance templates, we have chosen Extensible Stylesheet Language Transformations (XSLT) [18] to implement the patterns. More specifically, we have defined two XSLT transformation files, each one tackling a type of diagram (*Sq* and *SM diagrams*). The diagrams are expected to be encoded in XMI format, a standardized XML representation for UML diagrams supported by mainstream UML designers such as UML 2 Eclipse plug-in, Modelio [19] or Papyrus [20]. We use Papyrus which not only is able to represent *UML* diagrams graphically, serialising them into XMI, but it is also able to check OCL constraints on UML diagrams, that is, it allows us to verify our OCL constraints on the source diagrams before applying UML2PROV. The XMI files are taken as input by each XSLT transformation, which automatically generates the corresponding provenance templates in PROV-N.

Aiming at generating bindings for Java applications, we provide a Java class named as `ProxyProvGenerator` which relies on the `java.lang.reflect` package. Basically, this class has a method which receives a *subject object* implementing its

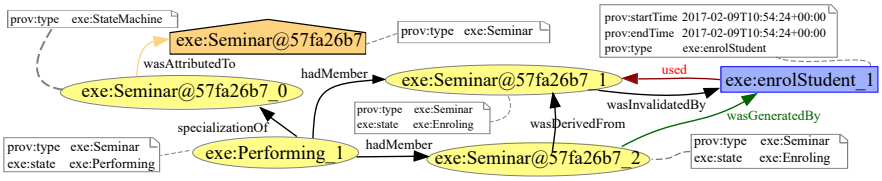

Fig. 9. Expanded PROV document

corresponding *subject interface* and then, the method returns the *subject object*'s *proxy*. Such a *proxy* is created with all the bindings generation instructions within. The `ProxyProvGenerator` is application independent since it is agnostic about the *subject object* given. Providing the `ProxyProvGenerator` to the software developer is enough to automatically generate a proxy for each *subject object* with provenance capturing capabilities. Thus, this class constitutes the *context-independent component* in the *bindings generation module*.

We have applied the UML2PROV implementation to the case study in Figure 3 obtaining 3 and 6 templates from the *Sq diagram* and the *SM diagram*, respectively (Figures 5 and 7 show actual extracts of such provenance templates). Figure 9 depicts the PROV document generated from the set of bindings shown in Figure 8 and the template from Figure 7, by applying the *template expander*.

7 Quantitative Evaluation and Discussion

This section evaluates the strengths and weaknesses of UML2PROV. More specifically, we have applied it to five case studies and analysed the results in the light of several criteria pertaining to *design time*: (1) the number of generated provenance template elements, (2) the time that took to generate the templates, and (3) the amount of automatically generated code. As for *runtime execution*, we discuss (4) how much provenance is being generated after expansion.

Table 1 depicts the results given by applying UML2PROV to the five case studies, organized depending on the type of diagram. The first case study (CS1) corresponds to the complete seminars' system. The remainder case studies, which have been selected from Internet because their diagrams are varied in size, are associated to a water system (CS2), a system representing the Model-View-Controller pattern (CS3), a phone call system (CS4), and an elevator system (CS5). The relevant documents related to the case studies can be found on [15].

Regarding the analysis of (1) the number of provenance template elements that are generated, and (2) the time that took to generate such templates, we study the relation between the number of UML elements and the number of PROV elements, as well as, the relation between the number of UML elements and the translation time taken. With this study we check the capability of UML2PROV to handle the growing amount of UML elements and its potential to accommodate such a growth. In particular, we observe that the average time (in *Sq* and *SM diagrams*) is significantly larger for the CS5 case study, but likewise, the average size of generated PROV elements for this application is larger. This confirms that the cost per UML element remains constant. To validate this, we applied Pearson's correlation test and obtained a ρ-value of 0.9978 (relating to *Sq diagrams*' elements) and a ρ-value of 0.9713 (relating to *SM diagrams*' elements) showing a strong correlation. Similarly, we have computed the

Table 1. Results obtained from the cases studies using a personal computer, Intel(R) CoreR^{TM} i7 CPU, 3.6 GHz, with 16 GB RAM, running Windows 10 Enterprise.

Id	UML Elements			PROV elements			Interf. code lines	Avera. time (ms)
	Diagr. Type	Num. Diagr.	Num. Diagr. Elemen.	Templ. Num.	Num. PROV Elemen.	Var. Num.		
CS1	SqD	1	10	3	18	83	11	45.4
	SMD	3	19	9	146			25
CS2	SqD	4	18	8	40	88	12	53
	SMD	2	22	8	163			22
CS3	SqD	1	17	5	34	56	13	50.2
	SMD	4	20	6	148			23.6
CS4	SqD	1	12	5	25	84	11	47.2
	SMD	3	16	8	117			22.2
CS5	SqD	2	50	9	67	131	47	90.4
	SMD	5	52	13	369			37.4

Legend:

Num. Diagr. Number of SqD and SMD diagrams modelling the system.

Num. Diagr. Elemen. The total number of elements within each diagram (lifelines, messages, arguments, transitions, and simple and composite states).

Templ. Num. The number of generated PROV templates.

Num. PROV Elemen. The number of PROV template elements.

Var. Num. The number of variables in these templates.

Inferf. code lines. The lines of code in the generated subject interfaces.

Avera. Time (ms). The average time taken by 12 executions of the translation process.

Pearson's correlation coefficient to measure the strength of the linear association between the number of source *UML* elements and the generated PROV elements, obtaining a ρ-value of 0.9660 (for *Sq diagrams*' elements) and a ρ-value of 0.9996 (for *SM diagrams*' elements), which demonstrates good performance results.

As for the code required to be created for bindings generation, as explained in Section 5, UML2PROV only requires the *Subject Interfaces* to be created, which are used together with the `ProxyProvGenerator` class. Since such interfaces are automatically generated by UML2PROV from the source *UML* diagrams, software developers do not have to develop them manually, and thus, they do not need to write the number of lines of code presented in Table 1 (see column "Interf. code lines"). Without using UML2PROV, software developers would have to write additional code within the application to create bindings. Typically, for each variable in a template, a method call is needed to assign a value to it, thus, a developer would need to write one line of code for each variable in a template. In our five case studies, although being relatively small, these number of lines of code are presented in column "Var. Num." in Table 1. With UML2PROV, writing such code is not required, since the proxy constructs that automatically.

Finally, regarding the provenance obtained after expansion, we would like to note that, in case of a repetitive cycle or sequence of actions in the *Sq diagrams*, the number of PROV documents obtained after the expansion process grows proportionally to the length of these cycles or sequences.

8 Related Work

Although provenance has been widely addressed from different perspectives [21,22,23,24], to the best of our knowledge, it has been scarcely investigated from the point of view of determining the provenance to be generated as software is being designed. In contrast to our proposal, other works undertake the development of provenance-aware systems by means of weaving provenance generation instructions into programs, which makes code maintenance a cumbersome task. Examples of these include PASS [25], which is a storage system which supports the automatic collection and maintenance of provenance; PERM [3], which is a provenance database middleware that enables provenance computation; and finally, workflow systems such as Taverna [4], Vistrails [5] and Kepler [6] which incorporate provenance capabilities into the workflow system.

Alternatively, there are different approaches that include provenance generation instructions into source code. For instance, Ghosal et al. [26] extract provenance from log files, Cheney et al. [27,28] use statistic analysis to create

executables that produce provenance information, and Brauer et al. [29] use an Aspect-Oriented Architecture to interweave aspects generating provenance. This approach bears relationship with our work since, as discussed previously, the *Proxy Pattern* used in our approach is widely applied in AOP. However, UML2PROV not only gives a general solution to include provenance with minimum interferences with the original system, but it also addresses the design of the provenance to be generated using PROV-Template [11].

Finally, it is worth mentioning the standalone methodology PrIMe [11]. It could be said that UML2PROV complements PrIMe, since UML2PROV integrates the design of provenance by means of PROV-Templates with the design of applications using the well-known de-facto standard notation *UML*.

9 Conclusions and Future Work

Bridging the gap between application design and provenance design remains an adoption hurdle for provenance technology. In this paper, we present UML2PROV that addresses such a challenge for the particular case of *Sq* and *SM Diagrams*, taken as design methodology, and PROV-Template, used as provenance design. Our contributions are as follows: (i) a mapping of *UML* diagrams to provenance templates, (ii) a code generation technique that creates libraries to be linked with the application to generate provenance, and (iii) a demonstration of the feasibility of UML2PROV by providing an implementation, and a preliminary quantitative evaluation that shows significant benefits of the approach. Our evaluation shows that our approach significantly reduces efforts in design time, resulting in an increased productivity. The automated provenance capture also provides clear benefits over the traditional approach of provenance capture, showing the amount of code that software developers will need to write without UML2PROV. The experiments also confirm that the approach is tractable, requiring milliseconds for generating PROV templates.

Although our proposal takes into account two of the most used *UML* behavioural diagrams, considering a wider number of *UML* elements, including other kind of *UML Diagrams* (such as *UML* Activity Diagrams), and other elements (such as *SM Diagram*'s *pseudostates*, not considered in our patterns) to constitute a more complete provenance-aware methodology, is a line of further work. Additionally, using a strategy based on, for example, *UML* stereotypes, to monitoring only concrete messages, constitutes an interesting direction of further work. We use XSLT as a first attempt to implement our patterns; other approach of future work is to consider using a Model Driven Development (MDD) tool chain based on MDD-based tools such as ATL [30] and XPand [31]. Finally, performing a systematic quantitative evaluation of the approach and a study of the quality of provenance being generated from a real situation (involving users, designers or developers) constitute another line of future work.

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. Groth, P., Moreau (eds.), L.: PROV-Overview. An Overview of the PROV Family of Documents. W3C Working Group Note NOTE-prov-overview-20130430, World Wide Web Consortium (April 2013) http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.
2. 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 the International Provenance and Annotation Workshop, (IPAW'08). (2008) 98–115
3. 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
4. Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., et al.: The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. Nucleic acids research (2013) 557–561
5. 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
6. Altintas, I., Barney, O., Jaeger-Frank, E. In: Provenance Collection Support in the Kepler Scientific Workflow System. (2006) 118–132
7. Moreau, L., Missier (eds.), P., Belhajjame, K., B'Far, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., Klyne, G., Lebo, T., McCusker, J., Miles, S., Myers, J., Sahoo, S., Tilmes, C.: 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/.
8. Moreau, L., Missier (eds.), P., Cheney, J., Soiland-Reyes, S.: PROV-N: The Provenance Notation. W3C Recommendation REC-prov-n-20130430, World Wide Web Consortium (April 2013) http://www.w3.org/TR/2013/REC-prov-n-20130430/.
9. A library for W3C Provenance Data Model supporting PROV-JSON, PROV-XML and PROV-O (RDF), P.: (October 2017) https://pypi.python.org/pypi/prov. Last visited on October 2017.
10. ProvToolbox. Java library to create and convert W3C PROV data model representations: http://lucmoreau.github.io/ProvToolbox/. Last visited on October 2017.
11. 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
12. OMG. Unified Modeling Language (UML). Version 2.5: (2015) formal/15-03-01, http://www.omg.org/spec/UML/2.5/.
13. Moreau, L., Batlajery, B.V., Huynh, T.D., Michaelides, D., Packer, H.: A templating system to generate provenance. IEEE Transactions on Software Engineering (2017 (In Press)) http://eprints.soton.ac.uk/405025/.
14. OMG: Object Constraint Language, Version 2.4 (2014) formal/2014-02-03 http://www.omg.org/spec/OCL/2.4/PDF.
15. Supplementary material of UML2PROV: (October 2017) https://uml2prov.github.io/. Last accessed October, 2017.
16. Kwasnikowska, N., Moreau, L., Bussche, J.V.D.: A formal account of the open provenance model. ACM Trans. Web **9**(2) (May 2015) 10:1–10:44

17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison Wesley (1995)
18. XSL Transformations (XSLT) Version 3.0: (February 2017) W3C Recommendation 8 June 2017. https://www.w3.org/TR/xslt-30/.
19. Modelio, UML modeling tool. Version 3.6: (february 2017) http://www.modeliosoft.com/. Last visited on October 2017.
20. Papyrus, Modeling environment . Version 2.0.2 (Neon release): (January 2017) https://eclipse.org/papyrus/. Last visited on October 2017.
21. Tan, W.C.: Provenance in Databases: Past, Current, and Future. IEEE Data Eng. Bull. **30**(4) (2007) 3–12
22. Davidson, S.B., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (MOD'08), New York, NY, USA, ACM (2008) 1345–1350
23. Moreau, L.: The Foundations for Provenance on the Web. Foundations and Trends in Web Science **2**(2–3) (2010) 99–241
24. Simmhan, Y.L., Plale, B., Gannon, D.: A Survey of Data Provenance Techniques. Technical Report 612 Extended version of SIGMOD Record 2005. Available at: http://www.cs.indiana.edu/pub/techreports/TR618.pdf.
25. Glavic, B., Dittrich, K.R.: Data Provenance: A Categorization of Existing Approaches. In: Proceedings of Datenbanksysteme in Bro, Technik und Wissenschaft (BTW'07). (2007) 227–241
26. Ghoshal, D., Plale, B.: Provenance from log files: a bigdata problem. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops, ACM (2013) 290–297
27. Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. **21**(6) (2011) 1301–1337
28. Cheney, J.: Program slicing and data provenance. IEEE Data Eng. Bull. **30**(4) (2007) 22–28
29. Brauer, P.C., Fittkau, F., Hasselbring, W.: The aspect-oriented architecture of the caps framework for capturing, analyzing and archiving provenance data. In: International Provenance and Annotation Workshop, Springer (2014) 223–225
30. Jouault, F., Kurtev, I.: Transforming models with atl. In: International Conference on Model Driven Engineering Languages and Systems, Springer (2005) 128–138
31. XPand: Eclipse platform (2017) https://wiki.eclipse.org/Xpand, Last visited on October 2017.