

Have we learned the lessons from metamodel-based environments?

Stephan Herrmann
Technische Universität Berlin
stephan@cs.tu-berlin.de

ABSTRACT

A possible merging of model driven software development (MDS) and aspect-oriented software development (AOSD) requires plenty of conceptual questions still to be answered. We argue that also the enabling technology for the envisioned way of software development using an integrated tool set determines a number of issues of how the two approaches can successfully be combined. We argue that lessons learned from research on software engineering environments are crucial also for future research. We elaborate how the five dimensions of tool integration – data, control, presentation, process and framework – relate to the issues at hand. We alert of the danger of inventing existing solutions again, but at a lesser quality.

Combining MDS and AOSD

It seems that many authors currently agree that model driven software development (MDS) and aspect-oriented software development (AOSD) not only have a lot in common, but also have the potential to contribute to a common way of producing software of much higher internal quality at very affordable efforts.

From a high level point of view a number of conceptual questions desperately await new ideas. How should aspects be modeled? Does advanced separation of concerns also require advanced technology for *integrating* modules that implement those separated concerns? Will weaving à la AOSD and transformations à la MDS converge towards a common integration mechanism? Should we strive for round-trip engineering or rather for single-source approaches, or will the „new“ way of producing software again be a waterfall-like approach?

It is important to advance software development by answering these questions, which focus on what software developers will do in the future. In this position paper we will on the other hand try to shed some light on rather technical issues: what will the envisioned software development environments look like? If we want seamlessness across artifacts, across models, across phases in the software life-cycle, across paradigms etc. we need to develop environments hosting a large number of tools which should cooperate very smoothly.

Software engineering environments

During the last twenty years of the 20th century, significant efforts have gone into the research and development regarding software engineering environments (SEE) (see [6]

for a summary). On the one hand this research has significantly triggered the development of several fundamental concepts and of technology we wouldn't want to miss today. Examples are component technology and middleware, object oriented databases and metamodel-based tool integration. On the other hand very little direct impact can be seen from that era: Today's tools seem not to be built in full knowledge of previous findings.

When discussing a possible convergence of MDS and AOSD, chances are high that concepts are invented again that have already been invented ten years ago or earlier. What should we have learned from the SEE research?

Open repositories

The most advanced SEEs were all built on an open repository. While the notion of repositories has no very precise definition, SEEs tend to associate a wealth of functionality to such a component. The main purpose of an open repository is to serve as a platform for integration. In this sense, repositories for SEEs were the direct precursors of today's application servers. The need for integration of different tools goes back directly to the insight that software development is more than just programming.

When considering models and aspects for software development it is evident that even more tools will need to be integrated. Probably those tools will cover a broader range of concepts and tasks than ever before. Will such environments be manageable at all? How can they raise productivity instead of paralyzing developers who are overburdened with too large a number of concepts and tools? A key to this will lie in integration in a common framework.

The most useful classification of a repository's functionality has been put down as an ECMA reference model[3]. According to this reference model, a *framework for software engineering environments* should support integration in five dimensions: data, control, presentation, process and framework integration.

None of these dimensions has lost its importance since. In contrast we argue that software development using models and aspects will to a large extent be driven by the development of more comprehensive tool environments. Such tool environments should in fact support the five dimensions of integration for the sake of advance on the road to a more seamless software development with higher productivity.

A closer look at the five dimensions of integration will result in a few critical remarks regarding today's hypes.

Data integration

A repository should be able to hold the data for all tools to be integrated. From this follows that the repository should not be bundled with any existing tool but should be a separate, central component of any SEE. The repository should provide facilities for meta modeling with strong support not only for different levels of abstractions but more importantly for several co-existing, perhaps overlapping views.

Two major strategies exist for supporting multiple views on one central repository: As much as possible should different tools operate actually on the same data within the repository. This reduces redundancy and avoids the necessity to transform between different documents, as it prevails in MDSD. The most ambitious approaches define views of the repository using mechanisms of the repository itself (see [5, 9, 7]). It is important to note, that each view definition implicitly includes two transformations, repository-to-view and view-to-repository. By their bidirectional nature these transformations can be hidden completely and developers can operate on different views simultaneously as to perform each change in that view where it is most conveniently done. Views are indispensable when development is seen as monotonous process of refinement and enrichment. No sequential chain of documents is artificially enforced, but each document/view remains editable throughout the development without introducing any danger of inconsistency. It should be noted that this one-model-many-views approach depends on consistency constraints to be enforced by the repository. For this purpose, some repository languages had support for predicates [10, 7], which can be compared to today's use of OCL constraints within meta models.

More recent research has, however, come to the conclusion, that redundancy and inconsistency can not completely be avoided (see e.g., [4]). In order to cope with inconsistencies dependency links and change propagation are indispensable. It is crucial to see that only some changes performed in one partial model could be propagated automatically to other partial models. Support is also needed for situations where the environment can inform the developer about potential conflicts but resolving these conflicts requires manual intervention. Such support requires explicit modeling of dependencies across artifacts. MDSD seems to imply exactly one kind of dependency: is-transformed-from. However, many other kinds of dependencies are relevant.

Using views a very specific problem of MDSD might be solved very easily: different reasons exist for adding some kinds of annotations (tagged values, stereotypes) to the model. Thus, when looking at the model it is cluttered with information relating to all different kinds of concerns. Grouping annotations into layers (cf. [1]) re-establishes the desired modularity as views can easily be defined to contain certain annotation layers only.

Other dimensions of integration

Today, tool integration is almost exclusively discussed in terms of data integration. We argue that the other dimensions of integration are essential, too. Disregarding any of

these dimensions will reduce the openness of the environment.

Control integration

In an environment with data integration only, the developer is responsible for invoking the right tools at the right time. Each switching between tools represents a full switch of contexts. Control integration refers to mechanisms that allow different styles of communication between tools and the repository. The basic style of control integration is update notifications according to the MVC architecture, where the repository holds the model and tools are seen as view-control units.

More ambitious approaches include facilities for (remotely) invoking operations of tools. Operations may be invoked directly at the user interface. Also, operations performed on the model may be delegated to some tool. Thus, the repository might, e.g., ask a tool to check a intended change using algorithms implemented in the tool. In this context the meta model is considered to contain not only structural definitions but also methods as parts of meta model classes. Also tools will be represented in the meta model, and finally scripting support at the meta model level will allow to easily write the glue code that will integrate the meta model and all installed tools into one homogeneous application. The cooperation between these modules is no longer manually controlled by the developer but tools and repository smoothly cooperate behind the scenes.

Presentation integration

This dimension of integration refers to a uniform GUI in which tools should come with a common look&feel. We see no specific issues that relate presentation integration to MDSD and AOSD.

Process integration

As the number of (sub-)models increases, it may be desirable to provide guidance for the developer regarding the sequence of development activities and tools used for each activity. Process integration means to use the repository for storing information about the development process, too. Thus a process model becomes an integral part of the meta model. The cooperation between tools may rely on process information in order to provide the desired guidance.

Framework integration

This is the most technical level of integration. It refers to support for configuring the SEE by self-application of its concepts and mechanisms. Using model-editors for creating specific meta models is just one late offspring of this concept. Developing an instance from an SEE framework should be no different from developing an application using that instance. Using framework integration the following elements will be modeled, configured and integrated: developer roles, artifacts, process policies, and most importantly: tools.

What's new?

The goals of MDSD and AOSD to use multiple models and views for describing software directly follow in the tradition of multi-view SEEs. Meta-modeling as a backbone for tool

integration is not new. MDSO raises the relevance of transformations in the software development process. In those cases, where transformations can not be defined in a bidirectional style, dependency management and consistency management are indispensable. Integrating automatic change propagation and manual repair is a most challenging task to be solved for evolutionary software development using MDSO. AOSD on the other hand introduces the concept of quantification and it should be checked whether the static part of this one-to-many relation will converge with queries on the repository, or what other techniques exist to model this kind of dependency.

Combining MDSO and AOSD means to combine an approach that focusses on (mostly graphical) models with another approach that has its strength at the code level. This emphasizes the need to allow development at the levels of models and source code simultaneously. Just as certain algorithms are best modeled using a programming language, aspect integration is best modeled using a pointcut-language. We don't see reasons to replace these languages with graphical notations.

Some SEEs have also experimented with combining coarse grained meta models with the full AST representation of programs. This may yield the following problem: normally any part of the model may refer to any other model element. References that point into the details of source code (or its AST) need specific protection because for this setting maintaining consistency is much harder than it is for the case where only references to coarse grained model elements are considered. Changing a few statements within a method implementation is normally considered as a local change without impact on other models. However, aspect-oriented pointcuts and the co-evolution of models and source code may require to breach this encapsulation. We have proposed references consisting of two parts: one part pointing to the course-grained (interface relevant) model element, a second part referring to the exact location within this element [7]. Consistency management using such two-part references will be easier. However, a general, full solution to such problems has not yet been given.

On the other hand, from an SEE perspective, the questions about weaving time (model, source code, binary code) is mainly a non-issue. If a model contains all information needed for weaving there is little use in actually performing the weaving. Once the integration is specified, a backend may produce a woven executable or a smart model interpreter will also interpret the crosscutting. From the SEE perspective, there is little difference. In fact some repository languages are in fact domain specific languages for distributed systems, where many aspects are interpreted by the repository, which in today's component technology are flattened using code generation. Perhaps, much of today's code generation would be solved much more conveniently by direct support for components at the level of programming languages, or code generation could happen at load-time based on annotations within the code.

AOSD tells us, there are transformations which are much more interesting than generating an interface and an implementation class. Resolving pointcuts, just like resolving

dependencies between different models is still a challenging task when thinking of smoothly integrated software engineering environments.

One final caveat: Developing software over time requires version control as a central mechanism for coordinating the work. Storing the model in a repository yields a –possibly quite fine grained– graph structure, say, a hyper-document. Consistent version control for fine-grained object structures with manifold non-tree-like links is a hard problem (see e.g., [2]). Computing model differences ([8]) also relates to the same issue and might become an essential technology for MDSO. If this problem is not solved in due time, it might eventually become a show stopper for wide spread adoption of MDSO.

1. REFERENCES

- [1] R. Buessow, W. Grieskamp, W. Heicking, and S. Herrmann. An open environment for the integration of heterogeneous modelling techniques and tools. In *Proc. of the International Workshop on Current Trends in Applied Formal Methods*, number 1641 in LNCS. Springer, October 1998.
- [2] Jr. E. J. Whitehead. Design spaces for link and structure versioning. In *Proceedings of the 12th ACM conference on Hypertext and Hypermedia, HT'01*, pages 195–204. ACM Press, 2001.
- [3] Reference Model for Frameworks of Software Engineering Environments – ECMA TR/55 3rd edition. Technical report, European Computer Manufacturers Association (ECMA), June 1993.
- [4] W. Emmerich, J. Arlow, J. Madec, and M. Phoenix. Tool construction for the British Airways SEE with the O₂ OODBMS. *Theory and Practice of Object Systems*, 3(3), 1997.
- [5] David Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, May 1987.
- [6] W. Harrison, H. Ossher, and P. Tarr. *The Future of Software Engineering*, chapter Software Engineering Tools and Environments: A Roadmap. ACM, 2000.
- [7] Stephan Herrmann. *Views and Concerns and Interrelationships - Lessons Learned from Developing the Multi-View Software Engineering Environment PIROL*. PhD thesis, Technical University Berlin, 2002.
- [8] D. Ohst, M. Welle, and U. Kelter. Differences between versions of uml diagrams. In *Proceedings of ESEC/FSE*, pages 227–236. ACM Press, 2003.
- [9] C. Santos, S. Abiteboul, and S. Delobel. Virtual schemas and bases. In *Proc. of the International Conference on Extending Database Technology*, volume 779 of LNCS, pages 81–93. Springer-Verlag, March 1994.
- [10] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming*. PhD thesis, University of Colorado, Boulder, Aug 1990.