Towards a framework for making applications provenance-aware
–Supplementary Material–

Pepito el de los palotes

No Institute Given

Introduction

This document contains supplementary material for the paper entitled “Towards a framework for making applications provenance-aware”, which has been organized as follows:.

- Section 1 “Transformation patterns”
- Section 2 “Implementation details”
- Section 3 “UML design of GelJ”
- Section 4 “Qualitative evaluation (extended)”
- Section 5 “Overall process” ?

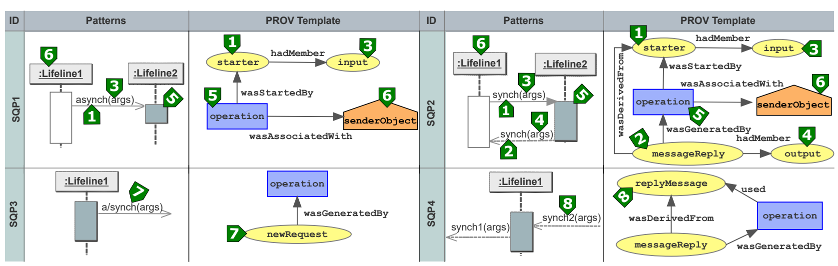
1 Transformation patterns

In this section we describe the set of patterns which lay the foundation for our strategy to transform the UML diagrams (UML SqDs, SMDs and CDs) into the provenance templates. Concretely, these patterns associate commonly used UML structures, included in UML SqDs, SMDs and CDs , with PROV templates. More specifically, we have identified 19 UML patterns in total, being 4 from SqD (see Subsection 1.1), 6 from SMD (see Subsection 1.2), and 10 from CD (see Subsection 1.3).

1.1 From Sequence Diagrams to templates

We have identified the *ExecutionSpecifications*, started by a *message* representing an *operation*, as cornerstone elements of the transformations patterns (see Table 1). The patterns are focused on the (1) *ExecutionSpecifications* started by a *message*, and (2) *ExecutionSpecifications* sending/receiving *messages* during their executions.

Table 1. Set of patterns identified from SqD together with their PROV templates. The set of patterns encompasses interactions representing an *ExecutionSpecification* started by an asynchronous message (**SQP1**), an *ExecutionSpecification* started by a synchronous message and its reply message (**SQP2**), an *ExecutionSpecification* sending an (a)synchronous message during its execution (**SQP3**), and *ExecutionSpecification* –started by an asynchronous message– receiving the reply of a synchronous message during its execution (**SQP4**).



(1) Regarding the start of an *ExecutionSpecification* by a *message*, we have defined the patterns **SQP1** and **SQP2**. While **SQP1** translates the *ExecutionSpecifications* started by an *asynchronous message*, **SQP2** translates those started by a *synchronous message* and finished by its *reply message*. Among both patterns we have identified five key UML elements which are translated as follows:

– The ***synchronous/asynchronous message*** which starts an *ExecutionSpecification*. The *message* is represented by a `prov:Entity` identified by `var:starter` (num. 1). In case the *message* is *synchronous* (SQP2), there is also a *reply message* representing its response. This ***reply message*** is translated into another `prov:Entity` identified by `var:messageReply` (num. 2). In order to represent this *reply message* (`var:messageReply`) has been generated as a reply of the synchronous message (`var:starter`), we use the relationship `prov:wasDerivedFrom` to link `var:messageReply` with `var:starter`.

- The ***in/inout arguments*** within the request *message* (`var:starter`). The *in arguments* are mapped to a `prov:Entity` identified by `var:input` (num. 3). Additionally, in order to reflect that *in arguments* (`var:input`) are a part of the *message* (`var:starter`), the relationship `prov:hadMember` is used to link both `prov:Entities`.
- The ***out/inout/return arguments*** within the reply *message* (`var:messageReply`). These *arguments* are translated into a `prov:Entity` identified by `var:output` (num. 4). In the same way as showed previously, with the aim of showing that *out/inout/return arguments* (`var:output`) belong to the *reply message* (`var:messageReply`), the relationship `prov:hadMember` is used to link `var:messageReply` with `var:output`.
- The ***ExecutionSpecification*** started by a *message* (`var:starter`). This *ExecutionSpecification* is modelled by a `prov:Activity` identified by `var:operation` (num. 5). In addition, to represent the fact that the *message* starts the *ExecutionSpecification*, we use the relationship `prov:wasStartedBy` between `var:operation` and `var:starter`. As previously said, in case the *message* is *synchronous* (**SQP2**), there is also a *reply message* identified by `var:messageReply` (num. 2). With the aim of showing that such a *reply message* has been created during the *ExecutionSpecification* (`var:operation`), the relationship `prov:wasGeneratedBy` between `var:messageReply` and `var:operation` is used.
- The ***object lifeline*** which sends the *synchronous/asynchronous message*. This UML element is mapped to a `prov:Agent` identified by `var:senderObject` (num. 6). In order to represent the responsibility of such an object (`var:senderObject`) for starting the *ExecutionSpecification* (`var:operation`), the relationship `prov:wasAssociatedWith` is used between them.

(2) Regarding the *synchronous/asynchronous messages* sent and received during the execution of an *ExecutionSpecification* (see **SQP3** and **SQP4** in Figure 1), we have identified the following mappings:

- The ***synchronous/asynchronous message*** **sent** during the execution of an *ExecutionSpecification* (identified by `var:operation`). As we can see on **SQP3**, this *message* is represented by a `prov:Entity` identified by `var:newRequest` (num. 7). Such a *message* (`var:newRequest`) is, in turn, related to the *ExecutionSpecification* (`var:operation`) through the relationship `prov:wasGeneratedBy`.
- The ***reply message*** **received** during the execution of an *ExecutionSpecification*. As it is illustrated in **SQP4**, such a *reply message* is translated into a `prov:Entity` identified by `var:replyMessage` (num. 8). Subsequently, the relationship `prov:used` is created between `var:operation` (which represents the *ExecutionSpecification*) and `var:replyMessage` to represent the fact that

the *reply message* is used by the *ExecutionSpecification*. If the *ExecutionSpecification* has been started by a *synchronous message* and it thus sends a *reply message* (identified by `var:messageReply`), we also identify the fact that the received *reply message* (`var:replyMessage`) is involved in the creation of the sent *reply message* (`var:messageReply`) by using the relationship `prov:wasDerivedFrom` between `var:messageReply` and `var:replyMessage`.

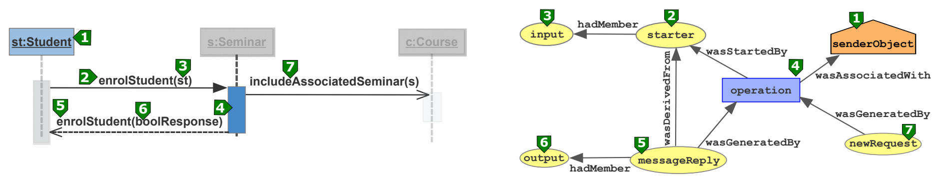


Fig. 1. On the left, a SqD showing the interaction between `Student`, `Seminar` and `Course` for enrolling a `Student` in a `Seminar`. On the right, the template generated from the UML *ExecutionSpecification* in dark.

Finally, we show in Figure 1 the example presented in the paper. This example depicts on left hand side a UML diagram with a dark *executionSpecification* (source of the transformation) which is started by a *synchronous message* (*enrolStudent*), and during its execution it is sent an *asynchronous message* (*includeAssociatedSeminar*). On the right hand side, it is depicted the resulted PROV template automatically generated from the *executionSpecification* in dark. This transformation has been done by applying the patterns **SQP2** and **SQP3**. While **SQP2** is applied since the *executionSpecification* is started by a *synchronous message* (*enrolStudent*), **SQP3** is applied because a *message* is sent during the execution of the *executionSpecification*. Concretely, on the bases of **SQP2**, the *synchronous message enrolStudent* is translated into the `prov:Entity` identified by `var:starter` (num. 2), and its *in arguments* into `var:input` (num. 3). Likewise, the *reply message enrolStudent* is conceptualized by the `prov:Entity` called `var:messageReply` (num. 5), and its *return arguments* by `var:output` (num. 6). The object which sends the message *enrolStudent* is represented by means of the `prov:Activity` identified by `var:senderObject` (num. 1). Finally, the *ExecutionSpecification* started by the message *enrolStudent* is modelled by the `prov:Activity` called `var:operation` (num. 4). On the other hand, based on **SQP3** the *synchronous message* sent from the *ExecutionSpecification* in dark, is translated into the `prov:Entity` with the identifier `var:newRequest` (num. 7).

1.2 From State Machine Diagrams to templates

We have identified the *operation*, which starts the *event* which triggers a *transition*, as cornerstone element of these transformation patterns. Based on the source and target elements of the *transition* and its location, we have identified 6 patterns identified by **StP1-6** (see Figure 3). In order to convey the intuition of those patterns in Figure 3, next to their identifiers (**StP1-6**), it is shown the UML elements addressed by each pattern ("pattern" column). For instance, **StP1** and **StP4** model a *transition* from a source *initial pseudostate* to a target *state* through a transition located within and without another *state*, respectively. **StP2** and **StP5** translate a *transition* from a source *state* to a target *final state* when the *transition* is located within and without another *state*, respectively. Finally, **StP3** and **StP6** are devoted to translate a *transition* from a source *state* to a target *state* through a transition located within and without another *state*, respectively. As we can see, the pairs **StP1-StP4**, **StP2-StP5** and **StP3-StP6**, model the same UML behaviour, but they are differentiated by the location of the *transition*. Although this fact makes their transformations have a great resemblance, they have differences that we will explain later.

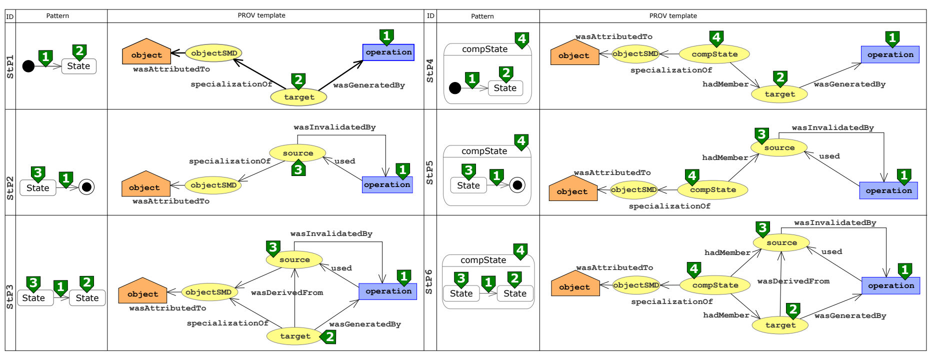


Fig. 2. Set of patterns identified from SMD together with their PROV templates. The patterns associated to *external transitions* are located on the left hand side (**StP1-3**), whereas the patterns related to *internal transitions* are on the right hand side (**StP4-6**).

As for the transformation patterns showed in Figure 3, we have identified the following common translations:

– The `operation`, which starts the `event` which triggers a `transition`, is translated into a `prov:Activity` identified by `var:opeartion` (num. 1).

– The *object* whose behaviour is modelled by the SMD is translated into a `prov:Agent` identified by `var:object`.

– The *object*'s *state machine* is mapped to the `prov:Entity` called `var:objectSMD`. In order to represent that the *state machine* represents its *object*, the relationship `prov:wasAttributedTo` links `var:object` with `var:lifeline`.

Furthermore, depending on the nature of the patterns –i.e., the source, and target elements in the transition and the location of the transition, the following elements are included in the PROV templates:

– Patterns with a source state (**StP2**, **StP3**, **StP5**, and **StP6**) in the transition translate such a source state as a `prov:Entity` identified as `var:source` (num. 3). To represent the fact that, after triggering the transition given by the *operation*'s *event*, the *object* is no longer in the source *state*, we have used the relationship `prov:wasInvalidatedBy` to link the source *state* `var:source` with the *event* `var:event`.

– Patterns with a target state (**StP1**, **StP3**, **StP4**, and **StP5**) in the transition, translate such a target state as a `prov:Entity` identified as `var:target` (num. 2). Since this target state has been reached after the execution of the event's operation, `var:opeartion` is linked with `var:target` by means of the relationship `prov:wasGeneratedBy`.

– Patterns with a source and target state in the transition (**StP3** and **StP6**), translate both states as `var:source` and `var:target` with the aforementioned relationships. Furthermore, to show the change of state from the source to the target, `var:target` and `var:source` are related through the relationship `prov:wasDerivedFrom`.

Finally, we note that in *transitions* outside a composite state (**StP1-3**), the `prov:Entities` representing the *source* and the *target states* (i.e., `var:source/var:target`) are directly linked with `var:object` through the relationship `prov:specializationOf`. Contrary, *transitions* inside a *composite state* (**StP4-6**), include a `prov:Entity` identified by `var:compState` representing the *composite state* at hand. Such a `prov:Entity` is related to `var:source` and `var:target` by means of `prov:hadMember` to represent that those *states* are located inside the *composite state*, and `var:compState` is related to `var:objectSMD` through `prov:specializationOf` to denote that the composite state belongs to the object's state machine.

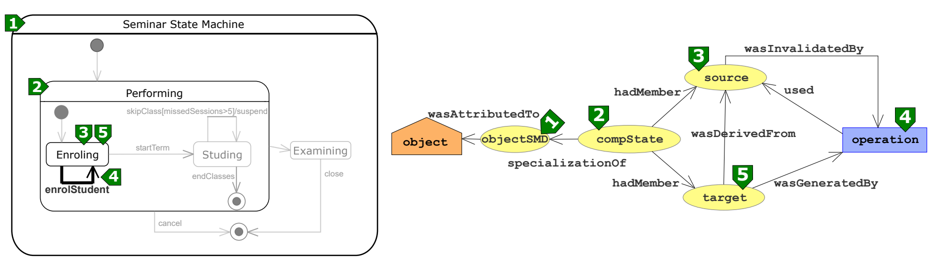


Fig. 3. On the left hand side, a SMD showing the behaviour of `Seminar`. On the right hand side, the PROV template automatically generated from the UML transition in dark

Finally, we show Figure 3 with the example of transformation presented in the paper. On the left hand side, it is depicted a SMD of a *Seminar*, which highlights in bold a *transition* triggered by the event *enrolStudent*. This *transition* is located within the composite *state* `Performing`, and goes from the source and target *state* called *Enroling*. On the right hand side, it is the PROV template generated after applying the pattern **StP6**, since the *transition* is located within a composite *state*, and goes from one *state* to another *state*. Thus, following the pattern **StP6**, the operation which starts the *event enrolStudent* is represented by a `prov:Activity` identified by `var:operation` (num. 4). The source and target *states* (`Enroling`) are conceptualized by the `prov:Entities` called `var:source` (num. 3) and `var:target` (num. 5), respectively. The composite *state* called *Performing* is translated into the `prov:Entity` identified by `var:compState`. Finally, both the *object* (*Seminar*) and its *state machine* are represented by the `prov:Agent` `var:object` and the `prov:Entity` `var:objectSMD` (num. 1), respectively.

1.3 From class diagrams to templates

In order to generate data provenance both covering the internal structure of *classes*' instances and explaining the process that has led the class to be as it is, we identify the *operations*, customized by the stereotypes showed in Table 2, as cornerstone elements in the transformations.

Table 2. Taxonomy showing the categories used in our proposal.

Category	Stereotype name	Description
Structural *Accessor*	get	Returns a data member.
	search*	Returns an element from a data member collection.
	predicate	Returns Boolean value which is not a data member.
	property	Returns information about data members.
	void-accessor	Returns information through a parameter.
	process*	Returns information based on the object.
Structural *Mutator*	set	Sets a data member.
	modify	Modifies a data member.
	add*	Adds an element within a data member collection.
	remove*	Removes an element within a data member collection.
	command	Performs a complex change.
	non-void-command	
Creational	create/destroy	Creates/Destroys objects.

The set of mappings comprises 10 transformation patterns identified as **CDP1-10**, referring **CDP** to *C*lass *D*iagram *P*attern. Table 3 shows patterns **CDP1-2**, translating operations stereotyped with *creational* stereotypes (i.e. *create* and *destroy*). Table 4 depicts patterns **CDP3-5**, representing the transformations of operations stereotyped with *Structural Accessor stereotypes* (i.e. *get, predicate, property*, *void-accessor*, and *process*). Finally, Table 5 presents patterns **CDP6-10**, addressing the transformation of operations with *Structural Mutator stereotypes* (i.e. *command, set, modify, add* and *remove*). All these tables are organized as follows: the first column shows the name of the category (e.g. *creational*), the second column represents common PROV structures shared by all the patterns in the category (base elements), and the third column depicts the specific elements for each one of the stereotypes in the category. Additionally, we note that in order to convey the intuition of the patterns throughout the explanation, (1) the PROV elements within the tables are associated to a UML element from Figure 4 by means of an identifier; (2) the PROV elements which represent optional components in UML are underlined (e.g., input arguments); and (3) in the third column, the specific PROV elements generated by such a pattern are highlighted. Next, we explain each pattern organized by the category of the operation it translates.

Fig. 4. Taxonomy showing the categories used in our proposal.

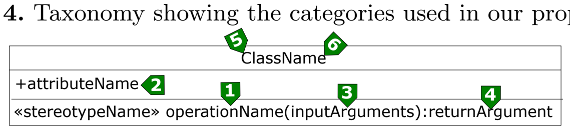
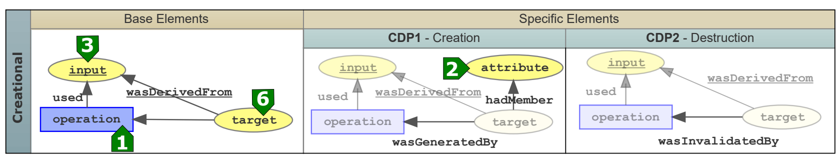


Table 3. Set of patterns identified for mapping operations with a stereotype belonging to the ***Creation* category**. *Base Elements* column depicts the common elements shared by both **CDP1** and **CDP2** patterns, whereas the columns located within the *Specific Elements* column show the PROV elements (highlighted) which conceptualize the nuances of the concrete stereotyped operations' behaviour.

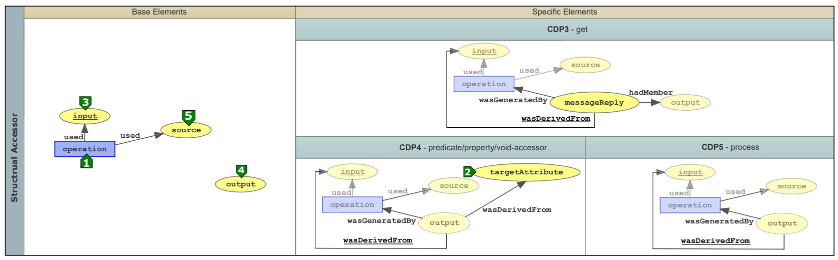


*The **Creational** category.* This category includes the *create* and *destroy stereotypes*, whose translation is addressed respectively by **CDP1** and **CDP2**. We note that, while the *create stereotype* categorizes those operations which create a new object, the *destroy stereotype* denotes operations which state an object no longer available. The translation of operations with such stereotypes share a common set of PROV elements shown in the first column of Table 3 (named *Base Elements*). There, we can see how (i) the *creation/destruction* operation is represented by means of a `prov:Activity` identified by `var:operation` (num. 1), (ii) the input parameters (if any) are depicted as a `prov:Entity` called `var:input` (num. 3), which is associated with `var:operation` through the relationship `prov:used`, and (iii) the *created/destroyed* object is modelled by the `prov:Entity` identified by `var:target` (num. 6), which is associated to `var:input` (if any) through the relationship `prov:wasDerivedFrom`.

CDP1 and **CDP2** include specific elements explicitly related to the corresponding operation's semantic (see second column in Table 3 named *Specific Elements*). For instance, **CDP1** establishes the `prov:wasGeneratedBy` relationship between `var:operation` and `var:target` in order to show that the object was created by the operation, whereas **CDP2** uses the `prov:wasInvalidatedBy` relationship to represent the destruction of the object. Furthermore, **CDP1** also includes the `prov:Entity` identified by `var:attribute` for representing the object's attributes (num. 2). To show that those attributes belong to the object, the relationship `prov:hadMember` links `var:target` with `var:attribute`.

*The **Structural Accessor** category.* It refers to operations that return information regarding a data member of the object to which it belongs, without changing the internal structure of the object. Thus, they share common transformations (see *Base Elements* column in Table 4). The operation marked with any *structural accessor stereotype* is represented by means of a `prov:Activity`

Table 4. Set of patterns identified for mapping operations with a stereotype belonging to the ***Structural Accessor* category**. *Base Elements* column depicts the PROV elements shared by both the **CDP3-5** patterns, whereas the columns located within the *Specific Elements* column show the PROV elements (highlighted) which conceptualize the nuances of the concrete stereotyped operations' behaviour.



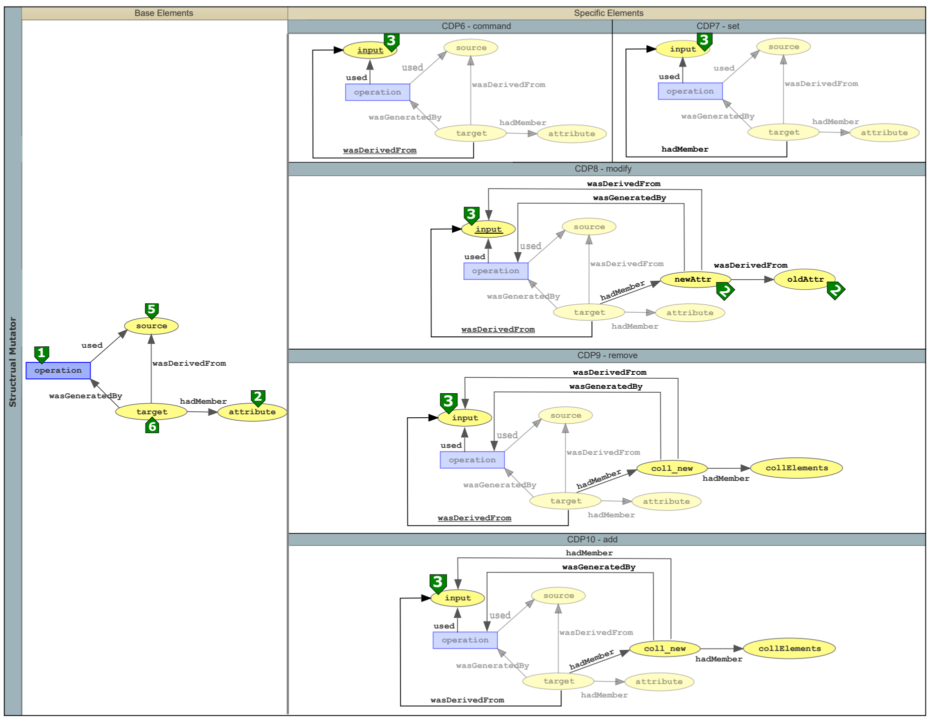
called `var:operation` (num. 1). Such operation returns information that we represent by a `prov:Entity` identified by `var:output` (num. 4). Additionally, the operation uses both input arguments (if any) – modelled as a `prov:Entity` identified by `var:input` (num. 3)– and the object to which it belongs –translated into a `prov:Entity` called `var:source` (num. 5). In order to reflect these facts, `var:operation` is related to both `var:input` and `var:source`, through the relationship `prov:used`.

Depending on the source of the information, we have defined two types of transformation patterns: **CDP3**, and **CDP4** together with **CDP5** (see Table 4). On the one hand, **CDP3** encompasses operations with the stereotypes `get` and `search`; that is, operations which directly return either a data member or an element from a collection data member. Contrary, **CDP4** and **CDP5** refer to operations which provide information derived from the object's internal structure.

- **CDP3** represents operations which return information not generated by them (only retrieved). To represent this fact, we use the `var:messageReply` `prov:Entity`. This `prov:Entity` is created by the operation (`var:operation`), and encapsulates the retrieved information (`var:output`). Thus, the relationship `prov:wasGeneratedBy` links `var:messageReply` and `var:operation`, `prov:hadMember` relates `var:messageReply` to `var:output`, and finally, if there exist input arguments, the relationship `prov:wasDerivedFrom` associates `var:messageReply` to `var:input`.
- **CDP4** and **CDP5** addresses operations which generate new information (not only return information). Hence, both patterns use the relationship `prov:wasGeneratedBy` to associate the returned information (`var:output`) to the operation (`var:operation`), and the relationship `prov:wasDerivedFrom` to link the returned information (`var:output`) with the input arguments (`var:input`, if there exist). Additionally, since **CDP5** addresses operations which involve a specific data member, it defines the `prov:Entity` `var:targetAttribute` (num. 2) for modelling such a data member. To show the influence of this

data member (`var:targetAttribute`) in the returned information (`var:output`), the relationship `prov:wasDerivedFrom` associates them.

Table 5. Set of patterns identified for mapping operations with a stereotype belonging to the ***Structural Mutator* category**. *Base Elements* column depicts the common elements shared by all the patterns in this category, whereas the columns located within the *Specific Elements* column show the PROV elements (highlighted) which conceptualize the nuances of the concrete stereotyped operations' behaviour.



The ***Structural Mutator*** *category.* This category represents *operations* that change the internal structure of the object to which their belong. Thus, the PROV templates resulted from the translation of the stereotyped operations included in this category, must represent the evolution from a source object's internal structure to a target object's internal structure. As it is depicted in the *Based elements* column of Table 5, the execution of this kind of operations is modelled by the `prov:Activity` called `var:operation` (num. 1). Additionally, the change from the source object's internal structure –represented by the `prov:Entity` identified by `var:source` (num. 5)– to the target object's internal structure –conceptualized by `var:target` (num. 6)– is modelled by means of three relationships: (i) `prov:used` between `var:operation` and `var:source`, (ii) prov:wasGeneratedBy from var:target to var:operation, and (iii) prov:wasDerivedFrom linking var:target with var:source. These relationships represent how the execution of the operation (i) uses the source object's internal structure at the beginning of the execution, (ii) generates a new object's internal structure, and (iii) changes the object's internal structure. Finally, in order to provide a finer detail about the target object's internal structure –represented by var:target– the attributes belonging to the target object are represented by the prov:Entity identified by var:attribute (num. 2), and associated with var:target by means of the relationship prov:hadMember.

In addition to these base elements shared by all the patterns in the the *structural mutator* category, each pattern includes specific PROV elements which represent the particular semantics given by the stereotyped operations tackled by the pattern.

- Operations with the *command* stereotype (**CDP6**) may include input arguments. Such arguments are translated into the prov:Entity var:input (num. 3) which is associated with the operation (var:operation) through the relationship prov:used, and with the target object's structure (var:target) by means of prov:wasDerivedFrom.
- Operations with the *set* stereotype (**CDP7**) involve a compulsory input parameter, which is also modelled by var:input (num. 3), and associated with var:operation using the prov:used relationship. In addition, to represent that the input parameter (var:input) is directly added as a new attribute in the object (var:target), the relationship prov:hadMember is used between var:target and var:input.
- Operations with the *modify* stereotype (**CDP8**) change a specific attribute without setting an input argument directly (**CDP8**). This new attribute is represented through a prov:Entity identified as var:newAttr (num. 2), and it is, in turn, associated with: (i) the previous version of the attribute (identified by var:oldAttr) and the input parameters (identified by var:input (num. 3), if exist) by means of the relationships prov:wasDerivedFrom; (ii) the operation (identified by var:operation) through the relationship prov:wasGeneratedBy; and (iii) the target object's internal structure (identified by var:target) using the relationship prov:hadMember.
- Operations with the *remove* (**CDP9**) and *add* (**CDP10**) stereotypes, represent the addition and removal of elements from a collection data member. Thus, the have a set of common elements in their transformations. The new collection data member resulted from the *removal/addition* is translated into a prov:Entity identified by var:coll_new, and all the elements in such a collection are modelled by means of the prov:Entity called var:collElements. In order to represent that these elements belong to the collection data member, the relationship prov:hadMember links var:coll_new with var:collElements. Finally, since the collection data member (var:coll_new) belongs to the target object's internal structure (var:target), they are associated through the relationship named prov:hadMember. Likewise, to represent that the new col-

lection data member (`var:coll_new`) has been generated by the operation (`var:operation`), the relationship `prov:wasGeneratedBy` links both elements. As for the differences between **CDP9** and **CDP10**, **CDP9** could not have input arguments, whereas **CDP10** must have them since such input arguments correspond to the new elements added to the collection data member. Thus, although both **CDP9** and **CDP10** translate the input arguments into a `prov:Entity` called `var:input` (num. 3), the relationship with `var:coll_new` is different. In **CDP9**, `var:input` is linked with `var:coll_new` through the relationship `prov:wasDerivedFrom`, whereas in **CDP10**, `var:input` is associated to `var:coll_new` using the relationship `prov:hadMember` to denote that the input arguments are new elements within the collection.

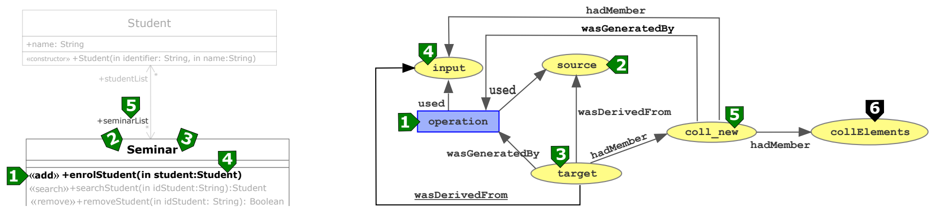


Fig. 5. On the left hand side, the CD showing the structure of `Student` and `Seminar` classes, and their relation between them. On the right hand side, the PROV template automatically generated from the UML *operation* in bold.

As a way of example, we depict in Figure 5 the example of transformation showed in the paper. On the left hand side, it is presented a class diagrams with two *classes* (*Student* and *Seminar*), and a *Seminar*'s *operation* highlighted in bold (*enrolStudent*). On the right hand side, it is shown the PROV template resulted from the transformation of the *operation* `enrolStudent`. Since such an *operation* is linked with the stereotype *add*, the transformation follows the pattern **CDP10**. More specifically, following the pattern **CDP10**, the *operation* (*enrolStudent*) is represented by a `prov:Activity` identified by `var:operation` (num. 1). The *input argument* –i.e. the new student, is conceptualized by the `prov:Entity` called `var:input` (num. 4). The data member collection with the list of students is translated into the `prov:Entity var:coll_new` (num. 5), and the remainder students who previously were added in the list are represented by the `prov:Entity` identified by `var:collElements` (num. 6). Finally, both the *object*'s internal structure before and after the execution are modelled by the `prov:Entities` called `var:source` and `var:target`, respectively.

2 Implementation details

Whilst in the paper we have given an overview, herein we provide a full detailed description of our Model Driven Development (MDD) approach for implementing UML2PROV. The section is organized by the two main processes which encompass the whole UML2PROV architecture (see Figure 6). First, the *PROV templates generation process* (Subsection 2.1). Second, the *bindings generation module generation process* (Subsection 2.2).

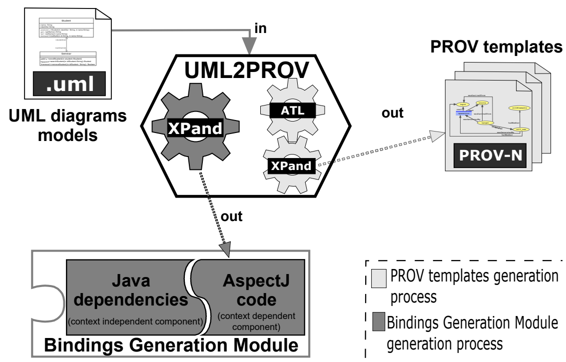


Fig. 6. MDD-based implementation proposal.

2.1 PROV templates generation process

This process takes as source the UML diagrams models and automatically generates the PROV templates files. Our proposal for template's generation follows an MDD-based tool chain, comprising two transformations which are identified as `T1` and `T2` (see Figure 7).

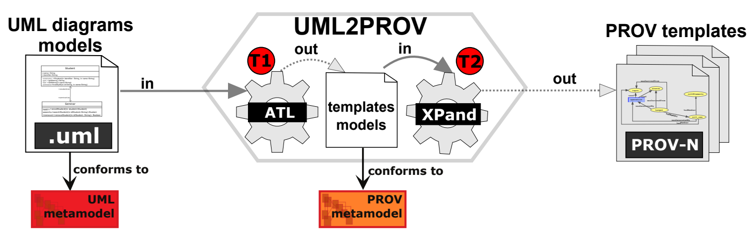


Fig. 7. Detailed MDD-based implementation of the PROV template generation process

Transformation `T1`. This transformation, which follows the transformation patterns showed in Section 1, takes as source the *UML diagram models*, conforming to the UML metamodel [1], and generates the corresponding *templates models*, conforming to the PROV metamodel [2]. To that end, the transformation patterns have been implemented in an ATL module which is made up of a set of ATL rules. These rules produces target elements (i.e. PROV elements) from a set of source elements (i.e. UML elements). In our case, such rules allows us to automatically translate each diagram model (SqD, SMD and CD) into the corresponding provenance *template models*. As a way of example, in Table 6, we show an ATL rule. This rule implements the pattern **CDP1** shown in Table 3, transforming UML operations marked with the *creation* stereotype. In order to provide an insight of the the process, next to the ATL instructions we have included the PROV template being generated (element by element). It is worth noting that the new PROV elements generated by means of such instruction(s) are highlighted in saturated colours, whereas the PROV elements which already exist are in desaturated colours.

Transformation T2. It takes as source the *templates models*, returned by *T1*, and generates the PROV templates files in PROV-N. This transformation is implemented in an XPand module which associates each PROV element with its PROV-N representation. For instance, each `prov:Entity` in the *template models* is translated according to the expression defined in its PROV-N serialization grammar, i.e. `entity(<identifier>,[<attributes>])`.

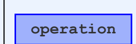
Table 6. MDD-based implementation proposal.

ATL Source Code	PROV Template

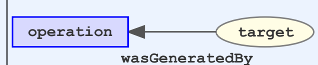
`rule OperationCreation2Document {`
`from`
`operation:UML!Operation(operation.hasStereotype('creation'))`
 	
`using {`
`existIn: Boolean=operation.ownedParameter`
`->exists(ip | ip.direction.toString() = 'in');`
`hasAttr: Boolean =`
`operation.class.ownedAttribute->size() > 1;`
`}`
 	
`to`	

`targetEn: PROV!Entity (`
`id <- 'var:target',`
`type <- 'exe:' + operation.class.name,`
`type<- 'u2p:Object'`
`),`
	

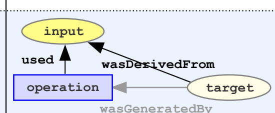
`targetID: PROV!IDRef (`
`ref <- targetEn.id`
`),`	

`operationAct: PROV!Activity (`
`id <- 'var:operation'`
`),`
	

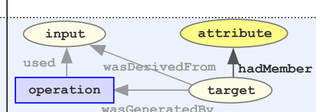
`activityID: PROV!IDRef (`
`ref <- operationAct.id`
`),`	

`wgb: PROV!Generation (`
`entity <- targetID,`
`activity <- activityID`
`),`
	

`doc: PROV!Document (`
`id <- operation.class.name + '_new',`
`activity <- operationAct,`
`entity <- targetEn`
`)`
 	
`do {`	

`if(existIn){`
`thisModule.newInputEntity(doc);`
`thisModule.genUsage('var:input', operationAct, doc);`
`thisModule.genWDF('var:input', targetEn.id, doc);`
`}`
	

`if(hasAttr){`	

`thisModule.newAttributeEntity(doc);`
`thisModule.genHM(targetEn.id,`
`'var:attribute',`
`'exe:ownedAttribute',`
`doc);`
	

`}`
 	
`thisModule.documentRoot.document<-doc;`
`}`
`}`	

2.2 Bindings Generation Module generation process

Throughout this process, it is generated a bindings generation module from the UML diagrams models. This module is composed by a *context-dependent component*, which is generated from the system's UML diagrams and includes the bindings' generation code specific to the concrete application, and a *context-*

independent component, which contains the bindings' generation code that is common to all applications (see Figure 8).

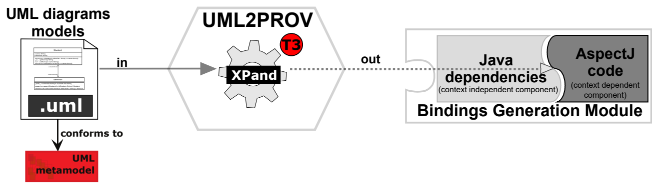


Fig. 8. Detailed MDD-based implementation of the PROV template generation process

Concretely, the *context-dependent component* is implemented in AspectJ. In order to improve the understandability of the component, Figure 9 depicts a generic internal structure of it. This structure encompasses four blocks. The first block contains all the *pointcuts* defining the moments where it is required to execute the bindings generation instructions; for instance, the first line identifies the construction of a new `object`, whereas the second line identifies an `object`'s `operation` call. The remainder blocks represent the set of instructions to be executed when a pointcut is reached during the execution of the application. Concretely, the second and forth blocks contain the method's invocation with the instructions for the binding generation, which are executed before and after the actual method's behaviour, respectively. The third block executes the actual method's behaviour.

```
public aspect BindingsGenerationModule{

     Object around():                              Block 1
           initialization(<object>.new(..)) ||
           call(* <object>.<operation>(..))
      {
Block 2   bindingsBeforeMethodExecution();

Block 3   Object rtn = proceed();

Block 4   bindingsAfterMethodExecution();

          return rtn;
      }
}
```

Fig. 9. Structure overview of an *aspect* in AspectJ for collecting bindings.

After explaining the *context-dependent component*'s structure, now we focus on the process to automatically generate it –that is, how we automatically generate the AspectJ code which provides the application with the behaviour to collect bindings during the application's execution. Our proposal is an MDD-based approach, encompassing a single transformation implemented in Xpand which is identified as *T3* (see Figure 8). This transformation takes as source the *UML diagram models* (conforming to the UML metamodel), and generates the AspectJ code for bindings generation. More specifically, this transformation traverses the UML diagrams models and identifies the operations to be traced. Such operations are included in the Block 1 of Figure 9 as pointcuts, thereby the instructions for binding generation (Block 2 and 4) will be executed when an identified operation is called. The resulted code constitutes the *context-dependent component* of the *bindings generation module*, whose structure follows the structure showed in Figure 9. Finally, we remark that this *context-dependent component* relies on a common set of instructions for the bindings generation and some java dependencies. Such set of instructions and dependencies are located separately in the "Java dependencies" module (*the context-independent component*).

3 Design of GelJ

GelJ [3] is a Java application used in the context of biology for analysing DNA fingerprint gel-images. Briefly speaking, it automatically detects lanes and bands within gel-images by means of an assistant called *experiment wizard*. This wizard guides the user throughout different steps, generating an *experiment* as a result.

All the files containing the UML design of the application is provided in [4], which encompasses 14 sequence diagrams, 13 state machine diagrams, and 1 class diagram. Here, we present an excerpt of the UML class diagram (see Figure 10), a sequence diagram and a state machine diagram (see Figure 11 and 12, respectively).

The UML class diagram depicted in Figure 10 shows 4 classes (`step1_1`, `step1_2`, `step1_3`, and `step1_4`), each class representing a sub-step during the process to generate a new experiment in GelJ. Concretely, these sub-steps are located in the first phase (step 1), in which the gel-image is preprocessed. Among all the operations, we remark those with the name `jButton2ActionPerformed`. This operation name appears in all the classes and its invocation represents the change to the next sub-step. Finally, we note that this UML class diagram has been automatically generated by applying reverse engineering, supported by the main developer of GelJ. Once we have obtained the UML class diagram, we have linked each operation with its corresponding behaviour through the stereotypes described in the paper.

Fig. 10. An excerpt of the UML class diagram representing the internal structure of GelJ

The UML sequence diagram presented in Figure 11 shows the interaction between the objects `step1_1` and `step1_2` (i.e., steps 1.1 and 1.2 in the *experiment wizard*). More specifically, it shows the optional arriving of two messages called *jbCropActionPerformed* and *jbUndoActionPerformed*, which correspond to the action of "crop" and "undo" in the step 1.1 of GelJ. In addition, the change of step from step 1.1 to step 1.2 is carried out after the reception of the message *jButton2ActionPerformed*, which starts an `ExecutionSpecification` from which is sent the message to create the object `step1_2`. The set of sequence diagrams corresponding to the design of GelJ has been developed through a process of manual reverse engineering supported by the main developer of GelJ.

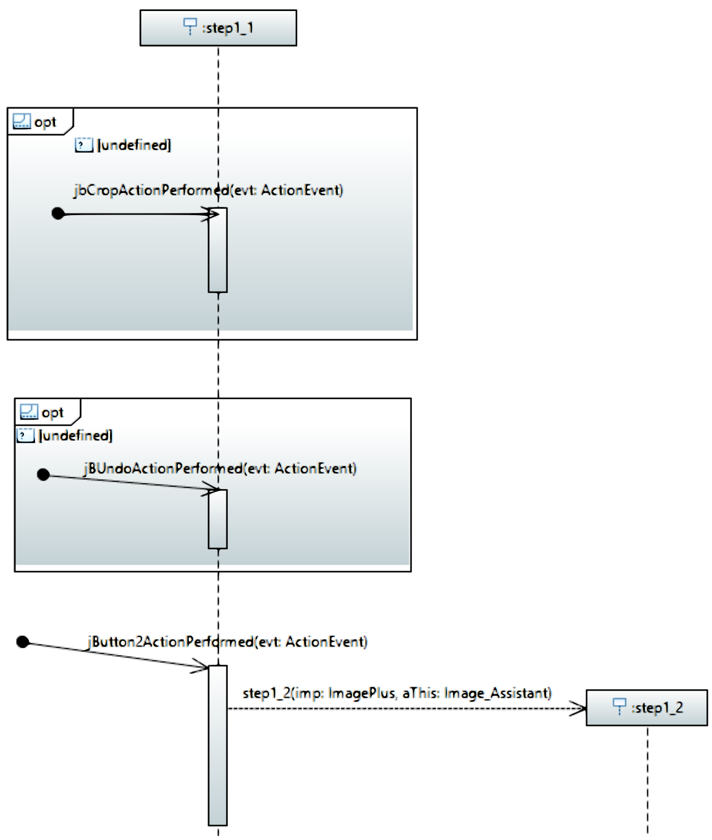


Fig. 11. An UML Sequence Diagram representing the interaction between `step1_1` and `step1_2` (`steps 1.1` and `1.2` in the *experiment wizard*)

The UML state machine diagram depicted in Figure 12 shows the behaviour of the object `step1_1` (step 1.1 in GelJ). It is made up of three states, representing when: the user reaches the step 1.1 (`Activated`), the user crops the image

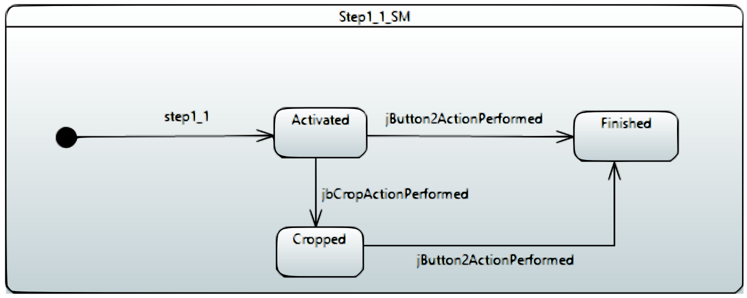


Fig. 12. An UML State Machine Diagram of `Step1_1`

(`Cropped`), and when the user leaves the step (`Finished`). In the same way as sequence diagrams, the state machine diagrams presented herein have been generated by applying a manual process of reverse engineering supported by the main developer of GelJ.

4 Qualitative evaluation (extended)

To evaluate if the generated provenance is meaningful, we have involved a group of GelJ users, asking them for questions they would be interested in getting answer. As a result we have identified nine questions identified from Q1 to Q9 which are addressed throughout the following subsections. In such subsections, we provide a complete description for each question, together with SPARQL query defined to answer it and the final answer. Here we note that with the aim of being as generic as possible, the SPARQL queries have a variable identified as "NameOfExperiment" which corresponds to the name of the experiment which is the subject of the query. In addition, we also want to remark that the final answers correspond to the SPARQL queries applied to the experiment `exp8` described in the paper.

Q1. What is the origin of the experiment?

Gel-images are analyzed throughout experiments, and such experiments might be created in different ways (from scratch, duplicating, or importing another experiment). The scientist is interested to know if the experiment was created from scratch, by duplicating another experiment or by importing an experiment from a file. In case of duplicating or importing an experiment, the scientist also wants to know the identifier of the source experiment.

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select ?result
where{
 ?experiment a exe:Experiment;
  prov:hadMember ?attribute;
  prov:wasGeneratedBy ?act.

 ?attribute u2p:attName ?attribute_name; prov:value ?attribute_value.
 filter(?attribute_value="NameOfExperiment"^^xsd:string)
 filter(?attribute_name="name"^^xsd:string)

 ?act prov:wasStartedBy ?actStarter.
 ?actStarter a prov:Entity; prov:wasGeneratedBy ?act2.

 ?act2 a ?act2type; prov:startedAtTime ?act2Start.

 BIND (IF(?act2type = exe:jButton2ActionPerformed,
                "From scratch",
                IF(?act2type = exe:jbDuplicateExpActionPerformed,
                        "Duplicated", "Imported" ))
       AS ?result).

}
order by asc(?act2Start)
limit 1
```

Listing 1.1. SPARQL query for answering Q1

```
------------------
| result         |
==================
| "From scratch" |
------------------
```

Listing 1.2. Result for Q1

In case of an experiment "Duplicated" or "Imported" the following query returns the name of the source experiment.

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select ?attribute_value
WHERE {
        #experiment generated
        ?experiment a exe:Experiment;  a ?experimentType;
                prov:hadMember ?attribute; prov:wasGeneratedBy ?act.

                ?attribute u2p:attName ?attribute_name;
                    prov:value ?attribute_value.

                filter(?attribute_value!="NameOfExperiment"^^xsd:string)
                filter(?attribute_name="name"^^xsd:string)

    ?act prov:wasStartedBy ?actStarter.
        ?actStarter a prov:Entity; prov:wasGeneratedBy ?act2.

        ?act2 a exe:jbDuplicateExpActionPerformed.
}
order by asc(?act)
limit 1
```

Q2. What is the sequence of activities that has led an experiment to be as it is?

GelJ provides an experiment wizard, which guides the user in 4 steps required to analyze a gel-image and generating, as a result, an experiment. The scientists not only are interested in these steps but also in the activities carried out throughout the process.

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select  ?genwithImageAssistant
where{
    #get the experiment using the name provided the name
    {
        select ?exp2
            where{
                ?experiment a exe:Experiment;
                    prov:hadMember [u2p:attName ?attribute_name;
                                        prov:value ?attribute_value].
                filter(?attribute_value="NameOfExperiment"^^xsd:string)
                filter(?attribute_name="name"^^xsd:string)

                ?experiment prov:qualifiedDerivation* [prov:entity ?exp2].
            }
        order by asc(?exp2)
        limit 1
    }
    #based on the experiment (?exp2) obtain the activity representing
    #the last step
     ?exp2 prov:wasGeneratedBy/prov:wasStartedBy/prov:wasGeneratedBy/
        prov:qualifiedUsage [a exe:StructureUsage; prov:entity ?lastStep].

    #get the image assistant of the step (?lastStep)
     ?lastStep prov:hadMember ?imageAssistant.
     ?imageAssistant u2p:attName ?imageAssistant_name.
     filter(?imageAssistant_name="ia"^^xsd:string)

    #obtain all the activities which have generate an element with the
    #imageAssistant as an attribute
     ?withImageAssistant prov:hadMember ?imageAssistant;
        prov:wasGeneratedBy ?genwithImageAssistant.

    #obtain information about the previous elements
     ?genwithImageAssistant a ?genwithImageAssistant_type;
        prov:startedAtTime ?genwithImageAssistant_start;
        prov:endedAtTime ?genwithImageAssistant_end.
}
```

Listing 1.3. SPARQL query for answering Q2

```
------------------------------------------------------
| genwithImageAssistant                              |
======================================================
| <http://example.org/new-0224>                      |
| <http://example.org/jbCropActionPerformed-0309>    |
| <http://example.org/jButton2ActionPerformed-0314>  |
| <http://example.org/new-0317>                      |
|.........                                           |
------------------------------------------------------
```

Listing 1.4. Result for Q2

Q3. Which background (dark or light) has been used during the experiment construction?

GelJ provides an option to invert the colours of the image, the user has the capability to chose to work with dark- or light-background images. The scientist wants to know whether the experiment was created with dark or light background.

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select ?result
where{
  ?experiment a exe:Experiment;
    prov:hadMember ?attribute;
    prov:wasGeneratedBy ?act.

  ?attribute u2p:attName ?attribute_name; prov:value ?attribute_value.
  filter(?attribute_value="NameOfTheExperiment"^^xsd:string)
  filter(?attribute_name="name"^^xsd:string)

  ?act prov:wasStartedBy ?actStarter.
  ?actStarter a prov:Entity; prov:wasGeneratedBy ?act2.

  ?act2 a ?act2type; prov:startedAtTime ?act2Start.
  ?act2 prov:qualifiedUsage [a exe:StructureUsage;  prov:entity ?entU].

  ?entU prov:hadMember ?attribute2.
  ?attribute2 u2p:attName ?attribute_name2; prov:value ?attribute_value2.
  filter(?attribute_name2="darkbg"^^xsd:string)

BIND (IF(?attribute_value2 = "true"^^xsd:string, "Dark", "Light") AS ?result).

}
order by asc(?act2Start)
limit 1
```

Listing 1.5. SPARQL query for answering Q3

```
----------
| result |
==========
| "Dark" |
----------
```

Listing 1.6. Result for Q3

Q4. Who is the user who has carried out a specific step of the experiment wizard?

The experiment wizard is used by a person who has been logged in the system. The scientist is interested in the name of the user who has carried out a specific step of the experiment wizard. In this case, such a step is represented by a variable called "NameOfStep" in the following SPARQL query.

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select  ?user_id
where{
    #get the experiment using the name provided the name
    {
        select ?exp2
            where{
                ?experiment a exe:Experiment;
                    prov:hadMember [u2p:attName ?attribute_name;
                                        prov:value ?attribute_value].
                filter(?attribute_value="NameOfExperiment"^^xsd:string)
                filter(?attribute_name="name"^^xsd:string)

                ?experiment prov:qualifiedDerivation* [prov:entity ?exp2].
            }
        order by asc(?exp2)
        limit 1
    }

    ?exp2 prov:wasGeneratedBy/ prov:wasStartedBy/ prov:wasGeneratedBy/
        prov:qualifiedUsage [a exe:StructureUsage; prov:entity ?lastStep].

    #get the image assistant of the experiment
    ?lastStep prov:hadMember ?imageAssistant.
    ?imageAssistant u2p:attName ?imageAssistant_name.
    filter(?imageAssistant_name="ia"^^xsd:string)

    #get the elements with the image assistant as an element
    ?withImageAssistant a ?type; prov:hadMember ?imageAssistant.
    FILTER (?type IN (NameOfStep) )

   #user
     ?imageAssistant prov:hadMember/prov:hadMember ?user.
     ?user u2p:attName ?user_name; prov:value ?user_id.
     filter(?user_name="userid"^^xsd:string)
}
limit 1
```

Listing 1.7. SPARQL query for answering Q4

```
--------------
| user_id    |
==============
|"0"         |
--------------
```

Listing 1.8. Result for Q4 (applied to step "exe:step1_1")

Q5. How many lanes have been added/removed during the experiments generation process?

One of the major GelJs capabilities is the lane detection, GelJ automatically identifies the lanes of a gel-image. However, in some situations users might have to add and remove any lane. The scientist wants to know the number of lanes manually added and removed during the experiment's generation process.

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select ?genwithImageAssistant_type
       (count(?genwithImageAssistant_type) as ?numberOfBands)
where{
    #get the experiment using the name provided the name
    {
        select ?exp2
            where{
                ?experiment a exe:Experiment;
                    prov:hadMember [u2p:attName ?attribute_name;
                                    prov:value ?attribute_value].
                filter(?attribute_value="NameOfExperiment"^^xsd:string)
                filter(?attribute_name="name"^^xsd:string)

                ?experiment prov:qualifiedDerivation* [prov:entity ?exp2].
            }
        order by asc(?exp2)
        limit 1
    }


    ?exp2 prov:wasGeneratedBy/ prov:wasStartedBy/ prov:wasGeneratedBy/
        prov:qualifiedUsage [a exe:StructureUsage; prov:entity ?lastStep].

    #get the image assistant of the experiment
    ?lastStep prov:hadMember ?imageAssistant.
    ?imageAssistant u2p:attName ?imageAssistant_name.
    filter(?imageAssistant_name="ia"^^xsd:string)

    #get the elements with the image assistant as an element
    ?withImageAssistant prov:hadMember ?imageAssistant;
    prov:wasGeneratedBy ?genwithImageAssistant;  a exe:step2_2.
    ?genwithImageAssistant a ?genwithImageAssistant_type.
    filter(?genwithImageAssistant_type=exe:jButton1ActionPerformed)
}
group by ?genwithImageAssistant_type
```

Listing 1.9. SPARQL query for answering Q5

```
----------------------------------------------
| genwithImageAssistant_type | numberOfBands |
==============================================
|                            | 0             |
----------------------------------------------
```

Listing 1.10. Result for Q5

Q6. What is the height-threshold used for band detection during the experiment's generation process?

Although GelJ provides a default height-threshold in order to automatically identify bands, the optimum height-threshold can vary in gel-images, so the user can manually fix this value. The scientist is interested to know "What is the height-threshold used for band detection during the experiment's generation process?."

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select ?threshold_value
where{
    #get the experiment using the name provided the name
    {
        select ?exp2
            where{
                ?experiment a exe:Experiment;
                    prov:hadMember [u2p:attName ?attribute_name;
                                        prov:value ?attribute_value].
                filter(?attribute_value="NameOfExperiment"^^xsd:string)
                filter(?attribute_name="name"^^xsd:string)

                ?experiment prov:qualifiedDerivation* [prov:entity ?exp2].
            }
        order by asc(?exp2)
        limit 1
    }



    ?exp2 prov:wasGeneratedBy/ prov:wasStartedBy/ prov:wasGeneratedBy/
        prov:qualifiedUsage [a exe:StructureUsage; prov:entity ?lastStep].

    #get the image assistant of the experiment
    ?lastStep prov:hadMember ?imageAssistant.
    ?imageAssistant u2p:attName ?imageAssistant_name.
    filter(?imageAssistant_name="ia"^^xsd:string)


    #get the elements with the image assistant as an element
    ?withImageAssistant a exe:Step4_1; prov:hadMember ?imageAssistant;
        prov:wasGeneratedBy ?genwithImageAssistant; prov:hadMember ?threshold.
    ?threshold u2p:attName ?threshold_name; prov:value ?threshold_value.
    filter(?threshold_name="threshold"^^xsd:string)
}
order by desc(?withImageAssistant)
limit 1
```

Listing 1.11. SPARQL query for answering Q6

```
-------------------
| threshold_value |
===================
| "11.0"           |
-------------------
```

Listing 1.12. Result for Q6

Q7. How many bands have been added/removed during the experiment's generation process?

Whilst, GelJ automatically detects bands of gel-images based on a height-threshold, it also provides users with the capability of adding and removing bands. The scientist wants to know "How many bands have been added/removed during the experiment's generation process? ."

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select ?genwithImageAssistant_type
       (count(?genwithImageAssistant_type) as ?numberOfLanes)
where{
    #get the experiment using the name provided the name
    {
        select ?exp2
            where{
                ?experiment a exe:Experiment;
                    prov:hadMember [u2p:attName ?attribute_name;
                                        prov:value ?attribute_value].
                filter(?attribute_value="NameOfExperiment"^^xsd:string)
                filter(?attribute_name="name"^^xsd:string)

                ?experiment prov:qualifiedDerivation* [prov:entity ?exp2].
            }
        order by asc(?exp2)
        limit 1
    }
    ?exp2 prov:wasGeneratedBy/ prov:wasStartedBy/ prov:wasGeneratedBy/
        prov:qualifiedUsage [a exe:StructureUsage; prov:entity ?lastStep].

    #get the image assistant of the experiment
    ?lastStep prov:hadMember ?imageAssistant.
    ?imageAssistant u2p:attName ?imageAssistant_name.
    filter(?imageAssistant_name="ia"^^xsd:string)

    #get the elements with the image assistant as an element
    ?withImageAssistant prov:hadMember ?imageAssistant;
        prov:wasGeneratedBy ?genwithImageAssistant;
        a exe:Step4_3.
    ?genwithImageAssistant a ?genwithImageAssistant_type.
    filter(?genwithImageAssistant_type=exe:jButton1ActionPerformed)
}
group by ?genwithImageAssistant_type
```

Listing 1.13. SPARQL query for answering Q7

```
----------------------------------------------
| genwithImageAssistant_type | numberOfLanes |
==============================================
|                            | 0             |
----------------------------------------------
```

Listing 1.14. Result for Q7

Q8. What is the sequence of activities carried out during the pre-processing phase?

The first step (Step 1) during the creation of a new experiment using the *experiment wizard* is the pre-processing step. The scientist is interested to know if the image has been cropped, inverted or adjusted; what is the time expended in each sub-step?, what is the name of the image used?, which is the user involved?.

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select ?type ?state ?startTime ?imageName  ?user_id
where{
    #get the experiment using the name provided the name
    {
        select ?exp2
            where{
                ?experiment a exe:Experiment;
                    prov:hadMember [u2p:attName ?attribute_name;
                                        prov:value ?attribute_value].
                filter(?attribute_value="NameOfExperiment"^^xsd:string)
                filter(?attribute_name="name"^^xsd:string)

                ?experiment prov:qualifiedDerivation* [prov:entity ?exp2].
            }
        order by asc(?exp2)
        limit 1
    }

    ?exp2 prov:wasGeneratedBy/ prov:wasStartedBy/ prov:wasGeneratedBy/
        prov:qualifiedUsage [a exe:StructureUsage; prov:entity ?lastStep].

    #get the image assistant of the experiment
    ?lastStep prov:hadMember ?imageAssistant.
    ?imageAssistant u2p:attName ?imageAssistant_name.
    filter(?imageAssistant_name="ia"^^xsd:string)

    #get the elements with the image assistant as an element
    ?withImageAssistant a ?type; prov:hadMember ?imageAssistant.
    FILTER (?type IN (exe:step1_1, exe:step1_2,exe:step1_3, exe:Step1_4) )

   #the state
   optional{
       ?withImageAssistant u2p:state ?state.
    }

   #time
      ?withImageAssistant prov:wasGeneratedBy [prov:startedAtTime ?startTime].

   #user
     ?imageAssistant prov:hadMember/prov:hadMember ?user.
     ?user u2p:attName ?user_name; prov:value ?user_id.
     filter(?user_name="userid"^^xsd:string)

   #image
     ?exp2 prov:hadMember ?image.
     ?image u2p:attName ?image_name; prov:value ?imageName.
     filter(?image_name="imagename"^^xsd:string)

}
order by ?startTime
```

Listing 1.15. SPARQL query for answering Q8

```
--------------------------------------------------------------------------------
|type        | state            | startTime                 | imageName | user_id |
================================================================================
|exe:step1_1 | exe:Activated> | "...17:23:45.000+02:00"| "3.tif"   | "0"     |
|exe:step1_1 | exe:Cropped>   | "...17:23:56.000+02:00"| "3.tif"   | "0"     |
|exe:step1_1 | exe:Finished   | "...17:23:57.000+02:00"| "3.tif"   | "0"     |
|exe:step1_2 | exe:Activated  | "...17:23:57.000+02:00"| "3.tif"   | "0"     |
...
```

Listing 1.16. Result for Q8

Q9. What is the time-cost of creating a new experiment?

The experiment's creation is a user-friendly process supported by the *experiment wizard*. If the cost of create a new experiment is more than 10 minutes, it means that the user is not using the wizard properly. The scientist is interested to know the time-cost of creating a new experiment.

```
prefix prov: <http://www.w3.org/ns/prov#>
prefix exe: <http://example.org/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix xsd:   <http://www.w3.org/2001/XMLSchema#>
prefix u2p:  <http://uml2prov.org/>

select  ?genwithImageAssistant_type
    (max(?genwithImageAssistant_end) - min(?genwithImageAssistant_start)
        as ?timeConsumed)
where{
    #get the experiment using the name provided the name
    {
        select ?exp2
            where{
                ?experiment a exe:Experiment;
                    prov:hadMember [u2p:attName ?attribute_name;
                                        prov:value ?attribute_value].
                filter(?attribute_value="NameOfExperiment"^^xsd:string)
                filter(?attribute_name="name"^^xsd:string)

                ?experiment prov:qualifiedDerivation* [prov:entity ?exp2].
            }
        order by asc(?exp2)
        limit 1
    }
    ?exp2 prov:wasGeneratedBy/ prov:wasStartedBy/ prov:wasGeneratedBy/
        prov:qualifiedUsage [a exe:StructureUsage; prov:entity ?lastStep].

    #get the image assistant of the experiment
    ?lastStep prov:hadMember ?imageAssistant.
    ?imageAssistant u2p:attName ?imageAssistant_name.
    filter(?imageAssistant_name="ia"^^xsd:string)

    #get the elements with the image assistant as an element
    ?withImageAssistant prov:hadMember ?imageAssistant;
        prov:wasGeneratedBy ?genwithImageAssistant.

    ?genwithImageAssistant a ?genwithImageAssistant_type;
        prov:startedAtTime ?genwithImageAssistant_start;
        prov:endedAtTime ?genwithImageAssistant_end.
    filter(?genwithImageAssistant_type=prov:Activity)
}
group by ?genwithImageAssistant_type
```

Listing 1.17. SPARQL query for answering Q8

```
--------------------------------------------------------------
| timeConsumed                                               |
==============================================================
| "PT2M28.000S"^^<http://www.w3.org/2001/XMLSchema#duration> |
--------------------------------------------------------------
```

Listing 1.18. Result for Q9

5 Overall process

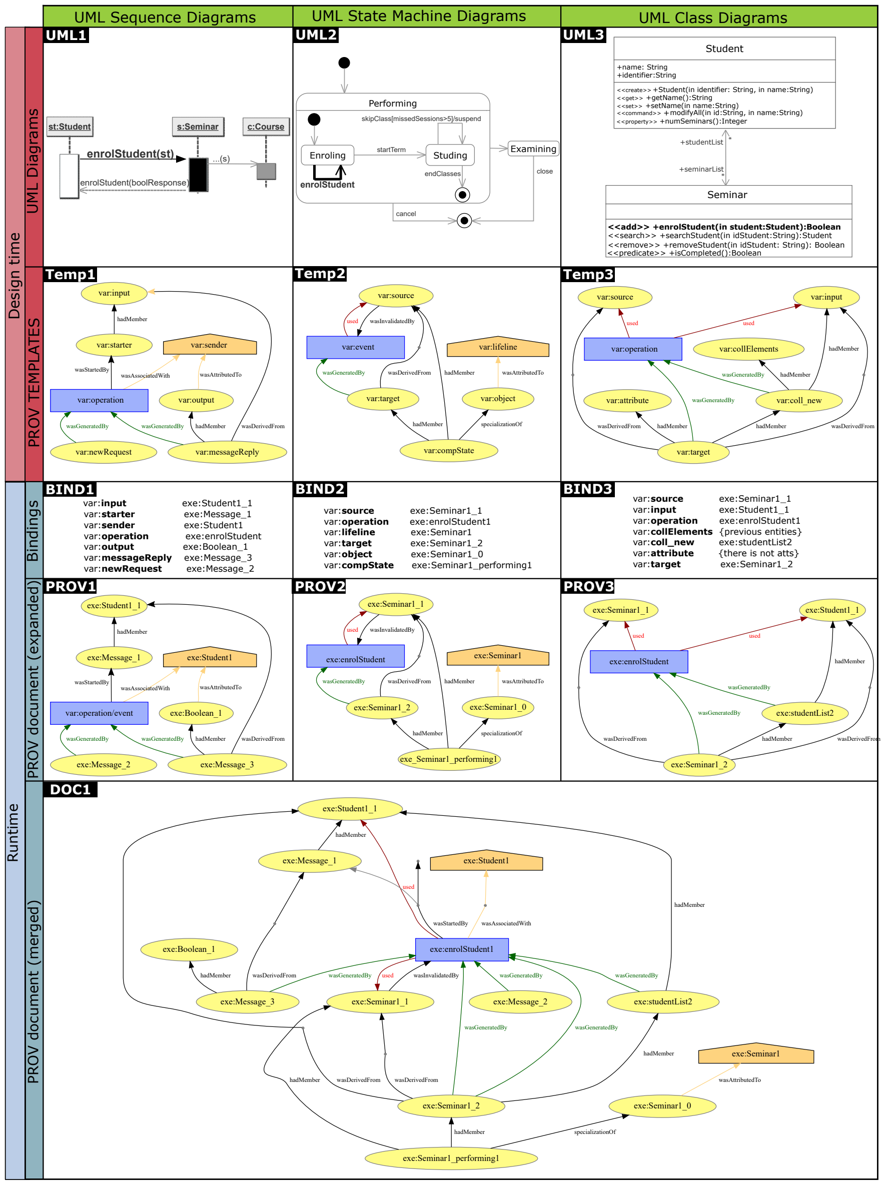
Here we provide an example of all the elements which conform the architecture of UML2PROV in a summarized way (see Table 7). This Table is organized as follows: the first and second row depict the design time facets, whereas the third, forth and fifth show runtime facets.

As for the design time facets, the first row depicts the design of the *University* case study, which refers to the enrolment and attendance of students to seminars that are held during specific University courses. In particular, we have used an UML Sequence (Figure UML1), State Machine (Figure UML2), and Class (Figure UML3) Diagram; highlighting in bold the *message*, *transition*, and *operation* called **enrolStudent**. Based on the patterns from Section 1:

- The message called enrolStudent from the UML Sequence Diagram has been translated according to SQP2 and SQP3 (see Subsection 1.1) to the PROV template in Figure Temp1.
- The event named enrolStudent from the UML State Machine Diagram has been translated based on StP4 (see Subsection 1.2) to the PROV template in Figure Temp2.
- The operation with the name enrolStudent from the UML Class Diagram has been translated according to CDP9 (see Subsection 1.3) to the PROV template in Figure Temp3.

Subsequently, during the execution of the University case study, we have collected a set of bindings linking each one of the variables from the PROV templates with values from the execution. Concretely, an example of such a bindings is depicted in the third column (Figures BIND1-3). Then, the expansion process [?] takes the PROV templates (Figures Temp1-3) generated from the UML diagrams together with the bindings collected during the execution (Figures BIND1-3), and replaces each variable from the PROV template with the value(s) given by the binding document, generating the final PROV documents showed in the forth row (Figures PROV1-3). Finally, after merging all the previous PROV documents we obtain the final PROV document (Figure DOC1) which contains all the provenance information.

Table 7. Set of elements involved in the UML2PROV architecture.



References

1. OMG: Unified Modeling Language (UML). Version 2.5 (2015) Document formal/15-03-01, March, 2015.
2. Moreau, L., Missier (eds.), P., Belhajjame, K., B'Far, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., Klyne, G., Lebo, T., McCusker, J., Miles,

S., Myers, J., Sahoo, S., Tilmes, C.: PROV-DM: The PROV Data Model. W3C Recommendation REC-prov-dm-20130430, World Wide Web Consortium (2013)
3. Heras, J., Domínguez, C., Mata, E., Pascual, V., Lozano, C., Torres, C., Zarazaga, M.: GelJ – a tool for analyzing DNA fingerprint gel images. BMC Bioinformatics **16**(1) (Aug 2015)
4. : UML design of GelJ (2018) `https://github.com/uml2prov/GelJUML`, Last visited on September 2018.