On Lightweight Metamodel Extension to Support Modeling Tools Agility

Hugo Bruneliere[1(✉)], Jokin Garcia[1], Philippe Desfray[2], Djamel Eddine Khelladi[3], Regina Hebig[3], Reda Bendraou[3], and Jordi Cabot[4]

[1] AtlanModTeam (Inria, Mines Nantes & LINA), Nantes, France
{hugo.bruneliere,jokin.garcia-perez}@inria.fr
[2] SOFTEAM Cadextan, Paris, France
philippe.desfray@softeam.fr
[3] UPMC - LIP6, Paris, France
{djamel.khelladi,regina.hebig,reda.bendraou}@lip6.fr
[4] ICREA - UOC, Barcelona, Spain
jordi.cabot@icrea.cat

Abstract. Modeling in real industrial projects implies dealing with different models, metamodels and supporting tools. They continuously have to be adapted to changing requirements, involving (often costly) problems in terms of traceability, coherence or interoperability. To this intent, solutions ensuring a better adaptability and flexibility of modeling tools are needed. As metamodels are cornerstones in such tools, metamodel extension capabilities are fundamental. However, current modeling frameworks are not flexible or dynamic enough. Thus, following the ongoing OMG MOF Extension Facility (MEF) RFP, this paper proposes a generic lightweight metamodel extension mechanism developed as part of the MoNoGe collaborative project. A base list of metamodel extension operators as well as a DSL for easily using them are introduced. Two different implementations of this extension mechanism (including a model-level support when (un)applying metamodel extensions) are also described, respectively based on Eclipse/EMF and the Modelio modeling environment.

Keywords: Modeling tool · Metamodel extension · Adaptability · Flexibility

1 Introduction

Model Driven Engineering (MDE) in general and modeling environments/tools in particular are used within the industry in various contexts and for varied purposes [6]. In many cases, companies (both solution providers and users) have to adapt their model-based infrastructure because of changing requirements or technological constraints. This usually comes with a range of potential issues including traceability, coherence or interoperability ones regarding both the modeling artifacts and data conforming to them. This is particularly true for modeling tools that heavily rely on their core supported metamodel(s). Indeed, such

G. Taentzer and F. Bordeleau (Eds.): ECMFA 2015, LNCS 9153, pp. 62–74, 2015.
DOI: 10.1007/978-3-319-21151-0_5

metamodels may need to evolve over time and new/other ones may have to be additionally supported by these tools (e.g. due to customer or market requirements). Compatibility with already existing models must be preserved, but new models (conforming to completely different metamodels not yet supported or to slightly modified versions of existing ones) have to be considered too. Both cases should be addressed, ideally in such a way that the effort implied by the corresponding modifications to the tools is limited as much as possible. Thus, there is a clear need for adaptability and flexibility in modeling tools/environments. This *agility* requires lightweight metamodel extension capabilities having several interesting properties such as compatibility preservation but also genericity, non-intrusiveness, transparency or some dynamicity (as explained later in the paper).

Intending to face up current limitations and the lack of standard solutions (e.g. the OMG MOF Extension Facility (MEF) is still an ongoing RFP [13]), we propose a dedicated solution in the context of the MoNoGe French collaborative project[1]. A generic lightweight metamodel extension approach is being developed and experimented in an industrial environment where rapid and efficient adaptations of the used modeling tools are required. Of course, these tools have to be modified once to somehow integrate the proposed mechanism. However, among other reasons detailed later, we consider it *lightweight* because it then does not require model migration/transformation processes anymore. It provides metamodel extension operations to cover real scenarios involving addition, updating and filtering changes to existing metamodels. Metamodel extension declarations can be defined and then shared between different modeling tools using a dedicated Domain Specific Language (DSL). Thus, the main contributions of the paper are: i) a base list of metamodel extension operators and corresponding generic DSL, ii) an overall architecture for implementing a metamodel extension mechanism based on Eclipse/EMF, transparent from an end-user point of view and iii) (complementarily) an alternative DSL-compliant solution relying on the Modelio modeling environment as needed in the MoNoGe project.

The remainder of the paper is structured as follows. We start by explaining with more details our motivation in Section 2, setting the goals and scope of our work. In Section 3, we introduce a core list of metamodel extension operators and the related textual DSL we propose. Then, we describe in Section 4 the proposed capabilities and architecture to implement a corresponding metamodel extension mechanism relying on Eclipse/EMF modeling technologies. We also present in Section 5 an alternative DSL-compliant solution based on the Modelio modeling environment. We discuss the related work in Section 6 before we finally conclude in Section 7 with some remaining challenges and future work.

2 Motivation and Industrial Background

As introduced before, the use of MDE-based environments and modeling tools is relatively widespread in the industry. For various reasons (e.g. new customer

[1] http://www.images-et-reseaux.com/en/content/monoge

needs, technical constraints or business decisions to cite a few), these solutions have to evolve quite frequently. Related changes can concern several different aspects: UI can be modified, new features can be added or previous ones removed, tool's core can be restructured, etc. In all cases, it is important for software providers to be able to adapt their tools as easily as possible when implementing these modifications. According to the promises of MDE, minimizing the cost/-effort of such evolution is fundamental.

In the particular context of modeling environments, core supported metamodels are key elements since most components are derived from them (parsers, editors, verifiers, generators, etc.). Core modification of such environments generally implies the adaptation of these metamodels and related tooling features. Modelio is a concrete example of a modeling tool implementing popular standards such as UML, BPMN, SysML, etc. Users frequently need to reuse pieces of these standards and create extensions related to domain-specific solutions for System, Enterprise Architecture or Requirement modeling (for instance). Thus, already supported metamodels need to be modified to reflect some changes: new complementary concepts could be added, previously existing ones could be updated or even filtered if not relevant anymore. In addition, brand new metamodels may also have to be supported ensuring quality properties such as traceability of the different versions or coherence between dependent artifacts. While compatibility with existing models must usually also be preserved, new models (that can conform to modified or different metamodels) have to be taken into account too. Both kinds of models need to coexist smoothly within the tool. As a consequence, modeling environments have to be able to adapt to all these situations with as much agility as possible.

Illustrating this situation, the MoNoGe main industrial use case comes from DCNS, a world-leading company in naval defense and energy that notably develops *CMS* (*Combat Management Systems*) for ships. In one of its programs, DCNS is using two separate modeling tools: one (System Architect) for system-level modeling using the *DoDAF* (*U.S. Department of Defense Architecture Framework*) standard, the other (Modelio) supporting software design and development. DCNS needs to manage permanent consistency between the system and software modeling levels (plus related traceability and impact analysis), but cannot customize System Architect.

Thus, part of the work in MoNoGe consists in building a metamodel extension, in Modelio, to trace and enrich software models with DoDAF elements (from a subset of the DoDAF metamodel). The objective is to allow architects and developers to work as before on their current models while, at the same time, both types of models can be exchanged between the two modeling environments and linked together. Only the users who need to see traceability and impact analysis have access to these extended models combining software- and system- levels. Interoperability and consistency management stay straightforward as there is no actual model transformation/migration, just this extended view of the models in Modelio depending on the user profile.

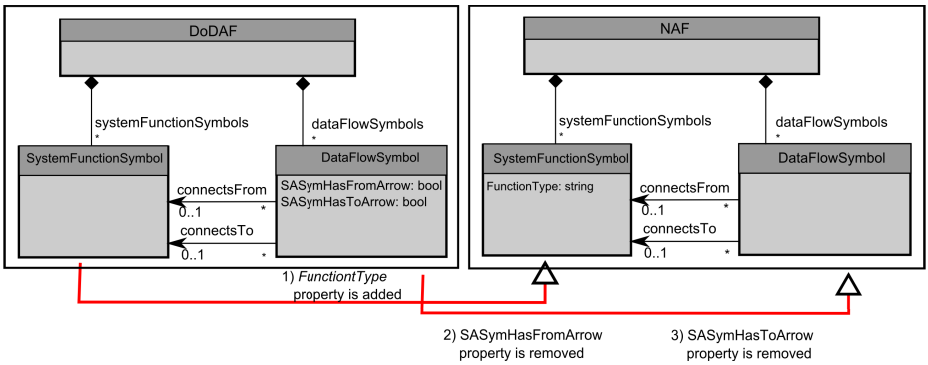

Fig. 1. Defining NAF by extending DoDAF (excerpt)

Another use case, still in the same DCNS domain, is also being conducted using an Eclipse/EMF-based environment to demonstrate the genericity of the proposed extension mechanism and to provide an open source alternative. DCNS wants to evolve their existing DoDAF models and related tooling support to *NAF* (*NATO Architecture Framework*), which is another architecture framework deriving its main concepts from *MoDAF* (*British Ministry of Defense Architecture Framework*). As NAF is based on Enterprise Architecture concepts relatively close to the ones existing in DoDAF, the more direct way to make this happen is to define an extension of the DoDAF metamodel for supporting NAF. The goal is notably to enable the automatic reuse of existing DoDAF models in a NAF context. Due to space limit, we introduce here only small parts of the concerned metamodels and highlight a few required changes. As examples, the following modifications can be seen in Figure 1:

1. Adding a property *FunctionType* in the concept *SystemFunction*.
2. Deleting a property *SASymHasFromArrow* from the concept *DataFlowSymbol*.
3. Deleting a property *SASymHasToArrow* from the concept *DataFlowSymbol*.

Based on these two case studies we conclude that flexible extensions that do not necessarily require to migrate existing models, and that allow preserving viewpoints/models of current stakeholders, are an efficient mean to smoothly integrate modeling tools and increase their scope. To support this, a lightweight metamodel extension mechanism is needed, like the one we describe in the next sections.

3 Defining Metamodel Extensions

The first key ingredient of our metamodel extension mechanism is to have an easy way to express extensions. For that, we provide in this section a textual DSL that offers an initial list of extension operators (providing base semantics for extension) to be used when specifying metamodel extensions. After a few

introductory definitions, we review the list of operators and textual DSL we are proposing based on them.

3.1 Terminology and Definitions

In this paper we consider the following definitions. An **original metamodel** is an already existing metamodel that has a life on its own (e.g. is integrated in various tools/solutions, has models that conform to it, etc.). A **metamodel extension** is the definition of an extension that is, partially or completely, relying on concepts coming from original metamodel(s) or from other previously extended metamodel(s). An **extended metamodel** is the result of the application of one (or several) metamodel extension(s) onto original or already extended metamodel(s). An **existing/legacy model** is an already existing model that conforms to an original metamodel, but not necessarily to the extended metamodel(s) that could have been specified from this metamodel.

3.2 A Base Set of Metamodel Extension Operators

A metamodel extension specification consists in a set of atomic extension operations, usually applied on existing metamodel elements. Notably to simplify the management of extensions (see next section), our goal is to minimize the number of base operators. These operators can be combined later on to express more complex changes. Such combinations could also be offered on modeling infrastructures supporting them, e.g. as a more powerful predefined extension library.

Our definition of these operators is not linked to any particular technical environment and therefore could be adopted by all modeling frameworks. Since metamodels are typically specifying a set of concepts with properties (possibly attributes or references), we follow the same approach for introducing the operators hereafter:

- **ADD** (a new concept to the metamodel)
  - **Create** "from scratch" a completely new concept.
  - **Specialize** (subtype) a concept.
  - **Generalize** (supertype) one or several concept(s).
- **MODIFY** (an existing concept in the metamodel)
  - **Add property** to an existing concept.
  - **Filter property** from an existing concept.
  - **Modify property** of an existing concept (equivalent to Filter + Add).
  - **Add constraint** to an existing concept or one of its properties.
  - **Filter constraint** from an existing concept or one of its properties.
- **FILTER** (an existing concept in the metamodel).

Constraints on metamodels can be expressed using either natural language or more dedicated languages depending on implementations (cf. Section 4 for instance). Note that we are voluntarily using the term FILTER and not DELETE. For coherence and compatibility with the existing/legacy models, we

want our extension mechanism to be as little intrusive as possible. Thus, we do not want to actually delete elements but rather hide them when asked for. Filtering is applied on cascade (e.g. in the case of generalizations or derived properties) and related constraints updated accordingly[11].

3.3 A Textual DSL for Metamodel Extension

Extensions should be easily written by modelers/engineers in a comprehensive way, justifying the need for a DSL [19]. A textual DSL has been designed in order to make available the previously introduced extension operators via a textual concrete syntax very close to our metamodel extension terminology. This syntax is intended to be intuitive and easy-to-learn for people already familiar with (meta)modeling, and reflects the full list of base extension operators as presented before. Having genericity and portability in mind, it has been defined independently from any particular metamodel or modeling framework/environment.

The overall structure to declare an extension includes its name, the metamodel(s) it extends and the list of applied operators (as well as the metamodel elements they are applied to). Figure 2 presents the full grammar of our textual DSL, thus highlighting its main concepts and structure.

```
Model:'define' extensionName=ID 'extending' metamodel+=Metamodel ':'
    prefix+=Prefix ("," metamodel+=Metamodel ':'  prefix+=Prefix)*
    '{' extensions += Extension* '}';
Extension: Create | Refine | Generalize | ModifyClass | FilterClass;
Metamodel:  name=ID;
Prefix: name=ID;
Create: 'add class' class=ID;
Refine: 'add class' classNew=ID 'specializing' prefix=[Prefix] '.'
    classOriginal=ID;
Generalize: 'add class' classNew=ID 'supertyping' prefix+=[Prefix]
    '.' class+=ID ("," prefix+=[Prefix] '.' class+=ID)*;
ModifyClass:
    'modify class' prefix=[Prefix] '.' class=ID '{'
        modifyOperators += ModifyOperator*
    '}';
ModifyOperator: AddProperty | ModifyProperty | FilterProperty |
    AddConstraint | FilterConstraint;
AddProperty:    'add property' property=ID 'type' type=ID;
ModifyProperty: 'modify property' property=ID value+=ValueAssignment
    ("," value+=ValueAssignment)*;
ValueAssignment:    attribute=ID '=' value=EString;
FilterProperty: 'filter property' property=ID;
FilterClass:    'filter class' prefix=[Prefix] '.' class=ID;
AddConstraint:  'add constraint' constraint=ID value=EString;
FilterConstraint:   'filter constraint' constraint=EString;
```

Fig. 2. Grammar of our metamodel extension textual DSL

Based on the same small example than introduced at the end of Section 2, Figure 3 shows a sample metamodel extension illustrating the defined concrete syntax.

```
//Extension to transform DoDAF into NAF
define DoDAFextension extending DoDAF:dodaf{
    modify class dodaf.SystemFunctionSymbol {
        add property FunctionType type String
    }
    modify class dodaf.DataFlowSymbol{
        filter property SASymHasFromArrow
        filter property SASymHasToArrow
    }
}
```

Fig. 3. Example of a metamodel extension definition using our textual DSL

4 Architecture of a Metamodel Extension Mechanism

Once extensions are defined, we need to provide a modeling infrastructure able to understand and deploy them as part of a normal modeling process. This mainly includes (de)activating the use of extensions for specific models, and eventually storing the extension data to be reused in the future. This section presents such an infrastructure for the Eclipse/EMF framework.

4.1 Expected Characteristics

There are different ways to implement a metamodel extension mechanism (cf. Section 6). However, we believe such a mechanism should comply with the following list of characteristics, as determined mainly by the industrial partners in MoNoGe according to their actual needs:

- **Genericity.** The extension approach cannot be linked to a particular metamodel, tool or implementing framework. Relying on the same base mechanism, metamodel extensions can be defined on all metamodels and should be exchangeable between different modeling environments.
- **Non-intrusiveness.** Defined extensions should not directly modify original metamodels but rather complement them in an external manner. Thus, tools relying on these metamodels do not need to be deeply modified when their metamodels are extended.
- **Persistence and Interoperability.** Extensions should be specified, stored and shared in a user-comprehensive format, but also be easily machine-readable for reusability purposes. For separation of concerns (cf. also Non-intrusiveness), they should be persisted separately from metamodels.

- **Compatibility/conformance preserving.** Models should not be altered when extensions are defined on their respective metamodels: prior metamodel conformance should always be preserved. Backward conformance is also interesting: models that conform to a given extension could "forget" elements brought by this extension (e.g. default values could be used).
- **Transparency.** From user and tooling perspectives, an extended metamodel should be presented and manipulated as any regular metamodel. Models can conform to extended metamodels and dedicated tooling can directly rely on them.
- **Dynamicity and synchronization.** Metamodel extensions can be applied and removed. Corresponding models and tooling should be able to react/adapt accordingly in order to preserve consistency and usability (notably concerning compatibility and conformance).
- **Runtime computation.** (Parts of) Models conforming to an extended metamodel could be computed at runtime, i.e. from predefined expressions at extension-level (e.g. queries on the original metamodel). Related models and tooling should reflect the result of such computations.

4.2 An Eclipse/EMF Implementation

The proposed architecture relies on several existing technologies, reused and/or refined when needed, from the lively open source ecosystem around Eclipse and its well-known Eclipse Modeling Framework.

Our Eclipse/EMF implementation first comprises a dedicated parser and editor for the textual DSL (based on Xtext[2]) so that users can create their own metamodel extensions at development time. These extensions are then managed and processed using the architecture shown in Figure 4.

A *Base Operators API* consumes, in addition to the original metamodel, the DSL model generated by Xtext from the user textual definitions of the extension. Thanks to an ATL[3] model transformation, this component produces the appropriate data required by the *Virtualization API* to realize the metamodel extension and corresponding model.

There are different options for linking (meta)models together (cf. Section 6). In our implementation, we rely on model virtualization techniques to interconnect (meta)models together transparently on an on-demand basis. A virtual (meta)model is a (meta)model that do not hold concrete data but rather kind of proxies to original (meta)models, making it relevant in a lightweight metamodel extension context. As already providing virtualization capabilities, we adapted EMF Views[4] (a refinement of Virtual EMF [1]) to implement the required *Virtualization API* supporting the previously introduced extension operators. Thus, "virtual" extended metamodels and models are realized automatically by this API using the original (meta)models and complementary information computed

[2] https://eclipse.org/Xtext
[3] https://eclipse.org/atl
[4] http://atlanmod.github.io/emfviews

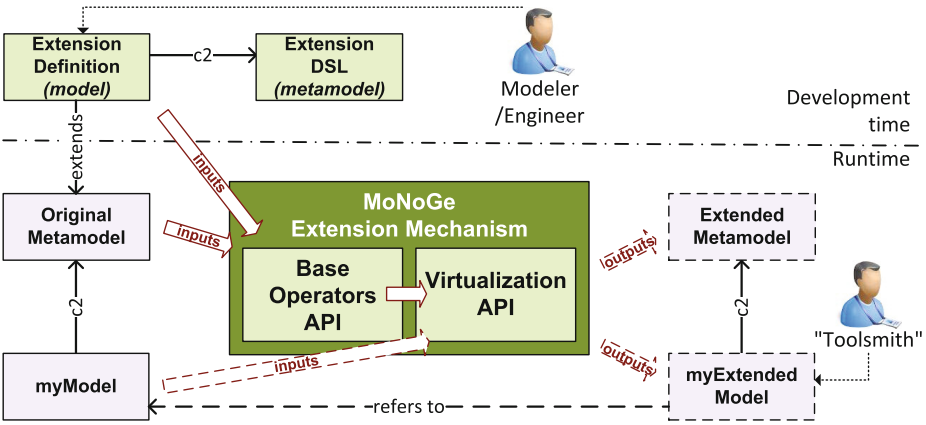

Fig. 4. Overall architecture for the Eclipse-EMF implementation

by the *Base Operators API*. For conformance reasons, and in case deriving from already existing models (i.e. prior to metamodel extension), "virtual" models may need to be completed at runtime (e.g. with some default values) according to the applied extension operators at metamodel-level (e.g. when a new property is added). Some initial support is provided via the use of ECL[5] as an automated matching engine in EMF Views. However, this has not been extensively tested so far in the current version. It can also be noted that constraints on metamodels are expressed using OCL[6].

Interestingly, such extended (meta)models can be manipulated as any EMF (meta)models in Eclipse, by other existing EMF-based technologies relying on the standard EMF model handling API or by both kinds of users (cf. Figure 4). Source code and screencasts of the current implementation are available online[7].

The described architecture and Eclipse/EMF implementation globally satisfy the expected characteristics, as introduced in Section 4.1. It is *generic* as extensions can be defined and then applied on top of any metamodel. Keeping the DSL tooling independent from the other components makes the overall extension mechanism even more generic, as defined extensions can be reused by different modeling environments. The proposed solution is also *interoperable* because extension declarations are *persisted* separately from original metamodels and thus can be shared easily between various modeling tools using the same base extension mechanism (cf. Section 5). The EMF Views *non-intrusive* and *transparent* approach, as well as its extensible architecture, made it a natural good candidate for our extension mechanism and offers concrete support to these important properties. *Synchronization* is ensured because the "virtual" extended (meta)models simply hold proxies to the real data actually contained in different

[5] http://eclipse.org/epsilon/doc/ecl
[6] http://wiki.eclipse.org/OCL
[7] https://github.com/atlanmod/monoge

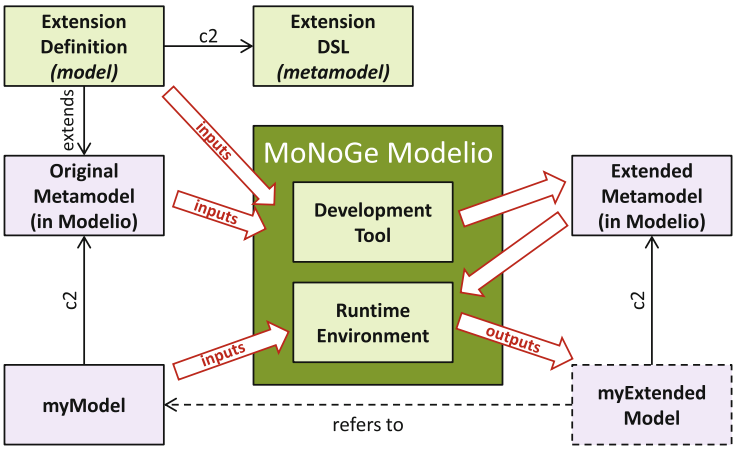

Fig. 5. Overall architecture for the Modelio implementation

(meta)models. Moreover, *compatibility* and *conformance* are also preserved as original metamodels are not actually modified. Finally, partial *runtime* support is also available via the use of an automated matching engine connected to EMF Views.

5 An Alternative Compliant Solution

Our textual DSL for specifying metamodel extensions is independent from any specific modeling tooling, framework or environment. This way, related metamodel extension mechanisms can be implemented on different technological platforms while still exchanging extension definitions based on the same proposed DSL. This is an important validation requirement from the MoNoGe industrial project's perspective. Thus, in addition to the Eclipse/EMF architecture and implementation presented before, this section briefly describes an alternative but DSL-compliant solution being integrated within the Modelio environment[8].

In contrast with the Eclipse/EMF solution, the Modelio-based one relies on a more generative approach (thus partially affecting some of the characteristics introduced in Section 4.1). There are two very distinct phases in the Modelio implementation, as summarized in Figure 5.

Firstly, there is a so-called *development phase* where the extension declaration is processed and transformed into a UML Class model, that is then transformed into a Java implementation model. This later is further processed by a Java code generator to produce a corresponding metamodel extension Jar (using the Modelio internal module mechanism) to be loaded in Modelio. Thus, the Modelio-based solution is able to consume metamodel extensions defined using the previously introduced DSL. Importantly, this solution also relies on

[8] http://www.modelio.org

the same base extension operators (excepting the MODIFY operations which will be supported in later developments).

Secondly, there is a *runtime phase* where the packaged metamodel extension is actually deployed within Modelio. This results in the modification of the Modelio original metamodel with content from the deployed extension, thus forming the extended metamodel. Any model that conforms to an extended metamodel can be imported, seen and used in the Modelio environment. The possible dynamic loading/unloading of such metamodel extension modules is currently being evaluated (as impacting more deeply the Modelio application's core).

6 Related Work

We compare here our work with other existing metamodel extension approaches, or solutions that can be applied in this context even if originally designed with a different purpose in mind.

A first group of related work is the one of metamodel evolution approaches. Metamodel evolution consists in supporting metamodel changes and their impact on related models, transformations, etc. An evolution can be perceived as an extension but, in an evolution context, the old (original) metamodel is generally abandoned and all the effort is put on adapting related artifacts to the new metamodel. Several approaches have been proposed to semi-automate the process concerning model migration [17], transformation migration [4] or DSL migration [2]. We aim to avoid these complex migration processes and make the two versions of the metamodel (and related models) coexist.

Metamodel extensions have also been addressed via the the concept of profiles, starting with the case of the well-known UML Profile mechanism [14] or its generalization to an EMF/Ecore context [12]. Following the same underlying principles, (meta)model decoration/annotation approaches [10] have also been used in an extension context, to represent usage-specific information for instance. However, both kinds of approaches have a limited expressivity as they are mainly restricted to adding complementary information or metadata. As presented before, we intend to address a wider range of possible extension operations.

The proposed extension mechanism can be also related to model composition techniques that may present similar operators to manipulate the models to be composed [3,5,7,9,16,18]. Model composition can be defined as the creation of a single model by merging elements coming from several ones [15]. Main problems then concern the synchronization between original and resulting models, as well as scalability issues regarding the needed memory and time to actually perform the merge. In our case, the "new" and "old" models are the same (only the extensions can be kept separated) and therefore our solution does not suffer from these problems.

In addition to these approaches, runtime-oriented solutions have been proposed such as EMF Facet[9] that allows (meta)model extension by runtime

[9] http://eclipse.org/facet

instantiation of additional concepts, attributes or references (computed from queries defined at metamodel-level). Nevertheless, EMF Facet can only manage derived information and no new "materialized" data can be part of the extension.

As explained in Section 4, the proposed MoNoGe solution intends to combine the best of different existing approaches. While the extension declaration is actually persisted via a DSL and can include the three main types of extension operations, the extended metamodel and related models are concretely realized by relying on a model virtualization mechanism.

7 Conclusion

We have proposed a lightweight metamodel extension mechanism, based on a textual DSL for specifying metamodel extensions. We have also described two alternative implementations, based on Eclipse/EMF and Modelio respectively, that concretely enable them. Our main objective is to improve the agility of modeling frameworks by allowing them to be more flexible and adaptable to changes on the metamodels they provide support for. The results obtained so far, according to our industrial partners in the MoNoGe project, are quite promising in terms of capabilities such as genericity, compatibility preservation, non-intrusiveness, transparency or dynamicity.

However, the already available DSL and mechanism have to be stressed-out in more contexts. First, we plan to explore with a couple of other concrete syntaxes (including a graphical one) that may be easier to use for some different user profiles. Moreover, some basic validation support for the defined extensions is required, e.g. to make sure that a set of extensions applied on a same metamodel (or extension of metamodel) is coherent. Another interesting aspect would be to build more elaborated metamodel extension algebra(s) by combining the proposed based operators, and to tackle related issues such as complex changes detection [8], consistency management, etc. In addition, the potential use of our extension approach within other already existing EMF-based tools (such as Papyrus for instance) could be explored. We could also provide general feedback to the ongoing OMG MEF standardization process whenever relevant.

Acknowledgments. The presented work is co-funded by the MoNoGe collaborative project (french FUI 15). We would also like to thank Juan David Villa Calle for the important amount of work performed on improving the EMF Views tooling in the past couple of years.

References

1. Clasen, C., Jouault, F., Cabot, J.: VirtualEMF: a model virtualization tool. In: De Troyer, O., Bauzer Medeiros, C., Billen, R., Hallot, P., Simitsis, A., Van Mingroot, H. (eds.) ER Workshops 2011. LNCS, vol. 6999, pp. 332–335. Springer, Heidelberg (2011)
2. Di Ruscio, D., Lämmel, R., Pierantonio, A.: Automated Co-evolution of GMF Editor Models, pp. 143–162 (2010). CoRR,abs/1006.5761

3. Didonet Del Fabro, M., Bézivin, J., Valduriez, P.: Weaving models with the eclipse AMW plugin. In: Eclipse Modeling Symposium, Eclipse Summit Europe (2006)
4. Garcés, K., Vara, J.M., Jouault, F., Marcos, E.: Adapting Transformations to Metamodel Changes via External Transformation Composition. Software & Systems Modeling **13**(2), 789–806 (2014)
5. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On language-independent model modularisation. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on Aspect-Oriented Software Development VI. LNCS, vol. 5560, pp. 39–82. Springer, Heidelberg (2009)
6. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: 33rd ICSE, 2011, pp. 633–642. IEEE, May 2011
7. Jayaraman, P., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007)
8. Khelladi, D.E., Hebig, R., Bendraou, R., Robin, J., Gervais, M.-P.: Detecting complex changes during metamodel evolution. In: Zdravkovic, J., Kirikova, M., Johannesson, P. (eds.) CAiSE 2015. LNCS, vol. 9097, pp. 263–278. Springer, Heidelberg (2015)
9. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Merging models with the epsilon merging language (EML). In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 215–229. Springer, Heidelberg (2006)
10. Kolovos, D.S., Rose, L.M., Drivalos Matragkas, N., Paige, R.F., Polack, F.A.C., Fernandes, K.J.: Constructing and navigating non-invasive model decorations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 138–152. Springer, Heidelberg (2010)
11. Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schoenboeck, J., Schwinger, W., Wimmer, M.: Systematic Co-evolution of OCL expressions. In: 11th APCCM), vol. 27, p. 30 (2015)
12. Langer, P., Wieland, K., Wimmer, M., Cabot, J.: EMF Profiles: A Lightweight Extension Approach for EMF Models. Journal of Object Technology **11**(1), 1–29 (2012)
13. OMG. Metamodel Extension Facility (MEF) RFP (2011). http://www.omg.org/cgi-bin/doc.cgi?ad/2011-6-22. (Accessed March-2015)
14. OMG. Unified Modeling Language (UML) (2011). http://www.omg.org/spec/UML/2.4.1/. (Accessed March-2015)
15. Pottinger, R.A., Bernstein, P.A.: Merging models based on given correspondences. In: 29th VLDB 2003, pp. 862–873. Morgan Kaufmann, San Fransisco, September 2003
16. Reddy, R., France, R., Ghosh, S., Fleurey, F., Baudry, B.: Model composition - a signature-based approach. In: Aspect Oriented Modeling (AOM) Workshop (2005)
17. Rose, L.M., Herrmannsdoerfer, M., Williams, J.R., Kolovos, D.S., Garcés, K., Paige, R.F., Polack, F.A.C.: A Comparison of model migration tools. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 61–75. Springer, Heidelberg (2010)
18. Sabetzadeh, M., Easterbrook, S.: View Merging in the Presence of Incompleteness and Inconsistency. Requirements Engineering **11**(3), 174–193 (2006)
19. Völter, M.: MD*/DSL best practices (version 2.0), April 2011