

---

# IMPLEMENTATION DETAILS

---

Carlos Sáenz-Adán<sup>1\*</sup>, Beatriz Pérez<sup>1</sup>, Francisco J. García-Izquierdo<sup>1</sup>, Luc Moreau<sup>2</sup>

<sup>1</sup>*Dept. of Mathematics and Computer Science, Univ. of La Rioja, La Rioja, Spain,*

*{carlos.saenz,beatriz.perez,francisco.garcia}@unirioja.es*

<sup>2</sup>*Dept. of Informatics, King's College London, London, UK,*

*luc.moreau@kcl.ac.uk*

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

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

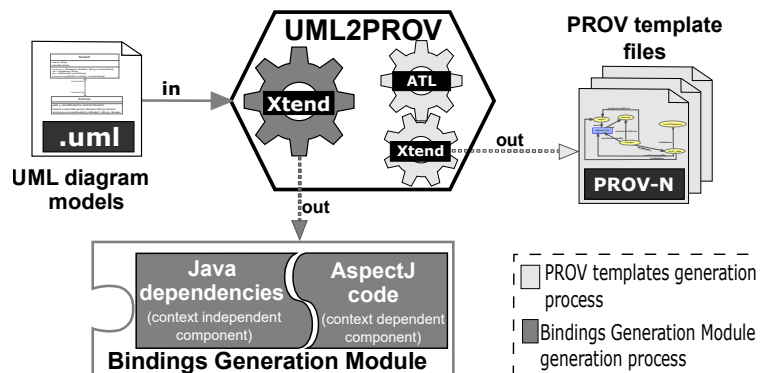


Figure 1: Our MDD-based implementation proposal.

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

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

## 1 Automatization of the UML to PROV Templates transformation patterns

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

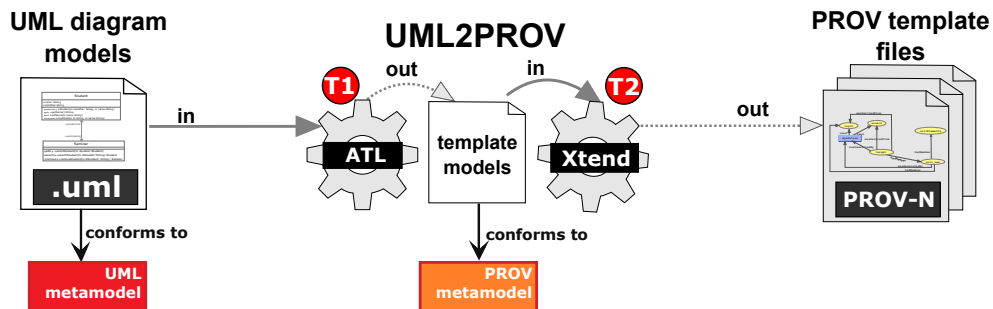





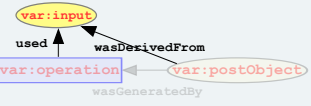
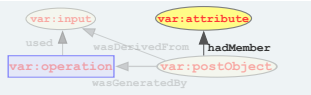
Figure 2: Detailed MDD-based implementation of the PROV templates generation process

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

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

As an example of such ATL rules, Table 1 shows how an excerpt of the ATL rule defined to implement the *CIP1* pattern looks like. Such a pattern deals with operations that construct an object. This table depicts, per each fragment of the

Table 1: An excerpt of the ATL rule implementing *CIP1*. For each fragment in the excerpt (“ATL source code” column), the PROV elements it generates are provided (“Template model” column) together with a description of the transformation (“Description” column) as well as the graphical notation of the template model (last column).

ATL source code	Template model	Description	Graphical representation of the template model
rule Operation2Document{ from operation: UMLIOperation( operation.hasStereotype('create')) [...]		It states that the rule is applied to all the UML operations to which CIP1 refers. That is, those operations with the stereotype «create»	
to postObjectEn: PROV!Entity ( id <- 'var:postObject', ...),	<entity id="var:postObject"/>	It creates an <entity> with identifier var:postObject.	
operationAct: PROV!Activity ( id <- 'var:operation', ...),	<activity id="var:operation"/>	This excerpt is in charge of generating an <activity> identified by var:operation.	
wgb: PROV!Generation ( entity <- postObjectEnID, activity <- operationActID),	<wasGeneratedBy> <entity ref="var:postObject"/> <activity ref="var:operation"/> </activity>	This excerpt is responsible for linking var:postObject with var:operation by means of the PROV relation <wasGeneratedBy>.	
do{ if(existIn){ thisModule.newEnt('var:input', doc); thisModule.genDer('var:postObject', 'var:input', doc); thisModule.genU('var:input', ' var:operation', doc); }	<entity id="var:input"/> <wasDerivedFrom> <generatedEntity ref="var:postObject"/> <usedEntity ref="var:input"/> </wasDerivedFrom> <used> <activity ref="var:operation"/> <entity ref="var:input"/> </used>	In case there are UML Input Parameters, it creates an <entity> identified by var:input*, and its relations <wasDerivedFrom> and <used> with var:postObject and var:operation, respectively.	
if(hasAttr){ thisModule.newEnt('var:attribute', doc); thisModule.genMe('var:attribute', 'var:postObject', doc); }	<entity id="var:attribute"/> <hadMember> <collection ref="var:postObject"/> <entity ref="var:attribute"/> </hadMember>	If there are UML Attributes in the class to which the operation belongs, it generates an <entity> identified by var:attribute*, and the PROV relation <hadMember> between var:postObject and var:attribute.	
[...]	</document>		

\*Although CIP1 states that each attribute/input parameter is a separate prov:Entity identified as var:attribute/var:input, we have decided to merge all the entities with the same identifier. Nevertheless, this decision does not have any effect on the bindings, since each var:input and var:attribute will be given several values (one for each input parameter and attribute, respectively).

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

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

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

```

1: class PROVNGenerator {
2:   def manageDocument(Document doc, PrintStream o) {
      o.println('
      document
      prefix prov <http://www.w3.org/ns/prov#>
      prefix tpl <http://openprovenance.org/tmpl#>
      prefix var <http://openprovenance.org/var#>
      prefix exe <http://example.org/>
      prefix u2p <http://uml2prov.org/>
      bundle exe:bundle1
      ')
3:   for (entity : doc.entity)    {o.println(manageEntity(entity))}
4:   for (agent : doc.agent)     {o.println(manageAgent(agent))}
5:   for (activity : doc.activity) {o.println(manageActivity(activity))}
6:   for (wsb : doc.wasStartedBy) {o.println(wStartedByTemplate(wsb))}
7:   for (wgb : doc.wasGeneratedBy) {o.println(wgbTemplate(wgb))}
8:   for (u : doc.used)          {o.println(usedTemplate(u))}
9:   for (wInfB : doc.wasInformedBy) {o.println(wInfByTemplate(wInfB))}
10:  for (wInvB : doc.wasInvalidatedBy) {o.println(wibTemplate(wInvB))}
11:  for (wdf : doc.wasDerivedFrom) {o.println(wdfTemplate(wdf))}
12:  for (hm : doc.hadMember)      {o.println(hmTemplate(hm))} //
13:  for (so : doc.specializationOf) {o.println(spOTemplate(so))} //
14:  for (wat : doc.wasAttributedTo) {o.println(watTemplate(wat))}
15:  for (waw : doc.wasAssociatedWith) {o.println(wawTemplate(waw))}
      o.println('
      endBundle
      endDocument');
      }
16:  def manageEntity(Entity entity) {
      ""entity(«entity.id», «entityAttributeTemplate(entity)»)""
      }
      ...
      }

```

Fixed text

Fixed text

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

## 2 BGM generation automation

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

### 2.1 Towards an implementation of the BGM

Aimed at providing an implementation of a BGM, there are several issues a developer may consider to manage the provenance data for creating bindings. The first one is referred to when and how the bindings are generated and stored. For example, applications may store the provenance data using usual logs, delaying the construction of bindings

until after runtime. Alternatively, applications could directly construct the bindings at runtime. The second aspect refers to when provenance documents are generated and which storage system is used. For example, the bindings could be accumulated locally in memory, delaying the generation of the provenance documents (i.e., the expansion of templates), and thus their storage (e.g., database, files, . . .) until after runtime. Alternatively, the strategy could be to expand the templates with the accumulated bindings on runtime, storing the provenance documents as the application is executed.

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

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

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

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

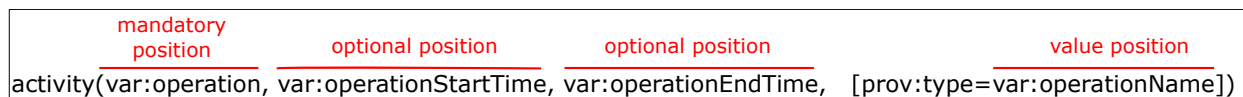


Figure 4: PROV activity in PROV-N [8] with different types of variables. Additionally, it is shown a table associating each variable with its type.

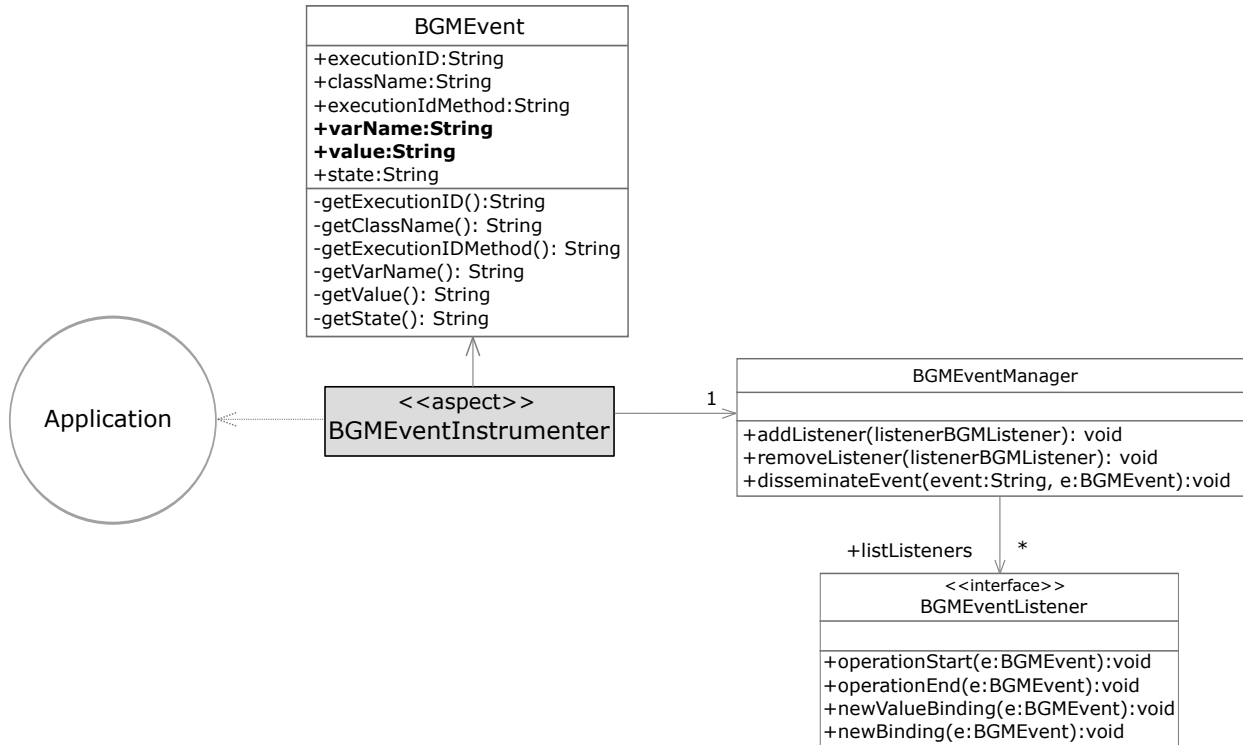


Figure 5: UML CD depicting our reference implementation for the BGM.

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

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

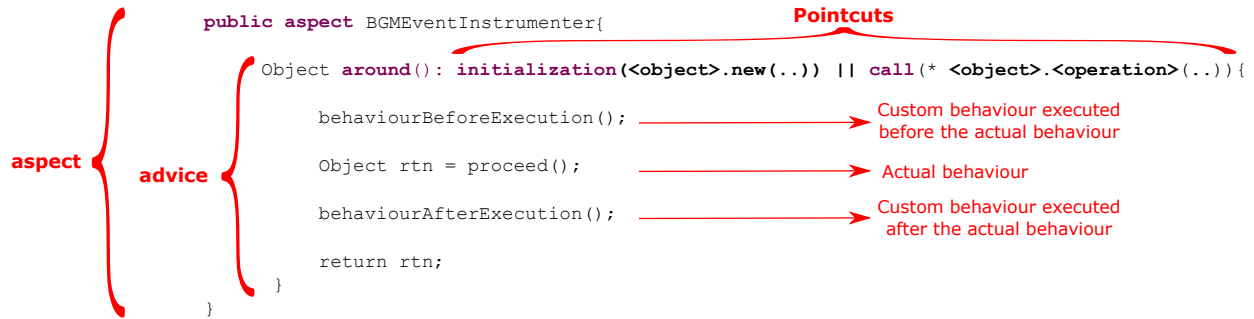


Figure 6: Structure overview of a reference implementation of the `BGMEventInstrumenter` in AspectJ

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

- `BGMEventManager`. In some cases, to have only one `listener` for generating, managing, and storing bindings could be not enough, and the same happens with the mechanisms for generating and storing provenance. For instance, one provenance consumer may be interested in replicating the information by storing both the provenance data, and the bindings generated from them in different storage systems. Aiming at addressing these scenarios, we have included the `BGMEventManager` with two responsibilities: to manage a list of subscribed `listeners`, and to disseminate the objects of type `BGMEvent` among them.
- `BGMEventInstrumenter`. As we stated in the paper [1], to manually adapt the source code of an application would be a valid option to capture provenance data. However, this option would require to traverse the whole code of the concrete application identifying the classes that will be the source of the events, and additionally, those places inside these classes where `events` will be fired. Then, the manual adaptation of the source code would need to include in those places instructions for constructing `BGMEvent` objects with the provenance data, and disseminating them among the `listeners`. This task constitutes a tedious, time-consuming and error-prone process. What is worse, the manual adaptation could incur in such provenance capture code instructions scattered across all the application classes, making their maintenance a cumbersome task.

In contrast, we propose to use the Aspect Oriented Programming (AOP) [10] paradigm for implementing what we have named `BGMEventInstrumenter`. AOP aims at improving the modularity of software systems, by capturing inherently scattered functionality, often called *cross-cutting concerns*, (e.g., the capture of provenance), and placing that functionality apart from the actual application's source code. Our reference implementation is developed in AspectJ, an AOP extension created for Java [11], and it consists of an `aspect` which is made up of an `advice` with `pointcuts` (see Figure 6). On the one hand, the `pointcuts` identify locations within the application code where a concern may be included. In our case, we identify operation calls and constructor invocations from which we want to fire `events` (i.e., to collect provenance). On the other hand, the `advice` is the behaviour executed when the `pointcuts` are matched. In AspectJ, `advices` can be executed at three different places: *before*, *around*, and *after* the `pointcuts`. Due to the fact that our identified `events` can occur both before and after operations calls and constructors

invocations, we have used an *around advice* for executing custom behaviours before and after the actual behaviour. These custom behaviours consist of constructing objects of type *BGMEvent* and disseminating them to the *listeners* (by invoking the `disseminateEvent` operation from *BGMEventManager*). In the end, as a pre-compilation step, the AspectJ *weaver* automatically integrates the behaviour from the *aspects* into the locations specified by the *pointcuts* at compilation time. In this way, our AOP approach does not require a manual intervention for adapting the source code, and automatically collects provenance data in a transparent way for software developers, which directly incurs in fulfilling the requirements *R1-R3* stated in the paper [1].

### 2.1.1 Example of a class implementing the *BGMEventListener*

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

## 2.2 Automatization of the implementation of the BGM

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

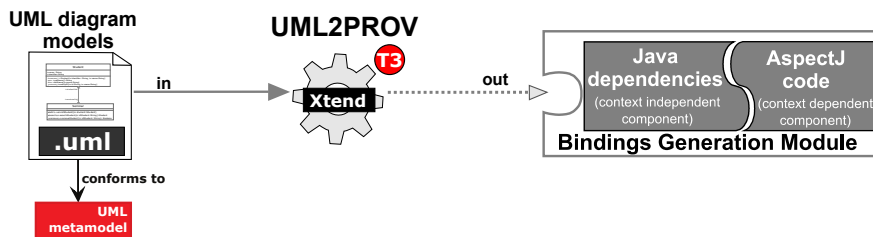


Figure 7: Detailed MDD-based implementation of the BGM for an application.

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

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



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

### 2.3 Fulfilment of BGM requirements

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

## References

- [1] C. Sáenz-Adán, B. Pérez, F. J. García-Izquierdo, and L. Moreau, “Integrating Provenance Capture and UML with UML2PROV: Principles and Experience,” submitted for publication in IEEE Transactions on Software Engineering.
- [2] B. Selic, “The pragmatics of model-driven development,” *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.
- [3] ATL - a model transformation technology, version 3.8. Available at <http://www.eclipse.org/at1/>. Last visited on January 2020.
- [4] Xtend, “General-purpose high-level programming language.” Available at <https://www.eclipse.org/xtend/>. Last visited on January 2020.
- [5] OMG, “Unified Modeling Language (UML). Version 2.5,” 2015. Document formal/15-03-01, March, 2015.
- [6] L. Moreau, P. Missier (eds.), K. Belhajjame, R. B’Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, J. McCusker, S. Miles, J. Myers, S. Sahoo, and C. Tilmes, “PROV-DM: The PROV Data Model,” W3C Recommendation REC-prov-dm-20130430, World Wide Web Consortium, 2013.
- [7] Supplementary material for the paper entitled “Supplementary material of Integrating Provenance Capture and UML with UML2PROV: Principles and Experience” containing the *Description of patterns*, submitted for publication in IEEE Transactions on Software Engineering. Available at <https://uml2prov.unirioja.es/>. Last visited on January 2020.

- [8] L. Moreau, P. Missier (eds.), J. Cheney, and S. Soiland-Reyes, “PROV-N: The Provenance Notation,” W3C Recommendation REC-prov-n-20130430, World Wide Web Consortium, Apr. 2013.
- [9] D. Michaelides, T. D. Huynh, and L. Moreau, “PROV-TEMPLATE: A Template System for PROV Documents,” 2014. Available at <https://provenance.ecs.soton.ac.uk/prov-template>. Last visited on January 2020.
- [10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *Proc. of the European Conference on Object-Oriented Programming (ECOOP 1997)*, (Berlin, Heidelberg), pp. 220–242, 1997.
- [11] The AspectJ Project. Available at [www.eclipse.org/aspectj/](http://www.eclipse.org/aspectj/). Last visited on January 2020.