PAMELA: an annotation-based Java Modeling Framework

Sylvain Guérin, Guillaume Polet, Caine Silva, Joel Champeau,
Jean-Christophe Bach, Salvador Martínez, Fabien Dagnat, Antoine Beugnard

PAMELA: an annotation-based Java Modeling Framework

Sylvain Guérin[a,*], Guillaume Polet[a], Caine Silva[a], Joel Champeau[a], Jean-Christophe Bach[b], Salvador Martínez[b], Fabien Dagnat[b], Antoine Beugnard[b]

[a] *ENSTA Bretagne, Lab-STICC, UMR 6285, Brest, France*
[b] *IMT Atlantique, Lab-STICC, UMR 6285, Brest, France*

Abstract

This article presents PAMELA, an annotation-based Java modeling framework. PAMELA provides a smooth integration between model and code and enables Java developers to handle software development both at conceptual level and at source-code level, without code transformation and/or generation, avoiding round-trip-related issues.

Keywords: Model Driven Engineering, Object Oriented Programming, Metaprogramming

1. Introduction

Model-Driven Software Development focuses on managing abstract models representing the conceptual level of the solution space. These models are generally represented as various artifacts, corresponding to different languages and using different representations. But models are rarely executable and/or lack the required level of expressiveness or performance to be exploited directly in production. Thus, the solution they offer is often implemented by using code generators that produce a representation of the solution in a target programming language or framework[22]. The underlying semantics of the code to be executed is generally encoded in those code generators and can be inlined or implicitly defined by the code generation process, or may sometimes be explicit to the code generation. Figure 1 shows the classical vision for Model-Driven Engineering.

This approach, model first, then generation, raises two major issues. First, it creates an important gap between the conceptual level (the model) and the source code, where semantics may be totally hidden or implicit. Second, the synchronization of the pair models-code in a co-evolution scenario where model

*Corresponding author

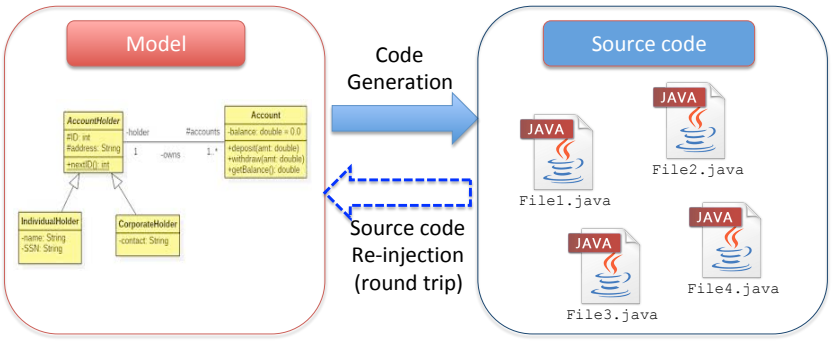

Figure 1: Classical vision for Model-Driven Engineering

and source code may evolve independently (and often by different actors, e.g., architect and developer) becomes hard to maintain.

Round-trip mechanisms are commonly used to overcome those difficulties, but they are difficult to use and to maintain. Indeed, the model/code co-evolution problem remains an open subject in the software engineering research community and, consequently, software development projects often abandon the idea of maintaining the synchronization between model and source code during development process. In that context, the model is developed in the early stages of development process, used mainly to prototype software applications. Then, it may be manually reviewed back at the end of development process, for documentation purposes.

Conversely, we propose a shift in the modeling paradigm in which models and code are developed together and at the same time in what we call a continuous modeling process. The PAMELA framework supports this paradigm shift by providing the means to: 1) weave model-based annotations with Java source code and 2) interpret model annotations at run-time.

The rest of the paper is organized as follows. Section 2 describes our approach and its associated development process. Section 3 presents the main building blocks of the PAMELA framework together with an illustrative example. Implementation details are discussed in Section 4, followed by a description of industrial experimentation and validation cases in Section 5. We end the paper in Section 6 by discussing related work.

2. Approach and Development Process

We advocate for a strong coupling between model and sources code, to give architects and developers a way to both interact during the whole development cycle.

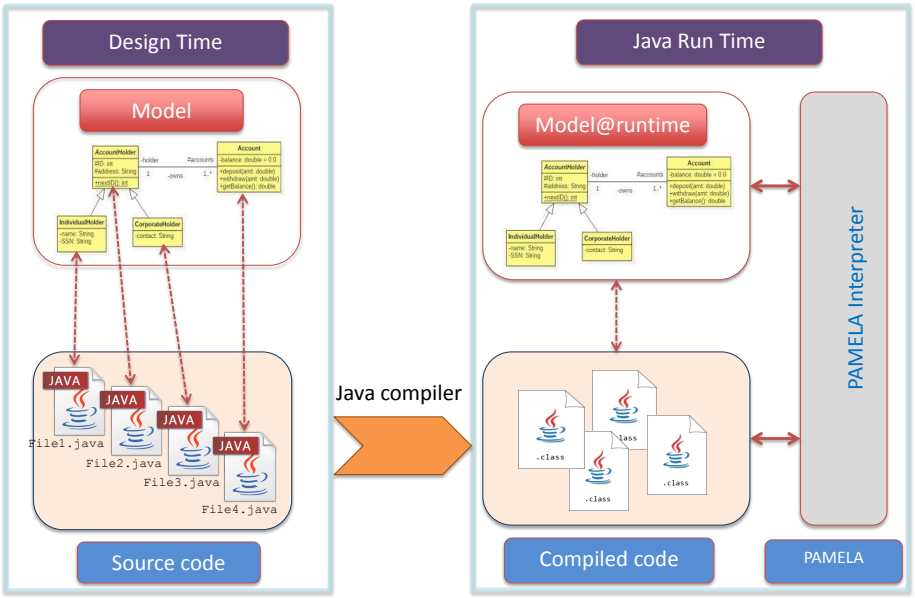

Figure 2: PAMELA approach for modeling

2.1. Architecture overview

PAMELA is an annotation-based Java modeling framework providing a smooth integration between model and code, without code generation nor externalized model serialization. Instead of generating the code, the API (mostly Java interfaces with abstract method declarations) is locally executed (interpreted). The idea is to avoid separation between modeling and code to facilitate consistency management and to avoid round-trip issues. Figure 2 summarizes the PAMELA architecture. The left side shows the structure of the application source code where java files contain the code and annotations that link part of the code to PAMELA model entities such as class, attribute, method, etc. The right side shows that at runtime the PAMELA interpreter maintains the relationships between the application binary, result of the java compilation, and the internal representation of PAMELA models. The preservation of the structure and of the whole information allow a very good and high level of control over the execution. In the following section, we present some indications on the different ways of programming using PAMELA.

2.2. Usage of PAMELA

Coupling model and code into the same artifact opens new ways of programming. The classical (metadata enabled) programming process relies on

programmers that produce code reusing pre-existing modeling concepts. These concepts are implemented by *modelers* that provide the right annotations the programmers use. This is, for instance, the process followed by Jakarta EE (JEE) developers reusing JEE specific annotations. The evolution rhythm between models and code is low. This programming way is still possible with PAMELA, but we allow the ability to reach a high evolution rhythm when the programmer also becomes the modeler. In fact, when the programmer identifies a pattern, an abstraction, a generalization, s/he can use PAMELA to develop and capitalize on this abstraction by extending PAMELA's metamodel.

The developed metamodels are implemented by annotations that rely on Java/JVM entities and mechanisms. They include consistency checking, which constrain their use and help the programmer to avoid inconsistencies or errors. We have first experimented their use with setters/getters to define Plain Old Java Objects (POJO), with traits to implement multiple inheritance or with roles and rules to set security rules on classes.

Our experience shows that introducing and reusing new concepts (1) reduce the size of the code (2) reduce the risk of errors and (3) improve the code structure. The cycle of development between the model and the code can then be drastically reduced, leading to what we call *continuous modeling*.

The code size is reduced because abstractions factorize recognized concepts so that the code using such concepts is replaced by the use of the abstraction at the right place. This also reduces the risk of errors since the code is now managed by the PAMELA framework with all the required checks. Finally, the code structure is improved since it matches the way the programmer conceptualizes (models) her/his code.

Here are various conceivable scenarios for PAMELA use:

- **Programming use** : programmers reuse existing annotations and write model and code at the same time (the modelers and programmers share the same artifacts : the Java code). This scenario includes the case where the model (made by the modelers) is pre-existing.
- **Reengineering** : programmers start from an existing code base (legacy) and refactor it while replacing this code by abstract method declarations in Java interface, reducing risks of errors.
- **Aspect-oriented programming** : programmers may use or redefine "patterns" (e.g., Security Patterns [20]) which offers code weaving at runtime, and runtime monitoring.
- **Advanced programmers use** : programmers may extend PAMELA with their own annotations, implementations or patterns.

In this presentation, we focus on the programming use of PAMELA, when programmers reuse existing annotations. The full power of PAMELA arises when programmers become modelers defining their own abstractions/annotations.

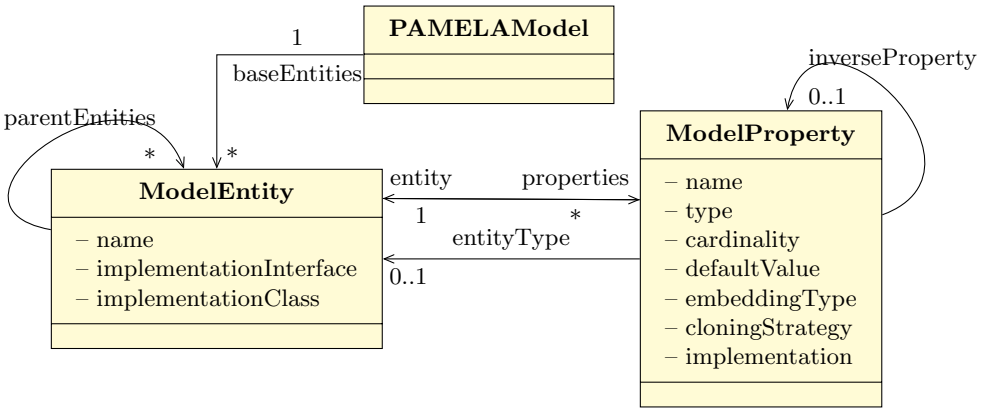

Figure 3: PAMELA metamodel

3. Software Framework and Features

PAMELA is composed of one design time component (the PAMELA metamodel, plus a number of predefined annotations) presented in the next section and one runtime component (the PAMELA interpreter), described in Section 3.2.

3.1. Design time

The PAMELA metamodel is depicted in Figure 3, and allows, through the use of annotations, the definition of *simple* UML-like models directly on the code.

A `PAMELAModel` is defined as a set of references to `ModelEntity`. A `ModelEntity` reflects a concept and is encoded in a Java `interface`. Then, a `ModelEntity` only defines an API (Application Programming Interface) without any implementation for methods. Note that *ModelEntities* may declare *ImplementationClasses*, which will be responsible for providing custom code implementing domain or application-specific behavior. A partially implemented `abstract` Java `class` may be defined as partial base implementation (conforms to implemented interface), as default methods in Java interfaces would be (since Java 8). The PAMELA metamodel allows multiple inheritance: thus `ModelEntity` may define a set of parent entities.

A `ModelEntity` also defines some properties, encoded as `ModelProperty`. A `ModelProperty` is identified by a name, a cardinality (simple or multiple) and a type, which can be a reference to another `ModelEntity`, or a Java type (a primitive or an arbitrary complex Java type). Depending on its cardinality, a `ModelProperty` is bound to a set of methods reflecting use of property.

- A *read-only single property* will define read access of its value using a *getter* (a Java method defined in Java interface taking no argument and returning the desired value).

- A *read-write single property* will define a *getter* and a *setter* (a Java method taking the value to be set as unique argument).
- A *read-write multiple property* will define a *getter*, an *adder* (a Java method taking the value to be added as unique argument), a *remover* (a Java method taking the value to be removed as unique argument), and may define additional methods for extended features such as re-indexing, for example.

In the following we list the most common Java annotations used in the context of `ModelEntity` and `ModelProperty` definitions:

- `@ModelEntity`: tag annotating `interface` as *ModelEntity*. May also declare an abstract entity.
- `@ImplementationClass`: tag annotating *ModelEntity* `interface` and precising abstract Java `class` to be used as base implementation.
- `@Implementation`: tag annotating a partial implementation (abstract inner `class` defined in implemented `interface`), and used in the context of multiple inheritance.
- `@Getter(String)`: tag annotating method as unique getter for implicit *ModelProperty* whose identifier is the declared String value. May also declares cardinality, eventual inverse property, default value and some other features.
- `@Setter(String)`: tag annotating method as unique setter for implicit *ModelProperty* whose identifier is the declared String value.
- `@Adder(String)`: tag annotating method as unique adder for implicit multiple cardinality *ModelProperty* whose identifier is the declared String value.
- `@Remover(String)`: tag annotating method as unique remover for implicit multiple cardinality *ModelProperty* whose identifier is the declared String value.
- `@Reindexer(String)`: tag annotating method as unique re-indexer for implicit multiple cardinality *ModelProperty* whose identifier is the declared String value.
- `@Initializer`: tag annotating a method used as a constructor for related *ModelEntity*.
- `@Deleter`: tag annotating a method used as explicit destructor for related *ModelEntity*.
- `@Finder(String,String)`: tag annotating method as a fetching request for a given *ModelProperty* with a given attribute.

- `@CloningStrategy`: allows to customize cloning strategy for a given *ModelProperty*.
- `@Embedded`: allows to declare a given *ModelProperty* as contained.
- `@Imports` and `@Imports`: allows to declare some entities to be included in PAMELA model.
- `@XMLElement` and `@XMLAttribute`: used to specify XML serialization for PAMELA instances.

3.2. Runtime

The aforementioned models are executed at runtime as a combination of two components as illustrated by the right part of Figure 2: 1) plain Java bytecode, as the result of the basic compilation of source code; and 2) an embedded PAMELA interpreter, executing semantics reflected by *ModelEntity* and *ModelProperty* declarations (together with custom annotations where available).

The main idea for the approach is to override Java dynamic binding. Invoking a method on an object which is part of a PAMELA model, causes the real implementation to be called when existing (more precisely dispatch code execution between all provided implementations), or the required interpretation according to the underlying model to be executed.

The PAMELA interpreter will intercept any method call for all instances of *ModelEntity* and conditionally branches code execution.

- If the accessed method is part of a *ModelProperty* (a getter, or a setter, etc.), and no custom implementation is defined neither in the class declared as implementation, nor in a class declared as partial implementation in the context of traits, then, execution is delegated to the related property implementation (generic code provided by the PAMELA interpreter).
- If the accessed method is defined in a class declared as implementation, or in a class declared as partial implementation, then this method is executed. The PAMELA API through the `AccessibleProxyObject` interface also provides access to generic behavior (super implementation), allowing the developer to define an overriding composition.

This general scheme also provides an extension point allowing to instrument the code. This extension point is used in order to integrate other features such as notification management, undo/redo stack management, assertion checking at runtime (support for *Design by Contract*, aka JML), and dynamic code weaving in the context of *Aspect Programming*.

The PAMELA model at runtime is computed dynamically, working on the classpath of the launched Java application, and starting from a simple Java interface (or a collection of Java interfaces) which is (are) PAMELA-annotated. From a mathematical point of view, internal representation of the underlying model is a graph whose vertices are PAMELA *ModelEntities* (annotated Java interface), and edges are either inheritance links or reference links (a property

whose type is another *ModelEntity*). `@Imports` and `@Import` annotations allow to include some other *ModelEntities* in the model. An annotation attribute `@Getter(...ignoreType=true)` allows to ignore the link. In that context, PAMELA model computation is a graph closure computation, starting from a collection of vertices.

A PAMELA model at runtime is represented by a `ModelContext`.

PAMELA instances (instances of *ModelEntity*) are handled through the use of `ModelFactory`, which is instantiated from a `ModelContext`.

3.3. Additional Features

Programmers may already reuse a number of advanced programming features (some already mentioned) such as: multiple inheritance and traits, containment management, cloning, fine-grained notification management, object graph comparison and diff/merge, visiting patterns, clipboard management, validation, support for *design by contract* by integrating assertions from Java Modeling Language (JML), Metaprogramming, Aspect Oriented Programming. Each of those features is described on a dedicated page that is reachable from the official web site[1]. Experienced PAMELA programmers may extend PAMELA defining new annotations.

3.4. Example

Listing 1 shows a very basic model with two entities: *Book* and *Library*. Entity *Book* defines two read-write single properties (*title* and *ISBN*) with single cardinality and with `String` type. Entity *Book* also defines a constructor with initial *title* value. Entity *Library* defines a read-write multiple properties *books* referencing *Book* instances. Note that this code is sufficient to execute the model, while no additional line of code is required (only Java interfaces and API methods are declared here).

```
1  @ModelEntity
2  public interface Book extends AccessibleProxyObject {
3
4    @Initializer
5    public Book init(@Parameter("title")String aTitle);
6
7    @Getter("title")
8    public String getTitle();
9
10   @Setter("title")
11   public void setTitle(String aTitle);
12
13   @Getter("ISBN")
14   public String getISBN();
15
16   @Setter("ISBN")
17   public void setISBN(String value);
18 }
```

[1] `https://pamela.openflexo.org`

```
19
20 @ModelEntity
21 public interface Library extends AccessibleProxyObject {
22
23   @Getter(value = "books", cardinality = Cardinality.LIST)
24   public List<Book> getBooks();
25
26   @Adder("books")
27   public void addToBooks(Book aBook);
28
29   @Remover("books")
30   public void removeFromBooks(Book aBook);
31
32   @Reindexer("books")
33   public void moveBookToIndex(Book aBook, int index);
34
35   @Finder(collection = "books", attribute = "title")
36   public Book getBook(String title);
37 }
```

Listing 1: Model creation

The execution and the management of this model may be performed using the following simple lines of code:

```
1 // Instantiate the meta-model
2 // by computing the closure of concepts graph
3 ModelContext modelContext
4     = ModelContextLibrary.getModelContext(Library.class);
5 // Instantiate the factory
6 ModelFactory factory  = new ModelFactory(modelContext);
7 // Instantiate a Library
8 Library myLibrary = factory.newInstance(Library.class);
9 // Instantiate some Books
10 Book myFirstBook
11     = factory.newInstance(Book.class, "Lord of the rings");
12 Book anOtherBook = factory.newInstance(Book.class, "Holy bible");
13 myLibrary.addToBooks(myFirstBook);
14 myLibrary.addToBooks(anOtherBook);
```

Listing 2: Model execution/manipulation

The lines 3–4 instantiate a `ModelContext` by introspecting and computing the closure of concepts graph obtained while starting from `Library` entity and following `parentEntities` and `properties` relationships. This call builds at runtime a *PAMELAModel*, while dynamically following links reflected by compiled bytecode. A factory `ModelFactory` is then instantiated using that `ModelContext`, allowing to create *Library* and *Book* instances.

Custom code can be easily added to this model as we show in Listing 3. It shows how to integrate custom code to the fully interpreted *Book* entity described above. The partial custom implementation is offered by a partial class (note the `abstract` keyword), declared in the annotation header of model entity. Custom implementations are defined using classical Java implementation/override schemes. Here we define the implementation of the `read()` method, which has no annotation (and thus, cannot be processed by the PAMELA framework), and also the implementation of a custom getter for *title*, returning a default

value when no value is defined for that property. Note that this implementation references a default interpreted implementation (call to `performSuperGetter(String)` method).

```
@ModelEntity
@ImplementationClass(BookImpl.class)
public interface Book extends AccessibleProxyObject {

  static final String TITLE = "title";

  @Getter(TITLE)
  String getTitle();
  // ... title property declarations ...

  void read();
}

// Provides a partial implementation for Book
public static abstract class BookImpl implements Book {

  @Override
  public String getTitle() {
    String title = performSuperGetter(TITLE);
    if (title == null) {
      return "This book has no title";
    }
    return title;
  }

  @Override
    public void read() {
      // do the job
    }
}
```

Listing 3: Custom Code

4. Implementation

The PAMELA implementation is available online[2].

4.1. Exposed API at design time

The model-code integration we advocate requires facilities to encode metadata in the source code. This requires an annotation-enabled language. Such a language supports the attribute-oriented programming if its grammar allows adding custom declarative tags to annotate standard program elements. The Java programming language is a good candidate, as it supports annotations.

The PAMELA API exposed to the developer mainly consists of: 1) a set of annotations; and 2) a set of unimplemented Java interfaces exposing required features.

[2] `https://github.com/openflexo-team/pamela`

The `org.openflexo.pamela.annotations` package exposes the set of annotations which were presented in Subsection 3.1.

The `org.openflexo.pamela` package contains the following feature-related Java interfaces:

- `AccessibleProxyObject` is the interface that PAMELA objects should extend in order to benefit from base features such as generic default implementation, containment management, notification, object graph comparison and diff/merge, visiting patterns, etc.
- `CloneableProxyObject` exposes features related to cloning.
- `DeletableProxyObject` exposes features related to deletion management.
- `SpecifiableProxyObject` exposes dynamic assertion checking features in the context of JML (contract management) use.

4.2. PAMELA interpreter

The package `org.openflexo.pamela.factory` contains the PAMELA interpreter implementation. The core of the interpreter is implemented in the class `ProxyMethodHandler`.

From a technical point of view, the PAMELA implementation uses the *javassist* reflection library (see [9]) which provides the `MethodHandler` mechanism, which is a way to override the Java dynamic binding. Invoking a method on an object which is part of a PAMELA model, causes the real implementation to be called when existing (more precisely dispatch code execution between all provided implementations), or the required interpretation according to the underlying model to be executed. This also provides an extension point allowing to instrument the code, which is used for other features such as undo/redo stack management, and assertion checking at runtime (support for Design by Contract, aka JML).

The PAMELA framework is a 100% pure Java ($\geq$1.8), compilable by a classical Java compiler and executable in a classical Java Virtual Machine.

4.3. PAMELA code base metrics

The PAMELA implementation is modularized.

- Core implementation (`pamela-core`) provides all base features. It contains 20k lines of code involving 184 classes. This code is covered with unit tests (6k lines of code, 112 classes). Reached code coverage is about 66%.
- `pamela-security-patterns` is an add-on library containing some security-pattern implementations.
- `pamela-perf-tests` contains benchmarking tools whose purpose is to quantify PAMELA performances.

4.4. Performance analysis

Compared to a basic POJO implementation, the use of PAMELA implies a CPU and memory footprint overhead. This is due to the partially interpreted nature of some of its features. We have developed a performance workbench in order to measure it [3].

We first defined a base model composed of four entities declaring four to five properties (single and multi-valued) each. From this simple model, two Java implementations have been derived using code generation. The first implementation uses fully implemented Java classes (POJO), while the second one is fully interpreted (it uses the PAMELA framework and defines only interfaces and API methods). Then, for each implementation, we have instantiated the base model (by creating 1010101 objects with their properties initialized to default values) and carried out performance measures. As expected, we measured a CPU overhead for the fully interpreted implementation (from $\times 20$). This is due to `MethodHandler` mechanism which is fully interpreted and contains many hooks. Therefore, it cannot be compared to a fully compiled and optimized Java dynamic binding. We also measured a memory footprint overhead, but of a less important weight that the CPU overhead ($\times 5$ factor).

Note, however that the measured overhead only applies to a very small portion of the code (mostly property accessors methods). A "real-world" model implementation generally involves a bigger "business code" part, which is generally the first CPU-time consumer. Using PAMELA won't increase CPU use on "business code" if this code is implemented with plain Java code (which is generally the case). We have used the Yourkit Java Profiler[4] on PAMELA based applications and, as expected, the time passed on accessors was negligible compared to the time used on methods implementing business logic. As we will see in the next section, PAMELA has been successfully applied to many industrial projects, with no performance issues reported. Moreover, the PAMELA implementation offers many functional features which are not present in the base implementation (such as undo/redo, clipboard management, runtime monitoring and weaving, etc.) which outweigh the performance overhead.

5. PAMELA industrial use cases and experiments

The PAMELA framework has been successfully applied in a variety of complex programming and modeling scenarios and we continue to use it daily as part of our modeling toolbox. In the following, we describe three important use cases in which PAMELA was a core component.

5.1. Openflexo infrastructure

Model Federation [11] is an approach that provides the means to integrate multiple models conforming to different paradigms, and giving to each stake-

[3] `https://github.com/openflexo-team/pamela/tree/1.6.1/pamela-perf-tests`
[4] https://www.yourkit.com/

holder a specific view adapted to its needs. Model federation approach is developed as a possible response to SIMF RFP (Semantic Information Modeling for Federation) [16] by OMG (Object Modeling Group). This RFP (Request For Proposal) requests submissions for a standard addressing "federation of information across different representations, levels of abstraction, communities, organizations, viewpoints, and authorities". Thus model federation allows the integration of heterogeneous models to develop new cross-concern viewpoints/-models or to synchronize the models used for designing a system.

Openflexo[24] is a software infrastructure providing support for model federation across multiple technological spaces. Conceptualization is addressed through the proposition of a language called FML (Flexo Modeling Language), which is executable on the platform. Openflexo infrastructure introduces connectors (also called Technology Adapters) to support various technological spaces and paradigms.

This open source initiative is now mature at the infrastructure level, and many projects and applications have been developed and powered using Openflexo infrastructure. More than 15 technology adapters have been developed, with various maturity stage regarding their industrialization (Microsoft Word, Excel and PowerPoint, EMF, OWL, Diagraming, JDBC, XML, OSLC, etc.)

The full Openflexo infrastructure is composed of about 50 components. In most components, the PAMELA framework is largely used. As an example the `diana` component[5] (a component providing diagraming features) is composed of 998 classes. 159 of those classes (mostly the diagraming model) are defined as Pamela `ModelEntity`.

The total base of code for Openflexo infrastructure represents around 900 000 lines of Java code. Regarding backend modeling, the PAMELA framework is extensively used in Openflexo core as well as in most technology adapters, with very few specific implementations for properties. The observer/observable pattern is generally used for graphical user interfaces, which also rely on the PAMELA framework.

An interesting experiment has been done in the context of Openflexo development process. When PAMELA was integrated to the code base, a big portion of the former legacy code has gradually and iteratively been migrated to PAMELA. Refactoring mainly consisted in removing code, and replacing method implementation by API method declaration. In some parts of core model implementation, code has been reduced by 80% (in terms of lines of Java code), and many bugs disappeared, as they were caused by programming errors.

The PAMELA implementation is now really stable and mature. According to Openflexo infrastructure developers, maintenance of code base (about 10 years of development) raises no PAMELA-specific issue, while co-evolution of models and code is greatly improved compared to a 'code generation'-based solution.

[5] `https://diana.openflexo.org`

5.2. Formose project

The Formose ANR (French National Agency for Research) project (ANR-14-CE28-0009)[23] aimed to design a formally-grounded, model-based requirements engineering (RE) method for critical complex systems, supported by an open-source environment. The main partners were: ClearSy, LACL, Institut Mines-Telecom, OpenFlexo, and THALES.

The main results of the project are a requirements modeling multi-view language, its associated design process and the development of an open-source platform called Formod[2], built using Openflexo infrastructure and the PAMELA framework. The requirements modeling language is based on KAOS [25] for goal modeling and SysML for the structural part of a system. The associated domain modeling language, used to describe system domain knowledge, extends the two ontology languages OWL [19] and PLIB [17]. The graphical notations are then translated into Event-B [1], a formal specification method supported by verification tools.

The Formose method and Formod tool have been evaluated on different case studies provided by the industrial partners of the project.

5.3. SecurityPatterns experiment

A significant experiment in PAMELA is the implementation of security patterns weaved on domain code [20]. In this context, the PAMELA framework is extended to include the notion of Pattern, i.e. a composition of multiple classes. Included to this experiment, the security pattern is specified by expected behavior defined and formalized by a pattern contract. This contract is defined by formal properties and the PAMELA framework ensures the property verification at runtime.

Related to the security pattern implementation, PAMELA enables the definition of additional security behavior to existing Java code. Patterns are defined in PAMELA using three classes, each one representing a different conceptual level `PatternFactory`, `PatternDefinition`, `PatternInstance`.

To declare a Pattern on existing code, pattern elements such as Pattern Stakeholders and methods need to be annotated with provided security pattern-specific annotations. These annotations will be discovered at runtime by the `PatternFactory` and stored in `PatternDefinition` attributes.

Summarizing, implementing Patterns with PAMELA provides the ability to monitor the execution of the application code; the ability to offer extra structural and behavioral features, executed by the PAMELA interpreter; a representation of Patterns as stateful objects. Such objects can then evolve throughout runtime and compute assertions.

6. Related Work

The PAMELA framework can be seen as a CASE tool that focuses on the design and verification and validation phases of the software development lifecycle. In that sense, it presents similarities to other existing object oriented

CASE tools and/or language workbenches such as Kermeta [12] and the Platypus [18] meta-case tool as they permit the definition and manipulation of models, metamodels and constraints/behavior. Two main features separate PAMELA from those approaches though: 1) The main artifact for the PAMELA framework is standard java code; and 2) PAMELA does not rely on code generation but on the interpretation/construction of models at runtime guided by code annotations. This allows for the seamlessly blending of the metamodeling/programming phases.

When the observed artifact is code (e.g., for runtime analysis or verification of properties) we could see our approach as similar to some contributions appertaining to the models@runtime topic [3]. Indeed models@runtime approaches often rely on the use of the reflection architectural pattern [6] in order to separate core application logic from a metalevel that contains information about properties, types, etc. This is also the case of PAMELA. Examples of such approaches are: AC-Contrat [15] which provides runtime verification of properties for context-aware applications; Ramses [8] focused on dynamic adaptations; and [10] that performs feature analysis. [21] wraps running systems in standard UML-like models in order to perform analysis and management tasks by using off-the-shelf MDE tools and techniques whereas FAME [13] is a polyglot library that keeps metamodels accessible and adaptable at runtime (Synchronization of changes between models and code are, however, limited for languages such as Java).

Different to the aforementioned approaches, the focus of PAMELA is not on adapting or observing ever running programs (although it can be used for the verification of runtime properties as demonstrated in [20]) but on: 1) providing a mechanism to blend coding and metamodeling so that the code and the metamodel may be built incrementally without the need for code generation and thus, avoiding round-tripping issues; and 2) providing default implementations for frequently used abstractions in order to ease development. In that sense PAMELA can be seen as a mix between classical CASE tools and the aforementioned models@runtime approaches.

More similar to PAMELA (as it focus on avoiding round-tripping issues) in [7] the authors present an approach to keep the (bidirectional) synchronization between feature-based models and generated code alternatives. They rely on Pharo[4] reflective capabilities. More recently, in [5] the authors construct and maintain at runtime model-based views on the data manipulated in object-oriented code.

In a different approach, UMPLE[14] mixes programming and modeling by integrating UML constructs into languages such as Java. However, they use code generation for the runtime part of the system.

7. Conclusion

The PAMELA framework promotes a modeling paradigm where models and code are jointly developed to provide a continuum between model and source

code. The support is supplied by Java annotations and the PAMELA interpreter at runtime.

The different experiments provide efficient examples to argue the benefits of the PAMELA framework but it is better to make your own experiences through `https://pamela.openflexo.org/`.

References

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press.

[2] IMT Atlantique. 2020. *Formod prototype*. ENSTA Bretagne, IMT Atlantique and Lab-STICC. `http://formose.lacl.fr/deliverable-d22c/`

[3] Nelly Bencomo, Sebastian Götz, and Hui Song. 2019. Models@ run. time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling* 18, 5 (2019), 3049–3082.

[4] Andrew P. Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. 2010. *Pharo by example*. Lulu.com.

[5] Artur Boronat. 2019. Code-First Model-Driven Engineering: On the Agile Adoption of MDE Tooling. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 874–886. `https://doi.org/10.1109/ASE.2019.00086`

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 2008. *Pattern-Oriented Software Architecture: A System of Patterns, Volume 1*. Vol. 1. John Wiley & Sons.

[7] Glenn Cavarlé, Alain Plantec, Steven Costiou, and Vincent Ribaud. 2016. Dynamic Round-Trip Engineering in the Context of FOMDD. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies* (Prague, Czech Republic) *(IWST'16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 7 pages. `https://doi.org/10.1145/2991041.2991056`

[8] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. 2004. RAMSES: a reflective middleware for software evolution. In *Proceedings of the 1st ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04), in 18th European Conference on Object-Oriented Programming (ECOOP'04)*. Citeseer, 21–26.

[9] Shigeru Chiba. 2000. Load-Time Structural Reflection in Java. In *ECOOP 2000 — Object-Oriented Programming*, Elisa Bertino (Ed.). Springer, Berlin, Heidelberg, Germany, 313–336.

[10] Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. 2010. Modeling features at runtime. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, Berlin, Heidelberg, Germany, 138–152.

[11] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. 2016. Addressing Modularity for Heterogeneous Multi-Model Systems Using Model Federation. In *Companion Proceedings of the 15th International Conference on Modularity* (Málaga, Spain) *(MODULARITY Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 206–211. https://doi.org/10.1145/2892664.2892701

[12] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. 2009. Model driven language engineering with kermeta. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, Berlin, Heidelberg, 201–221.

[13] Adrian Kuhn and Toon Wim Jan Verwaest. 2008. FAME, a polyglot library for metamodeling at runtime. *3rd International Workshop on Models@Runtime* (2008), 10.

[14] Timothy C. Lethbridge, Vahdat Abdelzad, Mahmoud Husseini Orabi, Ahmed Husseini Orabi, and Opeyemi Adesina. 2016. Merging modeling and programming using Umple. In *International Symposium on Leveraging Applications of Formal Methods*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, Springer International Publishing, Cham, 187–197.

[15] Marina Mongiello, Patrizio Pelliccione, and Massimo Sciancalepore. 2015. AC-contract: Run-time verification of context-aware applications. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, IEEE, 24–34.

[16] OMG. 2012. Semantic Information Modeling for Federation (SIMF) Request for Proposal. OMG Document. https://www.omg.org/cgi-bin/doc.cgi?ad/2011-12-10

[17] Guy Pierra. 2004. The PLIB ontology-based approach to data integration. In *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France (IFIP)*, René Jacquart (Ed.), Vol. 156. Springer US, Boston, MA, 13–18. https://doi.org/10.1007/978-1-4020-8157-6_2

[18] Alain Plantec and Vincent Ribaud. 2006. PLATYPUS: A STEP-based integration framework. In *IDIMT 2006*. 261–274.

[19] Kunal Sengupta and Pascal Hitzler. 2014. Web Ontology Language (OWL). In *Encyclopedia of Social Network Analysis and Mining*. Springer New York, New York, NY, 2374–2378. https://doi.org/10.1007/978-1-4614-6170-8_113

[20] Caine Silva, Sylvain Guérin, Raul Mazo, and Joel Champeau. 2020. Contract-based design patterns: A Design by Contract Approach to Specify Security Patterns. In *Proceedings of the 15th International Conference on Availability, Reliability and Security* (Virtual Event, Ireland) *(ARES '20)*, Melanie Volkamer (Ed.). Association for Computing Machinery, New York, NY, USA, Article 66, 9 pages. `https://doi.org/10.1145/3407023.3409185`

[21] Hui Song, Gang Huang, Franck Chauvel, and Yanshun Sun. 2010. Applying MDE tools at runtime: experiments upon runtime models. In *Proceedings of the 5th International Workshop on Models at Run Time*, Nelly Becomo, Gordon Blair, and Franck Fleurey (Eds.). Oslo, Norway, 15. `https://hal.inria.fr/inria-00560785`

[22] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. 2006. *Model-driven software development: technology, engineering, management.* John Wiley & Sons, Inc., Chichester.

[23] Formose team. 2020. *Formose ANR project.* ENSTA Bretagne, IMT Atlantique and Lab-STICC. `http://formose.lacl.fr/`

[24] Openflexo team. 2020. *Openflexo infrastructure.* ENSTA Bretagne, IMT Atlantique and Lab-STICC. `https://www.openflexo.org`

[25] Axel van Lamsweerde. 2009. *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley.