TOWARDS A FRAMEWORK FOR MAKING APPLICATIONS PROVENANCE-AWARE

IMPLEMENTATION DETAILS

Herein we provide a full detailed description of the Model Driven Development (MDD) approach for implementing UML2PROV, which we succinctly explained in the paper (see Figure 1). With a MDD-based approach, we will focus on models, rather than on computer programs, so that the code programs are automatically generated from them using a refinement process [7]. Among other things, this process entails one or various transformations that describe the way in which a model should be translated into another one. For the interest of this paper, depending on the type of artifacts that are used/generated in the transformation, we distinguishes between *model to model transformations* (M2M), in which both the used and the generated artifacts are models, and *model to text tranformations*, which define transformations from a model to the final text. Our solution for implementing a MDD-based approach comprises both M2M and M2T transformations. In case of M2M transformations, we have used the Atlas Transformation Language (ATL) [1] for being one of the most widely used languages, in addition to provide an IDE developed on top of Eclipse. On the other hand, M2T transformations have been developed by means of XPand [9] due to it also support an IDE developed on top of Eclipse, in addition to have a large user community and significant number of available examples.

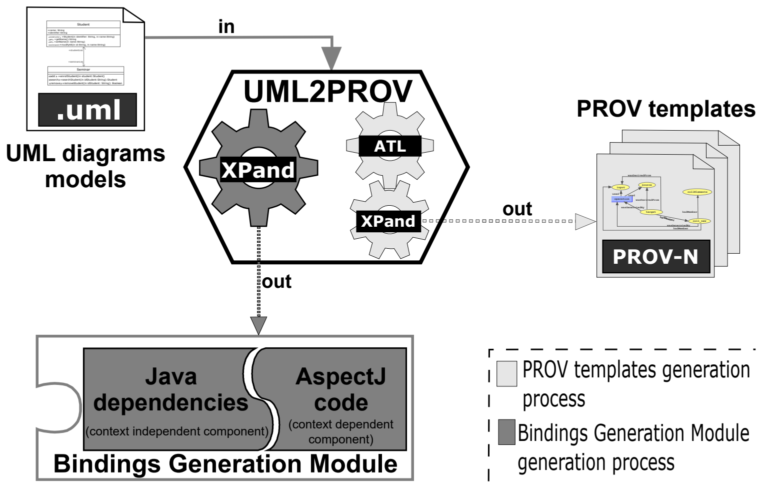

in
.uml
UML diagrams models
UML2PROV
XPand
ATL
XPand
out
PROV templates
PROV-N
out
Java dependencies
(context independent component)
AspectJ code
(context dependent component)
Bindings Generation Module
PROV templates generation process
Bindings Generation Module generation process

Figure 1: MDD-based implementation proposal.

Next, in Section 1, we explain the implementation of the artifacts in charge of the PROV template generation process. Subsequently, in Section 2, we give details regarding the implementation of the BGM and the reference implementation for generating it.

1 PROV templates generation process

Generally speaking, this process takes as source the *UML diagrams models* and automatically generates the *PROV templates files* (Figure 1). Our proposal for implementing this process in UML2PROV follows an MDD-based tool chain that comprises two transformation (see Figure 2). First, a M2M transformation identified by `T1`, whose implementation is explained in Subsection 1.1, and second, a M2T transformation identified by `T2` which is explained in Subsection 1.2.

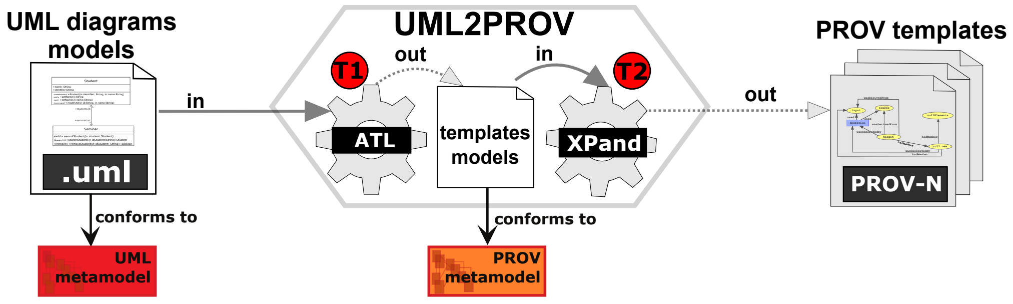


Figure 2: Detailed MDD-based implementation of the PROV template generation process

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

This transformation of type M2M, takes as source the *UML diagram models*, conforming to the UML metamodel [4], and generates the corresponding *templates models*, conforming to the PROV metamodel [3]. To that end, the transformation patterns have been implemented in an ATL module that is made up of a set of ATL rules. Each rule tackles one transformation pattern, describing how certain UML elements are mapped to the specific PROV elements that will conforms the PROV template.

Since the implementation of the patterns using ATL rules is important for our approach, we explain how these ATL rules look like by using the concrete implementation of *ClP1*. To do this, we will rely on Table 1. This table depicts in the middle column the ATL rule implementing *ClP1*; in the left column, the identifier given to the excerpt of the ATL rule next to it; and in the right column, the PROV elements generated from such an excerpt using the graphical notation. The behaviour executed by each excerpt is as follow:

- **E1**. It states that the rule is applied to all the UML operations with the stereotype `«create»`.
- **E2**. This excerpt is in charge of generating a `prov:Entity` with the identifier `var:postObject`.
- **E3**. It creates an `prov:Activity` with the identifier `var:operation`.
- **E4**. This excerpt is responsible for linking `var:postObject` with `var:operation` by means of the PROV relation `prov:wasGeneratedBy`.
- **E5**. In case of existing UML `Input Parameters`, it creates the `prov:Entity` with the identifier `var:input`. In addition, it generates the PROV relation `prov:used` to associate `var:operation` with `var:input`, as well as the PROV relation `prov:wasDerivedFrom` between `var:postObject` and `var:input`.

- **E6**. In case of existing UML `Attributes` in the class to which the addressed UML `Operation` belongs, it generates a new `prov:Entity` identified by `var:attribute`, and the PROV relation `prov:hadMember` between `var:postObject` and `var:attribute`.

Table 1: Excerpt of an ATL rule implementing *ClP1*

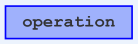
Excerpt ID	ATL Source Code	PROV Template
	`rule OperationCreation2Document {`	

E1	`from`
	` operation: UML!Operation(operation.hasStereotype('create'))`	

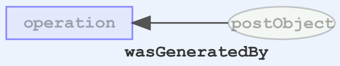
	`[...]`
	`  to`	

E2	`   targetEn: PROV!Entity (`
	`    id <- 'var:target',`
	`    type <- 'exe:' + operation.class.name,`
	`    type<- 'u2p:Object')`
	`    ,`	

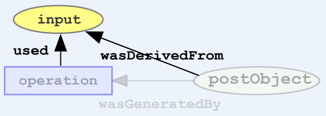
	`[...]`	

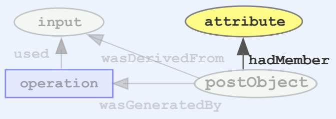
E3	`   operationAct: PROV!Activity(`
	`    id <- 'var:operation'`
	`   ),`
		

	`[...]`	

E4	`   wgb: PROV!Generation(`
	`    entity <- targetID,`
	`    activity <- activityID`
	`   ),`
		

	`[...]`
	`  do {`	

E5	`    if(existIn){`
	`     thisModule.newInputEntity(doc);`
	`     thisModule.genUsage('var:input', operationAct, doc);`
	`     thisModule.genWDF('var:input', targetEn.id, doc);`
	`    }`
		 input
		used
		wasDerivedFrom
		operation
		postObject
		wasGeneratedBy

E6	`    if(hasAttr){`
	`     thisModule.newAttributeEntity(doc);`
	`     thisModule.genHM(targetEn.id,`
	`                       'var:attribute',`
	`                       'exe:ownedAttribute',`
	`                        doc);`
	`    }`
		 input
		attribute
		used
		wasDerivedFrom
		hadMember
		operation
		postObject
		wasGeneratedBy

	`[...]`	

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

`T2` corresponds to a M2T transformation that takes as source the *template models* returned by `T1`, and generates the *PROV templates* files in PROV-N format. This transformation is implemented in an XPand module which contains XPand templates that associate each PROV element/relation with its PROV-N representation. Figure 3 is an XPand template defined for the PROV document element. This template will be instantiated for each PROV document appearing in the *template models* (line 1), and will create a text file (line 2) with the text inside its definition (in green). This template instantiates, in turn, another template for each PROV element/relation included in the addressed PROV document (lines 3 to 15). As a way of example, Figure 3 also depicts the XPand template (next to line 3) defining the transformation of each `prov:Entity` into PROV-N.

```
1: «DEFINE documentTemplate FOR prov::Document»
2:   «FILE id+ ".provn"»
       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                          { «DEFINE entityTemplate FOR Entity»
3:         «EXPAND entityTemplate FOREACH entity»  {   entity(«this.id» «EXPAND entityAttributeTemplate FOR this»)
4:         «EXPAND activityTemplate FOREACH activity» { «ENDDEFINE»
5:         «EXPAND agentTemplate FOREACH agent»
6:         «EXPAND wgbTemplate FOREACH wasGeneratedBy»
7:         «EXPAND usedTemplate FOREACH used»
8:         «EXPAND wibTemplate FOREACH wasInvalidatedBy»
9:         «EXPAND wdfTemplate FOREACH wasDerivedFrom»
10:        «EXPAND hmTemplate FOREACH hadMember»
11:        «EXPAND spOTemplate FOREACH specializationOf»
12:        «EXPAND watTemplate FOREACH wasAttributedTo»
13:        «EXPAND wawTemplate FOREACH wasAssociatedWith»
14:        «EXPAND wInfByTemplate FOREACH wasInformedBy»
15:        «EXPAND wStartedByTemplate FOREACH wasStartedBy»
       endBundle
     endDocument
    «ENDFILE»»
  «ENDDEFINE»
```

Figure 3: XPand template defined for each PROV document.

2 The BGM generation process

Before digging into the implementation's depths, let us start by explaining how is the design of the BGM to be generated (Subsection 2.1). Once the BGM's design is explained, we will be able to be more precise in the explanation of the process to generate its implementation (Subsection 2.2).

2.1 UML design of the BGM

Here we will present a general design of the BGM. In order to provide a design that not only will allow developers to fulfil the requirements presented in the paper, but also will be agnostic about the technologies used to implement it, we have taken inspiration from the Event-driven Architecture (EDA) mediator topology [6] (see Figure 4). Briefly speaking, it consists of highly decoupled, single-purpose event processing modules that asynchronously receive and process events. We have identified four main components in this topology that we will use for the BGM design: *event*, *client*, *event mediator*, and *event processors*.

- *Event*. We will use the term *event* to refer to a notable occurrence that happens during the execution. Each *event* occurrence has a header and a body. In our case, the header contains the type of *event*, and the body will have data referring to the execution.
- *Client*. The event flow starts when a *client* identifies an *event* during the execution, and consequently, it sends an occurrence of this *event* to an *event mediator*.
- *Event mediator*. This component is responsible for disseminating the received *events* by sending them to the subscribed *event processors*. Additionally, the *event mediator* is also in charge of subscribing and unsubscribing *event processors*.
- *Event processors*. It contains the business logic necessary to process the received event. It is important to remark that each *event processor* is self-contained and independent.

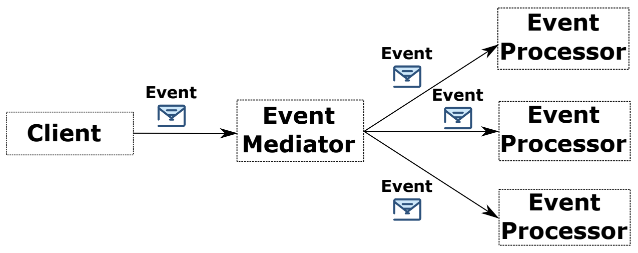


Figure 4: Event-driven architecture mediator topology

Inspired by the above architecture, the design of the BGM is made up of five components (see CD depicted in Figure 5): *BGMEvent*, *Trigger*, *EventManager*, *BGMListener*, and *ConcreteListener*. Following is the description of each component conforming the BGM design.

- *BGMEvent*. It models the component *event* from the EDA. We have defined four types of events related to *operations'* executions: *operationStart*, when an operation is called; *operationEnd*, when the execution of an operation is finished; *newBinding*, when is collected a provenance value linked with a PROV element identifier in a template (e.g., `var:operation`); and *newValueBinding*, when is collected a provenance value associated with a variable from a PROV template, but such a variable is not the identifier of a PROV element in a template (e.g., `var:operationStartTime`).

  The attributes of the *BGMEvent* are the body of the *event*–that is, the information collected during the execution (i.e., provenance data). For instance, *executionID* is the unique identifier of the application's execution; *className* is the name of the class to which the operation executed belongs to; *varName* is the variable from a template associated to a piece of information (attribute *value*).

- *Trigger*. It represents the *client* in the EDA. The *Trigger* is in charge of detecting the identified types of *events* during the execution, to construct an object of type *BGMEvent* representing the occurrence of the *event*, and subsequently, to invoke the operation *fireEvent* in *EventManager*, passing the type of the *event* together with an object of type *BGMEvent* as input parameters. See UML SqD in Figure 6.

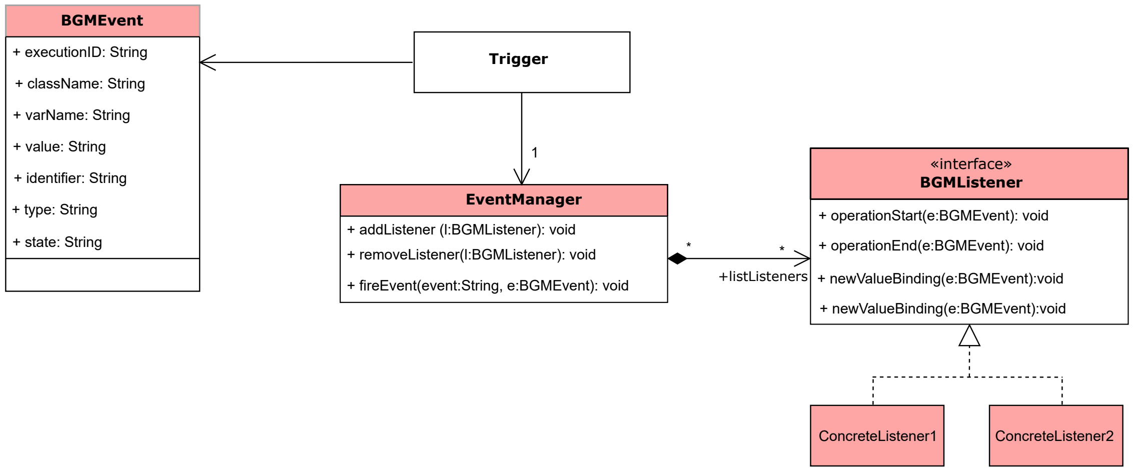

BGMEvent
+ executionID: String
+ className: String
+ varName: String
+ value: String
+ identifier: String
+ type: String
+ state: String
Trigger
1
EventManager
+ addListener (l:BGMListener): void
+ removeListener(l:BGMListener): void
+ fireEvent(event:String, e:BGMEvent): void
*
*
+listListeners
«interface»
BGMListener
+ operationStart(e:BGMEvent): void
+ operationEnd(e:BGMEvent): void
+ newValueBinding(e:BGMEvent):void
+ newValueBinding(e:BGMEvent):void
ConcreteListener1
ConcreteListener2

Figure 5: UML SqD depicting the design of the BGM, which follows the topology shown in Figure 4

- *BGMListener*. It is an interface that declares the operations that each *event processor* has to implement when an *event* is fired. The name of the operations matches with the name of the types of *events* identified (*operationStart*, *operationEnd*, *newBinding*, and *newValueBinding*). As we can see, all the operations have an input parameter of type *BGMEvent*, which contains the provenance data collected during the execution.

- *EventManager*. It models the *event mediator* from the EDA. Thus, the *EventManager* has two main responsibilities: to manage a list of subscribed *event processors*, and to disseminate the received *events* among them. The former is addressed by an operation to subscribe an object implementing *BGMListener* (*addListener*), and an operation to unsubscribe it (*removeListener*). The latter is tackled by the operation *fireEvent*. When this operation is invoked, for each subscribed *BGMListener*, the *EventManager* calls the operation corresponding to the input parameter *eventName*. For instance, if *event* is equals to *operationStart*, it calls the operation *operationStart* in a concrete implementation of the *BGMListener*.

- *ConcreteListener1-2*. They model the *event processors*. These classes implements the *BGMListener* interface; thus, they implement the behaviour to be executed when a declared operation in *BGMListener* is called–that is, when an *event* is fired. This implementation will constitute the mechanism used to manage the data provenance collected, and persitence approach to be followed. For instance, it could store the information as logs, and later generate *bindings*, as opposed to directly creating the bindings during the application's execution. In turn, it is also responsible for the provenance storage. In some circumstances it is preferable to store the data in a centralized or distributed way, or it can use the most convenient storage system. At this point, it is worth remarking that there could be several implementations of the *BGMListener* (e.g., *ConcreteListener1* and *ConcreteListener2*) at the same time, which enables several strategies for managing and storing data provenance at the same time, replicating the data provenance.

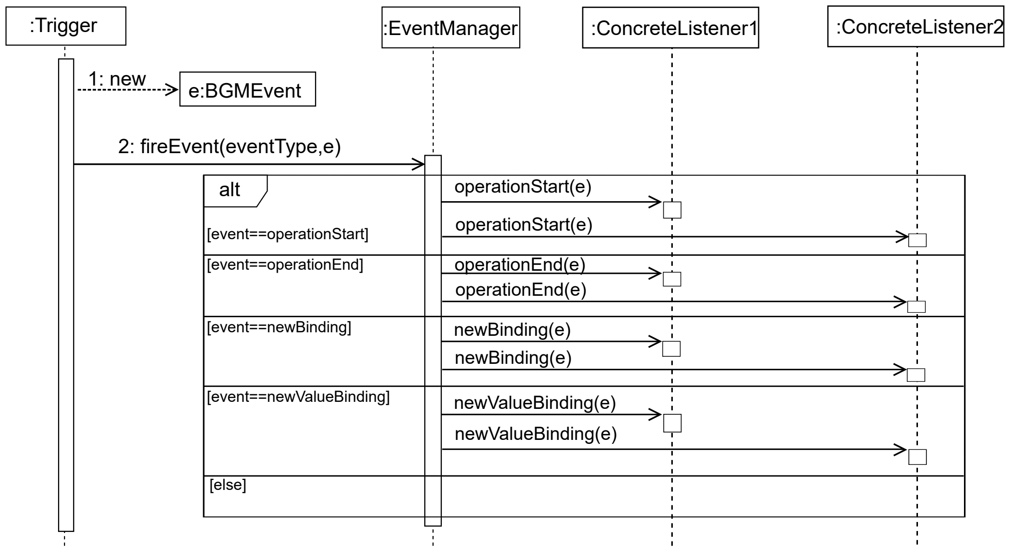


Figure 6: UML Sequence Diagram representing the interactions between *Trigger*, *EventManager*, and *ConcreteListener* when an *event* is identified.

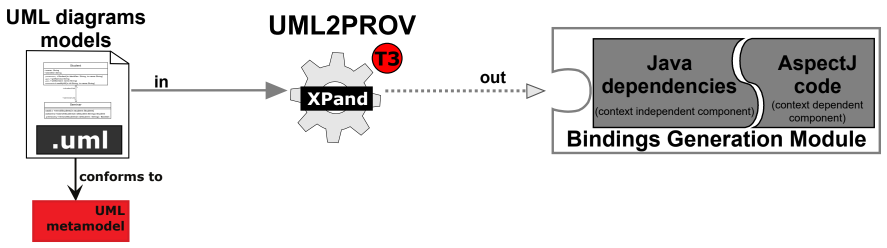


Figure 7: Detailed MDD-based implementation of the PROV template generation process

2.2 Reference implementation of the process for generating the BGM

This process performs one M2T transformation, referred to as `T3` in Figure 7. To that end, we have implemented an XPand module that takes as source the *UML diagram models*, conforming the UML metamodel [4], and generates the Java-related source code implementing the BGM design presented in Figure 5.

The generated the BGM's components (Figure 5) are divided into two groups. The first group encompasses those elements that do not depend on the source *UML diagram models*, and therefore, their implementations are the same in all the BGMs. This group, which is referred to as *context independent components*, is made up of *BGMEvent*, *EventManager*, *BGMListener*, and the concrete implementations of the *BGMListener* (e.g., *ConcreteListener1*). The second group, called *context dependent components*, is made up of those elements whose implementation depends on the source *UML diagram models*. The only element in this group is the *Trigger*, whose implementation will be in accordance with the information exposed in the source UML design.

2.2.1 *Context independent components* generation process

As for the generation of the *context independent components*, all of them will have the same implementation for all the generated BGMs, regardless of the source *UML diagrams models*. Thus, the XPand module will generate the same source code for all the BGMs. Concretely, the *BGMEvent* and *BGMListener* correspond to a Java class and a Java interface implemented as they are designed in Figure 5. The *EventManager* is also a Java class implementing the behaviour of the operations *addListener*, *removeListener*, and *fireEvent*. In the case of the operation *fireEvent*, the implementation follows the generic specification described in Figure 6.

Regarding the implementation of the *ConcreteListener1*, it is important to go into the details since different implementation strategies will yield different overheads. The implementation of this class corresponds with the implementation of the operations given by the interface *BGMListener*–that is, *operationStart*, *operationEnd*, *newValueBinding*, and *newBinding*. Below, we will explain our reference implementation of each operation, relying on the SqDs in Figures 8-10.

- *operationStart*. It implements the behaviour to be executed when an operation call is identified. We rely on Figure 8 for explaining such a behaviour. (**1**) data provenance regarding the identified operation call is passed as input parameter into the execution. (**2**) it creates an object of type *Bindings*, which represents the set of *bindings* associated to the called operation. (**3**) it obtains the identifier of the called operation from the input parameter. (**4**) it keeps in an object of type *Map* the identifier of the called operation, as key, and the *Bindings*, as value.

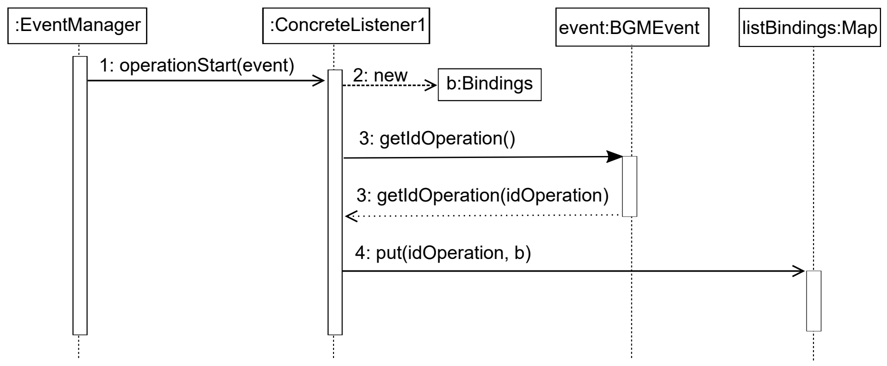


Figure 8:

- *operationEnd*. It implements the behaviour to be executed when an operation's execution ends. We rely on Figure 9 for explaining such a behaviour. (**1**) data provenance regarding the end of the operation's execution is passed as input parameter into the execution. (**2**) it obtains the identifier of the called operation from the input parameter. (**3**) from the Map, it gets the object of type *Bindings* whose key is the identifier previously retrieved. (**4**) it performs the behaviour corresponding with the storage of the information in the object *Bindings* in MongoDB database. (**5**) it removes the entry in the Map whose key corresponds to the previously retrieved operation's identifier.

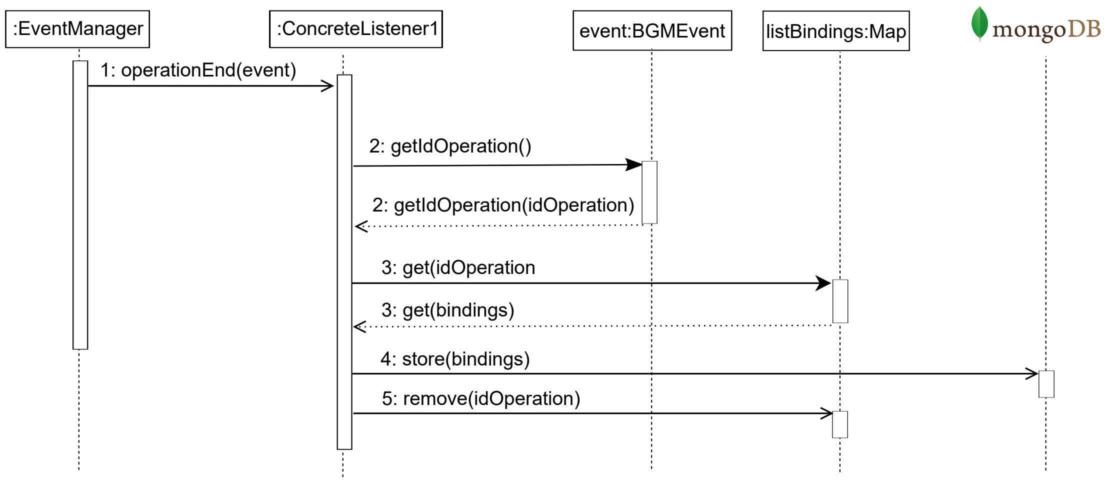


Figure 9:

- *newValueBinding* and *newBinding*. On the one hand, *newValueBinding* is executed when is collected a provenance value associated with a variable from a PROV template, and this variable is not the identifier of an element. On the other hand, *newBinding* is called when the variable associated with the collected value is an identifier of an element in a PROV template. Both implementations are similar, so we rely on Figure 10 for explaining both of them. (**1**) the data provenance collected is passed as input parameter into the execution. From the information passed into the execution, it gets (**2**) the identifier of the operation's execution, (**3**) the name of the logged variable, and (**4**) the value associated with the variable. (**5**) from the Map, it gets the object of type *Bindings* whose key is the identifier of the operation's execution. (**6**) It adds the pair variable-value into the retrieved object of type *Bindings*. In case of the operation *newValueBinding*, it uses the operation *addVariable*, whereas in *newValueBinding*, it uses the operation *addAttribute*. The difference between these operations can be see in the ProvToolbox documentation [5].

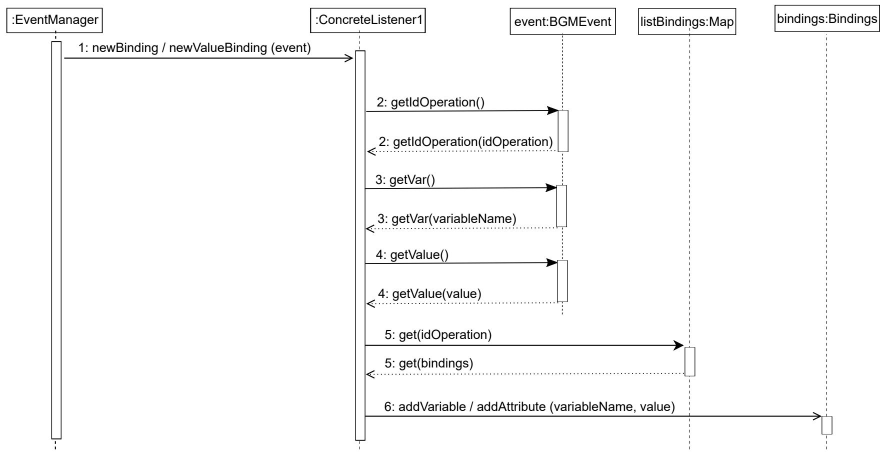
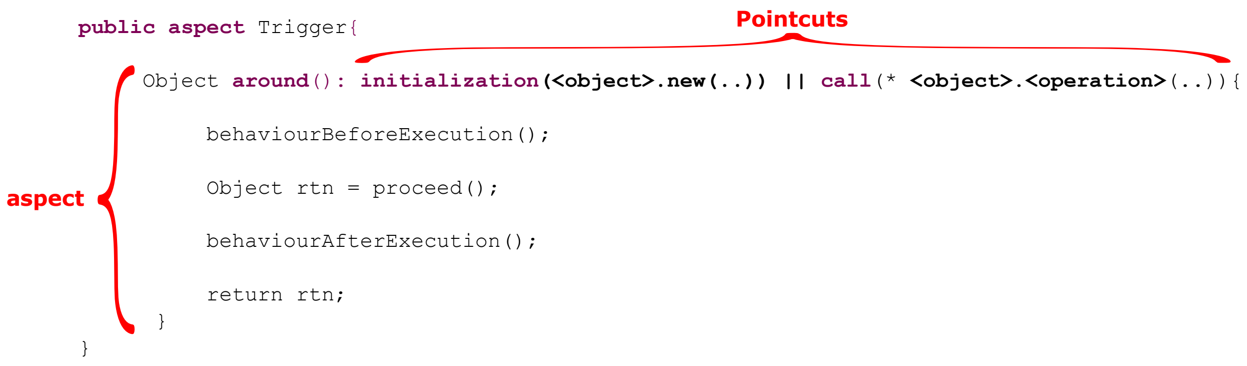


Figure 10:

2.2.2 *Context dependent components* generation process

The *context-dependent component* (i.e., *Trigger*) is the key component of the implementation since it is in charge of detecting the *events*, collecting data provenance, and additionally it has to inform the *EventManager* for disseminating the collected data provenance. To do that, we have chosen the Aspect Oriented Paradigm (AOP) [2] to implement it, and concretely, we have used Eclipse AspectJ [8], which is an AOP extension created for Java code.

The *Trigger* is an AOP *aspect* that represents the modularization of a cross-cutting concern, in this case the capture of data provenance. This *aspect* has two main types of components: *advices*, which define the behaviour for collecting data provenance, and informing the *EventManager*; and *pointcuts* for identifying where each *advice* is executed. The XPand module for generating the BGM creates the AspectJ implementation of the *Trigger*, which follows the structure depicted in Figure 11.



```
public aspect Trigger{

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

        behaviourBeforeExecution();

        Object rtn = proceed();

        behaviourAfterExecution();

        return rtn;
    }
}
```

Figure 11: Structure overview of a reference implementation of *Trigger* in AspectJ

The Trigger is always an AspectJ *aspect* with a list of *pointcuts*, as well as an *advice* of type *around*. The list of *pointcuts* is generated on the basis of the source *UML diagram models*. The XPand module creates a *pointcut* for each (i) operation call modelled by means of `Messages` in SqDs, (ii) the occurrence of an operation represented by an `Event` in SMDs, and (iii) an `Operation` from CDs. Regarding the *advice* of type *around*, it allows us to capture the call of an operation, which is given by the pointcuts, and proceed to the operation's execution when we consider. Concretely, we perform custom behaviour both before (method *behaviourBeforeExecution* in Figure 11) and after (method *behaviourAfterExectuion*) proceeding to the actual operation's execution (method *proceed*).

Briefly speaking, the method *behaviourBeforeExecution* is in charge of sending the *event* of type *startOperation*, collecting data provenance previous to the execution (e.g., the time value for the start of the operation, or the object in the status/state pre-operation's execution), and sending the event of type *newBinding* or *newValueBinding* with the collected data provenance. On the other hand, the method *behaviourAfterExectuion* is responsible for collecting data provenance after to the execution (e.g., the time value for the end of the operation, or the object in the status/state post-operation's execution), sending the event of type *newBinding* or *newValueBinding* with the collected data provenance, and finally, sending the *event endOperation*.

References

[1] ATL - a model transformation technology, version 3.8. Available at `http://www.eclipse.org/atl/`. Last visited on September 2018.

[2] 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'97)*, pages 220–242, Berlin, Heidelberg, 1997.

[3] 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.

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

[5] ProvToolbox. Java library to create and convert W3C PROV data model representations. `http://lucmoreau.github.io/ProvToolbox/`. Last visited on September 2018.

[6] M. Richards. *Software architecture patterns*. O'Reilly Media, Incorporated, 2015.

[7] B. Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[8] The AspectJ Project. Available at `www.eclipse.org/aspectj/`. Last visited on Febrary 2019.

[9] XPand. Eclipse platform. `https://wiki.eclipse.org/Xpand`, Last visited on September 2018.