Model Transformation Product Lines

Juan de Lara, Esther Guerra
Universidad Autónoma de Madrid (Spain)

Marsha Chechik, Rick Salay
University of Toronto (Canada)

ABSTRACT

Model transformations enable automation in Model-Driven Engineering (MDE) and are key to its success. The emphasis of MDE on using domain-specific languages has caused a proliferation of meta-models, many of them capturing variants of base languages. In this scenario, developing a transformation for a new meta-model is usually performed manually with no reuse, even if comparable transformations for similar meta-models exist. This is a suboptimal process that precludes a wider adoption of MDE in industry.

To improve this situation, we propose applying ideas from software product lines to transformation engineering. Our proposal enables the definition of meta-model product lines to capture the variability within a domain, on top of which transformations can be defined in a modular way. We call this construction *transformation product line* (TPL), and propose mechanisms for their construction, extension and analysis. TPLs are supported by a tool, Merlin, which is agnostic to the transformation language and lifts analyses based on model finding to the TPL. Finally, we report on an evaluation showing the benefits of building and analysing TPLs compared to building and analysing each individual transformation.

CCS CONCEPTS

• **Software and its engineering → Reusability**; *Domain specific languages*; Formal software verification;

KEYWORDS

Product Lines, Model Transformations, Reusability

ACM Reference Format:
Juan de Lara, Esther Guerra and Marsha Chechik, Rick Salay. 2018. Model Transformation Product Lines. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239377

1 INTRODUCTION

Model-Driven Engineering (MDE) uses models and transformations to automate the software process [43]. The syntax of models is defined by meta-models, capturing the relevant aspects of a domain. Often, meta-models are variations of a base language (e.g., state machines) with alternative realizations of some aspect (e.g., transitions represented as classes or references) or with primitives of different expressive power (e.g., parallel regions or nested states) [39]. Different communities have proposed variants of modelling languages tailored to a specific range of problems, like variants of Petri nets [33], state machines [15] or meta-modelling notations [22].

This proliferation of meta-models entails the development of specific transformations (e.g., simulators, optimizers, translators between languages, or code generators) for each of them. However, developing transformations from scratch is costly and error-prone [9]. Although transformation reuse techniques have been proposed [8, 10–12, 26, 45, 50], cost-effective methods to systematically build transformations for language families are missing.

To improve this situation, we bring ideas from Software Product Lines (SPLs) [35] to the construction of model transformations. Specifically, we propose the notion of *meta-model product line* (MMPL) [17] as a compact representation of a family of meta-model variants, each of which can be produced via a feature *configuration*. This way, a model transformation can be defined over an MMPL, giving rise to a *transformation product line* (TPL).

TPLs enable a succinct description of the variants of a transformation for a family of meta-models. They build on two abstractions that facilitate their construction in an extensible manner: *concepts* [11], and *transformation specifications*. Concepts are meta-models capturing the essential elements of a meta-model set (an MMPL). Transformation specifications are typed over a concept and specify the structure (operations, rules) and contracts that each implementation variant needs to fulfil. Hence, they ensure that all variants have the same intent [42]. Scalability of TPLs is achieved by composition techniques [2], so that adding a new meta-model variant only requires adding localized transformation fragments. Finally, meta-model consistency and contract analysis is leveraged to the TPL level, which improves the efficiency compared to analysing each meta-model and transformation variant separately.

We have implemented these ideas in Merlin, a tool agnostic of the transformation language. As illustration, we build TPLs for the Epsilon Object Language (EOL) [23]. We also show experiments indicating benefits of TPLs in terms of size reduction and analysis performance w.r.t. working with individual transformations.

This paper makes the following contributions: (i) a novel notion of TPL that supports the modular construction of transformation variants for a family of meta-models; (ii) analysis techniques applicable to TPLs; (iii) a prototype tool that permits building and analysing TPLs; (iv) experiments that evaluate the benefits of TPLs in terms of succinctness and analysis performance.

Paper organization. Sects. 2 and 3 introduce a running example and background on MMPLs. Next, Sect. 4 defines TPLs and Sect. 5 composition operators. Sect. 6 presents analysis techniques for TPLs and Sect. 7 tool support. Sect. 8 reports on an evaluation. Sect. 9 discusses related research, and Sect. 10 concludes the paper.

2 RUNNING EXAMPLE

As a specific transformation example, assume we would like to build a simulator for a set of Petri net variants [33]. A Petri net is a

MODELS '18, October 14–19, 2018, Copenhagen, Denmark

ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239377

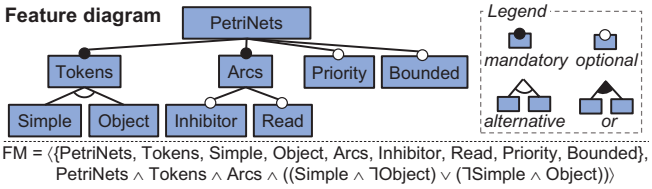


Figure 1: Feature model for variants of Petri nets.

bi-partite graph made of two kinds of nodes: transitions and places. Transitions can have any number of input and output places, while places may contain zero or more tokens. The simulation of a Petri net consists of finding *enabled* transitions and *firing* them. In basic Petri nets, a transition is enabled when every input place has at least one token. Firing a transition subtracts one token from every input place and adds one token to every output place.

Meta-models for Petri nets may represent tokens in different ways (as attributes in places or as a class), and may include or not some Petri nets extensions proposed in the literature [33]. In this example, we consider *inhibitor arcs* disabling a transition if the connecting place has tokens; *read arcs* whose connecting place must have tokens for a transition to be enabled, but maintains the tokens when the transition fires; *transitions with priorities* defining a firing order when several transitions are enabled; and *bounded places* that can hold up to a maximum number of tokens, preventing a transition firing if that makes a place exceed the bound.

Combining these six features yields 32 different types of Petri nets. Creating and maintaining meta-models and simulators for all of them is costly and error-prone. Moreover, introducing a new independent feature (e.g., reset arcs) causes an exponential growth in the number of Petri net variants (64 when adding reset arcs) and their associated simulators. Without proper support, building, maintaining and selecting suitable variants among the set becomes difficult. Hence, we propose TPLs to tackle this problem.

3 META-MODEL PRODUCT LINES

TPLs build on MMPLs. Hence, this section introduces the notions of MMPL, configuration and product derivation, based on an annotative approach from our previous work [17].

Def. 1 (Feature model) A *feature model* $FM = (F, \phi)$ consists of a set of features F and a propositional formula ϕ defining the allowable feature configurations. A feature model $FM' = (F', \phi')$ is said to *extend* FM (written $FM \sqsubseteq FM'$) if $F \subseteq F'$ and $\phi' \Rightarrow \phi$.

Example. Fig. 1 shows a feature model representing the considered types of Petri nets. The upper part uses the feature diagram notation [20], and the bottom part uses Def. 1. The feature model requires choosing between two alternative representations for tokens (Simple or Object), and selecting any combination of Inhibitor, Read, Priority, and Bounded. An extension of the feature model may add new features to F and conjoin additional clauses to ϕ.

MMPLs are defined on top of a standard notion of meta-model [32].

Def. 2 (Meta-model) A *meta-model* is a tuple $MM = (C, FI, I, WC)$ where:

- C is a set of *classes*, some of which may be *abstract*;
- FI is a set of attributes and references called *fields* where each field is owned by exactly one class and has cardinality $[min..max]$;

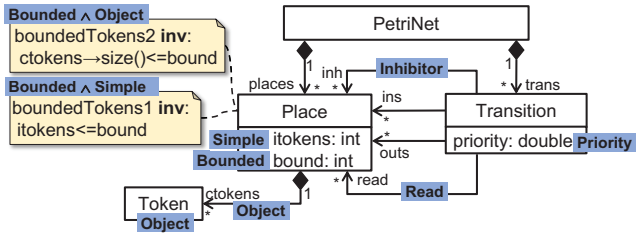


Figure 2: *150MM* for the Petri net variants.

- $I \subseteq C \times C$ is the class inheritance relation;
- WC is a set of well-formedness constraints called *invariants*, each of which is assigned to exactly one class.

In practice, we specify invariants using OCL [36]. Next, we define criteria for meta-model well-formedness.

Def. 3 (Meta-model well-formedness) A meta-model has *a well-formed structure* iff (1) every field is owned by a class, (2) every reference points to a class, (3) cardinality and inheritance are uniquely determined, and (4) there are no inheritance cycles. A meta-model is *well-formed* iff it has a well-formed structure and its invariants are syntactically correct.

A *meta-model product line* (MMPL) allows representing meta-model variants of a family of related languages.

Def. 4 (Meta-model product line [17]) A *meta-model product line* is a tuple $MMPL = (FM, MM, \Phi)$, where[1]:

- $FM = (F, \phi_{MMPL})$ is a feature model;
- $MM = (C, FI, I, WC)$ is a meta-model with well-formed structure, called the 150% meta-model (*150MM* in short);
- Φ is a tuple of mappings $(\Phi_C, \Phi_{FI}, \Phi_{WC})$ from the feature model to the *150MM*. Each mapping Φ_X, for $X \in \{C, FI, WC\}$, consists of pairs $\langle x, \Phi_x \rangle$ mapping an element (a class, a field, or an invariant) $x \in X$ to a propositional formula Φ_x over features, called the *presence condition* (PC). We require that the PC of a field f be stronger than that of its owning class C_i (i.e., we have an implication $\Phi_f \Rightarrow \Phi_{C_i}$), and same for invariants.

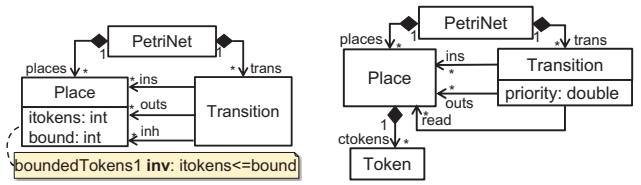
The *150MM* collects all elements in all meta-models of the MMPL. Its elements are decorated with PCs over the features F in FM. An element is present in a meta-model when its PC evaluates to true.

Example. Fig. 2 shows the *150MM* for the running example. It is decorated with PCs, depicted as blue boxes on top of classes, fields and invariants. For example, the attribute Place.itokens is present when the PC Simple is true, while invariant boundedTokens1 appears if the formula Bounded∧Simple is satisfied.

Invariants over the *150MM* must be aware of the structural variability of each variant. In our case, the two alternative representations for tokens requires duplicating invariant boundedTokens to cover both. Sect. 4.1 introduces *concepts* to avoid this problem.

Def. 5 (Feature configuration [41]) A *valid feature configuration* ρ of a product line $MMPL$ with a feature model $FM = (F, \phi_{MMPL})$ is a subset of its features satisfying ϕ_{MMPL}, i.e., ϕ_{MMPL} evaluates to true when each variable $f \in \phi_{MMPL}$ is substituted by true when $f \in \rho$,

[1] We use $MMPL$ (italics) to denote the tuple name, and MMPL for the acronym.



(a) ⟨Simple, Inhibitor, Bounded⟩ (b) ⟨Object, Read, Priority⟩

Figure 3: Meta-models derived from two configurations.

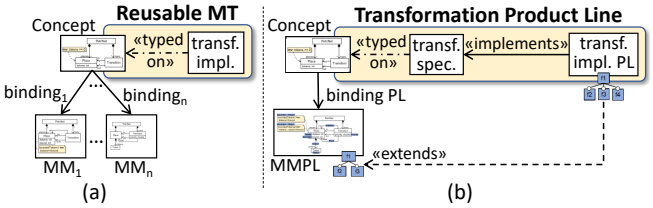


Figure 4: From (a) reusable transformations to (b) TPLs.

and false otherwise. We write $\phi_{MMPL}[\rho]$ to denote this evaluation, and $P = \{\rho_i\}_{i \in I}$ for the set of all valid configurations.

Example. The MMPL in Figs. 1 and 2 admits 32 configurations. Some of them are: $P = \{\langle$Simple$\rangle$, $\langle$Simple, Inhibitor, Bounded$\rangle$, $\langle$Object, Read, Priority$\rangle$, ...$\}$ (where configurations only show leaf features).

Def. 6 (Meta-model derivation) A meta-model MM_ρ is *derived* from a product line *MMPL* using a configuration $\rho \in P$ if MM_ρ contains exactly those elements from the *150MM* whose PCs are satisfied for the features in ρ. We write *Prod(MMPL)* for the set of all meta-models derivable from *MMPL*.

Example. Fig. 3 shows two meta-models derived from the *150MM* in Fig. 2. They correspond to two feature configurations, and contain the classes, fields and invariants of the *150MM* whose PC is true for the selected features, while the rest of the elements are deleted.

A well-formed *150MM* may still produce meta-models which are themselves not well-formed. We thus define an additional notion, that of MMPL well-formedness, as follows:

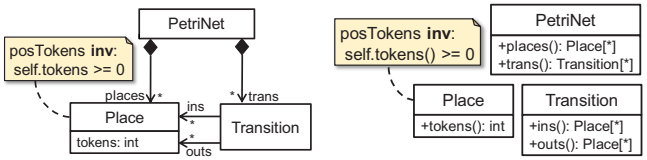
Def. 7 (Well-formed MMPL) A product line *MMPL* is *well-formed* iff every meta-model $MM \in$ *Prod(MMPL)* is well-formed.

MMPLs can be checked for well-formedness without checking every derivable meta-model [17]. Here, we assume well-formed MMPLs, as our focus is on TPLs.

4 TRANSFORMATION PRODUCT LINES

In previous work, we used concepts [11] as the reuse interface of transformations (cf. Fig. 4(a)). A *concept* is a minimal meta-model containing exactly the structural elements required by a transformation. Reuse is possible by binding the concept to specific meta-models (MM_1,..., MM_n in the figure). A *binding* is made of a set of expressions stating how each element in the concept is realized in a specific meta-model.

Concept-based reuse is *open*, as the transformation can be reused with new meta-models if a binding is provided. In this paper, we aim at providing a systematic, cost-effective way to define reusable transformations for a possibly large but closed set of language



(a) Using explicit structure (b) Using operations

Figure 5: Concept for Petri nets.

variants. We capture such a set by an MMPL, and provide a succinct way to specify bindings from the concept to each meta-model in the family via a *binding product line* (PL) (cf. Fig. 4(b)). A tuple made of a concept, a binding PL, and an MMPL is called a *conceptual* MMPL (CMMPL). This is introduced in Sect. 4.1.

We use concepts for two purposes. First, to express expectations on any meta-model variant of an MMPL. For this purpose, we extend concepts to include invariants. Second, to abstract away the dissimilarities between the meta-model variants within an MMPL. This simplifies the encoding of invariants and transformations.

Our reusable transformations are PLs themselves (TPLs). Thus, they need to be aware of the variability of the modelling language (the MMPL), and may introduce variants themselves (e.g., a simulator that fires all non-conflicting enabled transitions concurrently vs. a sequential simulator). This means that the feature model of the TPL may extend that of the MMPL. Such a feature model becomes the reuse interface of the TPL. This simplifies reuse as a transformation can become reused by just selecting a configuration, while in Fig. 4(a), a binding to a meta-model is also required.

To facilitate TPL construction and extension, we split its definition into two parts: a transformation specification, and a PL of transformation implementations. A specification is a template that declares the transformation structure (rules in the case of rule-based languages, or methods in the case of object-oriented model management languages like EOL [23]). It may also specify OCL contracts [31, 36] stating expectations on the possible implementation variants of the template parts (rules, methods). An implementation consists of rule or method bodies guarded by PCs. A TPL so defined can be bound to a CMMPL (cf. Fig. 4(b)) resulting in a *bounded* TPL. TPLs and bounded TPLs are presented in Sect. 4.2.

4.1 Conceptual MMPLs

Concepts capture the essential elements of a set of meta-models, and express expectations on their instances. Just like when designing a meta-model for a domain, a concept should include primitives and invariants applicable to all language variants. Hence, it contains all common elements of an MMPL, plus a primitive for each alternative set of features. This facilitates writing invariants and transformations for *all* meta-models in the MMPL, and ensures common properties among their instances.

Example. Fig. 5(a) shows a concept for the example MMPL. It contains the elements common to all meta-models, and a primitive `tokens` unifying the way in which the different variants express tokens. In addition, the concept adds an invariant stating that no model, in any variant, can have places with negative tokens.

Given a concept, each meta-model in the MMPL needs to declare how it realizes the concept elements. This is called a *binding*. The

binding for the common parts is the identity and can be generated automatically. The binding for the extra elements (e.g., tokens) depends on the variant, and hence, it consists of a set of expressions with PCs stating how the element is realized in each variant.

Although some DSLs have been designed to express bindings [8], we follow a pragmatic approach for their specification: we abstract the fields in the concept as operations (cf. Fig. 5(b)), so that the binding consists of OCL expressions implementing each operation.

Example. Listing 1 shows the binding PL for the running example. Binding the common parts to all meta-models is done through identity expressions (lines 1–6). For the additional element tokens(), we provide two binding variants corresponding to features Simple and Object (lines 8–9). By defining a binding PL, we avoid creating a separate binding for each one of the 32 meta-models of the MMPL.

```
1 context PetriNet
2     def: places() : Place[*] = self.places
3     def: trans() : Transition[*] = self.trans
4 context Transition
5     def: ins() : Place[*] = self.ins
6     def: outs() : Place[*] = self.outs
7 context Place
8     [Simple] def: tokens() : Integer = self.itokens
9     [Object] def: tokens() : Integer = self.ctokens→size()
```

Listing 1: Binding PL for the Petri net MMPL.

Def. 8 (Conceptual MMPL) A *conceptual MMPL*, called CMMPL, is a tuple $CMMPL = \langle MMPL, SC, \beta \rangle$, where:

- $MMPL$ is a meta-model product line;
- $SC = \langle C_{SC}, FI_{SC}, I_{SC}, WC_{SC} \rangle$ is a meta-model, called the *concept*, where $C_{SC} \subseteq C$ (with C the set of classes of the *150MM*);
- $\beta = \{\langle c.f_i, b_i, \Phi_{b_i} \rangle\}_{i \in I}$ is a set of tuples (called *binding PL*) where c is a class in C_{SC}, f_i is a field owned by c, b_i is an expression (called *binding*) which returns a result compatible with f_i, and Φ_{b_i} is the PC of the binding b_i.

We require that each binding from a field f_i returns a value compatible with f_i. That means that if f_i has a primitive type, the binding returns a value of that type, and if f_i is a reference, the binding returns an object of the class pointed to by the reference, or a subclass. Additionally, the cardinality of f_i needs to be respected. We do not deal here with techniques to check the validity of the binding expressions, analysed in detail in [8].

The concept abstracts different realizations of the same element in the MMPL. For example, tokens() unifies the realization of tokens for features Simple and Object. This simplifies the encoding of invariants and transformations.

Example. By using the concept, the two invariants in Figure 2 can be replaced by the one in Listing 2, which uses tokens(). The PC is also simplified as the new invariant is applicable whenever Bounded is selected, regardless of the selection of Simple or Object.

```
1 context Place
2     [Bounded] inv boundedTokens: self.tokens()<=bound
```

Listing 2: Simplified boundedTok invariant.

There are two aspects to check on a CMMPL: if the binding PL provides exactly one binding for each field of the concept (well-formedness), and if all instances of all meta-models of the MMPL satisfy the invariants introduced by the concept (consistency).

Def. 9 (Well-formed, consistent CMMPL) $CMMPL = \langle MMPL, SC = \langle C_{SC}, FI_{SC}, I_{SC}, WC_{SC} \rangle, \beta \rangle$ is *well-formed* if $MMPL$ is well-formed, and each field in SC is bound exactly once in each configuration:

$$\forall \rho \in P \; \forall f \in FI_{SC} \bullet \exists! \langle f, b, \Phi_b \rangle \in \beta \; s.t. \; \Phi_b[\rho] = \text{true}$$

A well-formed CMMPL is *consistent* if every instance M of every $MM \in Prod(MMPL)$ satisfies all invariants in WC_{SC}.

Well-formedness requires each derivable meta-model to have a correct binding to the concept. In Sect. 6.1, we show how to analyse this without generating every meta-model of the MMPL, as this may be computationally expensive. Regarding consistency, concepts may introduce invariants applicable to all meta-models in the MMPL. For example, the invariant in Fig. 5 disallows negative tokens in every meta-model. This invariant fails in meta-models derived from configurations that select feature Simple, as attribute itokens may be negative, but it is satisfied in meta-models with feature Object, as the size of collection ctokens cannot be negative. Sect. 6.2 introduces a technique for checking this property without analysing every meta-model in the MMPL.

4.2 Transformation product lines

Transformation behaviour may vary between language variants. For example, the enablement of transitions differs in the considered Petri net variants (all combinations of Read, Bounded and Inhibitor), and their firing depends on whether they have priorities. Moreover, a transformation may define its own variants (e.g., a concurrent simulator, or a sequential one). To cope with this complexity, we define transformations atop *transformation specifications* (specs) specifying their structure (methods, rules) and contracts for their implementation. Individual variants contribute implementations for the spec, resulting in a TPL.

Def. 10 (Transformation definition) A *transformation definition* is a tuple $TD = \langle MM, R, BO, CN \rangle$, where:

- MM is a meta-model;
- R is a set of rule signatures typed by MM;
- BO is a set of rule bodies typed by MM, each one owned by exactly a rule signature;
- CN is a set of contracts typed by MM, each one owned by exactly a rule signature.

$\langle R, CN \rangle$ is called a *transformation specification*. If each rule signature has (at least) a body, TD is called an *implementation*.

Def. 10 deliberately abstracts away many elements of transformation languages to focus on the essentials required for TPLs. Under this abstraction, R may contain methods, helpers, or rules. The set CN of contracts may be empty; however, contracts for TPLs are useful for two reasons. First, they provide guidelines that developers implementing rules for a variant should follow. Second, they allow using analysis methods for contract proving [7, 16, 27]. These may be unavailable for some transformation languages (e.g., ATL, ETL), but for query helpers where both contract and implementation are in OCL, one can resort to model finding, as Sect. 6.2 shows.

As a correctness condition for transformation definitions, we require each rule signature to have exactly one body, which must satisfy the rule contract.

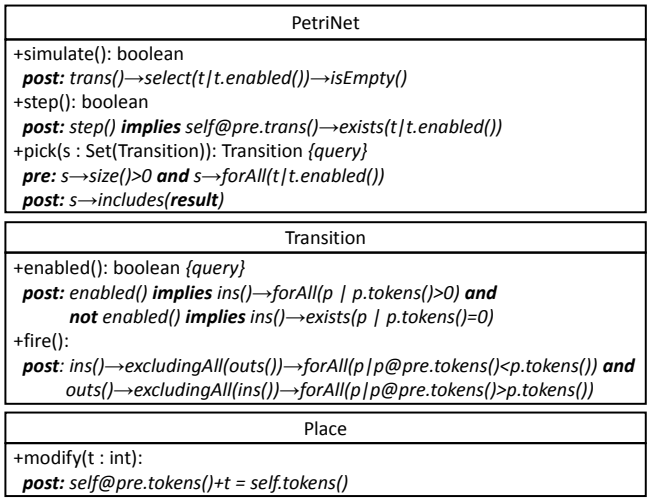


Figure 6: Specification of a simulator for Petri nets.

Def. 11 (Well-formed and consistent transformation definition) A transformation definition is *well-formed* if each rule signature has exactly one body. It is *consistent* if it is well-formed and each body satisfies the contract of the owner rule.

Example. Fig. 6 shows the simulator spec. The simulator is written in EOL. EOL uses an object-oriented approach, where rules (set R) are class methods, and bodies (set BO) are written in an extension of OCL with imperative constructs. Thus, the spec has the form of a set of method signatures defined in the context of concept classes, plus pre/postconditions expressed in OCL (set CN). For rule-based languages, like ATL [19] or ETL [24], a spec is comprised of helper signatures and rules. Some of these languages support OCL contracts [24], but others would require a different treatment, like embedding OCL into special comments.

The simulator is made of six methods, with simulate as the entry point. As its postcondition specifies, no transition should be enabled after the simulation finishes. Method step performs a simulation step, picking an enabled transition and firing it. It returns true if some transition was enabled. Method pick selects the transition to be fired from a set of enabled transitions. Class Place defines the method modify which changes the number of tokens. Class Transition has two methods: enabled and fire. The former is a query checking if all its input places have tokens. Method fire deletes tokens from the input places and adds tokens to the output places. The exact number of tokens is open, as some variants (e.g., weighted arcs) may add or delete several tokens. Note that contracts can use operations defined in the transformation or the concept (like tokens()), which makes them applicable to all meta-models in the MMPL.

While the body of some methods is common to all variants, others are different. Hence, a TPL is a transformation implementation where rule bodies have PCs over a feature model.

Def. 12 (Transformation product line and derivation) A TPL is a tuple $TPL = \langle TD, FM_I, \Phi_{BO} \rangle$ where:

- $TD = \langle MM, R, BO, CN \rangle$ is a transformation implementation;
- $FM_I = \langle F_I, \phi_I \rangle$ is a feature model;
- $\Phi_{BO} = \{\Phi_{bo} | bo \in BO\}$ is a set of PCs of the rule bodies, mapping elements of BO to propositional formulas over features F_I.

Given configuration ρ of FM_I, implementation TD_ρ is *derived* from TPL if TD_ρ contains exactly those bodies whose PC is true for the features in ρ. $Prod(TPL)$ is the set of all derivable implementations.

Example. In the example, methods pick and enabled are queries which we express in OCL for better analysability, while the rest of the methods change the net state and are expressed in EOL. Listing 3 shows the EOL operation modify. It has two versions that correspond to the PCs Simple and Object. Listing 4 shows some variants of the OCL queries pick and enabled. The former has two variants which depend on whether Priority is selected. Method enabled has a variant for every combination of features Inhibitor, Read and Bounded. This is not satisfying for two reasons: (1) building all these variants by hand is error-prone and (2) in the case of pick, defining the variant for Priority (lines 3–4 in Listing 4) requires changing the PC of existing variants (in this case to not Priority, see line 2), which means that adding new variants is not modular. Sect. 5 solves both problems with the addition of merging semantics (to solve the problem with enabled) and overriding policies (to solve the problem with pick).

```
1 [Simple] operation Place modify(t : Integer) { self.itokens += t; }
2 [Object] operation Place modify(t : Integer) {
3     if (t>0) for (i in Sequence{1..t}) self.ctokens.add(new Token);
4     ... }
5 ...
```

Listing 3: TPL: EOL methods (excerpt).

```
1  context PetriNet
2     [not Priority] def: pick(s: Set(Transition)): Transition= s→any(true)
3     [Priority] def: pick(s: Set(Transition)): Transition=
4         s→select(t | not s→exists(t1 | t1.priority>t.priority))→any(true)
5
6  context Transition
7     [not (Inhibitor or Read or Bounded)]
8     def: enabled() : Boolean = self.ins()→forAll(tokens()>0)
9     [Inhibitor and not (Read or Bounded)]
10    def: enabled() : Boolean = self.ins()→forAll(tokens()>0) and
11                               self.inh→forAll(tokens()=0)
12    [Inhibitor and Read and not Bounded]
13    def: enabled() : Boolean = self.ins()→forAll(tokens()>0) and
14                               self.inh→forAll(tokens()=0) and
15                               self.read→forAll(p.tokens()>0)
16 ...
```

Listing 4: TPL: OCL queries (excerpt).

Next, we define well-formedness and consistency of TPLs.

Def. 13 (Well-formed and consistent TPL) TPL is *well-formed* (resp. *consistent*) if $\forall TD \in Prod(TPL)$, TD is *well-formed* (resp. *consistent*).

While a TPL adds variability to a transformation using a feature model, our goal is to define reusable transformations for a family of languages. Hence, we couple the TPL with CMMPLs, requiring that the feature model of the TPL extends that of the CMMPL. We call this construct *bounded* TPL.

Def. 14 (Bounded TPL and derivation) A bounded TPL is a tuple $BTPL = \langle CMMPL, TPL \rangle$ where:

- $CMMPL = \langle MMPL = (FM, MM, \Phi), SC, \beta \rangle$ is a conceptual MMPL;
- $TPL = \langle TD = \langle SC, R, BO, CN \rangle, FM_I, \Phi_{BO} \rangle$ is a TPL typed by the concept SC in $CMMPL$, where $FM \sqsubseteq FM_I$.

Given a configuration ρ of FM_I, an implementation TD_ρ is *derived* from TPL if TD_ρ contains exactly those bodies whose PC is true for the features in ρ, and in addition, operations for exactly those binding expressions whose PC evaluates to true.

Example. Listing 5 shows an excerpt of the implementation derived for configuration ⟨Simple⟩. It contains the version of modify for Simple (line 2, in EOL), the version of pick for ¬Priority (line 5, in OCL) and the version of enabled applicable when none of Inhibitor, Read or Bound are selected (line 7, in OCL). It also contains the version of binding tokens() for Simple (line 9), and the bindings in lines 1–6 of Listing 1 (not shown in Listing 5).

```
1 -- EOL:
2 operation Place modify(t : Integer) { self.itokens += t; }
3 -- OCL:
4 context PetriNet
5   def: pick(s: Set(Transition)): Transition= s→any(true)
6 context Transition
7   def: enabled() : Boolean = self.ins()→forAll(tokens()>0)
8 context Place
9   def: tokens() : Integer = self.itokens -- from binding
```

Listing 5: A derived implementation (excerpt).

A bounded TPL *BTPL* is *well-formed* (resp. *consistent*) if *CMMPL* and *TPL* are well-formed (resp. consistent). *BTPL* is *semi-consistent* if *CMMPL* is consistent and *TPL* is well-formed. We show how to check semi-consistency in Sect. 6.1.

5 COMPOSING RULE BODIES

To avoid the combinatorial explosion of feature combinations, we introduce two composition operators for combining rule bodies in a modular way: *override* (Sect. 5.1) and *merge* (Sect. 5.2).

5.1 Override

We propose a mechanism to define precedence between rule variants whose PC is true in a certain configuration ρ. We took inspiration from overriding mechanisms in object orientation, but in our case, the selected rule body is given by the relationship between their PCs. Our default overriding mechanism is as follows: a body bo_1 for rule r with PC Φ_{bo_1} can override a body bo_2 for r with PC Φ_{bo_2}, if $\Phi_{bo_1} \Rightarrow \Phi_{bo_2}$. This means that bo_1 is more specific than bo_2. Hence, bo_1 is selected in configurations where Φ_{bo_1} is true regardless of whether Φ_{bo_2} is true or not, while bo_2 is selected when Φ_{bo_2} is true and Φ_{bo_1} is false. We call this mechanism "*override super*".

Example. Listing 6 shows an implementation of pick that makes use of the proposed override mechanism. The Priority variant (lines 4–5) overrides the variant with PC true (line 2). This is possible because Priority⇒true. In OCL, we indicate overriding using special annotations in comments (see line 3). Overriding makes the approach more modular because adding a new variant does not require changing the PC of more general variants.

```
1 context PetriNet
2   def: pick(s : Set(Transition)): Transition = s→any(true)
3   -- @Override super
4   [Priority] def: pick(s : Set(Transition)): Transition =
5   s→select(t | not s→exists(t1 | t1.priority> t.priority))→any(true)
```

Listing 6: Variants of pick defined using overriding.

Conversely to "*override super*", "*override subs*" permits giving precedence to more general bodies over more specific ones. Finally, if there are several bodies, none of which is more general or specific than the others, "*override all*" can be used to designate one of them as overriding the rest. In any of the three cases, we write $bo_i \xrightarrow{ovr} bo_j$ to indicate that body bo_i overrides bo_j.

5.2 Merge

If the PC of several rule variants is true for a given configuration, another option is composing their bodies. For this purpose, we provide merge operators which depend on the signature of the rule:

- for boolean queries: operators and, or;
- for queries returning a collection: union, intersection;
- for queries returning an integer: arithmetical operations;
- for operations changing state: sequential composition of actions.

We write $bo_i \overset{mrg\ op}{\longleftrightarrow} bo_j$ to indicate that bodies bo_i and bo_j are to be merged using operator *op* (and omit *op* if not needed). To have deterministic results, we restrict to commutative and associative operators. To be independent of composition order, we require each body to be tagged with the same operator across configurations.

Example. Listing 7 illustrates the use of merge operators to simplify the definition of query enabled. Without them, we need to define a rule variant for all possible combinations of Inhibitor, Read and Bounded (Listing 4 shows three of such combinations). Instead, we can specify a most general variant with PC true (line 2), and then more specific variants for each feature, which are conjoined with the rest (merge and). This way, only four body variants instead of eight are needed to obtain all behaviours. Moreover, the approach becomes modular, as adding a new feature (e.g., reset arcs) amounts to adding a new body and a merge annotation.

```
1 context Transition
2   def: enabled(): Boolean = ins()→forAll(tokens()>0)
3   -- @Merge and
4   [Inhibitor] def: enabled(): Boolean = inh→forAll(tokens()=0)
5   -- @Merge and
6   [Read] def: enabled(): Boolean = read→forAll(tokens()>0)
7   -- @Merge and
8   [Bounded] def: enabled(): Boolean=outs()→forAll(tokens()<bound)
```

Listing 7: Variants of enabled defined using merge.

6 ANALYSING TPLs

This section proposes methods to analyse well-formedness (Sect. 6.1) and consistency (Sect. 6.2). In both cases, we perform the analysis at the TPL level to gain efficiency, avoiding the separate analyses of each meta-model and transformation.

6.1 Analysing well-formedness

First, we deal with CMMPL well-formedness, which demands exactly one binding for each field in the concept, for each possible configuration. Thm. 1 shows how to perform this analysis without enumerating all configurations. We assume well-formedness of the MMPL, an analysis that we lifted to the PL level in [17].

Thm. 1 (CMMPL well-formedness) Given a well-formed meta-model PL $MMPL$, $CMMPL = \langle MMPL, SC, \beta \rangle$ is *well-formed* iff $\forall f \in FI_{SC}$:

(1) There are no binding collisions (no two different binding expressions for the same field are selected by a configuration):

$$\forall \langle f, b_i, \Phi_{b_i} \rangle, \forall \langle f, b_j, \Phi_{b_j} \rangle \in \beta \bullet (b_i \neq b_j) \Rightarrow UNSAT(\Phi_{b_i} \wedge \Phi_{b_j} \wedge \phi_{MMPL}) \quad (1)$$

(2) There is full coverage of all configurations:

$$UNSAT(\phi_{MMPL} \wedge \bigwedge_{\langle f, b_i, \Phi_{b_i} \rangle \in \beta} \neg \Phi_{b_i}) \quad (2)$$

with $UNSAT(\Phi)$ a predicate that holds if Φ is unsatisfiable.

Proof. By Def. 9, to show that *CMMPL* is well-formed we must show there is a unique binding for each field in each configuration:

$$\forall \rho \in P \, \forall f \in FI_{SC} \bullet \exists! \langle f, b, \Phi_b \rangle \in \beta \; s.t. \; \Phi_b[\rho] = \text{true}$$

By Defs. 4 and 8, any valid configuration must satisfy ϕ_{MMPL}. Choose some field $f \in FI_{SC}$. If Equation (2) holds, then there must exist a binding for f in some valid configuration since not all the binding PCs can be false. If Equation (1) holds, then the binding for f must be unique since for any two distinct bindings b_i and b_j, their PCs Φ_{b_i} and Φ_{b_j} cannot be mutually satisfiable for any valid configuration. Thus, Equations (1) and (2) hold for f iff there exists a unique binding in each valid configuration. □

Additionally, we check for well-formedness of TPLs (see Def. 13). This amounts to ensuring that all rules in the TPL have exactly one body in each possible configuration.

Thm. 2 (TPL well-formedness) A transformation PL $TPL = \langle TD, FM_I = \langle F_I, \phi_I \rangle, \Phi_{BO} \rangle$ is *well-formed* iff:

(1) There are no body collisions:

$$\begin{aligned} \forall \Phi_{bo_i}, \Phi_{bo_j} \in \Phi_{BO} \bullet (bo_i \neq bo_j) \Rightarrow \\ (UNSAT(\Phi_{bo_i} \wedge \Phi_{bo_j} \wedge \phi_I) \vee \\ bo_i \xrightarrow{ovr} bo_j \vee bo_j \xrightarrow{ovr} bo_i \vee bo_i \xleftrightarrow{mrg} bo_j) \end{aligned} \tag{3}$$

(2) There is full coverage of all configurations:

$$UNSAT(\phi_I \wedge \bigwedge_{\Phi_{bo_i} \in \Phi_{BO}} \neg \Phi_{bo_i}) \tag{4}$$

The proof of Thm. 2 follows the same pattern as the one of Thm. 1. Equation (3) requires either a unique body, or, if two bodies overlap, one overrides the other or both are merged. Equation (4) requires coverability of all configurations.

6.2 Analysing consistency

If contracts are specified in OCL, it is possible to verify them over each product meta-model with existing techniques [7, 16, 27]. However, the number of meta-models in a TPL may grow exponentially, and so we do not analyse consistency and contracts over each meta-model, but we lift the analyses to the TPL level. This way, we build an extended version of the *150MM* which embeds all meta-model variants, the OCL queries and contracts of all transformation variants, and the configurations of the MMPL. Then, we use a model finder to search for an instance of this extended meta-model that exemplifies (or falsifies, if no model is found) an inconsistency or a contract violation. Our analysis consists of the following steps:

(1) Embedding feature model and PCs. First, we build an extended version of the *150MM* that contains the meta-model elements introduced by every variant, invariants emulating the semantics of the PCs, and an additional class FMC with a boolean field for each feature in the feature model and an invariant controlling the allowed configurations. We call it *feature-explicit meta-model* (FEMM) [17].

Example. Fig. 7 shows an excerpt of the FEMM derived from the feature model and the *150MM* in Figs. 1 and 2. In the FEMM, all meta-model classes inherit from an abstract class BC which holds a reference to class FMC. This way, every class has access to the field

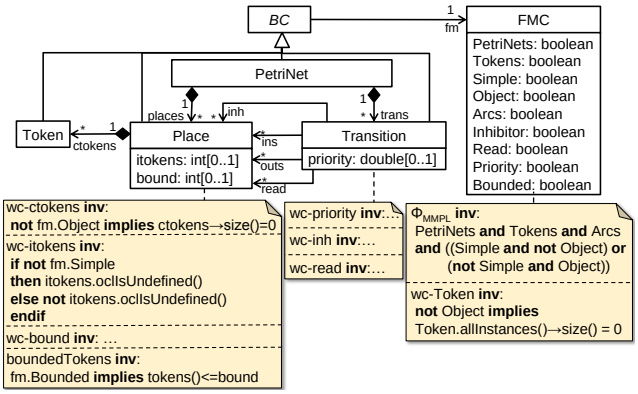


Figure 7: Feature explicit meta-model (excerpt).

values in FMC, i.e., to the feature configuration. PCs over classes are converted into invariants of FMC that disallow any instance of the class when its PC is false (see wc-Token). Similarly, PCs over fields are converted into invariants of the owner class requiring that the field is empty when its PC is false (see invariants wc-tokens and wc-itokens). In addition, the lower bound of fields having a PC is set to 0 (e.g., field itokens). Finally, invariants with a PC are rewritten to be enforced only when their PC holds (see boundedTokens, which assumes the simplification in Listing 2 was applied).

(2) Embedding implementations. Next, we extend the FEMM with the OCL methods of the binding and the TPL. The FEMM needs to emulate the PCs, the merging and the overriding semantics. Hence, if a method has overriding implementations, its body is built as a nested *if-then-else* that selects the appropriate implementation (*then*) when its PC is satisfied (*if*). This way, the first PC (*if*) that evaluates to true determines which implementation to apply, and the order in which the conditionals are placed is given according to the overriding variant (*super*, *subs*, *all*). To emulate merging, if n implementations are merged together, we create 2^n *if* statements that consider all combinations of the satisfaction of their PCs and compose the bodies corresponding to these PCs.

Example. Listing 8 shows the encoding of method tokens and part of method enabled. The former is defined in the binding PL of Listing 1, and has two different implementations for the disjoined PCs Simple and Object. The latter method belongs to the TPL, and has one default implementation and three variants (shown in Listing 7) that should be merged using the and operator. Hence, the body of the method considers eight possible implementations, though the listing only shows three of them due to space constraints.

(3) Analysing consistency of CMMPL. Analysing CMMPL consistency (i.e., whether the instances of all derivable meta-models satisfy the invariants in the concept, see Def. 9) is done using the FEMM, instead of checking each meta-model separately. Thus, for each invariant inv in the concept, we try to find an instance of the FEMM that violates it. For this purpose, in the FEMM, we encode the invariant as a boolean method of the invariant's owner class, and add the following invariant to class FMC:

<owner-class(inv)>.**allInstances**() → **exists**(**not** inv())

Then, we use a model finder to search for an instance of the FEMM having one FMC object. If an instance is found, it embeds both

```
1  context Place
2    def: tokens() : Integer =
3      if fm.Simple then self.itokens
4      else if fm.Object then self.ctokens→size()
5      else null endif
6      endif
7  context Transition
8    def: enabled() : Boolean =
9      if (not fm.Read and not fm.Bounded and not fm.Inhibitor)
10     then ins()→forAll(tokens()>0)
11     else
12       if (fm.Bounded and not fm.Inhibitor and not fm.Read)
13       then ins()→forAll(tokens()>0) and
14            outs()→forAll(tokens()<bound)
15       else
16         if (fm.Bounded and fm.Inhibitor and not fm.Read)
17         then ins()→forAll(tokens()>0) and
18              outs()→forAll(tokens()<bound) and
19              inh→forAll(tokens()=0)
20         else
21           ...
22         endif
23       endif
24     endif
```

Listing 8: Embedding of method implementations.

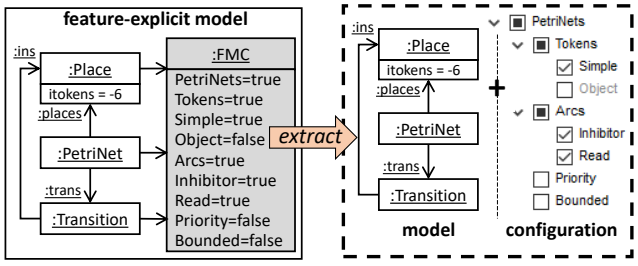


Figure 8: Configuration and model exemplifying a violation of invariant posTokens.

a model showing an invariant violation, and an object of type FMC holding the feature configuration producing the model's meta-model. A CMMPL is consistent (up to the search bound) if no instance violating the concept's invariants is found.

Example. To analyse invariant posTokens in the concept of Fig. 5(b), our technique adds method posTokens(): **Boolean** = tokens()>=0 to class Place in the FEMM, and invariant Place.allInstances()->exists(**not** posTokens()) to class FMC. The model finder finds the FEMM instance shown in the left side of Fig. 8, from which the model and configuration to the right can be extracted. The model does not satisfy invariant posTokens because attribute itokens is negative. In fact, any configuration selecting Simple produces meta-models with instances violating the invariant. This can be solved by adding an invariant tokens>=0 with PC Simple to class Place in the *150MM*.

(4) Analysing contracts. Analysing contracts is similar to analysing consistency, but it is the contract associated with each method that becomes converted into an operation and subsequently analysed. Our method is currently limited to the analysis of preconditions and invariants that do not use parameters of the owner method, and to postconditions of queries that do not contain @pre.

Example. Our analysis reports that configuration ⟨Simple, Read⟩ does not satisfy the postcondition of enabled (shown in Fig. 6), and returns a model that exemplifies it. This model violates the second implication of the postcondition as it contains a transition that is disabled because it has an empty read place, but it has no input place in relation ins with 0 tokens. This time the contract is overly restrictive as it requires a transition is disabled when some input place has zero tokens, but this does not hold for read arcs.

7 TOOL SUPPORT

We have implemented the presented ideas in a tool called Merlin, freely available at http://miso.es/tools/merlin-tpl.

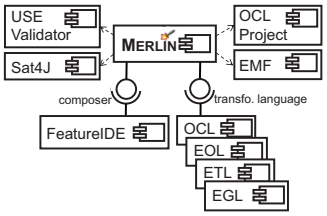


Figure 9: Tool architecture.

Merlin is an Eclipse plugin that extends FeatureIDE (a framework to build SPLs) [30] to allow the creation, analysis and usage of MMPLs and TPLs. Fig. 9 shows its architecture. The tool uses EMF to handle MMPLs, the Eclipse OCL project to parse OCL invariants and contracts, and Sat4J [5] and the USE Validator [25] for the analyses. Sat4J is a SAT solver used for the satisfiability checks over propositional formulas needed by the well-formedness analyses of Sect. 6.1. The USE Validator is a model finder (a constraint solver over models) that we use for the consistency analyses of Sect. 6.2.

Merlin is agnostic w.r.t. the transformation language used, as it offers an extension point that is to be implemented for integrating each new language. Merlin provides support for OCL, and integrates some of the languages of the Epsilon family [37] to define PLs of in-place transformations (EOL), model-to-model transformations (ETL) and code generators (EGL). Integrating a new language requires handling the merge and override annotations and implementing their semantics. Moreover, for languages that do not have direct access to OCL operations written as .ecore or .ocl files (e.g., ATL [19], EOL [23]), we require an exporter from OCL to the transformation language (EOL operations, ATL helpers).

Fig. 10 shows a screenshot of Merlin for the running example. Label 1 shows the feature model, and label 2 a fragment of the transformation written in OCL. For OCL, composition annotations are embedded as comments, and PCs are expressed as external configuration files (label 3 corresponds to the PC Read). Merlin handles PCs transparently to the transformation language. The 150*MM* is a standard ecore meta-model with PCs given as annotations.

The package explorer to the left of the figure displays the structure of a Merlin project. It contains a folder (label 4) used to store the bindings (.ocl files), concepts (.ecore files) and transformations (files using the chosen transformation language). From them, Merlin can generate all possible meta-model and transformation variants (folder products with label 5) or the variant for a specific configuration. Merlin supports the analyses described in Sect. 6, triggered by a wizard (label 6). The results are displayed in a dialog box (label 7) and in the problem's view.

8 VALIDATION

Next, we assess the benefits of TPLs compared with building a family of separate meta-models and transformations. Our evaluation is guided by two research questions: *RQ1: how much specification effort*

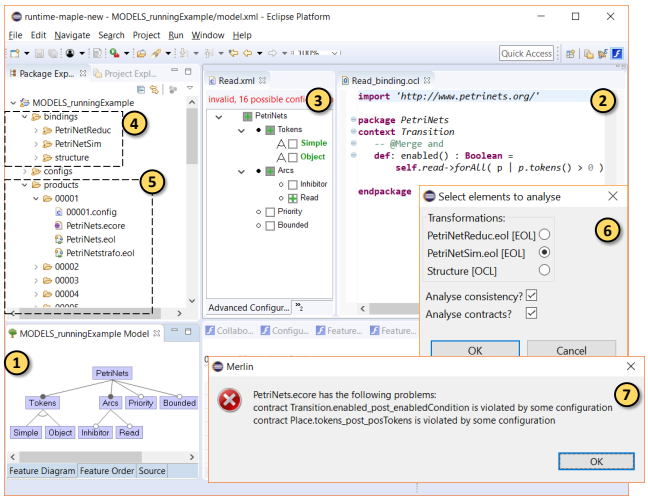


Figure 10: MERLIN in action.

Name	*CMMPL* size Cs/PCs/LOC	*TPL* size CTs/PCs/LOC	#mm/ #diff. tran.	size in LOC (avg/total)	avg diffs in LOC
Running example	4/8/14	8/6/91	32/32	61/1952	7.9
PN reduction [49]	4/8/14	5/3/110	32/25	118.5/3792	19.1
Graph alg-num [28]	5/11/67	0/5/93	192/189	90.3/17344	10.5
Graph alg-ccs [28]	5/11/67	0/3/89	192/177	86.7/16640	3.6

Table 1: Metrics for TPLs and their products.

do TPLs save, addressed in Sect. 8.1, and *RQ2: how much analysis effort do TPLs save*, answered in Sect. 8.2.

8.1 Specification effort

To answer *RQ1*, we have written four TPLs and compared their specification size w.r.t. the size of all variants of the transformations. We have used two TPLs over Petri Nets (the running example, and a reduction transformation which simplifies a net by merging sequential places and transitions [49]), and two over a PL of graphs typically used as a benchmark for SPL approaches [28]. The TPLs for graphs permit configuring the type of graph (directed, undirected, weighted, etc.) and the search method (depth-first, breadth-first). The first graph TPL assigns a unique number to each node, and the second calculates the connected components. All TPLs are available at http://miso.es/tools/merlin-tpl.

Table 1 summarizes the results. The first two columns show the CMMPL size (number of classes and PCs, lines of OCL code of the binding) and the TPL size (number of contracts and PCs, LOC). The next three columns give an intuition of the size and diversity of the transformation variants. Column 3 shows the number of product meta-models and different transformation variants. Column 4 shows the average size of each transformation variant (in LOC), and the total size of all transformations. The last column measures the difference between transformation variants. We use Myers' algorithm [34] – used by version control systems like Git – to calculate the average different LOC between any two transformation variants (including lines changed, inserted or deleted).

Two conclusions can be obtained. First, most transformations generated from each TPL are different. In the running example, every generated transformation is different, with about an 8 LOC difference between them. Second, the size of the TPL specification is much smaller than the total size of the generated transformations. For instance, in the running example, the TPL size is 91 LOC (with 8 contracts and 6 PCs), while without a TPL, we need to write 32 different transformations of about 61 LOC each, amounting to more than 1900 LOC altogether. This gain of at least one order of magnitude in size is also apparent in the other cases. Hence, these results suggest considerable effort gains by using TPLs w.r.t. encoding each transformation in isolation. The exponential increase in the number of variants that new features may introduce suggests even bigger gains for larger feature models.

Threats to validity. We have used LOC and the number of contracts and PCs as an indication of TPL size. An additional experiment with developers may help to assess the correlation between such size and the actual effort to build the transformations. We have used only 4 TPLs over 2 different CMMPLs. More extensive experiments would be needed for extra confidence in our claims.

8.2 Analysis effort

To answer RQ2, we compare the performance of the lifted consistency analysis described in Sect. 6.2, w.r.t. analysing every meta-model of a CMMPL one by one. The latter requires generating every meta-model and checking whether its instances satisfy the invariants in the CMMPL, which is done through model finding. The analysis finishes when a meta-model with problematic instances is found, or when all meta-models have been successfully analysed.

The experiment was conducted on a Windows 10 computer, with i7-6500U processor and 16Gb of RAM. We used an extension of the running example with three additional features (weighted arcs, reset arcs and hierarchical nets) which resulted in 256 meta-models. Moreover, we added the invariant tokens()>0 (a variation of invariant posTokens in the running example) to the concept. While all meta-models in this CMMPL have instances that violate the invariant (e.g., Petri nets with empty places), we manually built 20 versions of the CMMPL where the number of meta-models violating the invariant was different (i.e., one version had one meta-model that violated the invariant, another had 2, and so on). This was done to emulate CMMPLs with different ratios of incorrect meta-models. Then we used both approaches (lifted and enumerative) to analyse CMMPL consistency of each version, 40 times each, computing the average time. In the enumerative approach, the traversal of the meta-models was randomized in each analysis.

Fig. 11 shows the average analysis time of each approach in milliseconds (vertical axis) w.r.t. the ratio of meta-models that violate the invariant (horizontal axis). It also shows the best and worst times for the enumerative approach, but not for the lifted one because it is similar to the average.

On average, the lifted analysis is faster than the enumerative one when less than 42% meta-models violate the invariant, becoming 10x-120x faster if less than 3% meta-models violate it. When no meta-model violates the invariant, the lifted analysis takes less than 1s, and the enumerative one takes 20s. If more than 42% meta-models are incorrect, the lifted approach is only slightly slower but still reasonable (of the order of 200ms vs 100ms). In any case, one may expect invariant violations to be sparse in the meta-model distribution (where our lifted approach performs much better), or

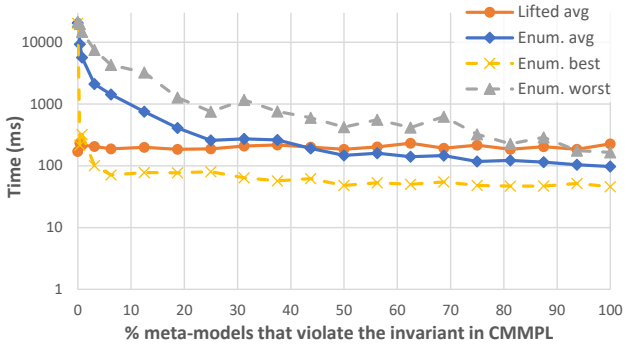


Figure 11: Lifted and enumerative consistency analyses.

otherwise be trivial to detect. The best case of the enumerative approach corresponds to detecting an invariant violation for the first analysed meta-model, in which case it is up to 2x faster than the lifted analysis because the constraint solving problem is easier. Overall, the lifted analysis is only slightly slower than the best case of the enumerative one (and always below 230 ms), but it is several orders of magnitude faster than the corresponding worst case.

Threats to validity. This experiment shows very promising results, but their generality should be strengthened by expanding the evaluation to further CMMPLs, like those in Table 1 and [17]. Hence, although our study considers 20 CMMPLs, they are variations of the same example. While we have not evaluated the contract checking performance, the results should be similar to the consistency ones, as both analyses rely on the same techniques.

9 RELATED WORK

Featured Model Types (FMT) [38] is the closest idea to TPLs. An FMT contains a PL of related meta-models and transformations that apply to different meta-model variants. A configuration identifies both the meta-model and the applicable transformations. Yet so far FMTs are a preliminary idea, without details or an available implementation, making them difficult to assess. Our approach separates the MMPL from the transformations, and each TPL focuses on variants of a single transformation. This improves modularity, and adding concepts to an MMPL makes the approach practical. In addition, we provide concrete details on all aspects of the transformation adaptation process, concrete analyses and a supporting tool.

A few approaches propose adding variability to transformation languages, like QVT-R [21] and ATL [44]. The former assigns transformation fragments (containing either rules or statements) to features, while the latter can define variants for a base rule. None of them can handle variability of the meta-model. Our TPLs are language-independent and the variability granularity is the rule body, which can have arbitrary PCs (instead of single features). Neither [21] nor [44] provide an explicit notion of specification or guarantee correctness. Both approaches [21, 44] focus on model-to-model transformations. We have presented PLs of in-place transformations, but our approach is applicable to model-to-model transformations as well (Merlin supports ETL). However, exploring variability of *both* source and target meta-models is future work.

Variability rules [47] were proposed to express similar graph transformation rules, in a compact way. Hence, they are a structuring mechanism for transformations. To achieve modularity, other approaches inject variability in transformation rules through aspect orientation [29]. Instead, TPLs express variants of a transformation for a language family.

Other works focus on transforming model PLs (i.e., models annotated with PCs) [46, 48]. This is different from our proposal, as we define TPLs whose products (transformations) are applicable on "regular" models, and not on product lines.

Other transformation reuse approaches have been proposed [6], like model subtyping [18, 45], concepts [8, 11], a-posteriori typing [12], multi-level modelling [13] or transformation patterns [14]. They allow reusing a transformation for a different meta-model it was defined for, by providing some kind of mapping (a subtyping relation, a binding, a retyping specification, an instantiation relation, a mapping) between the original and the target meta-models. In all these cases, a reusable transformation has *one behaviour*, which adapts to the structure of the target meta-model. Instead, TPLs also permit *specializing* the behaviour of the transformation to the specific language variant. Those other approaches focus on syntactic aspects of reuse. Thus, there is no guarantee that the reused transformation behaves in the intended way for the target meta-model [42]. As TPLs address reuse only within an explicit family of meta-models, they enable a better semantic alignment of the reused transformation due to the transformation specification.

Our approach also builds on ideas of Feature Oriented Programming (FOP) [1]. Our *override* and *merge* operators can be seen as a special case of *superimposition* [2]. Composition approaches have proposed ways to access one method body from another, e.g., using special calls like `original()` [4], keywords like `super` [3], or defining *lifters* [40]. In comparison, our operators are non-intrusive, independent on the merge order, and permit analysis (cf. Sect. 6.2).

In summary, our proposal is novel by considering variability of both the language (MMPL) and the transformation (TPL). The use of concepts and composition operators makes the approach practical, and we have also proposed novel analyses for TPLs.

10 CONCLUSIONS AND FUTURE WORK

We have presented a new notion, *transformation product lines* (TPLs), which admits both transformation variability and meta-model variability. The approach facilitates systematic creation of transformations for language families. We have proposed analysis mechanisms at the PL level and presented a tool (Merlin) and an evaluation showing its benefits in terms of size and analysis performance w.r.t. explicitly building and analysing each transformation separately.

Our notion of TPLs opens the door to applying SPL technology to a whole range of MDE artefacts, like model-to-model transformations or code generators. We are investigating methods for cost-effective testing of TPLs, and mechanisms to factor out common code between TPLs defined over the same concept.

ACKNOWLEDGMENT

Work funded by NSERC, the Spanish MINECO (TIN2014-52129-R) and the R&D programme of Madrid (S2013/ICE-3006).

REFERENCES

[1] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8, 5 (2009), 49–84.

[2] Sven Apel and Christian Lengauer. 2008. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. ICSC'08 (LNCS)*, Vol. 4954. Springer, 20–35.

[3] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.* 30, 6 (2004), 355–371.

[4] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. 2005. Classbox/J: Controlling the Scope of Change in Java. In *Proc. of OOPSLA'05*. ACM, 177–189.

[5] Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *JSAT* 7, 2-3 (2010), 59–6.

[6] Jean-Michel Bruel, Benoit Combemale, Esther Guerra, Jean-Marc Jézéquel, Jörg Kienzle, Juan de Lara, Gunter Mussbacher, Eugene Syriani, and Hans Vangheluwe. 2018. Model Transformation Reuse across Metamodels: A Classification and Comparison of Approaches. In *Proc. of ICMT'18 (LNCS)*, Vol. 10888. Springer, 92–109.

[7] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Proc. of FMCO'05 (LNCS)*, Vol. 4111. Springer, 342–363.

[8] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2014. A Component Model for Model Transformations. *IEEE Trans. Software Eng.* 40, 11 (2014), 1042–1060.

[9] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2017. Static Analysis of Model Transformations. *IEEE Trans. Software Eng.* 43, 9 (2017), 868–897.

[10] Juan de Lara, Juri di Rocco, Davide di Ruscio, Esther Guerra, Ludovico Iovino, Alfonso Pierantonio, and Jesús Sánchez Cuadrado. 2017. Reusing Model Transformations Through Typing Requirements Models. In *Proc. of FASE'17 (LNCS)*, Vol. 10202. Springer, 264–282.

[11] Juan de Lara and Esther Guerra. 2013. From Types to Type Requirements: Genericity for Model-Driven Engineering. *J. Software and System Modeling* 12, 3 (2013), 453–474.

[12] Juan de Lara and Esther Guerra. 2017. A Posteriori Typing for Model-Driven Engineering: Concepts, Analysis, and Applications. *ACM TOSEM* 25, 4 (2017), 31:1–31:60.

[13] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2015. Model-Driven Engineering with Domain-Specific Meta-Modelling Languages. *J. Software and System Modeling* 14, 1 (2015), 429–459.

[14] Hüseyin Ergin, Eugene Syriani, and Jeff Gray. 2016. Design Pattern Oriented Development of Model Transformations. *Computer Languages, Systems & Structures* 46 (2016), 106–139.

[15] Rik Eshuis. 2009. Reconciling Statechart Semantics. *Sci. Comput. Program.* 74, 3 (2009), 65–99.

[16] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 – Where Programs Meet Provers. In *Proc. of ESOP'13 (LNCS)*, Vol. 7792. Springer, 125–128.

[17] Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. 2018. Analysing Meta-Model Product Lines. Submitted. Available at http://miso.es/pubs/merlin.pdf.

[18] Clément Guy, Benoît Combemale, Steven Derrien, Jim R.H. Steel, and Jean-Marc Jézéquel. 2012. On Model Subtyping. In *Proc. of ECMFA'12*. 400–415.

[19] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A Model Transformation Tool. *Sci. Comput. Program.* 72, 1-2 (2008), 31–39.

[20] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[21] Lucia Kapová and Thomas Goldschmidt. 2009. Automated Feature Model-Based Generation of Refinement Transformations. In *Proc. of 35th Euromicro SEAA*. IEEE Computer Society, 141–148.

[22] Heiko Kern, Axel Hummel, and Stefan Kühne. 2011. Towards a Comparative Analysis of Meta-metamodels. In *Proc. of SPLASH '11 Workshops*. ACM, New York, NY, USA, 7–12.

[23] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006. The Epsilon Object Language (EOL). In *Proc. of ECMDA-FA'06 (LNCS)*, Vol. 4066. Springer, 128–142.

[24] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2008. The Epsilon Transformation Language. In *Proc. of ICMT'08 (LNCS)*, Vol. 5063. Springer, 46–60.

[25] Mirco Kuhlmann and Martin Gogolla. 2012. From UML and OCL to Relational Logic and Back. In *Proc. of MODELS'12 (LNCS)*, Vol. 7590. Springer, Berlin, Heidelberg, 415–431.

[26] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. 2015. Reuse in Model-to-Model Transformation Languages: Are We There Yet? *J. Software and Systems Modeling* 14, 2 (2015), 537–572.

[27] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR-16 (LNCS)*, Vol. 6355. Springer, 348–370.

[28] Roberto E. Lopez-Herrejon and Don Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. of GCBSE'01*, Jan Bosch (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 10–24.

[29] Rodrigo Machado, Luciana Foss, and Leila Ribeiro. 2009. Aspects for Graph Grammars. *ECEASST* 18 (2009).

[30] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[31] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (1992), 40–51.

[32] MOF. 2016. http://www.omg.org/spec/MOF.

[33] T. Murata. 1989. Petri Nets: Properties, Analysis and Applications. *Proc. IEEE* 77, 4 (1989), 541–580.

[34] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 2 (1986), 251–266.

[35] L. Northrop and P. Clements. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[36] OCL. 2014. http://www.omg.org/spec/OCL/.

[37] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, and Fiona A. C. Polack. 2009. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proc. of ICCECCS'09*. IEEE Computer Society, Washington, DC, USA, 162–171.

[38] Gilles Perrouin, Moussa Amrani, Mathieu Acher, Benoît Combemale, Axel Legay, and Pierre-Yves Schobbens. 2016. Featured Model Types: Towards Systematic Reuse in Modelling Language Engineering. In *Proc. of MiSE@ICSE'16*. ACM, New York, NY, USA, 1–7.

[39] Ana Pescador, Antonio Garmendia, Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. 2015. Pattern-Based Development of Domain-Specific Modelling Languages. In *Proc. of MoDELS'15*. IEEE Computer Society, 166–175.

[40] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. of ECOOP'97 (LNCS)*, Vol. 1241. Springer, 419–443.

[41] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. 2014. Lifting Model Transformations to Product Lines. In *Proc. of ICSE'14*. ACM, New York, NY, USA, 117–128.

[42] Rick Salay, Steffen Zschaler, and Marsha Chechik. 2016. Correct Reuse of Transformations is Hard to Guarantee. In *Proc. of ICMT'16 (LNCS)*, Vol. 9765. Springer, 107–122.

[43] D. C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. *Computer* 39, 2 (Feb. 2006), 25–31.

[44] Marten Sijtema. 2010. Introducing Variability Rules in ATL for Managing Variability in MDE-Based Product Lines. *Proc. of MtATL'10* (2010), 39–49.

[45] Jim Steel and Jean-Marc Jézéquel. 2007. On Model Typing. *J. Software and Systems Modeling* 6, 4 (2007), 401–414.

[46] Daniel Strüber, Sven Peldzsus, and Jan Jürjens. 2018. Taming Multi-Variability of Software Product Line Transformations. In *Proc. of FASE'18*. Springer, 337–355.

[47] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. 2018. Variability-Based Model Transformation: Formal Foundation and Application. *Formal Asp. Comput.* 30, 1 (2018), 133–162.

[48] Gabriele Taentzer, Rick Salay, Daniel Strüber, and Marsha Chechik. 2017. Transformations of Software Product Lines: A Generalizing Framework based on Category Theory. In *Proc. of MODELS'17*. IEEE, 101–111.

[49] H. M. W. Verbeek, Moe Thandar Wynn, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. 2010. Reduction Rules for Reset/Inhibitor Nets. *J. Comput. Syst. Sci.* 76, 2 (2010), 125–143.

[50] Steffen Zschaler. 2014. Towards Constraint-Based Model Types: A Generalised Formal Foundation for Model Genericity. In *Proc. VAO@STAF'14*. ACM, 11–18.