Toward Families of QVT DSL and Tool

Benoît Langlois, Daniel Exertier, Ghanshyamsinh Devda
Thales Research & Technology
RD 128 – 91767 Palaiseau, France
{benoit.Langlois, daniel.exertier, ghanshyamsinh.devda}@thalesgroup.com

Abstract
QVT (Query/View/Model transformation) is a modeling core technology for the development of MDD (Model-Driven Development) tools. Thus, it is strategic for a company to have QVT techniques improving productivity of development teams, especially when standards, tools, user requirements, and practices are ever evolving. The interest of introducing QVT DSL (Domain-Specific Language) is to offer higher level QVT languages in order to produce QVT tools in a faster and safer way. For assessing this position, this paper presents, at first, the case study of a view DSL for producing tools generating model diagrams. From this proof of concept of QVT DSL, this paper studies the introduction of families of QVT DSL and tool to fit to multiple project contexts.

Keywords
MDD, DSL, PRODUCT LINE, QVT, SOFTWARE FACTORIES.

1. Introduction

To produce high quality software both on budget and schedule, companies usually face productivity improvement issue. This is particularly true in the context of large-scale systems. In the current MDD context, while projects encounter evolving environment of standards, tools, user requirements, and practices, they have no other choice than to use low level QVT languages, essentially code-based, for the development of MDD tools. Then, it then becomes judicious to introduce higher level languages, for instance with wizards, easing QVT descriptions which contain sufficient information for translation toward low level QVT languages. This is the purpose of DSL dedicated to QVT.

In order to validate this category of DSL, this paper presents the case study of Diagram DSL. From a description conforming to a Diagram DSL can be deduced MDD tools generating model diagrams. The interest is such that the diagram designer does not code anything, development and maintenance tasks are easier and safer. However, this technique is limited to the production of one type of QVT DSL development. The next step is the production of QVT DSL variants meeting contextual requirements. For instance, a Diagram DSL cannot presume and contain all descriptions of diagram layout; such is the case for a serializer for which it is impossible to determine all exchange formats. Then, from core QVT assets, variations can be applied to produce the expected DSL and tool. For instance, a serialization language can be specialized into several serialization languages for meeting specific model exchange formats. This opens the way of families of QVT DSL and tool.

This paper is organized as follows. Section 2 studies the link between QVT and DSL. Section 3 presents the case study of a view DSL for producing tools generating model diagrams. Section 4 extends this result toward families of QVT DSL and tool. Section 5 presents further work and section 6 concludes.

2. Link between QVT and DSL

The MOF 2.0 QVT standard [18], a key technology for the OMG's MDA™ (Model-Driven Architecture), defines a language for model transformation, such as a PIM (Platform-Independent Model) to PSM (Platform-Dependent Model) transformation. A query is an expression that is evaluated over a model; it returns one or more instances of types defined in the metamodel of the model or defined by the query language. A view is a model derived from the base model, such as diagrams or code; a view does not modify the base model. A transformation generates a target model from a source model, such as a PIM to PIM transformation. Source and target models conform to their metamodels [10]. In this paper, we do not restrict our vision to the OMG's standard. For instance, the Epsilon Object Language (EOL) [7][13] of the Epsilon Model Management Framework provides all mechanisms for model navigation and transformation. From this standalone language, specific languages can be constructed, for instance for model merging or text generation from models. Kermeta is a metamodeling language, compliant with EMOF [17], which provides an action language for specifying behavior and for implementing executable metamodels, *e.g.* for implementing transformation languages, or action languages. AMMA (Atlas Model Management Architecture) [1], built atop EMF, is a general-purpose framework for model management. Based on ATL [2], it provides a virtual machine and infrastructure tool support for combining model transformation, model composition and resource management into an open model management framework.

A DSL, for its part, is a specialized, problem-oriented language [6], in that DSL focuses on a problem contrarily to a general-purpose language, such as UML. From a DSL to target language, a problem-to-solution transformation encapsulates complexity and hides decisions or irrelevant implementation details. During transformation, automation avoids repetitive and tedious user tasks and guarantees systematic practices. Regarding the process engineering, wizards can guide users in their development tasks and domain rules ensuring that data are reliable. Thus, DSLs are means toward easing problem expression. The objective of complementary DSLs is actually productivity improvement by industrialization of modeling chains from requirements down to the packaging of produced assets, such as models, model transformations, model views, configuration files, or documentation.

In the QVT context, a QVT DSL has for role to ease the development process where the considered domain is QVT. For MDD end-users that means QVT DSLs ease domain modeling, such as consulting model or applying patterns. For MDD users at the metamodel level that means QVT DSLs ease tool creation, for instance for designing model transformations or defining modeling processes. A DSL which is not a QVT DSL is a DSL which does not apply QVT action over a model. This is for instance the case of a profiling tool which audits models from data contained in files without QVT action.

The following, non exhaustive, table proposes a categorization of QVT DSL. The levels of QVT usages with DSL are presented in abscissa: 1/ The "Core technology" aspect covers the OMG's QVT standard and QVT languages which conform to it; 2/ The "Development" aspect covers the category of DSL enriching the QVT languages for the development of MDD tools, such as a DSL for traceability management; 3/ The "Business" aspect targets DSL for end-users, such as a Diagram DSL. In ordinate, we find the Query, View, and Transformation aspects. By crossing the two dimensions, a Diagram DSL is for instance a view DSL for business activities of development that a end-user can use in his modeler.

Development DSLs depend on core technology DSLs; business DSLs depend on development DSLs or directly on core technology DSLs. As a consequence, core technology QVT DSLs evolve more slowly and are less numerous than the others; business DSLs have on the contrary shorter lifecycles, they are

more numerous and business- or project-specific. For instance, a few QVT languages implement the OMG's standard while every project can decide to customize properties for its diagram presentations.

Table 1. Categories of QVT DSL

	Core technology	Development	Business
Q	QVT languages	Model information	Model checking
V		–	Diagram Documentation
T		Merging Traceability Transformation pattern Versioning	Abstraction/Refinement Architecture Business process Domain wizard Quality of Service

3. A DSL for producing tools for model diagram generation

This section presents the case study of a QVT DSL for producing tools generating model diagrams, which has been developed by DAE, a Department of the Thales Software Research Group.

3.1 Paradigm shift from code to DSL with Software Factories

A few years ago, we started to manually develop tools generating model diagrams. After the development of several one of these, we noticed duplications, redundancies in code, and non reliable schedule. From a first refactoring emerged a diagram framework. This framework capitalized common practices; it reduced but did not prevent the same error-prone defects of manual developments. Then, we therefore decided to use MDSoFa [14], a DAE software factory tool. The objective was to realize a paradigm shift from a handcrafted to an industrialized development.

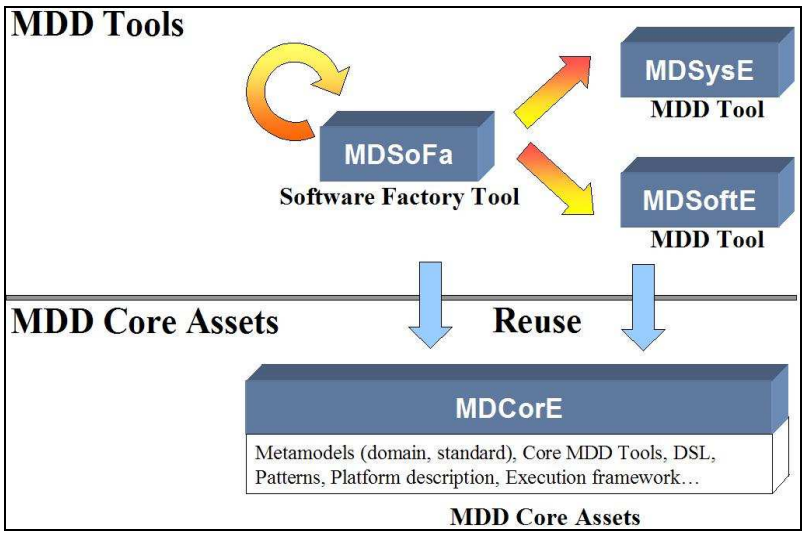

Figure 1. MDD Tool Architecture

MDSoFa is a software factory tool for producing MDD tools in series. It generates for instance the infrastructure of two large-scale MDD tools: MDSysE [8][16], a modeling tool for system-engineering, and MDSoftE, a modeling tool for software engineering. For improving reusability, MDSysE and MDSoftE, as well as MDSoFa, share a set of common core assets, gathered in a common product called MDCorE. All of these MDD tools are in line with the product line approach. MDSysE and MDSoftE are variations of MDCorE core assets. These core assets and the product lifecycle are managed by MDSoFa. A Diagram DSL has typically its place in MDCorE because it can be fitted to different MDD tool contexts.

3.2 Diagram DSL representation and usage

A first key point is to understand the relationship between a Diagram DSL and diagrams, and the relationship between a Diagram DSL and a Diagram DSL tool.

3.2.1 A 3-level architecture

Regarding the relationship between a Diagram DSL and diagrams, we have adopted a 3-level architecture (Figure 2), as MOF. The D2 level represents the language for describing a QVT domain, the D1 level a description of this QVT domain to be applied at the model level, and the D0 level the application of a D1 description at the model level. In the case study, the QVT domain is the diagram management with the following levels.

D2 – Diagram DSL	At the D2 level, the Diagram DSL represents the language for describing any type of diagrams. It is solution- and platform-independent and contains all criteria understandable by a user who wants to specify diagrams. This level is problem-oriented for specifying diagrams.
D1 – Diagram DSL instance	At the D1 level, a Diagram DSL instance describes a type of diagram. It contains the view model description for producing a type of diagram, that is model elements to be displayed with their layout properties. This description respects the language defined by the Diagram DSL. This level contains all data for generating tools producing diagrams.
D0 - Diagram	At the D0 level, we have diagrams expected by end-users in their modeler.

The Diagram DSL is described by a model which defines the grammar for model diagrams. The one we have developed contains simply four classes: 1/ *Diagram root* gives information for starting diagram generation, 2/ *Node* specifies model elements displayed in diagrams with their layout, 3/ *Navigation* specifies navigation in model, 4/ *Condition*, complementary to Navigation, specifies model element selection. Attributes represent Diagram features, *e.g.* a color of a model element type. Associations between classes declare possible relationships, *e.g.* a Node can contain Nodes, but a Node cannot contain a Diagram root. At a Diagram DSL instance, there is one Diagram Root element by type of diagram. The Nodes and Navigations describe the successive navigations in model and how model elements are displayed in the diagram.

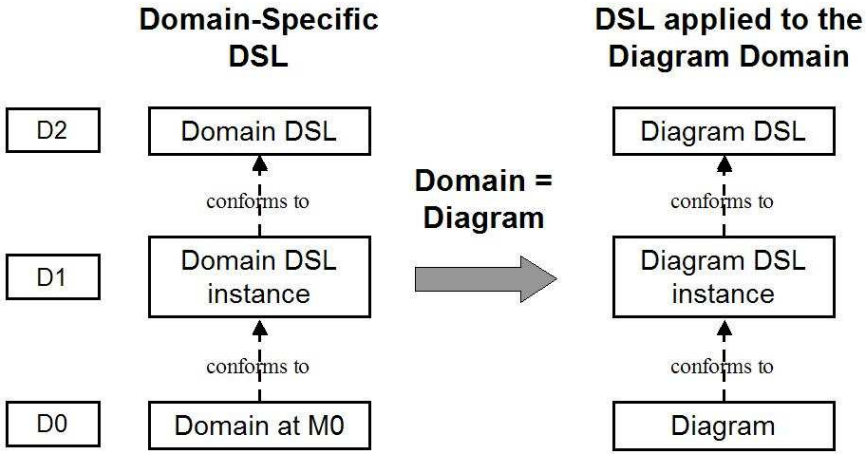

Figure 2. Diagram DSL levels

3.2.2 Diagram DSL and Diagram DSL Tool relationship

A Diagram DSL tool is the tool which puts the Diagram DSL in action. The principle for managing a Diagram DSL instance with a Diagram DSL tool is the same than for editing a domain model, such as MDSysE. Instead of editing a model, a Diagram DSL tool manages diagram DSL instances conforming to the Diagram DSL. This means the DSL tool is always consistent with the language it implements, the Diagram DSL. Thanks to this conformance and with a full generation adoption, the Diagram DSL tool can be generated from a "Diagram DSL to Diagram DSL tool" translation. Therefore, when Diagram DSL properties change, the DSL tool can be generated for a Diagram DSL / Diagram DSL tool synchronisation.

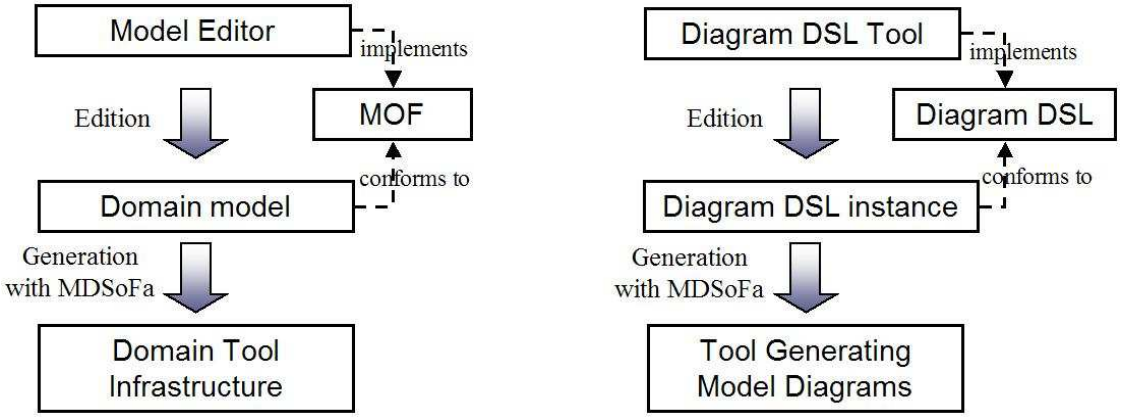

Figure 3. Analogy of domain and DSL models

3.3 Diagram DSL Lifecycle

This sub-section explains the process, which is depicted in the following figure, intending to produce tools generating model diagrams from the Diagram DSL definition.

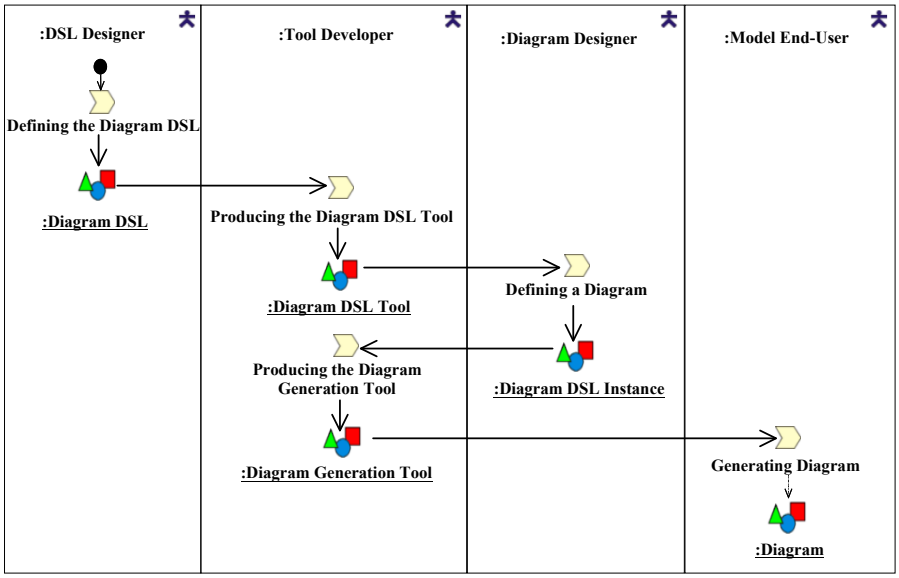

Figure 4. Diagram DSL lifecycle

3.3.1 Defining the Diagram DSL

This activity, effectuated by the DSL Designer, consists in modeling the Diagram DSL. Starting either from code, and abstracting it into the Diagram DSL, or starting directly from the Diagram user viewpoint, eliciting the Diagram DSL is not a straightforward situation. It requires several iterations before finding criteria meaningful for the Diagram Designer. These criteria must be simple, pertinent and complete enough to deduce an implementation solution.

3.3.2 Producing the Diagram DSL Tool

The Diagram DSL tool, which serves to edit Diagram DSL instances, evolves as long as the Diagram DSL does. Several presentations may exist for the same DSL. Its production can be generated from a DSL to Tool transformation or developed manually by a Tool Developer. Here again, the DSL tool must be simple and pertinent. Actually, its utility depends on its ability to increase the Diagram Designer productivity. The progress, in this way, is when the user has no code to write. Even more, it is assisted to reach faster what she/he expects to build.

3.3.3 Defining Diagram

Every Diagram DSL instance realized at this stage contains all criteria for creating and updating every kind of diagram, *e.g.* all MDSysE diagrams (interface, domain element, component, deployment diagrams, etc.). The actor of this activity is really a designer and not a developer.

3.3.4 Producing the Diagram Generation Tool

Every Diagram DSL instance is consumed by MDSoFa to produce a tool generating model diagrams. After its production, the tool is packaged, ready to be deployed and integrated in a largest tool, *e.g.* MDSysE. Diagram generation becomes a function among others.

3.3.5 Generating Diagram

During this activity, the model end-user applies the diagram tool on his model, *e.g.* interface diagrams of the model are generated. Diagrams created or updated conform to a Diagram DSL model, which conforms to the Diagram DSL. For instance, diagram layout reflects layout specifications, which conform to the properties defined in the Diagram DSL.

4. Stepping toward families of QVT DSL and tool

The case study of Diagram DSL focuses on the view aspect. It has been tested on more than 50 types of diagram and is in production with MDSysE. However, from our previous lessons learned with MDSoFa, a main point emerged: that one DSL can be, not necessarily specialized but, tailored in function of the project context. With software factories, this opens the way for families of QVT DSL.

4.1 Need of QVT DSL and tool families

Several reasons justify the need of QVT DSL families.

[N1] From the functionality viewpoint, a QVT DSL in a project context can have more or less properties that an original QVT DSL. Neither specialization nor parameterization is able to support multiple structural modifications, especially for several projects managed in parallel.

[N2] From the process viewpoint, different processes are possible for the same QVT DSL in function of the project context or the adopted methodology.

[N3] From the language viewpoint, it is illusory that one language addresses all types of modeling problems with expressiveness and accuracy simultaneously. The need is the management of variation of "abstract to concrete syntax" transformation. For low level languages, in function of user communities, from the same QVT core language can be derived various forms of programming languages: textual *vs*. graphical, declarative *vs.* imperative, etc. For high level languages, *i.e.* DSLs abstracting the most a software description, a DSL can also adopt various forms: description with formal textual language, wizard, or even with table.

[N4] From the design and implementation viewpoints, the solution can change in function of architectural or non-functional decisions. The problem to solution transformation implies to have variants of generation.

[N5] From the capitalization viewpoint, in order to meet the requirement of durability of the QVT descriptions, the need is the management of the platform variability (standards, languages, frameworks, tools).

[N6] From the reusability viewpoint, different QVT DSLs can share common features. For instance, a view DSL and a model transformation DSL can be expressed in a tree form. Then, the need is to manage common assets which can be reused in different QVT DSL contexts.

This list of needs justifies this interest of QVT families but simultaneously shows its complexity. We can continue to develop QVT tools without product line but reusability and productivity will assuredly decrease when specifications and environment evolve, or when project contexts are multiple. Managing efficiently variability of QVT DSL and tool turns out to be profitable but also a real technical challenge.

4.2 QVT DSL and tool families

The software product line is “a set of software intensive systems sharing a common, managed set of features that satisfy specific needs of a particular market or mission, and that are developed from a common set of core assets in a prescribed way” [3]. This means that core assets are created to be reused in the production of multiple products in order to reduce the cost and the time to produce an individual product [4]. A product line uses two distinct but interacting processes: 1/ the product line development process, which produces reusable assets, and 2/ the product development, which produces products. To continuously improve the core assets, the product line development process uses the feedback of the product development (promotion and improvement of core assets, architecture evolution, production process improvement, etc.).

The main activities for the product line process are: i) for the domain analysis, domain scoping, identification of the commonalities and variability among all products in the family, ii) for the domain design, architecture development, production process and definition of the production plan, and iii) implementation of the core assets for the domain implementation.

Activities of the product development have for objective the production a member of a product family. Analysis, design, and implementation activities select, integrate, and customize the core assets.

4.2.1 A QVT DSL itself as domain

The key point for QVT DSL to reap profit from experience in product line engineering is to consider a QVT DSL itself as a domain. Instead of applying variations on domains such as system or software engineering, variations are applied on QVT domains. Section 3.1 has presented the experience of product line with MDSoFa. MDSysE and MDSoftE are variations of a common domain located in MDCorE, location of MDD core assets. This common domain is defined by a model. Section 3.2 has presented the DSL Diagram defined by a model as well. The way for variations on the Diagram DSL is similar than with the MDSysE and MDSoftE domains. On the other hand, the way for generating the tool producing generating diagrams is the same than for generating the MDSysE and MDSoftE infrastructures. Representation and tooling are the same. The strength of this reflexivity is auto-consistency: end products are built in the same way than the development tool.

4.2.2 Core asset management

The level of reusability and durability of core assets comes from the ability to accept evolutions on core assets from new requirements and product evolutions. This implies a rationale management both of every core asset and the core asset architecture (modularity, asset lifecycle, interactions, etc.).

In the QVT context, a core asset can be:

- A domain description, *i.e.* a description at the D2 level for QVT DSL description, and at the D1 level for QVT DSL instance. A QVT DSL instance is a core asset when this QVT DSL instance can be tailored, or for addressing variations, such abstract to syntax concrete derivations. Refer to needs [N1] [N2][N3] in section 4.1.
- A QVT DSL tool description for QVT DSL tool families. Refer to needs [N1][N2][N3][N4][N5].
- Patterns, generations, frameworks, and tools required for building QVT DSL tools. Refer to need [N4][N5].

- The QVT environment, such as the QVT standard or platform descriptions. Refer to need [N5].

Core asset evolution is a major issue when managing core assets. Four major impacts are possible:

- Impact on domain, with three kinds of impact:
    - Intra-domain impact. Evolutions are located in the same QVT DSL, *e.g.* new layout properties in the Diagram DSL. Refer to needs [N1][N2].
    - Inter-domain impact. Evolutions are located in several QVT DSLs, *e.g.* several DSLs share common elements. Refer to need [N6].
    - Extra-domain impact. A feature is reusable by domains external to QVT DSL domains. For instance, a tree-organization can be used by Diagram and model transformation (inter-domain relationship) but also by interface description.
- Impact on feature model. In this case, the feature model evolves for taking into account new requirements or product evolutions. Refer to needs [N1][N2][N3][N5].
- Impact on architecture. Architecture must be reconsidered for domain, or feature evolutions, but also to take into account pattern, framework, or tool evolutions. Refer to need [N5].
- Impact on production plan. The production plan describes how products are constructed from the core assets. It directly depends on the previous impacts.

Regarding the architecture, Epsilon [7] is illustrative for its tree-organization of QVT languages. EOL is the core language from which other languages can be constructed atop. Epsilon is activity-oriented with languages used for instance for merging, model transformation, or code generation. This tree-organization can be reused for building a hierarchy of QVT languages with QVT DSL and tool variants. Referring to Table 1, one can find core technology, development, and business derivations of DSL but also "core technology to development", "core technology to business" or "business to business" derivations of DSL. The interest of such an organization is the robustness of the foundation: a branch meet a MDD segment (a kind of activity, a technology, a project context, etc.) and each QVT derivation is consistent thanks to a "QVT to QVT" translation.

4.2.3 Product production

A major stage is the production of product from core assets. A production plan describes how to manage this production. It takes into consideration the production method of each asset, the available resources, the linkage among the core assets, and the product line constraints.

Figure 4 describes an example of a Diagram DSL lifecycle. In order to be integrated in a product-line, each activity must declare how to fit into product production contexts. For instance, for a platform variability, the "Producing the Diagram DSL tool" needs to know the target platform, *e.g.* UML version, modeling tool, programming language. Depending on these parameter values, the right generator is selected.

4.3 Issues

Even if the core assets, the production method and the production plan are clarified, in the QVT context, a set of issues must be settled.

1/ Unification of QVT DSL. For complexity reduction and efficiency of MDD tool development, the QVT DSL architecture must be rationalized. We recommend the adoption of a reduced number of QVT DSLs, a tree-organization with few bridges, and reflexivity according to the 3-level architecture principle.

2/ Constitution of uniform QVT DSL workbench. When building MDD tools, a requirement is to have the same ergonomics and logic of action among the DSLs. With a product line, the issue is to keep this uniformity in function with the selected features.

3/ Full automation. The production plan can be a document, partially or completely automated. At a managed maturity model level, automation of the product production is maximized.

4/ Standardization. Families of QVT DSL and tools can be developed by a company for internal or external usage. Despite several open source initiatives and projects on model transformation, the lack of standard for product line prevents any QVT DSL product line in the open source segment.

5. Further Work

Our future work will focus on the development of core assets for building DSL tools, and variants of DSL tools.

6. Conclusion

This paper has presented the case study of a Diagram DSL allowing specification of model diagram. From models conforming to this DSL, DSL tools are produced for generation of model diagrams. These DSL tools are integrated afterward in MDD tools for end-users. Furthermore, in the context of software production with software factories, we figured out that this Diagram DSL could be reused for other purposes, such as model transformation. This clears the way toward families of QVT DSL and tool. We have given in this paper elements for building such families. We believe this is the next step to meet various companies and projects requirements, and to build customized QVT workbenches.

7. Acknowledgments

We thank Serge Salicki, head of DAE of the Thales Software Research Group, the members of DAE, and especially Stéphane Bonnet, Sébastien Demathieu, and Catherine Lucas.

8. References

[1] AMMA (Atlas Model Management Architecture). http://www.sciences.univ-nantes.fr/lina/atl/AM-MAROOT/.

[2] Atlas Transformation Language, official web-site. http://www.sciences.univ-nantes.fr/lina/atl.

[3] Clements, P., Northrop, L., *Software Product Lines, Practices and Patterns*, Addison-Wesley, 2002.

[4] Chastek, G., J., Mc Gregor, J.D., *Integrating Domain Specific Modeling into the Production Method of a Software Product Line*, OOPSLA 2005, Workshop on DSM.

[5] Chauvel, F., Fleurey, F., *Kermeta Language Overview*. http://www.kermeta.org.

[6] Czarnecki, K., and Eisenecker, U.W. *Generative Programming*, Addison-Wesley, 2000.

[7] Epsilon, University of York. http://www-users.cs.york.ac.uk/~dkolovos/epsilon/.

[8] Exertier, D., and Normand, V. MDSysE: A Model-Driven Systems Engineering Approach at Thales. Incose, 2-4 November, 2004.

[9] Fowler, M., *Language Workbenches: The Killer-App for Domain Specific Languages?*, http://www.martinfowler.com/articles/languageWorkbench.html

[10] Gardner, T., Griffin, C., Koehler, J. , Hauser, R. *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*. July, 2003.

[11] Greenfield, J., Short, K., Cook, S., and Kent, S., *Software Factories, Assembling applications with Patterns, Models, Framework, and Tools*, Wiley, 2004.

[12] Hamza, H.S., *On the Impact of Domain Dynamics on Product-Line Development*, OOPSLA 2005, Workshop on DSM.

[13] Kolovos, D.S., Paige R.F., Polack F.A.C., *The Epsilon Object Language (EOL)*, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 2006, Springer, 2006.

[14] Langlois, B., Exertier, D., *MDSoFa: a Model-Driven Software Factory*, OOPSLA 2004, MDSD Workshop.

[15] Langlois, B., Barata, J., Exertier, D., *Improving MDD Productivity with Software Factories*, OOPSLA 2005, First International Workshop on Software Factories.

[16] Normand, V., Exertier, D. *Model-Driven Systems Engineering: SysML & the MDSysE Approach at Thales*. Ecole d'été CEA-ENSIETA, Brest, France, September, 2004.

[17] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*, ptc/04-10-15.

[18] OMG. *MOF QVT Final Adopted Specification*, ptc/05-11-0