Integrating Provenance Capture and UML with UML2PROV: Principles and Experience

Carlos Sáenz-Adán, Beatriz Pérez, Francisco J. García-Izquierdo, Luc Moreau

Abstract—In response to the increasing calls for algorithmic accountability, UML2PROV is a novel approach to address the existing gap between application design, where models are described by UML diagrams, and provenance design, where generated provenance is meant to describe an application's flows of data, processes and responsibility, enabling greater accountability of this application. The originality of UML2PROV is that designers are allowed to follow their preferred software engineering methodology to create the UML Diagrams for their application, while UML2PROV takes the UML diagrams as a starting point to automatically generate: (1) the design of the provenance to be generated (expressed as *PROV templates*); and (2) the software library for collecting runtime values of interest (encoded as variable-value associations known as *bindings*), which can be deployed in the application without developer intervention. At runtime, the PROV templates combined with the bindings are used to generate high-quality provenance suitable for subsequent consumption. UML2PROV is rigorously defined by an extensive set of 17 patterns mapping UML diagrams to provenance templates, and is accompanied by a reference implementation based on Model Driven Development techniques. A systematic evaluation of UML2PROV uses quantitative data and qualitative arguments to show the benefits and trade-offs of applying UML2PROV for software engineers seeking to make applications provenance-aware. In particular, as the UML design drives both the design and capture of provenance, we discuss how the levels of detail in UML designs affect aspects such as provenance design generation, application instrumentation, provenance capability maintenance, storage and run-time overhead, and quality of the generated provenance. Some key lessons are learned such as: starting from a non-tailored UML design leads to the capture of more provenance than required to satisfy provenance requirements and therefore, increases the overhead unnecessarily; alternatively, if the UML design is tailored to focus on addressing provenance requirements, only relevant provenance gets to be collected, resulting in lower overheads.

Index Terms—provenance, PROV, provenance generation, template

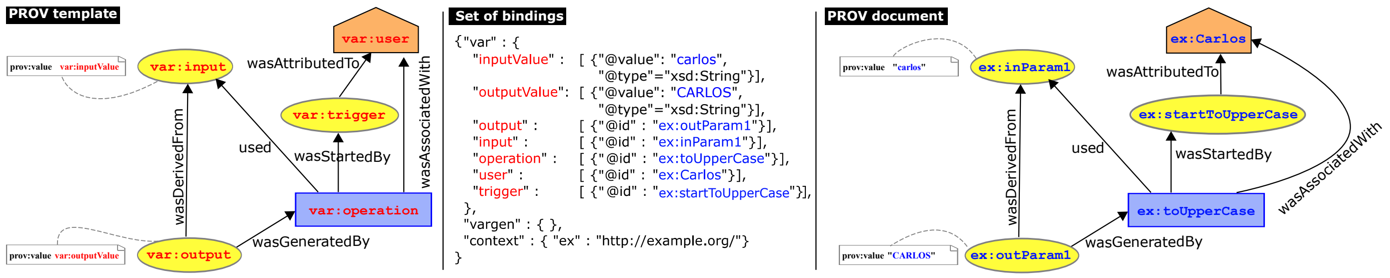
1 Introduction

PROVENANCE has been defined by the W3C as "the record 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]. Harnessing the potential of provenance allows a great number of benefits such as making systems accountable –e.g., by explaining how decisions were reached– or verifying and reproducing a data product [2]. With such tangible benefits, it is no wonder that in recent years, provenance has rapidly gained much traction in different application domains ranging from forest management to climate science, from medicine to business application [2], [3], [4]. This growing interest in provenance is evidenced by the vast scientific literature that has been reviewed by several surveys [2], [5] which study different solutions to make applications provenance-aware, i.e. to design applications keeping in mind the provenance capabilities. The traction in provenance together with the emerging provenance-aware systems have led the provenance community to publish the W3C PROV family of specifications [1] in order to increase interoperability between systems. Several toolkits have been developed supporting PROV aiming at facilitating the software engineer's tasks of creating, storing, reading and exchanging provenance [6], [7]. Nevertheless, not only do such tools lack a way of deciding what provenance information should be considered, but they also do not explain how software should be designed and implemented for provenance collection. In this context, considering the use of provenance at design time constitutes a great advantage to support software designers in making provenance-aware systems. In this line, the *Provenance Incorporation Methodology* (PrIMe) [8] was introduced to adapt applications to be provenance-aware. However, whilst this methodology has demonstrated promising results [8], PrIMe is standalone, that is, it does not integrate with existing software engineering methodologies, which makes it challenging to use it in practice.

In the realm of software engineering, a vast amount of techniques have been proposed to increase the quality of software products, while reducing their development time and preventing failures in applications' conception, design, and construction. Of those techniques, the U*nified* M*odelling* L*anguage* (UML) [9] is considered as the most widespread standard for designing object-oriented applications in industry [10]. Although UML comprises many types of diagrams aiming at describing different perspectives of a system (e.g., structural information or behaviour), it does not provide specific support for provenance. Concretely, it lacks specific tools to design systems that can answer questions such as what is the set of activities that have led a particular entity to be as it is?, or what are the previous states that the entity has gone through? Moreover, developing or adapting

- *Carlos Sáenz-Adán, Beatriz Pérez, and Francisco J. García-Izquierdo are with the Department of Mathematics and Computer Science, University of La Rioja, La Rioja, Spain.*
  E-mail: {carlos.saenz,beatriz.perez,francisco.garcia}@unirioja.es
- *Luc Moreau is with the Department of Informatics, King's College London, London, UK.*
  E-mail: luc.moreau@kcl.ac.uk

Manuscript received on Wednesday 15th January, 2020



```
{"var" : {
   "inputValue" :   [ {"@value": "carlos",
                       "@type"="xsd:String"}],
   "outputValue":  [ {"@value": "CARLOS",
                       "@type"="xsd:String"}],
   "output" :        [ {"@id" : "ex:outParam1"}],
   "input" :         [ {"@id" : "ex:inParam1"}],
   "operation" :     [ {"@id" : "ex:toUpperCase"}],
   "user" :          [ {"@id" : "ex:Carlos"}],
   "trigger" :       [ {"@id" : "ex:startToUpperCase"}],
 },
 "vargen" : { },
 "context" : { "ex" : "http://example.org/"}
}
```

Fig. 1. From left to right, a graphical illustration of: a *PROV template*, representing the design of the provenance, a set of *bindings*, linking variables from the PROV template with values, and a *PROV document*, resulting after expanding the previous PROV template with the set of bindings.

software applications for collecting provenance from the design phase constitutes a cumbersome task that may entail significant changes to the application design [8]. Thus, the problem with the current state of the art is that designers and developers are forced to cope with several issues such as being knowledgeable about provenance, interweaving provenance aspects into the application's diagrams (usually making them complex and unreadable), or maintaining both application and provenance-specific code. Against this background, PROV-Template [3] has been proposed as a declarative approach to creating provenance compatible with the PROV standard. It offers great benefits with respect to previous proposals in terms of reduction of both storage needs, and development and maintenance effort, thanks to the separation of concerns between the design of the provenance to be generated (represented by *PROV templates*) and the collected provenance data (expressed by *bindings*). However, the use of PROV-Template does not free software designers from the need to be familiar with provenance. Thus, a research question is how to facilitate the design and collection of provenance by using as source the application's UML diagrams that model the functionality of the application, while leveraging existing tools such as PROV-Template to generate provenance. This question is to be investigated from a software engineering perspective, with a view to understand how a solution can facilitate development undertaken by software engineers.

This paper's proposal is UML2PROV, with three contributions over the state of the art. First, we present UML2PROV, a conceptual proposal that, starting from the UML design of an application, automatically generates: (i) the structure of the provenance to be generated (i.e. the *PROV templates*), and (ii) a library to be linked with the application to capture provenance data during the application execution (i.e. the *Bindings Generation Module*, also called *BGM*). Second, we give a reference implementation of UML2PROV. Third, we demonstrate the feasibility of UML2PROV by means of an evaluation that shows significant benefits of the approach. These benefits, which will appeal to designers in early stages of the development process, are mainly related to: (1) *design and development*, since we provide a way to include provenance capabilities during the design phase without changing the way in which software designers use UML (provenance is handled automatically from such UML diagrams), and (2) *capture of provenance*, since it is performed automatically in a non-intrusive manner thanks to the BGM generated by UML2PROV, and which provides clear benefits over more traditional approaches of provenance capture. While we partially described in previous publications [11], [12] our first attempts towards the definition of UML2PROV, we now provide a rigorous revision and an extended version of those works. This paper, together with its extensive supplementary material [13], explains our approach for the first time in its entirety. More specifically, we propose a new set of 17 patterns mapping UML diagrams to PROV templates, which rigorously revises and reformulates those partially presented in [11], [12]. In addition, the approach is now accompanied by a complete reference implementation of UML2PROV [14], which includes an event-based proposal of *BGMs*. Finally, this paper comprises an extensive evaluation, not included in previous works, which not only shows the benefits and costs of using UML2PROV, but it also analyses how different aspects regarding the UML design (the level of detail, number of UML diagrams involved, etc.) could affect the design and capture of provenance.

This paper is organized as follows: after the background Section 2, Section 3 gives an overview of UML2PROV. Section 4 describes our approach to translate UML Diagrams to PROV templates, while the bindings generation is described in Section 5. Section 6 explains a reference implementation of UML2PROV. We evaluate the UML2PROV feasibility in Section 7. Sections 8 and 9 discuss our proposal and its threats to validity, respectively. Last Sections present related work, conclusions and further work.

2 Background

We assume the reader is familiar with basic terminology of UML Sequence (SqDs), State Machine (SMDs) and Class (CDs) Diagrams (further details we refer the reader to [9]). As for PROV, and because of its relevance to this work, we provide some background information regarding the PROV Data Model (PROV-DM) [15], as well as PROV-Template [3], focusing on those aspects involved in our proposal.

The PROV Data Model (PROV-DM) [15] is based around three key elements, together with their relationships. Figure 1 depicts on the right-hand side a graphical example of a PROV document, where the three key elements of PROV are included: *Entity*, *Activity* and *Agent*, respectively represented by yellow ovals, blue rectangles and orange pentagons. PROV defines an *Entity* as a physical, digital, conceptual or other kind of thing with some fixed aspects. An *Activity* is defined as an occurrence of something taking place over a period of time and acting upon or with *entities*. Finally, an *Agent* is something that bears some form of responsibility for an *activity*, an *entity* or another

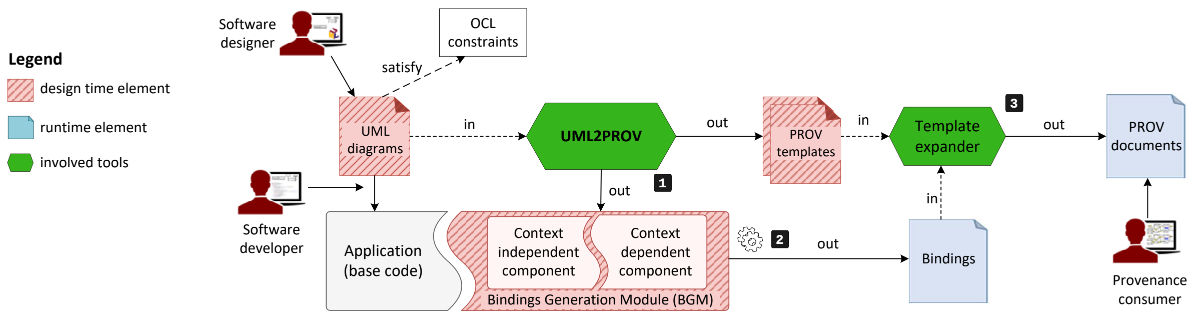


Fig. 2. The architecture of UML2PROV approach

agent. These elements can be related with each other by means of associations, among which we highlight: *used*, which is the beginning of utilizing an *entity* by an *activity*; *wasGeneratedBy*, which is the completion of production of a new *entity* by an *activity*; *wasDerivedFrom*, which is a transformation of an *entity* into another; *wasAssociatedWith*, which is an assignment of responsibility to an *agent* for an *activity*; *wasAttributedTo*, which is the ascribing of an *entity* to an *agent*; and finally, *wasStartedBy*, which is when an *activity* is deemed to have been started by an *entity*. In addition, the PROV Data Model includes other relationships (not appearing in Figure 1) of relevance to this work, such as *wasInvalidatedBy*, which is the start of the destruction, cessation or expiring of an existing *entity* by an *activity*; *specializationOf*, utilised for showing an *entity* which shares the aspects of another *entity*, but also has more aspects; and *hadMember*, used for stating the members of an *entity*.

PROV-Template [3] builds on top of PROV as a templating system for provenance by drawing a distinction between the provenance design and the creation of provenance data. It is made up of three key elements. First, *PROV Templates* offer a language to design the provenance to be generated. They are mainly PROV documents containing variables (also called placeholders). Second, *Bindings* are associations between variables and values, which a provenance-aware application is meant to construct at runtime. Finally, an *Expansion Algorithm* replaces each variable from the *templates* with data values from the *bindings*, generating an *expanded PROV document*. The left-hand side of Figure 1 depicts a *template* as a PROV document, in which the prefix `var` identifies variables. This *template* shows an activity `var:operation`, which *used* an entity `var:input` for its execution, and generated another entity `var:output` (*derived* from `var:input`). The activity was *started* by the entity `var:trigger` and is *associated with* the agent `var:user`. The entity `var:trigger` is also related to the agent `var:user`. Based on this *template*, together with the values associated to its variables given by the set of *bindings* collected during the application execution (center of Figure 1), the *expansion algorithm* generates a final PROV *document* (right of Figure 1).

3 UML2PROV Architectural Overview

Figure 2 depicts the key facets of UML2PROV and the different stakeholders involved in the process –the *software designer* and the *developer* at the beginning, and the *provenance consumer* at the end. The architecture distinguishes between the artifacts used or created at *design time*, which can be documents or code (red background with a stripped texture), from those generated at *runtime* (blue plain background). The *design time* facets encompass three elements. First, the *UML diagrams* depict the application's design. Second, the *PROV templates* form the design of the provenance to be generated. Finally, the generated *BGM* has two components: (i) the *context-dependent component*, which includes the code for generating bindings, and (ii) the *context-independent component*, which contains the bindings' generation code that is common to all applications. On the other hand, the *runtime execution* facets correspond to the generated sets of *bindings*, and the *PROV documents* containing the provenance information suitable for consumption.

The overall process starts with the assumption that there is already a UML design of the application we want to enrich with provenance capabilities. The diagrams comprising this design can be those used to guide the development of the application (scenario that we call *proactive*), or, in the case of legacy applications built without UML, those obtained by means of reverse engineering [16] (*retroactive* scenario). Among the *UML diagrams* considered for data provenance to be extracted, we have focused on those that not only have a strong relation with provenance, but are also mostly used by *software designers* [17]: (i) Sequence Diagrams (SqDs) and State Machine Diagrams (SMDs), because they are widely used to represent the behaviour of a system (one of the main purposes of capturing provenance information); and (ii) Class Diagrams (CDs), since they are the most widely adopted formalisms for modelling the intentional structure of a software system (that is, low level aspects from objects' internal status, information not given by the considered behavioural diagrams). The consistency among such diagrams can be checked against a set of defined OCL [18] constraints (see details of the defined constraints in the Supplementary Material [13]). Taking such *UML diagrams* as input, UML2PROV is used (**Step 1** in Figure 2) to automatically obtain (i) the *PROV templates* for the concrete application, and (ii) the *BGM* responsible for generating sets of *bindings* during the application's execution. In both the retroactive and the proactive scenarios, the application is then enriched with the *BGM*. To do so, the *BGM* is non–intrusively integrated into the application, so no changes to its base code are required. Subsequently, while the application is running, the generated *BGM* comes into play generating the sets of *bindings* with values logged from the execution (**Step 2**). Finally, the *expansion algorithm* [3] (**Step 3**) takes the *PROV templates* and the sets of *bindings*, replaces the *templates'* variables by the concrete values from *bindings*, and generates the *PROV documents* for the executed application, ready to be used by the *provenance consumer*.

All in all, UML2PROV provides both an automatic design of provenance (i.e., *PROV templates*), and an automatic generation of the artifacts in charge of obtaining *bindings* (i.e., *BGM*), by taking as input the design of the application (*UML diagrams*). The generation of the final *PROV documents* is accomplished transparently to the *provenance consumer*, needing neither a provenance designer to create the *PROV templates*, nor a developer with provenance capabilities to include the required provenance instructions in the application. Additionally, this process makes the maintenance of the resulting provenance-aware systems a straightforward task, since every time the design of the application changes, both the provenance design and the artifacts for capturing provenance can be automatically updated accordingly.

4 From UML to PROV templates

UML2PROV supports a subset of UML diagrams modelling a system (SqDs, SMDs and CDs) and automatically transforms them into PROV templates specifying the design of the provenance to be generated for such a system. Since UML diagrams model a system (or an excerpt of it) from different perspectives (depending on the diagram used), the resultant templates will represent the provenance aspects associated with the perspective at hand. For instance, SqDs are used to model the interactions among collaborating objects and the exchange of information between them; thus, the templates generated from a SqD will focus on the flow of logic within the system. SMDs specify the behaviour of individual objects of a system; hence, the templates resulting from a SMD will comprise information regarding the evolution of the objects' state as modelled by the SMD. CDs model the static structure of a system by means of UML *classes* and their relationships, which classify sets of objects and specify the features that characterise the structure and behaviour of those objects. Thus, the templates obtained from CDs will represent low-level aspects, comprising not only the objects' characteristics at some point (we refer these characteristics as *status*), but also the operations that has led the objects' status to be as they are. From now on, we will use the term *state* and *status* as follows. In SMDs, in accordance to UML terminology, the *state* of an object denotes a situation during which some invariant conditions holds [9]. In CDs, to avoid confusion, we use the term object *status* with a broad scope, referring to the values of the object's attributes at some moment, which particularly could correspond to a concrete *state*. Although modeling different perspectives of a system, UML diagrams can have a rich semantic overlap. Similarly, the PROV templates generated from the UML diagrams of a system can also share elements which enable the merging of the associated PROV documents.

Our proposal relies on a set of transformation patterns that ultimately associate UML elements with PROV elements. These patterns are the result of a rigorous revision and thorough reformulation of those we presented in a semi-structured and non-systematic way in [11], [12]. Now, we have rigorously reformulated and extended them to cover the translation of more UML elements, ensuring that the overall set of patterns have consistent definitions. The new description facilitates their understandability and maintenance, enabling other researchers to replicate the presented functionality. More specifically, we have established a common structure for all patterns consisting of four blocks: *Context*, expressed in natural language, corresponds to a concrete situation of the system to be developed that is addressed by the UML representation identified in the pattern; *UML diagram*, which refers to a prototypical UML design whose translation into PROV is ruled by the pattern; *Mapping to PROV*, that corresponds to the PROV template proposed as translation for the corresponding *UML diagram* (or an excerpt of it); and *Discussion*, that presents issues related to the transformation at hand. Concretely, our overall approach identifies a total of 17 *contexts*, being 4 modelled by SqDs, 3 by SMDs, and 10 by CDs. We note that our intention is not covering all the UML elements that may appear in these types of diagrams, but focusing on those we have considered of interest for provenance purposes. For space reasons, we do not include in this section all patterns, but they can be consulted in the Supplementary Material [13]. For the sake of brevity, here we illustrate the transformation patterns with an example, slightly modified from [19], related to the enrolment and attendance of students to seminars that are held during a University course (from now on *University example*). Herein, the SqDs, SMDs, and CDs to PROV templates transformations are briefly explained in Sections 4.1-4.3, and illustrated using Figure 3. The top of each column in Figure 3 is an excerpt of a UML diagram (corresponding to the *UML diagram* block in the specification of the patterns in the Supplementary Material) that is associated with a specific context in a pattern (*Context* block). In this UML diagram, the name of the UML elements considered in the transformation pattern are in red courier font. The PROV template obtained after applying the corresponding pattern appears under each UML diagram (*Mapping to PROV* block). The correspondence between a UML element involved in the transformation and a PROV element in the PROV template, is denoted by means of a numeric identifier inside a green label ID in UML diagrams, and a purple label ID in PROV templates. As for the PROV templates, the elements of type `prov:Entity`, `prov:Activity`, and `prov:Agent` are identified with variables with non-context-dependent names.

4.1 UML Sequence Diagrams

SqDs are used to model the *interactions* among collaborating objects in terms of `Messages` exchanged over time for a specific purpose. Thus, SqDs transformation patterns focus on an `ExecutionSpecification`, representing an *operation execution*, started by a `Message`.

To give an insight into the SqDs transformation patterns, we explain the *Seq*uence Diagram *P*attern 2 (*SeqP2*). This pattern refers to a context illustrated in our University example by means of the excerpt of UML SqD depicted at the top of column "UML Sequence Diagram Pattern" in Figure 3. This diagram shows a sender component (`Lifeline` 1) interacting with a recipient component by

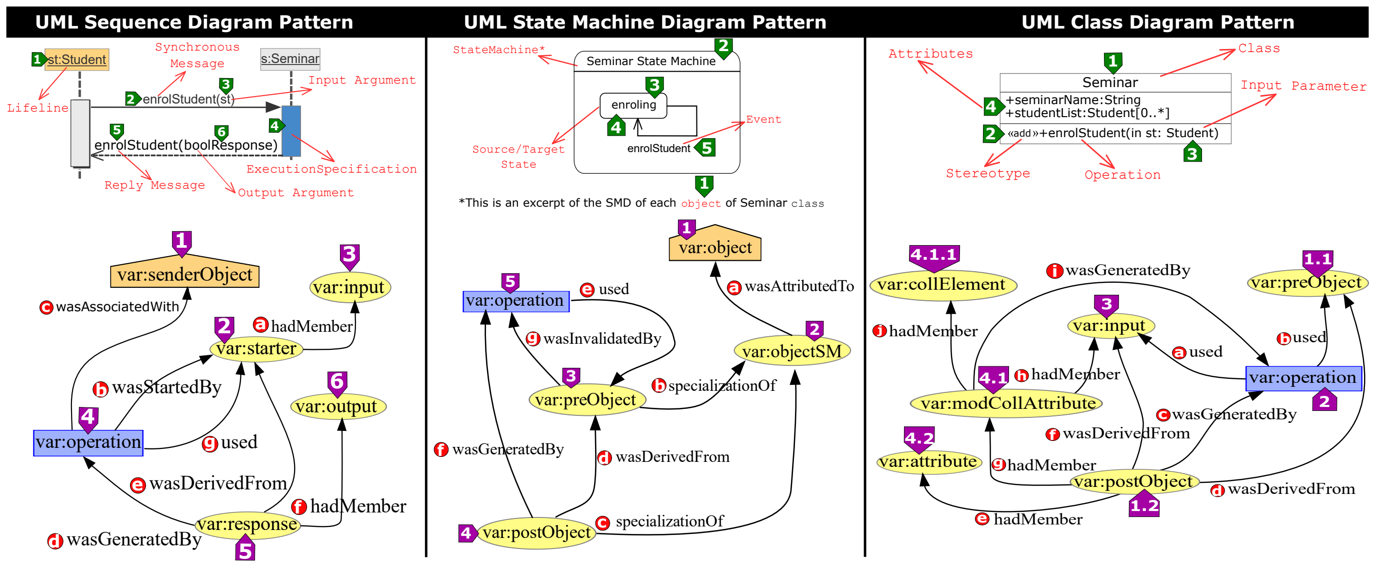


Fig. 3. Above, UML diagrams (SqDs, SMDs, and CDs) modelling the enrolment of a `Student` in a `Seminar` from different perspectives. Below, the template generated from the UML diagram depicted at the top.

calling an operation (`Synchronous Message` 2), which contains information to pass into the *operation execution* (`Input Arguments` 3). The call causes the recipient to execute the operation (`ExecutionSpecification` 4), which results in a response (`Reply Message` 5) to the sender that contains output information (`Output Arguments` 6). This UML diagram is translated into a PROV template as follows:

1. The sender `Lifeline` 1 is mapped to a `prov:Agent` identified by `var:senderObject` 1 which assumes the responsibility for starting the *operation execution*.
2. The `Synchronous Message` 2 that initiates the `ExecutionSpecification` 4 of the recipient is a `prov:Entity` identified as `var:starter` 2.
3. Each of the `Input Arguments` 3 is a separate `prov:Entity` identified as `var:input` 3. The relation ⓐ `prov:hadMember` states that `var:input` is an element of `var:starter`.
4. The `ExecutionSpecification` 4 is a `prov:Activity` identified as `var:operation` 4, which *used* and *was started by* `var:starter` (ⓖ `prov:used`, ⓑ `prov:wasStartedBy`). The assignment of responsibility to `var:senderObject` for `var:operation` is ⓒ `prov:wasAssociatedWith`.
5. The `Reply Message` 5 (response to the `Synchronous Message` 2) is a `prov:Entity` with identifier `var:response` 5. This `Reply Message` *was generated* (ⓓ `prov:wasGeneratedBy`) by `var:operation`, and *derived* (ⓔ `prov:wasDerivedFrom`) from `var:starter`.
6. Each argument of `Output Argument` 6 is a separate `prov:Entity` identified as `var:output` 6. The relation ⓕ `prov:hadMember` states that `var:output` is an element of `var:response`.

4.2 UML State Machine Diagrams

SMDs specify the discrete behaviour of individual elements of a system. They mainly consist of `States`, `Transitions` and other types of vertexes called `Pseudostates`. Taking this into account, the SMDs transformation patterns have been defined based on the `Transition`'s source and target elements (e.g., `States` or `Pseudostates`). To illustrate the patterns referring to SMDs, we have chosen the *St*ate Machine Diagram *P*attern 3 (*StP3*). *StP3* addresses a context illustrated in the University example by the excerpt of UML SMD depicted at the top of the middle column in Figure 3. This SMD (`StateMachine` 2) represents the fact that the triggering of the event (`Event` 5), consequence of an operation execution, causes an object (`Object` 2) to change from one source state (`State` 3) to a target state (`State` 4). The associated PROV template is defined as follows:

1. Each `Object` 1 of the `Class`, whose behaviour is defined by the `StateMachine` 2, is mapped to a `prov:Agent` identified by `var:object` 1, which bears the *object*'s responsibilities. This fact is shown by means of the relation ⓐ `prov:wasAttributedTo` between `var:objectSM` 2 and `var:object`.
2. The `StateMachine` 2 itself is a `prov:Entity` identified by `var:objectSM` 2.
3. The source `State` 3 is a `prov:Entity` identified by `var:preObject` 3, which is a *specialization of* (ⓑ `prov:specializationOf`) `var:objectSM`.
4. The target `State` 4 is a `prov:Entity` identified by `var:postObject` 4. It is also a *specialization of* (ⓒ `prov:specializationOf`) `var:objectSM`, *derived* (ⓓ `prov:wasDerivedFrom`) from `var:preObject`.
5. The `Event` 5 represents the occurrence of an operation execution. This execution is a `prov:Activity` identified as `var:operation` 5. This `prov:Activity`: *used* `var:preObject` (ⓔ `prov:used`); *generated* `var:postObject` (ⓕ `prov:wasGeneratedBy`); and *invalidated* `var:preObject` (ⓖ `prov:wasInvalidatedBy`).

4.3 UML Class Diagrams

Depending on their nature, operations have specific semantics which can also provide information of interest for provenance capture. More specifically, operations can be viewed from a number of different perspectives, being categorized by aspects such as how they access data (i.e., a method changes the object's status or leaves it constant) or their behavioral characteristics (i.e., creational, structural, or collaborational) [20]. We have stated a taxonomy of operations based on that given by Dragan et al. [20]. We have enriched it by providing additional categorization elements (for more details see the Supplementary Material [13]). The resulting taxonomy is expressed as a set of UML stereotypes (represented between guillemets, such as «add») which are linked to the operations to categorized them. Inspired by the semantics of the operations identified in our taxonomy, we have defined 10 transformation patterns that give concrete PROV templates depending on such semantics.

Below we give an insight into these transformation patterns, by explaining the *Class* Diagram *P*attern 10 (hereinafter *ClP10*). The excerpt of UML CD diagram appearing in the third column of Figure 3 includes a class (`Class` 1) taken from our University example. This class has an operation (`Operation` 2) stereotyped with «add», which means that its execution adds a new element, or elements, (`Input Parameters` 3) into a collection modelled as one of the object's `Attributes` 4. This causes a change in the object's status. The associated PROV template is defined as follows:

1. The object that changes its status is described in the CD by the `Class` 1, which is translated into:

   1.1. a `prov:Entity` identified as `var:preObject` 1.1 that is the object with the status before the execution of the operation.

   1.2. a `prov:Entity` identified as `var:postObject` 1.2 that is the object with the status after the execution of the operation. It *derived* from `var:preObject` (d `prov:wasDerivedFrom`).

2. The `Operation` 2, stereotyped with «add», is a `prov:Activity` identified by `var:operation` 2. This `prov:Activity`: *used* `var:preObject` (b `prov:used`) and *generated* `var:postObject` (c `prov:wasGeneratedBy`).
3. Each parameter of `Input Parameters` 3 is a separate `prov:Entity` identified as `var:input` 3. It was *used* by `var:operation` (a `prov:used`), and `var:postObject` *was derived* from it (f `prov:wasDerivedFrom`).
4. The `Attributes` 4 are separated into:

   4.1. the modified collection attribute, which is translated into the `prov:Entity` identified by `var:modCollAttribute` 4.1. It *was generated* by `var:operation` (i `prov:wasGeneratedBy`), and is a *member* of `var:postObject` (g `prov:hadMember`). Also, the elements belonging to the collection `var:modCollAttribute` consist of the new member(s) (`var:input` 3), and the members preceding the operation execution (`var:collElement` 4.1.1). To state that they are members of `var:modCollAttribute`, relations h `prov:hadMember` and j `prov:hadMember` link them, respectively.

   4.2. the attributes not modified by the execution of the operation, which are mapped to a `prov:Entity` identified with `var:attribute` 4.2. To express that `var:attribute` is an element of `var:postObject`, the relation e `prov:hadMember` links them.

```
{"var" : {
   "starter" :          [{"@id":"ex:message_1"}],
   "preObject":         [{"@id":"ex:seminar1_1 "}],
   "senderObject":      [{"@id":"ex:student1 "}],
   "output":            [{"@id":"ex:boolean_1"}],
   "object":            [{"@id":"ex:seminar1 "} ],
   "operation":         [{"@id":"ex:enrolStudent1"}],
   "objectSM":          [{"@id":"ex:seminar1_0"}],
   "response":          [{"@id":"ex:message_2 "}],
   "postObject":        [{"@id":"ex:seminar1_2"}],
   "input":             [{"@id":"ex:student1_1"}],
   "attribute":         [{"@id":"ex:seminar1_2_seminarName"}],
   "modCollAttribute": [{"@id":"ex:seminar1_2_studentList2" }]},
   "vargen":{},
   "context":{"ex" : "http://example.org/"}
  }
```

Fig. 4. A set of bindings serialized in JSON for the templates in Figure 3.

5 Bindings Generation Requirements

The capture of provenance data during the execution of an application involves providing it with additional instructions for the generation of bindings. Aimed at making this task transparent to *software developers*, UML2PROV provides each application with a *Bindings Generation Module* or *BGM* (see Figure 2), so that the application becomes provenance-aware. The disparate nature of applications prevents us from establishing one and only one module for all applications. Instead, we have stated a set of requirements for establishing the characteristics such a module must satisfy, independently of both the concrete nature of the application and technologies used to develop it. These requirements are organized into the following categories:

Manual vs. automatic. While manually adapting the source code of an application to generate bindings could be a valid option, it constitutes a tedious, time-consuming and error-prone process. *Software developers* would have to work hard on traversing the application's source code, carefully analysing it to add suitable instructions to generate the bindings structures. In addition, manual code adaptation negatively affects the maintainability of the application, since changes in the application code may affect the provenance-specific code. Therefore, *requirement 1 (R1)* states that the adaptation of the application to include provenance capabilities must be carried out automatically.

Intrusive vs non-intrusive. Making an application provenance-aware could result in provenance capture code scattered across all its source code, fact that would make the application maintenance a cumbersome task. *Requirement 2 (R2)* states that 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. *Requirement 3 (R3)* states that such a module not only has to contain the instructions for bindings generation, but it also has to identify the specific moments within the application's source code where such instructions must be executed.

Consistency. To ensure the generation of well-defined PROV documents after expansion, consistency between the generated PROV templates and the bindings must be guaranteed. In our case, this consistency encompasses two main aspects. First, *requirement 4 (R4)* claims that each binding obtained from an application's execution must be associated with at least one PROV template generated from the UML diagrams. Second, *requirement 5 (R5)* states that the variables included in a set of bindings must correspond with the variables in their associated PROV templates. For example, the set of bindings depicted in Figure 4 satisfies *R4* and *R5*: it has been generated during the execution of the *operation* `enrolStudent` of our example, and it links each variable of the templates in Figure 3 with the corresponding value collected during such an execution.

6 Implementation

In this section, we discuss a reference implementation that generates the two main key elements of the UML2PROV architecture: *PROV templates* and *BGM* (see Figure 5).

Given the disparate nature of applications, the manual creation of both elements is not the best option. Thus, we have followed Model Driven Development (MDD), which focuses on models, conforming metamodels, rather than on computer programs [21]. Following MDD, we automatically generate *PROV Templates* (text files in PROV-N [22]) and the *BGM* (code) using a refinement process from the *UML diagrams* modeling the system.

As for the elements involved in the generation of PROV templates (light-grey elements in Figure 5), our implementation adopts an MDD-based tool-chain comprising two transformations. The first takes as source the *UML diagram models*, conforming to the UML metamodel [9], and generates the corresponding *template models*, conforming the PROV metamodel [15]. This model–to–model (M2M) transformation has been carried out by means of the ATL Eclipse plugin [23] (see the Supplementary Material [13] for more details). The second transformation takes such *template models* as source, and generates the final *PROV templates* expressed in PROV-N [22]. This model–to–text (M2T) transformation relies on a set of one–to–one transformations implemented in Xtend [24], so that each element of the PROV metamodel is linked with an element of the PROV-N text representation.

For the reference implementation of the *BGM*, we have chosen the Aspect Oriented Programming paradigm (AOP) [25] which allows us to implement the *BGM* according to the requirements stated in Section 5. AOP aims at improving the modularity of software systems, by capturing inherently scattered functionality, often called *cross-cutting concerns* (e.g., data provenance capture can be considered as a cross-cutting concern). Concretely, we have used Eclipse AspectJ [26], an AOP extension created for Java. Regarding the process to automatically generate the *BGM*'s source code (dark-grey elements in Figure 5), it is made up of a single M2T transformation, implemented in Xtend, taking as source the *UML diagram models*, and generating the *BGM*. The *BGM* encompasses the AspectJ code (the *context–dependent component*) and Java libraries supporting this component's code for bindings generation (the *context–independent component*).

The resulting *BGM* is event-based. Hence, the generation of bindings is decoupled from the management of the corresponding provenance data regarding, for example, volume and data storage or distribution. Thus, 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. UML2PROV is agnostic about when to compute the final PROV documents (*lazy* or *eager* approaches [2]). UML2PROV facilitates decision making on these aspects.

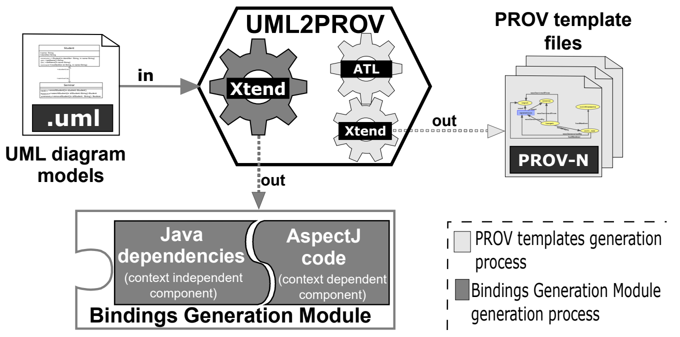

UML2PROV
PROV template files
.uml
in
Xtend
ATL
Xtend
out
PROV-N
UML diagram models
out
Java dependencies
(context independent component)
AspectJ code
(context dependent component)
Bindings Generation Module
PROV templates generation process
Bindings Generation Module generation process

Fig. 5. MDD-based implementation proposal.

The *BGM* recognises the occurrence of certain circumstances during the execution of the instrumented application (start and end of operations, new bindings), and consequently triggers events. These events must be captured by classes that implement the `BGMEventListener` interface. There, UML2PROV users program their preferred storage or distribution approaches, persistence system (or systems), format, and so on. These listeners are reusable among all applications sharing the same bindings management policy.

UML2PROV generates the *BGM* as a Java library (*jar* file). Integrating it into the application is as easy as using the AspectJ compiler to weave the *BGM* with the original application, thus obtaining the instrumented application ready to produce provenance data. For more information, see the Supplementary Material [13] and the UML2PROV User Guide [14].

7 Evaluation

As with any other computational functionality, the provenance capture has associated temporal and spatial costs [5]. The purpose of this section is to show UML2PROV benefits and costs to software engineers seeking to make their applications provenance-aware.

Since UML diagrams drive both the design and capture of provenance, the intuition is that the systems' diagrams themselves have implications on the generation of provenance. For reasons that will be clear later on, taking a detailed UML design as source for UML2PROV may lead to capture more provenance than required to satisfy certain provenance requirements, resulting in unnecessary overheads. Conversely, if the UML design is tailored to address certain provenance requirements, only relevant provenance will be collected, thus reducing the overheads. However, adapting the UML design to precisely fit provenance requirements demands additional effort from the designer. The more effort is devoted to tailoring the UML diagrams to selectively expose the relevant information for certain provenance requirements, the less provenance collected and the less overhead in time and storage. Although other authors have previously applied UML2PROV (e.g., in the field of astronomical workflows [27]), to show the implications of the UML design taken as source for UML2PROV, and the trade-offs of using it, we will apply our approach both to a legacy application built without UML (*retroactive scenario*), and to an application built from a UML design (*proactive scenario*). The size of both applications is intentionally disparate, which allows us to study the implications of this

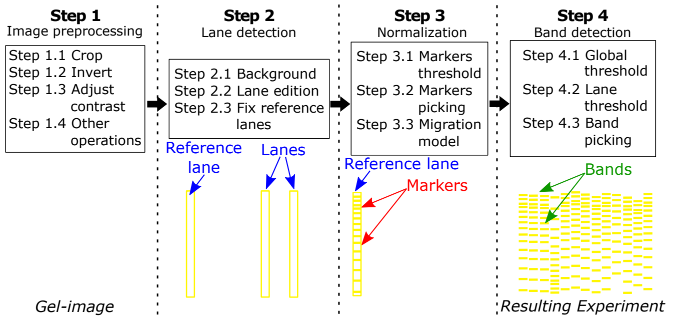


Fig. 6. Workflow of GelJ's experiment wizard.

variable when applying UML2PROV. Our reference implementation will serve as UML2PROV tool for the evaluation.

7.1 Case studies: GelJ and the University example

For the retroactive case study we chose GelJ [28], a bioinformatics application used for analysing DNA fingerprint gel-images (hereinafter gel-images), and comparing DNA patterns; this allows the analysis of the genomic relatedness among different samples, as well as their classification. The analysis of DNA patterns has applications in medical diagnosis, parentage testing, food industry, and many others [29]. GelJ was chosen for several reasons. First, GelJ has a community of users with needs for provenance for their DNA analysis. Second, we have direct contact both with its developers, who can provide us with useful background about the tool, and with its potential users, who can assist us to assess the quality of the provenance generated by UML2PROV against their real needs. Lastly, GelJ lacks a UML design, so it must be obtained, for example, automatically by reverse engineering. Thus, using a retroactive scenario in our evaluation we are considering the most complex case of application of UML2PROV, illustrating the effort that it may require, and offering a realistic vision of what applying UML2PROV can entail for applications not initially designed with UML.

The proactive case study is based on the academic application used to illustrate Section 4. Space limitations prevent us from including all its details in the main paper (they are detailed in the Supplementary Material [13]).

GelJ. Briefly speaking, GelJ has an *experiment wizard* that, after choosing a gel-image as source, guides the user through a four steps process (Figure 6). In Step 1, the user may preform actions to increase the source gel-image quality. In Step 2, GelJ automatically detects the lanes of the gel-image; then, the user might have to perform some adjustments over the lanes such as adding new ones, removing or editing.Among all the detected lanes the user selects one or several as reference. In Step 3, GelJ automatically determines markers in the reference lane; the user might have to handle these markers by adding or removing some of them. In Step 4, GelJ automatically detects bands inside the detected lanes; the user might modify these bands later (e.g., adding, moving, and removing). The result of this workflow is an *experiment* consisting of the source gel-image together with a set of detected bands. It is remarkable that GelJ's database only persists concrete characteristics pertaining to an experiment (e.g., its name, the user who performed it, and biological information like genus, species. . .). It does not store aspects such as the steps followed during the experiment creation, or its origin (an experiment could be created from scratch, duplicated or imported). Hence, scientists usually have to ask other colleagues about the process followed to create an experiment, its authorship, or its origin.

In order to give an unbiased evaluation of our approach, based on a representative use of GelJ, we analyse the successive actions of a user to generate 10 different experiments, each one for a different source gel-image. While the user worked, we wrote down the performed interactions with GelJ for creating those experiments. Among the 10 collected traces, we selected the one with the higher number and diversity of interactions to be used in obtaining the data for the performance evaluation. The selected experiment comprises 114 interactions among the 13 substeps of the GelJ experiment wizard, which involve the execution of about 46,000 operations in the source code (see the chosen sequence of interactions in the Supplementary Material [13]).

7.2 Objectives and Application Modes

Our evaluation should answer questions about the tradeoffs of using UML2PROV. How complex is the generation of the provenance design? How does UML2PROV affect the provenance instrumentation of the application? To what extent the maintenance of these two previous aspects is facilitated? What is the impact in the provenance storage needs and run-time overhead of the application? Does the collected provenance meet user expectations?

To evaluate these aspects we have devised three *Application Modes* for UML2PROV that provide us with a basis for benchmarking our approach. These Modes are illustrative of a range of ways of applying UML2PROV, though they are not the only possible, nor are they in any way strict UML2PROV methodologies. Each Mode is characterized by a different required effort to tailor the initial application's UML design before using the UML2PROV tool. Thus, Mode 1, by tailoring the UML design before applying the UML2PROV tool, generates provenance more adjusted to the users needs. Conversely, Modes 2 and 3 correspond to situations in which the UML2PROV tool is directly applied to UML diagrams without prior preprocessing, which leads to capture more provenance than in Mode 1. We would like to note that, in the end, our goal is not to conduct an analysis of the bounds of UML2PROV, but to draw conclusions on whether those different *Application Modes* result in benefits in terms of quantity and quality of the generated provenance.

Below, a description of each *Application Mode* is provided, including: 1) *aim* for which the Mode is proposed; 2) *considered UML design*, i.e., the UML diagrams that will feed the UML2PROV tool, and, depending on the Mode, how they are tailored to generate provenance more adjusted to users needs; and 3) *required effort*, in which we sum up the required effort to perform the tasks comprising the *Application Mode*. Note that assessing the level of effort, and the time required to perform certain tasks, is difficult, and often imprecise, because it closely depends on the software engineer's skills. This is why we do not provide data about the duration of the tasks. However, it is possible to give an

TABLE 1
Overview of tasks in UML2PROV *Application Modes*.

	Case study - GelJ			Case study - University		
Task	**App. Mode 1**	**App. Mode 2**	**App. Mode 3**	**App. Mode 1**	**App. Mode 2**	**App. Mode 3**
T1. Identify the complete CD	⏩	⏩	⏩	–	–	–
T2. Identify provenance requirements	🧍	–	–	🧍	–	–
T3. Identify classes/operations involved in provenance requirements	🧍	–	–	🧍	–	–
T4. Discard not identified classes/operations	🧍	–	–	🧍	–	–
T5. Add stereotypes to selected operations	🧍	–	–	🧍	–	–
T6. Identify SqDs	🧍 / ⏩	⏩	–	🧍	–	–
T7. Design SMDs of selected classes	🧍	–	–	–	–	–

⏩ Permorfed automatically | 🧍 Requires manual effort | 🧍/⏩ Requires semi-manual effort | – Non executed task

insight by identifying those tasks performed manually or automatically. The more manual tasks, the greater the effort. Table 1 depicts the set of tasks comprising each *Application Mode*, so it can be used to compare their characteristics.

Application Mode 1

Aim. Obtain a more adjusted provenance, aligned with certain requirements, by tailoring the initial UML design. To state these requirements the first phase of PrIMe [8] served as inspiration. In the first case study, we involved two GelJ users, asking them for typical questions they were seeking to answer, and which the current system cannot answer (task T2 in Table 1). Among these questions, we excluded those that can be answered by using information stored in the GelJ database; then, we refactored them in terms of PROV. The resulting questions, shown in Table 2, reflect the provenance requirements (called *provenance use case questions* in PrIMe). The provenance questions for the University example were adaptations of those appearing in the First Provenance Challenge [30] (see Supplementary Material [13]).

Considered UML design. Mode 1 is characterised by generating provenance with information about the perspectives covered by the supported types of UML diagrams (SqD, CD, SMD). As already mentioned, the retroactive case study did not have an initial UML design, so the UML design tailoring tasks were interleaved with additional reverse engineering tasks to identify that UML design. Thus, using static reverse engineering techniques [31] with Papyrus [32], the complete GelJ's CD was automatically obtained (task T1). Then, inspired by the second phase of PrIMe, in collaboration with GelJ's developers, the classes and operations (called *actors* in PrIMe) involved in answering the above questions were identified (T3), following this procedure: for each class included in the CD, the developers checked if the class was involved in answering any of the identified questions; in case of doubt, it was selected. Then, for each operation of the selected classes, the developers confirmed if it was involved or not in the identified questions; again, when in doubt, it was selected. The result of tailoring task T3 was the identification of 17 classes and 66 operations out of 279 classes and 1688 operations that compose GelJ (i.e., in this Mode, ~6% of the classes and ~3.9% of the operations of GelJ were used; the rest were discarded, T4). These classes are shown in column "Identified classes" of Table 2, where the names `StepN_M` refer to steps (`N`) and substeps (`M`) of the GelJ's wizard. Also, to obtain more meaningful provenance, we assigned stereotypes to the UML operations in the GelJ's CD (T5). ObjectAid [33] was used to identify SqDs with the interactions between the objects of the selected classes (T6). Each selected operation was statically reversed engineered, obtaining a SqD representing the interactions (including UML lifelines, messages, etc.) where the operation was involved. This process resulted in a set of 76 UML messages. Regarding SMDs, to the best of our knowledge, the few existing automatic approaches [34] are not able to extract high-level information not present in the code and only known by the designer (e.g. the representative names for the states). Therefore, the designers of GelJ were involved in obtaining

TABLE 2
Questions identified from Q1 to Q9 raised by GelJ users, together with GelJ classes involved in answering those questions.

ID	Provenance requirements	Identified classes
Q1	What is the origin of an experiment? (from scratch, duplicated or imported)	*Experiment* *Step4_3* *Image_Assistant*
Q2	What is the set of activities that has led an experiment as it is?	*Step1_1-Step1_4* *Step2_1-Step2_3* *Step3_1-Step3_3* *Step4_1-Step4_3*
Q3	Which background (dark or light) has been used during an experiment construction?	*Step1_1-Step1_4* *Step2_1-Step2_3* *Step3_1-Step3_3* *Step4_1-Step4_3*
Q4	Who is the user who has carried out a specific step of an experiment wizard?	*Step1_1-Step1_4* *Step2_1-Step2_3* *Step3_1-Step3_3* *Step4_1-Step4_3*
Q5	How many lanes have been added/removed during an experiment's generation process?	*Step2_2*
Q6	What is the height-threshold used for band detection during the experiment's generation process?	*Step4_1* *Image_Assistant*
Q7	How many bands have been added/removed during an experiment's generation process?	*Step4_3*
Q8	What is the detailed information regarding the pre-processing activities (source/target states, nested activities called)?	*Step1_1-Step1_4*
Q9	What is the time-cost of creating a new experiment?	*Menu* *Main* *Step1_1-Step1_4* *Step2_1-Step2_3* *Step3_1-Step3_3* *Step4_1-Step4_3*

the SMDs (T7). They designed a SMD for each class whose states are related to the provenance requirements. This led to a set of 13 SMDs with 33 states and 68 transitions, modelling the behaviour of the classes.

The proactive case study did not require reverse engineering (all tasks are tailoring tasks). All the UML SqD, CD and SMD diagrams were available at the beginning of the evaluation, so tasks T1 and T7 were not performed (T6 now deals with selecting the SqDs with the interactions between the objects of the identified classes). As explained in the Supplementary Material [13], after tasks T2-T5, ~27% of classes and ~29% of operations were used. This higher percentage with respect to the GelJ case is due to the smaller size of the University example. With few classes, the chances of a class/operation participating in answering a provenance question increases, so less classes/operations may be eliminated in the UML design tailoring.

Required Effort. The greatest effort attributable to this Mode corresponds to the identification of the provenance requirements (T2) and the selection of the involved classes and operations (T3 and T4). To perform these tasks, two users and the two designers of GelJ were consulted. Whilst the SMDs were handcrafted (T7), the SqDs were obtained via a semi-automatic process (T6) consisting in manually iterating over the selected operations from the CD, and generating for each of them its SqDs using automatic reverse engineering. In contrast, the CD was mostly automatically generated (T1), except for the stereotyping task (T5), performed manually based on the operations' names, and consulting the developers when necessary.

The effort required for the second case is substantially the same: all automatic tasks regarding the reverse engineering are not necessary, but manual tasks related to the provenance requirements, the identification of SqDs, and the CD tailoring remain.

Application Mode 2

Aim. Generate provenance as directly and automatically as possible. UML diagrams were not tailored, so no requirements were used.

Considered UML design. This Mode involves considering only the perspectives covered by CDs and SqDs, using these diagrams directly, without tailoring them. Consequently, the 100% of the classes and operations were taken into account in both case studies. In the end, this led to collecting more provenance data than the user needed. Neither SMDs nor the stereotyping of class operations are considered, so that the consequences of their exclusion in the evaluation can be checked. For GelJ, all the 279 classes and 1688 operations obtained in task T1 were considered. Then, applying dynamic reverse engineering techniques with MaintainJ [35] to every class (considering all its operations), a set of SqDs with 133 messages were determined (T6). In the second case study, a CD with 11 classes with 37 operations, and SqDs with 25 messages were considered.

Required effort. The effort required here was considerably reduced: since there was no tailoring of the UML, there was no need for manual tasks (T2-T5); task T6 was fully automated; and task T7 was excluded. Only automatic reverse engineering tasks were performed (T1 and T6). In fact, for the second case study, no effort at all was needed (the UML design was processed directly with the UML2PROV tool).

Application Mode 3

Aim. Generate provenance as automatically as possible. Use only the UML diagrams most commonly used. UML diagrams were not tailored, so no requirements were used.

Considered UML design. The only difference from the previous Mode was that here only the perspective covered by CDs was considered. This Mode represents many real situations, since CDs are the most widely used UML diagrams [10] (even applications are often developed with the sole help of such diagrams) and, if not available, CDs are the most easily obtainable with reverse engineering.

Effort required. Only the automatic reverse engineering of the CD (T1) was necessary, and solely for the GelJ case study.

7.3 Evaluation platform

The evaluation was ran on a personal computer, Intel(R) Core(TM) i7 CPU, 2.8GHz, with Oracle JDK1.8 and a Windows 10 Enterprise OS. The used `BGMEventListener` performs an asynchronous recording, in which bindings are accumulated in memory before being shipped, as sets of bindings, to a MongoDB database [36] after finishing the execution of each operation (bulk submission).

7.4 Analysis

7.4.1 Aspect 1: Generation of the provenance design

Prior to UML2PROV, software engineers had to manually develop the PROV templates corresponding to the design of the provenance to be generated. This cumbersome, time-consuming and error-prone task was facilitated by the assistance of PROV experts working closely with the application developer or designer, to reflect the application's functionality in the templates [3]. To make matters worse, this procedure does not scale up when the amount of provenance to be designed increases. UML2PROV makes the design of provenance straightforward, while providing significant benefits for the software engineer. The automatic generation of the templates avoids human intervention, thus preventing any kind of human mistake. In addition, the generation time for the templates, a few milliseconds per template, may be considered negligible compared to the hours, even days, needed to create them manually (we omit this data in Table 3 because we consider it irrelevant). Regarding the influence of the *Application Mode* followed to tailor the UML design, it is no wonder that results collected in Table 3 show that the closer the UML design fits the application provenance requirements, the less templates are generated, and the smaller their total size and the faster their generation. Table 3 shows that *Mode* 1 (more adjusted design, with less UML elements, but requiring more effort to obtain) presents the lowest number of templates: 198 for GelJ (25 for the University case study). In contrast, Modes 2 and 3 (less precise designs, with more elements, and automatically generated), result in a higher number of templates, 1,821 (63) and 1,688 (40), respectively. These results demonstrate that a greater initial effort to more accurately tailor the UML according to a set of provenance requirements, as

TABLE 3
Variables evaluated for the considered UML2PROV *Application Modes* in both case studies.

	Application Mode (UML diagrams)	No. templates	Total size of templates	No. Variables	No. Sets of bindings	Sets of bindings size (MB)	Expanded templates size (MB)	No. executions instrumented operations	% instrumented executed operations	Execution time (ms)	Time overhead
GelJ	**Mode 1** (SqD, SMD, CD)	198	0.3MB	954	159	2.3	3.6	159	0.34%	9,484.72	1.26%
	Mode 2 (SqD, CD)	1,821	1.48MB	5,980	46,526	64.9	109.3	46,526	100%	14,522.66	55.05%
	Mode 3 (CD)	1,688	1.3MB	5,236	45,329	57.2	57.6	45,329	100%	13,904.89	48.45%
University	**Mode 1** (SqD, SMD, CD)	25	20KB	115	687	1.4	2.3	687	41.79%	656.19	25.92%
	Mode 2 (SqD, CD)	63	47KB	269	1,644	2.0	3.3	1,644	100%	802.29	53.96%
	Mode 3 (CD)	40	26KB	137	1,644	1.9	2.0	1,644	100%	747.31	43.41%

made in *Mode* 1, results in fewer templates. This positively affects the instrumented application, and the performance of the provenance capture. However, it has little impact in templates size (Table 3), since the differences between the Modes (few kilo bytes) are considered negligible.

7.4.2 Aspect 2: Instrumentation of the application

Once the templates are generated, developers have to instrument the application to collect *bindings* conforming to these templates. As described in Section 5, UML2PROV provides each application with an automatically generated BGM, which must be integrated with the application.

However, the BGM changes depending on the UML design used to generate it. As concluded from Section 7.4.1, the more UML elements conforming the UML design, the more templates are generated (and variables in these templates). Since assigning a value to a variable in a template usually requires the addition of one instruction in the code, the intuition is that the number of instructions included in the BGM grows as the number of UML elements does. The column "No. variables" of Table 3 corresponds to the number of instructions for bindings generation contained in the BGM. The strategy with least UML elements (*Application Mode 1*) leads to the BGM with fewest instructions (954 for GelJ), versus Modes 2 and 3 that generate BGMs with more instructions (5,980 and 5,236, respectively). Results are consistent with the University case (Table 3), though the difference between Modes 2 and 3 is more pronounced because SqDs templates, defined using more variables [13], have a greater relative weight in this case. Again, the effort devoted to more precisely tailoring the UML design according to the provenance requirements results in a simplification of the BGM, which has significant implications in the performance (Section 9). Another result of the tailoring process of *Mode* 1 is the reduction of the number of instrumented operations of the application with respect to the rest of Modes. When GelJ was executed, *Mode* 1 led us to collect 159 executions of instrumented operations (as many as sets of bindings), i.e. 0.34% of the total of operation executions. For the second case study, a smaller application, the tailoring process did not reject so many operations from the initial design, so the percentage of instrumented operations was higher, a 41.79% of the total of operation executions.

7.4.3 Aspect 3: Maintenance of provenance capabilities

To ensure that the generated provenance always describes what the application actually does, it is necessary to study if the PROV templates and the instrumented code have to be updated when the application is redesigned. Moreau et al. [3] deal with some types of template changes that are likely to occur in practice, such as *rename/add/remove template* and *add/drop variables or relations*, analysing their consequences in the bindings. As stated in that work, whilst the bindings remain correct in most cases, some modifications to templates result in a partially generated provenance, or even in errors. UML2PROV entails huge benefits for the compatibility and synchronisation between the UML design and the application's provenance facets. Every time the UML design is updated, the templates and the BGM can be regenerated, which guarantees the immediate and automatic redesign of the application provenance facets.

7.4.4 Aspect 4: Storage and Run-time overhead

As the capture of provenance is an additional consideration to the primary function of the system, it is desirable to minimise its influence on the application execution.

The overhead attributable to provenance capture is related to the number of executions of instrumented operations (see Table 3, where the column "No. Sets of bindings" matches this number). The higher the percentage of instrumented executed operations with respect to the total of executed operations, the greater the overhead. Once again, we note that the effort devoted to tailoring the UML design focused on specific provenance requirements resulted in an overhead reduction. Thus, for GelJ, *Application Mode 1* yielded the best results, with the least number of sets of bindings (159), and the least run-time overhead (1.26%) and storage needs (2.3MB). This may be considered a low overhead, which is aligned with similar provenance instrumentation made by works such as [37], [38]. Modes 2 and 3 generated more sets of bindings (46,526 and 45,329, respectively), with a higher time overhead (55.05% and 48.45%) and storage (64.9MB and 57.2MB). These results even constitute an improvement over other proposals where it is common to observe overheads of up to 75% [39]. Note that *Mode* 2 got the worst results because it did not discard any class or SqD from the UML design, resulting in the capture of provenance for all the classes and operations appearing in those diagrams, which is much more than the necessary to meet the application provenance requirements. In the University case, Modes 2 and 3 behaved similarly to those of GelJ. This result was not surprising, since in both cases, in these Modes, 100% of executed operations were instrumented. However, for *Mode* 1, the run-time overhead (25.92%) was higher than for GelJ. The greater percentage of instrumented executed operations with respect to the total of executed operations commented in Section 7.4.2, 41.79% vs. 0.34%, explains this discrepancy.

Finally, Table 3 (columns "Sets of bindings size" vs "Expanded templates size") shows how the use of PROV-Templates reduced the storage requirements for the sets of bindings compared to the expanded templates.

7.4.5 Aspect 5: Quality of provenance

Throughout this section we will focus on GelJ; the analysis of the provenance generated from the University example is given in the Supplementary Material. Since the different tailoring strategies followed produce UML designs that expose different levels of detail about the application, a different provenance is generated in each case. How does that level of detail affect the quality of the obtained provenance? To analyse this aspect, we study if the collected provenance can answer *completely* (C), *sufficiently* (S), *partially* (P), or it cannot answer (N) the questions in Table 2. We will say that a question has been *completely* answered when GelJ users indicated that the answer was more detailed than what they expected. When the user indicated that the level of detail was enough, we will say that the question has been *sufficiently* answered. Additionally, in those cases in which the answer did not satisfy the user, we will talk about questions *partially* answered. Table 4 summarises our conclusions, also showing, in those cases in which the provenance can give an answer, the number of elements (`prov:Entity`, `prov:Activity`, `prov:Agent`) involved in such an answer. This information has helped us identify three kinds of implications the used *Application Mode*, and how it tailors the UML design, may have on the ability to produce provenance answers: *no effect*, when the *Mode* does not affect the results; *more detailed information*, when the retrieved answer gives different level of detail depending on the *Mode*; and *crucial*, when concrete information included in the UML diagrams is crucial for responding the question.

No effect. The answer to Q3 relies upon the value of an attribute belonging to classes `StepN_M`, whereas answers to Q5 and Q7 are based on identifying the execution of certain operations located in classes `Step2_2` and `Step4_3`, respectively. Provenance from *Mode* 1 answers such questions because the mentioned classes and operations needed to answer them were identified in the tailoring process carried out in that *Mode*. Provenance from Modes 2 and 3 can also answer the questions because these Modes include all the classes and operations of GelJ in their class diagrams. Thus, we can conclude that the *Application Mode* used does not influence the ability to answer questions Q3, Q5, and Q7.

More detailed information. An adjusted UML design may lead to the generation of more provenance, obtaining more detailed (i.e., *complete*) answers. E.g., the information refering to nested operations calls is crucial to answer Q1. Since nested operations are specially modeled by SqDs, *Mode* 2, which has the most detailed SqDs, *completely* answers Q1. The answers to Q2 and Q9 rely upon the operations identified in classes `StepN_M`; therefore, Modes 2 and 3, which identify all the operations in classes `StepN_M`, answer *completely*, whereas *Mode* 1 only provides information about the operations of classes `StepN_M` identified in the tailoring process. Although, a priori, giving *complete* answers seems to be a valuable fact, it is important to consider the extra time and storage that this implies (see Section 7.4.4). Considering that *Application Mode 1* has been designed precisely to expose only the information necessary for answering the questions, the additional provenance elements in Modes 2 and 3 could be considered unnecessary, instead of a good characteristic of the result.

TABLE 4
How questions Q1-Q9 of Table 2 are responded: completely (C), sufficiently (S), partially (P) or it cannot be answered (N) (in parenthesis, number of provenance elements involved in the response).

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Mode 1	S(5)	S(206)	S(3)	S(152)	S(12)	S(3)	S(10)	S(198)	S(234)
Mode 2	C(7)	C(386)	S(3)	P(35)	S(12)	N	S(10)	P(155)	C(483)
Mode 3	N	C(386)	S(3)	P(35)	S(12)	N	S(10)	N	C(489)

Crucial. Certain aspects of an application's behavior are modeled only by certain types of UML diagrams, even by certain elements of those diagrams. E.g., the answer to question Q6 relies upon information provided by SMDs. As *Application Mode 1* is the only one that considers SMDs, it is the only *Mode* that can help answer such a question. Similarly, to answer Q8, we need information provided by SqDs and SMDs, a fact that explains why *Mode* 2, which has SqDs but lacks SMDs, can only partially respond, and why *Mode* 3, which lacks both SqDs and SMDs, cannot even give an answer. Another example is that of the operations' stereotypes, which explain the nuances of class operations, thus helping to capture a more meaningful provenance (see Section 4.3). These nuances are crucial to answer Q4. Therefore, *Mode* 1, which is the only one that has stereotyped operations, can completely respond to Q4, while Modes 2 and 3 only partially respond to it. These examples delve into the arguments in favour of the convenience of making a prior investment of time to conduct a detailed fine-tunning of the application's UML design, guided by a set of provenance requirements.

Taking into account these results, next we provide some insight into what type of provenance questions can be answered with what type of diagram.

8 Discussion

UML2PROV is based on *transformation patterns* that cover a wide range of situations and addresses different provenance perspectives by generating PROV templates from different types of UML diagrams. Concretely, SqDs patterns generate PROV templates focused on the flow of logic within the system, SMDs patterns result in PROV templates comprising information about the evolution of objects' state, and CDs patterns produce PROV templates with information about objects' status at some point, as well as the behaviour that led to those objects' status. As shown above, considering such types of diagrams covering different perspectives of a system allows software designers to collect different provenance information, which can give answer to questions related to different aspects. For example, SqDs could serve to answer questions regarding: (1) the entity participants involved in interactions as well as their responsibilities in the execution of operations, (2) the executor of operations, and who called such operations, (3) messages' arguments, representing the data that are exchanged between the collaborating objects (such as in/out or return), and (4) the interaction process itself, including nested operations. Provenance data registered by using SMDs can answer questions regarding: (1) states an entity can go through during its lifetime in response to operations' executions, (2) operations' executions that trigger changes in entities' state. Finally, CDs could serve to answer questions regarding: (1) entities' status (i.e., values of the attributes) before and after operations' executions, and (2) entities' internal changes, i.e. the specific behaviour that can be triggered on individual entities (operations' executions), distinguishing among their different nature as identified by our stereotypes.

We also want to point out that one of the aspects our proposal benefits from is given by the use of the PROV-Template approach [3]: the reduction of both spatial overhead with respect to the expanded templates, and the development and maintenance effort, by separating responsibilities between software and provenance designers.

9 Threats to Validity

Assumptions made by our proposal. First, our overall proposal is based on UML, so a possible threat to validity could be related to whether it is used, even used enough, in industry. Several studies based on personal opinions coincide in remarking that UML may be considered complex. That is why there has been a tentative to find the "essential UML", analysing which types of UML diagrams are the most used in practice, concluding that Class, Sequence and State Machine diagrams are among the most frequently used UML diagram types [10] (which particularly support our decision on choosing such type of diagrams). Several surveys had investigated the adoption of UML in the software development community, showing empirical evidence (e.g., 85% of the survey participants [10] or 74% of the surveyed projects [40] use UML as modeling language). Second, UML2PROV requires that any element about which provenance is to be captured is present in the UML design. In this line, it is worth noting that PrIME [8] remarks that to address some provenance requirements, it is needed to adapt the application in order to surface some data. Concretely, this situation could occur when (1) a data item may not be part of the application model (i.e., the UML design, as we discussed early); (2) part of the desired provenance may not be part of the adaptable application; and (3) components that have access to a data item may not be able to record documentation to a provenance store. Due to UML2PROV is not about changing the application, its users must avoid provenance requirements that require the adaptation of the application. Thus, provenance use cases need to be compatible with application use cases. Third, our proposal works on the assumption that the implementation of the application conforms to its UML design. However, although it is not a good practice, applications that do not strictly follow the design specified by the UML diagrams are not unusual [41]. In these cases, users could leverage reverse-engineering to obtain the UML design according to the source code, as was done with GelJ. In fact, users in this situation could apply UML2PROV in the Application Mode that best suits their needs.

Alternatives to apply UML2PROV. In our evaluation, we have applied UML2PROV in different ways, but always to the whole application source code. However, a software engineer may be interested in applying UML2PROV differently in different application's modules to collect different provenance information from each module. For instance, an engineer may be interested in the provenance about the whole application, but she/he has specific provenance questions regarding a concrete module. In this case, our *Application Mode 1*, which requires the greatest manual effort, should be used only in the module of interest, and use Modes 2 or 3 in the remaining modules. In this way, users can answer their provenance questions, not needing to devote an extra effort to apply Mode 1 to the whole application. Similarly, in case of software engineers only interested in the specific provenance about a concrete module, they only need to apply UML2PROV (in different Mode depending on their needs) to such a module, avoiding the collection of provenance from the rest of the application.

Runtime overhead and storage needs. Our approach captures provenance at the level of operations. The literature recognizes that this kind of approaches involves more runtime overhead as more traced operations are executed [37], [38]. That is why the behaviour of UML2PROV when using Mode 1 is better than that of Modes 2 and 3. Additionally, as shown in the evaluation, when using Mode 1, the measured performance overhead grows as the percentage of instrumented executed operations with respect to the total of executed operations increases. Also, the nature of the traced operations (such as the number of inputs/outputs, type of pattern applied) affects the overhead since they determine the number of bindings to collect, and consequently, they have an impact on the performance. These results support the well-known fact that the more detailed provenance data is, the greater the impact recording it will have on application performance [37]. From our evaluation we can conclude that one way to determine how detailed provenance data should be (affecting as little as possible the application performance) is to formulate concrete requirement questions so that we can record, at minimum, provenance data at a level of detail great enough to answer these questions (such as in Mode 1). This recommendation is part of the PrIMe methodology for designing provenance-aware applications [8]. The configuration chosen for the BGM also affects the performance. Our evaluation followed a bulk submission strategy, approach that is aligned with other proposals [37], [38] that, e.g., led to savings in the overhead of establishing extra database connections. Thus, we recommend potential users to use bulk submission.

10 Related work

Provenance, an active topic of research within a wide range of areas, has been analyzed by several surveys from different point of views such as databases, workflow management systems, Big Data, and so on. For example, in [2] we presented a systematic literature review of provenance systems, and defined a six-dimensional taxonomy of provenance systems' characteristics, including also an exhaustive analysis and comparison of 25 systems attending to such a taxonomy. Among the existing literature, we remark the Kepler provenance module [42] and COMAND [43] since they extend the well-known workflow system called Kepler [44] with provenance capabilities. While these works undertake the adaptation of systems (in this case, Kepler) from scratch, other works provide mechanisms to adapt applications from the design phase. Among the scarce literature related to the latter –the context in which UML2PROV is applied– the PrIMe [8] was introduced to adapt applications to become provenance-aware or, to design applications keeping in mind the provenance capabilities from the design phase. Whilst it shows efficiency in several works such as [45] (healthcare domain), and [46] (High Performance Computing domain), it is not integrated with existing software engineering methodologies and also needs a manual adaptation of the application. At this point, it could be said that our approach complements PrIMe, since UML2PROV uses a well-known software engineering notation for the design of applications (i.e. UML) in order to obtain the design of the provenance.

As for technology aspects regarding to the capture of provenance, different approaches may be taken into account in order to avoid interleaving provenance generation code into the application's source code. Our proposal relies on AOP which has long been used for monitoring purposes, specially to check whether a system complies with its intended behavior [47], [48]. However, to our knowledge, the CAPS framework [49], which uses AspectJ to weave cross-cutting concerns of provenance generation into Java applications, is the only other proposal that uses AOP to capture provenance apart from UML2PROV. Other examples are PASS [50], which intercepts and translates system calls into sequences of meaningful provenance entries, or noWorkflow tool [51], that captures provenance of experiment scripts using a listener in the Python profiling API. Contrary to these approaches, UML2PROV does not require a kernel modified for automatic collecting provenance (like PASS) and it is not restricted to a specific programming language (such as noWorkflow or CAPS). UML2PROV is a generic solution based on the design of the application rather than specific technologies or programming languages.

Finally, MDD has been widely applied in industry in a broad range of companies, in a number of different domains [10], [52], and in many different ways, ranging from industry-wide efforts to define precise models for an entire application domain, to very restricted and limited uses such as code generation for a single application in a single company [52]. An interesting study showing significant figures is [53] about modelling during software development and about any variation of Model driven engineering, which concludes that approximately 68% of the studied sample uses models (such as UML) during software development and, among them, 44% generate code starting from models. In the particular case of using MDD to generate instrumentation code, we can distinguish those proposals that take as source of the MDD process the same models (also annotated or slightly changed) than the ones used for designing the system (such as our proposal or [54]), from those that require designing new models [48], [55]. Although in some of these works the generated instrumentation code is scattered along the system code [54], our proposal, as others such as [48], [55] (this latter one also suggests to use AOP), bets on a separation of concerns of the system design and instrumentation. However, to the best of our knowledge UML2PROV is the first contribution in using model driven instrumentation for monitoring provenance data.

11 Conclusions and Future work

The gap between application design and provenance design is an adoption hurdle for provenance technology. To address this challenge, we presented UML2PROV, a conceptual proposal exploiting the UML design of an application for automatically obtaining (i) the design of the provenance to be generated (i.e. the *PROV templates*), and (ii) a library to be linked with the application to collect provenance (i.e. the *BGM*). Concretely, UML2PROV is rigorously defined by an extensive set of 17 patterns mapping UML diagrams to templates, which are explained in a systematic way [13] to allow other implementers to replicate the functionality.

We also gave a reference implementation of UML2PROV and demonstrated its feasibility by means of a detailed evaluation that analyses its benefits and costs depending on the UML diagrams used as source. This analysis showed that a non-tailored UML design leads to capture more provenance than necessary to satisfy provenance requirements and, therefore, results in an increase of the overhead: ~55%, in the worst case, versus the ~1.5% corresponding to the best case. A particular strength of the evaluation is the range of perspectives considered: development effort, memory requirements, time performance, and quality of provenance.

There are a number of opportunities to build upon UML2PROV as further work. For instance, an interesting point would be to manage the level of detail of the provenance to be captured. The defined transformation patterns consider a high level of detail in the flow of logic within the system (SqDs), the change of objects' states (SMDs), or in the objects' status (CDs). In this line, we could adapt the transformation patterns, by selectively discarding some PROV elements or relations, and consequently, generating coarser grained data provenance. Likewise, providing the UML designer with a mechanism to specify the elements in the UML diagrams for provenance capture would be an interesting direction for future research. Additionally, the implementation of UML2PROV in other languages (such as Python), or providing UML2PROV as a service, constitute other lines of future work. Finally, as a complement of its evaluation, we consider interesting to compare UML2PROV to other proposals according to our taxonomy in [2].

Acknowledgment

This work is partially supported by the Spanish Ministerio de Economia y Competitividad (Project EDU2016-79838-P) and by the University of La Rioja (Grant FPI-UR-2015).

References

[1] P. Groth and L. Moreau (eds.), "PROV-Overview. An Overview of the PROV Family of Documents," World Wide Web Consortium, W3C Working Group Note NOTE-prov-overview-20130430, Apr. 2013. [Online]. Available: www.w3.org/TR/2013/NOTE-prov-overview-20130430/
[2] B. Pérez, C. Sáenz-Adán, and J. Rubio, "A systematic review of provenance systems," *Knowl. Inf. Syst.*, vol. 57, no. 3, pp. 495–543, 2018.
[3] L. Moreau, B. V. Batlajery, T. D. Huynh, D. T. Michaelides, and H. S. Packer, "A templating system to generate provenance," *IEEE Trans. Software Eng.*, vol. 44, no. 2, pp. 103–121, 2018.

[4] F. Curbera, Y. Doganata, A. Martens, N. K. Mukhi, and A. Slominski, "Business provenance – a technology to increase traceability of end-to-end operations," ser. Lecture Notes in Computer Science, vol. 5331, 2008, pp. 100–119.

[5] L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Seltzer, and A. Hopper, "A primer on provenance," *Commun. ACM*, vol. 57, no. 5, pp. 52–60, 2014.

[6] ProvToolbox. [Online]. Available: lucmoreau.github.io/ProvToolbox/

[7] ProvPy. [Online]. Available: pypi.python.org/pypi/prov

[8] S. Miles, P. T. Groth, S. Munroe, and L. Moreau, "PrIMe: A methodology for developing provenance-aware applications," *ACM T. Softw. Eng. Meth.*, vol. 20, no. 3, pp. 8:1–8:42, 2011.

[9] OMG. (2015) Unified Modeling Language (UML), v. 2.5. Document formal/15-03-01, March, 2015. [Online]. Available: www.omg.org/spec/UML/2.5/

[10] J. E. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proc. 33rd Int. Conf. Soft. Eng.*, 2011, pp. 471–480.

[11] C. Sáenz-Adán, L. Moreau, B. Pérez, S. Miles, and F. J. García-Izquierdo, "Automating provenance capture in software engineering with UML2PROV," in *Proc. 7th Int. Provenance and Annotation Workshop*, 2018, pp. 58–70.

[12] C. Sáenz-Adán, B. Pérez, T. D. Huynh, and L. Moreau, "UML2PROV: automating provenance capture in software engineering," in *Proc. 44th Int. Conf. on Current Trends in Theory and Practice of Computer Science*, 2018, pp. 667–681.

[13] UML2PROV. Supplementary material. [Online]. Available: uml2prov.unirioja.es/

[14] UML2PROV User Guide. [Online]. Available: https://github.com/uml2prov/uml2prov

[15] L. Moreau and P. Missier (eds.), "PROV-DM: The PROV Data Model," World Wide Web Consortium, W3C Recommendation REC-prov-dm-20130430, 2013. [Online]. Available: www.w3.org/TR/2013/REC-prov-dm-20130430/

[16] I. D. Baxter and M. Mehlich, "Reverse engineering is reverse forward engineering," *Sci. Comput. Program.*, vol. 36, no. 2, pp. 131 – 147, 2000.

[17] G. Reggio, M. Leotta, F. Ricca, and D. Clerissi, "What are the used UML diagrams? A preliminary survey," in *Proc. 3rd Int. Workshop on Experiences and Empirical Studies in Softw. Modeling*, 2013, pp. 3–12.

[18] OMG. (2014) Object Constraint Language (v. 2.4). Formal/2014-02-03. [Online]. Available: www.omg.org/spec/OCL/2.4/PDF

[19] M. Seidl, M. Scholz, C. Huemer, and G. Kappel, *UML@Classroom: An Introduction to Object-Oriented Modeling*. Springer Publishing Company, Incorporated, 2015.

[20] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic identification of class stereotypes," in *Proc. 26th IEEE Int. Conf. on Software Maintenance*, 2010, pp. 1–10.

[21] C. Martínez-Costa, M. Menárguez-Tortosa, and J. T. Fernández-Breis, "Clinical data interoperability based on archetype transformation," *J. Biomed. Inform.*, vol. 44, no. 5, pp. 869–880, 2011.

[22] L. Moreau and P. Missier (eds.), "PROV-N: The Provenance Notation," World Wide Web Consortium, W3C Recommendation REC-prov-n-20130430, Apr. 2013. [Online]. Available: www.w3.org/TR/2013/REC-prov-n-20130430/

[23] ATL - a model transformation technology, version 3.8. [Online]. Available: www.eclipse.org/atl/

[24] Eclipse. XTend. [Online]. Available: www.eclipse.org/xtend/

[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proc. European Conf. on Object-Oriented Programming*, 1997, pp. 220–242.

[26] AspectJ Project. [Online]. Available: www.eclipse.org/aspectj/

[27] M. Johnson, L. Moreau, A. Chapman, P. Gandhi, and C. Sáenz-Adán, "Using the provenance from astronomical workflows to increase processing efficiency," in *Proc. 7th Int. Provenance and Annotation Workshop*, 2018, pp. 101–112.

[28] J. Heras, C. Domínguez, E. Mata, V. Pascual, C. Lozano, C. Torres, and M. Zarazaga, "GelJ – a tool for analyzing DNA fingerprint gel images," *BMC Bioinformatics*, vol. 16, no. 1, Aug 2015.

[29] M. M. Read, *Trends in DNA Fingerprint Research*. New York, USA: Nova Science Publishers, Inc, 2005.

[30] L. Moreau et al., "Special issue: The first provenance challenge," *Concurr. Comp.-Pract. E.*, vol. 20, no. 5, pp. 409–418, Apr. 2008. [Online]. Available: http://dx.doi.org/10.1002/cpe.v20:5

[31] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of software engineering (2. ed.)*. Prentice Hall, 2003.

[32] Papyrus, Modeling environment (v. 2.0.2). (2018, December). [Online]. Available: eclipse.org/papyrus/

[33] The ObjectAid UML Explorer for Eclipse (v. 1.2.2). (2017, November). [Online]. Available: www.objectaid.com/

[34] N. Walkinshaw, K. Bogdanov, S. Ali, and M. Holcombe, "Automated discovery of state transitions and their functions in source code," *Softw. Test. Verif. Reliab.*, vol. 18, no. 2, pp. 99–121, 2008.

[35] MaintainJ, v. 4.2.0. (2014). [Online]. Available: maintainj.com/

[36] MongoDB Inc. (v. 4.0.2). (2018, August). [Online]. Available: www.mongodb.org/

[37] P. Groth. (2007) The origin of data. Enabling the determination of provenance in multi-institutional scientific systems through the documentation of processes. University of Southampton, School of Electronics and Computer Science, Doctoral Thesis. Retrieved from https://eprints.soton.ac.uk/264649/.

[38] Z. Chen and L. Moreau, "Implementation and evaluation of a protocol for recording process documentation in the presence of failures," in *Proc. 2nd Int. Provenance and Annotation Workshop*, 2008, pp. 92–105.

[39] H. Park, R. Ikeda, and J. Widom, "RAMP: A system for capturing and tracing provenance in mapreduce workflows," *PVLDB*, vol. 4, no. 12, pp. 1351–1354, 2011.

[40] G. Scanniello, C. Gravino, and G. Tortora, "Investigating the role of UML in the software modeling and maintenance - A preliminary industrial survey," in *Proc. 12th Int. Conf. on Enterprise Information Systems*, 2010, pp. 141–148.

[41] P. V. Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards automating source-consistent UML refactorings," in *Proc. 6th Int. Conf. on The Unified Modeling Language*, 2003, pp. 144–159.

[42] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance Collection Support in the Kepler Scientific Workflow System," in *Proc. 1st Int. Provenance and Annotation Workshop*, 2006, pp. 118–132.

[43] S. Bowers, T. M. McPhillips, and B. Ludäscher, "Provenance in collection-oriented scientific workflows," *Concurr. Comp.-Pract. E.*, vol. 20, no. 5, pp. 519–529, 2008.

[44] The Kepler Project. [Online]. Available: kepler-project.org/

[45] S. Álvarez-Napagao, J. Vázquez-Salceda, T. Kifor, L. Z. Varga, and S. Willmott, "Applying provenance in distributed organ transplant management," in *Proc. 1st Int. Provenance and Annotation Workshop*, 2006.

[46] V. Silva, R. Souza, J. J. Camata, D. de Oliveira, P. Valduriez, A. L. G. A. Coutinho, and M. Mattoso, "Capturing provenance for runtime data analysis in computational science and engineering applications," in *Proc. Int Provenance and Annotation Workshop*, 2018, pp. 183–187.

[47] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir, "A survey of runtime monitoring instrumentation techniques," in *Proc. 2nd Int. Workshop on Pre- and Post-Deployment Verification Techniques*, 2017, pp. 15–28.

[48] M. Boskovic, T. Warns, and W. Hasselbring, "Model Driven Instrumentation for Relational Event Traces," *Radioelektronic and Computer Systems*, no. 6(18), 2006.

[49] P. C. Brauer, F. Fittkau, and W. Hasselbring, "The Aspect-Oriented Architecture of the CAPS Framework for Capturing, Analyzing and Archiving Provenance Data," in *Proc. 5th Int. Provenance and Annotation Workshop*, 2014, pp. 223–225.

[50] D. A. Holland, M. I. Seltzer, U. Braun, and K.-K. Muniswamy-Reddy, "PASSing the provenance challenge," *Concurr. Comp.-Pract. E.*, vol. 20, no. 5, pp. 531–540, 2008.

[51] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts," in *Proc. of the VLDB Endowment*, vol. 10, no. 12, 2017, pp. 1841–1844.

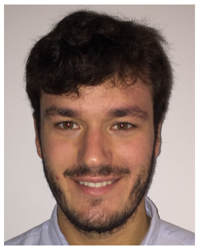
[52] J. Whittle, J. E. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014.

[53] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio, "Relevance, benefits, and problems of software modelling and model driven techniques - A survey in the italian industry," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2110–2126, 2013.

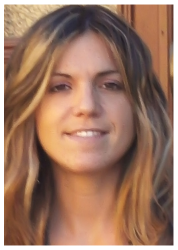
[54] M. Funk, P. Hoyer, and S. Link, "Model-driven instrumentation of graphical user interfaces," in *Proc. 2nd Int. Conf. on Advances in Computer-Human Interaction*, 2009, pp. 19–25.

[55] C. Momm, T. Detsch, and S. Abeck, "Model–driven instrumentation for monitoring the quality of web service compositions,"

in *Proc. 12th Enterprise Distributed Object Computing Conference Workshops*, 2008, pp. 58–67.



Carlos Sáenz-Adán received his B.Sc. degree in Computer Science from the University of La Rioja, in 2013, a Master degree in Advanced Computer Systems from the University of País Vasco (UPV/EHU), in 2014, and his Ph.D. in Computer Science from the University of La Rioja, in 2019. His research, which has been partially supported by the University of La Rioja grant (FPI-UR-2015), is mainly focused on Business process modeling and provenance.



Beatriz Pérez is an Assistant Professor in Computer Science at the University of La Rioja, Spain. She received her B.Sc. degree in mathematics from the University of La Rioja, Spain, in 2003, and her Ph.D. in Computer Science from the University of Zaragoza, Spain, in 2011. Her main research interests are in Meta-modelling, Model-Driven Development, Business process modeling and provenance.

Francisco J. García-Izquierdo received the Ph.D. degree in telecommunications engineering from the University of Zaragoza, Spain in 1999. He is now an Assistant Professor at the University of La Rioja, Spain. His research interests include technologies for web applications, modeling theories, provenance and innovative ways to teach engineering concepts.

Luc Moreau is a Professor of Computer Science and Head of the department of Informatics, at King's College London. He was co-chair of the W3C Provenance Working Group, which resulted in four W3C Recommendations and nine W3C Notes, specifying PROV, a conceptual data model for provenance the Web, and its serializations in various Web languages. Previously, he initiated the successful Provenance Challenge series, which saw the involvement of over 20 institutions investigating provenance inter-operability in 3 successive challenges, and which resulted in the specification of the community Open Provenance Model (OPM).

INTEGRATING PROVENANCE CAPTURE AND UML WITH UML2PROV: PRINCIPLES AND EXPERIENCE

SUPPLEMENTARY MATERIAL

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

[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@kcl.ac.uk

Table of content

- Specification of the UML to PROV patterns
- Implementation details
- Taxonomy of Class' operations
- Sequence of interactions with GelJ
- Second Evaluation Case Study
- OCL Constraints

INTEGRATING PROVENANCE CAPTURE AND UML WITH UML2PROV: PRINCIPLES AND EXPERIENCE

–SUPPLEMENTARY MATERIAL–

SPECIFICATION OF THE UML TO PROV PATTERNS

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

[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@kcl.ac.uk

1 Introduction

This specification defines in detail the patterns for translating UML into PROV. The main objective of this specification is to provide a tool for implementing systems that include provenance capabilities during their design phase.

This document has been organized into three main parts:

- Section 2 provides an insight into the information required for the accurate understanding of this specification such as the notational conventions used throughout the document (Section 2.1), or the description of the structure for the pattern's explanations (Section 2.2).
- Section 3 shows a table that associates each pattern's identifier with the page where it is explained.
- Sections from 4 to 6 provide a systematic explanation of each pattern classified by the addressed UML diagram.

2 Before reading

As we see this document as the reference specification for UML2PROV, each pattern has been written in a self-contained way. A reader who reads all the patterns sequentially from the first to the last will find similar explanations, even repeated ones, in several patterns. We have preferred to make the reader suffer this small inconvenience, instead of running the risk that an occasional reader of a particular pattern loses part of the explanations that are discussed elsewhere.

We assume that the reader is familiar with the following UML diagrams: UML Sequence Diagrams (SqDs), UML State Machine Diagrams (SMDs), and UML Class diagrams (CDs). Readers unfamiliar with these diagrams are encouraged to read the UML specification [1]. Additionally, due to the fact that transformations referring to CDs make use of concrete UML stereotypes used to classify UML Class's operations, we refer to Appendix A for an overview about them.

Likewise, we assume that the reader is knowledgeable about both the PROV data model (PROV-DM) [2], to represent provenance information, and the PROV template approach [3], for designing provenance. If this is not the case, she/he is referred to [2] and [3], respectively.

2.1 Notational conventions

More than a terminological nuance, the distinction between the *state* and the *status* of an object is fundamental to understand this document. More specifically:

- In SMDs, in accordance to UML terminology [1], the *state* of an object denotes a situation during which some invariant conditions holds.
- In CDs, we use the term object's *status* with a broad scope, referring to the values of the object's attributes at some moment, which particularly could correspond to a concrete *state* but not necessarily.

The PROV templates throughout this document are represented following the PROV graph conventions given in [4].

We also use *qualified names* (e.g., `prov:value`) in accordance to PROV-DM [2]. In compliance with PROV-DM, we note that a *qualified name* can be mapped to an Internationalized Resource Identifier (IRI) [5] by concatenating the IRI associated with the prefix (e.g., `prov`) and the local part (e.g., `value`). Every *qualified name* with a prefix refers to the namespace of the prefix. The following namespaces and prefixes are used throughout this document.

prefix	**namespace IRI**	**definition**
var	`http://openprovenance.org/var#`	The namespace for template variables
prov	`http://www.w3.org/ns/prov#`	The PROV namespace
xsd	`http://www.w3.org/2000/10/XMLSchema#`	XML Schema namespace
u2p	`http://uml2prov.unirioja.es/ns/u2p#`	UML2PROV namespace

Table 1. Prefix and Namespaces used in this specification

2.2 Structure of the patterns

We have structured the explanation of the defined patterns in the same five blocks: **Identifier**, **Context**, **UML Diagram**, **Mapping to PROV**, and **Discussion**. See the explanation of each block below.

2.2.1 Identifier

Unique identifier of the transformation pattern. It is an acronym that refers to the type of UML diagram together with a numeric identifier. The UML *Seq*uence diagram *P*atterns are referred to as *SeqP<N>*, where `N` is the numeric identifier. Likewise, *StP<N>* corresponds to the UML *St*ate Machine *P*atterns, and *ClP<N>* to the UML *Cl*ass Diagram *P*atterns.

2.2.2 Context

The behaviour addressed by the pattern. In order to give a free of context explanation, being as agnostic as possible about the modeling language used to represent such a behaviour, we will use the natural language including well-known software engineering terminology (e.g., *object*, *operation*...), to identify the part of the domain for which the corresponding pattern proposes a translation.

Each pattern context block will include a detailed description of its *key elements*. When necessary, we will use nested elements to describe the different alternatives through which certain *key elements* participate in the context. We remark that not all the identified *key elements* explicitly appear in the context. Some patterns identify specific *key elements* that are inferred from the context because they play an important role in the pattern.

2.2.3 UML Diagram

This block will depict the excerpt of the UML diagram with the elements that model the previous *key elements*. In addition, we provide a table, whose structure is illustrated below, that explains the representation of each *key element*, by means of UML elements. Additionally, we assign a green label containing a numeric identifier to each UML element, which makes it easier its location in the UML diagram.

Key Element	**UML**	**Rationale**
Name of the element	`UML element` 🟩	The fundamental reasons serving to account for the use of the *UML element* for modelling the *key element*.

2.2.4 Mapping to PROV

This block contains the PROV template generated from the previous excerpt of *UML Diagram*, together with a explanation about the transformation, that is, the *PROV elements*, *attributes*, and *PROV relations* generated from the UML elements in the *UML Diagram*. We assign a numeric identifier to each *PROV element* that corresponds to the identifier of the *UML* element from which it comes from. Additionally, each *relation* among PROV elements appearing in the PROV template is labeled with a letter that helps link such a relation with its description. The structure used to specify this block is the following:

PROV elements

UML	PROV / id	Rationale
`UML element` id	`PROV element` id / `var:<identifier>`	The explanation of the mapping between *UML element* and *PROV element*.

Attributes

PROV Element	Attribute / Value	Description
`PROV element` id	name of attribute / assigned value	The description of the meaning of the attribute and its value.

Note: Throughout this specification, we have included the attributes `tmpl:startTime` and `tmpl:endTime` associated with `Activities` because we consider such an information very useful from a provenance point of view. Nevertheless, both attributes are optional and the user is free to include them.

PROV relations

PROV Relation	Description
PROV relation id	Description of the relation.

Note: In PROV, two relationships of the form (B, `prov:used`, A) and (C, `prov:wasGeneratedBy`, B) may be enriched with (C, `prov:wasDerivedFrom`, A) to express the dependency of C on A. This structure is a provenance construction called *use-generate-derive triangle* [3] which explicitly connects a *generated* `prov:Entity` to a *used* `prov:Entity`. In the realm of this work, it may be applied in those templates in which a `prov:Entity` is *used* by a `prov:Activity`, and such a `prov:Activity` *generated* another `prov:Entity`. However, aiming at avoiding the overburden of the PROV template explanations with information that can be inferred, we have decided to include the relation `prov:wasDerivedFrom` only when the context of the pattern explicitly refers to such a derivation.

2.2.5 Discussion

Issues related to the transformation of UML to PROV. Concretely, we will focus on the explanation and justification of our transformation decisions together with alternative solutions (if any), and some questions that are likely to come up to the reader.

3 Index of patterns

Modelled by means of UML Sequence Diagrams

Pattern identifier	Context	Page
SeqP1	A participant (the sender) interacts with another participant (the recipient) by calling an operation in the recipient, and then, it continues immediately. The call causes the recipient to execute the operation.	6
SeqP2	A participant (the sender) interacts with another participant (the recipient) by calling an operation in the recipient and waiting for a response. The call causes the recipient to execute the operation and to respond the sender after the execution.	9
SeqP3	During the execution of an operation (main operation), a nested operation call is made. After this call, the execution of the main operation can either continue immediately or wait for the response of that nested operation call. This way, this pattern complements *SeqP1* and *SeqP2*.	12
SeqP4	During the execution of an operation (main operation), a response of a previously issued nested operation call is received. The main operation's execution uses this response to complete its behaviour. This way, this pattern complements *SeqP1* and *SeqP2* with additional information regarding the response to the *nested operation call* (addressed by *SeqP3*).	15

Modelled by means of UML State Machine Diagrams

Pattern identifier	Context	Page
StP1	As a consequence of the execution of an operation, an object is created in its first state. This operation is usually the constructor of the object.	19
StP2	As a consequence of the execution of an operation, the behaviour of an object is completed.	23
StP3	As a consequence of the execution of an operation, an object changes its state.	27

Modelled by means of UML Class Diagrams

Pattern identifier	Context	Page
ClP1	The execution of an operation provokes the creation of a new object.	33
ClP2	The execution of an operation provokes the destruction of an object.	36
ClP3	The execution of an operation on an object returns values of concrete object's attributes. The values are returned as they are, without any further processing. This execution does not provoke the change of the object's status.	38
ClP4	The execution of an operation on an object returns values that are computed based on the object's status as a whole (the values of concrete attributes involved in the computation are unknown or irrelevant). This execution does not provoke the change of the object's status.	41
ClP5	The execution of an operation on an object returns values that are computed based on values of concrete object's attributes. This execution does not provoke the change of the object's status.	44
ClP6	The execution of an operation on an object changes the object's status as a whole (the concrete modified attributes are unknown or irrelevant).	48
ClP7	The execution of an operation on an object directly sets the information passed to the operation as values of concrete object's attributes, thus provoking a change in the object's status.	53
ClP8	The execution of an operation on an object changes the values of concrete object's attributes, thus provoking a change in the object's status.	58
ClP9	The execution of an operation on an object removes element(s) from a concrete object's collection attribute, thus provoking a change in the object's status.	63
ClP10	The execution of an operation on an object directly adds the information passed to the operation as new element(s) of a concrete object's collection attribute, thus provoking a change in the object's status.	68

4 UML Sequence Diagrams

Pattern identifier	**Context**	**Page**
SeqP1	A participant (the sender) interacts with another participant (the recipient) by calling an operation in the recipient, and then, it continues immediately. The call causes the recipient to execute the operation.	6
SeqP2	A participant (the sender) interacts with another participant (the recipient) by calling an operation in the recipient and waiting for a response. The call causes the recipient to execute the operation and to respond the sender after the execution.	9
SeqP3	During the execution of an operation (main operation), a nested operation call is made. After this call, the execution of the main operation can either continue immediately or wait for the response of that nested operation call. This way, this pattern complements *SeqP1* and *SeqP2*.	12
SeqP4	During the execution of an operation (main operation), a response of a previously issued nested operation call is received. The main operation's execution uses this response to complete its behaviour. This way, this pattern complements *SeqP1* and *SeqP2* with additional information regarding the response to the *nested operation call* (addressed by *SeqP3*).	15

Identifier *Sequence diagram Pattern 1* (*SeqP1*)

Context

A participant (the sender) interacts with another participant (the recipient) by calling an operation in the recipient, and then, it continues immediately. The call causes the recipient to execute the operation.

Key elements

Sender	The participant that makes the operation call.
Operation call	The call that starts the execution of the operation.
Input data	The information (if any) passed to the operation through the *Operation call*.
Operation execution	The execution of the operation.

UML Diagram

Key Element	UML	Rationale
Sender	`Lifeline` 1	It models the *Sender* participant involved in the interaction.
Operation call	`Asynchronous Message` 2	It models the *Operation call* when the *Sender* does not wait for a response, but instead continues immediately after sending the message.
Input data	`Input Arguments` 3	They specify the information passed to the operation through the *Operation call*.
Operation execution	`ExecutionSpecification` 4	It shows the period of time that the recipient's participant devotes to the *Operation execution*.

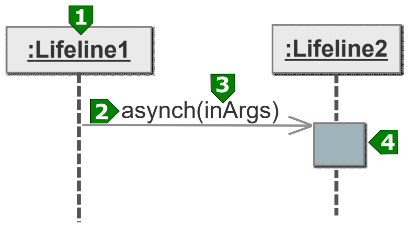


Figure 1. UML representation that models the context given by *SeqP1*

Mapping to PROV

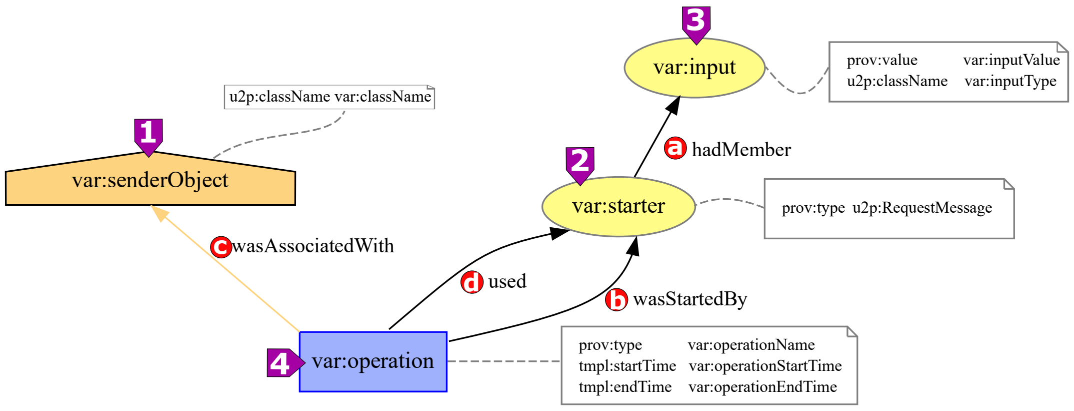

3
var:input
prov:value var:inputValue
u2p:className var:inputType
u2p:className var:className
1
var:senderObject
a hadMember
2
var:starter
prov:type u2p:RequestMessage
c wasAssociatedWith
d used
b wasStartedBy
4
var:operation
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime

Figure 2. PROV template generated from the UML representation used in *SeqP1* (Figure 1)

PROV elements

UML	PROV / id	Rationale
`Lifeline` 1▶	`prov:Agent` 1▶ / `var:senderObject`	The sender `Lifeline` 1▶ is mapped to a `prov:Agent` identified by `var:senderObject`. It assumes the responsibility for starting the `ExecutionSpecification` 4▶.
`Asynchronous Message` 2▶	`prov:Entity` 2▶ / `var:starter`	The `Asynchronous Message` 2▶ that initiates the `ExecutionSpecification` 4▶ of the recipient is a `prov:Entity` with identifier `var:starter`.
`Input Arguments` 3▶	`prov:Entity` 3▶ / `var:input`	Each argument of `Input Arguments` 3▶ is a separate `prov:Entity` identified as `var:input`.
`ExecutionSpecification` 4▶	`prov:Activity` 4▶ / `var:operation`	The `ExecutionSpecification` 4▶ is a `prov:Activity` with identifier `var:operation`.

Attributes

PROV Element	Attribute / Value	Description
`var:senderObject` 1▶	`u2p:typeName` / `var:className`	The value `var:className` is the string with the name of the class to which the `var:senderObject` 1▶ belongs.
`var:starter` 2▶	`prov:type` / `u2p:RequestMessage`	The value `u2p:RequestMessage` shows that `var:starter` 2▶ is a request message.
`var:input` 3▶	`prov:value` / `var:inputValue`	The value `var:inputValue` is the direct representation of `var:input` 3▶.
	`u2p:typeName` / `var:inputType`	The value `var:inputType` is the string with the name of the class to which `var:input` 3▶ belongs.
`var:operation` 4▶	`prov:type` / `var:operationName`	The value `var:operationName` is the name of the operation `var:operation` 4▶.
	`tmpl:startTime` / `var:operationStartTime`	`var:operationStartTime` is an `xsd:dateTime` value for the start of `var:operation` 4▶.
	`tmpl:endTime` / `var:operationEndTime`	`var:operationEndTime` is an `xsd:dateTime` value for the end of `var:operation` 4▶.

PROV relations

PROV Relation	Description
ⓐ `prov:hadMember`	It states that `var:input` is one of the elements in `var:starter`.
ⓑ `prov:wasStartedBy`	`var:operation` is deemed to have been started by `var:starter`.
ⓒ `prov:wasAssociatedWith`	It is the assignment of responsibility to `var:senderObject` for `var:operation`.
ⓓ `prov:used`	It is the beginning of utilizing `var:starter` by `var:operation`.

Discussion

- Figure 2 depicts the responsibility of the *Sender* lifeline (`var:senderObject`) for the recipient lifeline to execute the operation (`var:operation`). However, the recipient lifeline is not modelled in this PROV template, even though it is the participant that executes the operation. This decision is based on other patterns' better ability to both (1) identify the participant responsible for executing that operation, and (2) give a more detailed information about the implications that the execution of that operation has in the recipient participant. More specifically, these patterns are: *StP1*-*StP3*, which mainly focus on representing possible changes in an object's state caused by an *Operation execution*; and patterns *ClP1*-*ClP10*, which put more stress on how the execution affects the status of the object responsible for performing such an execution.
- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we have considered this circumstance with the aim of covering a wider spectrum of cases. When the *Operation call* lacks *Input*

data, the UML representation in Figure 1 will not include `Input Arguments` ❸. As a consequence, the resulting PROV template in Figure 2 will also lack `var:input` ❸ and its associated PROV relations. Finally, we remark that the resulting PROV template does not reflect the *usage* of `var:input` ❸ by `var:operation` ❹ because SqDs stick to the flow of information, not its usage. Patterns addressing CDs (*ClP1*-*ClP10*) are better suited for this purpose.

Identifier *Sequence diagram Pattern 2* (*SeqP2*)

Context

A participant (the sender) interacts with another participant (the recipient) by calling an operation in the recipient and waiting for a response. The call causes the recipient to execute the operation and to respond the sender after the execution.

Key elements

Sender	The participant that makes the operation call.
Operation call	The call that starts the execution of the operation.
Input data	The information (if any) passed to the operation through the *Operation call*.
Operation execution	The execution of the operation.
Response	The recipient's response to the *Operation call*.
Output data	The information contained in the *Response*.

UML Diagram

Key Element	UML	Rationale
Sender	`Lifeline` 1	It models the *Sender* participant involved in the interaction.
Operation call	`Synchronous Message` 2	It models the *Operation call* when the *Sender* waits for a response.
Input data	`Input Arguments` 3	They specify the information passed to the operation through the *Operation call*.
Operation execution	`ExecutionSpecification` 4	It shows the period of time that the recipient's participant devotes to the *Operation execution*.
Response	`Reply Message` 5	It specifies the response to the *Operation call*.
Output data	`Output Arguments` 6	They specify the information contained in the *Response*.

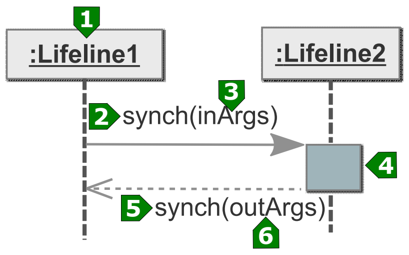
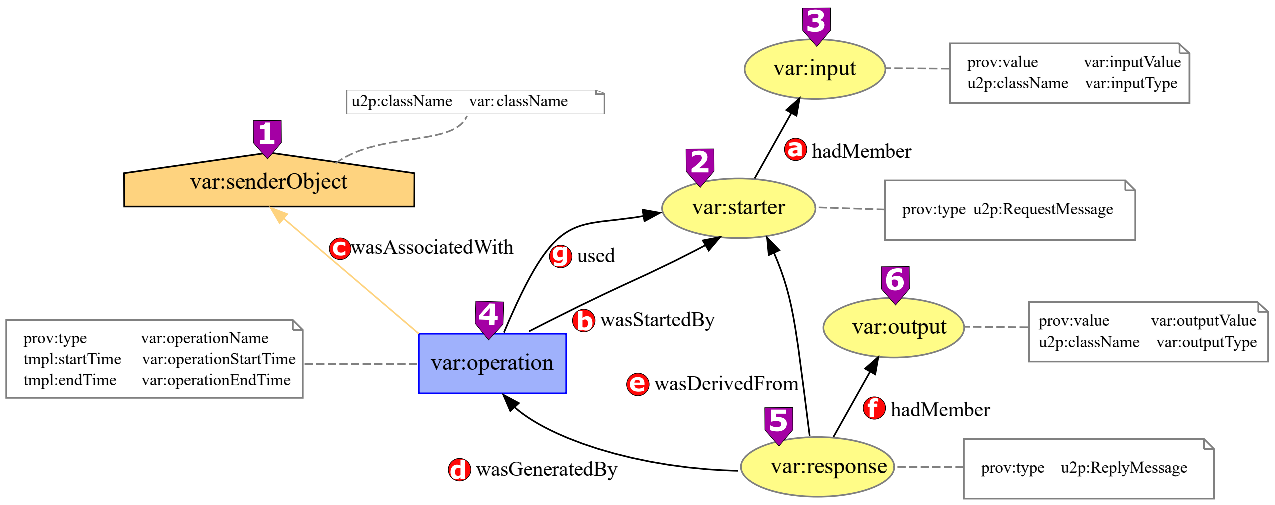


Figure 3. UML representation that models the context given by *SeqP2*

Mapping to PROV



1
var:senderObject
u2p:className var: className
3
var:input
prov:value var:inputValue
u2p:className var:inputType
a hadMember
2
var:starter
prov:type u2p:RequestMessage
c wasAssociatedWith
g used
b wasStartedBy
6
var:output
prov:value var:outputValue
u2p:className var:outputType
4
var:operation
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
e wasDerivedFrom
f hadMember
5
var:response
prov:type u2p:ReplyMessage
d wasGeneratedBy

Figure 4. PROV template generated from the UML representation used in *SeqP2* (Figure 3)

PROV elements

UML	PROV / id	Rationale
`Lifeline` 1▶	`prov:Agent` 1▶ / `var:senderObject`	The sender `Lifeline` 1▶ is mapped to a `prov:Agent` identified by `var:senderObject`. It assumes the responsibility for starting the `ExecutionSpecification` 4▶.
`Synchronous Message` 2▶	`prov:Entity` 2▶ / `var:starter`	The `Synchronous Message` 2▶ that initiates the `ExecutionSpecification` 4▶ of the recipient is a `prov:Entity` with identifier `var:starter`.
`Input Arguments` 3▶	`prov:Entity` 3▶ / `var:input`	Each argument of `Input Arguments` 3▶ is a separate `prov:Entity` identified as `var:input`.
`ExecutionSpecification` 4▶	`prov:Activity` 4▶ / `var:operation`	The `ExecutionSpecification` 4▶ is a `prov:Activity` with identifier `var:operation`.
`Reply Message` 5▶	`prov:Entity` 5▶ / `var:response`	The `Reply Message` 5▶ that responds to the `Synchronous Message` 2▶ is a `prov:Entity` with identifier `var:response`.
`Output Arguments` 6▶	`prov:Entity` 6▶ / `var:output`	Each argument of `Output Arguments` 6▶ is a separate `prov:Entity` identified as `var:output`.

Attributes

PROV Element	Attribute / Value	Description
`var:senderObject` ❶	`u2p:typeName` / `var:className`	The value `var:className` is the string with the name of the class to which the `var:senderObject` ❶ belongs.
`var:starter` ❷	`prov:type` / `u2p:RequestMessage`	The value `u2p:RequestMessage` shows that `var:starter` ❷ is a request message.
`var:input` ❸	`prov:value` / `var:inputValue`	The value `var:inputValue` is the direct representation of `var:input` ❸.
	`u2p:typeName` / `var:inputType`	The value `var:inputType` is the string with the name of the class to which the `var:input` ❸ belongs.
`var:operation` ❹	`prov:type` / `var:operationName`	The value `var:operationName` is the name of the operation `var:operation` ❹.
	`tmpl:startTime` / `var:operationStartTime`	The `var:operationStartTime` is an `xsd:dateTime` value for the start of `var:operation` ❹.
	`tmpl:endTime` / `var:operationEndTime`	The `var:operationEndTime` is an `xsd:dateTime` value for the end of `var:operation` ❹.
`var:response` ❺	`prov:type` / `u2p:ReplyMessage`	The value `u2p:ReplyMessage` shows that `var:response` ❺ is a reply message.
`var:output` ❻	`prov:value` / `var:outputValue`	The value `var:outputValue` is the direct representation of `var:output` ❻.
	`u2p:typeName` / `var:outputType`	The value `var:outputType` is a string with the name of the class to which `var:output` ❻ belongs.

PROV relations

PROV Relation	Description
ⓐ `prov:hadMember`	It states that `var:input` is one of the elements in `var:starter`.
ⓑ `prov:wasStartedBy`	`var:operation` is deemed to have been started by `var:starter`.
ⓒ `prov:wasAssociatedWith`	It is the assignment of responsibility to `var:senderObject` for `var:operation`.
ⓓ `prov:wasGeneratedBy`	It is the completion of production of `var:response` by `var:operation`.
ⓔ `prov:wasDerivedFrom`	It is the construction of `var:response` based on `var:starter` reception.
ⓕ `prov:hadMember`	It states that `var:output` is one of the elements in `var:response`.
ⓖ `prov:used`	It is the beginning of utilizing `var:starter` by `var:operation`.

Discussion

- Figure 4 depicts the responsibility of the *Sender* lifeline (`var:senderObject`) for executing the operation (`var:operation`) in a recipient lifeline. However, the recipient lifeline is not modelled in this PROV template, even though it is the participant that executes the operation. This decision is based on other patterns' better ability to both (1) identify the participant responsible for executing that operation, and (2) give a more detailed information about the implications that the execution of that operation has in the recipient participant. More specifically, these patterns are: *StP1*-*StP3*, which mainly focus on representing possible changes in an object's state caused by an *Operation execution*; and patterns *ClP1*-*ClP10*, which put more stress on how the execution affects the status of the object responsible for performing such an execution.

- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we have considered this circumstance with the aim of covering a wider spectrum of cases. When the *Operation call* lacks *Input data*, the UML representation in Figure 3 will not include `Input Arguments` ❸. As a consequence, the resulting PROV template in Figure 4 will also lack `var:input` ❸ and its associated PROV relations. Finally, we remark that the resulting PROV template does not reflect the *usage* of `var:input` ❸ by `var:operation` ❹ because SqDs stick to the flow of information, not its usage. Patterns addressing CDs (*ClP1*-*ClP10*) are better suited for this purpose.

Identifier *Seq*uence diagram *P*attern *3* (*SeqP3*)

Context

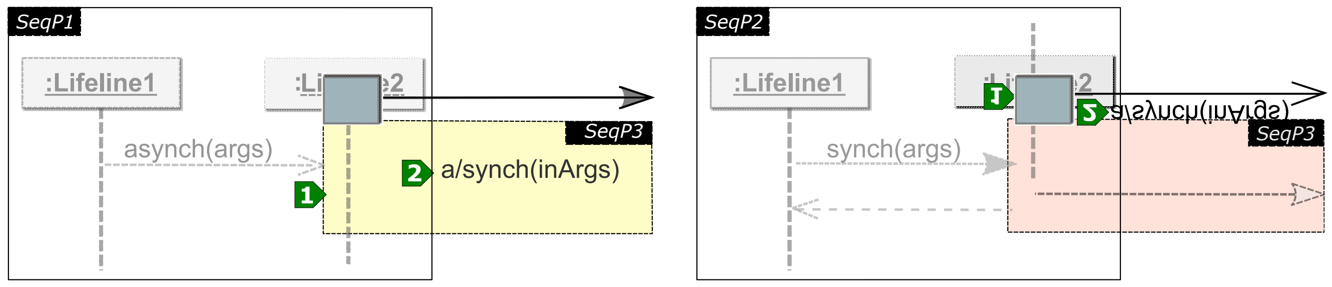
During the execution of an operation (main operation), a nested operation call is made. After this call, the execution of the main operation can either continue immediately or wait for the response of that nested operation call. This way, this pattern complements *SeqP1* and *SeqP2*.

Key elements

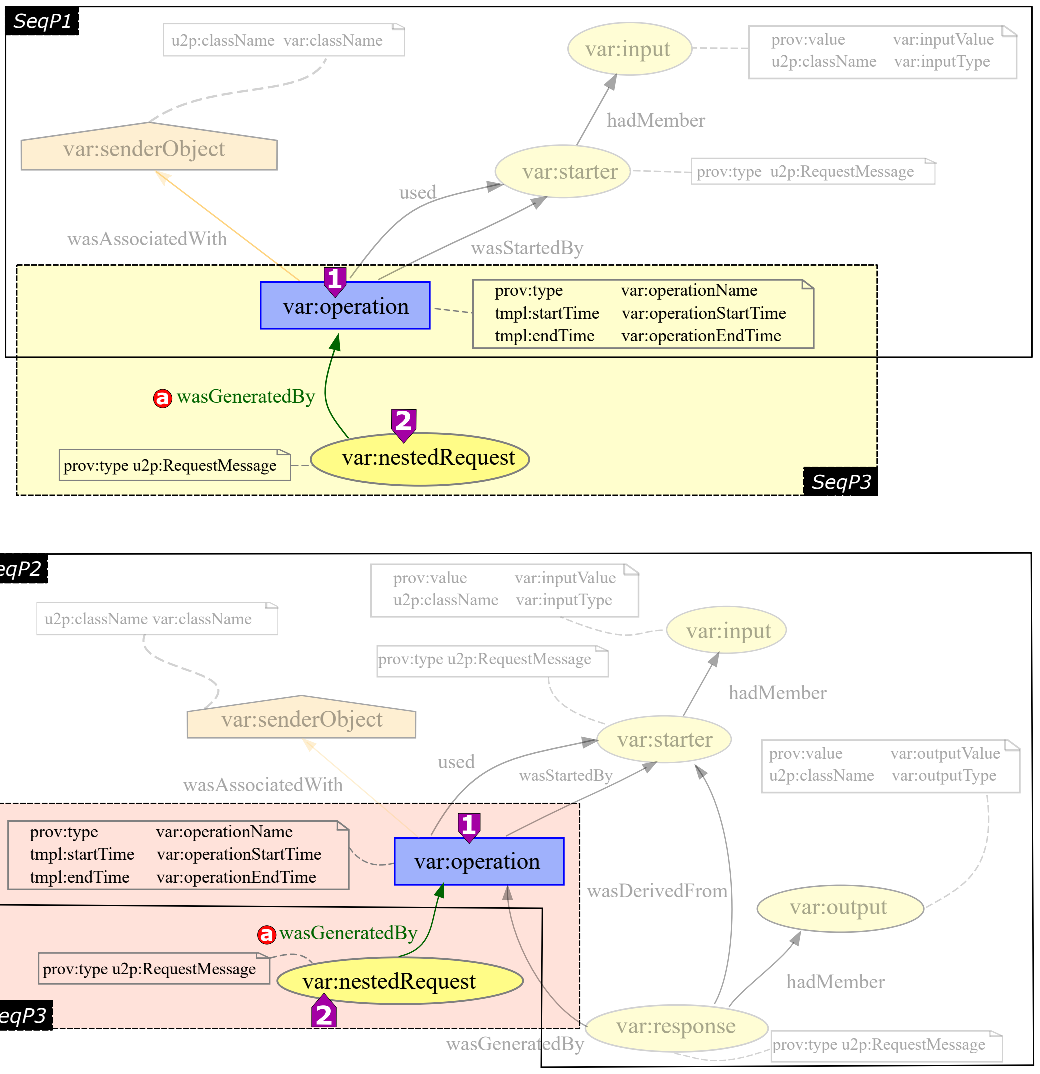
(Main) *Operation execution*	The execution of the main operation.
(Nested) *Operation call*	The nested operation call sent during the *Main operation execution*.

UML Diagram

Key Element	UML	Rationale
Main operation execution	`ExecutionSpecification` 1	It shows the period of time that takes the *Main operation execution*.
Nested operation call	`Synchronous Message` 2 or `Asynchronous Message` 2	It models the *Nested operation call* either when its sender waits for a response, or when it does not wait for a response, but instead continues immediately after sending the message.



Mapping to PROV



SeqP1
u2p:className var:className
var:input
prov:value var:inputValue
u2p:className var:inputType
hadMember
var:senderObject
var:starter
prov:type u2p:RequestMessage
used
wasAssociatedWith
wasStartedBy
1
var:operation
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
a wasGeneratedBy
2
prov:type u2p:RequestMessage
var:nestedRequest
SeqP3
SeqP2
prov:value var:inputValue
u2p:className var:inputType
u2p:className var:className
var:input
prov:type u2p:RequestMessage
hadMember
var:senderObject
var:starter
used
wasStartedBy
prov:value var:outputValue
u2p:className var:outputType
wasAssociatedWith
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
1
var:operation
wasDerivedFrom
var:output
a wasGeneratedBy
prov:type u2p:RequestMessage
var:nestedRequest
hadMember
SeqP3
2
var:response
wasGeneratedBy
prov:type u2p:RequestMessage

Figure 6. At the top, it is a PROV template generated from the UML representation in the left side of Figure 5. At the bottom, it is a PROV template generated from the UML representation in the right side of Figure 5. Only the shaded areas correspond to the PROV elements contributed by this pattern.

PROV elements

UML	PROV / id	Rationale
`ExecutionSpecification` 1▶	`prov:Activity` 1▶ / `var:operation`	The `ExecutionSpecification` 1▶ is a `prov:Activity` with identifier `var:operation`.
`Synchronous Message` 2▶ or `Asynchronous Message` 2▶	`prov:Entity` 2▶ / `var:nestedRequest`	The `Synchronous Message` or `Asynchronous Message` 2▶ sent from the `ExecutionSpecification` 1▶ is a `prov:Entity` with identifier `var:nestedRequest`.

Attributes

PROV Element	Attribute / Value	Description
var:operation 1▶	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 1▶
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 4▶.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 4▶.
var:nestedRequest 2▶	prov:type / u2p:RequestMessage	The value u2p:RequestMessage shows that var:nestedRequest 5▶ is a request message.

PROV relations

PROV Relation	Description
ⓐ prov:wasGeneratedBy	It is the completion of production of var:nestedRequest by var:operation.

Discussion

- The same element (‘request message’ in this case) appears in different patterns playing different roles. In *SeqP3* the request message models the call started from an ExecutionSpecification. However, in *SeqP1* and *SeqP2*, this same request message models the call that starts an ExecutionSpecification. The former way of looking at the request message is translated into var:nestedRequest (in *SeqP3*), and the latter is translated into var:starter (in *SeqP1* and *SeqP2*). Consequently, despite var:nestedRequest and var:starter being two different elements of type prov:Entity appearing in two different PROV templates, both must be assigned to the same value during the execution of the application. Therefore, after merging all the expanded PROV templates, a single prov:Entity will be generated.

Identifier *Sequence* diagram *Pattern 4* (*SeqP4*)

Context

During the execution of an operation (main operation), a response of a previously issued nested operation call is received. The main operation's execution uses this response to complete its behaviour. This way, this pattern complements *SeqP1* and *SeqP2* with additional information regarding the response to the *nested operation call* (addressed by *SeqP3*).

Key elements

(Main) *Operation execution*	The execution of the main operation.
(Nested) *Response*	The response to a nested operation call.
(Main) *Response*	The response of the *Main operation execution*. This element is only identified when this pattern complements *SeqP2*.

UML Diagram

Key Element	UML	Rationale
Main operation execution	`ExecutionSpecification` 1	It shows the period of time that takes the *Main operation execution*.
Nested response	`Reply Message` 2	It specifies the response received in the *Main operation execution*.
Main response	`Reply Message` 3	In case of complementing *SeqP2*, it specifies the response of the *Main operation execution*.

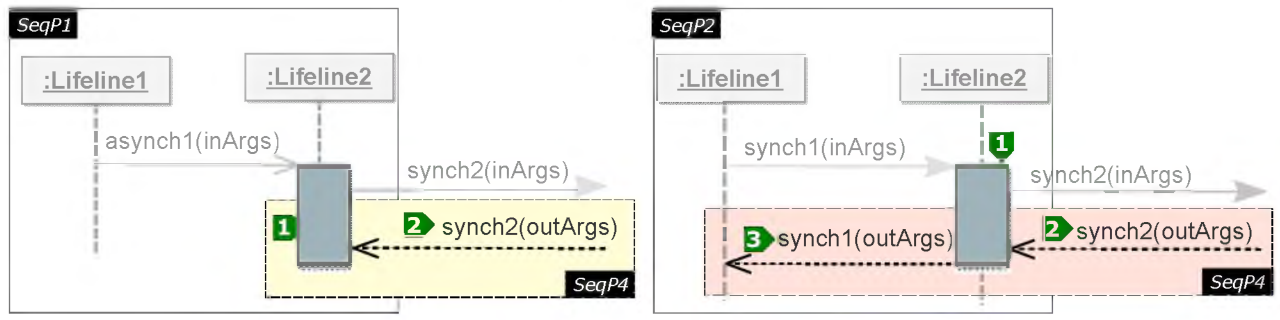
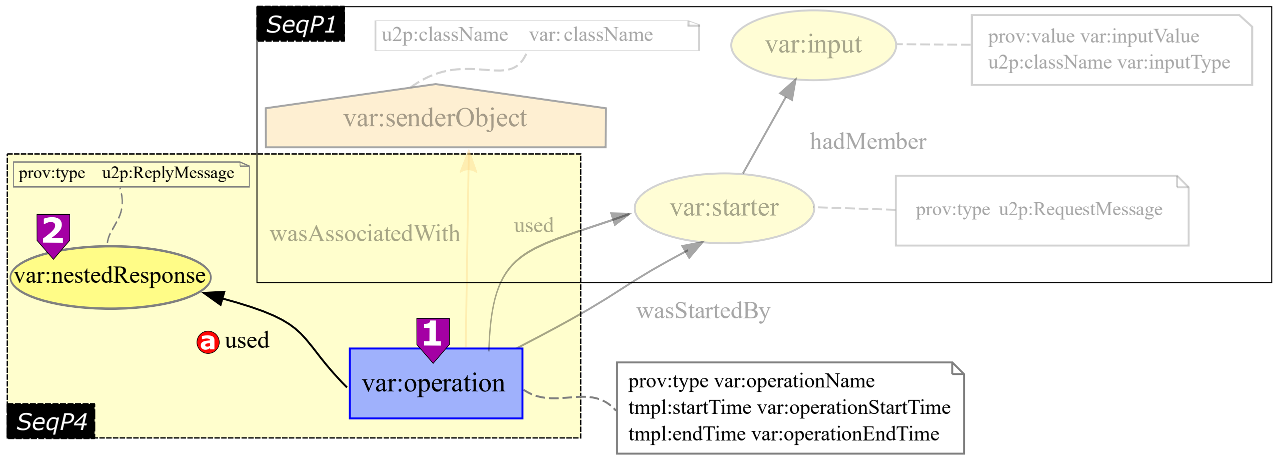


Figure 7. The left hand side is the UML representation that models the context given by *SeqP1* complemented by *SeqP4*, wheres the right hand side is the UML representation that models the context given by *SeqP2* complemented by *SeqP4*. Only the shaded areas correspond to the UML elements contributed by this pattern.


SeqP1
u2p:className var: className
var:senderObject
var:input
prov:value var:inputValue
u2p:className var:inputType
hadMember
var:starter
prov:type u2p:RequestMessage
prov:type u2p:ReplyMessage
2
var:nestedResponse
wasAssociatedWith
used
wasStartedBy
a used
1
var:operation
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
SeqP4

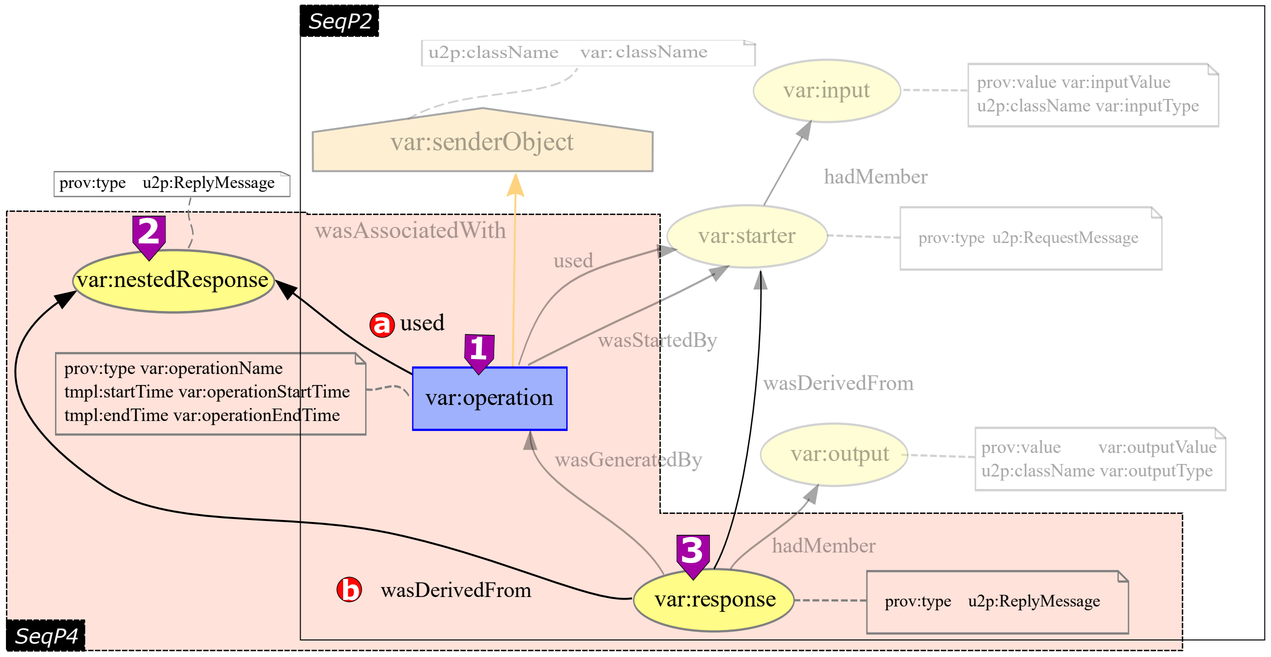

SeqP2
u2p:className var: className
var:senderObject
var:input
prov:value var:inputValue
u2p:className var:inputType
hadMember
prov:type u2p:ReplyMessage
wasAssociatedWith
var:starter
prov:type u2p:RequestMessage
2
var:nestedResponse
used
a used
wasStartedBy
1
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
var:operation
wasDerivedFrom
wasGeneratedBy
var:output
prov:value var:outputValue
u2p:className var:outputType
hadMember
3
b wasDerivedFrom
var:response
prov:type u2p:ReplyMessage
SeqP4

Figure 8. At the top, it is the PROV template generated from the UML representation in the left side of Figure 7. At the bottom, it is a PROV template generated from the UML representation in the right side of Figure 7. Only the shaded areas correspond to the PROV elements contributed by this pattern.

PROV elements

UML	PROV / id	Rationale
`ExecutionSpecification` 1▶	`prov:Activity` 1▶ / `var:operation`	The `ExecutionSpecification` 1▶ is a `prov:Activity` with identifier `var:operation`.
`Reply Message` 2▶	`prov:Entity` 2▶ / `var:nestedResponse`	The `Reply Message` 2▶ that is received in the `ExecutionSpecification` 1▶ is a `prov:Entity` with identifier `var:nestedResponse`.
`Reply Message` 3▶	`prov:Entity` 3▶ / `var:response`	In case of complementing *SeqP2*, the `Reply Message` 3▶ sent from the `ExecutionSpecification` 1▶ is a `prov:Entity` with identifier `var:response`. For details, see *SeqP2*.

Attributes

PROV Element	Attribute / Value	Description
`var:operation` 1▶	`prov:type` / `var:operationName`	The value `var:operationName` is the name of the operation `var:operation` 1▶.
	`tmpl:startTime` / `var:operationStartTime`	The `var:operationStartTime` is an `xsd:dateTime` value for the start of `var:operation` 1▶.
	`tmpl:endTime` / `var:operationEndTime`	The `var:operationEndTime` is an `xsd:dateTime` value for the end of `var:operation` 1▶.
`var:nestedResponse` 2▶	`prov:type` / `u2p:ReplyMessage`	The value `u2p:ReplyMessage` shows that `var:response` 2▶ is a reply message.
`var:response` 3▶	`prov:type` / `u2p:ReplyMessage`	The value `u2p:ReplyMessage` shows that `var:response` 3▶ is a reply message.

PROV relations

PROV Relation	Description
ⓐ `prov:used`	It is the beginning of utilizing `var:nestedResponse` by `var:operation`.
ⓑ `prov:wasDerivedFrom`	It is the construction of `var:response` based on `var:nestedResponse`.

Discussion

- As it could be inferred from the context, a requirement for this pattern to be applied is that the *Main operation execution* uses the *Nested response* during its execution. This causes the relations ⓐ `prov:used` and ⓑ `prov:wasDerivedFrom` to appear in the template; the former showing that when the `ExecutionSpecification` 1▶ receives the nested `Reply Message` 2▶, it *utilises* that `Reply Message` 2▶ to complete its behaviour; and the latter showing that the main `Reply Message` 3▶ is *influenced* by the nested `Reply Message` 2▶ (this last one can only be applied if the *Main operation execution* is triggered by a synchronous message, i.e. when *SeqP4* complements *SeqP2*). If a specific scenario does not meet the aforementioned requirement, i.e., the *Main operation execution* does not use the *Nested response* or it is not worth recording such a dependency, this pattern should not be applied. Even in this case, the provenance about the nested operation call and its corresponding response would be captured thanks to *SeqP1* and *SeqP2*, respectively.

5 UML State Machine Diagrams

Pattern identifier	**Context**	**Page**
StP1	As a consequence of the execution of an operation, an object is created in its first state. This operation is usually the constructor of the object.	19
StP2	As a consequence of the execution of an operation, the behaviour of an object is completed.	23
StP3	As a consequence of the execution of an operation, an object changes its state.	27

Identifier *State machine diagram Pattern 1 (StP1)*

Context

As a consequence of the execution of an operation, an object is created in its first state. This operation is usually the constructor of the object.

Key elements

Object	The object created as a consequence of the execution of the operation.	
	First object's state	The first state after the object creation. This is the first state the object may undergo during its lifetime.
Object creation	The execution of the operation that creates the object.	

UML Diagram

Key Element	UML	Rationale
Object	`Object` 1	It represents the created object. *Note*: since `Object` lacks a graphical representation in UML State Machine diagrams, Figure 9 does not depict this element.
	`StateMachine` 2	In UML, a `StateMachine` represents the set of states an *Object* can go through during its lifetime in response to events.
Object creation	`Initial Pseudostate` 3	It refers to the execution of the operation that creates the *Object*, leading it to its first state.
First object's state	`State` 4	It models the first state of the *Object*.

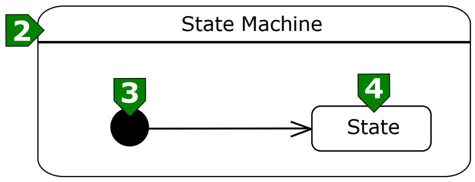


Figure 9. UML representation that models the context given by *StP1*

Mapping to PROV

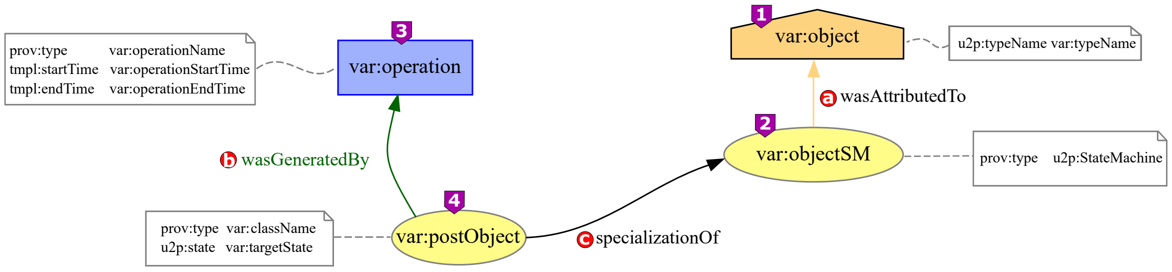

prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
3
var:operation
1
var:object
u2p:typeName var:typeName
a wasAttributedTo
b wasGeneratedBy
2
var:objectSM
prov:type u2p:StateMachine
prov:type var:className
u2p:state var:targetState
4
var:postObject
c specializationOf

Figure 10. PROV template generated from the UML representation used in *StP1* (Figure 9)

PROV elements

UML	PROV / id	Rationale
`Object` 1▶	`prov:Agent` 1▶ / `var:object`	The `Object` 1▶ bears some form of responsibility for the existence of the `StateMachine` 2▶, since the existence of `StateMachine` 2▶ does not make sense without an `Object` 1▶. To reflect this fact, the `Object` 1▶ is mapped to a `prov:Agent` identified by `var:object`.
`StateMachine` 2▶	`prov:Entity` 2▶ / `var:objectSM`	The `StateMachine` 2▶ is a `prov:Entity` identified by `var:objectSM`. It reflects the abstraction of the object's states, which will be specialized by each concrete state the object goes through.
`Initial Pseudostate` 3▶	`prov:Activity` 3▶ / `var:operation`	The `Initial Pseudostate` 3▶, referring to the execution of the operation that creates the `Object` 1▶, is a `prov:Activity` with the identifier `var:operation`.
`State` 4▶	`prov:Entity` 4▶ / `var:postObject`	The `State` 4▶ is a `prov:Entity` identified by `var:postObject`. We use this name for this identifier because it corresponds to the state of the `Object` 1▶ after (post) the object creation.

Attributes

PROV Element	Attribute / Value	Description
`var:object` 1▶	`u2p:typeName` / `var:className`	The value `var:className` is the string with the name of the class to which `var:object` 1▶ belongs.
`var:objectSM` 2▶	`prov:type` / `u2p:StateMachine`	The value `u2p:StateMachine` shows that `var:objectSM` 2▶ is a state machine.
`var:operation` 3▶	`prov:type` / `var:operationName`	The value `var:operationName` is the name of the operation `var:operation` 3▶
	`tmpl:startTime` / `var:operationStartTime`	The `var:operationStartTime` is an `xsd:dateTime` value for the start of `var:operation` 3▶.
	`tmpl:endTime` / `var:operationEndTime`	The `var:operationEndTime` is an `xsd:dateTime` value for the end of `var:operation` 3▶.
`var:postObject` 4▶	`prov:type` / `var:className`	The value `var:className` is the name of the class to which the object in the state `var:postObject` 4▶ belongs.
	`u2p:state` / `var:targetState`	The value `var:targetState` is the string with the name of the state `var:postObject` 4▶.

PROV relations

PROV Relation	Description
ⓐ `prov:wasAttributedTo`	It is the assignment of responsibility to `var:object` for `var:objectSM`.
ⓑ `prov:wasGeneratedBy`	It is the completion of production of `var:postObject` by `var:operation`.
ⓒ `prov:specializationOf`	`var:postObject` is a specialization of `var:objectSM`.

Discussion

- Note that Figure 9 only contains simple states. We do not deal with composite or submachine states, and focus only on simple states, because the former may be transformed into the latter by resorting to a flattening process consisting of removing composite states as well as submachine states. In fact, to flatten State Machine diagrams is a very common approach in contexts such as model checking and code generation [6]. However, the user might be interested in representing composite states directly into the PROV templates, perhaps because she/he is interested in collecting information about them, or just because she/he does not want to flatten the State Machine diagram. We can give an insight into how composite states can be mapped to PROV by placing the elements from Figure 9 inside a `Composite State` 5▶ (see Figure 11). A reader familiar with the UML specification will realize that the semantics of the `Initial Pseudostate` 3▶ in Figures 9 and 11 are different, but these semantic nuances would have no effect on the PROV transformation. The transformation of Figure 11 is shown in Figure 12. Both Figure 11 and 12 highlight the added elements by blurring the elements coming from Figure 9 and Figure 10, respectively. Briefly speaking, the new `Composite State` 5 is translated into a `prov:Entity` identified by `var:compState` 5, which is associated with `var:objectSM` 2 and `var:targetState` 4 by means of the relations d `prov:specializationOf` and e `prov:hadMember`, respectively. At this point, it is also worth remarking that for this example we have used a *simple composite state* (i.e., `Composite State` 5), which means that only one substate is active at a given time within such a state; but we could have used *orthogonal composite states* instead, which means that within such a state several substates are active at the same time. Note that both types of *composite states* would be translated into the same PROV template (see Figure 12); nevertheless, the generated bindings would be different. In case of a *simple composite state*, as there can be only one active substate at the same time, there would be only one value associated with the variable `var:postObject` 4. Conversely, in case of an *orthogonal composite state*, `var:postObject` 4 will be associated with several values (as many as active states).

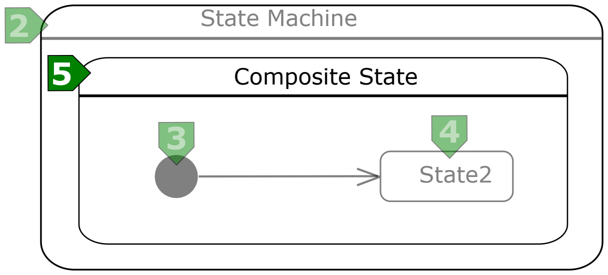
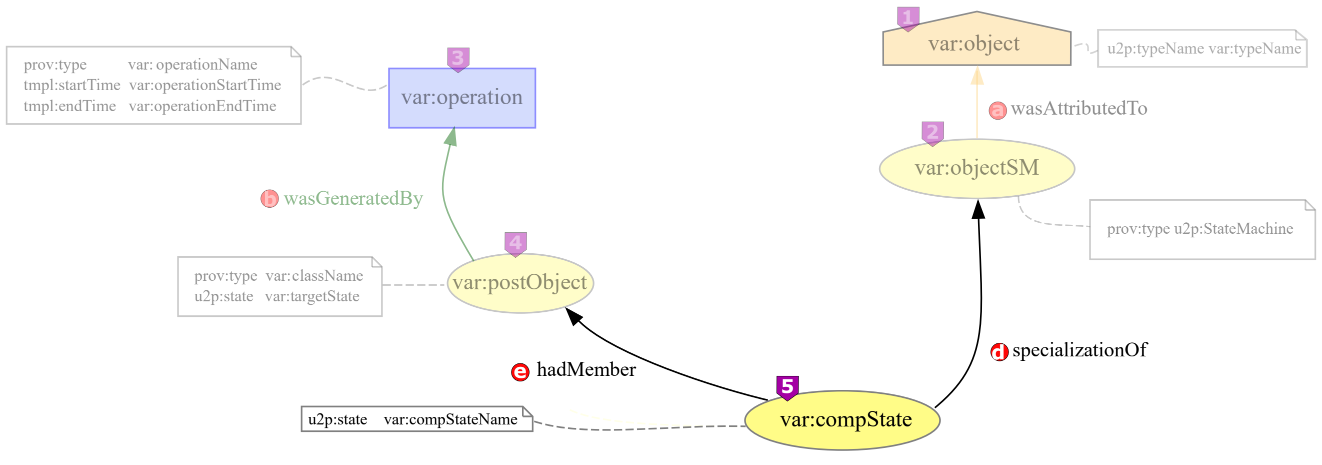


Figure 11. Excerpt of a UML State Machine diagram locating the UML elements from *StP1* in a *simple composite state*.



prov:type var: operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
3
var:operation
1
var:object
u2p:typeName var:typeName
a wasAttributedTo
2
var:objectSM
prov:type u2p:StateMachine
b wasGeneratedBy
4
prov:type var:className
u2p:state var:targetState
var:postObject
e hadMember
d specializationOf
5
u2p:state var:compStateName
var:compState

Figure 12. PROV template generated from the UML diagram in Figure 11

PROV elements

UML	PROV / id	Rationale
`Composite State` 5	`prov:Entity` 5 / `var:compState`	The `Composite State` 5 is a `prov:Entity` identified by `var:compState`.

Attributes

PROV Element	Attribute / Value	Description
`var:compState` 5	`u2p:state` / `var:compStateName`	The value `var:compStateName` is the string with the name of the state `var:compState` 5

PROV relations

PROV Relation	**Description**
ⓓ `prov:specializationOf`	`var:compState` is a specialization of `var:objectSM`.
ⓔ `prov:hadMember`	It states that `var:postObject` is one of the elements in `var:compState`.

Identifier *State machine diagram Pattern 2* (*StP2*)

Context

As a consequence of the execution of an operation, the behaviour of an object is completed.

Key elements

Object	The object that completes its behaviour.	
	Pre-operation object's state	The state of the object before the execution of the operation. This is one of the states the object may undergo during its lifetime.
	Final object's state	The state that represents that the object's behaviour is completed.
Operation execution	The execution of the operation that leads the object to complete its behaviour.	

UML Diagram

Key Element	UML	Rationale
Object	`Object` 1	It represents the object whose behaviour is completed. *Note*: since `Object` lacks a graphical representation in UML State Machine diagrams, Figure 13 does not depict this element.
	`StateMachine` 2	In UML, a `StateMachine` can be used to express the set of states through which the *Object* goes during its lifetime in response to events.
Pre-operation object's state	`State` 3	It models the state of the *Object* before the *Operation execution*.
Final object's state	`FinalState` 4	It models the state of the *Object* after the *Operation execution*.
Operation execution	`Event` 5	It specifies that the *Operation execution* that triggers the change in the *Object*'s state has taken place.

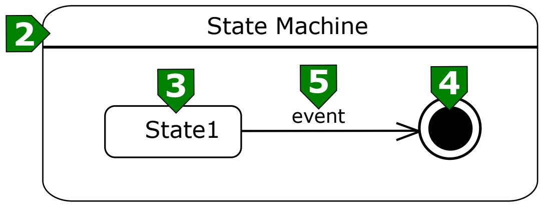


Figure 13. UML representation that models the context given by *StP2*

Mapping to PROV

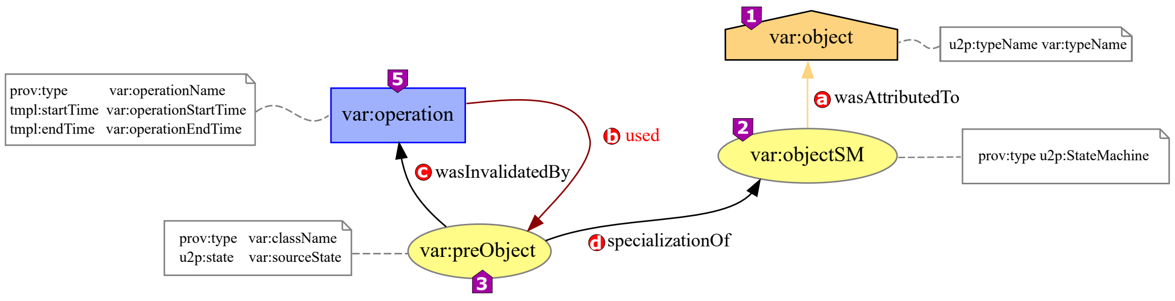

1
var:object
u2p:typeName var:typeName
a wasAttributedTo
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
5
var:operation
b used
2
var:objectSM
prov:type u2p:StateMachine
c wasInvalidatedBy
prov:type var:className
u2p:state var:sourceState
var:preObject
3
d specializationOf

Figure 14. PROV template generated from the UML representation used in *StP2* (Figure 13)

PROV elements

UML	PROV / id	Rationale
`Object` 1▶	`prov:Agent` 1▶ / `var:object`	The `Object` 1▶ bears some form of responsibility for the existence of the `StateMachine` 2▶, since the existence of `StateMachine` 2▶ does not make sense without an `Object` 1▶. To reflect this fact, the `Object` 1▶ is mapped to a `prov:Agent` identified by `var:object`.
`StateMachine` 2▶	`prov:Entity` 2▶ / `var:objectSM`	The `StateMachine` 2▶ is a `prov:Entity` identified by `var:objectSM`. It reflects the abstraction of the object's states, which will be specialized by each state the object goes through.
`State` 3▶	`prov:Entity` 3▶ / `var:preObject`	The `State` 3▶ is a `prov:Entity` identified by `var:preObject`. We use this name for this identifier because it corresponds to the state of the `Object` 1▶ before (pre) the execution of the operation.
`FinalState` 4▶	None /	Without mapping (see the discussion block for an explanation about this decision).
`Event` 5▶	`prov:Activity` 5▶ / `var:operation`	The `Event` 5▶ represents that the execution of an operation has taken place. Such an execution is a `prov:Activity` with the identifier `var:operation`.

Attributes

PROV Element	Attribute / Value	Description
`var:object` 1▶	`u2p:typeName` / `var:className`	The value `var:className` is the string with the name of the class to which `var:object` 1▶ belongs.
`var:objectSM` 2▶	`prov:type` / `u2p:StateMachine`	The value `u2p:StateMachine` shows that `var:objectSM` 2▶ is a state machine.
`var:preObject` 3▶	`prov:type` / `var:className`	The value `var:className` is the string with the name of the class to which the object in the state `var:preObject` 3▶ belongs.
	`u2p:state` / `var:sourceState`	The value `var:sourceState` is the string with the name of the state `var:preObject` 3▶.
`var:operation` 5▶	`prov:type` / `var:operationName`	The value `var:operationName` is the name of the operation `var:operation` 5▶.
	`tmpl:startTime` / `var:operationStartTime`	The `var:operationStartTime` is an `xsd:dateTime` value for the start of `var:operation` 5▶.
	`tmpl:endTime` / `var:operationEndTime`	The `var:operationEndTime` is an `xsd:dateTime` value for the end of `var:operation` 5▶.

PROV relations

PROV Relation	Description
ⓐ `prov:wasAttributedTo`	It is the assignment of responsibility to `var:object` for `var:objectSM`.
ⓑ `prov:used`	It is the beginning of utilizing `var:preObject` by `var:operation`.
ⓒ `prov:wasInvalidatedBy`	It shows that `var:preObject` is not longer available for use.
ⓓ `prov:specializationOf`	`var:preObject` is a specialization of `var:objectSM`.

Discussion

- This pattern is consistent with *ClP2* because the completion of the object's behaviour usually involves its destruction. Among the different reasons why an *Object* can complete its behaviour, we can distinguish its destruction, from the remainder cases. In order to be consistent with *ClP2* (that addresses the execution of an operation which provokes the destruction of an object), we have decided not to explicitly map the `FinalState` 4▶ in the PROV template but including its semantics (the completion of the object's behaviour) by the relation ⓒ `prov:wasInvalidatedBy` showing that `var:preObject` 3▶ is not longer available. However, if the user is interested in explicitly representing the `FinalState` 4 into the PROV templates, we refer him/her to *StP3*, where the state of the *Object* before and after an *Operation execution* is included.

- Figure 13 only contains simple states. We do not deal with composite or submachine states, and focus only on simple states, because the former may be transformed into the latter by resorting to a flattening process consisting of removing composite states as well as submachine states. In fact, to flatten State Machine diagrams is a very common approach in contexts such as model checking and code generation [6]. However, the user might be interested in representing composite states directly into the PROV templates, perhaps because she/he is interested in collecting information about them, or just because she/he does not want to flatten the State Machine diagram. We can give an insight into how composite states can be mapped to PROV by placing the elements from Figure 13 inside a `Composite State` 5 (see Figure 15). A reader familiar with the UML specification will realize that the semantics of the `FinalState` 4 in Figures 13 and 15 are different, but these semantic nuances would have no effect on the PROV transformation. The transformation of Figure 15 is shown in Figure 16. Both Figure 15 and 16 highlight the added elements by blurring the elements coming from Figure 13 and Figure 14, respectively. Briefly speaking, the new `Composite State` 6 is translated into a `prov:Entity` identified by `var:compState` 6, which is associated with `var:objectSM` 2 and `var:preObject` 3 by means of the relations ⓕ `prov:specializationOf` and ⓔ `prov:hadMember`, respectively. At this point, it is also worth remarking that for this example we have used a *simple composite state* (i.e., `Composite State` 6), which means that only one substate is active at a given time within such a state; but we could have used *orthogonal composite states* instead, which means that within such a state several substates are active at the same time. Note that both types of *composite states* would be translated into the same PROV template (see Figure 16); nevertheless, the generated bindings would be different. In case of a *simple composite state*, as there can be only one active substate at the same time, there would be only one value associated with the variable `var:preObject` 3. Conversely, in case of an *orthogonal composite state*, `var:preObject` 3 will be associated with several values (as many as active states).

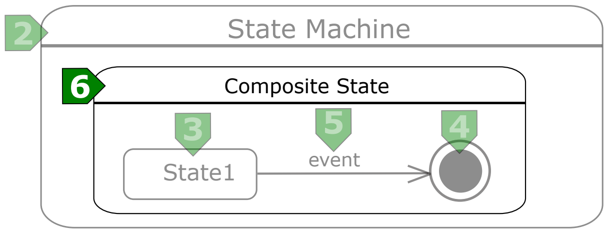


Figure 15. Excerpt of a UML State Machine diagram locating the UML elements from *StP2* in a *simple composite state*.

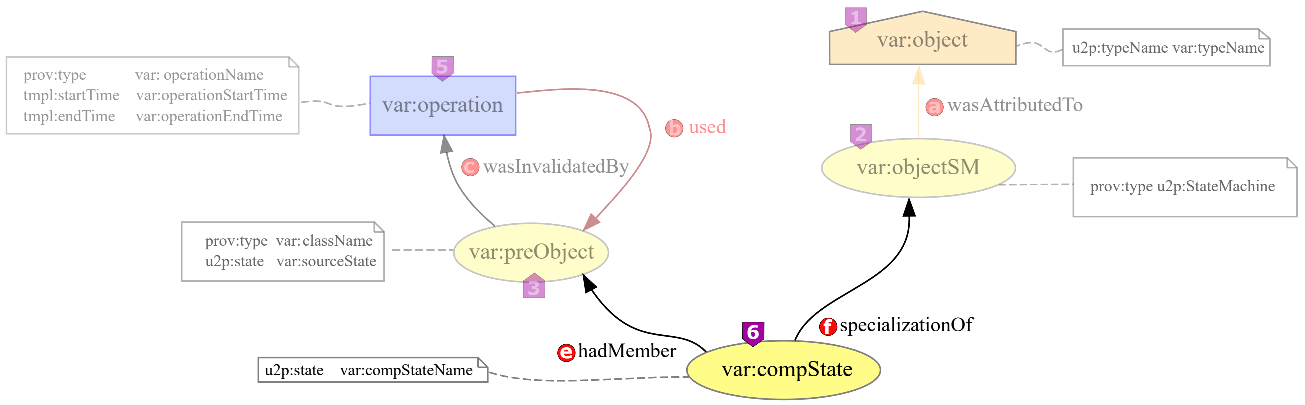

1
var:object
u2p:typeName var:typeName
ⓐ wasAttributedTo
2
var:objectSM
prov:type u2p:StateMachine
prov:type var: operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
5
var:operation
ⓑ used
ⓒ wasInvalidatedBy
prov:type var:className
u2p:state var:sourceState
var:preObject
3
ⓔ hadMember
6
ⓕ specializationOf
u2p:state var:compStateName
var:compState

Figure 16. PROV template generated from the UML diagram in Figure 15

PROV elements

UML	PROV / id	Rationale
`Composite State` 6	`prov:Entity` 6 / `var:compState`	The `Composite State` 6 is a `prov:Entity` identified by `var:compState`.

Attributes

PROV Element	Attribute / Value	Description
`var:compState` 6	`u2p:state` / `var:compStateName`	The value `var:compStateName` is the string with the name of the state `var:compState` 6

PROV relations

PROV Relation	Description
e `prov:hadMember`	It states that `var:preObject` is one of the elements in `var:compState`.
f `prov:specializationOf`	`var:compState` is a specialization of `var:objectSM`.

Identifier *State machine diagram Pattern 3* (*StP3*)

Context

As a consequence of the execution of an operation, an object changes its state.

Key elements

Object	The object that changes its state.	
	Pre-operation object's state	The state of the object before the execution of the operation. This is one of the states the object may undergo during its lifeline.
	Post-operation object's state	The state of the object after the execution of the operation. This is one of the states the object may undergo during its lifeline.
Operation execution	The execution of the operation that leads a change in the *Object*'s state.	

UML Diagram

Key Element	UML	Rationale
Object	`Object` 1	It represents the object that changes its state. *Note*: since `Object` lacks a graphical representation in UML State Machine diagrams, Figure 17 does not depict this element.
	`StateMachine` 2	In UML, a `StateMachine` can be used to express the set of object's states through which the *Object* goes during its lifetime in response to events.
Pre-operation object's state	`State` 3	It models the state of the *Object* before the *Operation execution*.
Post-operation object's state	`State` 4	It models the state of the *Object* after the *Operation execution*.
Operation execution	`Event` 5	It specifies that the *Operation execution* that triggers the change in the *Object*'s state has taken place.

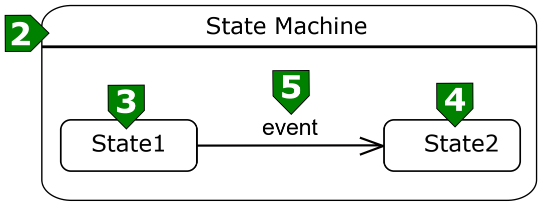


Figure 17. UML representation that models the context given by *StP3*

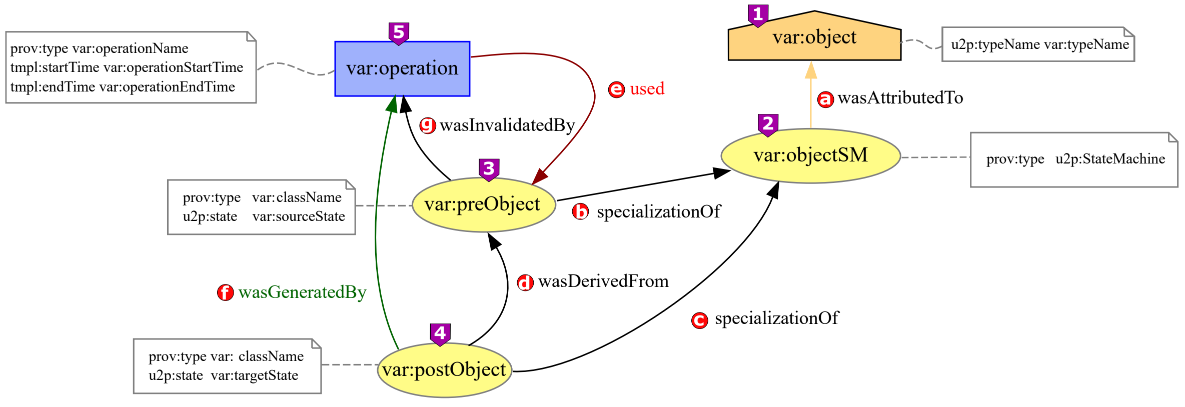


Figure 18. PROV template generated from the UML representation used in *StP3* (Figure 17)

PROV elements

UML	PROV / id	Rationale
`Object` 1	`prov:Agent` 1 / `var:object`	The `Object` 1 bears some form of responsibility for the existence of the `StateMachine` 2, since the existence of `StateMachine` 2 does not make sense without an `Object` 1. To reflect this fact, the `Object` 1 is mapped to a `prov:Agent` identified by `var:object`.
`StateMachine` 2	`prov:Entity` 2 / `var:objectSM`	The `StateMachine` 2 is a `prov:Entity` identified by `var:objectSM`. It reflects the abstraction of the object's states, which will be specialized by each state the object goes through.
`State` 3	`prov:Entity` 3 / `var:preObject`	The `State` 3 is a `prov:Entity` identified by `var:preObject`. We use this name for this identifier because it corresponds to the state of the `Object` 1 before (pre) the execution of the operation.
`State` 4	`prov:Entity` 4 / `var:postObject`	The `State` 4 is a `prov:Entity` identified by `var:postObject`. We use this name for this identifier because it corresponds to the state of the `Object` 1 after (post) the execution of the operation.
`Event` 5	`prov:Activity` 5 / `var:operation`	The `Event` 5 represents that the execution of an operation has taken place. Such an execution is a `prov:Activity` with the identifier `var:operation`.

Attributes

PROV Element	Attribute / Value	Description
var:object 1▶	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:object 1▶ belongs.
var:objectSM 2▶	prov:type / u2p:StateMachine	The value u2p:StateMachine shows that var:objectSM 2▶ is a state machine.
var:preObject 3▶	prov:type / var:className	The value var:className is the name of the class to which the object in the state var:preObject 3▶ belongs.
	u2p:state / var:sourceState	The value var:sourceState is the string with the name of the state var:preObject 3▶.
var:postObject 4▶	prov:type / var:className	The value var:className is the name of the class to which the object in the state var:postObject 4▶ belongs.
	u2p:state / var:targetState	The value var:targetState is the string with the name of the state var:postObject 4▶.
var:operation 5▶	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 5▶.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 5▶.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 5▶.

PROV relations

PROV Relation	Description
ⓐ prov:wasAttributedTo	It is the assignment of responsibility to var:object for var:objectSM.
ⓑ prov:specializationOf	var:preObject is a specialization of var:objectSM.
ⓒ prov:specializationOf	var:postObject is a specialization of var:objectSM.
ⓓ prov:wasDerivedFrom	It is the update of var:preObject resulting in var:postObject.
ⓔ prov:used	It is the beginning of utilizing var:preObject by var:operation.
ⓕ prov:wasGeneratedBy	It is the completion of production of var:postObject by var:operation.
ⓖ prov:wasInvalidatedBy	It shows that var:preObject is not longer available for use.

Discussion

- Figure 17 only contains simple states. We do not deal with composite or submachine states, and focus only on simple states, because the former may be transformed into the latter by resorting to a flattening process consisting of removing composite states as well as submachine states. In fact, to flatten State Machine diagrams is a very common approach in contexts such as model checking and code generation [6]. However, the user might be interested in representing composite states directly into the PROV templates, perhaps because she/he is interested in collecting information about them, or just because she/he does not want to flatten the State Machine diagram. We can give an insight into how composite states can be mapped to PROV by placing the elements from Figure 17 inside a Composite State 5▶ (see Figure 19). A reader familiar with the UML specification will realize that the semantics of the UML representation in Figures 17 and 19 are different, but these semantic nuances would have no effect on the PROV transformation. The transformation of Figure 19 is shown in Figure 20. Both Figure 19 and 20 highlight the added elements by blurring the elements coming from Figure 17 and Figure 18, respectively. Briefly speaking, the new Composite State 6▶ is translated into a prov:Entity identified by var:compState 6▶, which is associated with var:objectSM 2▶, var:preObject 3▶, and var:postObject 4▶ by means of the relations ⓙ prov:specializationOf, ⓗ prov:hadMember, and ⓘ prov:hadMember, respectively. At this point, it is also worth remarking that for this example we have used a *simple composite state* (i.e., Composite State 6▶), which means that only one substate is active at a given time within such a state; but we could have used *orthogonal composite states* instead, which means that within such a state several substates are active at the same time. Note that both types of *composite states* would be translated into the same PROV template (see Figure 20); nevertheless, the generated bindings would be different. In case of a *simple composite state*, as there can be only one active substate at the same time, there would be only one value associated with the variable `var:preObject` 3 and another value with `var:postObject` 4. Conversely, in case of an *orthogonal composite state*, `var:preObject` 3 and `var:postObject` 4 will be associated with several values (as many as active states).

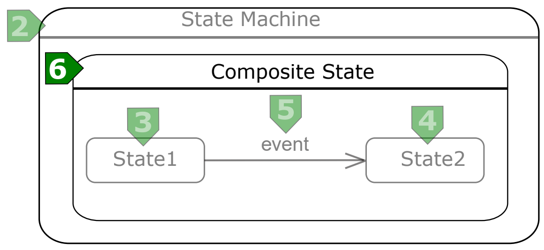


Figure 19. Excerpt of a UML State Machine diagram locating the UML elements from *StP3* in a *simple composite state*.

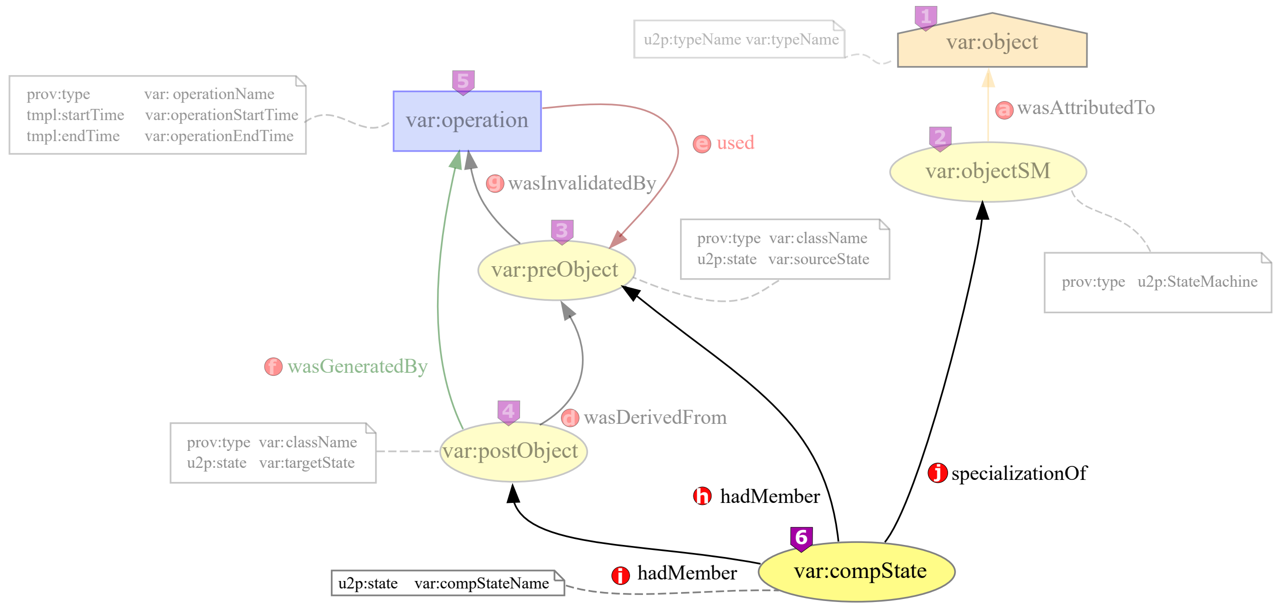


Figure 20. PROV template generated from the UML diagram in Figure 19

PROV elements

UML	PROV / id	Rationale
`Composite State` 6	`prov:Entity` 6 / `var:compState`	The `Composite State` 6 is a `prov:Entity` identified by `var:compState`.

Attributes

PROV Element	Attribute / Value	Description
`var:compState` 6	`u2p:state` / `var:compStateName`	The value `var:compStateName` is the string with the name of the state `var:compState` 6

PROV relations

PROV Relation	Description
ⓗ prov:hadMember	It states that var:preObject is one of the elements in var:compState.
ⓘ prov:hadMember	It states that var:postObject is one of the elements in var:compState.
ⓙ prov:specializationOf	var:compState is a specialization of var:objectSM.

6 UML Class Diagrams

Pattern identifier	Context	Page
ClP1	The execution of an operation provokes the creation of a new object.	33
ClP2	The execution of an operation provokes the destruction of an object.	36
ClP3	The execution of an operation on an object returns values of concrete object's attributes. The values are returned as they are, without any further processing. This execution does not provoke the change of the object's status.	38
ClP4	The execution of an operation on an object returns values that are computed based on the object's status as a whole (the values of concrete attributes involved in the computation are unknown or irrelevant). This execution does not provoke the change of the object's status.	41
ClP5	The execution of an operation on an object returns values that are computed based on values of concrete object's attributes. This execution does not provoke the change of the object's status.	44
ClP6	The execution of an operation on an object changes the object's status as a whole (the concrete modified attributes are unknown or irrelevant).	48
ClP7	The execution of an operation on an object directly sets the information passed to the operation as values of concrete object's attributes, thus provoking a change in the object's status.	53
ClP8	The execution of an operation on an object changes the values of concrete object's attributes, thus provoking a change in the object's status.	58
ClP9	The execution of an operation on an object removes element(s) from a concrete object's collection attribute, thus provoking a change in the object's status.	63
ClP10	The execution of an operation on an object directly adds the information passed to the operation as new element(s) of a concrete object's collection attribute, thus provoking a change in the object's status.	68

Identifier *Class* diagram *Pattern 1* (*ClP1*)

Context

The execution of an operation provokes the creation of a new object.

Key elements

Object	The object created as a consequence of the execution of the operation.
Operation execution	The execution of the operation.
Input data	The information (if any) passed into the *Operation execution*.
Object's attributes	The characteristics of the *Object*.

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the *Object* after the execution of the operation.
Operation execution	`Operation` 2 `«create»`	The `Operation` 2 stereotyped by `«create»` represents the executed operation that creates the *Object*.
Input data	`Input Parameters` 3	They specify the information passed into the *Operation execution*.
Object's attributes	`Attributes` 4	They represent the characteristics of the *Object*.

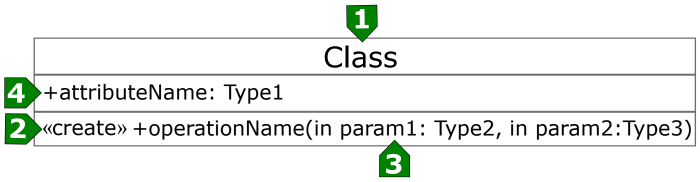


Figure 21. UML representation that models the context given by *ClP1*

Mapping to PROV

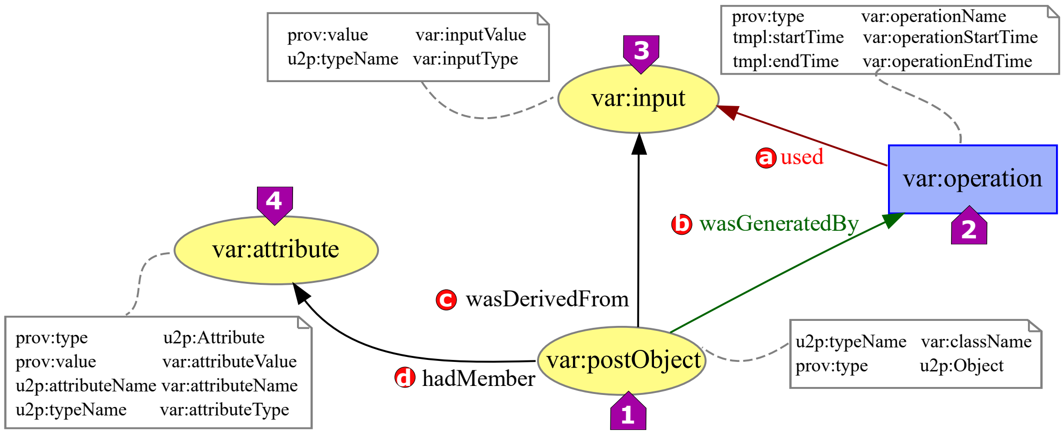

prov:value var:inputValue
u2p:typeName var:inputType
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
3
var:input
a used
var:operation
b wasGeneratedBy
2
4
var:attribute
c wasDerivedFrom
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:typeName var:attributeType
d hadMember
var:postObject
1
u2p:typeName var:className
prov:type u2p:Object

Figure 22. PROV template generated from the UML representation used in *ClP1* (Figure 21)

PROV elements

UML	PROV / id	Rationale
`Class` 1▶	`prov:Entity` 1▶ / `var:postObject`	The `Class` 1▶ that models the object that is created by the operation is a `prov:Entity` identified as `var:postObject`. We use the prefix *post* in this identifier because the object is the result of the executed operation.
`Operation` 2▶ `«create»`	`prov:Activity` 2▶ / `var:operation`	The execution of `Operation` 2▶ stereotyped by `«create»` is a `prov:Activity` identified by `var:operation`.
`Input Parameters` 3▶	`prov:Entity` 3▶ / `var:input`	Each parameter of `Input Parameters` 3▶ is a separate `prov:Entity` identified as `var:input`.
`Attributes` 4▶	`prov:Entity` 4▶ / `var:attribute`	Each attribute of `Attributes` 4▶ is a separate `prov:Entity` with identifier `var:attribute`.

Attributes

PROV Element	Attribute / Value	Description
`var:postObject` 1▶	`u2p:typeName` / `var:className`	The value `var:className` is the string with the name of the class to which `var:postObject` 1▶ belongs.
	`prov:type` / `u2p:Object`	The value `u2p:Object` shows that `var:postObject` is an object.
`var:operation` 2▶	`prov:type` / `var:operationName`	The value `var:operationName` is the name of the operation `var:operation` 2▶.
	`tmpl:startTime` / `var:operationStartTime`	The `var:operationStartTime` is an `xsd:dateTime` value for the start of `var:operation` 2▶.
	`tmpl:endTime` / `var:operationEndTime`	The `var:operationEndTime` is an `xsd:dateTime` value for the end of `var:operation` 2▶.
`var:input` 3▶	`prov:value` / `var:inputValue`	The value `var:inputValue` is the direct representation of `var:input` 3▶.
	`u2p:typeName` / `var:inputType`	The value `var:inputType` is the string with the name of the type of `var:input` 3▶.
`var:attribute` 4▶	`prov:type` / `u2p:Attribute`	The value `u2p:Attribute` shows that `var:attribute` 4▶ is an attribute.
	`prov:value` / `var:attributeValue`	The value `var:attributeValue` is the direct representation of `var:attribute` 4▶.
	`u2p:attributeName` / `var:attributeName`	The value `var:attributeName` is the string with the name of `var:attribute` 4▶.
	`u2p:typeName` / `var:attributeType`	The value `var:attributeType` is the string with the name of the type of `var:attribute` 4▶.

PROV relations

PROV Relation	Description
ⓐ `prov:used`	It is the beginning of utilizing `var:input` by `var:operation`.
ⓑ `prov:wasGeneratedBy`	It is the completion of production of `var:postObject` by `var:operation`.
ⓒ `prov:wasDerivedFrom`	It is the construction of `var:postObject` based on `var:input`.
ⓓ `prov:hadMember`	It states that `var:attribute` is one of the elements in `var:postObject`.

Discussion

- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we have considered this circumstance with the aim of covering a wider spectrum of cases. When the operation that creates the object lacks *Input data*, the UML representation in Figure 21 will not include `Input Parameters` 3. As a consequence, the resulting PROV template in Figure 22 will also lack `var:input` 3 and its associated PROV relations.

Identifier *Class* diagram *Pattern 2* (*ClP2*)

Context

The execution of an operation provokes the destruction of an object.

Key elements

Object	The object destroyed as a consequence of the execution of the operation.
Operation execution	The execution of the operation.

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the destroyed *Object*.
Operation execution	`Operation` 2 «`destroy`»	The `Operation` 2 stereotyped by «`destroy`» represents the executed operation that destroys the *Object*.

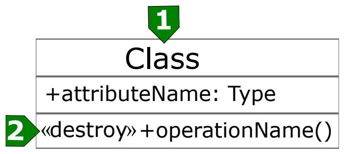
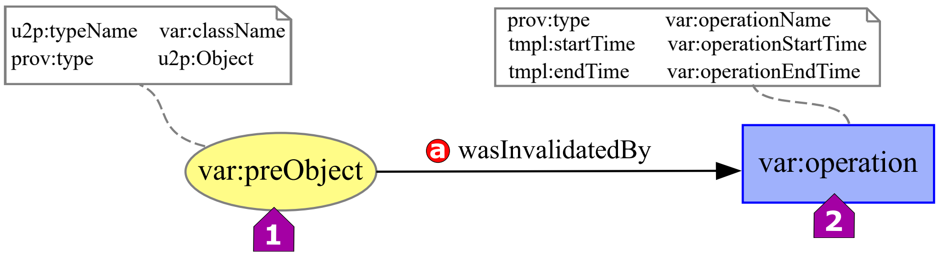


Figure 23. UML representation that models the context given by *ClP2*

Mapping to PROV



u2p:typeName var:className
prov:type u2p:Object
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
var:preObject
a wasInvalidatedBy
var:operation
1
2

Figure 24. PROV template generated from the UML representation used in *ClP2* (Figure 23)

PROV elements

UML	PROV / id	Rationale
`Class` 1	`prov:Entity` 1 / `var:preObject`	The `Class` 1 that models the object that is destroyed by the operation is a `prov:Entity` identified as `var:preObject`. We use the prefix *pre* in this identifier because it is the object before the execution of the operation.
`Operation` 2 «`destroy`»	`prov:Activity` 2 / `var:operation`	The execution of `Operation` 2 stereotyped by «`destroy`» is a `prov:Activity` identified by `var:operation`.

Attributes

PROV Element	Attribute / Value	Description
var:preObject ❶	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:preObject ❶ belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:preObject ❶ is an object.
var:operation ❷	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation ❷.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation ❷.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation ❷.

PROV relations

PROV Relation	Description
ⓐ prov:wasInvalidatedBy	It shows that var:preObject is not longer available for use.

Discussion

- This pattern is consistent with *ClP2* because the completion of the object's behaviour usually involves its destruction.

Identifier *Class* diagram *Pattern 3* (*ClP3*)

Context

The execution of an operation on an object returns values of concrete object's attributes. The values are returned as they are, without any further processing. This execution does not provoke the change of the object's status.

Key elements

Object	The object on which the operation is executed.
Operation execution	The execution of the operation.
Input data	The information (if any) passed into the *Operation execution*.
Output data	The information obtained from the *Operation execution*.

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the *Object* on which the operation is executed.
Operation execution	`Operation` 2 `«get»/«search»`	The `Operation` 2 stereotyped by `«get»/«search»` represents the executed operation. Concretely, operations stereotyped by `«get»` return values of concrete *Object*'s attributes, whereas `«search»` is used when the operation returns elements belonging to a collection attribute of the *Object*.
Input data	`Input Parameters` 3	They specify the information passed into the *Operation execution*.
Output data	`Output Parameters` 4	They depict the information obtained from the *Operation execution*.

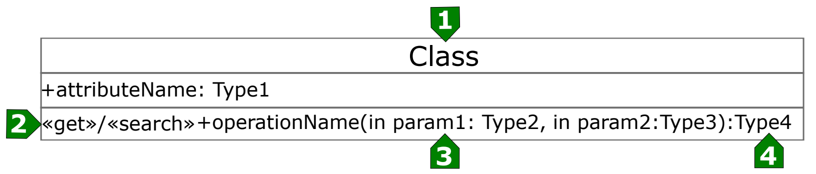

1
Class
+attributeName: Type1
2
«get»/«search»+operationName(in param1: Type2, in param2:Type3):Type4
3
4

Figure 25. UML representation that models the context given by *ClP3*

Mapping to PROV

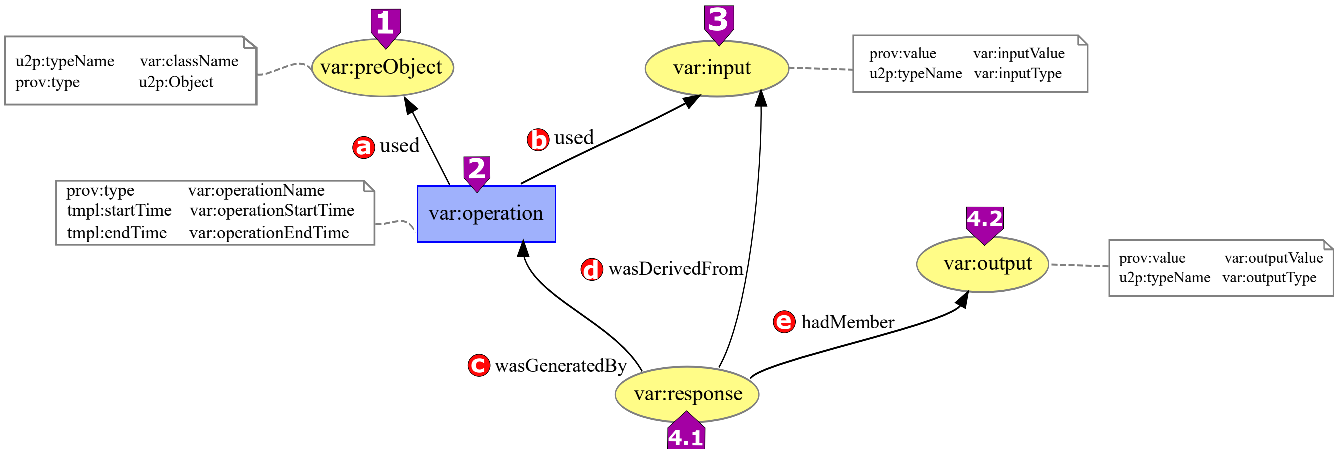

u2p:typeName var:className
prov:type u2p:Object
1
var:preObject
3
var:input
prov:value var:inputValue
u2p:typeName var:inputType
a used
b used
2
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
var:operation
4.2
var:output
prov:value var:outputValue
u2p:typeName var:outputType
d wasDerivedFrom
e hadMember
c wasGeneratedBY
var:response
4.1

Figure 26. PROV template generated from the UML representation used in *ClP3* (Figure 25)

PROV elements

UML	PROV / id	Rationale
`Class` 1	`prov:Entity` 1 / `var:preObject`	The `Class` 1 that models the object on which the operation is executed is a `prov:Entity` identified as `var:preObject`. We use the prefix *pre* in this identifier because it is the object before the execution of the operation.
`Operation` 2 `«get»/«search»`	`prov:Activity` 2 / `var:operation`	The execution of `Operation` 2 stereotyped by `«get»/«search»` is a `prov:Activity` identified by `var:operation`.
`Input Parameters` 3	`prov:Entity` 3 / `var:input`	Each parameter of `Input Parameters` 3 is a separate `prov:Entity` identified as `var:input`.
`Output Parameters` 4	`prov:Entity` 4.1 / `var:response`	The information obtained by the execution of the operation is a `prov:Entity` identified by `var:response`. See the discussion block for an explanation about the existence of `var:response`.
	`prov:Entity` 4.2 / `var:output`	Each parameter of `Output Parameters` 4 is a separate `prov:Entity` identified as `var:output`.

Attributes

PROV Element	Attribute / Value	Description
`var:preObject` 1	`u2p:typeName` / `var:className`	The value `var:className` is the string with the name of the class to which `var:preObject` 1 belongs.
	`prov:type` / `u2p:Object`	The value `u2p:Object` shows that `var:preObject` 1 is an object.
`var:operation` 2	`prov:type` / `var:operationName`	The value `var:operationName` is the name of the operation `Operation` 2.
	`tmpl:startTime` / `var:operationStartTime`	The `var:operationStartTime` is an `xsd:dateTime` value for the start of `var:operation` 2.
	`tmpl:endTime` / `var:operationEndTime`	The `var:operationEndTime` is an `xsd:dateTime` value for the end of `var:operation` 2.
`var:input` 3	`prov:value` / `var:inputValue`	The value `var:inputValue` is the direct representation of `var:input` 3.
	`u2p:typeName` / `var:inputType`	The value `var:inputType` is the string with the name of the type of `var:input` 3.
`var:output` 4.2	`prov:value` / `var:outputValue`	The value `var:outputValue` is the direct representation of `var:output` 4.2.
	`u2p:typeName` / `var:outputType`	The value `var:outputType` is the string with the name of the type of `var:output` 4.2.

PROV relations

PROV Relation	Description
ⓐ `prov:used`	It is the beginning of utilizing `var:preObject` by `var:operation`.
ⓑ `prov:used`	It is the beginning of utilizing `var:input` by `var:operation`.
ⓒ `prov:wasGeneratedBy`	It is the completion of production of `var:response` by `var:operation`.
ⓓ `prov:wasDerivedFrom`	It is the construction of `var:response` based on `var:input`.
ⓔ `prov:hadMember`	It states that `var:output` is one of the elements in `var:response`.

Discussion

- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we have considered this circumstance with the aim of covering a wider spectrum of cases. When the executed operation lacks *Input data*, the UML representation in Figure 25 will not include `Input Parameters` 3▶. As a consequence, the resulting PROV template in Figure 26 will also lack `var:input` 3▶ and its associated PROV relations.

- In order to homogenise the UML Class representations in *ClP1*-*ClP10*, *Output data* have been specified by `Output Parameters` with *return* direction. Nevertheless, these `Output Parameters` could have been modelled with either *inout* or *out* directions, having no effect in their transformation to PROV.

- A concrete nuance in this pattern is that the *Output data* (`var:output`) are not computed by the *Operation execution* (`var:operation`); that is, these data already existed before the *Operation execution*. Thus, a `prov:wasGeneratedBy` relation between `var:output` and `var:operation` does not make sense in this pattern (in contrast to *ClP4*-*ClP6*, and *ClP7*-*ClP10* when they consider *Output data*). To reflect this pattern's nuance in the PROV template and taking into account the consistency between the different kinds of patterns, we have taken inspiration from how UML sequence diagram patterns (e.g., *SeqP2*) address the *Output data*. We have made this decision because the semantics of the sequence diagrams patterns (in terms of *Output data*) bears a strong resemblance with this pattern. Thus, we have included a `prov:Entity` identified by `var:response` 4.1▶ related to (1) `var:operation` 2▶, by means of Ⓒ `prov:wasGeneratedBy` (to represent the fact that it is the response who is generated by the *Operation execution*), and (2) `var:output` 4.2▶, through ⓔ `prov:hadMember` (to show that such a response is composed by the concrete output values).

Identifier *Class* diagram *Pattern 4* (*ClP4*)

Context

The execution of an operation on an object returns values that are computed based on the object's status as a whole (the values of concrete attributes involved in the computation are unknown or irrelevant). This execution does not provoke the change of the object's status.

Key elements

Object	The object on which the operation is executed.
Operation execution	The execution of the operation.
Input data	The information (if any) passed into the *Operation execution*.
Output data	The information obtained from the *Operation execution*.

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the *Object* on which the operation is executed.
Operation execution	`Operation` 2 `«process»`	The `Operation` 2 stereotyped by `«process»` represents the executed operation. Concretely, operations stereotyped by `«process»` return values that are computed based on the object's status as a whole.
Input data	`Input Parameters` 3	They specify the information passed into the *Operation execution*.
Output data	`Output Parameters` 4	They depict the information obtained from the *Operation execution*.

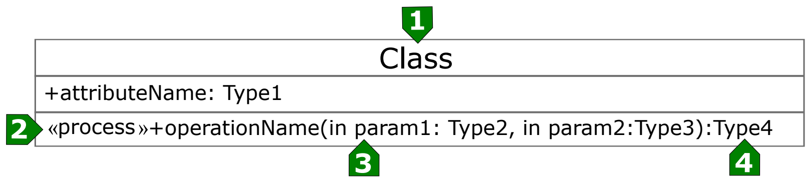
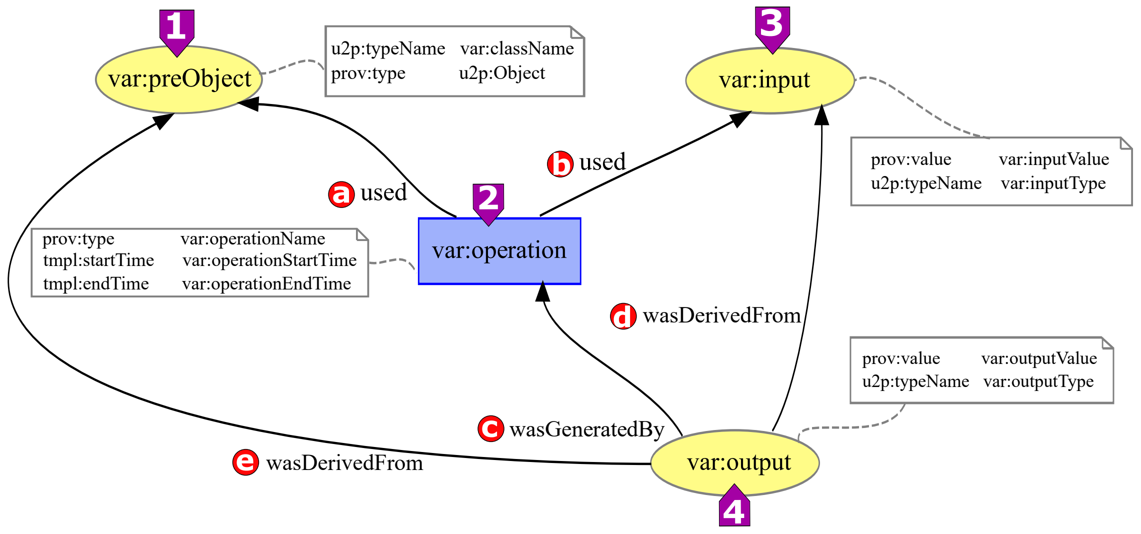

1
Class
+attributeName: Type1
2
«process»+operationName(in param1: Type2, in param2:Type3):Type4
3
4

Figure 27. UML representation that models the context given by *ClP4*



1
var:preObject
u2p:typeName var:className
prov:type u2p:Object
3
var:input
prov:value var:inputValue
u2p:typeName var:inputType
a used
b used
2
var:operation
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
d wasDerivedFrom
prov:value var:outputValue
u2p:typeName var:outputType
c wasGeneratedBy
e wasDerivedFrom
var:output
4

Figure 28. PROV template generated from the UML representation used in *ClP4* (Figure 27)

PROV elements

UML	PROV / id	Rationale
`Class` 1	`prov:Entity` 1 / `var:preObject`	The `Class` 1 that models the object on which the operation is executed is a `prov:Entity` identified as `var:preObject`. We use the prefix *pre* in this identifier because it is the object before the execution of the operation.
`Operation` 2 «`process`»	`prov:Activity` 2 / `var:operation`	The execution of `Operation` 2 stereotyped by «`process`» is a `prov:Activity` identified by `var:operation`.
`Input Parameters` 3	`prov:Entity` 3 / `var:input`	Each parameter of `Input Parameters` 3 is a separate `prov:Entity` identified as `var:input`.
`Output Parameters` 4	`prov:Entity` 4 / `var:output`	Each parameter of `Output Parameters` 4 is a separate `prov:Entity` identified as `var:output`.

Attributes

PROV Element	Attribute / Value	Description
`var:preObject` 1▸	`u2p:typeName` / `var:className`	The value `var:className` is the name of the class to which `var:preObject` 1▸ belongs.
	`prov:type` / `u2p:Object`	The value `u2p:Object` shows that `var:preObject` 1▸ is an object.
`var:operation` 2▸	`prov:type` / `var:operationName`	The value `var:operationName` is the name of the operation `Operation` 2▸.
	`tmpl:startTime` / `var:operationStartTime`	The `var:operationStartTime` is an `xsd:dateTime` value for the start of `var:operation` 2▸.
	`tmpl:endTime` / `var:operationEndTime`	The `var:operationEndTime` is an `xsd:dateTime` value for the end of `var:operation` 2▸.
`var:input` 3▸	`prov:value` / `var:inputValue`	The value `var:inputValue` is the direct representation of `var:input` 3▸.
	`u2p:typeName` / `var:inputType`	The value `var:inputType` is the string with the name of the type of `var:input` 3▸.
`var:output` 4▸	`prov:value` / `var:outputValue`	The value `var:outputValue` is the direct representation of `var:output` 4▸.
	`u2p:typeName` / `var:outputType`	The value `var:outputType` is the string with the name of the type of `var:output` 4▸.

PROV relations

PROV Relation	Description
ⓐ `prov:used`	It is the beginning of utilizing `var:preObject` by `var:operation`.
ⓑ `prov:used`	It is the beginning of utilizing `var:input` by `var:operation`.
ⓒ `prov:wasGeneratedBy`	It is the completion of production of `var:output` by `var:operation`.
ⓓ `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:input`.
ⓔ `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:preObject`.

Discussion

- Among the Class Diagrams patterns, both *ClP4* and *ClP5* address the execution of an operation that returns values computed based on information included on an object. While *ClP4* considers the object's status as a whole (without taking into account the concrete attributes' values considered for its computation), *ClP5* identifies the concrete attributes used to compute such information. Thus, *ClP4* gives a coarser grained provenance than *ClP5*.

- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we have considered this circumstance with the aim of covering a wider spectrum of cases. When the executed operation lacks *Input data*, the UML representation in Figure 27 will not include `Input Parameters` 3▸. As a consequence, the resulting PROV template in Figure 28 will also lack `var:input` 3▸ and its associated PROV relations.

- In order to homogenise the UML Class representations in *ClP1*-*ClP10*, *Output data* have been specified by `Output Parameters` with *return* direction. Nevertheless, these `Output Parameters` could have been modelled with either *inout* or *out* directions, having no effect in their transformation to PROV.

Identifier *Class* diagram *Pattern 5* (*ClP5*)

Context

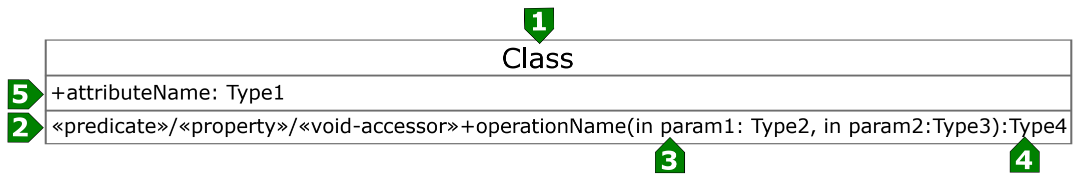
The execution of an operation on an object returns values that are computed based on values of concrete object's attributes. This execution does not provoke the change of the object's status.

Key elements

Object	The object on which the operation is executed.
Operation execution	The execution of the operation.
Input data	The information (if any) passed into the *Operation execution*.
Output data	The information obtained from the *Operation execution*.
Source Object's attributes	The concrete characteristics of the *Object* that are used to compute the *Output data*.

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the *Object* on which the operation is executed.
Operation execution	`Operation` 2 `«predicate»/` `«property»/` `«void-accessor»`	The `Operation` 2 stereotyped by `«predicate»/«property»/` `«void-accessor»` represents the executed operation. Each stereotype denotes a behaviour with specific nuances (see the discussion block); nevertheless, all of them process the *Output data* based on values of concrete object's attributes without modifying the object's status.
Input data	`Input Parameters` 3	They specify the information passed into the *Operation execution*.
Output data	`Output Parameters` 4	They depict the information obtained from the *Operation execution*.
Source Object's attributes	`Attributes` 5	They represent the characteristics of the *Object*, whose values are used to compute the *Output data*.



1
Class
5
+attributeName: Type1
2
«predicate»/«property»/«void-accessor»+operationName(in param1: Type2, in param2:Type3):Type4
3
4

Figure 29. UML representation that models the context given by *ClP5*

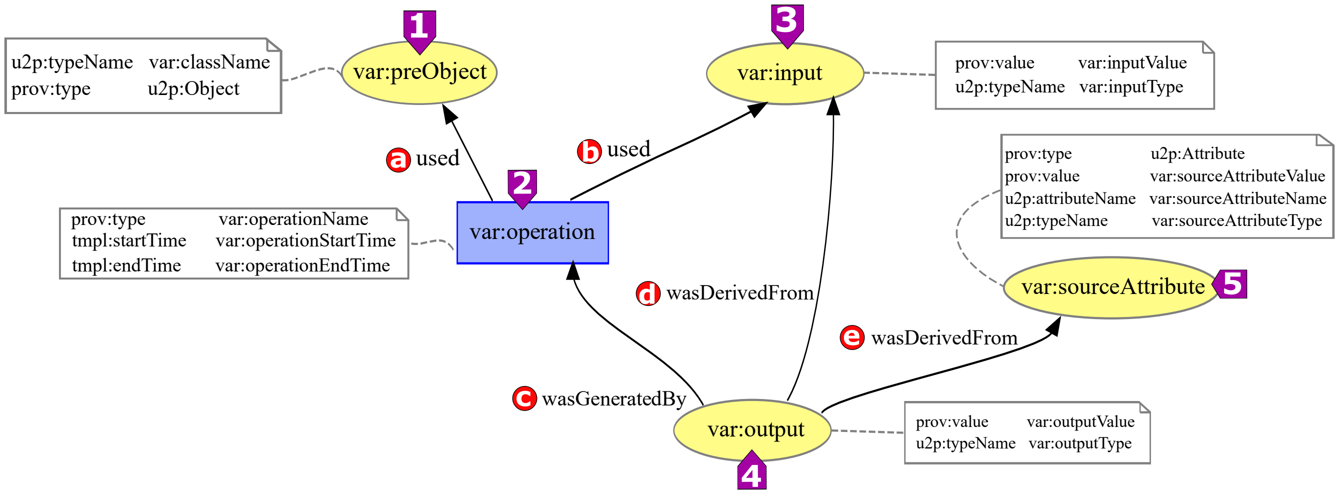

1
var:preObject
u2p:typeName var:className
prov:type u2p:Object
3
var:input
prov:value var:inputValue
u2p:typeName var:inputType
a used
b used
2
var:operation
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
prov:type u2p:Attribute
prov:value var:sourceAttributeValue
u2p:attributeName var:sourceAttributeName
u2p:typeName var:sourceAttributeType
var:sourceAttribute
5
d wasDerivedFrom
e wasDerivedFrom
c wasGeneratedBy
var:output
prov:value var:outputValue
u2p:typeName var:outputType
4

Figure 30. PROV template generated from the UML representation used in *ClP5* (Figure 29)

PROV elements

UML	PROV / id	Rationale
`Class` 1	`prov:Entity` 1 / `var:preObject`	The `Class` 1 that models the object on which the operation is executed is a `prov:Entity` identified as `var:preObject`. We use the prefix *pre* in this identifier because it is the object before the execution of the operation.
`Operation` 2 `«predicate»/` `«property»/` `«void-accessor»`	`prov:Activity` 2 / `var:operation`	The execution of `Operation` 2 stereotyped by `«predicate»/ «property»/ «void-accessor»` is a `prov:Activity` identified by `var:operation`.
`Input Parameters` 3	`prov:Entity` 3 / `var:input`	Each parameter of `Input Parameters` 3 is a separate `prov:Entity` identified as `var:input`.
`Output Parameters` 4	`prov:Entity` 4 / `var:output`	Each parameter of `Output Parameters` 4 is a separate `prov:Entity` identified as `var:output`.
`Attributes` 5	`prov:Entity` 5 / `var:sourceAttribute`	Each attribute of `Attributes` 5 is a separate `prov:Entity` identified by `var:sourceAttribute`.

Attributes

PROV Element	Attribute / Value	Description
`var:preObject` 1▶	`u2p:typeName` / `var:className`	The value `var:className` is the string with the name of the class to which `var:preObject` 1▶ belongs.
	`prov:type` / `u2p:Object`	The value `u2p:Object` shows that `var:preObject` 1▶ is an object.
`var:operation` 2▶	`prov:type` / `var:operationName`	The value `var:operationName` is the name of the operation `Operation` 2▶.
	`tmpl:startTime` / `var:operationStartTime`	The `var:operationStartTime` is an `xsd:dateTime` value for the start of `var:operation` 2▶.
	`tmpl:endTime` / `var:operationEndTime`	The `var:operationEndTime` is an `xsd:dateTime` value for the end of `var:operation` 2▶.
`var:input` 3▶	`prov:value` / `var:inputValue`	The value `var:inputValue` is the direct representation of `var:input` 3▶.
	`u2p:typeName` / `var:inputType`	The value `var:inputType` is the string with the name of the type of `var:input` 3▶.
`var:output` 4▶	`prov:value` / `var:outputValue`	The value `var:outputValue` is the direct representation of `var:output` 4▶.
	`u2p:typeName` / `var:outputType`	The value `var:outputType` is the string with the name of the type of `var:output` 4▶.
`var:sourceAttribute` 5▶	`prov:type` / `u2p:Attribute`	The value `u2p:Attribute` shows that `var:sourceAttribute` 5▶ is an attribute.
	`prov:value` / `var:sourceAttributeValue`	The value `var:sourceAttributeValue` is the direct representation of `var:sourceAttribute` 5▶.
	`u2p:attributeName` / `var:sourceAttributeName`	The value `var:sourceAttributeName` is the string with the name of `var:sourceAttribute` 5▶.
	`u2p:typeName` / `var:sourceAttributeType`	The value `var:sourceAttributeType` is the string with the name of the type of `var:sourceAttribute` 5▶.

PROV relations

PROV Relation	Description
ⓐ `prov:used`	It is the beginning of utilizing `var:preObject` by `var:operation`.
ⓑ `prov:used`	It is the beginning of utilizing `var:input` by `var:operation`.
ⓒ `prov:wasGeneratedBy`	It is the completion of production of `var:output` by `var:operation`.
ⓓ `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:input`.
ⓔ `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:sourceAttribute`.

Discussion

- Among the Class Diagrams patterns, both *ClP5* and *ClP4* address the execution of an operation that returns values computed based on information included on an object. While *ClP5* identifies the concrete attributes used to compute such information, *ClP4* considers the object's status as a whole (without taking into account the concrete attributes' values considered for its computation). Thus, *ClP5* gives a finer grained provenance than *ClP4*.

- A question that might arise is why in Figure 30 `var:sourceAttribute` 5▶ is not associated with `var:preObject` 1▶ (which represents the object with the status before the execution of the operation) by means of a `prov:hadMember` relation, whether `var:sourceAttribute` 5▶ is an attribute of `var:preObject` 1▶. We have made this decision because an object that acts as a `var:preObject` in an operation execution, was a `var:postObject` (which represents the object with the status after the execution of the operation) in a previous operation execution. Thus, the attributes associated to such an object in a `var:preObject` were registered when it previously played the role of `var:postObject`.

- The stereotypes «`predicate`», «`property`», and «`void-accessor`» denote behaviours with specific nuances. Nevertheless, these nuances do not have impact in the translation into PROV since all of them compute *Output data* based on concrete object's attributes without modifying the object's status. Concretely, «`predicate`» denotes that the operation returns boolean values, «`property`» does not restrict the type of the returned values, and «`void-accessor`» returns the information through a parameter. As we said previously, there is no distinction in the transformation into PROV; however, some of the nuances given by the stereotypes will be included in the generated provenance through the values assigned to the template's variables. For instance, «`predicate`» defines the *Output data* as boolean, fact that is included in the provenance through the value assigned to `var:outputType` in `var:output` 4▶.
- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we have considered this circumstance with the aim of covering a wider spectrum of cases. When the executed operation lacks *Input data*, the UML representation in Figure 29 will not include `Input Parameters` 3▶. As a consequence, the resulting PROV template in Figure 30 will also lack `var:input` 3▶ and its associated PROV relations.

Identifier *Class* diagram *Pattern 6* (*ClP6*)

Context

The execution of an operation on an object changes the object's status as a whole (the concrete modified attributes are unknown or irrelevant).

Key elements

Object	The object on which the operation is executed.	
	Pre-operation object	The object with the status before the execution of the operation.
	Post-operation object	The object with the status after the execution of the operation.
Operation execution	The execution of the operation.	
Input data	The information (if any) passed into the *Operation execution*.	
Object's attributes	All the characteristics of the *Object*.	

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the *Object* both before and after the execution of the operation (*Pre-operation object* and *Post-operation object*, respectively).
Operation execution	`Operation` 2 «`command`»/ «`non-void-command`»	The `Operation` 2 stereotyped by «`command`»/ «`non-void-command`» represents the executed operation. These stereotypes denote that the object changes its status, without considering the concrete modified attributes. They differ in that an operation stereotyped by «`non-void-command`» returns information, while a «`command`» stereotyped operation does not. *Note*: the PROV template depicted in Figure 32 corresponds to an operation stereotyped by «`command`» (see the discussion block for an explanation of the transformation of the operations stereotyped by «`non-void-command`»).
Input data	`Input Parameters` 3	They specify the information passed into the *Operation execution*.
Object's attributes	`Attributes` 4	They represent the characteristics of the *Object*.

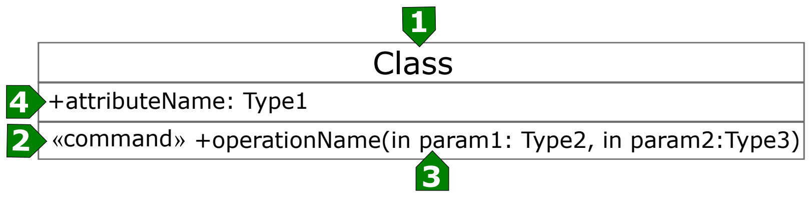


Figure 31. UML representation that models the context given by *ClP6*

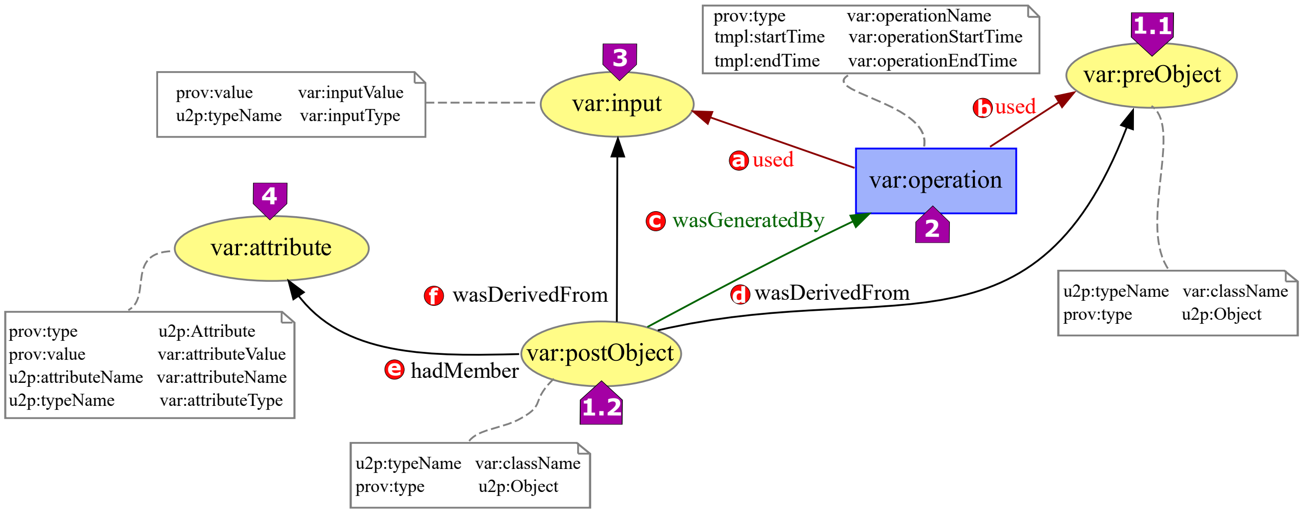

prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
1.1
var:preObject
b used
3
var:input
prov:value var:inputValue
u2p:typeName var:inputType
a used
var:operation
2
c wasGeneratedBy
4
var:attribute
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:typeName var:attributeType
f wasDerivedFrom
d wasDerivedFrom
u2p:typeName var:className
prov:type u2p:Object
var:postObject
e hadMember
1.2
u2p:typeName var:className
prov:type u2p:Object

Figure 32. PROV template generated from the UML representation used in *ClP6* (Figure 31)

PROV elements

UML	PROV / id	Rationale
`Class` 1	`prov:Entity` 1.1 / `var:preObject`	The *Pre-operation object*, i.e. the object with the status before the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:preObject`.
	`prov:Entity` 1.2 / `var:postObject`	The *Post-operation object*, i.e. the object with the status after the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:postObject`.
`Operation` 2 `«command»/` `«non-void-command»`	`prov:Activity` 2 / `var:operation`	The execution of `Operation` 2 stereotyped by `«command»/«non-void-command»` is a `prov:Activity` identified by `var:operation`.
`Input Parameters` 3	`prov:Entity` 3 / `var:input`	Each parameter of `Input Parameters` 3 is a separate `prov:Entity` identified as `var:input`.
`Attributes` 4	`prov:Entity` 4 / `var:attribute`	Each attribute of `Attributes` 4 is mapped to a separate `prov:Entity` identified by `var:attribute`.

Attributes

PROV Element	Attribute / Value	Description
var:preObject 1.1	u2p:typeName / var:className	The value var:className is the name of the class to which var:preObject 1.1 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:preObject 1.1 is an object.
var:postObject 1.2	u2p:typeName / var:className	The value var:className is the name of the class to which var:postObject 1.2 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:postObject 1.2 is an object.
var:operation 2	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 2.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 2.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 2.
var:input 3	prov:value / var:inputValue	The value var:inputValue is the direct representation of var:input 3.
	u2p:typeName / var:inputType	The value var:inputType is the string with the name of the type of var:input 3.
var:attribute 4	prov:type / u2p:Attribute	The value u2p:Attribute shows that var:attribute 4 is an attribute.
	prov:value / var:attributeValue	The value var:attributeValue is the direct representation of attribute 4.
	u2p:attributeName / var:attributeName	The value var:attributeName is the string with the name of attribute 4.
	u2p:typeName / var:attributeType	The value var:attributeType is the string with the name of the type of var:attribute 4.

PROV relations

PROV Relation	Description
ⓐ prov:used	It is the beginning of utilizing var:input by var:operation.
ⓑ prov:used	It is the beginning of utilizing var:preObject by var:operation.
ⓒ prov:wasGeneratedBy	It is the completion of production of var:postObject by var:operation.
ⓓ prov:wasDerivedFrom	It is the update of var:preObject resulting in var:postObject.
ⓔ prov:hadMember	It states that var:attribute is one of the elements in var:postObject.
ⓕ prov:wasDerivedFrom	It is the construction of var:postObject based on var:input.

Discussion

- Among the Class Diagrams patterns, patterns from *ClP6* to *ClP10* address the execution of operations that change an object's status. While, *ClP6* changes the object's status as a whole (being the concrete modified attributes unknown or irrelevant), in patterns *ClP7*-*ClP10* the concrete attributes modified by the *Operation execution* are explicitly known. In contrast to *ClP7* which directly sets the information passed into the *Operation execution* as values of concrete object's attributes, the other mentioned patterns use such information to change the object's status as a whole or the values of concrete object's attributes. It must also be noted that patterns *ClP9* and *ClP10* address the execution of operations which remove or add elements from/into an object's collection attribute, while patterns *ClP7* and *ClP8* affect either a univalued attribute or a collection attribute as a whole.
- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we do not consider this circumstance with the aim of covering a wider spectrum of cases. When the executed operation lacks *Input*

data, the UML representation in Figure 31 will not include `Input Parameters` 3. As a consequence, the resulting PROV template in Figure 32 will also lack `var:input` 3 and its associated PROV relations.

- A question that might arise is why in Figure 32 `var:attribute` 4 is associated with `var:postObject` 1.2 (which represents the object with the status after the execution of the operation), but it is not associated with `var:preObject` 1.1 (the object with the status before the execution). We have made this decision because an object that acts as a `var:preObject` in an operation execution, was a `var:postObject` in a previous operation execution. Thus, the attributes associated to such an object in a `var:preObject` were registered when it previously played the role of `var:postObject`.

- Stereotypes «`command`» and «`non-void-command`» denote that the operation performs a change to the object's status as a whole. They differ in that an operation stereotyped by «`non-void-command`» returns information, while a «`command`» stereotyped operation does not. Due to the fact that the context of this pattern does not explicitly state that output data are obtained from the *Operation execution*, we represented this context in UML using the stereotype «`command`» (see Figure 31).

  Aiming at giving an insight into how the inclusion of *Output data* affects both the UML representation and the resulting PROV template, Figure 33 depicts a UML representation with (1) the stereotype «`non-void-command`» and (2) *Output data* modelled as `Output Parameters` 5 (in this case with *return* direction, though the translation of *inout* and *out* directions would be equivalent). Figure 34 depicts its transformation into PROV. Both Figure 33 and 34 highlight the elements related to the inclusion of the *Output data* by blurring the elements coming from Figure 31 and 32, respectively.

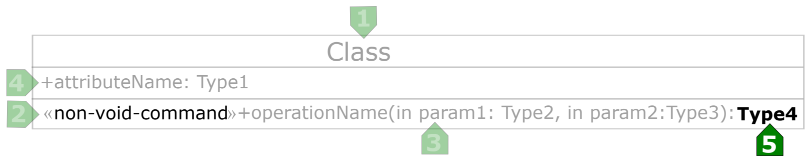

1
Class
4
+attributeName: Type1
2
«non-void-command»+operationName(in param1: Type2, in param2:Type3): Type4
3
5

Figure 33. UML representation that models the context given by *ClP6*, including `Output Parameters`.

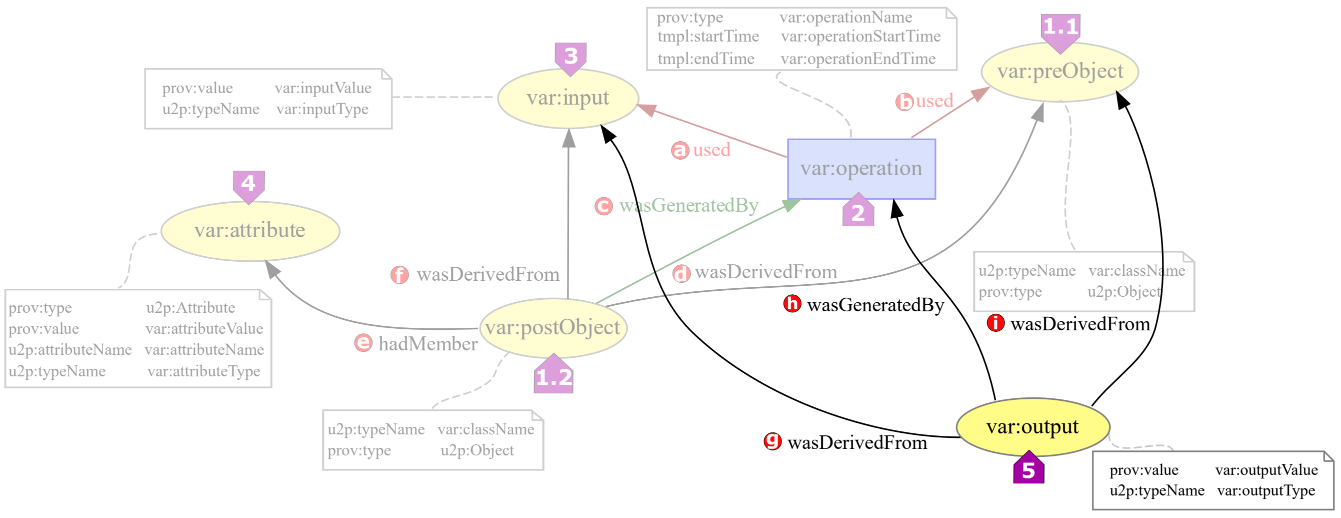

prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
1.1
var:preObject
3
var:input
prov:value var:inputValue
u2p:typeName var:inputType
b used
a used
var:operation
c wasGeneratedBy
2
4
var:attribute
f wasDerivedFrom
d wasDerivedFrom
u2p:typeName var:className
prov:type u2p:Object
h wasGeneratedBy
i wasDerivedFrom
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:typeName var:attributeType
e hadMember
var:postObject
1.2
u2p:typeName var:className
prov:type u2p:Object
g wasDerivedFrom
var:output
5
prov:value var:outputValue
u2p:typeName var:outputType

Figure 34. PROV template generated from the UML representation used in *ClP6*, including `Output Parameters` (Figure 33)

PROV elements

UML	PROV / id	Rationale
`Output Parameters` 5	`prov:Entity` 5 / `var:output`	Each parameter of `Output Parameters` 5 is a separate `prov:Entity` identified as `var:output`.

PROV relations

PROV Relation	Description
g prov:wasDerivedFrom	It is the construction of var:output based on var:input.
h prov:wasGeneratedBy	It is the completion of production of var:output by var:operation.
i prov:wasDerivedFrom	It is the construction of var:output based on var:preObject.

Attributes

PROV Element	Attribute / Value	Description
var:output 5	prov:value / var:outputValue	The value var:outputValue is the direct representation of var:output 5.
	u2p:typeName / var:outputType	The value var:outputType is the string with the name of the type of var:output 5.

Identifier *Class* diagram *Pattern 7* (*ClP7*)

Context

The execution of an operation on an object directly sets the information passed to the operation as values of concrete object's attributes, thus provoking a change in the object's status.

Key elements

Object	The object on which the operation is executed.	
	Pre-operation object	The object with the status before the execution of the operation.
	Post-operation object	The object with the status after the execution of the operation.
Operation execution	The execution of the operation.	
Input data	The information passed into the *Operation execution*.	
Object's attributes	All the characteristics of the *Object*. Since, as a consequence of the *Operation execution*, the values of some attributes change, we have identified:	
	Modified attributes	The modified *Object's attributes*.
	Unmodified attributes	The not modified *Object's attributes*.

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the *Object* both before and after the execution of the operation (*Pre-operation object* and *Post-operation object*, respectively).
Operation execution	`Operation` 2 «`set`»	The `Operation` 2 stereotyped by «`set`» represents the executed operation. Concretely, the stereotype «`set`» denotes that *Input data* are directly set as values of concrete attributes of the object.
Input data	`Input Parameters` 3	They specify the information passed into the *Operation execution*.
Object's attributes	`Attributes` 4	They represent the characteristics of the *Object*.

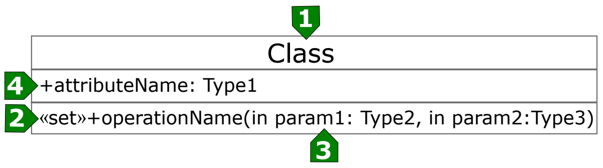


Figure 35. UML representation that models the context given by *ClP7*

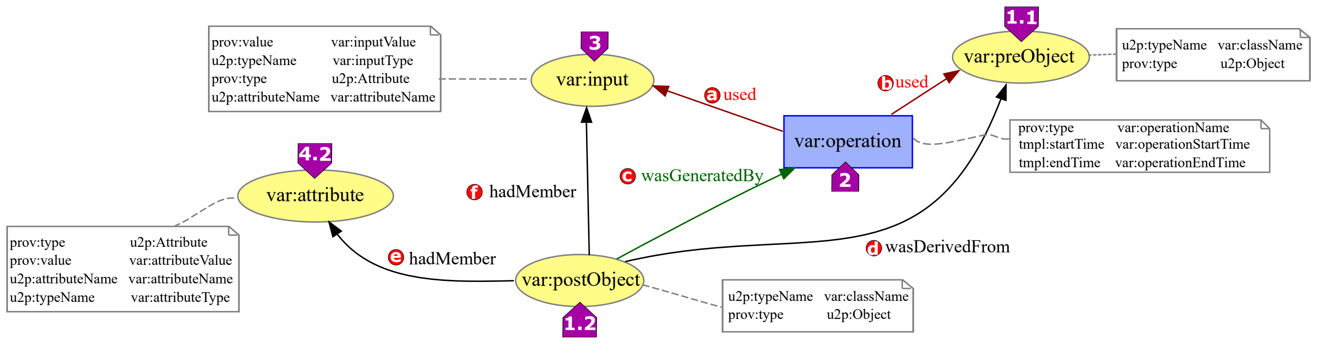

1.1
var:preObject
u2p:typeName var:className
prov:type u2p:Object
3
var:input
prov:value var:inputValue
u2p:typeName var:inputType
prov:type u2p:Attribute
u2p:attributeName var:attributeName
a used
b used
var:operation
2
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
4.2
var:attribute
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:typeName var:attributeType
f hadMember
c wasGeneratedBy
e hadMember
d wasDerivedFrom
var:postObject
1.2
u2p:typeName var:className
prov:type u2p:Object

Figure 36. PROV template generated from the UML representation used in *ClP7* (Figure 35)

PROV elements

UML	PROV / id	Rationale
`Class` 1	`prov:Entity` 1.1 / `var:preObject`	The *Pre-operation object*, i.e. the object with the status before the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:preObject`.
	`prov:Entity` 1.2 / `var:postObject`	The *Post-operation object*, i.e. the object with the status after the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:postObject`.
`Operation` 2 «set»	`prov:Activity` 2 / `var:operation`	The execution of `Operation` 2 stereotyped by «set» is a `prov:Activity` identified by `var:operation`.
`Input Parameters` 3	`prov:Entiy` 3 / `var:input`	Each parameter of `Input Parameters` 3 is a separate `prov:Entity` identified as `var:input`.
`Attributes` 4	None /	The *Modified attributes* (belonging to `Attributes` 4) are already mapped to `var:input`. For further information, see the discussion.
	`prov:Entity` 4.2 / `var:attribute`	Each *Unmodified attribute* (belonging to `Attributes` 4) is mapped to a separate `prov:Entity` with identifier `var:attribute`.

Attributes

PROV Element	Attribute / Value	Description
var:preObject 1.1	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:preObject 1.1 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:preObject 1.1 is an object.
var:postObject 1.2	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:postObject 1.2 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:postObject 1.2 is an object.
var:operation 2	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 2.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 2.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 2.
var:input 3	prov:value / var:inputValue	The value var:inputValue is the direct representation of var:input 3.
	u2p:typeName / var:inputType	The value var:inputType is the string with the name of the type of var:input 3.
	prov:type / u2p:Attribute	The value u2p:Attribute shows that var:input 3 is an attribute.
	u2p:attributeName / var:attributeName	The value var:attributeName is the string with the name of the attribute var:input 3.
var:attribute 4.2	prov:type / u2p:Attribute	The value u2p:Attribute shows that var:attribute 4.2 is an attribute.
	prov:value / var:attributeValue	The value var:attributeValue is the direct representation of var:attribute 4.2.
	u2p:attributeName / var:attributeName	The value var:attributeName is the string with the name of var:attribute 4.2.
	u2p:typeName / var:attributeType	The value var:attributeType is the string with the name of the type of var:attribute 4.2.

PROV relations

PROV Relation	Description
ⓐ prov:used	It is the beginning of utilizing var:input by var:operation.
ⓑ prov:used	It is the beginning of utilizing var:preObject by var:operation.
ⓒ prov:wasGeneratedBy	It is the completion of production of var:postObject by var:operation.
ⓓ prov:wasDerivedFrom	It is the update of var:preObject resulting in var:postObject.
ⓔ prov:hadMember	It states that var:attribute is one of the elements in var:postObject.
ⓕ prov:hadMember	It states that var:input is one of the elements in var:postObject. This is due to the fact that in this context the input information is directly set as values of certain attributes of the *Object*.

Discussion

- Among the Class Diagrams patterns, patterns from *ClP6* to *ClP10* address the execution of operations that change an object's status. While, *ClP6* changes the object's status as a whole (being the concrete modified attributes unknown or irrelevant), in patterns *ClP7*-*ClP10* the concrete attributes modified by the *Operation execution* are explicitly known. In contrast to *ClP7* which directly sets the information passed into the *Operation execution* as values of concrete object's attributes, the other mentioned patterns use such information to change the object's status as a whole or the values of concrete object's attributes. It must also be noted that patterns *ClP9* and *ClP10* address the execution of operations which remove or add elements from/into an object's collection attribute, while patterns *ClP7* and *ClP8* affect either a univalued attribute or a collection attribute as a whole.

- A question that might arise is why in Figure 36 `var:attribute` 4.2 is associated with `var:postObject` 1.2 (which represents the object with the status after the execution of the operation), but it is not associated with `var:preObject` 1.1 (the object with the status before the execution). We have made this decision because an object that acts as a `var:preObject` in an operation execution, was a `var:postObject` in a previous operation execution. Thus, the attributes associated to such an object in a `var:preObject` were registered when it previously played the role of `var:postObject`.

- This context states that the *Input data* are directly set as values of certain object's attributes, which means that the *Input data* correspond directly to the *Modified attributes*. This fact is represented in the PROV template by means of the pair attribute/value `prov:type`/`u2p:Attribute` of `var:input` 3, and the relation f `prov:hadMember` between `var:postObject` 1.2 and `var:input` 3. Additionally, `var:input` 3 has the attribute `u2p:attributeName` whose value `var:attributeName` denotes the name of the modified attribute.

- Although the *context* of this pattern does not explicitly state that output data should be obtained from the *Operation execution*, this could be the case. However, we do not include this output data in this pattern description to avoid overburden both the UML and PROV explanations with information out of the scope of the *context*.

  Aiming at giving an insight into how the inclusion of *Output data* affects both the UML representation and the resulting PROV template, Figure 37 depicts a UML representation with the *Output data* modelled as `Output Parameters` 5 (in this case with *return* direction, though the translation of *inout* and *out* directions would be equivalent). Figure 38 depicts its transformation into PROV. Both Figure 37 and 38 highlight the elements related to the inclusion of the *Output data* by blurring the elements coming from Figure 35 and 36, respectively.

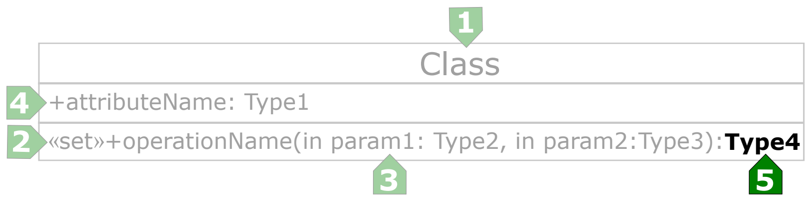

1
Class
4
+attributeName: Type1
2
«set»+operationName(in param1: Type2, in param2:Type3):Type4
3
5

Figure 37. UML representation that models the context given by *ClP7*, including `Output Parameters`.

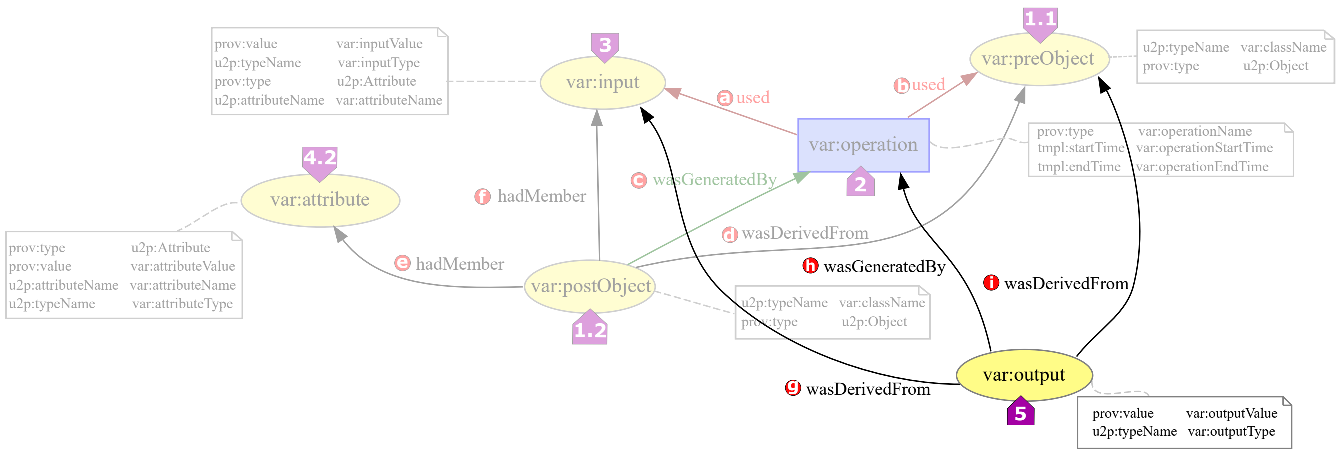

prov:value var:inputValue
u2p:typeName var:inputType
prov:type u2p:Attribute
u2p:attributeName var:attributeName
3
var:input
a used
1.1
var:preObject
b used
u2p:typeName var:className
prov:type u2p:Object
var:operation
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
2
4.2
var:attribute
f hadMember
c wasGeneratedBy
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:typeName var:attributeType
d wasDerivedFrom
e hadMember
h wasGeneratedBy
i wasDerivedFrom
var:postObject
u2p:typeName var:className
prov:type u2p:Object
1.2
g wasDerivedFrom
var:output
5
prov:value var:outputValue
u2p:typeName var:outputType

Figure 38. PROV template generated from the UML representation used in *ClP7*, including `Output Parameters` (Figure 37)

PROV elements

UML	PROV / id	Rationale
`Output Parameters`	`prov:Entity` / `var:output`	Each parameter of `Output Parameters` is a separate `prov:Entity` identified as `var:output`.

PROV relations

PROV Relation	Description
g `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:input`.
h `prov:wasGeneratedBy`	It is the completion of production of `var:output` by `var:operation`.
i `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:preObject`.

Attributes

PROV Element	Attribute / Value	Description
`var:output`	`prov:value` / `var:outputValue`	The value `var:outputValue` is the direct representation of `var:output`.
	`u2p:typeName` / `var:outputType`	The value `var:outputType` is the string with the name of the type of `var:output`.

Identifier *Class diagram Pattern 8* (*ClP8*)

Context

The execution of an operation on an object changes the values of concrete object's attributes, thus provoking a change in the object's status.

Key elements

Object	The object on which the operation is executed.	
	Pre-operation object	The object with the status before the execution of the operation.
	Post-operation object	The object with the status after the execution of the operation.
Operation execution	The execution of the operation.	
Input data	The information (if any) passed into the *Operation execution*.	
Object's attributes	All the characteristics of the *Object*. Since, as a consequence of the *Operation execution*, the values of some attributes change, we have identified:	
	Modified attributes	The modified *Object's attributes*.
	Unmodified attributes	The not modified *Object's attributes*.

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the *Object* both before and after the execution of the operation (*Pre-operation object* and *Post-operation object*, respectively).
Operation execution	`Operation` 2 «`modify`»	The `Operation` 2 stereotyped by «`modify`» represents the executed operation. Concretely, the stereotype «`modify`» denotes that concrete attributes of the object are modified.
Input data	`Input Parameters` 3	They specify the information passed into the *Operation execution*.
Object's attributes	`Attributes` 4	They represent the characteristics of the *Object*.

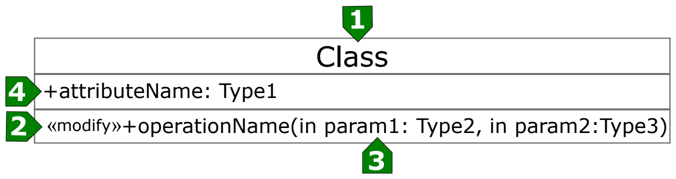


Figure 39. UML representation that models the context given by *ClP8*

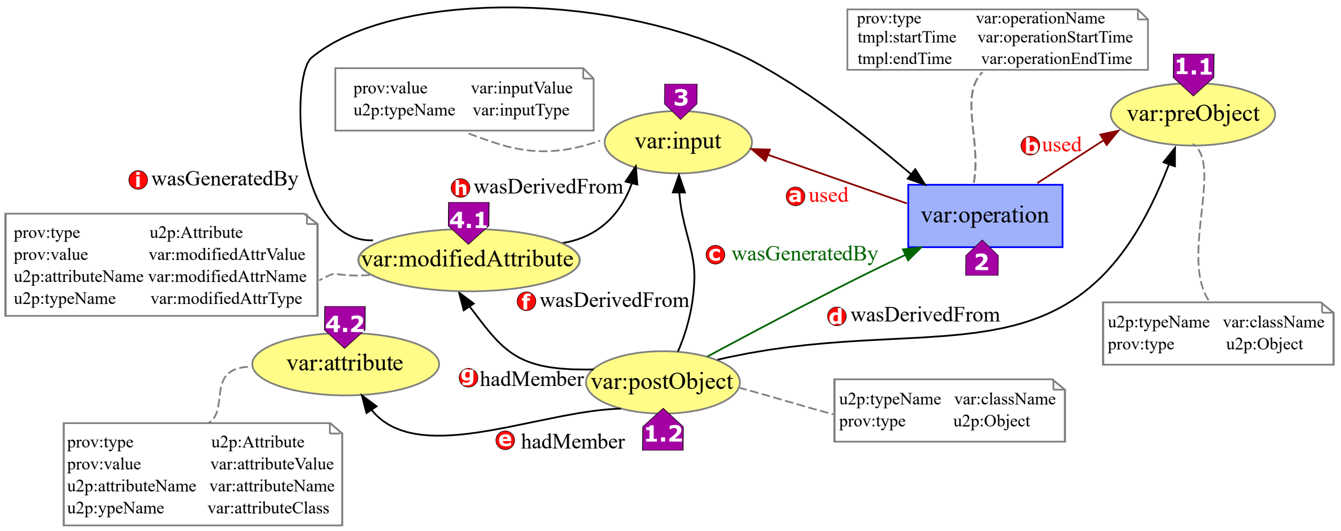

prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
1.1
var:preObject
prov:value var:inputValue
u2p:typeName var:inputType
3
var:input
b used
a used
var:operation
2
i wasGeneratedBy
h wasDerivedFrom
4.1
var:modifiedAttribute
prov:type u2p:Attribute
prov:value var:modifiedAttrValue
u2p:attributeName var:modifiedAttrName
u2p:typeName var:modifiedAttrType
c wasGeneratedBy
f wasDerivedFrom
d wasDerivedFrom
u2p:typeName var:className
prov:type u2p:Object
4.2
var:attribute
g hadMember
var:postObject
u2p:typeName var:className
prov:type u2p:Object
e hadMember
1.2
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:ypeName var:attributeClass

Figure 40. PROV template generated from the UML representation used in *ClP8* (Figure 39)

PROV elements

UML	PROV / id	Rationale
`Class` 1	`prov:Entity` 1.1 / `var:preObject`	The *Pre-operation object*, i.e. the object with the status before the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:preObject`.
	`prov:Entity` 1.2 / `var:postObject`	The *Post-operation object*, i.e. the object with the status after the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:postObject`.
`Operation` 2 `«modify»`	`prov:Activity` 2 / `var:operation`	The execution of `Operation` 2 stereotyped by `«modify»` is a `prov:Activity` identified by `var:operation`.
`Input Parameters` 3	`prov:Entiy` 3 / `var:input`	Each parameter of `Input Parameters` 3 is a separate `prov:Entity` identified as `var:input`.
`Attributes` 4	`prov:Entity` 4.1 / `var:modifiedAttribute`	Each *Modified attribute* (belonging to `Attributes` 4) is mapped to a separate `prov:Entity` with identifier `var:modifiedAttribute`.
	`prov:Entity` 4.2 / `var:attribute`	Each *Unmodified attribute* (belonging to `Attributes` 4) is mapped to a separate `prov:Entity` with identifier `var:attribute`.

Attributes

PROV Element	Attribute / Value	Description
var:preObject 1.1	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:preObject 1.1 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:preObject 1.1 is an object.
var:postObject 1.2	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:postObject 1.2 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:postObject 1.2 is an object.
var:operation 2	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 2.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 2.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 2.
var:input 3	prov:value / var:inputValue	The value var:inputValue is the direct representation of var:input 3.
	u2p:typeName / var:inputType	The value var:inputType is the string with the name of the type of var:input 3.
var:modifiedAttribute 4.1	prov:type / u2p:Attribute	The value u2p:Attribute shows that var:modifiedAttribute 4.1 is an attribute.
	prov:value / var:modifiedAttrValue	The value var:attributeValue is the direct representation of var:modifiedAttribute 4.1.
	u2p:attributeName / var:modifiedAttrName	The value var:modifiedAttrName is the string with the name of var:modifiedAttribute 4.1.
	u2p:typeName / var:modifiedAttrType	The value var:attributeType is the string with the name of the type of var:modifiedAttribute 4.1.
var:attribute 4.2	prov:type / u2p:Attribute	The value u2p:Attribute shows that var:attribute 4.2 is an attribute.
	prov:value / var:attributeValue	The value var:attributeValue is the direct representation of var:attribute 4.2.
	u2p:attributeName / var:attributeName	The value var:attributeName is the string with the name of var:attribute 4.2.
	u2p:typeName / var:attributeType	The value var:attributeType is the string with the name of the type of var:attribute 4.2.

PROV relations

PROV Relation	Description
a prov:used	It is the beginning of utilizing var:input by var:operation.
b prov:used	It is the beginning of utilizing var:preObject by var:operation.
c prov:wasGeneratedBy	It is the completion of production of var:postObject by var:operation.
d prov:wasDerivedFrom	It is the update of var:preObject resulting in var:postObject.
e prov:hadMember	It states that var:attribute is one of the elements in var:postObject.
f prov:wasDerivedFrom	It is the construction of var:postObject based on var:input.
g prov:hadMember	It states that var:modifiedAttribute is one of the elements in var:postObject.
h prov:wasDerivedFrom	It is the construction of var:modifiedAttribute based on var:input.
i prov:wasGeneratedBy	It is the completion of production of var:modifiedAttribute by var:operation.

Discussion

- Among the Class Diagrams patterns, patterns from *ClP6* to *ClP10* address the execution of operations that change an object's status. While, *ClP6* changes the object's status as a whole (being the concrete modified attributes unknown or irrelevant), in patterns *ClP7*-*ClP10* the concrete attributes modified by the *Operation execution* are explicitly known. In contrast to *ClP7* which directly sets the information passed into the *Operation execution* as values of concrete object's attributes, the other mentioned patterns use such information to change the object's status as a whole or the values of concrete object's attributes. It must also be noted that patterns *ClP9* and *ClP10* address the execution of operations which remove or add elements from/into an object's collection attribute, while patterns *ClP7* and *ClP8* affect either a univalued attribute or a collection attribute as a whole.

- A question that might arise is why in Figure 40 `var:attribute` 4.2 is associated with `var:postObject` 1.2 (which represents the object with the status after the execution of the operation), but it is not associated with `var:preObject` 1.1 (the object with the status before the execution). We have made this decision because an object that acts as a `var:preObject` in an operation execution, was a `var:postObject` in a previous operation execution. Thus, the attributes associated to such an object in a `var:preObject` were registered when it previously played the role of `var:postObject`.

- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we do not to consider this circumstance with the aim of covering a wider spectrum of cases. When the executed operation lacks *Input data*, the UML representation in Figure 39 will not include `Input Parameters` 3. As a consequence, the resulting PROV template in Figure 40 will also lack `var:input` 3 and its associated PROV relations.

- Although the *context* of this pattern does not explicitly state that output data should be obtained from the *Operation execution*, this could be the case. However, we do not include this output data in this pattern description to avoid overburden both the UML and PROV explanations with information out of the scope of the *context*.

  Aiming at giving an insight into how the inclusion of *Output data* affects both UML representation and the resulting PROV template, Figure 41 depicts a UML representation with the *Output data* modelled as `Output Parameters` 5 (in this case with *return* direction, though the translation of *inout* and *out* directions would be equivalent). Figure 42 depicts its transformation into PROV. Both Figure 41 and 42 highlight the elements related to the inclusion of the *Output data* by blurring the elements coming from Figure 39 and 40, respectively.

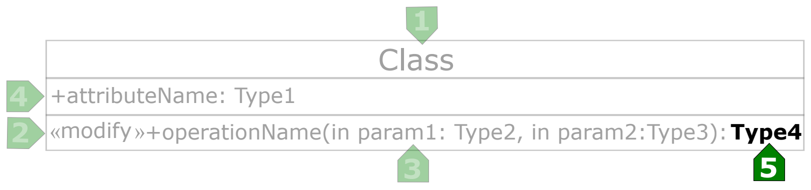
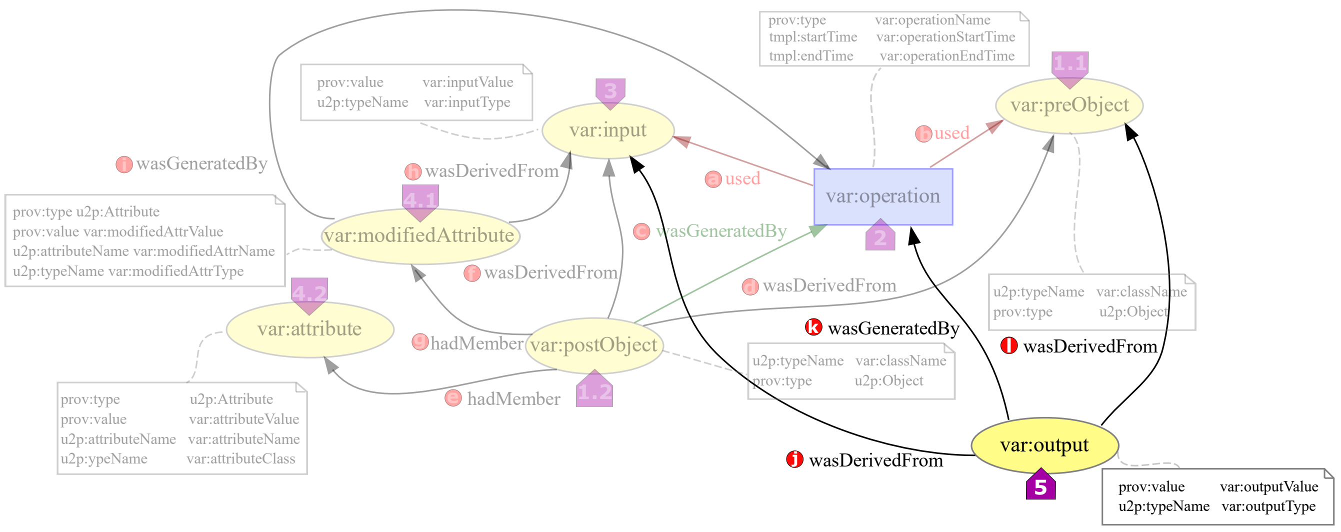

1
Class
4
+attributeName: Type1
2
«modify»+operationName(in param1: Type2, in param2:Type3): Type4
3
5

Figure 41. UML representation that models the context given by *ClP8*, including `Output Parameters`.



prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
1.1
var:preObject
prov:value var:inputValue
u2p:typeName var:inputType
3
var:input
a used
b used
i wasGeneratedBy
h wasDerivedFrom
var:operation
prov:type u2p:Attribute
prov:value var:modifiedAttrValue
u2p:attributeName var:modifiedAttrName
u2p:typeName var:modifiedAttrType
4.1
var:modifiedAttribute
c wasGeneratedBy
2
f wasDerivedFrom
d wasDerivedFrom
4.2
var:attribute
u2p:typeName var:className
prov:type u2p:Object
k wasGeneratedBy
l wasDerivedFrom
g hadMember
var:postObject
u2p:typeName var:className
prov:type u2p:Object
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:ypeName var:attributeClass
e hadMember
1.2
j wasDerivedFrom
var:output
5
prov:value var:outputValue
u2p:typeName var:outputType

Figure 42. PROV template generated from the UML representation used in *ClP8*, including `Output Parameters` (Figure 41)

PROV elements

UML	PROV / id	Rationale
`Output Parameters` 5	`prov:Entity` 5 / `var:output`	Each parameter of `Output Parameters` 5 is a separate `prov:Entity` identified as `var:output`.

PROV relations

PROV Relation	Description
ⓙ `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:input`.
ⓚ `prov:wasGeneratedBy`	It is the completion of production of `var:output` by `var:operation`.
ⓛ `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:preObject`.

Attributes

PROV Element	Attribute / Value	Description
`var:output` 5	`prov:value` / `var:outputValue`	The value `var:outputValue` is the direct representation of `var:output` 5.
	`u2p:typeName` / `var:outputType`	The value `var:outputType` is the string with the name of the type of `var:output` 5.

Identifier *Class* diagram *Pattern 9* (*ClP9*)

Context

The execution of an operation on an object removes element(s) from a concrete object's collection attribute, thus provoking a change in the object's status.

Key elements

Object	The object on which the operation is executed.	
	Pre-operation object	The object with the status before the execution of the operation.
	Post-operation object	The object with the status after the execution of the operation.
Operation execution	The execution of the operation.	
Input data	The information (if any) passed into the *Operation execution*.	
Object's attributes	All the characteristics of the *Object*. Since, as a consequence of the *Operation execution*, a concrete collection attribute changes, we have identified:	
	Modified collection attribute	The modified *Object's attribute*.
	Unmodified attributes	The not modified *Object's attributes*.

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the *Object* both before and after the execution of the operation (*Pre-operation object* and *Post-operation object*, respectively).
Operation execution	`Operation` 2 «`remove`»	The `Operation` 2 stereotyped by «`remove`» represents the executed operation. Concretely, the stereotype «`remove`» denotes that an element (or elements) of a concrete collection attribute is removed.
Input data	`Input Parameters` 3	They specify the information passed into the *Operation execution*.
Object's attributes	`Attributes` 4	They represent the characteristics of the *Object*.

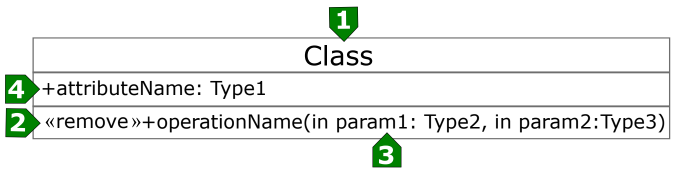


Figure 43. UML representation that models the context given by *ClP9*

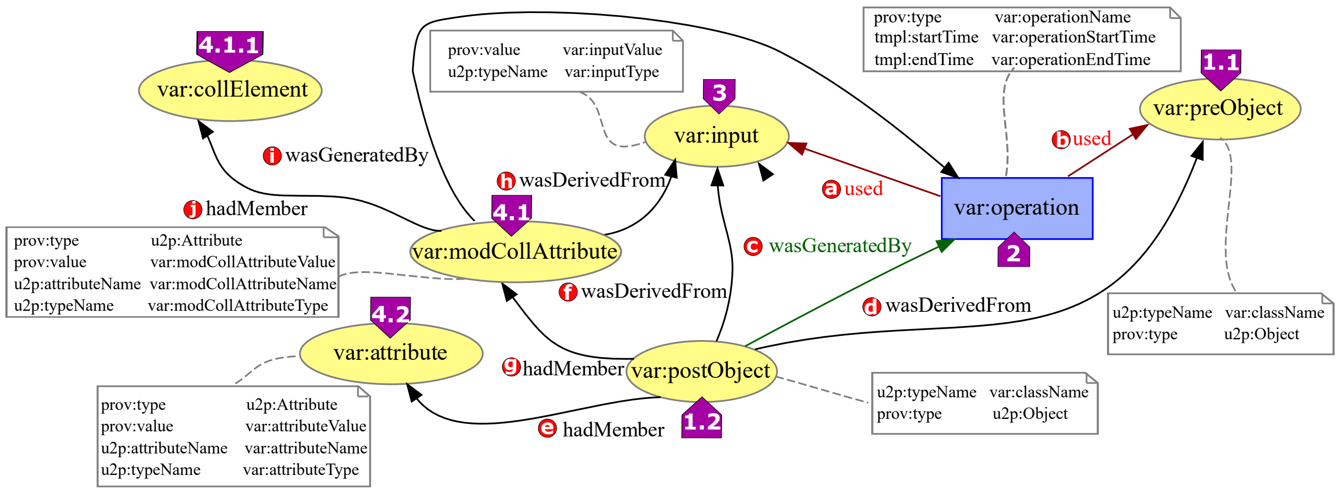

4.1.1
var:collElement
prov:value var:inputValue
u2p:typeName var:inputType
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
1.1
var:preObject
3
var:input
i wasGeneratedBy
h wasDerivedFrom
b used
a used
j hadMember
4.1
var:operation
prov:type u2p:Attribute
prov:value var:modCollAttributeValue
u2p:attributeName var:modCollAttributeName
u2p:typeName var:modCollAttributeType
var:modCollAttribute
c wasGeneratedBy
2
f wasDerivedFrom
d wasDerivedFrom
u2p:typeName var:className
prov:type u2p:Object
4.2
var:attribute
g hadMember
var:postObject
u2p:typeName var:className
prov:type u2p:Object
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:typeName var:attributeType
e hadMember
1.2

Figure 44. PROV template generated from the UML representation used in *ClP9* (Figure 43)

PROV elements

UML	PROV / id	Rationale
`Class` 1	`prov:Entity` 1.1 / `var:preObject`	The *Pre-operation object*, i.e. the object with the status before the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:preObject`.
	`prov:Entity` 1.2 / `var:postObject`	The *Post-operation object*, i.e. the object with the status after the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:postObject`.
`Operation` 2 `«remove»`	`prov:Activity` 2 / `var:operation`	The execution of `Operation` 2 stereotyped by `«remove»` is a `prov:Activity` identified by `var:operation`.
`Input Parameters` 3	`prov:Entity` 3 / `var:input`	Each parameter of `Input Parameters` 3 is a separate `prov:Entity` identified as `var:input`.
`Attributes` 4	`prov:Entity` 4.1 / `var:modCollAttribute`	The *Modified collection attribute* (belonging to `Attributes` 4) is a `prov:Entity` with identifier `var:modCollAttribute`. Additionally, each element in this collection is a separate `prov:Entity` identified by `var:collElement` 4.1.1
	`prov:Entity` 4.2 / `var:attribute`	Each *Unmodified attribute* (belonging to `Attributes` 4) is mapped to a separate `prov:Entity` with identifier `var:attribute`.

Attributes

PROV Element	Attribute / Value	Description
var:preObject 1.1	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:preObject 1.1 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:preObject 1.1 is an object.
var:postObject 1.2	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:postObject 1.2 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:postObject 1.2 is an object.
var:operation 2	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 2.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 2.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 2.
var:input 3	prov:value / var:inputValue	The value var:inputValue is the direct representation of var:input 3.
	u2p:typeName / var:inputType	The value var:inputType is the string with the name of the type of var:input 3.
var:modCollAttribute 4.1	prov:type / u2p:Attribute	The value u2p:Attribute shows that var:modCollAttribute 4.1 is an attribute.
	prov:value / var:modCollAttributeValue	The value var:modCollAttributeValue is the direct representation of var:modCollAttribute 4.1.
	u2p:attributeName / var:modCollAttributeName	The value var:modCollAttributeName is the string with the name of var:modCollAttribute 4.1.
	u2p:typeName / var:modCollAttributeType	The value var:modCollAttributeType is the string with the name of the type of var:modCollAttribute 4.1.
var:attribute 4.2	prov:type / u2p:Attribute	The value u2p:Attribute shows that var:attribute 4.2 is an attribute.
	prov:value / var:attributeValue	The value var:attributeValue is the direct representation of var:attribute 4.2.
	u2p:attributeName / var:attributeName	The value var:attributeName is the string with the name of var:attribute 4.2.
	u2p:typeName / var:attributeType	The value var:attributeType is the string with the name of the type of var:attribute 4.2.

PROV relations

PROV Relation	Description
ⓐ `prov:used`	It is the beginning of utilizing `var:input` by `var:operation`.
ⓑ `prov:used`	It is the beginning of utilizing `var:preObject` by `var:operation`.
ⓒ `prov:wasGeneratedBy`	It is the completion of production of `var:postObject` by `var:operation`.
ⓓ `prov:wasDerivedFrom`	It is the update of `var:preObject` resulting in `var:postObject`.
ⓔ `prov:hadMember`	It states that `var:attribute` is one of the elements in `var:postObject`.
ⓕ `prov:wasDerivedFrom`	It is the construction of `var:postObject` based on `var:input`.
ⓖ `prov:hadMember`	It states that `var:modCollAttribute` is one of the elements in `var:postObject`.
ⓗ `prov:wasDerivedFrom`	It is the construction of `var:modCollAttribute` based on `var:input`.
ⓘ `prov:wasGeneratedBy`	It is the completion of production of `var:modCollAttribute` by `var:operation`.
ⓙ `prov:hadMember`	It states that `var:collElement` is one of the elements in `var:modCollAttribute`.

Discussion

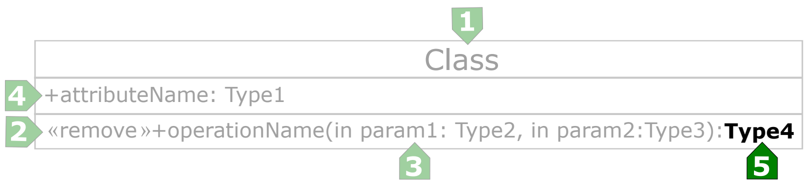
- Among the Class Diagrams patterns, patterns from *ClP6* to *ClP10* address the execution of operations that change an object's status. While, *ClP6* changes the object's status as a whole (being the concrete modified attributes unknown or irrelevant), in patterns *ClP7*-*ClP10* the concrete attributes modified by the *Operation execution* are explicitly known. In contrast to *ClP7* which directly sets the information passed into the *Operation execution* as values of concrete object's attributes, the other mentioned patterns use such information to change the object's status as a whole or the values of concrete object's attributes. It must also be noted that patterns *ClP9* and *ClP10* address the execution of operations which remove or add elements from/into an object's collection attribute, while patterns *ClP7* and *ClP8* affect either a univalued attribute or a collection attribute as a whole.

- A question that might arise is why in Figure 44 `var:attribute` 4.2 is associated with `var:postObject` 1.2 (which represents the object with the status after the execution of the operation), but it is not associated with `var:preObject` 1.1 (the object with the status before the execution). We have made this decision because an object that acts as a `var:preObject` in an operation execution, was a `var:postObject` in a previous operation execution. Thus, the attributes associated to such an object in a `var:preObject` were registered when it previously played the role of `var:postObject`.

- Although the *context* of this pattern does not explicitly state that *Input data* should be passed to the operation, we have considered this circumstance with the aim of covering a wider spectrum of cases. When the executed operation lacks *Input data*, the UML representation in Figure 43 will not include `Input Parameters` 3. As a consequence, the resulting PROV template in Figure 44 will also lack `var:input` 3 and its associated PROV relations.

- Although the *context* of this pattern does not explicitly state that output data should be obtained from the *Operation execution*, this could be the case. However, we have decided not to include this output data in this pattern description to avoid overburden both the UML and PROV explanations with information out of the scope of the *context*.

  Aiming at giving an insight into how the inclusion of *Output data* affects both UML representation and the resulting PROV template, Figure 45 depicts a UML representation with the *Output data* modelled as `Output Parameters` 5 (in this case with *return* direction, though the translation of *inout* and *out* directions would be equivalent). Figure 46 depicts its transformation into PROV. Both Figure 45 and 46 highlight the elements related to the inclusion of the *Output data* by blurring the elements coming from Figure 43 and 44, respectively.



1
Class
4
+attributeName: Type1
2
«remove»+operationName(in param1: Type2, in param2:Type3):Type4
3
5

Figure 45. UML representation that models the context given by *ClP9*, including `Output Parameters`.

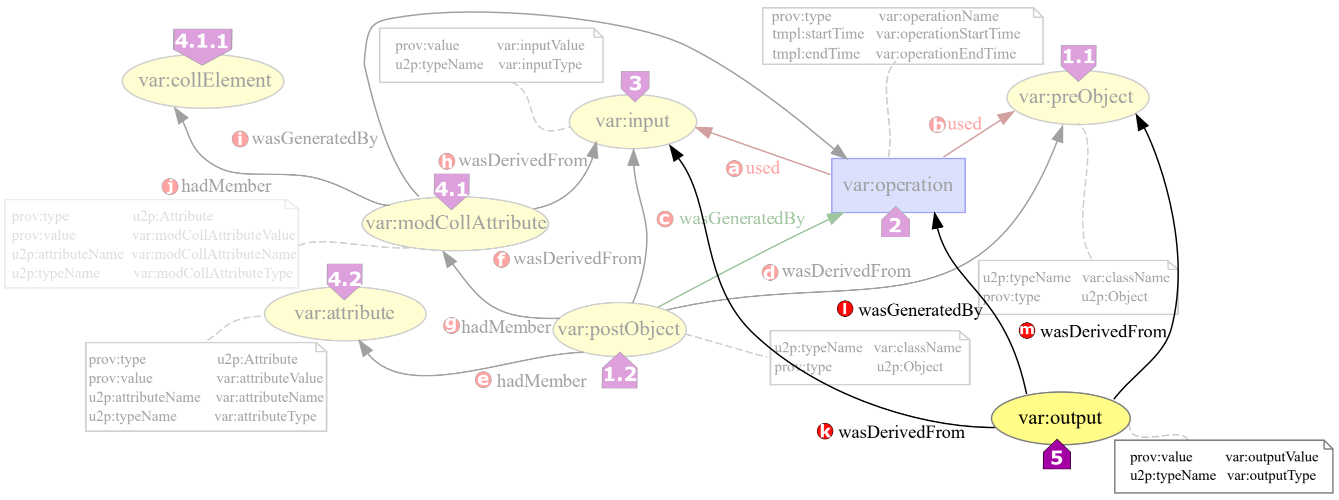


Figure 46. PROV template generated from the UML representation used in *ClP9*, including `Output Parameters` (Figure 45)

PROV elements

UML	PROV / id	Rationale
`Output Parameters` 5	`prov:Entity` 5 / `var:output`	Each parameter of `Output Parameters` 5 is a separate `prov:Entity` identified as `var:output`.

PROV relations

PROV Relation	Description
k `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:input`.
l `prov:wasGeneratedBy`	It is the completion of production of `var:output` by `var:operation`.
m `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:preObject`.

Attributes

PROV Element	Attribute / Value	Description
`var:output` 5	`prov:value` / `var:outputValue`	The value `var:outputValue` is the direct representation of `var:output` 5.
	`u2p:typeName` / `var:outputType`	The value `var:outputType` is the string with the name of the type of `var:output` 5.

Identifier *Class* diagram *Pattern 10* (*ClP10*)

Context

The execution of an operation on an object directly adds the information passed to the operation as new element(s) of a concrete object's collection attribute, thus provoking a change in the object's status.

Key elements

Object	The object to which the operation to be executed belongs.	
	Pre-operation object	The object with the status before the execution of the operation.
	Post-operation object	The object with the status after the execution of the operation.
Operation execution	The execution of the behaviour specified by the operation.	
Input data	The information passed into the *Operation execution*.	
Object's attributes	All the characteristics of the *Object*. Since, as a consequence of the *Operation execution*, a concrete collection attribute changes, we have identified:	
	Modified collection attribute	The modified *Object's attribute*.
	Unmodified attributes	The not modified *Object's attributes*.

UML Diagram

Key Element	UML	Rationale
Object	`Class` 1	*Objects* are classified attending to their characteristics and behaviour by means of `classes`. Thus, we use `Class` 1 to represent the *Object* both before and after the execution of the operation (*Pre-operation object* and *Post-operation object*, respectively).
Operation execution	`Operation` 2 `«add»`	The `Operation` 2 stereotyped by `«add»` represents the executed operation. Concretely, the stereotype `«add»` denotes that a new element (or elements) is directly added to a concrete collection attribute.
Input data	`Input Parameters` 3	They specify the information passed into the *Operation execution*.
Object's attributes	`Attributes` 4	They represent the characteristics of the *Object*.

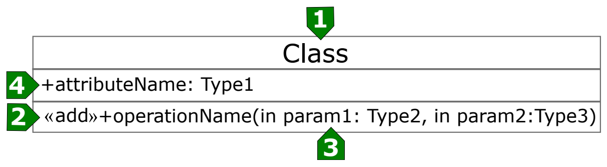
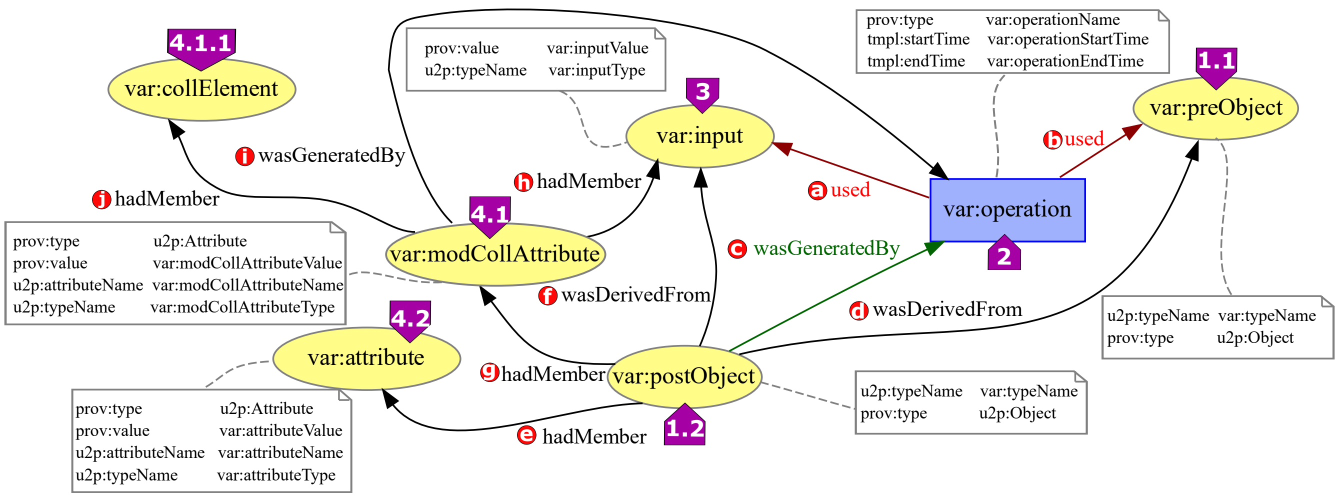


Figure 47. UML representation that models the context given by *ClP10*



4.1.1
var:collElement
prov:value var:inputValue
u2p:typeName var:inputType
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
1.1
var:preObject
3
var:input
i wasGeneratedBy
j hadMember
h hadMember
b used
a used
var:operation
prov:type u2p:Attribute
prov:value var:modCollAttributeValue
u2p:attributeName var:modCollAttributeName
u2p:typeName var:modCollAttributeType
4.1
var:modCollAttribute
c wasGeneratedBy
2
f wasDerivedFrom
d wasDerivedFrom
u2p:typeName var:typeName
prov:type u2p:Object
4.2
var:attribute
g hadMember
var:postObject
u2p:typeName var:typeName
prov:type u2p:Object
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:typeName var:attributeType
e hadMember
1.2

Figure 48. PROV template generated from the UML representation used in *ClP10* (Figure 47)

PROV elements

UML	PROV / id	Rationale
`Class` 1	`prov:Entity` 1.1 / `var:preObject`	The *Pre-operation object*, i.e. the object with the status before the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:preObject`.
	`prov:Entity` 1.2 / `var:postObject`	The *Post-operation object*, i.e. the object with the status after the execution of the operation, which is represented by `Class` 1, is a `prov:Entity` identified as `var:postObject`.
`Operation` 2 «add»	`prov:Activity` 2 / `var:operation`	The execution of `Operation` 2 stereotyped by «add» is a `prov:Activity` identified by `var:operation`.
`Input Parameters` 3	`prov:Entity` 3 / `var:input`	Each parameter of `Input Parameters` 3 is a separate `prov:Entity` identified as `var:input`.
`Attributes` 4	`prov:Entity` 4.1 / `var:modCollAttribute`	The *Modified collection attribute* (belonging to `Attributes` 4) is a `prov:Entity` with identifier `var:modCollAttribute`. Additionally, each element in this collection is a separate `prov:Entity` identified by `var:collElement` 4.1.1
	`prov:Entity` 4.2 / `var:attribute`	Each *Unmodified attribute* (belonging to `Attributes` 4) is mapped to a separate `prov:Entity` with identifier `var:attribute`.

Attributes

PROV Element	Attribute / Value	Description
var:preObject 1.1	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:preObject 1.1 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:preObject 1.1 is an object.
var:postObject 1.2	u2p:typeName / var:className	The value var:className is the string with the name of the class to which var:postObject 1.2 belongs.
	prov:type / u2p:Object	The value u2p:Object shows that var:postObject 1.2 is an object.
var:operation 2	prov:type / var:operationName	The value var:operationName is the name of the operation var:operation 2.
	tmpl:startTime / var:operationStartTime	The var:operationStartTime is an xsd:dateTime value for the start of var:operation 2.
	tmpl:endTime / var:operationEndTime	The var:operationEndTime is an xsd:dateTime value for the end of var:operation 2.
var:input 3	prov:value / var:inputValue	The value var:inputValue is the direct representation of var:input 3.
	u2p:typeName / var:inputType	The value var:inputType is the string with the name of the type of var:input 3.
var:modCollAttribute 4.1	prov:type / u2p:Attribute	The value u2p:Attribute shows that var:modCollAttribute 4.1 is an attribute.
	prov:value / var:modCollAttributeValue	The value var:modCollAttributeValue is the direct representation of var:modCollAttribute 4.1.
	u2p:attributeName / var:modCollAttributeName	The value var:modCollAttributeName is the string with the name of var:modCollAttribute 4.1.
	u2p:typeName / var:modCollAttributeType	The value var:modCollAttributeType is the string with the name of the type of var:modCollAttribute 4.1.
var:attribute 4.2	prov:type / u2p:Attribute	The value u2p:Attribute shows that attribute 4.2 is an attribute.
	prov:value / var:attributeValue	The value var:attributeValue is the direct representation of attribute 4.2.
	u2p:attributeName / var:attributeName	The value var:attributeName is the string with the name of attribute 4.2.
	u2p:typeName / var:attributeType	The value var:attributeType is the string with the name of the type of attribute 4.2.

PROV relations

PROV Relation	Description
ⓐ prov:used	It is the beginning of utilizing var:input by var:operation.
ⓑ prov:used	It is the beginning of utilizing var:preObject by var:operation.
ⓒ prov:wasGeneratedBy	It is the completion of production of var:postObject by var:operation.
ⓓ prov:wasDerivedFrom	It is the update of var:preObject resulting in var:postObject.
ⓔ prov:hadMember	It states that var:attribute is one of the elements in var:postObject.
ⓕ prov:wasDerivedFrom	It is the construction of var:postObject based on var:input.
ⓖ prov:hadMember	It states that var:modCollAttribute is one of the elements in var:postObject.
ⓗ prov:hadMember	It states that var:input is one of the elements in var:modCollAttribute. This is due to the fact that in this context the input information is directly added to the object's collection attribute.
ⓘ prov:wasGeneratedBy	It is the completion of production of var:modCollAttribute by var:operation.
ⓙ prov:hadMember	It states that var:collElement is one of the elements in var:modCollAttribute.

Discussion

- Among the Class Diagrams patterns, patterns from *ClP6* to *ClP10* address the execution of operations that change an object's status. While, *ClP6* changes the object's status as a whole (being the concrete modified attributes unknown or irrelevant), in patterns *ClP7*-*ClP10* the concrete attributes modified by the *Operation execution* are explicitly known. In contrast to *ClP7* which directly sets the information passed into the *Operation execution* as values of concrete object's attributes, the other mentioned patterns use such information to change the object's status as a whole or the values of concrete object's attributes. It must also be noted that patterns *ClP9* and *ClP10* address the execution of operations which remove or add elements from/into an object's collection attribute, while patterns *ClP7* and *ClP8* affect either a univalued attribute or a collection attribute as a whole.
- A question that might arise is why in Figure 48 var:attribute 4.2 is associated with var:postObject 1.2 (which represents the object with the status after the execution of the operation), but it is not associated with var:preObject 1.1 (the object with the status before the execution). We have made this decision because an object that acts as a var:preObject in an operation execution, was a var:postObject in a previous operation execution. Thus, the attributes associated to such an object in a var:preObject were registered when it previously played the role of var:postObject.
- Although the *context* of this pattern does not explicitly state that output data should be obtained from the *Operation execution*, this could be the case. However, we do not include this output data in this pattern description to avoid overburden both the UML and PROV explanations with information out of the scope of the *context*.

  Aiming at giving an insight into how the inclusion of *Output data* affects both UML representation and the resulting PROV template, Figure 49 depicts a UML representation with the *Output data* modelled as Output Parameters 5 (in this case with *return* direction, though the translation of *inout* and *out* directions would be equivalent). Figure 50 depicts its transformation into PROV. Both Figure 49 and 50 highlight the elements related to the inclusion of the *Output data* by blurring the elements coming from Figure 47 and 48, respectively.

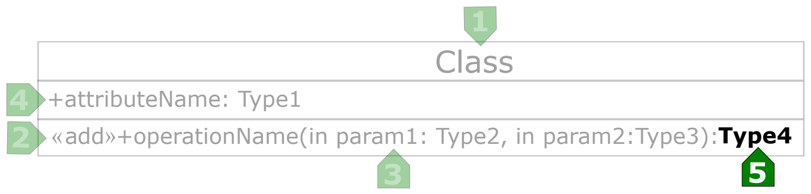

1
Class
4
+attributeName: Type1
2
«add»+operationName(in param1: Type2, in param2:Type3):Type4
3
5

Figure 49. UML representation that models the context given by *ClP10*, including Output Parameters.

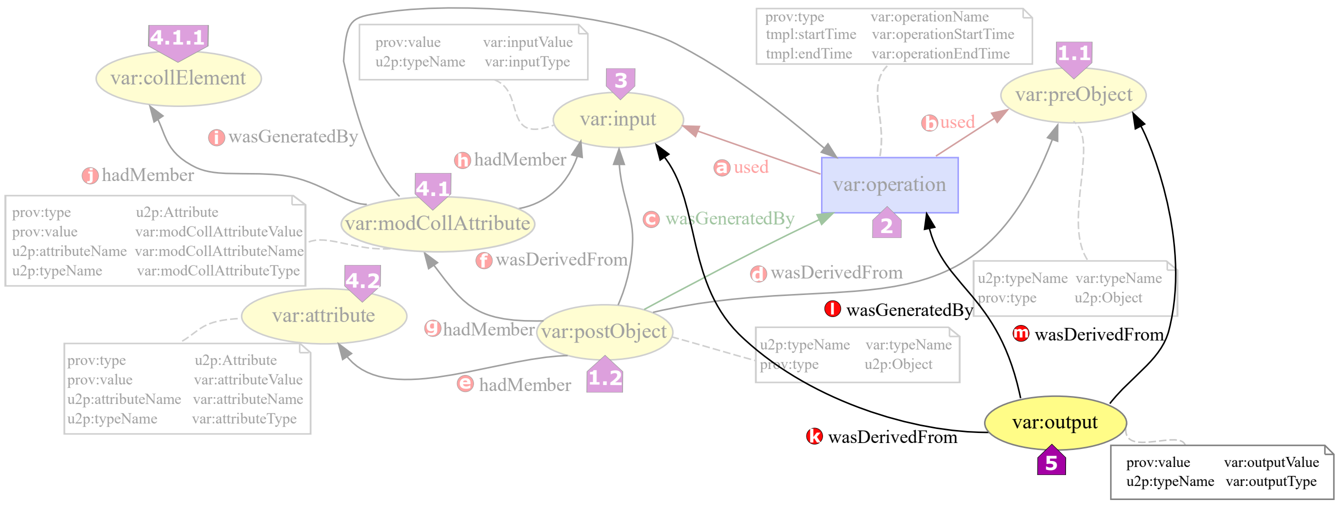

4.1.1
var:collElement
i wasGeneratedBy
j hadMember
prov:type u2p:Attribute
prov:value var:modCollAttributeValue
u2p:attributeName var:modCollAttributeName
u2p:typeName var:modCollAttributeType
prov:value var:inputValue
u2p:typeName var:inputType
3
var:input
h hadMember
4.1
var:modCollAttribute
4.2
var:attribute
prov:type u2p:Attribute
prov:value var:attributeValue
u2p:attributeName var:attributeName
u2p:typeName var:attributeType
f wasDerivedFrom
g hadMember
var:postObject
e hadMember
1.2
prov:type var:operationName
tmpl:startTime var:operationStartTime
tmpl:endTime var:operationEndTime
a used
c wasGeneratedBy
var:operation
2
b used
1.1
var:preObject
d wasDerivedFrom
l wasGeneratedBy
u2p:typeName var:typeName
prov:type u2p:Object
m wasDerivedFrom
k wasDerivedFrom
var:output
5
prov:value var:outputValue
u2p:typeName var:outputType

Figure 50. PROV template generated from the UML representation used in *ClP10*, including `Output Parameters` (Figure 49)

PROV elements

UML	PROV / id	Rationale
`Output Parameters` 5	`prov:Entity` 5 / `var:output`	Each parameter of `Output Parameters` 5 is a separate `prov:Entity` identified as `var:output`.

PROV relations

PROV Relation	Description
k `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:input`.
l `prov:wasGeneratedBy`	It is the completion of production of `var:output` by `var:operation`.
m `prov:wasDerivedFrom`	It is the construction of `var:output` based on `var:preObject`.

Attributes

PROV Element	Attribute / Value	Description
`var:output` 5	`prov:value` / `var:outputValue`	The value `var:outputValue` is the direct representation of `var:output` 5.
	`u2p:typeName` / `var:outputType`	The value `var:outputType` is the string with the name of the type of `var:output` 5.

References

[1] OMG, “Unified Modeling Language (UML). Version 2.5,” 2015. Document formal/15-03-01, March, 2015.

[2] L. Moreau and P. Missier (eds.), “PROV-DM: The PROV Data Model,” W3C Recommendation REC-prov-dm-20130430, World Wide Web Consortium, 2013.

[3] N. Kwasnikowska, L. Moreau, and J. V. D. Bussche, “A formal account of the open provenance model,” *ACM Trans. Web*, vol. 9, pp. 10:1–10:44, May 2015.

[4] PROV Graph Conventions. Available at `www.w3.org/2011/prov/wiki/Diagrams`. Last accessed April, 2019.

[5] M. Dürst and M. Suignard., “Internationalized Resource Identifiers (IRIs) (RFC 3987).” January, 2005.

[6] A. Knapp and S. Merz, “Model checking and code generation for uml state machines and collaborations,” *Proc. 5th Wsh. Tools for System Design and Verification*, pp. 59–64, 2002.

[7] N. Dragan, M. L. Collard, and J. I. Maletic, “Automatic identification of class stereotypes,” in *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pp. 1–10, 2010.

Appendix A Taxonomy of Class' operations

Depending on their nature, operations implicitly have specific semantics that can also provide information of interest for provenance capture. In order to provide UML Class diagrams with such additional semantics to be included in the generated PROV templates, we have stated a taxonomy of operations given by a set of stereotypes to be included in such diagrams. The taxonomy is based on that given by Dragan et al. [7], which has been enriched with additional stereotypes aimed at identifying extra/further operation's semantics not considered in [7] (marked with an asterisk in Table 2).

Table 2. Extension of the taxonomy given in [7] showing the categories of UML Class's operations considered in our proposal. Stereotypes with an asterisk denote those included by our proposal.

Category	Stereotype name	Description
Creational	create	The operation creates an object.
	destroy	The operation destroys an object.
Structural *Accessor*	get	The operation returns values of concrete attributes of an object.
	search*	The operation returns elements belonging to a concrete collection attribute of an object.
	process*	The operation returns values that are computed based the object's status as a whole (the specific attributes used for the calculation are not relevant).
	predicate	The operation returns boolean values that are computed based on concrete attributes of an object.
	property	The operation returns values (of any type) that are computed based on concrete attributes of an object.
	void-accessor	The operation returns values (of any type) that are computed based on concrete attributes of an object. These values are returned by means of parameters.
Structural *Mutator*	command	The operation changes the status of an object as a whole (the modified attributes are unknown or irrelevant). It does not return information.
	non-void-command	The operation changes the status of an object as a whole (the modified attributes are unknown or irrelevant). It does return information.
	set	The operation directly sets the information passed to the operation as values of concrete attributes of an object.
	modify*	The operation modifies concrete attributes of an object.
	remove*	The operation removes an element from a concrete collection attribute of an object.
	add*	The operation adds an element on a concrete collection attribute of an object.

INTEGRATING PROVENANCE CAPTURE AND UML WITH UML2PROV: PRINCIPLES AND EXPERIENCE

IMPLEMENTATION DETAILS

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

[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@kcl.ac.uk

This appendix provides a detailed description of the Model Driven Development (MDD) approach we have followed for implementing UML2PROV, which we succinctly explained in the paper [1]. This approach is presented schematically in Figure 1, below. MDD focuses on models, rather than on computer programs, so that the code programs are automatically generated from them by using a refinement process [2]. This process could entail one or various transformations that describe the way in which a source model is translated into another final target. Depending on the type of source and target elements of the transformation, we can distinguish between *model to model transformations* (M2M), in which both are models, and *model to text transformations*, which define transformations from a model to a final text.

Our solution for implementing UML2PROV following an MDD approach comprises both M2M and M2T transformations. Among the different existing tools to implement M2M and M2T transformations, we have used the Atlas Transformation Language (ATL) [3] and Xtend [4]. On the one hand, in case of M2M transformations, we have used

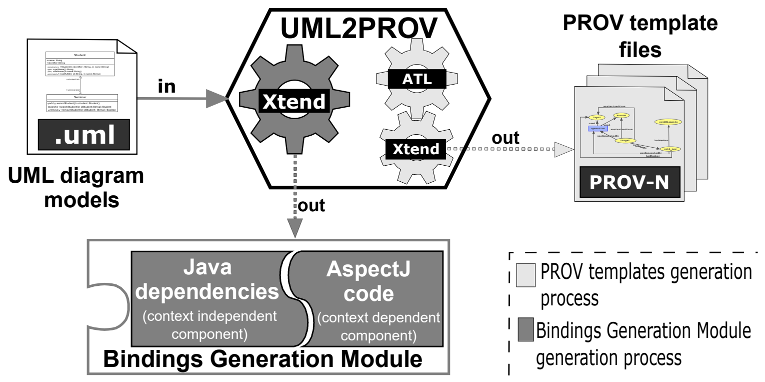

UML2PROV
Xtend
ATL
Xtend
in
out
out
.uml
UML diagram models
PROV template files
PROV-N
Java dependencies
(context independent component)
AspectJ code
(context dependent component)
Bindings Generation Module
PROV templates generation process
Bindings Generation Module generation process

Figure 1: Our MDD-based implementation proposal.

ATL [3] for being one of the most widely used M2M transformation languages, in addition to provide an IDE developed on top of Eclipse. On the other hand, M2T transformations have been implemented by means of Xtend [4] for several reasons, among which we note that it integrates seamlessly with the Eclipse Java IDE, and that it has a large user community and a significant number of available examples.

Next, in Section 1, we explain the MDD transformations we have defined to implement our UML to PROV templates transformation patterns. Later, in Section 2, we first give details regarding our strategy to implement the BGM (*Bindings Generation Module*) for an application and, second, we provide our MDD-based proposal to automatically generate it.

1 Automatization of the UML to PROV Templates transformation patterns

Generally speaking, our proposal for implementing our transformation patterns takes as source the *UML diagram models* of the application and automatically generates the *PROV template files* (Figure 1). Instead of performing a one-step direct transformation between such source and target elements, we have decided to define an intermediate step by means of which *UML diagram models* are first translated into a transitional model (*template models*) which will be finally translated into the *PROV template files* in PROV-N. This strategy allows us to draw a distinction between the translation from *UML diagram models* into *template models*, and the way in which the *template models* are serialised, in this case PROV-N. As a result, our proposal follows an MDD-based tool chain that comprises two transformations (see Figure 2): first, an M2M transformation identified by `T1`, whose implementation is explained in Subsection 1.1, and second, an M2T transformation identified by `T2`, which is explained in Subsection 1.2.

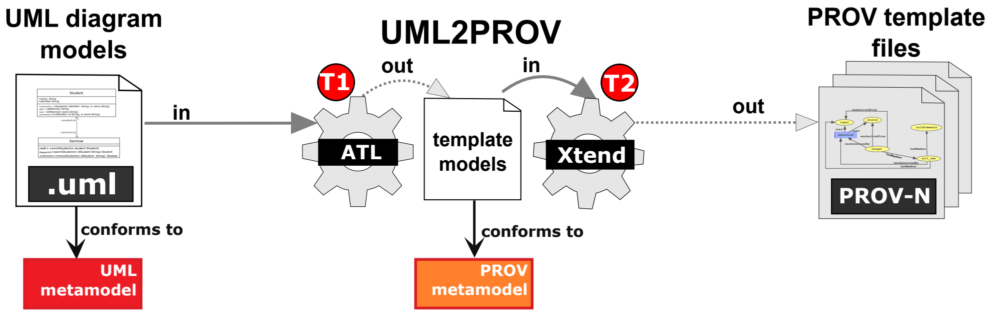


Figure 2: Detailed MDD-based implementation of the PROV templates generation process

1.1 Transformation `T1`: from UML diagram models to template models

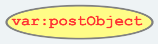
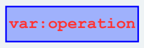
This M2M transformation takes as source the *UML diagram models*, conforming to the UML metamodel [5], and generates the corresponding *template models*, conforming to the PROV metamodel [6]. To that end, our transformation patterns [7] serve as the basis for the definition of an ATL module made up of a set of ATL rules. Each rule addresses one transformation pattern describing how the UML elements identified by the pattern are mapped to the specific PROV elements, and their relations, constituting a *template model*.

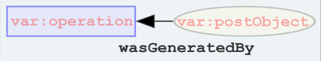
As an example of such ATL rules, Table 1 shows how an excerpt of the ATL rule defined to implement the *ClP1* pattern looks like. Such a pattern deals with operations that construct an object. This table depicts, per each fragment of the

Table 1: An excerpt of the ATL rule implementing *ClP1*. For each fragment in the excerpt ("ATL source code" column), the PROV elements it generates are provided ("Template model" column) together with a description of the transformation ("Description" column) as well as the graphical notation of the template model (last column).

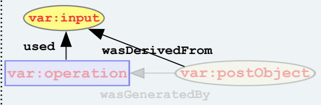
ATL source code	Template model	Description	Graphical representation of the template model
rule Operation2Document{			

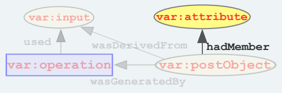
from
operation: UML!Operation(
operation.hasStereotype('create'))
		It states that the rule is applied to all the UML operations to which ClP1 refers. That is, those operations with the stereotype «create»	

[...]	<**document** id= "....">	It creates the PROV **<document>**.	
to			
postObjectEn: **PROV!Entity** (id <- 'var:postObject', ...),	<**entity** id="var:postObject"/>	It creates an **<entity>** with identifier `var:postObject`.	
operationAct: **PROV!Activity** (id <- 'var:operation', ...),	<**activity** id="var:operation"/>	This excerpt is in charge of generating an **<activity>** identified by `var:operation`.	

wgb: **PROV!Generation** (
entity <- postObjectEnID,
activity <- operationActID),
	<**wasGeneratedBy**>
	<entity ref="var:postObject"/>
	<activity ref="var:operation"/>
	</activity>
		This excerpt is responsible for linking `var:postObject` with `var:operation` by means of the PROV relation **<wasGeneratedBy>**.	

do{			

if(existIn){
thisModule.**newEnt**('var:input', doc);
 	
thisModule.**genDer**('var:postObject',
'var:input', doc);
 	
thisModule.**genU**('var:input', '
var:operation', doc);
}
	<**entity** id="var:input"/>
	<**wasDerivedFrom**>
	<generatedEntity ref="var:postObject"/>
	<usedEntity ref="var:input"/>
	</wasDerivedFrom>
	<**used**>
	<activity ref="var:operation"/>
	<entity ref="var:input"/>
	</used>
		In case there are UML Input Parameters, it creates an **<entity>** identified by `var:input`*, and its relations **<wasDerivedFrom>** and **<used>** with `var:postObject` and `var:operation`, respectively.	 var:input
			used
			wasDerivedFrom
			var:operation
			var:postObject
			wasGeneratedBy

if(hasAttr){
thisModule.newEnt('var:attribute',
doc);
 	
thisModule.**genMe**('var:attribute',
'var:postObject',
doc);
}
	<**entity** id="var:attribute"/>
	<**hadMember**>
	<collection ref="var:postObject"/>
	<entity ref="var:attribute"/>
	</hadMember>
		If there are UML Attributes in the class to which the operation belongs, it generates an **<entity>** identified by `var:attribute`*, and the PROV relation **<hadMember>** between `var:postObject` and `var:attribute`.	 var:input
			var:attribute
			used
			wasDerivedFrom
			hadMember
			var:operation
			var:postObject
			wasGeneratedBy

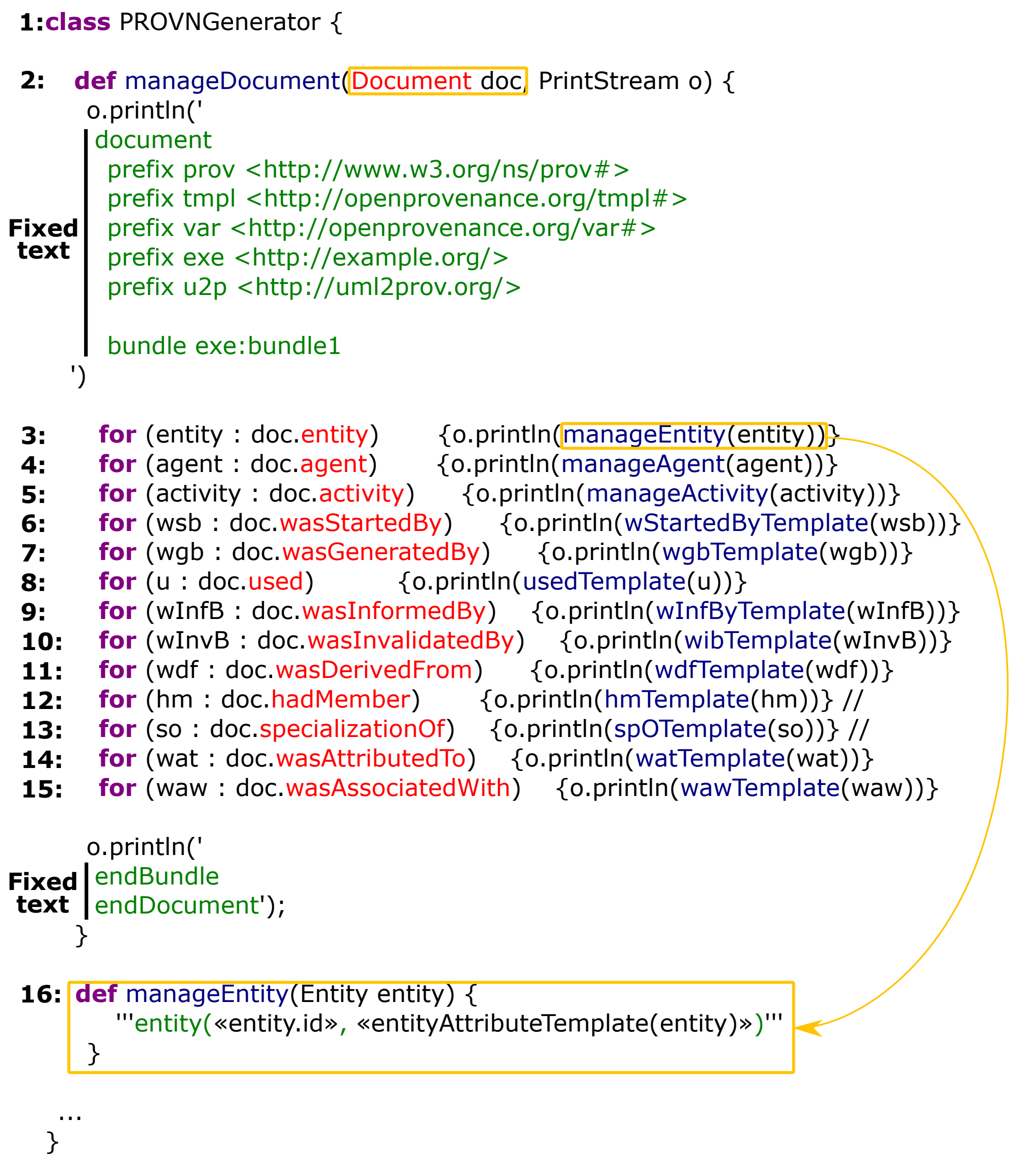
[...]	</document>		

*Although ClP1 states that each attribute/input parameter is a separate prov:Entity identified as var:attribute/var:input, we have decided to merge all the entities with the same identifier. Nevertheless, this decision does not have any effect on the bindings, since each var:input and var:attribute will be given several values (one for each input parameter and attribute, respectively).

rule (see first column), the PROV elements/relations in the template that are generated by such a fragment (see column "Template model"). We can see how PROV elements such as document, entity and activity, as well as PROV relations such as used and wasGeneratedBy, appear in the column as *<document>*, *<entity>*, *<activity>*, *<used>*, and *<wasGeneratedBy>*. Additionally, the description of the transformation together with the graphical notation of the template model being generated are given in the two right-hand columns.

1.2 Transformation `T2`: from template models to PROV template files

`T2` corresponds to an M2T transformation that takes as source the *template models* resulting from `T1`, and generates the *PROV template files* in PROV-N format. This transformation is implemented in an Xtend class (see Figure 3) which contains *template expressions* that associate each PROV element/relation with its associated PROV-N representation. Among the defined Xtend template expressions (declared by the explicit keyword `def`), there is a main template (line 2) which is in charge of translating each PROV *<document>* appearing in the *template models* into a *PROV template*, defined as a `.provn` extension text file. This PROV-N document will include not only fixed text (shown in green in Figure 3), but also the text resulting from instantiating those Xtend templates in charge of translating the PROV elements/relations included in the *<document>* (lines from 3 to 15). As a way of example, Figure 3 also depicts the Xtend template (line 16) which translates each *<entity>* into the corresponding `prov:Entity` in PROV-N.



```
class PROVNGenerator {

  def manageDocument(Document doc, PrintStream o) {
   o.println('
   document
    prefix prov <http://www.w3.org/ns/prov#>
    prefix tmpl <http://openprovenance.org/tmpl#>
    prefix var <http://openprovenance.org/var#>
    prefix exe <http://example.org/>
    prefix u2p <http://uml2prov.org/>

    bundle exe:bundle1
  ')

    for (entity : doc.entity)        {o.println(manageEntity(entity))}
    for (agent : doc.agent)          {o.println(manageAgent(agent))}
    for (activity : doc.activity)    {o.println(manageActivity(activity))}
    for (wsb : doc.wasStartedBy)     {o.println(wStartedByTemplate(wsb))}
    for (wgb : doc.wasGeneratedBy)   {o.println(wgbTemplate(wgb))}
    for (u : doc.used)               {o.println(usedTemplate(u))}
    for (wInfB : doc.wasInformedBy)  {o.println(wInfByTemplate(wInfB))}
    for (wInvB : doc.wasInvalidatedBy)  {o.println(wibTemplate(wInvB))}
    for (wdf : doc.wasDerivedFrom)   {o.println(wdfTemplate(wdf))}
    for (hm : doc.hadMember)         {o.println(hmTemplate(hm))} //
    for (so : doc.specializationOf)  {o.println(spOTemplate(so))} //
    for (wat : doc.wasAttributedTo)  {o.println(watTemplate(wat))}
    for (waw : doc.wasAssociatedWith)  {o.println(wawTemplate(waw))}

   o.println('
   endBundle
   endDocument');
  }

  def manageEntity(Entity entity) {
    '''entity(«entity.id», «entityAttributeTemplate(entity)»)'''
  }

 ...
}
```

Figure 3: Xtend class including the template defined for each *<document>* and *<entity>* in the *template models*.

2 BGM generation automation

Here, we explain in detail a reference implementation for the automatic generation of the BGM corresponding to a certain application, starting from its UML design. Below, we will explain our strategy for implementing the BGM for a concrete application starting from its UML design (Subsection 2.1), and later we move on to describe the process we have defined to automatically generate a BGM (Subsection 2.2).

2.1 Towards an implementation of the BGM

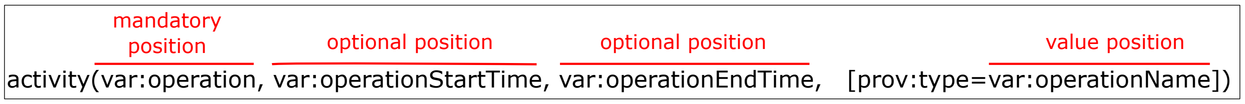
Aimed at providing an implementation of a BGM, there are several issues a developer may consider to manage the provenance data for creating bindings. The first one is referred to when and how the bindings are generated and stored. For example, applications may store the provenance data using usual logs, delaying the construction of bindings until after runtime. Alternatively, applications could directly construct the bindings at runtime. The second aspect refers to when provenance documents are generated and which storage system is used. For example, the bindings could be accumulated locally in memory, delaying the generation of the provenance documents (i.e., the expansion of templates), and thus their storage (e.g., database, files,...) until after runtime. Alternatively, the strategy could be to expand the templates with the accumulated bindings on runtime, storing the provenance documents as the application is executed.

Taking into account these issues, we have defined a generic *event*-driven proposal to implement the BGMs. *Events* are notable occurrences that happen while the application is running, whereas *listeners* contain the behaviour for processing the *events*. Concretely, our proposal for capturing the provenance data is driven by the execution of operations, for this reason, we have identified four notable types of occurrences that take place during the execution of an operation, and which correspond to four types of *events*, respectively. Two of these *events* are related to the start and end of an operation, whereas the two remaining *event* types refer to the collection of values associated with the two types of variables stated in [9] (*group variables* and *statement-level variable*). On the one hand, a *group variable* is a type of variable that occurs in a mandatory identifier position. On the other hand, a *statement-level variable* is a variable that occurs in an attribute-value pair (either in attribute position or in value position), or that occurs in optional identifier position. So as to give an insight into them, Figure 4 shows a `prov:Activity` in PROV-N with variables ocurring in different positions.

In this context, the *event* types we have identified are the following:

- (1) *operationStart* and (2) *operationEnd*. These types of *events* refer to the start and end of an operation execution, respectively. They are of interest when developers want to create and store sets of bindings associated with a concrete operation execution, instead of storing each binding independently.
- (3) *newBinding*. This type of *event* refers to the occurrence of the collection of a provenance value associated with a *group variable*. For instance, the collection of a value associated with the variable `var:operation` in Figure 4 will trigger an *event* of type *newBinding* since `var:operation` occurs in an identifier position.
- (4) *newValueBinding*. This type of *event* refers to the occurrence of the collection of a provenance value associated with a *statement-level variable*. For instance, the collection of values linked with `var:operationStartTime` and `var:operationName` in Figure 4 fires *newValueBinding events* due to `var:operationStartTime` occurring in an optional position, and `var:operationName` occurring in a value position.

Our reference implementation of BGM is made up of four main components written in Java (see Figure 5) which are divided into two main groups. The first group, which is referred to as *context independent components*, is made up



```
activity(var:operation, var:operationStartTime, var:operationEndTime,  [prov:type=var:operationName])
```

Figure 4: PROV activity in PROV-N [8] with different types of variables. Additionally, it is shown a table associating each variable with its type.

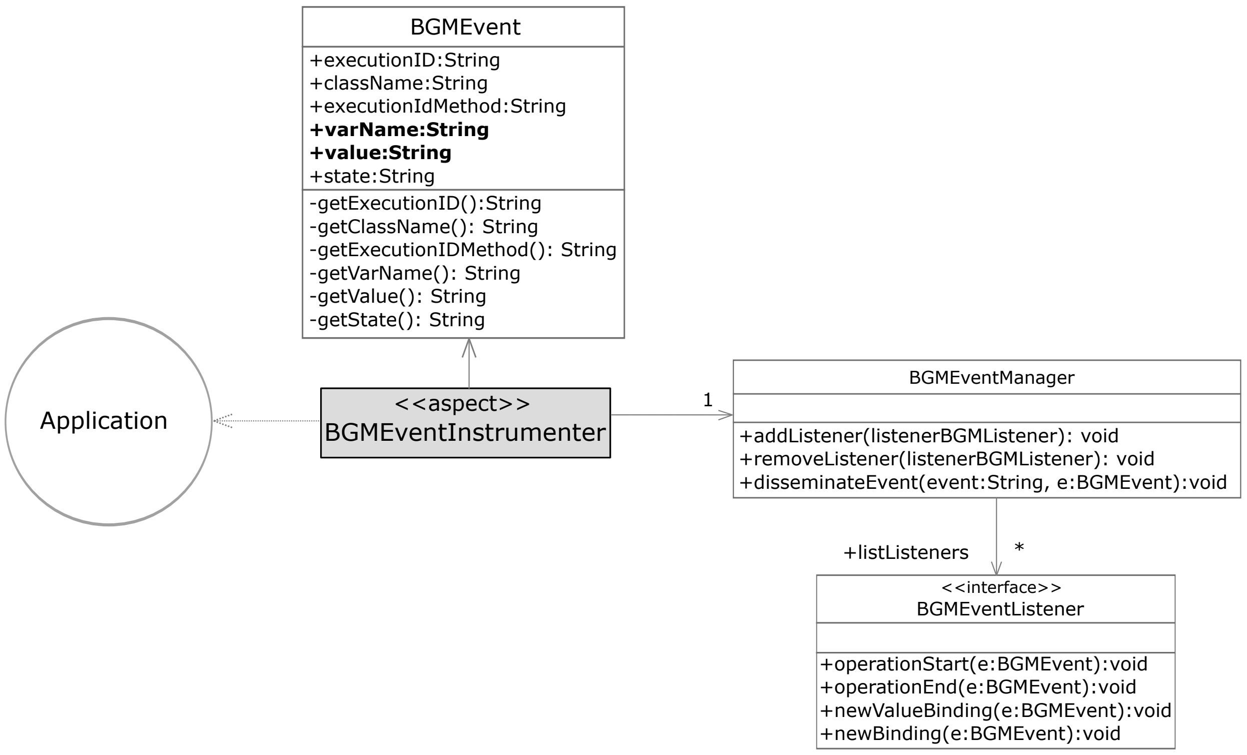
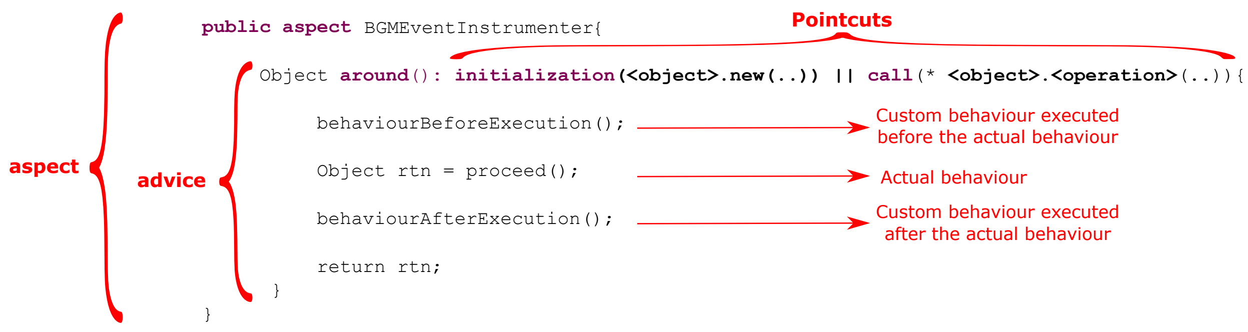

BGMEvent
+executionID:String
+className:String
+executionIdMethod:String
+varName:String
+value:String
+state:String
-getExecutionID():String
-getClassName(): String
-getExecutionIDMethod(): String
-getVarName(): String
-getValue(): String
-getState(): String
Application
<<aspect>>
BGMEventInstrumenter
1
BGMEventManager
+addListener(listenerBGMListener): void
+removeListener(listenerBGMListener): void
+disseminateEvent(event:String, e:BGMEvent):void
+listListeners
*
<<interface>>
BGMEventListener
+operationStart(e:BGMEvent):void
+operationEnd(e:BGMEvent):void
+newValueBinding(e:BGMEvent):void
+newBinding(e:BGMEvent):void

Figure 5: UML CD depicting our reference implementation for the BGM.

of those elements that do not depend on the source *UML diagram models*, and therefore, they are the same in all the BGMs. This group is made up of the *BGMEventListener*, *BGMEvent*, and *BGMEventManager* (see components in white background in Figure 5). The second group, called *context dependent components*, consists of those elements whose implementation depends on the source *UML diagram models*. In our reference implementation the only element included in this group is the *BGMEventInstrumenter* (depicted in grey background in Figure 5).

- *BGMEventListener*. It is an interface that defines four operations for managing each type of *event* (*operationStart*, *operationEnd*, *newBinding*, and *newValueBinding*). These operations have an input parameter of type *BGMEvent* (see below) that contains the provenance data to be processed. The implementation of these operations constitutes the mechanism used by a class implementing the *listener* interface to generate, manage, and store the bindings. As commented before, the developer just needs to choose the mechanisms that best suits her/his requirements by developing classes implementing the *BGMEventListener* interface. Later, in Section 2.1.1, we will give a reference implementation of this interface. At this point, we remark that with the aim of simplifying the design, we group all the operations for managing the abovementioned *event* types in the same interface (*BGMEventListener*). In case a developer is not interested in handling a concrete *event*, she/he can leave empty the implementation of its corresponding operation.

- *BGMEvent*. This component is used to carry information about the occurrence of an *event*. We have decided to use the same class *BGMEvent* to contain information about the four event types (*operationStart*, *operationEnd*, *newBinding*, *newValueBinding*) because this information can be stored using the same structure. Concretely, this structure will contain the provenance data necessary for constructing the bindings. Among them, we remark the attribute



```
public aspect BGMEventInstrumenter{

    Object around(): initialization(<object>.new(..)) || call(* <object>.<operation>(..)){

        behaviourBeforeExecution();

        Object rtn = proceed();

        behaviourAfterExecution();

        return rtn;
    }
}
```

Pointcuts
aspect
advice
Custom behaviour executed before the actual behaviour
Actual behaviour
Custom behaviour executed after the actual behaviour

Figure 6: Structure overview of a reference implementation of the *BGMEventInstrumenter* in AspectJ

`varName` for the name of the variable, and the attribute `value` for the value associated with such a variable. See Figure 5. For instance, in case of an *operationStart event*, a *BGMEvent* object could have an attribute `varName` containing the value `"var:operationStartTime"`, and an attribute `value` with the value `"2018-12-20T12:54:20"`. Another example could be a *BGMEvent* object with information about a *newBinding event*. It could contain an attribute `varName` with the value `"var:operation"`, and the attribute `value` containing `"exe:nameOfOperation-300691"`.

- *BGMEventManager*. In some cases, to have only one *listener* for generating, managing, and storing bindings could be not enough, and the same happens with the mechanisms for generating and storing provenance. For instance, one provenance consumer may be interested in replicating the information by storing both the provenance data, and the bindings generated from them in different storage systems. Aiming at addressing these scenarios, we have included the *BGMEventManager* with two responsibilities: to manage a list of subscribed *listeners*, and to disseminate the objects of type *BGMEvent* among them.

- *BGMEventInstrumenter*. As we stated in the paper [1], to manually adapt the source code of an application would be a valid option to capture provenance data. However, this option would require to traverse the whole code of the concrete application identifying the classes that will be the source of the events, and additionally, those places inside these classes where *events* will be fired. Then, the manual adaptation of the source code would need to include in those places instructions for constructing *BGMEvent* objects with the provenance data, and disseminating them among the *listeners*. This task constitutes a tedious, time-consuming and error-prone process. What is worse, the manual adaptation could incur in such provenance capture code instructions scattered across all the application classes, making their maintenance a cumbersome task.

  In contrast, we propose to use the Aspect Oriented Programming (AOP) [10] paradigm for implementing what we have named *BGMEventInstrumenter*. AOP aims at improving the modularity of software systems, by capturing inherently scattered functionality, often called *cross-cutting concerns*, (e.g., the capture of provenance), and placing that functionality apart from the actual application's source code. Our reference implementation is developed in AspectJ, an AOP extension created for Java [11], and it consists of an *aspect* which is made up of an *advice* with *pointcuts* (see Figure 6). On the one hand, the *pointcuts* identify locations within the application code where a concern may be included. In our case, we identify operation calls and constructor invocations from which we want to fire *events* (i.e., to collect provenance). On the other hand, the *advice* is the behaviour executed when the *pointcuts* are matched. In AspectJ, *advices* can be executed at three different places: *before*, *around*, and *after* the *pointcuts*. Due to the fact that our identified *events* can occur both before and after operations calls and constructors invocations, we have used an *around advice* for executing custom behaviours before and after the actual behaviour. These custom behaviours consist of constructing objects of type *BGMEvent* and disseminating them to the *listeners* (by invoking the `disseminateEvent` operation from *BGMEventManager*). In the end, as a pre-compilation step, the AspectJ *weaver* automatically integrates the behaviour from the *aspects* into the locations specified by the *pointcuts* at compilation time. In this way, our AOP approach does not require a manual intervention for adapting the source code, and automatically collects provenance data in a transparent way for software developers, which directly incurs in fulfilling the requirements *R1*-*R3* stated in the paper [1].

2.1.1 Example of a class implementing the *BGMEventListener*

Taking into account the structure depicted in Figure 5, we provide the users with a concrete implementation of the interface *BGMEventListener* (class that we name *ConcreteBGMEventListener*). This class implements the four operations defined in the *BGMEventListener* so that the bindings are generated and accumulated in memory, and when the execution of each tracked operation finishes, they are shipped to the MongoDB database. Thus, this implementation is only in charge of generating and storing bindings, delaying the expansion of templates. In this way, the users can decide not only which templates to expand, but also when to expand them.

2.2 Automatization of the implementation of the BGM

The BGM for an application is automatically generated by means of an M2T transformation referred to as `T3` in Figure 7. Such a transformation has been implemented by means of an Xtend class that takes as source the application's *UML diagram models*, conforming the UML metamodel [5], and generates the java code of the BGM.

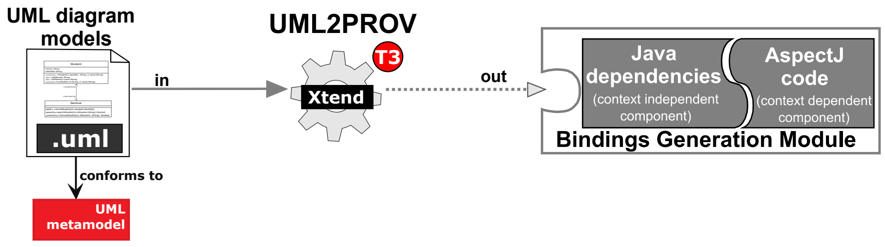


Figure 7: Detailed MDD-based implementation of the BGM for an application.

As we stated previously, the source code of the *context independent components* (i.e., *BGMEvent, BGMEventManager*, and *BGMEventListener*) is the same for all the BGMs; thus, it does not depend on the *UML diagram models* used as input in the transformation. Conversely, the implementation of the *context dependent component* (i.e., *BGMEventInstrumenter*) depends on the source *UML diagram models*.

Our strategy for automatically generating the BGM is to implement an Xtend class that (1) directly creates all the *context independent components*, and (2) generates the *BGMEventInstrumenter* based on the source UML design. Whilst we could have provided users with a separate library including all the *context independent components*, we have made the decision of generating them automatically together with the *BGMEventInstrumenter* in order to reduce the code dependencies.

In particular, the Xtend class generates the *BGMEventInstrumenter* so that its *pointcuts* identify the calls to operations and invocations of constructors. Concretely, these *pointcuts* correspond to (1) the invocations of the constructors of classes involved in the UML design, and (2) calls of operations that are involved in the source: SqDs (i.e., the operations whose calls are modelled by means of `UML Messages`), SMDs (i.e., the operations whose occurrences are associated with `UML Events`), and CDs, (i.e., the operations that are modelled by `UML Operations`). The remainder source code of the *BGMEventInstrumenter* (that is, the *advise*) is also shared by all the BGMs.

2.3 Fulfilment of BGM requirements

The reference implementation of the BGMs given in this document fulfils the five requirement stated in the paper [1] (identified from *R1* to *R5*). As we previously stated, requirements from *R1* to *R3* have been met thanks to the AOP implementation of the *BGMEventInstrumenter* explained in Section 2.1. Regarding the requirements *R4* and *R5*, we note that they have been satisfied because of the suitable ad hoc implementation of the *BGMEventInstrumenter* for a concrete application (explained in Section 2.2). On the one hand, the automatically generated *pointcuts* inside the *BGMEventInstrumenter* ensures that the collected bindings are associated with at least one PROV template (requirement *R4*). This is because the *pointcuts* correspond to operations calls and constructors invocations that are modelled in the UML design, and therefore they have an associated PROV template. On the other hand, the requirement *R5* is fulfilled since the transformation `T3` has been implemented so that it respects the names of the variables appearing in the PROV templates generated by the chain of transformations `T1-T2`.

References

[1] C. Sáenz-Adán, B. Pérez, F. J. García-Izquierdo, and L. Moreau, “Integrating Provenance Capture and UML with UML2PROV: Principles and Experience,” submitted for publication in IEEE Transactions on Software Engineering.

[2] B. Selic, “The pragmatics of model-driven development,” *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[3] ATL - a model transformation technology, version 3.8. Available at `http://www.eclipse.org/atl/`. Last visited on January 2020.

[4] Xtend, “General-purpose high-level programming language.” Available at `https://www.eclipse.org/xtend/`. Last visited on January 2020.

[5] OMG, “Unified Modeling Language (UML). Version 2.5,” 2015. Document formal/15-03-01, March, 2015.

[6] L. Moreau, P. Missier (eds.), K. Belhajjame, R. B’Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, J. McCusker, S. Miles, J. Myers, S. Sahoo, and C. Tilmes, “PROV-DM: The PROV Data Model,” W3C Recommendation REC-prov-dm-20130430, World Wide Web Consortium, 2013.

[7] Supplementary material for the paper entitled “Supplementary material of Integrating Provenance Capture and UML with UML2PROV: Principles and Experience” containing the *Description of patterns*, submitted for publication in IEEE Transactions on Software Engineering. Available at `https://uml2prov.unirioja.es/`. Last visited on January 2020.

[8] L. Moreau, P. Missier (eds.), J. Cheney, and S. Soiland-Reyes, “PROV-N: The Provenance Notation,” W3C Recommendation REC-prov-n-20130430, World Wide Web Consortium, Apr. 2013.

[9] D. Michaelides, T. D. Huynh, and L. Moreau, “PROV-TEMPLATE: A Template System for PROV Documents,” 2014. Available at `https://provenance.ecs.soton.ac.uk/prov-template`. Last visited on January 2020.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *Proc. of the European Conference on Object-Oriented Programming (ECOOP 1997)*, (Berlin, Heidelberg), pp. 220–242, 1997.

[11] The AspectJ Project. Available at `www.eclipse.org/aspectj/`. Last visited on January 2020.

Integrating Provenance Capture and UML with UML2PROV: Principles and Experience

TAXONOMY OF CLASS' OPERATIONS

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

[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@kcl.ac.uk

1 Introduction

Depending on their nature, operations have specific semantics which can also produce information of interest for provenance capture. For instance, the key factors involved in the execution of an operation such as *getName* (accessing an object's attribute) are different from the ones related to *setName* (modifying an object's attribute), and consequently, the provenance information captured from the execution of both operations must be different. Taking this into account, we are interested in identifying a taxonomy of operations which covers the vast varied range of operation's types of interest for provenance capture. Based on this taxonomy, we can define our CDs to PROV template patterns so that the generated templates can provide concrete provenance information depending on the operation's semantics.

2 A taxonomy of operations

To define our taxonomy, we undertook a literature search looking for different categorizations of operations based on their behaviours. We distinguished the approaches that present a more general classification of operations such as [3], from those that provide a more fine grained taxonomy of operations such as [1] or, more remarkably, the work presented by Dragan et al. in [2], which is one of the most complete. Dragan et al.'s taxonomy is based both on how an operation accesses data (i.e., an operation changes the object's status or leaves it unchanged), and on its behavioural characteristics (i.e., creational, structural…). Such a taxonomy is expressed as a classification of operations' `Stereotypes`. These UML `Stereotypes` are extension mechanisms that allow us to complement each CD's `operation` with specific semantic information regarding its category within the taxonomy, thus linking the `operation` with its corresponding semantics. The notation for a `Stereotype` is a string with the stereotype name between a pair of guillemets (e.g., `«add»`).

This taxonomy define five categories: (1) *Creational* refers to operations responsible for creating or destroying objects of the class. (2) *Structural Accessor* refers to operations that return information regarding the attributes of the object to which it belongs, without changing the state of the object. (3) *Structural Mutator* corresponds to operations that change the state of the object to which it belongs. (4) *Collaborational* which helps define the communication between objects and how objects are controlled in the system. Finally, (5) *Degenerate* corresponds to operations which give us little information about.

Our proposal (see Table 1), which is inspired from the Dragan et al.'s one [2], includes additional stereotypes (marked with an asterisk) not initially considered in the original taxonomy: the *search*, *add* and *remove* stereotypes, which cover operations for the management of collection attributes (such as search, addition or removal, respectively); the *process* stereotype, for operations returning information based on the whole object's internal structure; and *modify*, for operations that modify a specific attribute without setting an input value directly. We have not considered the categories *collaborational* and *degenerate* since they represent behaviours already modelled by SqDs (such as the communication between objects, given by *collaborational*), or reflect aspects that cannot be tackled without checking the source code (e.g., *degenerate* category). Below, we explain the behaviour represented by each stereotype.

Table 1: Extension of the taxonomy given in [2] showing the categories of UML Class' operations considered in our proposal. Stereotypes with an asterisk denote those included by our proposal.

Category	**Stereotype name**	**Description**
Creational	create	The operation creates an object.
	destroy	The operation destroys an object.
Structural *Accessor*	get	The operation returns values of concrete attributes of an object.
	search*	The operation returns elements belonging to a concrete collection attribute of an object.
	process*	The operation returns values that are computed based the object's status as a whole.
	predicate	The operation returns boolean values that are computed based on concrete attributes of an object.
	property	The operation returns values (of any type) that are computed based on concrete attributes of an object.
	void-accessor	The operation, by means of a parameter, returns values (of any type) that are computed based on concrete attributes of an object.
Structural *Mutator*	command	The operation changes the status of an object as a whole (the modified attributes are unknown or irrelevant). It does not return information.
	non-void-command	The operation changes the status of an object as a whole (the modified attributes are unknown or irrelevant). It does return information.
	set	The operation directly sets the information passed to the operation as values of concrete attributes of an object.
	modify*	The operation modifies concrete attributes of an object.
	remove*	The operation removes an element from a concrete collection attribute of an object.
	add*	The operation adds an element on a concrete collection attribute of an object.

References

[1] P. Clarke, B. Malloy, and P. Gibson. Using a taxonomy tool to identify changes in OO software. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering, (CSMR'03)*, pages 213–222, 2003.

[2] N. Dragan, M. L. Collard, and J. I. Maletic. Automatic identification of class stereotypes. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[3] OMG. Unified Modeling Language (UML). Version 2.5, 2015. Document formal/15-03-01, March, 2015.

Integrating Provenance Capture and UML with UML2PROV: Principles and Experience

SEQUENCE OF INTERACTIONS WITH GELJ

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

[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@kcl.ac.uk

In this document, we show the sequence of interactions (see Figure 1) among the 13 substeps of the experiment wizard of GelJ, which has been selected to conduct the evaluation of UML2PROV as unbiased as possible. Each rounded rectangle in Figure 1 corresponds to a task performed by the user. Each task is named using the label provided by the GelJ interface (e.g., *corp* for cropping the image). Empty rounded rectangles mean that no tasks are performed in that step. Additionally, rounded rectangles with associated text may include a label at the right top, denoting the number of times that such a task has been performed. Finally, we use green arrows to specify that the user has proceeded to the *next* step, or red arrows to show that the user goes *back* to the previous step.

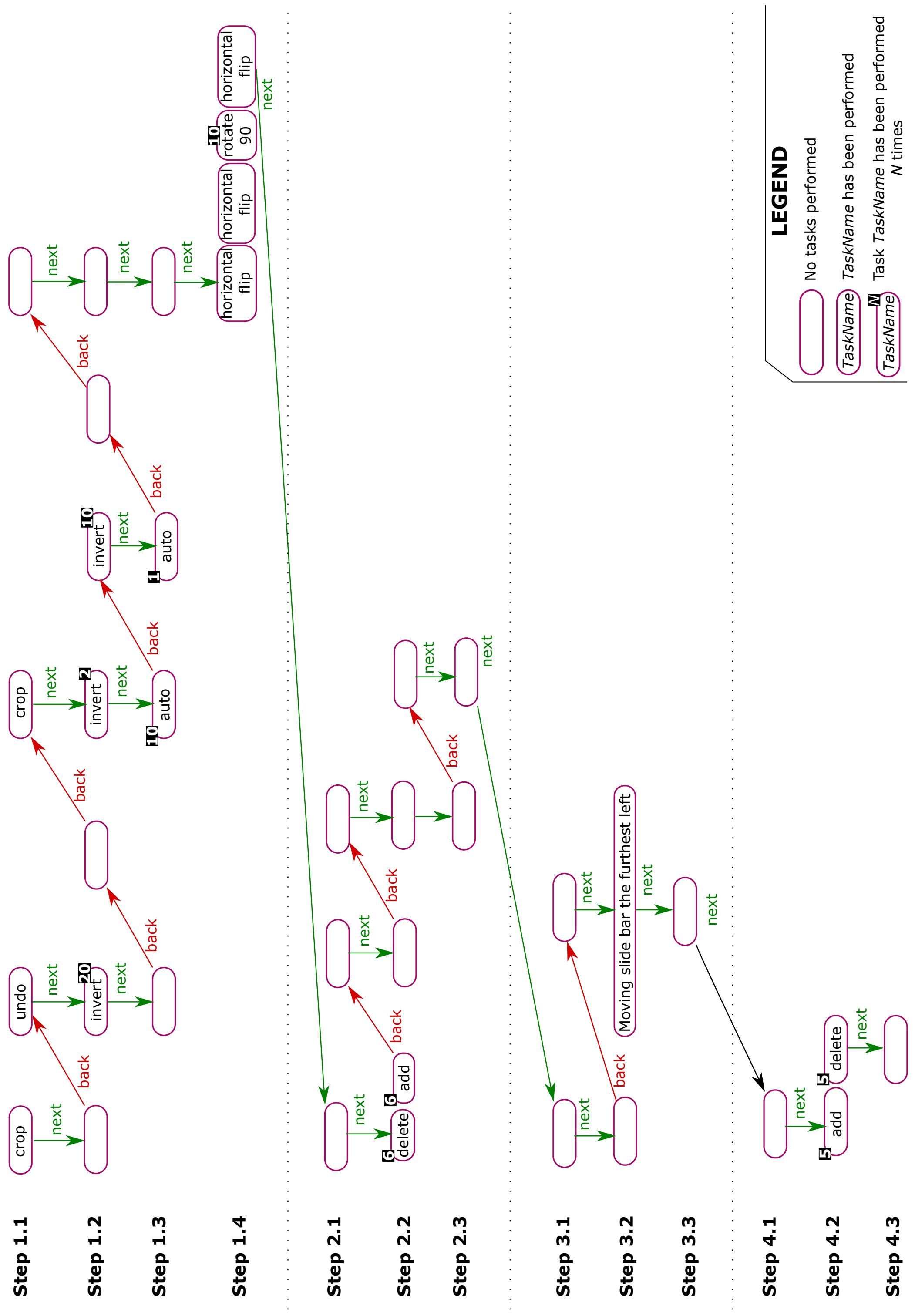


Figure 1: Sequence of interactions used to perform the evaluation.

Integrating Provenance Capture and UML with UML2PROV: Principles and Experience

CASE STUDY: UNIVERSITY EXAMPLE

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

[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@kcl.ac.uk

The results presented in this appendix are part of the evaluation we have made to show the feasibility of our proposal UML2PROV presented in the paper [1]. In that paper, we show the evaluation we have made of our proposal by applying it both to a legacy application named GelJ [2] built without UML (*retroactive* scenario) and to an academic application built from a UML design (*proactive* scenario). While in [2] we mainly focus on the GelJ case study because it is more complex, here we describe in detail the application of UML2PROV to the *proactive* scenario, the University example, using the same evaluation aspects we have taken into account in the paper [1]. We would like to note that, in contrast to the GelJ case study, in the academic University example we play the role of software designers, developers, provenance consumers and potential users.

The University case study correspond to an example, slightly modified from [3], related to the enrolment and attendance of students to seminars that are held during a University course. This application mainly allows users to perform three actions in the context of a university. The first allows administrative staff to check if a student can apply for a given seminar. The second offers the possibility of enrolling a student in a seminar. Finally, the third is related to the overall process which encompasses the evaluation of students' performance in such seminars through exams, considering from the time an exam is prepared until the student is informed of her/his mark. It is worth noting that the University application's database only contains information about specific characteristics of the elements conforming the system (e.g., students, seminars, exams, and so on). It does not keep information about the requests for checking if it is possible to apply for a seminar, or for example, the specific process of preparing and taking an exam, and informing about the mark.

In order to give an unbiased evaluation of our approach, based on a representative use of the University system, we have defined an example execution considering several scenarios (see Algorithm 1). Concretely, this execution is divided in two phases: in the first phase (lines from 2 to 9), 25 students ask for enrolling into a seminar (line 4), after the affirmative response, they are enrolled into the seminar (line 6), and finally, the student' performance evaluation begins by proceeding with an exam (line 7). In the second phase (lines 10-13), another 25 students (already enrolled) proceed with an exam (line 12). Consequently, the defined benchmark involves the execution of the three cited functionalities of the application. The performance of this example execution incurs in the execution of about 1700 operations in the tool's source code.

```
1  seminar ← findSeminar(idSeminar);
2  for $i$ ← 0 to 24 do
3      student ← findStudent($i$);
4      allowEnrol ← askStaffForEnrolling (student, seminar);
5      if allowEnrol then
6          enrolStudent (student, seminar);
7          proceedWithExam (student, seminar);
8      end
9  end
10 for $i$ ← 25 to 49 do
11     student ← findStudent($i$);
12     proceedWithExam (student, seminar);
13 end
```

Algorithm 1: Example execution algorithm in pseudocode

The evaluation was run in the same personal computer and under the same conditions as in the GelJ case study.

1 Particularities of the application modes in this case study

In Table 1 we show the tasks comprising each application mode for this case study. This information can be used to compare the characteristics of each mode.

Table 1: Overview of tasks in UML2PROV *Application Modes*.

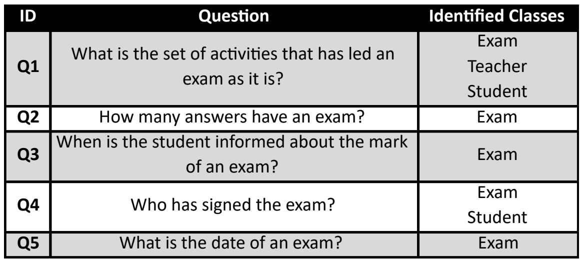
	Case study - University		
Task	**App. Mode 1**	**App. Mode 2**	**App. Mode 3**
T1. Identify the complete CD	–	–	–
T2. Identify provenance requirements	🧍	–	–
T3. Identify classes/operations involved in provenance requirements	🧍	–	–
T4. Discard not identified classes/operations	🧍	–	–
T5. Add stereotypes to selected operations	🧍	–	–
T6. Identify SqDs	🧍	–	–
T7. Design SMDs of selected classes	–	–	–

⏩ Permorfed automatically | 🧍 Requires manual effort | 🧍 / ⏩ Requires semi-manual effort | – Non executed task

Application Mode 1

Regarding task T2, since in this case study there are no final users interested in the generated provenance, it is not possible to obtain the provenance requirements from them. For this reason, as described previously, we have simulated that we are the final users that raised the provenance questions that serve as requirements. To conduct an unbiased evaluation, we have not defined the questions from scratch, but we have drawn inspiration from the questions appearing in the First Provenance Challenge [4], adapting them to obtain questions regarding the performance of exams. The resulting questions, depicted in Table 2, represent the provenance requirements (called *provenance use case questions* in PrIMe).

Table 2: Questions identified from Q1 to Q5 about the University case study, together with University classes involved in answering those questions.



ID	Question	Identified Classes
Q1	What is the set of activities that has led an exam as it is?	Exam Teacher Student
Q2	How many answers have an exam?	Exam
Q3	When is the student informed about the mark of an exam?	Exam
Q4	Who has signed the exam?	Exam Student
Q5	What is the date of an exam?	Exam

In this case study, all the UML CD, SqD and SMD diagrams are available at the beginning of the evaluation. Thus, all the performed tasks are tailoring tasks. Neither to reverse engineer the CD (T1) nor to design the SMDs (T7) are required. Similarly, it is not necessary to reverse engineer SqD. We just have to select the SqDs related to the provenance requirements (T6).

Inspired by the second phase of PrIMe, from the UML design we have identified those classes and operations (called *actors* in PrIMe) involved in answering the identified questions (T3), following the same procedure described for GelJ. The result was the identification of 3 classes and 11 operations out of 11 classes and 37 operations that compose University application (i.e., ∼27% of the classes and ∼29% of the operations of the University application were used. The rest of classes/operations were discarded (T4). These classes are shown in column "Identified Classes" of Table 2. In addition, to obtain more meaningful provenance, we assigned stereotypes to the UML operations in class diagram (T5). As for SqDs, we selected those related to the provenance requirements. This task resulted in a set of 25 messages. Regarding SMDs, we used the UML SMDs for those classes whose states are related to the provenance requirements. This led to 2 SMDs with 7 states, and 9 transitions.

As in the GelJ case study, the greatest effort goes into identifying the provenance requirements (T2) and selecting the classes and operations involved (T3 and T4).

Application Modes 2 and 3

In these modes no tasks are performed, so no effort has been made. More specifically, we have taken the original SqDs and CDs without tailoring them (CDs, in both modes and SqDs, in Mode 2). Concretely, the UML design encompasses a CD with 11 classes and 37 operations, and a set of SqDs with 25 messages in total.

1.1 Analysis

Next, we analyse those evaluation aspects explicitly related to the University case study.

Table 3: Variables evaluated for the considered UML2PROV *Application Modes* in the University case study.

Application Mode (UML diagrams)	No. templates	Total size of templates	No. Variables	No. Set of bindings	Sets of bindings size (MB)	Expanded templates size (MB)	No. executions instrumented operations	% instrumented executed operations	Execution time (ms)	Time overhead
Mode 1 (SqD, SMD, CD)	25	20KB	115	687	1,4	2,3	687	41,79%	656,19	25,92%
Mode 2 (SqD, CD)	63	47KB	269	1.644	2,0	3,3	1.644	100%	802,29	53,96%
Mode 3 (CD)	40	26KB	137	1.644	1,9	2,0	1.644	100%	747,31	43,41%

1.1.1 Aspect 1: Generation of the provenance design

In line with the results obtained for GelJ, the time-cost for generating the templates, a few milliseconds per template, is considered negligible compared to the time-cost of performing this task manually (this information is not depicted in Table 3 because we consider it trivial). As for the implications of the followed mode, the results show that the closer the UML design fits the application provenance requirements, the less templates are generated and consequently, the smaller their total size and the faster their generation. Table 3 shows that Mode 1 (with a tailored UML design), results in the lowest number of templates (25). However, Modes 2 and 3 present a higher number of templates (63 and 40, respectively). These figures confirm that a greater initial effort to more accurately tailor the UML according to a set of provenance requirements, results in fewer templates.

1.1.2 Aspect 2: Instrumentation of the application

The column "No. variables" of Table 3 depicts the number of instructions included in the BGM for bindings generation.

Mode 1, with least UML elements, leads to the BGM with fewest instructions (115) against Modes 2 and 3 that generate BGMs with more instructions within (269 and 137, respectively). Given these results, we can see that although Mode 1 has fewer number of instructions, the difference with Mode 3 is relatively small (115 vs 137 instructions). This small difference is because of three main reasons. First, due to the small difference between the considered operations in Mode 1 (11 operations, see Table 1), and Mode 3 (37 operations). Second, because Mode 1 generates templates from SqDs and SMDs, unlikely Mode 3. Third, since Mode 1 considers CDs with stereotypes, which incurs in templates with more variables.

Nevertheless, these results are in the line of those obtained from GelJ [1]: the effort devoted to more precisely tailor the UML design according to the provenance requirements results in a simplification of the BGM, which has significant implications in the performance.

1.1.3 Aspect 4: Storage and Run-time overhead

The overhead attributable to provenance capture is associated with the number of executions of instrumented operations (see Table 3, where column "No. set of bindings" matches this number). In this case study, Mode 1 generates the least number of set of bindings (687), and consequently, led to the least run-time overhead (25.92%) and storage needs (1.4MB). Conversely, Modes 2 and 3 generated more set of bindings (1,644 both of them), and yielded the more time overhead (53.96% and 43.41%, respectively) and storage needs (2MB and 1.9MB). Mode 2 will always require more storage since it takes into account the whole Class diagram (unlike Mode 1) and additionally, it does not discard the Sequence diagrams (in contrast to Mode 3).

Table 4: For each mode, it is indicated if questions Q1-Q5 of Table 2 can be answered completely (C), sufficiently (S), partially (P) or cannot be answered (N). If a question can be answered, the number of elements of the provenance involved in its answer appears in brackets.

	Q1	Q2	Q3	Q4	Q5
Mode 1	S(5)	S(6)	S(3)	S(3)	S(2)
Mode 2	C(12)	S(6)	N	S(3)	S(2)
Mode 3	C(12)	S(6)	N	S(3)	S(2)

Finally, the difference between column "Set of bindings size" and "Expanded templates size" in Table 3 confirms how the use of the PROV-Template approach for generating provenance reduces the storage requirements for the set of bindings compared to the expanded templates.

1.1.4 Aspect 5: Quality of provenance

In this section we will focus on the quality of the provenance generated from the University example. To analyse this aspect, we study if the collected provenance answers *completely* (C), *sufficiently* (S), *partially* (P) or it cannot answer (N) the questions in Table 2.

Completely When the user indicated that the answer was more detailed that what she/he expected.

Sufficiently When the user indicated that the level of detail was enough.

Partially When the answer did not satisfy the user.

No When it was not possible to answer the question.

Table 4 summarizes our conclusions, showing the number of elements (`prov:Entity`, `prov:Activity`, and `prov:Agent`) involved in such an answer, when it can be responded (i.e., when the response is not classified as N). Based on these results, we identified three kind of implications the mode used to tailor the UML design may have on the answers.

No effect. The followed mode had *no effect* on the ability to answer to questions Q2 and Q5. More specifically, the answer to questions Q2 and Q5 relies upon the values of attributes belonging to the class `Exam`. Since the three modes take into account a CD with all the attributes of class `Exam`, the provenance obtained from can answer these questions.

More detailed information. The answer to Q1 relies upon the operations identified in classes `Exam`, `Teacher`, and `Student`. Mode 1, which only identifies a set of operations, answers Q1 with a sufficient number of elements (5). However, Modes 2 and 3, encompassing the whole operations, answers Q1 with more elements than necessary (12 both of them).

Crucial. The UML diagrams supported by UML2PROV model only certain aspects of an application's behaviour. Consequently, the generated provenance will contain information about these aspects. The answer to question Q3 relies upon information provided by SMDs; thus, Mode 1, which is the only one considering SMDs, is the unique mode capable for answering Q3.

References

[1] C. Sáenz-Adán, B. Pérez, F. J. García-Izquierdo, and L. Moreau, “Integrating Provenance Capture and UML with UML2PROV: Principles and Experience,” submitted for publication in IEEE Transactions on Software Engineering.

[2] J. Heras, C. Domínguez, E. Mata, V. Pascual, C. Lozano, C. Torres, and M. Zarazaga, “GelJ – a tool for analyzing DNA fingerprint gel images,” *BMC Bioinformatics*, vol. 16, Aug 2015.

[3] M. Seidl, M. Scholz, C. Huemer, and G. Kappel, *UML@Classroom: An Introduction to Object-Oriented Modeling*. Springer Publishing Company, Incorporated, 2015.

[4] L. Moreau et al., “Special issue: The first provenance challenge,” *Concurr. Comput. : Pract. Exper.*, vol. 20, pp. 409–418, Apr. 2008.

Integrating Provenance Capture and UML with UML2PROV: Principles and Experience

OCL CONSTRAINTS

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

[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@kcl.ac.uk

These constraints (from OCL1 to OCL5) mainly impose interconnections among (OCL1 and OCL2) senders/receivers in SqD and objects modelled by SMD, (OCL3 and OCL4) incoming/outgoing messages in SqD with events/actions in SMD, and (OCL5) incoming messages in SqD with methods in objects modelled by SMD.

(OCL1-2) Each sender of a message in an interaction of a SqD must be an object modelled by a SMD. The same constraint is defined for a receiver, changing sender by receiver.

```
context: Interaction
inv: self.message.sender.base.behavior âĂŘ> notEmpty ()
```

(OCL3) Incoming messages to an object within a SqD are events in a SMD.

```
context: Message
inv: self.receiver.base.behavior.region.transition.trigger->exists(e|e.name=self.name)
```

(OCL4) Outgoing messages of an object within a SqD are actions in a SMD.

```
context: Message
inv: self.sender.base.behavior.region.transition.effect->exists(e|e.name=self.name)
```

(OCL5) Incoming messages of an object (receiver) within SqD must be object's methods.

```
context: Message
inv: let rec:ClassifiedRole = self.receiver in
          let ops:Operation = rec.base.ownedOperation in
          ops -> exists(oper| oper.name = self.name)
```