
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 distinguish between *model to model transformations* (M2M), in which both the used and the generated artifacts are models, and *model to text transformations*, 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.

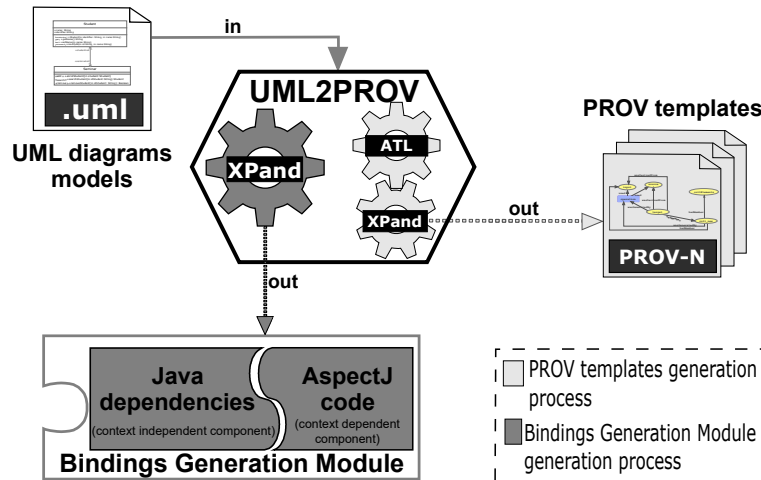


Figure 1: MDD-based implementation proposal.

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.

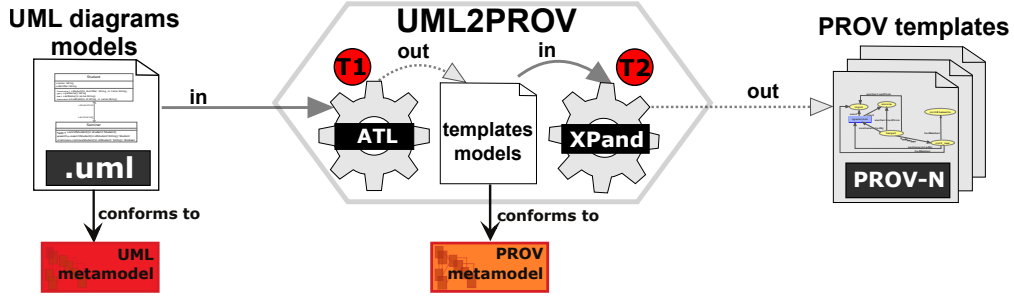


Figure 2: Detailed MDD-based implementation of the PROV template generation process

1.1 Transformation T1: from UML diagram models to template models


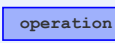
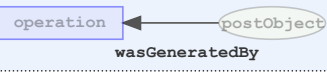
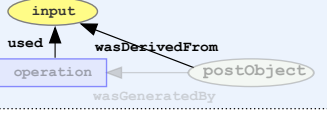
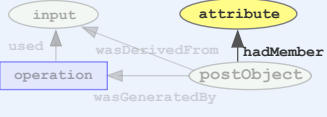
This transformation of type M2M, takes as source the *UML diagram models*, conforming to the UML metamodel [4], and generates the corresponding *templates models*, conforming to the PROV metamodel [3]. To that end, the transformation patterns have been implemented in an ATL module that is made up of 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 *CIP1*. To do this, we will rely on Table 1. This table depicts in the middle column the ATL rule implementing *CIP1*; 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 *CIP1*

Excerpt ID	ATL Source Code	PROV Template
E1	<pre>rule OperationCreation2Document { from operation: UML!Operation(operation.hasStereotype('create')) to [...]</pre>	
E2	<pre> targetEn: PROV!Entity (id <- 'var:target', type <- 'exe:' + operation.class.name, type<- 'u2p:Object') , [...]</pre>	
E3	<pre> operationAct: PROV!Activity(id <- 'var:operation'), [...]</pre>	
E4	<pre> wgb: PROV!Generation(entity <- targetID, activity <- activityID), [...]</pre>	
E5	<pre> do { if(existIn){ thisModule.newInputEntity(doc); thisModule.genUsage('var:input', operationAct, doc); thisModule.genWDF('var:input', targetEn.id, doc); } }</pre>	
E6	<pre> if(hasAttr){ thisModule.newAttributeEntity(doc); thisModule.genHM(targetEn.id, 'var:attribute', 'exe:ownedAttribute', doc); } [...]</pre>	

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
3:   «EXPAND entityTemplate FOREACH entity»
4:   «EXPAND activityTemplate FOREACH activity»
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»»
«ENDEDEFINE»

```

{ «DEFINE entityTemplate FOR Entity»
 entity(«this.id» «EXPAND entityAttributeTemplate FOR this»)
 «ENDEDEFINE»

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 implementation of the BGM to be generated (Subsection 2.1). Once the BGM's implementation is explained, we will be able to be more precise in the explanation of the process to generate it (Subsection 2.2).

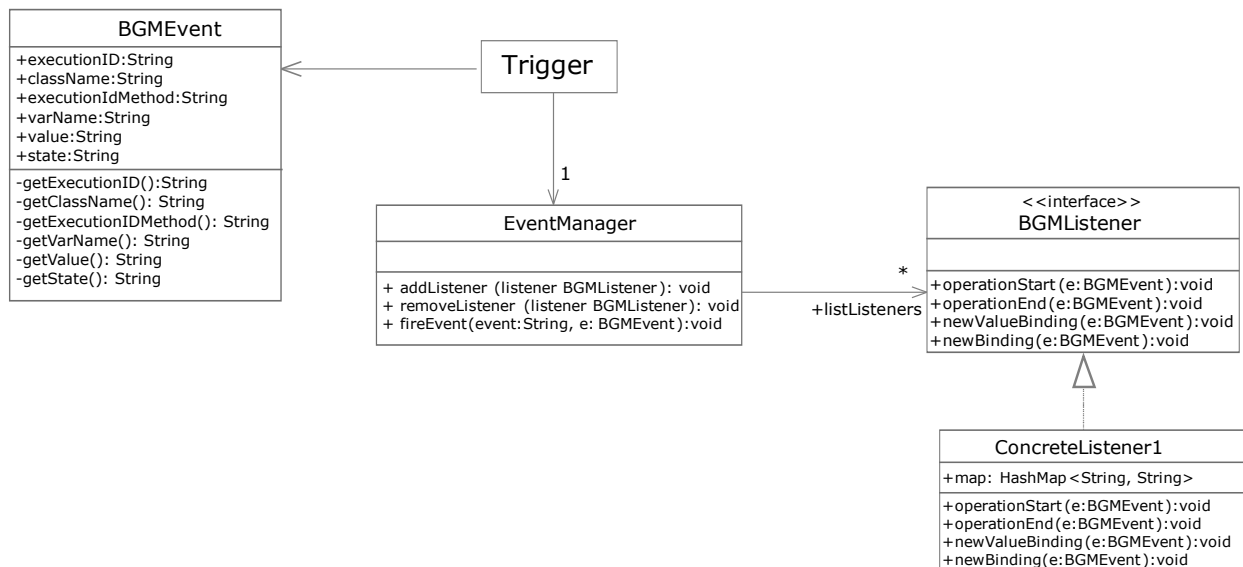


Figure 4: UML CD depicting the design of the BGM.

2.1 Towards the implementation of the BGM

In order to provide a reference implementation that fulfils the requirements presented in the paper (*R1-R5*), we have taken inspiration from the Event-driven Architecture mediator topology [6]. Briefly speaking, it consists of highly decoupled, single-purpose *event processing modules* that asynchronously receive and process *events*. An *event* is a notable occurrence that happens during the execution, it consists of a header (denoting the type of the event) and a body (containing information). The *event processing modules* are responsible for managing the information contained in the *events*.

For our purposes, we have used *events* to carry the provenance data collected during the operation's execution, and the *event processing modules* to implement the concrete strategy for managing and storing the collected provenance data. In this way, we decouple the collection of provenance from the management of it, which will allow developers to implement the persistence approach that best suits her needs.

Following is the description of each component conforming the BGM (see CD depicted in Figure 4): *BGMEvent*, *Trigger*, *EventManager*, and *ConcreteListener*, which implements the *BGMLListener* interface.

- *BGMEvent*. It represents the occurrence of an *event*. The values of the *BGMEvent*'s attributes are crucial for constructing bindings, they contain both the name of the variable (attribute *varName*) and its associate value (attribute *value*). We have identified four types of *events* for being of interest for the generation of bindings. The *events operationStart* and *operationEnd* are fired when an operation's execution is called and finished, respectively. Likewise, the *event newBinding* is fired when it is collected a provenance value linked with the identifier of a PROV element in a template (e.g., *var:operation*); and the *event newValueBinding*, when it 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 (e.g., *var:operationStartTime*).
- *Trigger*. The *event* flow starts when a *Trigger* identifies an *event* during the execution, and consequently, it sends an occurrence of this *event* (*BGMEvent*) to an *event mediator* (*EventManager*) for disseminating it among the *event processors* (*ConcreteListener*). To do this, it has to implement additional custom behaviour to collect data provenance for constructing objects of type *BGMEvent*. Likewise, it has to be able to identify the moments in which it has to construct the objects of type *BGMEvent*.

```
public aspect Trigger{  
    Object around(): initialization(<object>.new(..)) || call(* <object>.<operation>(..)){  
        behaviourBeforeExecution();  
        Object rtn = proceed();  
        behaviourAfterExecution();  
        return rtn;  
    }  
}
```

Figure 5: Structure overview of a reference implementation of the *Trigger* in AspectJ

In order to provide a fully automatic way for bindings generation, we propose to use the Aspect Oriented Programming (AOP) [2] paradigm. AOP aims at improving the modularity of software systems, by capturing inherently scattered functionality, often called *cross-cutting concerns* (thus, the construction and management of *BGMEvent* objects can be considered as a cross-cutting concern). Our reference implementation is developed in AspectJ [8] (an AOP extension created for Java), which consists of an AOP *aspect* (see Figure 5), which is made up of an *advice* with *pointcuts*. Whereas the *advice* contains the custom behaviour to be executed (*R2*), the *pointcuts* state the moments in which it has to execute the custom behaviour (*R3*). In the end, the AspectJ *weaver* automatically integrates the behaviour from the *aspects* into the locations specified by the *pointcuts* as a pre-compilation step. In this way, the AOP approach does not require a manual intervention for adapting the source code, and completely satisfies the requirement *R1*.

- *ConcreteListener1*. This class implements the *BGMLListener* interface. This implementation constitutes the mechanism used to manage the collected data provenance. Our reference implementation of this component directly generates the provenance ready to be used. While an operation is executed, we construct a set of bindings that is associated with PROV template(s). Once the operation's execution finishes, the expansion algorithm from Prov-Toolbox [5] instantiates the templates with the set of bindings, generating PROV documents to be exploited. These documents are available in a MongoDB database for provenance consumers. This strategy has disadvantages and advantages, it is clear that it requires extra time for creating and expanding the PROV templates; however, as a positive aspect, the generated provenance is ready to be exploited as the application is running. As opposed to this, another approach could devote less time during the application's execution by storing the data provenance as logs. However, the provenance consumer has to create the bindings and to expand the PROV templates for generating the final provenance.

It is worth remarking that there could be several implementations of the *BGMLListener* simultaneously, which enables several strategies for managing the collected data provenance at the same time. This fact allows developers to replicate data provenance in several repositories. For instance, there may be a repository with provenance data stored as logs, and another repository may contain the data provenance in the form of bindings.

- *EventManager*. It has two main responsibilities: to manage a list of subscribed *event processors* (objects implementing the interface *BGMLListener*), and to disseminate the received *events* (objects of type *BGMEvent*) among them. The former is addressed by an operation to subscribe an object implementing *BGMLListener* (*addListener*), and an operation to unsubscribe it (*removeListener*). The latter is tackled by the operation *fireEvent*. When this operation is invoked, for each subscribed *BGMLListener*, 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 the subscribers.

2.2 Reference implementation of the process for generating the BGM

This process performs one M2T transformation, referred to as T3 in Figure 6. 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 implementation of the BGM (see Figure 4).

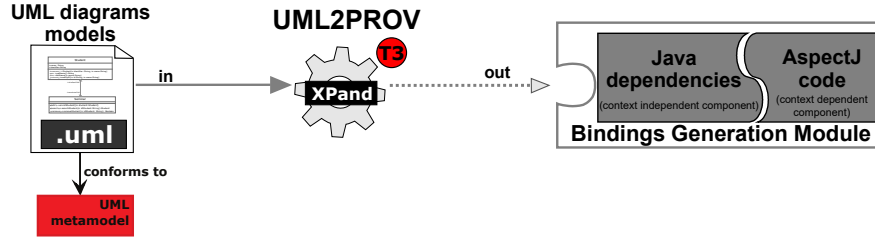


Figure 6: Detailed MDD-based implementation of the PROV template generation process

Going into the depths of this XPand module, it is important to remember that the components of the generated BGM are divided into two main groups. The first group is made up of those elements that do not depend on the source *UML diagram models*, and therefore, the source code is the same in all the BGMs. This group, which is referred to as *context independent components*, is made up of *BGMEvent*, *EventManager*, *BGMLListener*, and the concrete implementations of the *BGMLListener* (e.g., *ConcreteListener1*). The second group, called *context dependent components*, consists of those elements whose implementation depends on the source *UML diagram models*. The only element in this group is the *Trigger* (see its structure in Figure 5). The XPand module generates the implementation of the *Trigger* as an aspect in AspectJ with an *advice* of type *around* that has a list of *pointcuts*. Whereas the code inside the *around* is shared by all the BGM to be generated, the list of *pointcuts* is different depending on the source *UML diagrams models*. Concretely, it is created a *pointcut* for intercepting the calls of operations that are involved in the source SqDs (i.e., the operation whose calls are modelled by means of *Messages*), SMDs (i.e., the operation whose occurrences are associated with *Events*), and CDs, (i.e., the operations that are modelled by *Operations*).

References

- [1] ATL - a model transformation technology, version 3.8. Available at <http://www.eclipse.org/at1/>. 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 February 2019.
- [9] XPand. Eclipse platform. <https://wiki.eclipse.org/Xpand>, Last visited on September 2018.