The Grand Challenge of Scalability for Model Driven Engineering

Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York, UK.
(dkolovos,paige,fiona)@cs.york.ac.uk

Abstract. Scalability is particularly important for the adoption of Model Driven Engineering (MDE) in an industrial context. The current focus of research in MDE is on declarative languages for model management, and scalable mechanisms for persisting models (e.g., using databases). In this paper we claim that, instead, modularity and encapsulation in modelling languages should be the main focus. We justify this claim by demonstrating how these two principles apply to a related domain – code development – where the issue of scalability has been addressed to a much greater extent than in MDE.

1 Introduction

The adoption of MDE technologies in an industrial context involves significant benefits but also substantial risks. Benefits in terms of increased productivity, quality and reuse are easily foreseeable. On the other hand, the most important concerns raised of MDE are those of scalability [1], the cost of introducing MDE technologies to the development process (training, learning curve) and longevity of MDE tools and languages. To our perception, the latter two concerns (cost of induction and longevity) are not preventive for the adoption of MDE; however scalability is what is holding back a number of potential adopters.

2 Scalability in MDE

Large companies typically develop complex systems, which require proportionally large and complex models that form the basis of representation and reasoning. Moreover, development is typically carried out in a distributed context and involves many developers with different roles and responsibilities. In this context, typical exploratory questions from industrial parties interested in adopting MDE include the following:

1. *In our company we have huge models, of the order of tens of thousands of model elements. Can your tool/language support such models?*
2. *I would like to use model transformation. However, when I make a small change in my (huge) source model, it is important that the change is incrementally propagated to the target model; I don't want the entire target model to be regenerated every time.*

3. (similarly) *I would like to use code generation. However, when I make a small change in my (huge) model I don't want all the code to be regenerated.*
4. *In my company we have many developers and each manages only a specific part of the model. I would like each developer to be able to check out only a part of the model, edit it locally and then merge the changes into the master copy. The system should also let the developers know if their changes are in conflict with the rest of the model or with changes done by other developers.*

Instead of attempting to answer such questions directly, we find it useful to consider analogies with a proven and widely used environment that addresses those problems in a different – but highly relevant – domain. The domain is code development and the environment is the well known and widely used Eclipse Java Development Tools (JDT).

As a brief overview, JDT provide an environment in which developers can manage huge code-bases consisting of (tens of) thousands of Java source code files (concern 1). JDT supports incremental consistency checking and compilation (concerns 2,3) in the sense that when a developer changes the source code of a particular Java class, only that class and any other classes affected by the change – as opposed to all the classes in the project or the workspace – are re-validated and re-compiled. Finally, JDT is orthogonal to version control, collaborative development (concern 4), and multi-tasking tools such as CVS and SVN and Mylyn.

3 Managing Volume Increase

As models grow, tools that manage them, such as editors and transformation engines, must scale proportionally. A common concern often raised is that modelling frameworks such as EMF [2] and widely-used model management languages do not scale beyond a few tens of thousands of model elements per model. While this is a valid concern, it is also worth mentioning that the Java compiler does not allow Java methods the body of which exceed 64 KB, but in the code-development domain this is rarely a problem.

The reason for this asymmetry in perception is that in code development, including all the code of an application in a single method/file is considered – at least – bad practice. By contrast, in modelling it is deemed perfectly *reasonable* to store a model that contains thousands of elements in a single file. Also, it is *reasonable* that any part of the model can be hard-linked with an ID-based reference to any other part of the model.

To deal with the growing size of models and their applications, modelling frameworks such as EMF support lazy loading and there are even approaches, such as Teneo [3] and CDO [4], for persisting models in databases. Although useful in practice, such approaches appear to be temporary workarounds that attempt to compensate for the lack of encapsulation and modularity constructs in modelling languages. In our view, the issue to be addressed in the long run is not how to manage large monolithic models but how to separate them into

smaller modular and reusable models according to the well understood principles defined almost 40 years ago in [5], and similarly to the practices followed in code development.

4 Incrementality

In the MDE research community, incrementality in model management is sought mainly by means of purely declarative model transformation approaches [6, 7]. The hypothesis is that a purely declarative transformation can be analysed automatically to determine the effects of a change in the source model to the target model. Experience has demonstrated that incremental transformations are indeed possible but their application is limited to scenarios where the source and target languages are similar to each other, and the transformation does not involve complex calculations.

JDT achieves incrementality without using a declarative language for compiling Java source to bytecode; instead it uses Java which is an imperative language. The reason JDT can achieve incremental transformation lies mainly the design of Java itself. Unlike the majority of modelling languages, Java has a set of well-defined modularity and encapsulation rules that, in most cases, prevent changes from introducing extensive ripple effects.

But how does JDT know what is the scope of each change? The answer is simple: it is hard-coded (as opposed to being automatically derived by analysing the transformation). However, due to the modular design of the language, those cases are relatively few and the benefits delivered justify the choice to hard-code them. Also it is worth noting that the scope of the effect caused by a change is not related only to the change and the language but also to the intention of the transformation. For example, if instead of compiling the Java source code to bytecode we needed to generate a single HTML page that contained the current names of all the classes we would unavoidably need to re-visit all the classes (or use cached values obtained earlier).

5 Collaborative Development

As discussed in Section 2, a requirement for an MDE environment of industrial strength is to enable collaborative development of models. More specifically, it is expected that each developer should be able to check out an arbitrary part of the model, modify it and then commit the changes back to the master copy/repository. Again, the formulation of this requirement is driven by the current status which typically involves constructing and working with large monolithic models. With enhanced modularity and encapsulation, big models can be separated into smaller models which can then be managed using robust existing collaborative development tools such as CVS and SVN, augmented with model-specific version comparison and merging utilities such as EMF Compare [8]. Given the criticality

of version control systems in the business context, industrial users are particularly reluctant to switching to a new version control system[1]. Therefore, our view is that radically different solutions, such as dedicated model repositories, that do not build on an existing robust and proven basis are highly unlikely to be used in practice.

6 Modularity in Modelling Languages

The above clearly demonstrate the importance of modularity and encapsulation for achieving scalability in MDE. There are two aspects related to modularity in modelling: the design of the modelling language(s) used and the capabilities offered by the underlying modelling framework. In this section we briefly discuss how each of those aspects affect modularity and envision desirable capabilities of modelling frameworks towards this direction.

6.1 Language Design

With the advent of technologies such as EMF and GMF [9], implementing a new domain-specific modelling language and supporting graphical and textual editors is a straightforward process and many individuals and organizations have started defining custom modelling languages to harvest the advantages of the context-specific focus of DSLs. When designing a new modelling language, modularity must be a principal concern. The designers of the language must ensure that large models captured using the DSL can be separated into smaller models by providing appropriate model element *packaging* constructs. Such constructs may not be part of the domain and therefore they are not easily foreseeable. For example, when designing a DSL for modelling relational databases, it is quite common to neglect *packaging*, because relational databases are typically a flat list of tables. However, when using the language to design a database with hundreds of tables, being able to group them in conceptually coherent packages is highly important to the manageability and understandability of the model.

6.2 Modelling Framework Capabilities

In contemporary modelling frameworks there are three ways to capture relationships between two elements in a model: containment, hard references and soft references. Containment is the natural relationship of one element being a composite part of another, a hard reference is a unique-ID-based reference that can be resolved automatically by the modelling framework and a soft reference is a reference that needs an explicit resolution algorithm to navigate [10].

To enable users to split models over multiple physical files, contemporary modelling frameworks support cross-model containment (i.e. the ability of a

[1] Evidence of this is that CVS which was introduced in the 1980s is still the most popular version control system despite its obvious limitations compared to newer systems such as SVN

model element to contain another despite being stored in different physical files). With regard to hard and soft non-containment references, hard references are typically proffered because they can be automatically resolved by the modelling framework and thus, they enable smooth navigation over the elements of the model with languages such as OCL and Java. Nevertheless, in our view hard references are particularly harmful for modularity as they increase coupling between different parts of the model and prevent users from working independently on different parts. On the other hand, soft references enable clean separation of model fragments but require custom resolution algorithms which have to be implemented from scratch each time.

To address this problem, we envision extensions of contemporary modelling frameworks that will be able to integrate resolution algorithms so that soft references can be used, and the efficient and concise navigation achievable with languages such as OCL can still be performed.

7 Conclusions

In this paper we have demonstrated the importance of modularity and encapsulation for achieving scalability in MDE. We have identified two main problems: neglect of modularity constructs during the design of modelling languages and extensive use of ID-based references that lead to high coupling between different parts of the model. With regard to the first issue we have been working on preparing a set of guidelines for the design of scalable and modular DSLs and expect to report on this soon. The second issue is quite more complex and we plan to elaborate and prototype a solution based on EMF and Epsilon [11] in the near future.

Acknowledgements

The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2006-2009).

References

1. Jos Warmer, Anneke Kleppe. Building a Flexible Software Factory Using Partial Domain Specific Models. In *Proc. 6th OOPSLA Workshop on Domain-Specific Modeling*, Portland, Oregon, USA, October 2006.
2. Eclipse Foundation. Eclipse Modelling Framework. http://www.eclipse.org/emf.
3. Eclipse Foundation. Teneo, 2008. http://www.eclipse.org/modeling/emft/?project=teneo.
4. Eclipse Foundation. CDO, 2008. http://www.eclipse.org/modeling/emft/?project=cdo.
5. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of ACM*, 15(12):1053–1058, 1972.

6. David Hearnden, Michael Lawley, Kerry Raymond. Incremental Model Transformation for the Evolution of Model-Driven Systems. In *Proc. Model Driven Engineering Languages and Systems*, pages 321–335.
7. Holger Giese, Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, pages 1619–1374, March 2008.
8. Eclipse Foundation. EMF Compare, 2008. http://www.eclipse.org/modeling/emft/ ?project=compare.
9. Eclipse GMF - Graphical Modeling Framework, Official Web-Site. http://www.eclipse.org/gmf.
10. Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. Detecting and Repairing Inconsistencies Across Heterogeneous Models. In *Proc. 1st IEEE International Conference on Software Testing, Verification and Validation*, pages 356–364, Lillehammer, Norway, April 2008.
11. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon). http://www.eclipse.org/gmt/epsilon.