Reflective model driven engineering

Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, Damien Pollet

Reflective Model Driven Engineering

Jean Bézivin[1], Nicolas Farcet[2], Jean-Marc Jézéquel[3],
Benoît Langlois[2], and Damien Pollet[3]

[1]INRIA/Irin-Université de Nantes, France
[2]Thales Research and Technology, France
[3]INRIA/Irisa-Université de Rennes, France

Abstract In many large organizations, the model transformations allowing the engineers to more or less automatically go from platform-independent models (PIM) to platform-specific models (PSM) are increasingly seen as vital assets. As tools evolve, it is critical that these transformations are not prisoners of a given CASE tool. Considering in this paper that a CASE tool can be seen as a platform for processing a model transformation, we propose to reflectively apply the MDA to itself. We propose to describe models of transformations that are CASE tool independent (platform-independent transformations or PIT) and from them to derive platform-specific transformations (PST). We show how this approach might help in reaching a consensus in the RFP on MOF QVT, including a solution for the declarative/imperative dilemma. We finally explore the consequences of this approach on the development life-cycle.

Keywords: Model-driven engineering, Model transformation

1 Introduction

Large companies having to develop, maintain and evolve large scale software systems over long periods of time are considering strongly the OMG initiative on Model Driven Architecture (MDA) [19]. In the air traffic management domain for instance, the domain specific models are less likely to change rapidly than platform-specific ones. So the MDA core idea that it should be possible to capitalize on platform-independent models (PIM), and more or less automatically derive platform-specific models (PSM) –and ultimately code– from PIM through model transformations is extremely appealing. But in some business areas involving fault-tolerant, distributed real-time computations, there is a growing concern that the added value of a company not only lies in its know-how of the business domain (the PIM) but also in the design know-how needed to make these systems work in the field (the transformation to go from PIM to PSM).

Until now, model transformations have in most cases been developed within modeling tools using proprietary languages such as J in Softeam Objecteering or Visual Basic in Rational Rose. We consider this method for expressing model transformation as tool-specific. Such a method is unfortunately far from being

reusable and robust. If model transformations from domain models into platform models are all developed in a proprietary or unstable language, we may indeed loose the reusability of domain models which is one of the key advantages drawn from MDA.

Model transformations are thus increasingly seen as vital assets that must be managed with sound software engineering principles: they must be analyzed, designed, implemented, tested, maintained and be subject to configuration management. For the same reason that domain know-how should not be tied to a particular platform, it is critical that model transformations are not prisoners of a given CASE tool.

Considering a CASE tool as a platform for processing a model transformation, we propose to reflectively apply the MDA to itself: transformations should be developed along a cycle ranging from platform-independent transformations (PIT) down to platform-specific transformations (PST). Platforms here should be understood as the tools that allow the specification, design and execution of transformations (not to be confused with the "platform" term as used on the PIM to PSM application development life-cycle: both platforms are different ones!). Such tools may provide various languages to express platform-specific transformations such as J, Visual Basic, or XSLT.

We claim in this paper that UML is the ideal base language for describing these PIT. Because of its static structuring mechanisms (packages, classes and methods), UML has the power to model large and complex families of transformations that can be organized and evolved using familiar object-oriented principles. For finer grain transformations, a lot of formalisms are eligible, from a declarative specification of transformations using the Object Constraint Language (OCL), to more imperative approaches, such as sequence diagram, activity diagram or action semantics.

The rest of the paper is organized as follows. In Sect. 2, we propose several ideas to describe models of transformations that are CASE tool independent (PIT) and from them to derive platform-specific transformations (PST). We show in Sect. 3 how this approach might help in reaching a consensus in the RFP on MOF QVT (Query/View/Transformation) [14], including a solution for the declarative/imperative dilemma. We explore the consequences of this approach on the development life-cycle in Sect. 4. Related work is discussed in Sect. 5.

2 Modeling Transformations

2.1 The three ages of transformations

Looking generally on transformation systems, we may find three stages in their development.

The first stage may be exemplified as the Unix system as a software development workbench. Many Unix commands implement transformation operations and the framework itself (pipes, etc.) encourages programmers to build composite transformations from atomic ones. A variety of different operators, each

having a different format for writing transformation operations, allow building complex transformations applied to code or data. Such a typical transformation command is *awk*, which implements a declarative rule system applied to a linear record-oriented structure (files composed of lines).

The second stage corresponds mainly to the tree-transformation systems such as XSLT [5]. An XSLT script declaratively specifies how to explore the input tree and how to generate fragments of the output tree. The navigation in trees is more difficult than just addressing the various fields of sequential lines and has been addressed by the XPath [1] language, which has been factored out and is used in other proposals such as XQuery. Languages like XSLT are heavily used, but they may still be considered as initial proposals and several issues are not fully solved, such as modularity, efficiency or reusability.

What we are now witnessing is the progressive elaboration of a third generation of transformation systems, moving away from global interpretive approaches towards more transformational software development techniques. Modern model transformation systems operate on general graphs, not simply on tree structures, and a large body of knowledge already exists in the graph grammars and graph transformation community [18,6]. This has many consequences. It is still possible to use XML-based tree transformation tools like XSLT through a serialization of graphs such as the XML Model Interchange (XMI). However this could be compared to the tentative use of tools like *awk* to perform XML transformations. Instead, we need a powerful graph navigation formalism like XPath for trees. Fortunately the models we have to handle are specific graphs and the OCL language provides the basis of this navigation. OCL is then the natural candidate to be used in both the pattern part and also in the instruction part of the transformation rules (at least in version 2.0).

2.2 MDA and QVT

The OMG has built a meta-data management framework to support the MDA. It is mainly based on a unique M3 "meta-meta-model" called the Meta-Object Facility (MOF) [13] and a library of M2 meta-models, such as the Unified Modeling Language (UML) (or SPEM [15] for software process engineering), in which the user can base his M1 model (Fig. 1).

In the MDA, a specific model captures each aspect of a system, at different stages of its development. MDA capitalizes on standard languages such as UML, XMI and the MOF; however, what is lacking is the definition of a conceptual framework for model management, with a precise definition of model and model manipulation language.

This likely requires some kind of standardization work as it is the case with the OMG MOF QVT activities. One of the advanced ideas in the QVT request for proposals is to consider as uniformly as possible queries, views and transformations on models. This may be an advantage on the document area where many different languages like XSLT and XQuery are competing for transformation and querying.

However the RFP on QVT might not be the solution to all the problems. First, considering the dramatic differences in point of view that have appeared in the initial submissions, it might be difficult to reach a consensus on a unique, semantically well founded language for describing MOF-based queries, views and transformations. But even if we are lucky enough to reach a consensus on QVT at the OMG, experience with previous OMG standards teach us that 5 to 10 years are needed before we can really rely on inter-operable solutions, because the logic of standardization goes against the interests of implementation vendors. But business units want to leverage the MDA *now:* they are not wanting to wait such an eternity (from an industrial perspective) before the MDA can be safely deployed in the trenches.

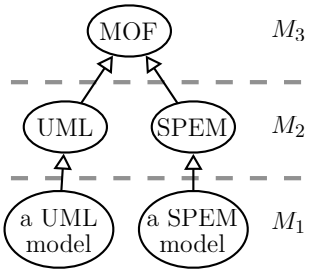


Figure 1. Standard OMG Layered Organization

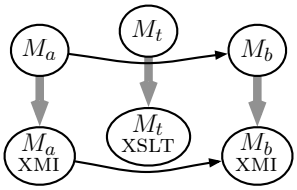


Figure 2. Transformation by a model M_t and compilation to XSLT

Studying model management in the context of the MDA has several advantages. First, by relying on open standards, we can reuse previous results work from others. Among the various meta-models at level M2, we find different categories like processes and products, static and dynamic, functional and non-functional, object and relational, PIM and PSM, etc. The main problem of interest here is to express how model M_a is transformed into model M_b. Since our vision is to consider models as first-class citizens, the transformation will be realized by model M_t and this could be achieved by serialization of M_a and M_b to XMI and compilation of M_t to XSLT or to XQuery as described in Fig. 2.

Now we must consider that similarly to M_a and M_b, M_t is a model and thus is written in the language of its meta-model. This has several consequences. The first one is that when we compile M_t to XSLT, we have access at compilation time to MM_a and MM_b –the meta-models of M_a and M_b. Hopefully these meta-models will be decorated –enriched– with OCL assertions, making the translation process easier and more precise. The second consequence, also discussed in [8], is that the meta-model of M_t is the –hypothetical– language for describing all kinds of transformations.

2.3 Requirements for a Transformation Meta-Model

Although models capture the design features of a product, model transformations capture the model manipulation expertise. For large companies, such expertise represents a long-term investment, and the initial cost of developing model transformations should be balanced over time, when transformations can be reused at a negligible cost compared to an ad hoc solution.

However the increasing complexity due to adaptation and evolution of transformations should not jeopardize reuse and thus return on investment. The problem is therefore to identify techniques and methods enabling transformation development and maintenance, by addressing the following points:

Reuse Most transformations are not "one-shot" but depend only on the metamodels involved and will be reused on several models.

Composition In the MDA view, the end product model is obtained through successive refinement and combination of higher-level models, that is through composition of successive transformations.

Genericity Some transformations such as a documentation generator need not know about the details or the precise place of the source model in the refinement chain, and can carry out the same task more or less generically.

Customization From one affair to the next one, or to deliver software for variants of the same platform, we generally want to reuse the same overall transformations while customizing a few specific points (hot-spots). The same goes for large product lines, where generic transformations may have to be specialized for a narrower range of activity.

Maintenance Transformations will need maintenance as they grow and adapt to the business evolution.

These points make it clear that transformations are complex software products, which we should develop using established software engineering techniques.

2.4 Using UML to Design Transformations

We believe that transformations should be first-class models in the MDA world; we propose here to adopt the object-oriented approach and to leverage the expressive power of UML as a metamodel defining the transformation language. Both as a modeling language and as a management tool, UML provides concepts useful to analysis, design, and development of transformations:

Model management diagrams address the macro-organization of the transformation components in UML packages.

Class diagrams reveal the structure and the patterns in the transformation design. Rules are expressed as operations, organized in classes and packages. Subclassing and dynamic binding can be used to handle variability, for example by leveraging the classical design patterns [7].

Activity diagrams express the transformation process by capturing the dependencies between transformation subtasks and can be used to combine multiple transformations.

Deployment diagrams specify platform-specific aspects, e.g. which CASE tool should be used to handle models of a given metamodel.

UML alone cannot model transformations directly, so a profile for transformations should be defined. Besides the use of UML, developers will eventually use a specific runtime platform to actually transform models; however the choice of a runtime platform should not impact on the transformation design. This is in line with the separation of concerns between PIM and PSM in the MDA, and leads to the concepts of platform-independent transformation (PIT) and platform-specific transformation (PST).

Platform-independent transformations are models of the transformation program, relying on a generic library of simpler transformations and transformation primitives. They will eventually be refined to the point where they can be used as the source to generate platform-specific transformations. Then if UML is the language of PIT, PST are models of tool-specific formalisms or API; for example, while XSLT is a textual language, it is possible to consider a MOF-compliant metamodel of XSLT; the PST is then an XSLT model which is actually serialized behind the scene to its XML representation.

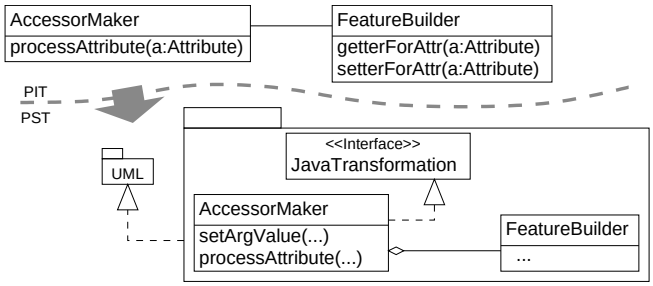


Figure 3. One refinement step in the development of a model transformation which adds setters and getters for an attribute of a given class, and makes the attribute private.

Fig. 3 illustrates the refinement of a PIT targeted at integration in an hypothetical tool which provides an API to a repository of UML models, and requires conformance to a JavaTransformation interface so that it can load and call the transformation code. The refinement changes the association to an aggregation, maps the imported UML metamodel to the repository API, and adds the JavaTransformation interface and redefined operations. Once written as an automated transformation, it is reusable to help specialize any transformation to the tool.

3 Solving the Declarative vs. Imperative Dilemma

Modeling transformations using UML has many advantages from the software engineering point of view, but the problem of actually applying transformations

to models remains: UML has no predefined execution model. The two main approaches identified, imperative and declarative, are competing in answering the OMG RFP on QVT.

3.1 Imperative Transformation Programs

In the imperative paradigm, transformations are programs of an imperative programming language, and modify a model through side-effects. We have to give semantics to method bodies and actions of the transformation model, by adjoining an action language to UML. There is many choices for such a language: *(1)* a programming language such as Java, Python or a case-tool scripting language, *(2)* Action Semantics, or *(3)* OCL extended with imperative features. While *(1)* is probably the most straightforward to get up and running by generating code in modern UML environments, it is certainly not the best approach as it nullifies the interest of PIT: any transformation sufficiently detailed would be specific to a given runtime platform or case-tool. Both action semantics and the forthcoming OCL 2 are tightly integrated with the UML metamodel. Model transformations are very similar to meta-programs (programs manipulating programs). In fact, they manipulate models, i.e. in the case of an UML model they modify Instances whose classes are Classifier, Package, State, etc. Such manipulations can be expressed using Action Semantics, but OCL has already been widely adopted as the language of choice to traverse models and select elements, and we believe it would integrate nicely with an extended syntax with side-effects and model manipulation features [16].

3.2 Declarative Transformation Rules

In the declarative paradigm, transformations are defined by composition of rules described using pre- and post-conditions. Preconditions define patterns that are to be matched in the source model, and are used to identify interesting elements; postconditions define the state of the destination model once the rule has been applied. This approach covers both OCL specifications and graph rewrite systems. It is both expressive and precise, as the transformation programmer can refine the conditions as much as necessary using a constraint language such as OCL (as shown in Fig. 4), and is technology independent.

3.3 Declarative vs. Imperative Sum-Up

The declarative approach is well suited to write incomplete or abstract transformations in the early stages of development. Such transformations need to be refined until they are executable. In simple cases, this refinement can be carried out efficiently by an inference mechanism *à la* Prolog. For complex rules it would probably have unrealistic computation times or counter-intuitive results, but it could still restrain the developer's choices in a semi-automatic mode. A more pragmatic method is to give the explicit imperative behavior. Here, the pre- and

```
context FeatureBuilder :: newGetter(a:Attribute):Operation
post: result.name = "get_".concat(a.name) and result.returnType = a.type

context FeatureBuilder :: newSetter(a:Attribute):Operation
  ... -- similar as newGetter

context AccessorMaker :: processAttribute(a:Attribute)
pre: a.visibility = #public -- not overwrite existing accessors
  and a.class.feature->select(name = "get_".concat(a.name)
    or name = "set_".concat(a.name))->isEmpty
post: a.visibility = #protected
  and a.class.feature->includes(o:Operation |
    o.name = "get_".concat(a.name) and o.returnType = a.type)
  and ... -- similar as above for setter

context AccessorMaker :: processClass(c:Class)
post: c.ownedAttribute->forAll(a | processAttribute(a))
```

Figure 4. Pseudo-OCL declarative specification of the AccessorMaker (Fig. 3)

post-conditions of the initial rule should be considered as specification contracts for this implementation.

If we assume transformation inputs are MOF-compliant models, primitive transformations (instance and link creation, setting attribute values, etc.) are reduced to a finite number and can be defined using a declarative formalism. Both approaches can thus coexist by sharing the same basic representation. They can also coexist at the level of transformation components: an imperative transformation can call a declarative sub-transformation, and the other way is possible by wrapping an imperative in a declarative interface.

4 Towards a New Development Life-cycle

There is an obvious analogy between manual development of code vs. code generation on one side, and ad hoc transformations vs. precise modeling of transformations rules on the other side: most ratios can lead to similar results but with different cost models. Most code development is one-shot work whereas most work on code generation templates is carried out with product-line development in mind. Whereas the later case is more costly initially (as it requires a good understanding of the product-line commonality and variability), it may however prove to bring a stronger return on investment (ROI) in the long run. In the case of model transformations, the initial cost of precise metamodeling and transformation rules elicitation is a form of capitalization and may lead to more robustness and reusability across affairs.

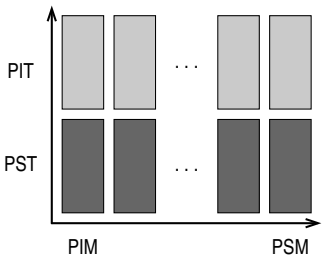
If model transformation is a key in model-driven application development, what can be said about the development of the transformations themselves? How does transformation development relate to application development?

4.1 Model Transformation Life-cycle

In the previous sections, we proposed to apply the MDA to itself with the notions of PIT and PST, which give two new steps in a model centric life-cycle:

1. Express model transformations in a tool-independent way;
2. Map this tool-independent expression into an actual tool (which leads to tool- or technology-specific model transformation expressions).

We now have a whole development cycle of transformations going from PIT to PST. This development life-cycle can for now be seen as orthogonal to the application development life-cycle as shown in Fig. 5. This figure represents a "two-dimensional MDA" approach of model layering: the first dimension is the usual layering of application models from PIM to PSM; in the second dimension, transformation models are also layered, from PIT to PST.



Vertical axis: Expression of the model transformation at the PIT and PST levels

Horizontal axis: Model transformations applied on the successive PIM to PSM models

Figure 5. Two-dimensional MDA

4.2 Model Transformation Process

Figure 6 depicts in SPEM [15] a typical MDA-like development life-cycle of a transformation. This corresponds to a vertical PIT to PST model transformation chain as depicted in Fig. 5. The model transformation actors are:

The transformation framework developer abstracts, and then facilitates, the model transformation expression for the transformation developer.

The transformation developer specifies and develops the domain model transformations at PIT and PST levels. By domain, we mean here what is relevant at a given abstraction level along the application development life-cycle, e.g. a design PIM to be transformed into an EJB PSM.

The transformation user applies the model transformations. This may happen at any given step along the application development life-cycle. The actual application of the transformation is handled by the tool and this is why the PST transformation is used here, but if the tool were to support PIT natively, we could remove all reference to PST in Fig. 6.

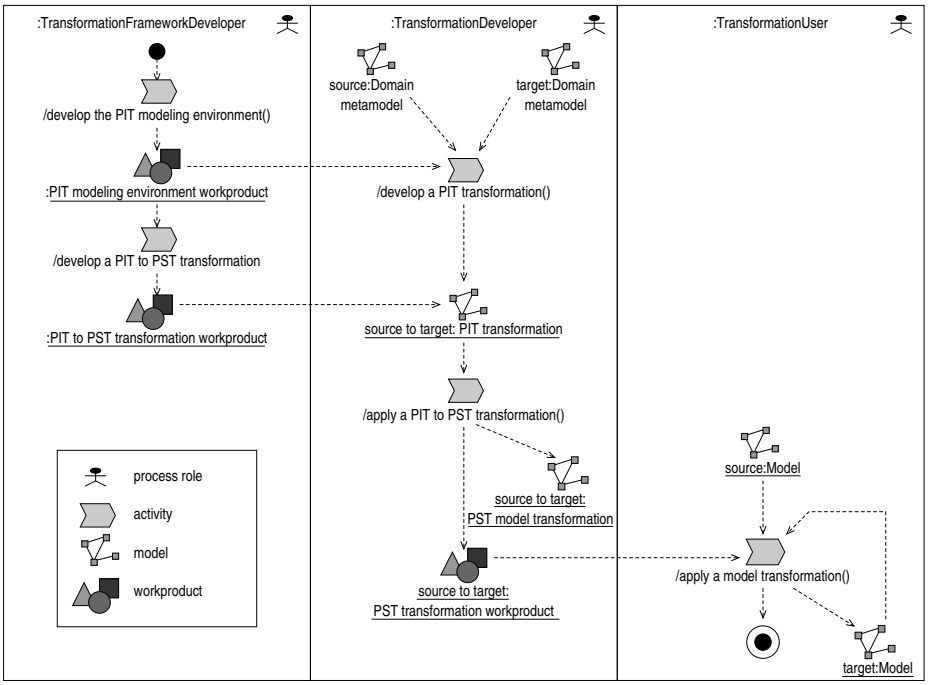


Figure 6. Model transformation life-cycle

The model transformation work products used and produced during the activities by the model transformation actors are:

The PIT modeling environment work product: it is a set of PIT materials, like UML profiles and facilities, required for modeling the transformations at the PIT level.

The PIT to PST transformation work product: it is a set of profiles and facilities required to transform a PIT into a PST.

The domain metamodel: it is a metamodel description at a given PIM or PSM abstraction level. The transformation developer uses a source and a target domain metamodel to build a PIT transformation across source and target modeling layers of application development. For example, to transform a design PIM into an EJB PSM, the design PIM metamodel is the source metamodel and the EJB PSM metamodel is the target metamodel.

The PIT transformation: it is a tool- and technology-independent model transformation model. For example, this kind of transformation may be modeled using UML activity diagrams and OCL or any other formalism which is standard and widely accepted.

The PST transformation: it is a tool- or technology-dependent model transformation model.

The PST transformation work product: it is a set of PST materials containing tool support for the actual model transformation that the transformation user may apply to its source model to produce a target model.

The model: it is a model at a given PIM or PSM abstraction level.

On one hand, the transformation framework developer defines the modeling environment required for the transformation developer to express its domain model transformation at the PIT level. Then, the transformation developer describes the source to target model transformation at the PIT level, e.g. from design PIM to EJB PSM without tool or technical considerations.

On the other hand, the transformation framework developer produces a PIT to PST transformation. Then, the transformation developer applies or re-applies the PIT to PST transformation for each concerned platform. For example, the PIT is mapped into a specific tool scripting language, into XSLT, or into a documentary form. Fig. 7 presents an example of derivation from a PIT model transformation into OCL, a tool language and a documentary form.

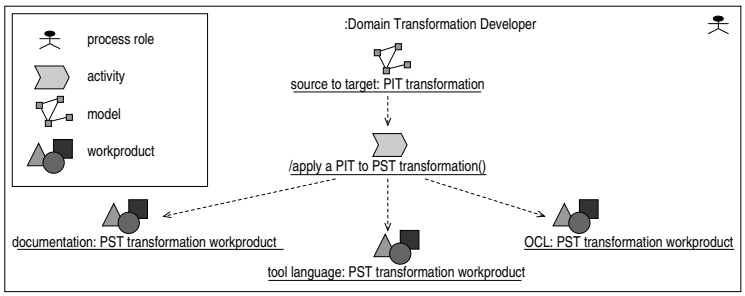


Figure 7. Example of derivation from a PIT to various PST

Whereas the transformation framework developer process usually occurs only once, when the application development tool-chain is setup and customized, the transformation builder process typically occurs at every modeling layer boundary along the application development life-cycle. The frequency of the transformation user process is even higher and eventually, every single developer within a project may be in position to use transformations built by the transformation developer, at any stage of the application development. These transformations, delivered as PST model transformation work products, may contain documentation, wizards, or anything that may render the work of final developer easier and more closely integrated within an overall well mastered development process.

4.3 Elements of Return On Investment (ROI) for the "Two-Dimensional" MDA

The reasons to choose a PIT to PST "two-dimensional" MDA approach are mainly an extension of the reasons to adopt a PIM to PSM "one-dimensional" MDA approach:

- To have a durable model transformation expression: the key point is to be independent of the technological evolution (language versions, tools, etc.).
- To raise the level of abstraction of model transformation description: the transformation developer can focus his efforts on the model transformation and not on the understanding of specific tool techniques, languages, and practices. The key point here is to be tool- (e.g. dedicated scripting language, UML tool-specific idioms) and technology-independent (e.g. imperative vs. declarative approach, XSLT). And the level of abstraction can be raised even higher so that the transformation developer can focus on the concepts he is manipulating within various modeling layers.

However, developing transformations from PIT to PST is an investment:

- Investment for each PIT to PST transformation: some bridges are not so expensive, for example to generate a documentation from an activity diagram, it is mainly an activity graph parsing; some bridges are more expensive, e.g. those that aim to seamlessly cope with to UML metamodel evolution. This cost occurs when building the transformation framework.
- Investment for each tool where these PIT to PST transformations are expressed: every facility and PIT to PST transformation is written for a dedicated tool with its specificities that are not portable. Choosing a new tool implies to rewrite all development; only the methodology remains. This cost occurs when building the transformation framework.

Other questions also arise in the evaluation of the ROI: Are PIT to PST transformations reusable along the PIM to PSM development life-cycle? Is the ratio PIT/PST constant along the PIM to PSM development life-cycle? The answer is probably *no:* when transformations become exomorphic (i.e. convert UML into non-UML), they will probably be expressed as PST, and rely on tool specific-features. This may occur rather on the PSM side of the application development life-cycle (e.g. for code template generation mechanisms) than on the PIM side.

More could be said on these topics, but doing an extended analysis of the ROI is out of the scope of this paper. We can however say now that the "two-dimensional MDA" that puts model transformation development life-cycle at the core of the application development life-cycle has a greater ROI than the conventional "one-dimensional MDA".

5 Related Work

There have been numerous attempts to provide transformation facilities in various environments. Each has its weak and strong points. One of the most notable

initiatives in the recent period is the model management project at Microsoft Research. In [17] a programming platform for generic model management is described, providing high-level operators to manipulate models and mapping between models. The key conceptual structures are models, morphisms and selectors. Morphisms are a simple class of mappings between models while selectors are sets of node identifiers originating from one or several models. The platform intends to help investigating whether meta-data management can be done in a generic fashion, which is really the central question to which present research efforts are trying to bring answers. The previously described project may be situated in the domain of data bases. In [10] a general view of technological spaces (TS) has been proposed. The DBMS TS for example contains plenty of other examples of transformation related to schema evolution, data base integration or optimization. Languages like SchemaSQL [11] or MetaOQL are examples of some realizations there.

In addition to the DBMS field, other TS may also be mentioned like documents, programming languages, formal languages and various theories, model engineering, ontologies and knowledge engineering, etc. Many TS are built on some basic typing system (documents/schemas for XML, programs/grammars, models/meta-models, etc.). Each space has its specific advantages and drawbacks, both on the theoretical and engineering levels. Some operations are best performed in some particular space, using well-defined space correspondence bridges.

We have already mentioned in this paper several examples of TS where transformation languages, engines and frameworks occupy a central place. Much more could be found. Notations like MOF/QVT and situations similar to the reflective transformation mentioned here are commonly found. In the related TS of theoretical formalisms, general graph rewriting techniques have been widely studied with systems like Attributed Graph Grammars, PROGRES [18], etc. Applicability in the model engineering TS has been demonstrated by building bridges in several experiments like [20]. Theoretical results like the NP-completeness of sub-graph detection may be of high relevance to the global transformation field.

The model engineering TS is presently trying to complete its applicability spectrum by acquiring a kernel transformation technology. Much inspiration may be drawn from other technologies that already went through this evolution. The precise solutions have yet to be defined.

6 Conclusion

In large software companies, model transformations are increasingly seen as vital assets that must be managed with sound software engineering principles. For the same reason that domain know-how should not be tied to a particular platform, it is critical that model transformations are not prisoners of a given CASE tool. Considering that a CASE tool is a platform for processing a model transformation, we have proposed in this paper to reflectively apply the MDA to itself: transformations themselves should be developed along a cycle rang-

ing from platform-independent transformations (PIT) down to platform-specific transformations (PST). Using a powerful and abstract language such as UML for modeling transformations also brings the additional benefit to foster a solution to the declarative/imperative dilemma. We have explored the consequences of this approach on the development life-cycle, with a two-dimensional MDA process.

We are currently implementing an open source tool-set to support these ideas in the framework of the CARROLL research program[1], launched by Thales and two French public research laboratories: CEA (Commissariat à l'Énergie Atomique) and INRIA (Institut National de Recherche en Informatique et en Automatique). The joint program is aimed at developing software engineering and middleware technology.

Over the coming years, the goal of CARROLL is to spearhead research that is focused on pinpointing increasingly competitive software developments for large-scale and embedded systems, the likes of which are vital in today's ever more demanding and complex software environment. The results will thus be of valuable worth not only to Thales, but also to software editors and other industry players.

Acknowledgment

We would like to thank Daniel Exertier, Madeleine Faugère, and Dominique Sueur, who have also been active contributors on the topic of model transformation within a pilot program called MIRROR, conducted by Thales Research and Technology.

References

1. XML path language (XPath) 2.0. W3C Working Draft, November 2002. http://www.w3.org/TR/xpath20/.
2. J. Bézivin and O. Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings of ASE'01*, 2001.
3. Jean Bézivin. From object-composition to model-transformation with the MDA. In *Proceedings of TOOLS-USA'2001*, 2001.
4. Jean Bézivin and Richard Lemesle. *Reflection and Software Engineering, Cazzola et al. Eds*, volume 1826 of *LNCS*, chapter Towards a True Reflective Modeling Scheme. Springer, 2000.
5. J. Clark (Eds). "XML Transformations (XSLT) Version 1.1". W3C Working Draft, December 2000. http://www.w3.org/TR/xslt11.
6. G. Engels, G. Rozenberg, and H.-J. Kreowski, editors. *6th International Workshop on Theory and Application of Graph Transformation*, volume 1764 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2000.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[1] See www.carroll-research.org for more details.

8. David Harel and Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff. Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, September 2000.
9. C. Kobryn. UML 2001: A standardisation odyssey. *Comunications of the ACM*, 42(10), 1999.
10. I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: An initial appraisal. In *Int. Federated Conf. (DOA,ODBASE, CoopIS), Industrial track*, Los Angeles, 2002.
11. Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. SchemaSQL—A language for interoperability in relational multi-database systems. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pages 239–250, Mumbai (Bombay), India, 3–6 September 1996. Morgan Kaufmann.
12. S. Melnik, E. Rahm, P. A. Bernstein, and P. A. Rondo. A programming platform for generic model management. In *Proceedings of Sigmod'2003*, June 2003.
13. OMG/MOF. Meta Object Facility (MOF) specification. OMG Document ad/97-08-14, September 1997.
14. OMG/RFP/QVT. MOF 2.0 Query/Views/Transformations RFP. OMG Document ad/2002-04-10, October 2002.
15. OMG/SPEM. Software Process Egineering Metamodel (SPEM). OMG Document ad/01-formal/02-11-14, version 1.0, November 2002.
16. Damien Pollet, Didier Vojtisek, and Jean-Marc Jézéquel. OCL as a core UML transformation language. WITUML 2002 Position paper, Malaga, Spain, June 2002.
17. R. A. Pottinger and P. A. Bernstein. Merging models based on given correspondences. Technical report, University of Washington, 2003.
18. A. Schürr, A. Winter, and A. Zündorf. Graph grammar engineering with PROGRES. In Botella and Schäfer, editors, *ESEC'95 Proceedings of the 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234, Berlin, 1995. Springer-Verlag.
19. R. Soley and the OMG Staff. Model-Driven Architecture. OMG Document, November 2000.
20. A. Tsiolakis and H. Ehrig. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Berlin, March 2000*, 2000. Technical Report no. 2000/2, Technical University of Berlin.