

Automated Variability Injection for Graphical Modelling Languages

Antonio Garmendia
Manuel Wimmer
(antonio.garmendia,manuel.wimmer)@jku.at
Johannes Kepler University
Linz, Austria

Esther Guerra
Elena Gómez-Martínez
Juan de Lara
(esther.guerra,mariaelena.gomez,juan.delara)@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Abstract

Model-based development approaches, such as Model-Driven Engineering (MDE), heavily rely on the use of modelling languages to achieve and automate software development tasks. To enable the definition of model variants (e.g., supporting the compact description of system families), one solution is to combine MDE with Software Product Lines. However, this is technically costly as it requires adapting many MDE artefacts associated to the modelling language – especially the meta-models and graphical environments.

To alleviate this situation, we propose a method for the automated injection of variability into graphical modelling languages. Given the meta-model and graphical environment of a particular language, our approach permits configuring the allowed model variability, and the graphical environment is automatically adapted to enable creating models with variability. Our solution is implemented atop the Eclipse Modelling Framework and Sirius, and synthesizes adapted graphical editors integrated with FeatureIDE.

CCS Concepts • **Software and its engineering** → **Software product lines**; Software design engineering;

Keywords Model-Driven Engineering, Product Lines, Graphical Modelling Language, Meta-Modelling, EMF

ACM Reference Format:

Antonio Garmendia, Manuel Wimmer, Esther Guerra, Elena Gómez-Martínez, and Juan de Lara. 2020. Automated Variability Injection for Graphical Modelling Languages. In *Proceedings of 19th International Conference on Generative Programming: Concepts & Experiences (GPCE'20)*. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/1122445.1122456>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GPCE'20, 15–20 November, 2020, Chicago, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . \$15.00

<https://doi.org/10.1145/1122445.1122456>

1 Introduction

Model-Driven Engineering (MDE) [1] promotes models as the main artefacts of the development process. In this software paradigm, models are used to specify, test, simulate, analyse and generate code for the final application, among other activities. By focussing on the essential aspects of the systems to be built, and leveraging on automation, MDE aims at improving the system quality and reducing the development effort [9, 13, 35].

Models are created using modelling languages, which can be general-purpose, like the UML [31], or domain-specific (DSLs) [13, 32]. The latter are specialized languages tailored to a domain, like chatbot development, or mobile user interfaces. DSLs offer high-level primitives that permit achieving more concise, intentional descriptions of the problem than would be obtained using a general-purpose language. In MDE, the abstract syntax of DSLs is defined through a meta-model, which is a model that captures the main primitives of the language, their properties, relations and constraints. Regarding their syntax, DSLs can broadly be classified as textual or graphical. In this paper we focus on the latter.

MDE has proven to be especially effective in narrow, well-understood domains [13], but it may fall short to handle a (potentially large) family of systems. In such cases, one can resort to Software Product Line (SPL) techniques [23] to avoid modelling explicitly each system of the family. SPLs permit specifying variation points on a given model, and hence elude the exponential cost of specifying each model variant separately [2, 29, 30].

While the combination of MDE and SPLs is appealing, its technical cost is high. This is so as enabling the creation of models with variability requires modifying or extending many MDE artefacts, most notably the meta-model and graphical editor of the models. Our work aims at automating this task. For this purpose, we propose a method to declare the elements of the modelling language which admit variability. Then, our method automatically modifies the language meta-model and associated graphical editor to support the specification of model variability. The modified editor is automatically integrated into a feature-oriented development framework to enable the specification of feature

diagrams [11], the selection of configurations, and the generation of their corresponding model variant. We have created a prototype tool, called *VERSO*, which implements these ideas. The tool is built atop the Eclipse Modeling Framework (EMF) [27] and Sirius [26], and produces FeatureIDE [16] plugins integrated with the modified editor.

Overall, this paper makes the following contributions: (i) a method to specify points of variability for models of modelling languages, (ii) an automatic mechanism to adapt existing graphical editors to handle models with variability, and (iii) automated integration of such adapted editors within a framework for feature-oriented development.

Paper organization. Section 2 introduces a motivating running example, and some background on MDE and SPLs. Section 3 presents our approach. Section 4 describes our tool and reports on an initial evaluation. Finally, Section 5 compares with related approaches and Section 6 concludes.

2 Motivation and background

In this section, we introduce a motivating running example (Section 2.1). Then, we explain how DSLs are defined in MDE (Section 2.2) and introduce basic notions of feature-oriented variability (Section 2.3).

2.1 Motivation

Flexible assembly lines are production systems that can be re-configured in different set-ups to produce a variety of goods or adapt to customer demands [19]. We plan to use Petri nets for their modelling, given their rich body of theoretical results for simulation and analysis [17]. This way, we may specify each configuration of the assembly line as a different Petri net. However, the set of configurations may be large, and so, we look for a compact way to represent the commonalities and variations of the assembly line configurations. Product lines are especially suited for this purpose.

However, while we have a graphical editor for Petri nets modelling, the editor does not permit specifying variability as required in this scenario. Modifying the editor by hand to support the definition of Petri nets with variability would be costly, as we would need to: (a) extend the Petri nets language to identify the language primitives that are subject to variability; (b) modify the editor to allow specifying the variants of a Petri model in a compact way; and (c) develop an environment to select configurations and retrieve specific Petri net variants. To facilitate these tasks, our objective is to devise automated mechanisms to adapt existing graphical editors for a modelling language (e.g., the Petri nets editor) to enable the definition of model families.

In addition to modelling editors, MDE solutions also comprise artefacts like model transformations enabling e.g., model verification or simulation. These could still be applicable on the individual models of the family, but it is normally more effective to *lift* these computations to the family level [28].

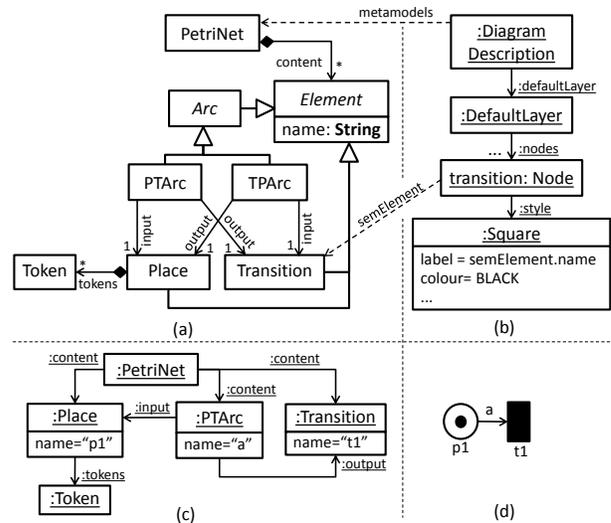


Figure 1. (a) Petri nets meta-model. (b) Excerpt of graphical concrete syntax definition. (c) Petri net model in abstract syntax. (d) Petri net model in concrete syntax.

Approaches have been proposed to lift model transformations to work on models with variability [4, 24, 33], and our goal is to support these automatically on widely used model transformation languages like ATL [10] or the Epsilon languages [21]. However, as a first step in this vision, here we focus only on injecting variability in modelling languages.

2.2 Defining Modelling Languages in MDE

Modelling languages comprise abstract syntax, concrete syntax and semantics [1]. The abstract syntax declares the primitives of the language, including their attributes, relations and integrity constraints. In MDE, the abstract syntax is defined via a meta-model, typically expressed as a class diagram with OCL constraints [1].

Figure 1(a) shows the meta-model for Petri nets in our example. It contains classes to represent Places, Transitions, Tokens, and directed Arcs between places and transitions (and vice versa). A container class `PetriNet` groups the elements belonging to the same net. Places, Transitions and Arcs have a name, while Places have a marking represented by contained Tokens. Figure 1(c) shows a model that *conforms to*, or is an *instance of*, the meta-model. The model contains one place that is connected to a transition and has one token.

To better convey the meaning of models, they are provided a concrete syntax with the rendering of the abstract syntax elements. The concrete syntax can be either graphical or textual; in this paper, we focus on graphical syntax. A typical approach to specify the graphical syntax for a modelling language is by establishing mappings between its meta-model elements and graphical primitives [13].

Figure 1(b) shows an excerpt of the graphical syntax definition for Petri nets. Our approach is independent on specific

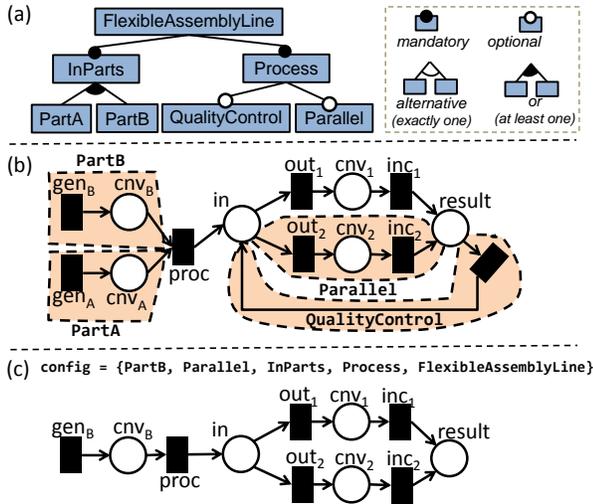


Figure 2. (a) Feature model with the variability of the flexible assembly line. (b) Flexible assembly line with variability annotations. (c) Product derived from a configuration.

graphical editor definition frameworks, but we use Sirius [26] since it is widely used within Eclipse. This way, the concrete syntax definition assigns a `DiagramDescription` to the root class of the meta-model (i.e., `PetriNet`), and a black Square to Transition, labelled with the transition’s name. The assignment of graphical elements to meta-model elements is done via cross references (e.g., reference `semElement` from transition to Transition). Figure 1(d) shows the model depicted in Figure 1(c) using the defined concrete syntax.

2.3 Annotative, Feature-Oriented Variability

To model the variability space of a family of models, the approach proposed in this paper relies on the use of feature diagrams [11]. Figure 2(a) depicts the feature diagram for our example assembly line. It defines the *features* that a model of the family may have (PartA, PartB, QualityControl, etc.), and how they can be combined to obtain a valid *feature configuration*. In the figure, the diagram requires choosing at least one of PartA and PartB, while QualityControl and Parallel can be selected or not. Overall, this feature diagram allows 12 valid feature configurations for the assembly line.

Our proposal follows an annotative approach to define a model family [12]. Specifically, a model family is represented by a so-called *150% model* that contains all elements that may appear in models of the family. Such elements may be annotated with presence conditions (PCs), that is, logic formulae over the features defined in the feature diagram. Figure 2(b) shows an example 150% model. We enclose the elements with same PC using a dashed region.

Given a 150% model and a valid configuration, the corresponding *product* model is built by keeping the 150% model elements whose PC evaluates to true, and deleting those whose

PC evaluates to false. Please note that, when we deleted one object, we deleted its input/output references as well. As an example, Figure 2(c) shows the product Petri net that results from selecting features PartB, Parallel, InParts, Process and FlexibleAssemblyLine; and discarding PartA and QualityControl.

3 Injecting variability in graphical DSLs

This section explains how to customize the variability for a given graphical modelling language. Section 3.1 deals with the abstract syntax and Section 3.2 with the concrete one.

3.1 Variability on the Abstract Syntax

Designing the variability for a DSL requires considering two aspects. First, deciding which elements can be subject to variability. For example, some works on product lines of Petri nets only permit variability either on transitions or on arcs [18] in order to enable an efficient analysis. Second, there may be the need to specify well-formedness rules among the PCs of different elements. For example, the PC of an arc should be stronger than the PC of its input and output place and transition. That is, there should be an implication from the PC formula of the arc to the PC of its inputs and outputs, since this ensures that any product net that contains the arc will also contain its adjacent elements.

To specify these two variability aspects, in a first step, we permit selecting the classes of the language meta-model whose instances are enabled to have PCs. If the instances of a class A are enabled to have PCs, then so are the instances of all direct and indirect subclasses of A. Moreover, to ensure consistent models, if there is a composition relation from a class A to a class B, we require the following conditions:

1. If A is PC-enabled, then so must be B. This avoids producing models where the instances of B lose its container object.
2. If both classes are PC-enabled, we assume that the PC of the instances of B is the conjunction of their own PC with the PC of their container A object. This ensures that the PC of the B objects is stronger than the PC of their containers, and prevents B objects from losing their containers in products.

Figure 3 (a–c) illustrates the rationale for condition (1). If Token would not be allowed to have PCs, but their container Places are (cf., (a)), then we could end up generating products where Token objects would be outside their containers. Hence, condition (1) forces Token to have PCs, while (2) ensures that we cannot have Tokens outside Places.

In a second step, we can use the predicate *stronger-than*(A, B, ref) to require the PC of any instance of class A to be stronger than the PC of any instance of class B reachable via the reference ref, where ref can go from A to B. Formally:

$$\text{stronger} - \text{than}(A, B, \text{ref}) \equiv \forall a \in A, \forall b \in B \cap a.\text{ref} \bullet PC(a) \implies PC(b)$$

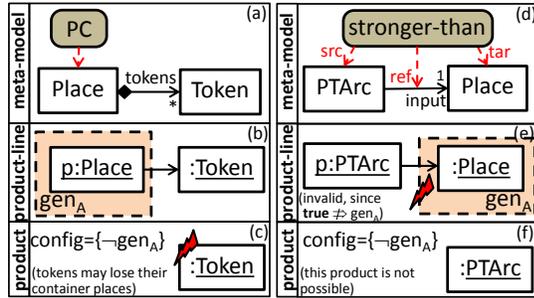


Figure 3. (a–c) Restrictions for compositions. (d–f) Illustration of the stronger-than predicate.

where a and b are objects of types A and B . In the predicate, b is required to be reachable via $a.ref$, and if so, an implication between the PCs of a and b is required.

We automatically add this predicate whenever there is a reference with same lower and upper cardinality bounds (e.g., 1..1), as in this case, the PC of the reference source needs to be stronger than the PC of the reference target to avoid violating the reference cardinality in products.

Figure 3(d–f) illustrates the working scheme of the stronger-than predicate. In (d), we configure PTArc to have stronger PC than Place. This disallows a variability specification like in (e), since it would lead a product like (f), where the minimum cardinality constraint if input would be violated.

Example Figure 4 shows the variability inscriptions on our example meta-model. Two classes are enabled to have PCs: Element and Token. Consequently, all concrete subclasses of Element become PC-enabled as well (PTArc, TPArc, Place and Transition). Since both Place and Token are PC-enabled, and there is a composition relation between them, the PC of the instances of Token will be conjoined with the PC of their container Place. This avoid Tokens outside Places. PetriNet instances are not allowed to have variability. Finally, predicate stronger-than is used four times to ensure that the PC of the arcs (either PTArc or TPArc) is stronger than the input/output Place/Transition. These predicates are added automatically to avoid arcs with undefined input or output in product nets.

The variability points so defined in the language meta-model are enforced at the model level. For this purpose, we use the in-house meta-model shown in Figure 5. This meta-model is instantiated to define PCs on the model elements. Specifically, each PC (class PresenceCondition) has a cross-reference to the model object it annotates (reference elements), and is checked for conformance according to the variability specification. This way, only the instances of PC-enabled classes can have variability and define PCs. In addition, we check that the defined stronger-than predicates and the constraints derived from composition relations hold.

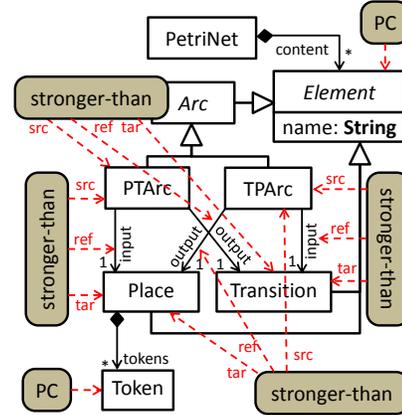


Figure 4. Specifying the variability for Petri net models.

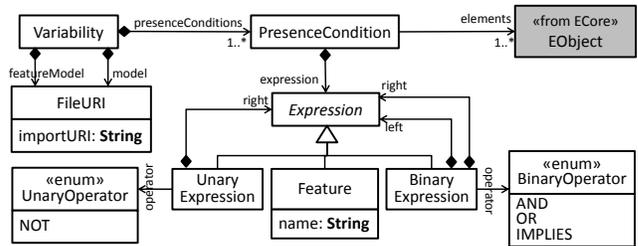


Figure 5. Variability meta-model.

3.2 Variability on the Concrete Syntax

Once we customize the variability allowed in the models of a modelling language, our approach automatically updates the language concrete syntax to enable the attachment of PCs to the instances of PC-enabled classes. In particular, given the language meta-model with the variability inscriptions and a specification of the concrete syntax, our approach adapts the latter to permit a relation between the graphical representation of a PC, and the model elements that may be subject to it. The PCs are specified in a dedicated layer added to the concrete syntax, and provides a palette for creating PCs and a textual editor for their editing.

The PC abstract syntax model is serialized separately from the language model. The cross references from the PCs to the model elements are based on the elements' id or name. Section 4.1 provides more details about the implementation of this process in our tool.

4 Tool Support and Evaluation

In this section we describe tool support (Section 4.1) and report on a preliminary evaluation (Section 4.2).

4.1 Tool Support

We have built tool support to automatically adapt existing graphical Sirius editors to allow the definition of models with variability. We target Sirius [26] as this is a widely

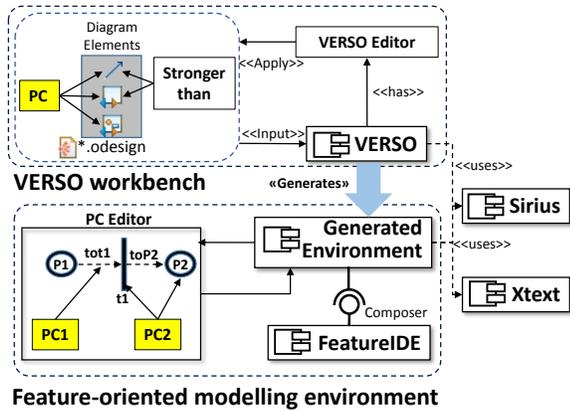


Figure 6. Architecture of VERSO.

used, up-to-date framework for building graphical languages within Eclipse. Our tool is an Eclipse plug-in called VERSO (Variability injeCtoR for Sirius editOrs) available at <https://github.com/antoniogarmendia/ecore-product-line>.

Figure 6 shows the architecture of VERSO. Its main component is the VERSO editor, which receives a model with the concrete syntax definition of an existing Sirius editor as input (extension **.odesign*). Then, the editor permits configuring the classes that can have PCs attached and defining stronger-than predicates. This information is back-propagated to the meta-model with the abstract syntax definition. From this information, VERSO extends the editor with a *Presence Condition* layer that is used to store the PCs of the models being edited, and can be (de-)activated on demand to hide or show these PCs. Moreover, the adapted graphical editor integrates a textual editor¹ that permits defining PCs textually and provides code completion and feature name resolution.

In addition, VERSO generates a FeatureIDE plug-in [16] which supports the specification of feature models, selecting configurations and producing the corresponding product models. For this purpose, an automatically generated *composer* synthesizes the appropriate product model given a 150% model and a configuration.

Figure 7(a) shows a screenshot of the tool being used to inject variability to an existing Petri net editor. Label 1 shows the initial Sirius model (**.odesign*) and the variability definition model (**.pcdef*). The former describes the existing graphical editor for Petri nets, and the latter corresponds to the model created by the user using the VERSO editor (label 2). This model has two tables, one identifying the PC-enabled classes, and the other with the stronger-than predicate.

From this specification, VERSO synthesizes the variability-enabled editor shown in Figure 7(b). Label 3 is the Eclipse plug-in generated, and label 4, the modified graphical editor. The PCs are represented as notes attached to the graphical elements, and an embedded Xtext editor (label 5) permits

Table 1. Evaluated case studies

DSL	<i>.odesign</i> size	MM size	# PCs	# stronger-than
Farming DSL	248	32	7	2
MoSaRT	2129	165	5	2
Ecore	1089	53	5	1

writing them. This editor has autocompletion, and checks that the PCs are well-formed and only make use of features from the feature diagram (label 6). Moreover, the plug-in validates the conformity of the stronger-than predicates. In the figure the editor would report an error as the PC of the arcs is weaker than the PC of their input and output nodes. The editor is embedded within FeatureIDE, which allows creating the feature diagram and selecting a configuration. In addition, a generated language-specific composer permits producing the models for each configuration (label 7).

4.2 Evaluation

Next, our goal is to assess whether VERSO is applicable to (potentially complex) editors and languages built by third parties. We applied our approach to two language workbenches provided by the Sirius Gallery² and to the Ecore standard graphical meta-model editor.

Table 1 summarizes the complexity of the concrete and abstract syntax (number of objects and classes) and the variability specifications, and Figure 8 shows some screenshots of the synthesized editors. The first case study is the Farming DSL, which has two diagram types, and contains 4 meta-models to support farming modelling requirements. We added the PCs to nodes within the Structure diagram representing the agricultural exploitation and the distribution of surfaces. The MoSaRT [20] DSL provides graphical editors to model and analyse real-time systems. In this case, we enabled the definition of PCs in the Software Operator Diagram, where task activities can be created to define the system behaviour. Finally, we extended EcoreTools [3], which has a graphical editor to design domain meta-models.

Overall, we were able to deal with large DSLs (e.g., the concrete syntax of MoSaRT has more than 2000 objects, and its meta-model has more than 150 classes). None of the automatically adapted editors required manual adjustment. Interestingly, we could apply our technique to the Ecore meta-model editor, since Ecore meta-models are persisted like models. This enables the definition of language families.

5 Related Work

Many authors have proposed ways to combine MDE and SPLs. For example, variability has been applied at the model level for specific languages, like uses cases [6], Petri nets [18], AutomationML [36] or Statecharts [15]. Our approach could

¹VERSO relies on the Sirius+Xtext integration in <https://bit.ly/2ZdFWAG>

²<https://www.eclipse.org/sirius/gallery.html>

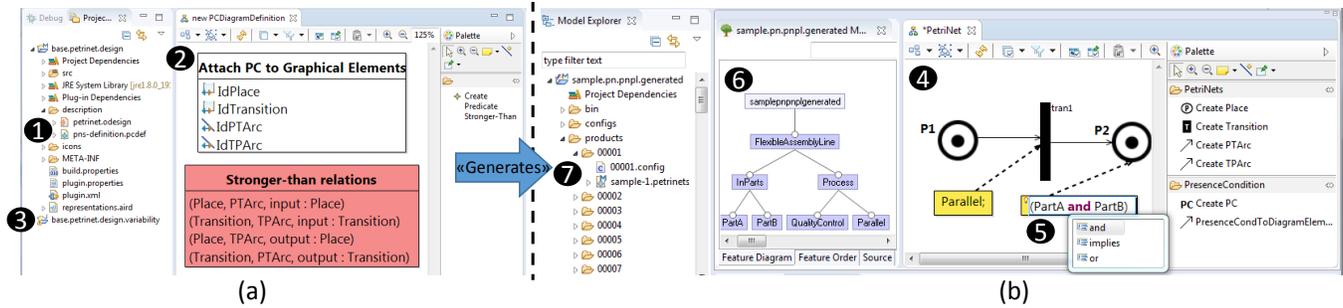


Figure 7. Screenshot of VERSO (a) and generated editor integrated within FeatureIDE (b).

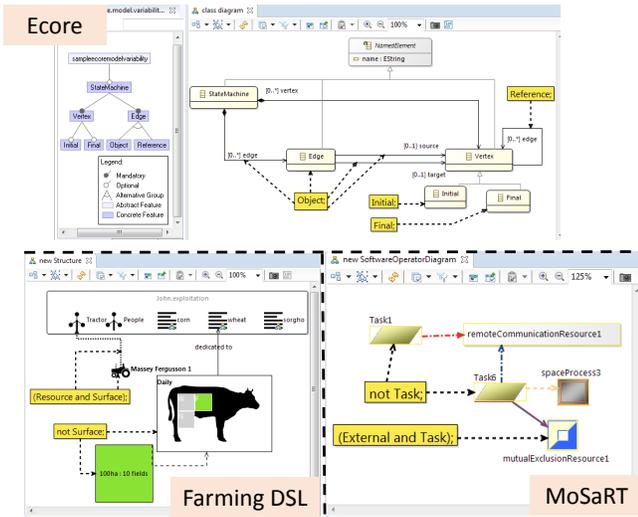


Figure 8. Screenshots of the PC-enabled graphical editors.

have been used to inject variability to these languages, instead of using an ad-hoc approach.

Other authors define variability for the modelling languages themselves. For example, White [34] defines product lines of modelling languages to enhance their reusability across projects, while in [5], techniques were developed for the instantiability analysis of language families. Instead, our variability is at the model level, and we adapt the corresponding graphical model editor. Moreover, since meta-models are models too, our technique can also be used to adapt *meta-model* editors, as seen in Section 4.2.

We use an annotative (also called negative) approach to define the variability. However, other approaches exist, like DeltaEcore [25], VML* [37], CVL [7, 8] or the weaving approach of [22]. The former supports defining variability over models of a given ecore using a transformative approach. VML* automates the definition of variability in arbitrary modelling languages. CVL uses aspect-orientation to inject variability on models of arbitrary DSLs, while [22] uses weaving techniques. While these approaches target the abstract syntax, VERSO adapts the concrete syntax as well.

Regarding graphical languages, in [14], the authors define product lines of graphical editors, so that a particular product editor can be obtained from a configuration. Instead, we adapt existing editors to enable the definition of models with PCs. This is done by defining variation points in the DSL.

In general, a rich combination of MDE and SPLs not only requires adapting the modelling editors, but also needs to inject variability in all sorts of model management programs. In this line, transformation and code generation approaches have been adapted to work over models with variability [4, 24, 33]. However, we are not aware of approaches dealing with the adaptation of graphical editors.

6 Conclusions and Future Work

In this paper, we have proposed a method to inject variability in graphical editors of modelling languages. The approach facilitates the use of product lines techniques within model-based development approaches. Our method has been realized within Eclipse/EMF, and is able to adapt existing Sirius editors, synthesizing a feature-oriented graphical environment on top of FeatureIDE.

We are currently seeking ways to improve the comprehensibility of complex models with variability. We are exploring the use of (dynamically created) graphical layers able to e.g., display only the part of the model with no PC, or with PCs involving certain features. In the future, we would like to adapt other MDE artefacts to deal with variability, like textual editors, code generators and transformations. We would also like to provide means to check the validity of integrity constraints at the family level, in the style of [2]. Our approach targets variability at the model level, but variability at the meta-model level can be useful as well, e.g., to enable the generation of product lines of editors [14]. We plan to study other forms of variability, since PCs alone do not allow, e.g., specifying different attribute values.

Acknowledgements

Work funded by by the R&D programme of Madrid (S2018/TCS-4314), and the Spanish Ministry of Science (RTI2018-095255-B-I00).

References

- [1] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition*. Morgan & Claypool Publishers.
- [2] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. of GPCE'06*. ACM, New York, NY, USA, 211–220.
- [3] EcoreTools. [n. d.]. EcoreTools. <https://www.eclipse.org/ecoretools/>. (last accessed in May 2020).
- [4] Sandra Greiner and Bernhard Westfechtel. 2018. Evaluating Multi-variant Model-To-Text Transformations Realized by Generic Aspects. In *MODELSWARD'18*, Vol. 991. Springer, 82–105.
- [5] Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. 2020. Property satisfiability analysis for product lines of modelling languages. *IEEE Transactions on Software Engineering* (2020), 1–20.
- [6] Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. 2018. Configuring use case models in product families. *SoSyM* 17, 3 (2018), 939–971.
- [7] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *SPLC*. IEEE Computer Society, 139–148.
- [8] Øystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. CVL: common variability language. In *SPLC'13*. ACM, 277.
- [9] John Edward Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Program.* 89 (2014), 144–161.
- [10] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Sci. Comput. Program.* 72, 1-2 (2008), 31–39.
- [11] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. SEI, Carnegie Mellon University.
- [12] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in software product lines. In *ICSE*. ACM, 311–320.
- [13] Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons.
- [14] Thomas Kühn, Kevin Ivo Kassin, Walter Cazzola, and Uwe Aßmann. 2018. Modular feature-oriented graphical editor product lines. In *SPLC'18*. ACM, 76–86.
- [15] Michael Lienhardt, Ferruccio Damiani, Lorenzo Testa, and Gianluca Turin. 2018. On checking delta-oriented product lines of statecharts. *Sci. Comput. Program.* 166 (2018), 3–34.
- [16] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. 2017. *Mastering software variability with FeatureIDE*. Springer.
- [17] T. Murata. 1989. Petri Nets: Properties, Analysis and Applications. *Proc. IEEE* 77, 4 (1989), 541–580.
- [18] Radu Muschevici, José Proença, and Dave Clarke. 2016. Feature Nets: behavioural modelling of software product lines. *SoSyM* 15, 4 (2016), 1181–1206.
- [19] H. Nabi and T. Aized. 2019. Modeling and analysis of carousel-based mixed-model flexible manufacturing system using colored Petri net. *Adv. in Mech. Eng.* 11, 12 (2019), 1–14.
- [20] Yassine Ouhammou, Emmanuel Grolleau, Michaël Richard, Pascal Richard, and Frédéric Madiot. 2015. Mosart framework: a collaborative tool for modeling and analyzing embedded real-time systems. In *CSDM'15*. Springer, 283–295.
- [21] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nikolaos Drivalos, and Fiona A. C. Polack. 2009. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *ICECCS'09*. IEEE Computer Society, 162–171.
- [22] Gilles Perrouin, Gilles Vanwormhoudt, Brice Morin, Philippe Lahire, Olivier Barais, and Jean-Marc Jézéquel. 2012. Weaving variability into domain metamodels. *SoSyM* 11, 3 (2012), 361–383.
- [23] K. Pohl, G. Böckle, and F. van der Linden. 2005. *Software Product Line Engineering. Foundations, Principles and Techniques*. Springer.
- [24] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. 2014. Lifting model transformations to product lines. In *ICSE'14*. ACM, 117–128.
- [25] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore – A Model-Based Delta Language Generation Framework. In *Modellierung (LNI)*, Vol. 225. GI, Bonn, 81–96.
- [26] Sirius. [n. d.]. Sirius. <https://www.eclipse.org/sirius/>. (last accessed in May 2020).
- [27] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd edition*. Addison-Wesley Professional, Upper Saddle River, NJ.
- [28] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45.
- [29] Salvador Trujillo, Don S. Batory, and Oscar Diaz. 2007. Feature Oriented Model Driven Development: A Case Study for Portlets. In *Proc. of ICSE'07*. IEEE Computer Society, 44–53.
- [30] Salvador Trujillo, Jose Miguel Garate, Roberto Erick Lopez-Herrejon, Xabier Mendialdua, Albert Rosado, Alexander Egyed, Charles W. Krueger, and Josune De Sosa. 2010. Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy. In *Proc. of ECMFA'10 (LNCS)*, Vol. 6138. Springer, 293–304.
- [31] UML 2017. UML 2.5.1 OMG specification. <http://www.omg.org/spec/UML/2.5.1/>.
- [32] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org. <http://www.dslbook.org>
- [33] Bernhard Westfechtel and Sandra Greiner. 2018. From Single- to Multi-Variant Model Transformations: Trace-Based Propagation of Variability Annotations. In *MODELS'18*. ACM, 46–56.
- [34] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt. 2009. Improving Domain-Specific Language Reuse with Software Product Line Techniques. *IEEE Software* 26, 4 (2009), 47–53.
- [35] Jon Whittle, John Edward Hutchinson, and Mark Rouncefield. 2014. The State of Practice in Model-Driven Engineering. *IEEE Software* 31, 3 (2014), 79–85.
- [36] Manuel Wimmer, Petr Novák, Radek Sindelár, Luca Berardinelli, Tanja Mayerhofer, and Alexandra Mazak. 2017. Cardinality-based variability modeling with AutomationML. In *(ETFA)'17*. IEEE, 1–4.
- [37] Steffen Zschaler, Pablo Sánchez, João Pedro Santos, Mauricio Alferez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. 2009. VML* - A Family of Languages for Variability Management in Software Product Lines. In *Proc. of SLE'09 (LNCS)*, Vol. 5969. Springer, 82–102.