



HAL
open science

Monitoring Association Constraints in Model-Oriented Programming

Sylvain Guérin, Joël Champeau, Antoine Beugnard, Salvador Martínez

► **To cite this version:**

Sylvain Guérin, Joël Champeau, Antoine Beugnard, Salvador Martínez. Monitoring Association Constraints in Model-Oriented Programming. MODELS-C 2023: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, Oct 2023, Vasteras, Sweden. 10.1109/MODELS-C59198.2023.00068 . hal-04240673

HAL Id: hal-04240673

<https://hal.science/hal-04240673>

Submitted on 13 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Monitoring Association Constraints in Model-Oriented Programming

Sylvain Guérin, Joel Champeau

ENSTA Bretagne

Lab-STICC, UMR 6285

Brest, France

{sylvain.guerin,joel.champeau}@ensta-bretagne.fr

Antoine Beugnard, Salvador Martínez

IMT Atlantique

Lab-STICC, UMR 6285

Brest, France

{antoine.beugnard,salvador.martinez}@imt-atlantique.fr

Abstract—Associations are a key concept in modeling languages as a way to formalize the relationships between domain concepts. Unfortunately, the support of semantically rich associations able to represent complex relationships is often missing, and this is both at the model and code level. At the model level, complex constraints on associations are often represented by using external, textual constraint languages which are difficult to understand and to maintain. At the code level, the situation is even worse, as mainstream object-oriented languages lack direct support for associations.

In order to alleviate this problem, in this paper we propose a reification of complex association constraints so that they can be easily specified at development time and monitored at runtime. We do this by leveraging PAMELA, an annotation-based Java modeling framework, which promotes blending classical programming with modeling through the use of annotations and runtime code instrumentalization and monitoring. PAMELA is in the scope of Model-Oriented Programming approaches. We provide a classification of association constraints and discuss different implementation strategies. Finally, we demonstrate the feasibility of our approach with a prototype implementation and an initial catalog of association patterns.

I. INTRODUCTION

Support for associations is a given in modeling languages. However, the semantics of the relations between entities are diverse and only a handful of association constraints helping to specify these semantics are normally supported with predefined constructs. As an example, the means to express cardinalities or navigability on associations are usually provided by mainstream modeling languages, but more complex constraints such as irreflexibility or injectivity need to resort to expressions written in external constraint languages such as OCL which are difficult to create, understand, and maintain (e.g., the co-evolution of model and constraints remains a complex problem [1], [2], [3]).

This situation gets worse when code generation follows modeling. Analysis shows that existing code generation tools lack good support for association constraints [4], [5], which means that many constraints are simply ignored and no mechanism for their verification or enforcement is generated. Furthermore, target languages rarely provide direct support for associations or association constraints. As an example, as early as 1987, Rumbaugh [6] stated the lack of direct syntax or semantic support for the direct representation of relationships

in Object Oriented languages. They can be implemented (e.g., different patterns are proposed in [7]), but the implementation will scatter different parts and properties of the association instead of representing it as a unit.

In order to tackle this problem we propose the reification of recurrent association constraints so that they can be easily specified at development time and verified at runtime. We do this by leveraging the *model-oriented programming* paradigm [8], which consist on integrating high-level modeling constructs within programming languages in order, among other benefits, to avoid round-tripping issues related to code generation. Concretely, we leverage PAMELA [9], an annotation-based Java modeling framework which supports blending classical Java programming with modeling through the use of annotations and runtime code instrumentation and monitoring.

Concretely, we extend the PAMELA framework with a set of ready to use annotations representing association constraints. These annotations permit the integration of association constraints in JAVA programs in a concise way, resulting in a code that is semantically nearer to the domain. PAMELA provides us with the ability to define an open and extensible library of association constraints, in contrast with the other Model-Oriented Programming approaches. The flexibility of the PAMELA framework also allows us to define and apply custom constraint monitoring so that it can be adapted to the development process. As an example, initial development phases may require monitoring to be performed very often (e.g., to help debugging errors), so that a strong confidence in the correctness of the implementation is gained whereas later phases may only require on-demand monitoring. Different domains may also impose different monitoring requirements, e.g., association constraints that participate in the implementation of security concerns.

The remainder of the paper is organized as follows. The next section presents the PAMELA framework, the Section III describes our approach related to association constraints in PAMELA with an example of an advanced association. The Section IV positions our approach in relation to the current domain literature. Finally, we conclude our presentation with promising perspectives on this work.

II. PREVIOUS CONCEPTS

We devote this section to the description of the basic elements of our approach. Concretely, we present here the Model-oriented programming paradigm, the PAMELA framework, an instance of such paradigm and the concept of association constraint and its relevance.

A. Model-oriented programming with PAMELA

We borrow here the concept of *Model-oriented programming* [10] to refer to a paradigm in which modeling and programming are blended so that high-level modeling concepts can be used in code. PAMELA [9] is an annotation-based Java modeling framework that corresponds to this paradigm.

PAMELA provides a smooth integration between the model and code, without code generation nor externalized model serialization. Instead of generating the code, the modeling API (mostly Java interfaces with abstract method declarations) is locally executed (interpreted), avoiding round-trip issues. Figure 1 summarizes the PAMELA architecture. The left side shows how at design time code and model are built together by annotating the code with PAMELA modeling concepts such as class, attribute, method, etc. The right side shows how the PAMELA interpreter blends compiled java code with an internal model representation in order to orchestrate user's code and modeling API code at runtime.

Figure 2 shows an excerpt of the PAMELA metamodel. Basically, a *PAMELA model* contains a number of *Entities* which may point to a given implementation interface or class overriding the default model implementation (details about the functioning of PAMELA can be found in [9]). Model entities may have a number of *Properties* with classical features such as type, cardinality, etc. Instances of this metamodel are created directly within Java code by the means of annotations as we show in the following 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. Note that properties are defined *implicitly*, i.e., they are defined together with the methods that use them and thus their type is inferred from them. String identifiers in annotations are used to identify all methods related to a given property. 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).

The execution and the management of this model may be performed using the code in Listing 2. 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

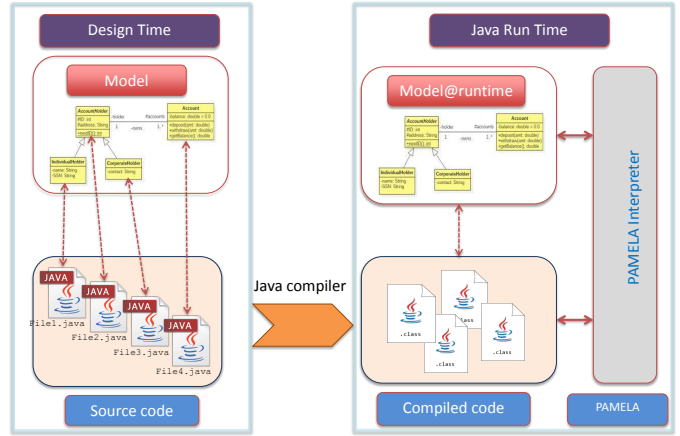


Fig. 1. PAMELA approach for modeling

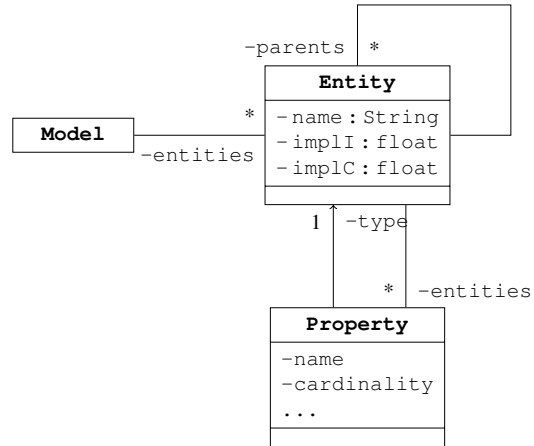


Fig. 2. PAMELA metamodel

by compiled bytecode. A factory `ModelFactory` is then instantiated using that `ModelContext`, allowing to create *Library* and *Book* instances.

```

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

```

Listing 2. Model execution/manipulation

```

1 @ModelEntity
2 public interface Book extends AccessibleProxyObject
3 {
4     @Initializer
5     public Book init(@Parameter("title")String aTitle)
6     ;
7
8     @Getter("title")
9     public String getTitle();
10
11     @Setter("title")
12     public void setTitle(String aTitle);
13
14     @Getter("ISBN")
15     public String getISBN();
16
17     @Setter("ISBN")
18     public void setISBN(String value);
19 }
20 @ModelEntity
21 public interface Library extends
22     AccessibleProxyObject {
23
24     @Getter(value = "books", cardinality = Cardinality
25         .LIST)
26     public List<Book> getBooks();
27
28     @Adder("books")
29     public void addToBooks(Book aBook);
30
31     @Remover("books")
32     public void removeFromBooks(Book aBook);
33
34     @Reindexer("books")
35     public void moveBookToIndex(Book aBook, int index)
36     ;
37
38     @Finder(collection = "books", attribute = "title")
39     public Book getBook(String title);
40 }

```

Listing 1. Model creation

B. Association constraints

Associations are a key concept in modeling languages as a way to formalize the relationships [11] between domain concepts. As an example, below is the UML’s definition of association:


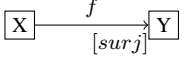
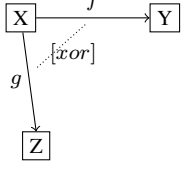
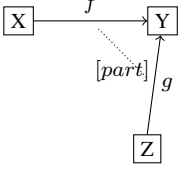
An Association specifies a semantic relationship that can occur between typed instances. It has at least two memberEnds represented by Properties, each of which has the type of the end. More than one end of the Association may have the same type. An Association declares that there can be links between instances whose types conform to or implement the associated types. A link is a tuple with one value for each memberEnd of the Association, where each value is an instance whose type conforms to or implements the type at the end. [12]

To these associations, constraints can be added so that they represent better the semantics of the domain being modeling by restricting the set of *valid* links. Some constraints can be directly represented by using modeling languages constructs. As an example, both UML and EMF support the definition of navigability and cardinality constraints directly. Other constraints require the modeler to resort to external (and normally textual) constraint languages. At the code level, associations

are rarely supported. Instead, a number of patterns are used to implement them [7]. Constraints are not supported either.

Without any claim to completeness we present in Table I four different association constraints¹. They all help capture important domain semantics which will be lost without the constraint (e.g., verification of a surjective association may be used in the implementation of garbage collections). The first two constraints (irreflexive and surjective) apply to the association itself (S) while the last two (xor and partition) apply to at least a pair of associations. Constraints one and three can be verified locally (L) while constraints two and four can not. None of them² are directly supported in mainstream modeling nor programming languages.

TABLE I
ASSOCIATION CONSTRAINT EXAMPLES

Constraint	Visualization	Semantics	S	L
(1) irreflexive		$\forall x \in X, x \notin f(x)$	Y	Y
(2) surjective		$f(X) = Y$	Y	N
(3) xor		$\forall x \in X, (f(x) = \emptyset \vee g(x) = \emptyset) \wedge (f(x) \neq \emptyset \vee g(x) \neq \emptyset)$	N	Y
(4) partition		$f(X) \cap g(Z) = \emptyset \wedge f(X) \cup g(Z) = Y$	N	N

III. APPROACH

The interest of Model-oriented programming and the need for advanced association constraints have been discussed in the previous sections. In the following, we present an approach integrating both concepts. Our approach is based on PAMELA which, as we have seen in Section II uses annotations to include high-level modeling concepts in Java programs (through the use of annotations) that are interpreted at run-time. In this sense, our approach enhances PAMELA by including:

- 1) An (extensible) infrastructure within the PAMELA framework supporting the definition and reification of advanced association constraints. These constraints are

¹The constraints along with its mathematical and graphical representation and extracted from the DPF Workbench [13]

²Note that the xor constraint can be represented in an UML diagram as a constraint, but not as a concept.

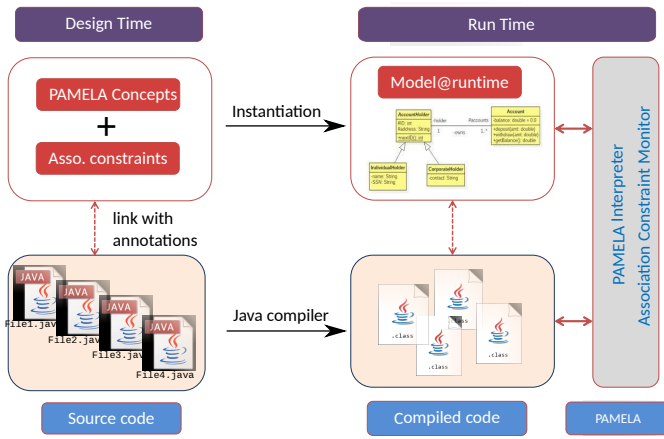


Fig. 3. PAMELA with association constraints

then exposed as a set of annotations ready to be used by programmers/modelers.

- 2) A configurable monitoring scheme that provides runtime verification of association constraints.

Figure 3 shows the aforementioned additions. The PAMELA framework including an initial set of implemented association constraints is available online³.

A. Infrastructure

There are two types of support for the association constraints. Single and local association constraints are supported as *PropertyPredicates* classes which are attached to any given PAMELA property. *PropertyPredicates* include a check method that performs the actual evaluation of the property and a number of methods to observe changes that are related to it. As an example, in Listing 3, we show the use of the constraint annotation `@Irreflexive` at design time. At runtime the use of this annotation causes the instantiation of the *Irreflexive PropertyPredicate* which is in turn monitored by the PAMELA interpreter all along the life of the program.

```

1 @ModelEntity
2 public interface X extends AbstractConcept,
   MonitorableProxyObject {
3
4     static final String SINGLE_X = "singleX";
5
6     @Getter(SINGLE_X)
7     @Irreflexive
8     X getSingleX();
9
10    @Setter(SINGLE_X)
11    public void setSingleX(X value);
12 }

```

Listing 3. Irreflexive constraint

Note that a similar functionality may be achieved by using a contract-based specification language such as JML [14] as we illustrate in Listing 4. However, we argue that directly specifying abstract contract-based invariants is harder and more difficult to maintain and understand than using reified

association constraints. Indeed the latter are more concise, explicit and declared nearer to the relevant part of the code, this is, the association declaration. Besides, contract-based approaches have limitations as they are designed to be local to a given class and thus, are not suited for non-local association constraints such as the *surjective* and *partition* constraints presented in Section II.

```

1 @ModelEntity
2 @Invariant("singleX != object")
3 public interface X extends AbstractConcept,
   MonitorableProxyObject {
4     ...
5 }

```

Listing 4. Irreflexive constraint with JML

For the more complex association constraints, e.g. those that are not local and/or represent a constraint between two or more associations, we rely on the concept of pattern, which is already implemented in the PAMELA framework (e.g., see [15]) through the *PatternDefinition* class. Complex association constraints are defined by extending this class. We show in Listing 5 the use of a *XOR* constraint by the specification at design time of the constraint annotation `@XOrAssociation` (lines 10 and 17) in the two properties that need to maintain a *xor* relation. Notice that the annotation is parametrized with the identifier defined in line 4. The behaviour at run-time is similar as the one described above for *PropertyPredicates* as patterns act as a container for properties.

B. Monitoring

Our association constraints are prescriptive and represent side-effect free constraints. In that sense they can be seen as (multi)object invariants. The PAMELA interpreter keeps track of all elements that are part of a PAMELA model, including creation, modification, etc, which facilitates monitoring of such invariants.

An in-depth discussion about the different verification techniques for object or multi-object constraints or invariants [16] lies out of the scope of the present paper, which focus is on facilitating the use of association constraints. Nevertheless, we provide here some details. For single object constraints PAMELA follows a *visible states* semantics [17], e.g., constraints are verified before and after method calls. For multi-object constraints, and to deal with collection aliasing problems, an ownership model technique [18] is also implemented.

One notable feature of our framework though is its flexibility. Indeed, the monitoring can be customized in order to adapt to the needs of different scenarios such as the stage of the development life-cycle (early phases arguably require more monitoring) or the application domain. As an example, within our framework the monitoring of constraints may be completely switched-off, performed only after specially labelled method calls or after all entity method calls or only after interpreted method calls.

C. Discussion

Reifying the association constraints taking advantage of PAMELA model@runtime features helps solve the problem

³<https://github.com/openflexo-team/pamela/tree/2.0>

```

1 @ModelEntity
2 public interface X extends AbstractConcept,
   MonitorableProxyObject {
3
4     static final String XOR_ASSOCIATION = "
       YandZareExclusive";
5
6     static final String Y_KEY = "y";
7     static final String Z_KEY = "z";
8
9     @Getter(Y_KEY)
10    @XOrAssociation(patternID = XOR_ASSOCIATION)
11    public List<Y> getY();
12
13    @Adder(Y_KEY)
14    public void addToY(Y y);
15
16    @Getter(Z_KEY)
17    @XOrAssociation(patternID = XOR_ASSOCIATION)
18    public List<Z> getZ();
19
20    @Adder(Z_KEY)
21    public void addToZ(Z z);
22 }

```

Listing 5. XOR constraint

of lack of support to such constraints in both, modeling and code. Here are a few advantages we consider:

- Association constraints are implemented in a location enabling a very simple reuse. The abstraction provided brings ease of use, as the programmer does not need to write, test and maintain complex constraints, she just has to annotate roles. Code is higher-level and more concise.
- As a proof of concept, we have implemented various kinds of constraint (see table I): on a single association (S) or can be evaluated locally (L). Literature identifies dozens of such constraints that could harden structural properties of code from high-level expressions. Associations become higher level and PAMELA implementation reinforce their semantics.
- Being reified constraints can be designed, implemented, tested and reused as other pieces of code.

On the other side, here are some limitations:

- We do not intend to *prove* the constraints; we think we improve the trust in the code, since we reuse tested code.
- We did not explore association interactions. For instance, when working with many constraint associations may lead to impossibilities.
- We assume PAMELA is correctly used. Usually, the compiler detects the omission of a role. However, a misplaced annotation may trigger unexpected behaviors. We left the exploration of the integration of annotation validation mechanisms [19] as a future work.

More generally, most design patterns can be regarded as collaboration among entities. As such, some patterns can be implemented with the proposed approach.

IV. RELATED WORK

Association constraints are discussed in the literature from different perspectives. In [4] the author discusses the limita-

tions of code generation tools w.r.t. their management, which in the vast majority of cases is left to the developers. The authors propose instead to reify associations as classes to deal with constraints. A similar report is done in [5] where the authors analyse the (limited) support associations have in code generation tools and discuss how rich UML associations are integrated in UMPLE [8]. As with PAMELA, UMPLE, is an approach that tries to blend programming and modeling by integrating UML abstractions in an object oriented language. UMPLE provides some support for associations including the representation and enforcement of multiplicities and referential integrity. The main difference w.r.t. PAMELA is that UMPLE uses code generation in order to compile UMPLE code to target languages such as Java. Conversely with PAMELA, modeling constructs are integrated directly in Java by using annotations which are then interpreted at runtime.

The need to reify complex association constraints is discussed in [20] and [21]. Both provide the means to represent complex association semantics and both approaches rely on annotations in order to constraint associations (the use of annotations as a means to reify constraints and other semantic information is widespread [22], [23], [24]). In [20] the authors provide the annotations as a profile for UML whereas in [21] the authors provide a new modeling language in which constraints can also be represented diagrammatically. Both approaches discuss code generation. Explicit support for symmetric unary association constraints is provided in [25] where the authors describe ConML. The need for complex association constraints is discussed also in the ontology realm. E.g., in [26] the authors discuss a numbers of *inter* association constraints on ontology relations and how they can be implemented with RDF. More similar to us, in [27], the authors present a framework that uses Concern-Oriented Reuse (CORE) to deal with associations and their constraints (uniqueness, multiplicity and referential integrity). CORE [28] uses aspect orientation to weave different concerns at the modeling level.

V. CONCLUSIONS & FUTURE WORK

We have presented an approach to reify complex association constraints in a model-oriented programming scenario. Concretely, we have described a framework in which association constraints can be specified directly on Java code by the means of annotations and verified at run-time.

As a future work we envision exploring the following research lines:

- Investigate the possibility, advantages and drawbacks of composing individual constraints in order to build more complex ones as an alternative to a direct definition.
- Apply the composition of constraints to the specification of patterns. We are specially interested in the definition of security patterns [15] as a composition of association constraints.
- Evaluate the usability and effectiveness of our approach (e.g., does it help developers to introduce fewer bugs and/or detect them earlier?)

REFERENCES

- [1] E. Cherfa, S. Mesli-Kesraoui, C. Tibermacine, S. Sadou, and R. Fleurquin, "Identifying metamodel inaccurate structures during meta-model/constraint co-evolution," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 24–34, IEEE, 2021.
- [2] D. E. Khelladi, R. Bendraou, R. Hebig, and M.-P. Gervais, "A semi-automatic maintenance and co-evolution of ocl constraints with (meta) model evolution," *Journal of Systems and Software*, vol. 134, pp. 242–260, 2017.
- [3] E. Batot, W. Kessentini, H. Sahaoui, and M. Famelis, "Heuristic-based recommendation for metamodel—OCL coevolution," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 210–220, IEEE, 2017.
- [4] D. Gessenharter, "Mapping the UML2 semantics of associations to a java code generation model," in *Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28-October 3, 2008. Proceedings 11*, pp. 813–827, Springer, 2008.
- [5] O. Badreddin, A. Forward, and T. C. Lethbridge, "Improving code generation for associations: enforcing multiplicity constraints and ensuring referential integrity," in *Software Engineering Research, Management and Applications*, pp. 129–149, Springer, 2014.
- [6] J. Rumbaugh, "Relations as semantic constructs in an object-oriented language," in *Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 466–481, 1987.
- [7] J. Noble, "Basic relationship patterns," *Pattern Languages of Program Design*, vol. 4, pp. 73–94, 1997.
- [8] T. C. Lethbridge, V. Abdelzad, M. H. Orabi, A. H. Orabi, and O. Adesina, "Merging modeling and programming using Umple," in *International Symposium on Leveraging Applications of Formal Methods (T. Margaria and B. Steffen, eds.), (Cham)*, pp. 187–197, Springer, Springer International Publishing, 2016.
- [9] S. Guérin, G. Polet, C. Silva, J. Champeau, J.-C. Bach, S. Martínez, F. Dagnat, and A. Beugnard, "PAMELA: an annotation-based java modeling framework," *Science of Computer Programming*, vol. 210, p. 102668, 2021.
- [10] O. B. Badreddin, A. Forward, and T. C. Lethbridge, "Model oriented programming: an empirical study of comprehension.," in *CASCON*, vol. 12, pp. 73–86, 2012.
- [11] A. Olivé, *Conceptual modeling of information systems*. Springer Science & Business Media, 2007.
- [12] OMG, "Unified modeling language (OMG UML) version 2.5. 1," *Object Management Group*, 2017.
- [13] Y. Lamo, X. Wang, F. Mantz, Ø. Bech, A. Sandven, and A. Rutle, "DPF workbench: a multi-level language workbench for MDE," *Proceedings of the Estonian Academy of Sciences*, vol. 62, no. 1, p. 3, 2013.
- [14] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: a Java modeling language," in *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pp. 404–420, Citeseer, 1998.
- [15] C. Silva, S. Guérin, R. Mazo, and J. Champeau, "Contract-based design patterns: a design by contract approach to specify security patterns," in *Proceedings of the The 6th International Workshop on Secure Software Engineering SSE@ARES*, pp. 1–9, 2020.
- [16] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers, "A unified framework for verification techniques for object invariants," in *ECOOP 2008—Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings 22*, pp. 412–437, Springer, 2008.
- [17] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens, "Modular invariants for layered object structures," *Science of Computer Programming*, vol. 62, no. 3, pp. 253–286, 2006.
- [18] J. Hogg, "Islands: Aliasing protection in object-oriented languages," in *Conference proceedings on Object-oriented programming systems, languages, and applications*, pp. 271–285, 1991.
- [19] C. Noguera and L. Duchien, "Annotation framework validation using domain models," in *European Conference on Model Driven Architecture—Foundations and Applications*, pp. 48–62, Springer, 2008.
- [20] D. Costal, C. Gómez, A. Queralt, R. Raventós, and E. Teniente, "Improving the definition of general constraints in UML," *Software & Systems Modeling*, vol. 7, pp. 469–486, 2008.
- [21] Y. Lamo, X. Wang, F. Mantz, W. MacCaul, and A. Rutle, "DPF workbench: A diagrammatic multi-layer domain specific (meta-) modelling environment," *Computer and Information Science 2012*, pp. 37–52, 2012.
- [22] M. Sulír, M. Nosál', and J. Porubán, "Recording concerns in source code using annotations," *Computer Languages, Systems & Structures*, vol. 46, pp. 44–65, 2016.
- [23] P. Kajsá and P. Návrát, "Design pattern support based on the source code annotations and feature models," in *SOFSEM 2012: Theory and Practice of Computer Science: 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings 38*, pp. 467–478, Springer, 2012.
- [24] D.-E. Khelladi, R. Bendraou, S. Baarir, Y. Laurent, and M.-P. Gervais, "A framework to formally verify conformance of a software process to a software method," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pp. 1518–1525, 2015.
- [25] C. Gonzalez-Perez and P. Martín-Rodilla, "A metamodel and code generation approach for symmetric unary associations," in *2017 11th International Conference on Research Challenges in Information Science (RCIS)*, pp. 84–94, IEEE, 2017.
- [26] D. J. Russomanno and C. R. Kothari, "Expressing inter-link constraints in OWL knowledge bases," *Expert Systems*, vol. 21, no. 4, pp. 217–228, 2004.
- [27] C. Bensoussan, M. Schöttle, and J. Kienzle, "Associations in mde: a concern-oriented, reusable solution," in *Modelling Foundations and Applications: 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings 12*, pp. 121–137, Springer, 2016.
- [28] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16*, pp. 604–621, Springer, 2013.