Consistency Recovery in Interactive Modeling

Juri Di Rocco[1], Davide Di Ruscio[1], Marcel Heinz[2], Ludovico Iovino[3], Ralf Lämmel[1,2], Alfonso Pierantonio[1]
[1] DISIM, University of l'Aquila, Italy
[2] Faculty of Computer Science, University of Koblenz-Landau Germany
[3] Gran Sasso Science Institute, L'Aquila, Italy

***Abstract*—MDE projects contain different kinds of artifacts such as models, metamodels, model transformations, and deltas. These artifacts are related in terms of relationships such as transformation or conformance. In this paper, we capture the types of artifacts and the relevant relationships in a megamodeling-based manner for the purpose of monitoring and recovering a MDE project's consistency in response to changes that users may apply to the project within an interactive modeling platform. The approach supports users in experimenting with MDE projects and receiving feedback upon changes on the grounds of a specific execution semantics for megamodels. The approach is validated within the web-based modeling platform MDEFORGE.**

***Index Terms*—Model management; Model repository; Consistency recovery; Megamodeling; Modeling platforms**

I. INTRODUCTION

Research context: This research is concerned with *model management* [1], [2] on top of *model repositories* [3], [4] which users can access through a *modeling platform* [5], [6]. Model repositories are a promising form of aggregating reusable MDE artifacts such as models, metamodels, and model transformations. Model management is the model-based (model-driven) approach to the automated management of collections of MDE artifacts instead of using ad-hoc tools or lacking good automation. Modeling platforms such as Eclipse, MDEFORGE, or MMINT are essential tools for users of model repositories—users may either explore MDE artifacts in a repository or they may be developers in some scope of the repository.

Research objective: We want to exercise an effective, declarative (model-based), and transparent (understandable) approach to organizing the artifacts in an MDE project (in a model repository or not) and the relationships between the artifacts. That is, a model-managed project has an associated megamodel so that a user (a developer within the project) can understand the structure of the project in megamodeling terms. Further, the project's consistency with the megamodel is continuously monitored in the background of the interactive modeling platform so that any changes can be mapped to corrective, automated actions to be proposed to and confirmed by the user.

Research contribution: We address the research objective by an emerging language design, definition, implementation, and integration into a modeling platform. The language and its implementation are referred to as MegaL/Forge because the language is inspired by our previous work on linguistic architecture, as a form of megamodeling, as realized by the MegaL family of languages [7], [8], [9], [10] and the integration targets the web-based modeling platform MDEFORGE [5]. An original aspect of MegaL/Forge is that its semantic model addresses consistency recovery; the approach is inspired by our previous work on relationship maintenance in software language repositories [11]. In this paper, we focus on defining the consistency-recovering response to a repository change while taking interaction with the user of a modeling platform into account.

Roadmap of the paper: Sec. II develops the running example of the paper. Sec. III defines syntax and semantics of the required megamodeling language. Sec. IV provides a semi-formal account on consistency recovery. Sec. V discusses the integration of the approach into the web-based modeling platform MDEFORGE. Sec. VI discusses related work. Sec. VII concludes the paper.

II. THE RUNNING EXAMPLE

For brevity and focus on the key idea, we commit to a very basic running example here: there are two models *m1* and *m2* that conform to the same metamodel *mm* with the difference being referred to as *delta* and model management operations in place to express conformance of *m1* and *m2* to *mm*, comparison so that *delta* represents the difference between *m1* and *m2*, and patching so that *m2* is the result of patching *m1* according to *delta*. This simple example involves enough semantical issues so that it suffices for an initial language design discussion. In our ongoing research, we also model more involved scenarios.

A. An EMF-related Prelude

The running example operates within the EMF technological space. We need these types of artifacts; we use the concrete syntax of MegaL/Forge for expressing the type definitions:

```
EmfModel // EMF−based models (XMI representation)
EmfMetamodel < EmfModel // Ecore models
EmfCompareModel < EmfModel // Delta models
```

That is, we declare types *EmfModel*, *EmfMetamodel*, and *EmfCompareModel*; we organize these types in a hierarchy ('<'). In the running example, we also need certain model-management operations on the types just declared; again, we use the concrete syntax of MegaL/Forge for expressing signatures of relations and functions:

```
conformsTo : EmfModel × EmfMetaModel
compare : EmfModel × EmfModel → EmfCompareModel
patch : EmfModel × EmfCompareModel → EmfModel
```

That is, we have access to conformance checking (*conformsTo*—a relation), model comparison (*compare*—a function), and delta application (*patch*—another function).

B. An MDE Project's Megamodel

The following megamodel declares artifact ids for models *m1* and *m2*, the shared metamodel *mm* to which both models are assumed to conform to, and the *delta* (difference) between the models. Conformance, comparison, and patch application are expressed by appropriate applications of relation *conformsTo* and functions *compare* and *path*. Thus:

```
m1, m2 : EmfModel
mm : EmfMetaModel
conformsTo(m1, mm)
conformsTo(m2, mm)
delta : EmfCompareModel
compare(m1, m2) ↦ delta
patch(m1, delta) ↦ m2
```

Let us apply the MegaL/Forge megamodel to an actual project. That is, artifact identifiers in the megamodel are linked to actual filenames in the underlying model repository (in fact, a project). We may assume links as follows:

```
m1 = "Family1.xmi"
m2 = "Family2.xmi"
mm = "FamilyMM.ecore"
delta = "Delta.xmi"
```

These links assume relative file names (relative to a base URI for the project). We are concerned with two versions of a model for describing 'families' (family members, i.e., persons with names and some relationships or attributes), subject to a suitable metamodel, and a delta (a difference) between the two models at hand.

C. Change scenarios

The modeling artefacts are subject to evolution [12]: models are modified and updated during the engineering process and the metamodels evolve over time to address changes to the requirements. Let us just imagine changes to artifacts of the project at hand. We should also explain how we expect to respond to these changes, thereby characterizing change scenarios. The key idea is that function applications in the megamodel may need to be used to recover consistency.

1) Modify delta: We propagate this change by applying the *patch* function, thereby deriving a new version of *m2* that is 'in sync' with *m1* and *delta*. Thus, the following function application facilitates consistency recovery:

```
patch(m1, delta) ↦ m2
```

We should note that we do not want to apply the *compare* function (beforehand or afterwards) because we consider changed artifacts (such as *delta* here) as 'authoritative' [13] which we do not want to overwrite along consistency recovery.

2) Modify m1: There are two options:

2.a) First compare, then patch: We apply the *compare* function to the (changed) model *m1* and the (unchanged) model *m2* to compute a new version of *delta* to be applied by means of the *patch* function, thereby deriving a new version of *m2* that is 'in sync' with *m1* and *delta*. (Adding domain knowledge ('algebraic reasoning'), we know that the new version of *m2* equals the original one.) Thus, the following list of function applications facilitates consistency recovery:

```
compare(m1, m2) ↦ delta
patch(m1, delta) ↦ m2
```

2.b) First patch, then compare: Thus:

```
patch(m1, delta) ↦ m2
compare(m1, m2) ↦ delta
```

An interactive user may disfavor this option on the grounds of domain knowledge such that (the older version of) *delta* readily captures aspects of *m1* (and *m2*) and thus, it may not work well for a new version of *m1*.

3) Modify m2: We could consider applying the *patch* function, thereby deriving a new version of *m2* that is 'in sync' with *m1* and *delta*. This is clearly not a useful strategy, as it would overwrite the changes just made to *m2*. Instead, we need to compare *m1* and *m2* to compute a new *delta*. Thus, the following function application facilitates consistency recovery:

```
compare(m1, m2) ↦ delta
```

In fact, now that we changed *delta*, we may want to check that an application of the *patch* function would derive a model that is equal to the existing model *m2*. In that case, we would have recovered consistency in the project. Of course, this is exactly the semantics of comparison: it provides a delta for two models so that the second model can be derived from the first one by applying the delta as a patch.

III. Language Definition

We provide a language definition of MegaL/Forge. In particular, we define the concrete syntax by means of a grammar and the abstract syntax by means of a metamodel. We also briefly discuss the static semantics for well-formed megamodels. Finally, we define what may account for a dynamic semantics by means of a consistency notion—an MDE project (its artifacts) to be consistent with a megamodel.

A. Concrete Syntax

The following grammar (in ANTLR notation) defines the concrete syntax of the MegaL/Forge constructs exercised in the present paper (Sec. II).

```
megamodel : declaration+;
declaration
  : type // Root entity type
  | subtype // Entity type as subtype
  | artifact // Entity
  | relation // Relation signature
  | function // Function signature
  | relatesTo // Relationship
  | mapsTo // Function application
  | link ; // Artifact binding

type : ID ; // Base type
subtype : ID '<' ID ; // Subtype < supertype
artifact : ID (',' ID)* ':' ID ; // Artifacts of a given type
relation : ID ':' ID ('#' ID)* ; // Signature
```

```
function : ID ':' ID ('#' ID)* '->' ID; // Signature
relatesTo : ID '(' ID (',' ID)* ')' ; // Relationship
mapsTo : ID '(' ID (',' ID)* ')' '|->' ID; // Apply function
link : ID '=' '"' LINK '"' ; // Bind artifact symbol to filename
```

The *type* and *subtype* forms of declaration facilitate the definition of a nominal classification hierarchy for artifacts. Actual artifacts are introduced by their name (an id); see declaration form *artifact*. The declaration forms *relation* and *function* facilitate signatures including names for arguments and results (the latter for functions only). There is a declaration form *relatesTo* for expressing relationships on artifacts. There is a declaration form *mapsTo* for expressing the specific relationship of function application. Finally, there is a special declaration form *link* for binding artifact symbols to files. Making the links part of the megamodel rather than designating a separate model for links can be compared to the use of annotations in OO programming rather than using XML-based configuration.

MegaL/Forge is a very simple member in the MegaL language family [7], [8], [9]. In particular, there is only one kind of types—as opposed to languages versus artifacts versus concepts in other MegaL languages.

B. Abstract Syntax

The metamodel defining the abstract syntax of the language is shown in Fig. 1. In particular, a megamodel specification consists of a set of *Artifact*s of different *Type*s. Each function (relation) is defined by means of a *Function* (*Relation*) declaration and the corresponding *MapsTo* (*RelatesTo*) definition. Artifacts are arguments of *Function*s and *Relation*s as shown by the constructors of the *MapsTo* and *RelatesTo* elements. The former consists of *input* and *output* elements, whereas the latter consists of the set of artifacts for which the specified relation holds. All the elements in the figure specialize a *NamedElement* class (not shown in the figure for brevity) consisting of the *name* attribute of type *String*.

C. Static Semantics = Well-formedness

A static semantics for well-formedness of MegaL/Forge-like megamodels was defined as a definite clause program in previous work [8]. We summarize the relevant constraints informally to make this text more self-contained.

a) Types, artifacts, relations, and function are declared before they are used. b) Each name can be declared once only ('no overloading' here of any kind). c) The arguments of relationships and function applications and results of function applications must be of the types as prescribed by the signatures of the corresponding relations and functions. d) Each artifact symbol is linked to some filename. (We do not consider incompletely bound megamodels here.)

D. Dynamic Semantics = Consistency

Megamodels may have various dynamic semantics [8], [11]; we are interested here specifically in a semantics that captures consistency of an MDE project with regard to a megamodel. We provide a simple semantics of this kind from the ground up here.

We take consistency to mean that all relationships on artifacts in the project, as expressed by the megamodel, i.e., applications of relations and functions, must hold, subject to suitable interpretations of the applied relations or functions. Details follow below.

1) Environments for Interpretation: In an effort to set up interpretations of symbols used in megamodels systematically, we assume an environment E which is, in fact, a triple $\langle E_A, E_R, E_F \rangle$ as follows:

- E_A maps artifact symbols, as they are used in the megamodel, to actual artifact representations in the sense of text, JSON, etc. In the MegaL/Forge notation (see Section II-B), we assume a mapping from artifact symbols to files. In the semi-formal model at hand, we assume a universe U for representations of artifacts. We use the type U in setting up interpretations for relation and function symbols; see below.
- E_R maps relation symbols to their interpretations; these are predicates of type $U^+ \rightarrow \mathit{Boolean}$. We use here U^+ for each predicate's argument because, in this manner, a simple generic type suffices for all possible relations. The idea is, of course, that a suitable interpretation enforces a certain length (a certain number of parameters) and appropriate representation types (subtypes of U) for the different parameters.
- E_F maps function symbols to their interpretations; these are functions of type $U^+ \rightarrow U$.

As an environment effectively represents what we think of as a 'project', we may also speak of consistency of a megamodel with an environment.

2) Consistency = Relational + Functional Consistency: We speak of *relational consistency* when the interpretation of all relation applications ('relationships) in a given megamodel m for a given environment E returns true. We speak of *functional consistency* when the interpretation of all function applications in the megamodel m with the environment E returns true. Details of the assumed notion of interpretation follow.

A relation application $r(a_1, \ldots, a_n)$ with the relation symbol r and artifact symbols a_1, ..., a_n as arguments is interpreted by applying the interpretation of r to the interpretations of a_1, ..., a_n, as defined by the environment. Thus:

$$E_R(r)(\langle E_A(a_1), \ldots, E_A(a_n) \rangle)$$

A function application $f(a_1, \ldots, a_n) \mapsto a_{n+1}$ with the function symbol f, artifact symbols a_1, ..., a_n as arguments, and an artifact symbol a_{n+1} for the result is interpreted by applying the interpretation of f to the interpretations of a_1, ..., a_n, as defined by the environment, and comparing the result with the interpretation of a_{n-1} for equality. Thus:

$$E_F(f)(\langle E_A(a_1), \ldots, E_A(a_n) \rangle) = E_A(a_{n+1})$$

For consistency to hold, the formulae as described above should evaluate to true for all relation and function applications.

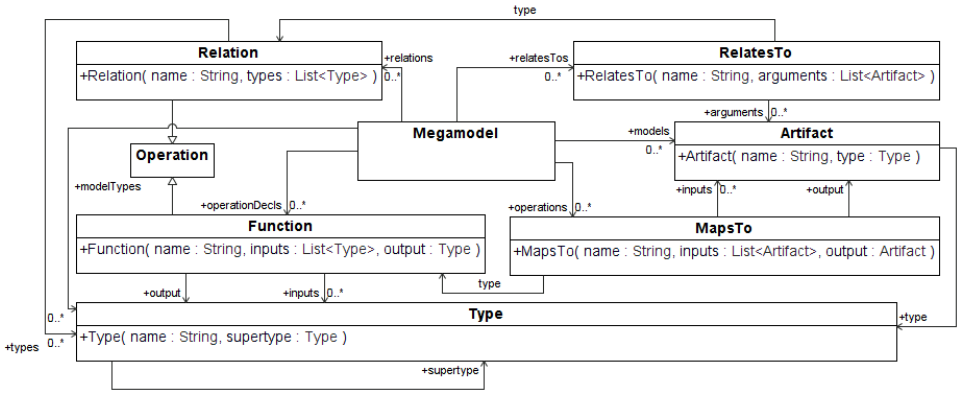

Fig. 1. The MegaL/Forge metamodel.

IV. Consistency recovery

In response to a change in a project, we perform a *recovery analysis* on the megamodel to determine the function applications (a *recovery sequence*) for recovering consistency, when applied to the artifacts in the project.

A. Recovery Sequence

When consistency does not hold, then we may try to 'overwrite' artifacts according to function applications so that consistency is recovered. The major assumption is here that function applications, as they are part of the megamodel at hand, suffice for consistency recovery and a suitable order can be determined. In more detail, given a sequence of function applications $fa_1, \ldots, fa_n$ from a given megamodel m, we call it a recovery sequence for a given environment E, if

- E is not consistent with m.
- Apply $fa_1, \ldots, fa_n$ in the given order to overwrite the artifact symbols for the results in the environment E.
- The updated environment E is now consistent with m.

B. Recovery Analysis

It remains to define an analysis for megamodels to map changes to recovery sequences. For simplicity, we start from the assumption that changes are atomic in the sense that single artifacts are changed on a discrete timeline and consistency is to be recovered after each change. Thus, the analysis is essentially a mapping from a megamodel m and an artifact symbol a_c identifying the actual change to a sequence of function applications.

Let us discuss expected properties of the analysis. We do not want to map a change to a sequence that would change an artifact that was changed previously, as such 'overwriting' may be semantically debatable and it may also lead to divergence. As a special case, we do not want to apply any function application twice. For instance, this could happen, in the running example, if we were responding to model changes with comparison and patching in a cyclic manner.

We also need to address 'nondeterminism' in the context of consistency recovery. That is, there may exist megamodels and changes for which different recovery sequences are possible; see the two scenarios for changing *m1* in Sec. II-C2. We may either delegate such nondeterminism to the interactive component or enhance megamodels and the analysis thereof to resolve nondeterminism automatically.

Let us now sketch a first attempt at the desired analysis; we defer proper treatment of nondeterminism to future work. We need helper functions as follows:

- $in(m, a)$ returns all the function applications in the megamodel m with the artifact symbol a as an argument.
- $out(fa)$ returns the artifact symbol for the result of the function application fa.

The main function for **r**ecovery **a**nalysis, *ra*, takes as arguments the megamodel m, an artifact symbol a_c indicating the change, a sequence S of function applications, and it returns a sequence of function applications that may be a recovery sequence. We begin with an empty S and extend it into the result sequence, step by step.

$$ra(m, a_c, S) = \begin{cases} ra(m, a_c, S +\!\!+ \langle fa \rangle), \text{with } fa \text{ drawn from } m \text{ such that} \\ \quad - \ fa \notin S, \text{and} \\ \quad - \ fa \in in(m, a_c) \cup \bigcup_{fa' \in S} in(m, out(fa')), \text{and} \\ \quad - \ out(fa) \neq a_c, \text{and} \\ \quad - \ out(fa) \neq out(fa') \text{ for all } fa' \in S. \\ S, \text{otherwise (if there is no such } fa) \end{cases}$$

('$+\!\!+$' is list append.) The conditions control that we select function applications that can be applied to a_c and results of previous applications without though any overwriting. The formulation is nondeterministic, as different function applications could be picked in a step.

For instance, for the megamodel of Sec. II and $a_c = m1$, starting from $S = \emptyset$, the analysis returns a sequence starting with a comparison, followed by a patch as follows:

compare(m1, m2) $\mapsto$ *delta*
patch(m1, delta) $\mapsto$ *m2*

Here we assume that the analysis respects the megamodel-defined order of function applications. (Also, '$\cup$' operates on sequences rather than sets.) Proper treatment of nondeterminism is deferred to future work.

V. Integration into MDEForge

This section presents the implementation of the presented consistency recovery approach, which has been integrated in the MDEForge platform [5]. MDEForge was proposed as an extensible platform enabling the adoption of model management tools as SaaS (software as a service). By resembling facilities of desktop IDEs, like Eclipse, MDEForge users have the possibility to create modeling artifacts and organize them in projects that are, in turn, contained in workspaces.

The consistency recovery mechanism presented in the previous section has been integrated in MDEForge by essentially extending the existing project management facilities. In particular, the Java packages *ProjectMonitoring* and *ConsistencyManagement* shown in Fig. 2 contain the new classes and interfaces that have been added in the MDEForge implementation. The existing package *CoreService* has been extended to work with such new packages.

The *ProjectMonitoring* package implements listeners that execute the consistency recovery manager when artifacts or projects are changed. In such cases, *ApplicationEvent*s are created as shown in the listing below and used by *ArtifactChangedListener* and *ProjectChangedListener* to interact with the *ConsistencyRecoveryManager*.

```
public void update(T artifact) {
    ...
    eventPublisher.publishEvent(new ArtifactChangedEvent(artifact, "
        UPDATE"));
    artifactRepository.save(artifact);
}
```

The *ConsistencyManagement* package implements the presented consistency recovery concepts. For each symbolic function or relation name a corresponding *IOperationApplier* implementation is available. For instance, the functions *compare* and *patch* discussed in Sec. II-B have the corresponding implementation consisting of the *ComparisonApplier* and *PatchApplier* classes, respectively. Such classes implement the method *apply* that executes the real behaviour of the corresponding function. For instance, the execution of the *apply* method of the class *ComparisonApplier* executes the comparison mechanism already available in MDEForge (that in turn is based on EMFCompare[1]) as shown in the following listing showing a fragment of the *apply* method of the class *ComparisonApplier*:

```
public Object apply(Object[] inputs)
{
  if (inputs.length != 2)
    throw new Exception();
  Artifact left = inputs[0];
  Artifact right = inputs[1];
  ModelsServiceImpl modelService=new ModelService();
  return modelService.compare(left, right);
}
```

The links between symbolic operation names and the corresponding appliers are specified in the *operationMapper* HashMap of the *ConsistencyRecoveryManager* class. The consistency check between a project and the corresponding megamodel is performed by the method *checkConsistency* shown below:

```
public boolean checkConsistency(Project project, Megamodel megamodel) {
    for (Artifact relatesTo : m.models) {
        List <ArtifactChangedEvent> changes= checkChanges(project);
        if (changes.size()==0) return true;
    }
    for (RelatesTo relatesTo : m.relatesTos) {
        IOperationApplier opApplier = operationMapper.get(relatesTo.
            getType());
        boolean result = (boolean) opApplier.apply(relatesTo.arguments);
        if (!result) return false;
    }
    return true;
}
```

The method *consistencyRecovery* implements the recovery mechanism by exploiting the *getFunctionsToRecoverConsistency* method shown in the listing below. It is responsible of identifying the functions to be applied and their execution order for recovering the consistency between the changed project and the corresponding megamodel.

```
public List<MapsTo> getFunctionsToRecoverConsistency(Project project,
     Megamodel megamodel){
    List<MapsTo> result = new ArrayList<MapsTo>();
    List <ArtifactChangedEvent> changes=checkChanges(project);
    for (MapsTo function : m.functions) {
        if (function.inputs.contains(changes)) result.add(function);
    }
    return result;
}
```

A fragment of the *consistencyRecovery* method is as follows:

```
public void consistencyRecovery(Project project, Megamodel megamodel) {
    checkConsistency(project, megamodel);
    List<MapsTo> toApply = getFunctionsToRecoverConsistency(project,
        megamodel);
    for (MapsTo function : toApply) {
        IOperationApplier opApplier = operationMapper.get(function.
            getType());
        opApplier.apply(m.inputs);
    }
    ...
}
```

Consistency recovery can generate new artifacts that might overwrite existing ones. In such cases, the user will be notified and will be asked for confirmation, as illustrated with the pop-up in Fig. 3. The user will also be notified when consistency recovery fails. We are also working on presenting options to the user.

VI. Related Work

In general, a few model management platforms exist, such as MMINT [6] and Mondo [14] that try to support any kind of operation needed in model driven engineering with a focus on models. The problem of well-formed metamodelling is directly addressed by the model management platform 'Modelverse' [15]. It is a platform emphasizing a consistently specified form of metamodelling based on the work by Kühne et al. [16]. Tools may not properly check conformance to metamodels as the static semantics is left unattended [17]. SERGe generates all possible metamodel consistency preserving transformations to be reused by other tools.

[1]https://www.eclipse.org/emf/compare/

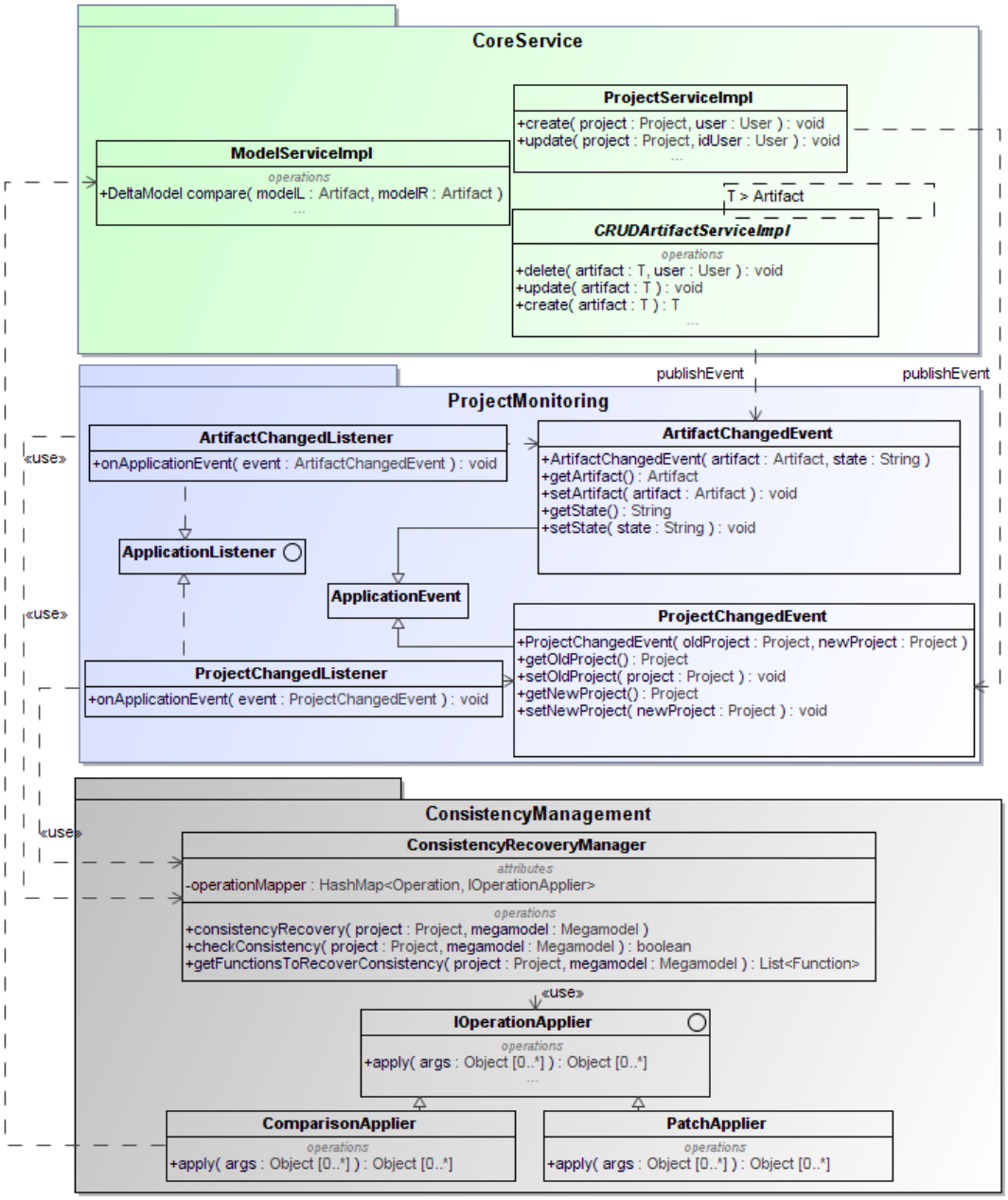

Fig. 2. Event based architecture of MDEForge.

In requirements engineering, the consistency between requirement artifacts needs to be maintained. In [18] authors propose to use the Snapmind Framework and a UML-based specification environment for user stories and domain models. The relation between elements in a mind map-based user story and domain models are checked. In [19] authors explicitly define correspondence relationships in architecture descriptions for all kinds of digital artifacts. Kowal et al. [20] explicitly aim at delta-aware consistency checking for models that are part of difference perspectives by using rules that describe a consistent UML-based architecture description. In [21] a systematic literature review is presented on consistency checking of business process models that pose further related approaches in the domain.

In [22] authors approach consistency checking for evolving models and consistency recovery using state space exploration based on postulates defining consistent states. Sunye [23] addresses collaborative modelling processes, where multiple editors write on the same model. An addressed challenge lies in reproducing the same order of operations for every accessing node. In [24] authors describe a system using SAT solvers to propagate all possible changes for source models

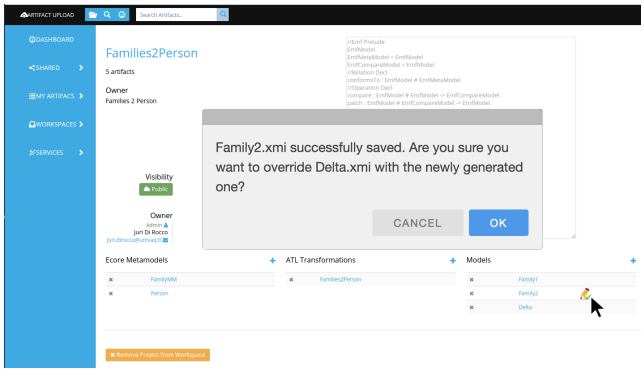

Fig. 3. Pop-up asking the user regarding overwriting.

such that the transformation to view models remains traceable.

Bidirectional transformations pose a need for consistency checking and change propagation. Demuth et al. [25] discuss failure detecting for co-evolving metamodels and domain models. When a consistency check fails, repair measure suggestions are automatically generated. Kusel et al. [26] explicitly state the properties that need to be verified after a coupled model transformation. Diskin et al. [27] classify the various kinds of model synchronizations that may have to be considered. Other kinds of bidirectional transformations and their specific needs are discussed in [28].

VII. Concluding Remarks

In this paper, we have described an approach towards consistency recovery in MDE projects such that megamodels are facilitated for expressing consistency and providing guidance for recovery thereof in an interactive setup. We are working on the following improvements. Firstly, we look at more complex scenarios with richer dependencies between the involved artifacts so that the issue of nondeterminism (Sec. IV-B) is (needs to be) properly addressed. For instance, we study examples of co-evolution [29]. Secondly, we look at making megamodels and the underlying MDE projects explicitly version-aware so that consistency recovery can be modeled atop versioning. Thirdly, we look at techniques for automatically creating (at least initial fragments of) megamodels out of existing MDE projects. Finally, we look at the incorporation of 'algebraic reasoning', possibly also subject to capturing more domain knowledge in the megamodel, for the benefit of omitting unnecessary recovery steps (e.g., *patch* after *commit* in the running example) or addressing nondeterminism.

References

[1] M. Barbero, F. Jouault, and J. Bézivin, "Model Driven Management of Complex Systems: Implementing the Macroscope's Vision," in *Proc. ECBS 2008*. IEEE, 2008, pp. 277–286.

[2] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot, "MoScript: A DSL for Querying and Manipulating Model Repositories," in *Proc. SLE 2011*, ser. LNCS, vol. 6940. Springer, 2012, pp. 180–200.

[3] F. Basciani, J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio, "Model Repositories: Will They Become Reality?" in *Proc. CloudMDE@MoDELS 2015*, ser. CEUR Workshop Procs, vol. 1563, 2016, pp. 37–42.

[4] J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio, "Collaborative Repositories in Model-Driven Engineering," *IEEE Software*, vol. 32, no. 3, pp. 28–34, 2015.

[5] F. Basciani, J. D. Rocco, D. D. Ruscio, A. D. Salle, L. Iovino, and A. Pierantonio, "MDEForge: an Extensible Web-Based Modeling Platform," in *Proc. CloudMDE@MoDELS 2014*, vol. 1242, 2014, pp. 66–75.

[6] A. D. Sandro, R. Salay, M. Famelis, S. Kokaly, and M. Chechik, "MMINT: A Graphical Tool for Interactive Model Management," in *Proc. MoDELS 2015 Demo and Poster Session*, ser. CEUR Workshop Procs, vol. 1554, 2016, pp. 16–19.

[7] J. Favre, R. Lämmel, and A. Varanovich, "Modeling the Linguistic Architecture of Software Products," in *Proc. MODELS 2012*, ser. LNCS, vol. 7590. Springer, 2012, pp. 151–167.

[8] R. Lämmel and A. Varanovich, "Interpretation of Linguistic Architecture," in *Proc. ECMFA 2014*, ser. LNCS, vol. 8569. Springer, 2014, pp. 67–82.

[9] J. Härtel, L. Härtel, M. Heinz, R. Lämmel, and A. Varanovich, "Interconnected Linguistic Architecture," *The Art, Science, and Engineering of Programming Journal*, vol. 1, 2017, 27 pages.

[10] M. Heinz, R. Lämmel, and A. Varanovich, "Axioms of linguistic architecture," in *Proc. MODELSWARD 2017*. SCITEPRESS, 2017, pp. 478–486.

[11] R. Lämmel, "Relationship maintenance in software language repositories," *The Art, Science, and Engineering of Programming Journal*, vol. 1, 2017, 27 pages.

[12] R. F. Paige, N. Matragkas, and L. M. Rose, "Evolving models in model-driven engineering: State-of-the-art and future challenges," *Journal of Systems and Software*, vol. 111, pp. 272 – 280, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001909

[13] P. Stevens, "Bidirectional Transformations in the Large," in *MoDELS*. ACM, 2017, to appear.

[14] D. S. Kolovos, A. García-Domínguez, R. F. Paige, E. Guerra, J. S. Cuadrado, J. de Lara, I. Ráth, D. Varró, G. Sunyé, and M. Tisi, "MONDO: scalable modelling and model management on the cloud," in *STAF*, 2016, pp. 55–64.

[15] S. V. Mierlo, B. Barroca, H. Vangheluwe, E. Syriani, and T. Kühne, "Multi-level modelling in the modelverse," in *MoDELS*, 2014, pp. 83–92.

[16] T. Kühne, "Matters of (meta-)modeling," *Software and System Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[17] M. Rindt, T. Kehrer, and U. Kelter, "Automatic generation of consistency-preserving edit operations for MDE tools," in *MoDELS*, 2014.

[18] F. Wanderley, A. Silva, J. Araújo, and D. S. da Silveira, "Snapmind: A framework to support consistency and validation of model-based requirements in agile development," in *MoDRE*, 2014, pp. 47–56.

[19] A. Chichignoud, F. Noyrit, L. Maillet-Contoz, and F. Terrier, "Use of architecture description to maintain consistency in agile processes," in *MODELSWARD*, 2017, pp. 459–466.

[20] M. Kowal and I. Schaefer, "Incremental consistency checking in delta-oriented uml-models for automation systems," in *Procs. 7th Intl. FMSPLE@ETAPS Workshop 2016*, 2016, pp. 32–45.

[21] A. Awadid and S. Nurcan, "A systematic literature review of consistency among business process models," in *CAiSE*, 2016, pp. 175–195.

[22] H. K. Dam and A. Ghose, "Towards rational and minimal change propagation in model evolution," *CoRR*, vol. abs/1402.6046, 2014.

[23] G. Sunyé, "Model consistency for distributed collaborative modeling," in *Proc. of ECMFA*, 2017, pp. 197–212.

[24] O. Semeráth, C. Debreceni, Á. Horváth, and D. Varró, "Incremental backward change propagation of view models by logic solvers," in *MoDELS*, 2016, pp. 306–316.

[25] A. Demuth, M. Riedl-Ehrenleitner, R. E. Lopez-Herrejon, and A. Egyed, "Co-evolution of metamodels and models through consistent change propagation," *JSS Journal*, vol. 111, pp. 281–297, 2016.

[26] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, W. Schwinger, and J. Schönböck, "Consistent co-evolution of models and transformations," in *MoDELS*, 2015, pp. 116–125.

[27] Z. Diskin, H. Gholizadeh, A. Wider, and K. Czarnecki, "A three-dimensional taxonomy for bidirectional model synchronization," *JSS Journal*, vol. 111, pp. 298–322, 2016.

[28] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *Proc. of ICMT*, 2009, pp. 260–283.

[29] D. D. Ruscio, L. Iovino, and A. Pierantonio, "Coupled evolution in model-driven engineering," *IEEE Software*, vol. 29, no. 6, pp. 78–84, 2012.