A Proxy Approach to External Model Repository Integration in Eclipse EMF Infrastructure

Oskars Vilitis[1], Audris Kalnins

Institute of Mathematics and Computer Science, University of Latvia, 29 Raina boulevard, Riga, LV-1459, Latvia, ph.: (+371) 6 7224 363

Oskars.Vilitis@gmail.com, Audris.Kalnins@mii.lu.lv

Abstract. Eclipse Modeling Framework (EMF) has become a very popular environment for the implementation of a structured model, which has resulted in increased need for integration with other modeling environments. In this paper, we describe a method for direct integration of external model repositories in the EMF infrastructure. Our approach is based on the application of the proxy pattern to extend the functionality of EMF base objects and to provide a runtime synchronization of the model data with the repository.
The approach presented here allows existing applications to interchange model data seamlessly with EMF, thus giving access to the services offered by EMF technologies. On the other hand, EMF-based applications can benefit from the services provided by external repositories and applications (for example, efficient model-to-model transformation implementation) without the need to adjust the application code. Applicability of the solution introduced is analyzed at the end of the paper. The approach has been successfully applied in the metamodel-based tool building platform METAclipse.

Keywords. Eclipse, EMF, Integration, Model Repository, MDA, MDE, MDSD, Real-Time, Synchronization

1 Introduction

Model-Driven Architecture (MDA) [1] concepts have been around for quite a while now and currently many software platforms, frameworks, tools, and applications work with models and model repositories. Various tools and frameworks provide a wide range of services for operation with models: transformation languages (both model-to-model and model-to-text), model querying and validation facilities, different persistence solutions, code generation, etc.

1.1 Features of the EMF Family Technologies

One of the most popular modeling frameworks currently is Eclipse Modeling Framework [2] (EMF). Having started as a MOF [3] implementation, EMF has gone its own way and relies on its own metamodel ECore, which is very similar to EMOF (Essential MOF), a subset of the MOF. In fact, there are just some small, mostly

[1] supported partially by ESF (European Social Fund), project 2004/0001/VPD1/ESF/PIAA/04/NP/3.2.3.1/0001/0001/0063

notational, differences between those two. By bringing an efficient implementation of the core subset of the MOF API to the developers, EMF has quickly become a de-facto standard of model handling in Eclipse-based tools, and is also becoming popular outside Eclipse in standalone applications.

EMF is widely used as a tool for the implementation of a structured model. However, this is not the only functionality the family of EMF technologies has to offer. By introducing an efficient and standardized approach to model handling, EMF has promoted the evolution of various EMF-based projects supporting the model-driven engineering process (such as EMF Query [2], EMF Transaction [2], OCL implementation [4], validation component [2], and various transformation language implementations, e.g., ATLAS Transformation Language ATL [5] for model-to-model transformations and JET for model-to-text transformations).

Other Eclipse frameworks and tools are built to operate on EMF models, allowing rich graphical editing of the models (like GMF [6]). There are a number of practical Eclipse applications built that even further extend EMF model handling possibilities and applicability of EMF models. Among those are various persistency and O/R mapping solutions (such as CDO and Teneo [7]), the MDSD supporting framework openArchitectureWare, and even commercial development and design tools like IBM Rational Software Architect, etc. The stack of technologies in the EMF family and the supply of tools operating on EMF models are growing continuously.

1.2 **Motivation for the External Repository Integration in EMF**

Having such a rich set of services available, EMF is an appealing environment for model handling. Existing applications can benefit from allowing their models to be transferred to EMF and back. For example, such interoperability could add missing features to existing model environments when needed, enabling XMI model serialization (provided by EMF as the default serialization mechanism), validation of the model against a defined rule set, code generation functionality or model-to-model transformations, the possibility to develop graphical model editors, etc.

Not only existing applications can benefit from integration with EMF. Another benefit of the external repository integration in EMF is the possibility to use additional services from EMF-based tools. In fact, this was the main reason why EMF proxy classes were developed at the Institute of Mathematics and Computer Science at the University of Latvia. We integrated our own model repository MIIREP [8] with EMF, so that our EMF-based tool METAclipse [9] (metamodel- and transformation-based graphical DSL-editor building platform) could gain access to our model-to-model transformation engine and use our transformation language MOLA [10, 11]. These transformations are compiled to C++ code and work on the MIIREP repository, which is specialized particularly for efficient execution of the operations needed by transformations. By this integration we gained the performance needed for transformations to work on huge models in a very efficient way.

The alternative to integration would have been transferring the MOLA transformation language to Java, so that it worked on EMF objects directly. This, however, would have meant massive work on a new implementation of MOLA and would have required much more effort. Furthermore, the transfer of MOLA itself would not have guaranteed the possibility to measure up with the efficiency of C++

implementation. Also, it would not have been sufficient to have only model import / export functionality, as in the case of METAclipse interaction with the repository is very dynamic. Each instance of user interaction with METAclipse results in the execution of some transformation, so a very rapid access and change of repository objects is required from both the Eclipse editor and the MOLA transformations. Therefore, integration of the existing repository was the most reasonable choice.

Another motivation for integration worth mentioning is the possibility to unite the EMF with different other model-handling frameworks, such as MDR [12], MS DSL [13], Generic Modeling Environment [14], and Fujaba [15]. All these frameworks are meta-model-based, and their meta-meta-models provide similar capabilities to EMOF. They all can handle models similar to EMF, and each provides distinctive features for model handling. For example, GME provides advanced facilities for building model-based simulators and debuggers, while MS DSL provides easy integration with Microsoft technologies. The features of each framework can turn out to be useful for EMF models. There are therefore good reasons for uniting them. There already exists such an attempt: Eclipse project GEMS [16] binds the GME to EMF.

1.3 **Integration Solutions**

In general, tool integration problem has been a topic of discussions and publications already for a long time. A survey [17] shows that the tool integration topic is very wide. Most of the covered papers discuss the integration problem generically. This paper, however, concentrates on the integration solution for a specific technology, namely EMF, fulfilling more stringent requirements than in the general case.

There already are some examples of model interchange between EMF and other technologies, based on the import and export of models. In the simplest cases it is done through some format supported by both EMF and the external repository (such as XMI), but others make use of the native repository APIs. Some of these are the integration of EMF with ARIS [18], MS/DSL [19] tools, GME [20], and EMT [21].

There is one significant problem with the import / export approach. In this process, the model is first exported, resulting in a copy of the model. Then changes to the copy are made, after which the modified copy is imported back in the model. If the whole model is transferred, this process is not complicated. However, usually models are big and it is inefficient to transfer them in their entirety. Normally, only a sub-set of the entire model needs to be exported for external modification. In this case, a huge problem is the merging of the transformed sub-model back into the original model. The main problem is that there can be references from the unmodified parts of the model to some parts of the model that have been deleted or changed. These references need to be traced and modified; sometimes perhaps redirected to newly created elements. This is not an easy task and requires knowledge of both the original and the modified models, and sometimes even about model transformation logic.

Another characteristic of the import / export approach is that it can support the integration needs only if the model transfer from one technical space to another is relatively infrequent (for the batch processing of the models). If more rapid model data interchange is needed, other integration solutions should be considered.

In this paper we describe another approach to model data transfer, namely, direct integration of the external repositories with the EMF environment. In addition to the

features provided by import / export and bridging of the technical spaces, the approach presented here allows lazy data loading and synchronization (only the relevant data will get transferred to and from EMF) and dynamic model integration in EMF (operations on external models can be carried out at runtime). The solution also does not create any problems with merging, as modifications of the model are carried out directly in the original model, and no export, import and merging are needed.

The main idea behind our approach is to alter the original implementations of the core EMF objects in such a way that they start acting as proxies to the external repository, and each operation on the EMF model is redirected to the corresponding operation(s) on the external repository. Any changes done to the model at the runtime outside the EMF are properly notified to listeners through the EMF notification API. More detailed description of our solution is presented in section 4.

Summing up, the presented integration approach allows existing applications to gain the benefits of the services offered by the EMF tools and vice versa. For example, in the context of transformation languages, applications not offering transformation languages can use the transformation languages operating on EMF models. On the other hand, EMF tools can use the transformation languages offered by external applications in order to gain efficiency and improve performance.

In further sections of this paper we will describe the proxy approach to repository integration in more detail. Section 2 will introduce the objectives of the integration approach proposed. Section 3 will demonstrate how the Proxy pattern can be applied in order to achieve the integration goals. In section 4 we will give more details on integration implementation and in section 5 we will present some exemplary applications of the offered integration approach.

2 Objectives of the Integration

The goal of our proposed repository integration solution is to provide a bridge between the external repository and EMF that would possess the same characteristics as import / export solutions (possibility to transfer the model data from the external repository to EMF and back), but at the same time would provide more sophisticated features, such as the ability to carry out the transfer of model data dynamically, as the models are changing during the runtime. Here we focus only on cases, where the meta-metamodel (M3) concepts of the application can be easily mapped to the meta-metamodel of EMF, namely ECore (otherwise, non-trivial model transformations would be required).

There has to be a possibility to synchronize models between the external repository and EMF, propagating changes made on either side to the other in real time. It must be possible to carry out synchronization in both directions. If the change is made to the synchronized model directly in the external repository by some external application, it must be transferred to the EMF and proper EMF notifications have to be called. And vice versa, if the change is made to the model by EMF, it must be transferred also to the external repository.

That being said, it must be noted that currently no objective has been established to allow simultaneous changing of models by external applications and by EMF – the

presented solution presumes that if there are changes on both sides, they are always sequential rather than parallel, and no concurrency is supported.

Another aspect to be considered is that we do not want to impose any additional requirements to the applications using the EMF code. This means that the EMF interface has to remain intact and applications already using the EMF classes should not have to change significantly if it was required for them to synchronize their model data with an external repository.

3 Applying the Proxy Pattern

Taking into account the aforementioned goals, an appropriate method for the implementation of external repository integration with EMF is the proxy pattern. The basic idea of this pattern is to provide a façade for another object in order to control the access to it. As Design Patterns book [22] suggests, some of the most typical cases when the proxy pattern is used are when:

1. it is necessary to provide a local representation of a remote object (*remote proxy*);
2. objects are expensive to create and should be created on-demand (*virtual proxy*).

Relating this to our goals, we want the EMF to act as a façade to the external repository and delegate the calls to the external repository API. To be more specific, what we need is a remote proxy with the features of the virtual proxy. The utilization of the remote proxy is obvious. The virtual proxy features are needed, because models tend to be very big, making it desirable to transfer to EMF only those objects that are really needed. Additionally, for increased performance, the caching mechanism needs to be implemented so that subsequent access to the object properties would result just in a single call to the repository API functions. See Fig. 1 for the class diagram of the proxy pattern adjusted to our needs.

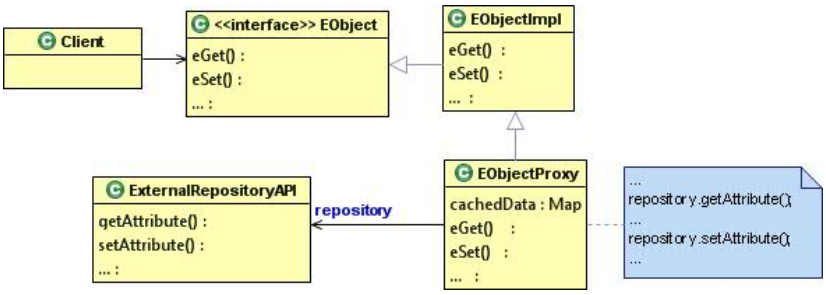

Fig. 1. Structure of the proxy pattern
applied for the bridging of EMF and the external repository

The figure depicts only high-level structural elements. The client is any application using proxied EMF objects. It is accessing the EMF object interfaces (only the root interface EObject is displayed with basic representative methods eGet and eSet, but it could be any sub-interface of EObject in ECore metamodel or any generated EMF class interface). As EMF has a top-level object defined in its metamodel, namely

EObject, it is enough to provide the proxy implementation for this object to get the proxy functionality spread throughout all EMF metamodel implementation classes.

Extension EObjectProxy of the EMF EObject interface implementation EObjectImpl acts as a proxy to the external repository API. This class implements the "remote" and "virtual" features of the proxy pattern by delegating the calls to the API of the external repository and providing a cachedData map that is consulted before calling the actual repository API functions.

ExternalRepositoryAPI class is the one being proxied. The difference between the variation shown in Fig. 1 and the original proxy pattern is that the external repository API does not implement the same interface as the proxy. It is possible that some calls to the EObject will result in multiple calls to ExternalRepositoryAPI, possibly even with some model transformation involved. However, external repository API cannot be absolutely arbitrary. It must operate with the same concepts as EMF, i.e. its capabilities must be isomorphous to EMOF. Therefore, it can be said that it "isomorphically" still implements the same interface as proxy.

By applying the proxy pattern we can solve the synchronization problem in one direction—from EMF to the repository. However, changes done in the external repository by external applications must be transferred back to the EMF, as both sides can actively change the models. For this reason, an additional change notification mechanism is needed. Such mechanism will be described in the next section (subsection 4.2) together with technical details of application of the proxy pattern.

4 Implementation of the Proxy for EMF: "Wise" Objects

Having established how to apply the proxy pattern, we can proceed to the technical details of the actual implementation of the proxy to the external repository. We will be giving the description based on the experience we had while integrating our repository MIIREP [8] with our EMF-based graphical model editing tool METAclipse [9], where the proxy approach to integration is already successfully implemented and working (see section 5 for more information about METAclipse). The actual implementation of the EMF proxy will differ from repository to repository, as there will be differences in repository APIs. Still the concepts of the integration will remain the same. Technical description of METAclipse, including some specific details about MIIREP and EMF integration is given in [23].

In case of integration of MIIREP in EMF, changes to the model can occur as a result of both model transformations working directly with the repository API and the METAclipse tool working with the EMF representation of the model. Therefore, both kinds of synchronization are involved—from the repository to EMF and vice versa.

4.1 "Wise" Objects as an EMF Extension

EMF ECore metamodel classes (ECore base classes) define the class hierarchy that forms the basis for the Java runtime. All EMF runtime classes generated for a particular metamodel extend these base classes. ECore base classes provide all the functionality to the generated classes and allow using them in EMF infrastructure by providing all the EMF framework features. Consequently, base classes are the best

place where repository synchronization should be implemented and, as it has already been roughly sketched in section 3, EMF proxies are implemented as an extension of the original EMF ECore objects, providing an alternative EMF runtime.

New proxy objects conform to EMF interfaces and externally look like normal EMF objects, but internally do all the synchronization with the repository. These objects were named “wise” objects, as they show certain “intelligence”: though from the interface perspective they look like normal EMF objects and support all EMF framework operations, internally they know when and how it is necessary to read or write information to the repository. For EMF tools “wise” objects can be considered a second level of repository abstraction, which introduces the caching mechanism, conforms to the EMF object interfaces and uses first level abstraction—repository interface—to read and write data to the repository.

Base ECore classes were extended and a set of “wise” object base classes was defined (see Fig. 2). By analogy to ECore classes, base “wise” object classes, together with some helper classes comprising the whole “wise” object concept, were called WCore. In WCore, the methods inherited from ECore for accessing the properties are extended with functionality of reading and writing data from and to the repository. For increased performance, “wise” objects keep track of the state of every object property and cache the data from the repository in the object instance, so that subsequent reads of the same property would require just a single repository access.

The fact that the parent of all ECore classes is a single class—EObject (see [2] for complete ECore structure)—simplified the extension of ECore. For “wise” object needs it was enough to extend just two ECore classes, EObject and EFactory, with the corresponding WObject and WFactory classes. WObject contains all the caching and synchronization logic and, as it is the superclass of all the other framework classes, the logic is available all across the framework. The WFactory extension of the factory class is needed, as some initialization of the “wise” object on its creation is required.

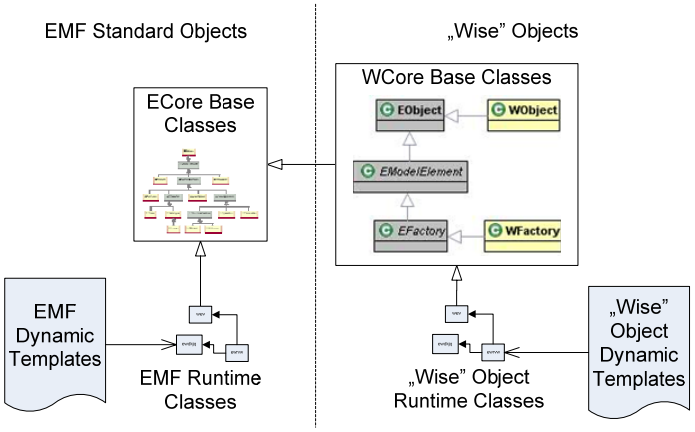

Fig. 2. “Wise” object dependencies

To put the WCore classes in action also for the generated code, the EMF generator had to be extended so that it produced “wise” objects extending WCore base classes. The EMF framework uses so-called dynamic code templates during the generation process of the runtime classes. The EMF generator reads the serialized form of the

metamodel and then, using the set of templates, generates the runtime classes (see Fig. 2). Default templates producing EMF runtime classes were extended so that they would generate the code using WCore instead of ECore.

The complete set of classes comprising the WCore can be seen in Fig. 3. The above-mentioned extension of getter and setter methods of ECore is divided into two classes. Reading of the attributes from the repository was easiest to implement in the WObjectImpl class itself, in the inherited getter methods. Writing the attributes, however, was easier to move to a separate class WObjectChangeObserver, which implements the EMF change listener and is attached to every instance of WObject. The change observer listens to any changes done to the WObject from the EMF side and whenever one occurs, writes the data to the repository.

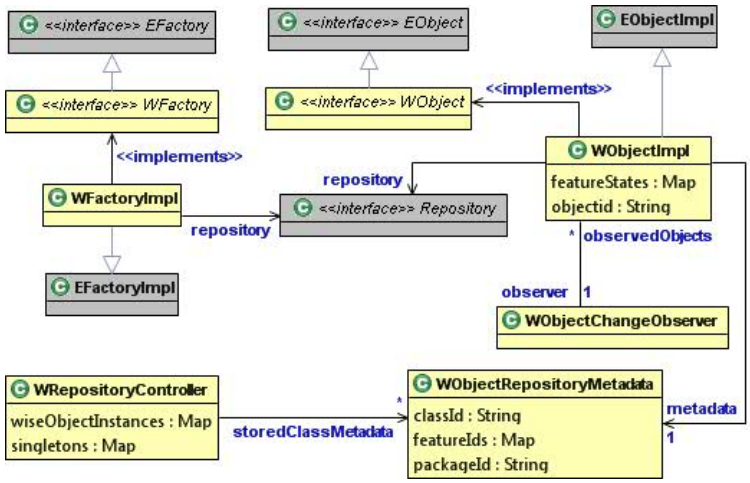

Fig. 3. WCore class diagram

To be able to read and write the repository data, “wise” objects need to have a possibility to map the classes, attributes and associations to the corresponding repository objects. Such mapping can be defined only at the M2 layer and thus it is needed to have the WCore class and feature mapping to the repository metadata at the M2 layer. As it is inefficient to read these mappings every time an object is accessed, class metadata mappings are cached. The WRepositoryMetadata object represents the class metadata. The map of WCore class to repository metadata mappings is held in the WRepositoryController object.

The two objects directly responsible for the synchronization of the model in the repository and in its representation in EMF (WObject and WObjectChangeObserver) act on the events of reading or changing the model information through the EMF API. When any operation on the model is performed, it translates the EMF API call to the corresponding call(s) to the repository API. It is easy to do this if the repository relies on a metamodel that is very close to the EMOF. However, the less the repository API resembles EMOF, the harder it becomes to map the EMF calls to it and the more intelligent transformations are necessary.

4.2 Repository Change Notification

Extending the ECore base classes covers the synchronization needs only from the EMF perspective, i.e., this part of the solution is applicable only when changes to the model are made from the environment working with EMF classes (wise objects). However, model changes can happen also on the other side (in the case of the MIIREP integration in METAclipse, the most intense changes to the model are made by the transformations in the repository directly). So, besides the proxy pattern applied to EMF objects, another missing piece is change notification back from the repository, which would trigger the EMF change events for all objects that have been changed.

The change notification is not a trivial task, as it is also constrained with tight performance requirements. It is very inefficient to detect the changes when they have already been made, as it requires inspection of all object instances in the repository. This means that support from the side of the repository or the tool performing the changes is required in order to implement efficient change notification.

In WCore, the WRepositoryController class (see Fig. 3) takes care of the repository model change tracking. There, a special method is defined for change detection, which has to be invoked after each change made to the model directly at the repository (what is calling this method depends on how the integration of the external repository is used). The implementation of the WRepositoryController, however, is strongly dependent on the possibilities offered by the repository being integrated.

For each repository the change tracking mechanism is different, as the possibilities of detecting changes differ from one to another. The worst case is if the external change source is making unpredictable changes in the repository and the repository itself does not provide any change tracking mechanism. In this case there are only two options: introduce a layer between the external tool and the repository that will implement the change tracking mechanism or, if the performance requirements allow it, do a full re-scan of all model elements residing in the repository and detect which elements and how have been changed. One possibility for the implementation of the change-tracking layer is to use aspect-oriented programming (AOP) in order to execute the change tracking code before or after repository API function calls.

A slightly better situation occurs when some kind of an algorithm exists that limits the number of the model elements to consider while detecting the changes. The best scenario, however, is when it is possible to rely on a repository-native service that allows explicit detection or monitoring of the changes by either defining the listeners on the repository objects or calling some method that returns the set of the changes.

To support the various scenarios of how the change detection can happen, WRepositoryController defines an abstract change notification method returning lists of the changed or deleted objects. Functionality of tracking changes is left to the implementations for individual repositories. When changed or deleted object lists are read from the repository, WRepositoryController issues the corresponding EMF notifications and the modified features of the object instances that have changed are tagged “dirty,” so that they are once again read from the repository when accessed instead of using the cached values from the WObject instances.

In case of the repository and transformations currently used in METAclipse, it was very easy to track object deletions, as the MIIREP repository itself has the

functionality to track such changes. However, the tracking of the changes to the existing objects had to be incorporated in the transformations. Each transformation is responsible for maintaining the lists of the changes to be returned to the WRepositoryController.

5 Applicability

As already mentioned, the main force that drove the development of the approach presented was the necessity for the use of an external repository in the METAclipse tool. This demonstrates a case when the discussed integration approach is applied to make extra features provided by an external repository available in EMF. METAclipse, presented in [9], is a metamodel-based graphical tool building platform for the development of domain-specific language (DSL) editors. The tool provides a platform for building rich DSL editors working on EMF models.

METAclipse editors are driven by model transformations that are executed on every user action in the editor (even a mouse click on some model element invokes a transformation). This and the fact that models being edited with the DSL editors tend to become fairly large (even millions of instances) creates very high efficiency requirements to the transformation engine. For transformations to work efficiently, it is important to have an appropriate repository. EMF itself lacks the functionality required for efficient implementation of operations like pattern matching. Therefore, for an efficient transformation engine implementation, it is required to extend the EMF to add the missing functionality.

In a similar situation, Tiger project [24] team has chosen to redesign their graph transformation language AGG [25] and transfer it to EMF. In case of METAclipse, we already had an efficient repository MIIREP [8], specialized for transformation languages and capable of handling huge models, and a stable and efficient transformation language MOLA [10, 11], working on this repository. It was more natural to integrate the named repository into EMF rather than redesign the MOLA language to work in the EMF environment and extend the functionality of EMF.

Another example, similar to the case of METAclipse, would be integration of the external simulation engine functionality (such as available in GME framework) into EMF. For example, if we use the graphical plugins of Eclipse for visualization and animation and external libraries for computation, there is a need of rapid model data interchange between the EMF and the external model storage.

We see that another applicability domain where our solution would be useful is for augmentation of the possibilities of the existing tools with the features provided by the EMF technology family. Papers [17], [19] and [20] demonstrate that there is a real need for such integration. Mentioned papers use the import / export approach with transformations involved in metamodel mapping from one technical space to other. This approach was natural for the problems addressed, as all three are examples of typical batch transfers of model data.

Things, however, get more complicated if there is a need to transfer only a part of the model and to merge the changes back. For example, if a MDSD transformation is applied to a sub-model, the results must be integrated in the common design model of a system. In this case some non-trivial reasoning is required in order to preserve the

integrity of the complete model. The approach described in this paper could be adapted to solve the problems of this use-case, and it seems to be the only reasonable solution. The only concern that has to be considered and affects the effort needed for implementation is the support of change tracking in the external repository.

It must be noted that the applicability of the proposed solution is constrained with the need for the external repository API to provide functionality that would cover all the capabilities of the EMOF. If the concepts behind the repository are not compatible with EMOF (meta-meta-models at M3 layer are not close enough), it is not possible to apply the presented approach. Also, there is no real need of using the introduced solution if there is no use for the runtime dynamic synchronization and lazy model handling, and all that is needed are some batch updates. In such cases it will probably be easier to implement the import / export features.

6 Conclusions

This paper proposes an alternative to the traditional (import / export-based) approach to external model repository integration in the EMF environment. The approach presented here integrates the external repositories directly, providing runtime model synchronization between EMF and repository model data. The discussed implementation extends the basic functionality of EMF objects by applying the proxy design pattern to carry out the synchronization tasks “behind the scenes”. The EMF interface is not changed, so that application logic does not have to be modified whenever it is needed to attach an external repository. The solution also limits the memory footprint of the loaded models through the use of lazy-loading.

The integration approach introduced in this paper can be applied only to model repositories that satisfy two basic requirements: their metamodels must be close to EMOF, and they must provide API capabilities similar to EMF. The paper does not provide an absolutely universal implementation that could fit all repositories. Because of the differences in the APIs of various repositories and variations in their capabilities, the presented approach has to be adjusted slightly differently for each of them. However, most of the implementation (a detailed description of which is given in sections 4.1 and 4.2) can be reused and does not have to change.

The typical application of the proposed solution would be for integration scenarios where it would be necessary to interchange data between the repository and EMF frequently and rapidly. The approach has been applied in practice to the metamodel-based transformation-driven modeling tool building platform METAclipse. In this platform, a highly efficient model repository that is specialized for the transformation language purposes is integrated with EMF. This provides the possibility to execute MOLA language transformations on the models visualized by EMF-technology-based tools.

There also exist other application possibilities for our approach. One such possibility could be MDSD transformation support in a more complicated environment, briefly sketched in section 5. This section also elaborates more on other possibilities of the applications.

References

1. OMG, Model Driven Architecture, http://www.omg.org/mda/
2. Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf/
3. OMG, Meta-Object Facility (MOF), http://www.omg.org/mof/
4. Model Development Tools (MDT) Project, http://www.eclipse.org/modeling/mdt/
5. Atlas Transformation Language (ATL) Project, http://www.eclipse.org/m2m/atl/
6. Graphical Modeling Framework (GMF) Project, http://www.eclipse.org/gmf/
7. Eclipse Modeling Framework Technology Project, http://www.eclipse.org/modeling/emft/
8. Barzdins, J., Barzdins, G., Balodis, R., Cerans, K., Kalnins, A., Opmanis, M., Podnieks, K.: Towards Semantic Latvia. Proceedings of Seventh International Baltic Conference on Databases and Information Systems, Communications, Vilnius, Lithuania, O. Vasileckas, J. Eder, A. Caplinskas (Eds.), Vilnius, Technika, 2006, pp. 203–218.
9. Kalnins, A., Vilitis, O., Celms, E., Kalnina, E., Sostaks, A., Barzdins, J.: Building Tools by Model Transformations in Eclipse. Proceedings of DSM'07 workshop of OOPSLA 2007, Montreal, Canada, Jyväskylä University Printing House, 2007, pp. 194–207.
10. Kalnins, A., Barzdins, J., Celms, E.: Model Transformation Language MOLA. Proceedings of MDAFA 2004, Vol. 3599, Springer LNCS, 2005, pp. 62–76.
11. UL IMCS, MOLA pages, http://mola.mii.lu.lv/
12. Metadata Repository (MDR), http://mdr.netbeans.org/
13. S. Cook, G. Jones, S. Kent and A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007.
14. Karsai G.: A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming, IEEE Computer Society Press, pp. 36-44, 1995.
15. Fujaba. Universitat Paderborn, Institut fur Informatik http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf
16. The Generic Eclipse Modeling System GEMS, http://www.eclipse.org/gmt/gems/
17. Tool Integration within Software Engineering Environments: An Annotated Bibliography, http://www.macs.hw.ac.uk:8080/techreps/build_table.jsp?id=0041
18. Kern, H., Kühne, S.: Model Interchange between ARIS and Eclipse EMF. Proceedings of DSM'07 workshop of OOPSLA 2007, Montreal, Canada, Jyväskylä University Printing House, 2007.
19. Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., Piers, W.: Bridging the MS/DSL Tools and the Eclipse Modeling Framework. Proceedings of the International Workshop on Software Factories at OOPSLA 2005, San Diego, California, USA, 2005.
20. Bézivin, J., Brunette, C., Chevrel, R., Jouault, F., Kurtev, I.: Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF). Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05, San Diego, California, USA, 2005.
21. Biermann, E., Ehrig, K., Koehler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. Proceedings of MoDELS'06, Genova, Italy, October 2006.
22. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston, MA, 1995.
23. Vilitis, O., Kalnins, A.: Technical Solutions for the Transformation-Driven Graphical Tool Building Platform METAclipse. Computer Science and Information Technologies, Acta Universitatis Latviensis, 2008.
24. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages using Tiger. Proceedings of GraBaTs'06, 2006, pp. 12.
25. Taentzer, G: AGG: A Graph Transformation Environment for Modeling and Validation of Software. AGTIVE'03, Vol. 3062, Springer LNCS, 2004.