Towards a framework for deriving platform-independent model-driven software product lines

Hacia un marco de trabajo para derivar líneas de producto de software dirigidas por modelos independientes de la plataforma

A. Paz[1] and H. Arboleda[2]

ABSTRACT

Model-driven software product lines (MD-SPLs) are created from domain models which are transformed, merged and composed with reusable core assets, until software products are produced. Model transformation chains (MTCs) must be specified to generate such MD-SPLs. This paper presents a framework for creating platform-independent MD-SPLs; such framework includes a domain specific language (DSL) for platform-independent MTC specification and facilities platform-specific MTC generation of several of the most used model transformation frameworks. The DSL also allows product line architects to compose generation taking the need for model transformation strategy and technology interoperability into account and specifying several types of variability involved in such generation.

Keywords: model-driven software product line, platform independent model transformation chain, domain specific language, interoperability.

RESUMEN

Las líneas de producto de software dirigidas por modelos (MD-SPLs) son creadas a partir de modelos de dominio que se transforman, combinan y componen con artefactos reutilizables hasta que finalmente se generan productos de software. Con el fin de generar dichas MD-SPLs, es necesario especificar cadenas de transformación de modelos (MTCs). En este artículo presentamos un marco de trabajo para la creación de MD-SPLs independientes de plataforma. El marco de trabajo incluye un lenguaje de dominio particular (DSL) para la especificación de MTCs independientes de plataforma y facilidades para la generación de MTCs en plataformas específicas a fin de llegar a varios de los *frameworks* de transformación de modelos más utilizados (en la práctica). El DSL, además, permite que los arquitectos de líneas de producto: 1) compongan un proceso de generación teniendo en cuenta la necesidad de interoperabilidad de estrategias y tecnologías de transformación de modelos, y 2) especifiquen varios tipos de variabilidad en dicho proceso de generación.

Palabras clave: líneas de producto de software dirigidas por modelos, cadenas de transformación de modelos, lenguaje de dominio específico, interoperabilidad.

Received: October 25th 2012
Accepted: July 11th 2013

Introduction

Product line engineering has attracted attention recently in what are known as software product lines (SPLs) (Linden, Schmid, & Rommes, 2007). An SPL focuses on creating a software system family through a semi-automatic process that builds individual products from reusable software artefacts, shared by all products, and specific software artefacts only for the product being constructed in accordance with a client's wishes. SPL scope (*i.e.* the range of products a particular SPL may address) is determined by variation amongst the individual systems which can be derived. One way of capturing such variation is by using a variability model (Pohl, Bckle, & van der Linden, 2005) (*e.g.* feature models). Variability models describe what can vary (variation points) in final systems, the options available (variants) for satisfying each variation point and the relationships between them.

Several approaches have been proposed for creating model-driven engineering (MDE) based SPLs. MDE claims to improve software development by using models as first-class artefacts during development (Awais Rashid, Jean-Claude Royer, 2011; Stahl & Czarnecki, 2006). Model-driven SPL (MD-SPL) approaches (*e.g.* Arboleda, Casallas, & Royer, 2009; Rashid, Royer & Rummler, 2011; Santos, Koskimies & Lopes, 2006; Voelter & Groher, 2007; Wagelaar, 2005)) are intended to create SPLs departing from domain models which are transformed, merged and composed with reusable core assets, thereby producing software products (Arboleda & Royer, 2012).

[1] Andrés Paz. Ingeniero de Sistemas, Universidad ICESI, Colombia. Affiliation: Universidad ICESI, Colombia. E-mail: afpaz@icesi.edu.co

[2] Hugo Arboleda. Ingeniero de Sistemas y Computación, Universidad del Valle, Colombia, Magíster en Sistemas y Computación, Universidad de los Andes, Colombia. Doctor en Ingeniería, Universidad de los Andes, Colombia. PhD Informatics, École des Mines de Nantes, France. Affiliation: Universidad ICESI, Colombia. E-mail: hfarboleda@icesi.edu.co

How to cite: Paz, A., Arboleda, H., Towards a framework for deriving platform-independent model-driven software product lines., Ingeniería e Investigación. Vol. 33, No. 2. August 2013, pp. 70 – 75.

Model transformation requires several stages during MD-SPL construction; each stage involved processing domain models to include more implementation details. A model transformation chain (MTC) (Yie, Casallas, Deridder & Wagelaar, 2012) is a sequence of transformation steps using one of these stages to convert a higher-level model rooted closer to the problem domain into a lower-level model rooted closer to the solution domain (e.g. Java, C#). Transformation steps must be adapted throughout each transformation stage to adapt variants chosen by product designers for their particular products; this requires pre-defining how transformation steps must be adapted according to variation points and SPL variants.

There is a significant gap between variability at a conceptual level (variation points and variants) and variability at implementation level (concrete software assets, such as model transformation rules). MTCs must thus be specified for gathering links between variants and transformation steps, including the usual sequence of transformation steps involved in producing SPLs and the variability involved in derivation. MTCs are currently specified in several ways, usually being embedded in model transformation rules, coupling commonalities and transformation step variability or using domain specific (modelling) languages (DSL). The former option is bad practice because of the drawbacks in maintainability and reuse of a product line's core assets whilst DSLs for MTC specification separate the concerns of capturing the transformation logic and capturing their scheduling along with the variation points involved in a particular product line.

This paper presents a framework for creating platform-independent MD-SPLs (platform-independent refers to independence from model transformation platforms and tools). The framework includes a DSL for platform-independent MTC specification and facilitates producing and handling platform-specific MTCs. Previous work by Arboleda, Casallas & Royer (2009) presented tool support for an MD-SPL approach. The MTC language at that time lacked constructs for specifying several types of variability involved in derivation and did not take the need for integrating several model transformation technologies into account. This work presents a solution to such drawbacks.

Illustrative example and DSL key concepts

Illustrative example

An MD-SPL of standalone software products for managing data collections was created to illustrate the problem and give an overview of the aforementioned approach. A detailed description of this MD-SPL can be found in Arboleda, Romero, Casallas & Royer (2009) and on our research group's web site[3]. An example of a product line member would be an application for managing a music store and information regarding songs, such as their titles, artists' names and genre. Another product may manage students from a school and their personal information: name, address, e-mail, etc. These products are usually structured in a kernel component at architecture level (implementing functional requirements for adding elements to the collection and sorting them using various sorting algorithms) and/or a graphical user interface (GUI) component (presenting the information to end-users and interacting with them and the kernel component).

Figure 1 shows an overview of configuring and deriving a collection manager system. A domain meta-model was created which included domain concepts for representing data collection structure. A product designer created a domain model conforming to such meta-model at the start of the derivation process.

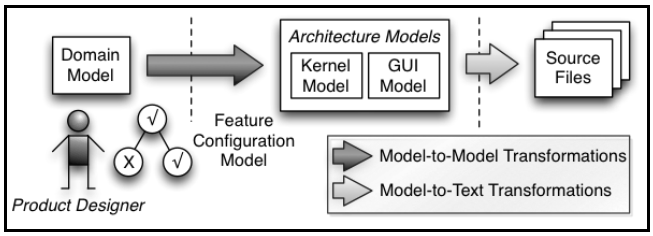

Figure 1. Derivation.

The domain model was automatically transformed at runtime during product derivation into architecture models capturing concepts from the solution space. The architecture models were composed of a kernel model including an aggregation structure to represent the entity being managed and its related attributes, and the concepts for expressing the ability to sort the entities using different algorithms and a GUI model representing GUI elements such as panels, lists, labels and images. These architecture models were then transformed to produce product line members' source files.

Similarly to many other MD-SPL approaches (Santos *et al*., 2006; Tessier, Gérard, Terrier & Geib 2005; Voelter & Groher 2007), feature models were used as core assets for modelling variability during each derivation stage. The example consisted of a feature model built using concepts introduced by Czarnecki, Helsen & Eisenecker (2004). A product designer creates a domain model and a configuration model based on such feature model having a selection of the features to be included in the desired product. This configuration model represents input to the two model transformation stages, one transforming the domain model into the architecture models and the other transforming the latter models into source files.

Figure 2 shows a domain model for a music store system (bottom left) conforming to the domain meta-model for a collection manager system (top left). This paper uses a class diagram-like representation to facilitate intuitive understanding of conformity between models and meta-models. The Figure shows the domain model defining a `musicStore` conforming to the `Domain`metaconcept. The `musicStore` has an entity, `song`, which has three characteristics, `name`, `artist` and `genre`. The feature model (right) represents variants for data sorting using `bubble`, `insertion` or `selection` algorithms.

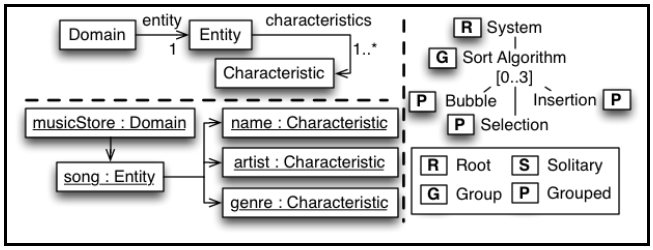

Figure 2. Collection manager example.

DSL Key concepts

Several DSL requirements were identified for specifying an MTC from practice and related work. Such DSL had to define derivation transformation stages, schedule such transformation stages (i.e. the order in which transformation rules process model elements to accomplish desired derivation), capture variation points and variants, represent how variation points can modify

[3]http://www.icesi.edu.co/i2t/driso/mdsplframework

scheduling and enable support for various transformation platforms, techniques or languages.

Transformation stages and scheduling

A meta-model was used for constructing a grammar system specifying our DSL's syntax and semantics. The concepts in this meta-model inherited some names from previous work by Arboleda, Casallas & Royer (2009). The left-hand side of Figure 3 presents a fragment of the meta-model showing the concepts `Workflow`, `TransformationProgram` and `TransformationRule` which target to satisfy the DSL transformation requirement. The second requirement, a sequence of model transformation stages, was named `Workflow`. The `TransformationProgram` concept was used for transforming models, using either the specialised `Model2Model` or `Model2Text` concept. The sequence of model transformation programmes incrementally added design decisions until products were obtained according to end-user choice. A `TransformationProgram` consisted of a set of `TransformationRules`, being sets of transformation instructions.

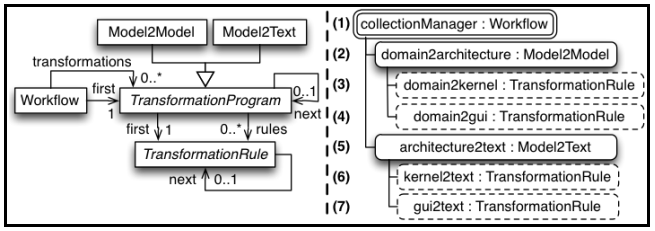

Figure 3. DSL meta-model fragment of the model transformation concepts (left) and fragment of the DSL model for the collection manager showing transformation stages, their transformation rules and sequence (right).

A graphical, tree-like, representation of the models forming our DSL meta-model was used in this paper to facilitate understanding. As an illustrative example, the MD-SPL model for the collection manager at the right-hand side of Figure 3 defines two transformation stages: `domain2architeture` in line 2 is a model-to-model transformation stage departing from the collection manager domain model to create two architecture models specified for derivation (Kernel and GUI). The transformation rules in this stage were `domain2kernel` in line 3 and `domain2gui` in line 4. The second transformation stage, `architecture2text` in line 5, was a model-to-text transformation stage. The architecture models in the previous stage were taken as input and transformed into source code. The transformation rules in this stage were `kernel2text` in line 6 and `gui2text` in line 7. The example was summarised to simplify all the transformation rules required in both stages.

Variation points, variants and scheduling modification

A sequence of variation points was also needed to satisfy variant and scheduling modification since transformation stages may include variants conditioning transformation rule execution.

Figure 4 shows that a `TransformationProgram` also contained a sequence of `VariationPoints` involved in generation. Concerning the ability to represent the way variation points can modify scheduling, variation point applicability was determined by a `Configuration`, which was as a set of `Variants`. Each `Variant` represented a feature-state pair. The `FeatureModel` concept covered collecting available features to be taken into account in a configuration. Our DSL did not consider modelling these features and their mutual relationships and constraints; however, this is planned for future work. pure::variants[4] were used instead, one of the many variability management tools available.

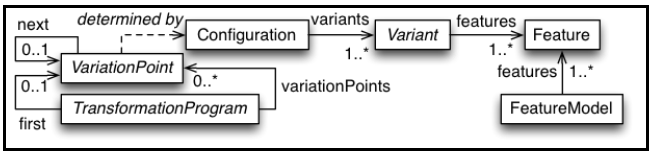

Figure 4. DSL meta-model fragment of variability modelling concepts.

Concepts were also included to represent how scheduling model transformations could be modified. According to different configurations, a model transformation may require elements being added to source models, removing elements from them, or transforming such elements into other elements. Groher & Voelter (2007) distinguished what they called positive and negative variability for describing such types of model operation; the present document has been based on such work.

Positive variability. Positive variability builds products from a minimal set of common elements. Additional elements were added to this set according to each particular product's configuration. Two strategies were distinguished for positive variability: model-oriented and programme-oriented.

Model-oriented positive variability. Model-oriented positive variability merges, or weaves, two separate models into a single model. One of these models holds the information regarding elements to be added and where such elements should be added. The other model acts as target. Figure 5 (bottom right) shows how model-oriented variability was captured in our DSL. A `base` model was taken as target and an `aspect` model contained the elements to be added and where they had to be placed. Both models were woven into a `result` model.

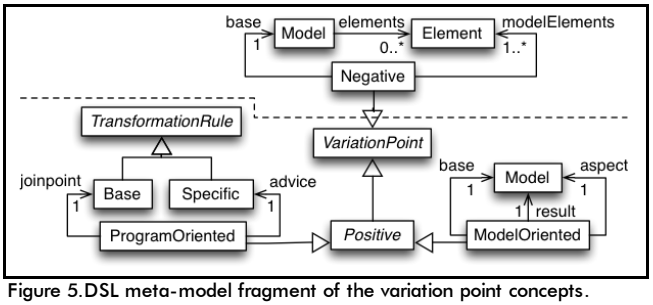

Figure 5. DSL meta-model fragment of the variation point concepts.

A source of variation in the collection manager concerned data-sorting being provided by a bubble, an insertion and/or a selection algorithm. Figure 6 (left-hand side) presents a model-oriented positive variability scenario. The `sortAlgorithm` configuration (line 9) with the `bubble` feature selected (lines 10-11) determined the use of the bubble sort algorithm in the collection manager. The `kernel` model was refined according to such configuration with a model-oriented positive variation point, `createBubble` (line 4) weaving the base model named `kernel` (line 5) with an aspect model named `bubble` (line 6). Such weaving was performed after the `domain2kernel` transformation rule (line 3) had been executed and, thus, the kernel model (line 5) had already been created.

Programme-oriented positive variability. Programme-oriented positive variability relied on the interception of a transformation rule, given a particular configuration, and the execution of an alternative transformation rule, i.e. specific transformation rules

[4] http://www.pure-systems.com/pure_variants.49.0.html. Last visit October 2012.

introducing variability. Figure 5 (bottom left) presents the concepts for representing this type of variability. Base transformation rules are those created for deriving product commonalities. Transformation rules introducing variability were presented as Specific transformation rules; base rules which had to be intercepted by the joinpoint relationship were mapped as were specific rules to be executed instead with the advice relationship.

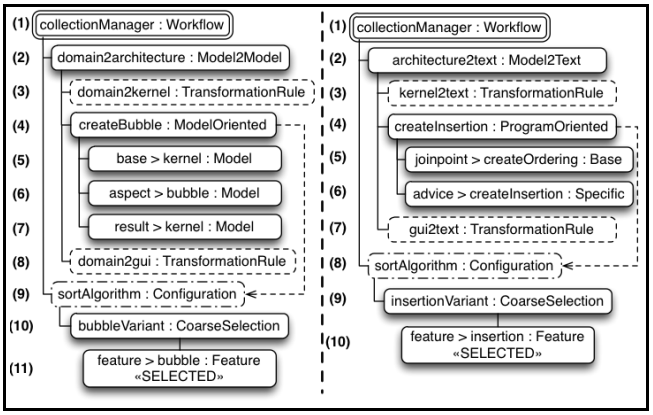

Figure 6. Fragment of the DSL model for the collection manager with model-oriented positive variability (left), Fragment of the DSL model for the music store with programme-oriented positive variability (right).

Figure 6 (right) shows a programme-oriented positive variability scenario. The sortAlgorithm configuration in (line 8) with the insertion feature selected (lines 9-10) determined the use of the insertion sort algorithm. The variation point createInsertion (line 4) intercepted (joinpoint) the base rule createOrdering (line 5) and executed (advice) the specific rule createInsertion (line 6) instead.

Negative variability. Negative variability is the opposite of positive variability; instead of elements being added they were eliminated from a target model by the absence of related features from the configuration for a particular product. Due to space limitation, an example of this case has been omitted.

Inter-operability

Regarding inter-operability, product line architect needed facilities for indicating which technology would be used in each product derivation transformation step. An attribute was thus added to the TransformationProgram concept holding information about the particular technology being used. A platform-independent MTC specified by using our DSL was thus transformed into platform-specific MTCs scheduled by a common mediator.

The MD-SPL framework

A framework for creating MD- SPLs was used as an eclipse rich client platform (RCP)[5] application. Eclipse RCP allowed creating a feature-rich, stand-alone application built upon a plug-in architecture which could be easily extended with additional components.

Architecture from a static viewpoint

Figure 1 presents our framework's high-level components from a static viewpoint. Eclipse modelling framework (EMF) was chosen as the modelling framework, meaning that all our meta-models were based on the Ecore meta-meta-model. Pure::variants were used as our feature modelling framework because this provided a complete variability management solution which has been successfully used in SPL engineering practice. Such framework supported integrating four of the most common model transformation engines: Xtend, Xpand (oAW), ATL and Acceleo. Ant files were used as intermediary between our platform-independent MTC specifications and the platform-specific MTC specifications running on the model transformation engines.

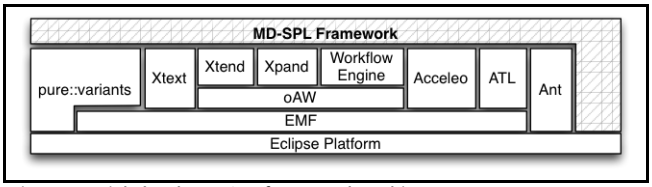

Figure 1. High-level MD-SPL framework architecture.

DSL and generating platform-specific MTCs

Our DSL for specifying platform-independent MTCs was at the core of our framework. Our DSL was defined from the meta-model presented in section 2.2. Product line architects can create MTC specification scripts using our DSL. A text editor was thus provided having syntax colouring, code completion, validation, quick fixes and several other features. List 1 presents a fragment of the MTC specification script built for the collection manager example, capturing the domain2architecture model-to-model transformation stage (line 1). This script was analogous to the model fragments shown in Section 2.2. Here, a source model, domainModel, was transformed into a kernel model by using transformation rule domain2kernel (line 2) and a GUI model by using transformation rule domain2gui (line 4). The model-oriented variation point, createBubble (line 3), was a condition for executing these transformation rules.

List 1.MTC specification fragment for the collection manager example

```
1.  Model2Model domain2architecture {
2.    firstRule := domain2kernel ( domainModel ) ;
3.    firstVariationPoint := createBubble ;
4.    nextRule := domain2gui ( domainModel ) ;
5.  }
6.  ModelOriented createBubble{...}
```

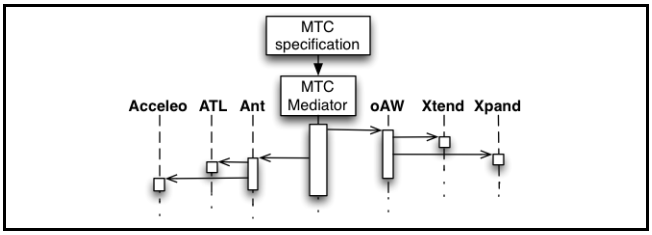

Figure 8. MTC execution flow.

Another feature was also included in our framework: facilities for generating platform-specific MTCs. Our platform-independent MTC specifications were translated into an executable general purpose language (GPL) code to create platform-specific MTCs able to run on particular model engines, ultimately carrying out model transformations. DSL and GPL use was combined for reducing GPL implementation complexity by providing an approach only requiring domain knowledge. MTC specification programmes were thus transformed into executable Ant build files using a model-to-text transformation. The Ant build file contained the required model transformation workflows to derive the configured products (Figure 8). This Ant build file, (named MTC mediator in the Figure) can call a sequence of model transformations through either an oAW workflow, in turn

[5] http://www.eclipse.org/home/categories/rcp.php. Last visit October 2012.

executing Xtend and/or Xpand model transformation, or another Ant build file, in turn executing ATL and/or Acceleo model transformation. The Ant build file on the Ant engine could thus be executed and products derived through an MTC consisting of assets corresponding to different technologies.

Discussion

A meta-model defining efficient expression of the domain concepts and their mutual relationships had to be carefully prepared to build a DSL. A review of the mapping between the modelled abstract entities and DSL syntax and semantics was also necessary. Téllez (2011) proposed using a set of DSL properties as an evaluation mechanism for validating DSLs: representation, absorption, standardisation, abstraction, expressiveness, compression, productivity and quality. It was considered that this mechanism was appropriate for evaluating our DSL; however, more experimentation must be conducted for collecting enough data for quantitative validation.

Representation and abstraction properties were concerned with the DSL's concrete and abstract syntaxes; they were also concerned with its ability to enable users to write unambiguous sentences depicting domain concepts and their relationships. Our DSL was defined using a meta-model representing common abstract concepts for an MTC domain, such as transformation stage, transformation rule, variation point, variant and mutual relationships. These concepts acted as reference for the constructs in our DSL's syntax and semantics. Our DSL included common best practices from MDE and SPL. The reference meta-model restricted our DSL grammar to specific, standard and widely-used MD-SPL constructs.

Our DSL has been used for about a year by our research group members. Two application examples have been developed to date: a collection manager presented in this paper and a smart home. Both examples consist of a set of MD-SPL core assets and an MTC specification built with our DSL. The expressiveness of the language has been enough to specify the decisions involved in them. The smart home example can also be found on the aforementioned website.

The complexity of building an MTC specification was hidden by our DSL. MTC specification programmes written with our DSL were easy to edit and our framework provided facilities for validation and code completion in them. The complexity of an MTC built with our DSL depended on the number of decisions involved and derivation complexity.

Using our framework was successful regarding model transformation technology interoperability and reusing product lines' core assets. The cost of capturing decisions within platform-independent MTCs and maintaining and extending them was significantly reduced. Regarding productivity metrics, experiments involving real industry projects are still lacking for quantitatively validating the adoption of our approach and framework.

Related work

Voelter & Groher(2007) have proposed a similar MD-SPL approach using domain models and complementary feature models to capture variability. Nonetheless, their mechanism for capturing derivation decisions was based on specifying relationships between model transformation rules and variants, which must be manually written in text files and are not easy to manage, adapt and reuse. This approach, unlike ours, was limited to oAWMTC generation.

Similar work by Clafer regarding MD-SPLs has been presented by Bak, Czarnecki & Wasowski (2010). Bak*et al.,* have used feature modelling to capture SPL variability. The variability model then related problem space models to solution space models from another viewpoint. The approach has the advantage of representing both meta-models and feature models using a common construct infrastructure but lacking a product derivation strategy.

Loughran, Sanchez, Garcia & Fuentes (2008) and Sanchez, Loughran, Fuentes & Garcia (2008) have presented VML* and shown that VML* is flexible; however, this requires a development phase prior to MTC specification, thereby limiting and delaying the use of the approach. VML is a platform-specific approach since derivation process consists of a suite of model transformations implemented only in oAW. Our approach has been based on VML's core principles of assembling architectural concerns; however, we propose platform-independent derivation process which can consist of a suite of model transformations implemented in various technology frameworks and which does not require a previous development phase. Heidenreich *et al*., (2010) and Heidenreich, Kopcsek & Wende (2008) have presented a similar approach to VML* called FeatureMapper. FeatureMapper is intended to map the relationships of an EMF-based[6] domain model with a feature model. FeatureMapper only supports negative variability action during one-stage derivation thereby limiting the scope of the products the SPLs can derive. To our knowledge, FeatureMapper is also a platform-specific approach.

Wagelaar (2005) captured SPL variation by creating variability models such as feature models. This approach, unlike ours, only facilitated variation binding prior to executing the model transformation stages, *i.e.* at domain level. This was a limitation on SPL scope as variations cannot be configured on following domains. Wagelaar used Ant build files at the top level of his approach to create MTCs according to the products which had to be derived. This makes the approach particularly difficult to use and also makes MTC maintenance a complex task. We based our framework's mechanism for creating MTCs on Ant build files. However, we added a level on top, which is our DSL, to hide the complexity of adapting MTCs at a low level and to cope with the maintenance and reuse issues involved in the model transformation rules.

Table 1 presents a comparative summary of our approach's characteristics and those of the aforementioned approaches. The requirements were presented in Section 3.2; the approach cited as 1 was ours, 2 was by Voelter & Groher (2007), 3 Bak *et al*., (2010), 4 Sanchez *et al*., (2008), 5 by Heidenreich *et al*., (2008), and 6 by Wagelaar (2005).

Table 1. Comparison with related approaches

Characteristic / approach	1	2	3	4	5	6
Provides a derivation mechanism	x	x		x	x	x
Supports transformation stages during derivation	x	x				
Schedules transformation rules	x	x		x	x	x
Has a dedicated DSL for building MTCs	x	x	x	x		
Manages variability	x	x	x	x	x	x
Captures variation points	x	x	x	x	x	x
Allows scheduling modification through actions or operations	x	x		x	x	
Supports more than one variability strategy (*e.g.* positive, negative) in MTC specifications	x			x		
Allows technology interoperability	x					
Is platform-independent	x					

[6] http://www.eclipse.org/modeling/emf/. Last visit October 2012

Conclusions

We have presented a framework for creating platform-independent MD-SPLs. We have introduced a DSL as part of our framework, enabling product line architects to capture the scope of product lines by adapting and composing model transformations with different implementation technologies according to configurations. Product line architects can use our framework to specify the variability involved in generation. Thus, transformation and composition logics were decoupled, facilitating the traceability management of variants and their related transformation rules to improve SPL evolution and maintenance. Part of our framework involved motivating the integration of model transformation technologies for coping with their limitations. We have also compared our work with related approaches, concluding that we have presented a relevant and needed innovation in the field of MD-SPL engineering. We have developed tool support and application examples which are available on our website for the MD-SPL community.

Future work will concentrate on integrating the use of legacy generative development techniques into our framework, such as templates, filtering and frame processing throughout the product derivation process. Legacy generative development techniques like these have been broadly adopted by companies which would want to reuse their already existing artefacts developed with them. This requires integration. Future work will focus on quantitative validation of our proposed approach based on the analysis of data collected from further experiments involving real industry projects.

References

Arboleda, H., Casallas, R., & Royer, J.-C. (2009). Dealing with Fine-Grained Configurations in Model-Driven SPLs. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)* (pp. 1–10). San Francisco, CA, USA: Carnegie Mellon University.

Arboleda, H., Romero, A., Casallas, R., & Royer, J.-C. (2009). Product Derivation in a Model-Driven Software Product Line using Decision Models. In *Proceedings of the 12th IDEAS'09*. Medellin, Colombia.

Arboleda, H., & Royer, J.-C. (2012). *Model-Driven and Software Product Line Engineering* (1st ed., p. 288). ISTE-Wiley.

Awais Rashid, Jean-Claude Royer, A. R. (2011). *Aspect-Oriented, Model-Driven Software Product Lines. The AMPLE Way*. Cambridge University Press.

Bak, K., Czarnecki, K., & Wasowski, A. (2010). Feature and meta-models in Clafer: mixed, specialized, and coupled. In *Proceedings of the Third International Conference on Software Language Engineering (SLE'10)* (pp. 102–122). Eindhoven, The Netherlands: Springer-Verlag.

Czarnecki, K., Helsen, S., & Eisenecker, U. W. (2004). Staged Configuration Using Feature Models. In *Proceedings of the Third Software Product Line Conference 2004* (pp. 266–282). Springer, LNCS 3154.

Groher, I., & Voelter, M. (2007). Expressing Feature-Based Variability in Structural Models. In *Workshop on Managing Variability for Software Product Lines*.

Heidenreich, F., Kopcsek, J., & Wende, C. (2008). FeatureMapper: Mapping Features to Models. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)* (pp. 943–944). Leipzig, Germany: ACM.

Heidenreich, F., Sanchez, P., Santos, J., Zschaler, S., Alférez, M., Araújo, J., … Rashid, A. (2010). Relating feature models to other models of a software product line: a comparative study of featuremapper and VML. In S. Katz & M. Mezini (Eds.), *Transactions on Aspect-oriented Software Development VII* (pp. 69–114). Berlin, Heidelberg: Springer-Verlag.

Linden, F. V. D., Schmid, K., & Rommes, E. (2007). *Software Product Lines in Actions: The Best Industrial Practices in Product Line Enginering*. Springer.

Loughran, N., Sanchez, P., Garcia, A., & Fuentes, L. (2008). Language support for managing variability in architectural models. In *Proceedings of the 7th International Conference on Software Composition (SC'08)* (pp. 36–51). Budapest, Hungary: Springer-Verlag.

Pohl, K., Bckle, G., & van der Linden, F. (2005). *Software Product Line Engineering - Foundations, Principles, and Techniques*. Heidelberg: Springer.

Rashid, A., Royer, J.-C., & Rummler, A. (2011). *Aspect-Oriented, Model-Driven Software Product Lines. The AMPLE Way*. Cambridge: Cambridge University Press.

Sanchez, P., Loughran, N., Fuentes, L., & Garcia, A. (2008). Engineering languages for specifying product-derivation processes in SoftwareProduct lines. In *Proceedings of the First International Conference in Software Language Engineering (SLE'08)*. Toulouse, France.

Santos, A. L., Koskimies, K., & Lopes, A. (2006). A Model-Driven Approach to Variability Management in Product-Line Engineering. *Nordic Journal of Computing*, *13*(3), 196–213.

Stahl, T., & Czarnecki, M. V. K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Tessier, P., Gérard, S., Terrier, F., & Geib, J.-M. (2005). Using variation propagation for model-driven management of a system family. In H. Obbink & K. Pohl (Eds.), *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)* (Vol. 3714, pp. 222–233). Rennes, France: Springer-Verlag. doi:10.1007/11554844

Téllez, L. (2011). *A Domain-Specific Language to Specify Behavior in a Management Game Simulator*. Universidad de los Andes.

Voelter, M., & Groher, I. (2007). Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the 11th SPLC* (pp. 233–242). Ieee. doi:10.1109/SPLINE.2007.23

Wagelaar, D. (2005). Context-driven model refinement. *MDAFA*, 189–203.

Yie, A., Casallas, R., Deridder, D., & Wagelaar, D. (2012). Realizing Model Transformation Chain Interoperability. *Software & Systems Modeling*, *11*(1), 55–75.