Migrating Automotive Product Lines: a Case Study

Michalis Famelis[1], Levi Lúcio[2], Gehan Selim[3], Alessio Di Sandro[1], Rick Salay[1], Marsha Chechik[1], James R Cordy[3], Juergen Dingel[3], Hans Vangheluwe[2], and Ramesh S[4]

[1] University of Toronto
[2] McGill University
[3] Queens University
[4] General Motors

Abstract. Software Product Lines (SPL) are widely used to manage variability in the automotive industry. In a rapidly changing industrial environment, model transformations are necessary to aid in automating the evolution of SPLs. However, existing transformation technologies are not well-suited to handling industrial-grade variability in software artifacts. We present a case study where we "lift" a previously developed migration transformation so that it becomes applicable to realistic industrial product lines. Our experience indicates that it is both feasible and scalable to lift transformations for industrial SPLs.

1 Introduction

The sprawling complexity of software systems has lead many organizations to adopt *software product line techniques* to manage large portfolios of similar products. For example, modern cars use software to achieve a large variety of functionality, from power train control to infotainment. To organize and manage the huge variety of software subsystems, many car manufacturers, such as General Motors (GM), make extensive use of software product line engineering techniques [14].

At the same time, *model-based techniques* are also actively used by companies, especially in domains such as automotive and aerospace, as a way to increase the level of abstraction and allow engineers to develop systems in notations they feel comfortable working with [25]. That also entails the active use of *model transformations* – operations for manipulating models in order to produce other models or generate code.

Currently, GM is going through the process of migrating models from a legacy metamodel to AUTOSAR [2]. In previous work, we have presented the transformation GmToAutosar [31]. Given a single GM legacy model, GmToAutosar produces a single AUTOSAR output model, based on a set of requirements followed by GM engineers. In order to study its correctness, GmToAutosar was implemented in DSLTrans [30,21], a model transformation language that specializes in helping developers create provably correct transformations.

Because of the extensive use of product lines, GM is now faced with the problem of migrating an entire product line of legacy models to a new product line of AUTOSAR models. To do this, GM engineers need to create purpose-specific migration transformations. Yet transforming product lines is inherently difficult: the relationships between the products need to be preserved, and a variety of properties between the input and output models in the transformation need to be established. Thus, the task of a product-line level transformation is not only to maintain relationships between the features and relationships between the products but also to make sure that the transformation maintains certain properties, expressed in terms of pre- and post- conditions. Existing tools and methodologies do not facilitate model transformations in the context of product lines.

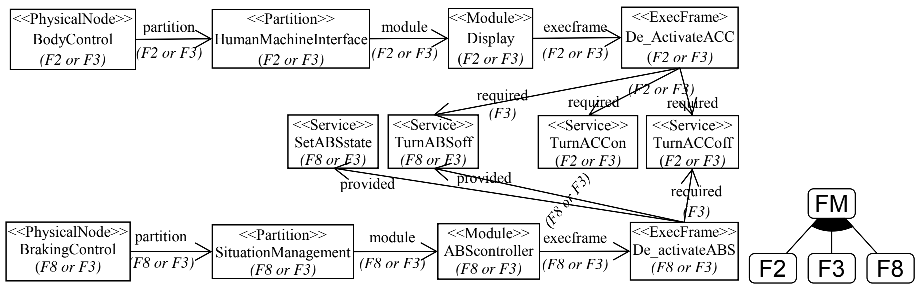
In our earlier work [27], we presented a technique for *"lifting"* a class of model transformations so that they can be applied to software product lines. *Lifting* here means *reinterpretation* of a transformation so that instead of a single product, it applies to the entire product line.

The goal of this paper is to demonstrate, using an empirical case study from an automotive domain, that it is tractable to lift industrial-grade transformations. Specifically, we report on an experience of lifting a previously published transformation [31], GmToAutosar, used in the context of automotive software and applying it to a realistic product line. We lifted GmToAutosar using the theory of lifting presented in [27]. In order to do this, we had to adapt parts of the existing model transformation engine, DSLTrans. The resulting lifted version of GmToAutosar is capable of transforming product lines of legacy GM models to product lines of AUTOSAR models, while preserving the correctness of individual product transformations. We also stress-tested the lifted GmToAutosar to investigate the effect of the size of the model and the variability complexity on the lifted transformation. Due to limitations to publication of sensitive industrial data, the product line we analyzed was created using publicly available data and calibrated with input from GM engineering.

The rest of the paper is organized as follows: we introduce background on to software product lines in Sec. 2. The GmToAutosar transformation is described in Sec. 3 and its lifting – in Sec. 4. We discuss the experience of applying the lifted transformation in Sec. 5. In Sec. 6 we present lessons learned and Sec. 7 discusses related work. We conclude in Sec. 8 with a summary of the paper and discussion of future work.

2 Product Lines in the Automotive Industry

Product Lines in GM. Modern cars at GM can contain tens of millions of lines of code, encompassing powertrain control, active and passive safety features, climate control, comfort and convenience systems, security systems, entertainment systems, and middleware to interconnect all of the above. In addition to software complexity, the variability is high – over 60 models with further variation to account for requirements differences in 150+ countries. The number of prod-



<<PhysicalNode>> BodyControl (F2 or F3)
partition (F2 or F3)
<<Partition>> HumanMachineInterface (F2 or F3)
module (F2 or F3)
<<Module>> Display (F2 or F3)
execframe (F2 or F3)
<<ExecFrame> De_ActivateACC (F2 or F3)
required (F3)
required
(F2 or F3)
required
<<Service>> SetABSstate (F8 or F3)
<<Service>> TurnABSoff (F8 or F3)
<<Service>> TurnACCon (F2 or F3)
<<Service>> TurnACCoff (F2 or F3)
provided
provided
(F8 or F3)
required (F3)
<<PhysicalNode>> BrakingControl (F8 or F3)
partition (F8 or F3)
<<Partition>> SituationManagement (F8 or F3)
module (F8 or F3)
<<Module>> ABScontroller (F8 or F3)
execframe (F8 or F3)
<<ExecFrame>> De_activateABS (F8 or F3)
FM
F2
F3
F8

Fig. 1. A fragment of the exemplar automotive product line model. The left side shows the domain model annotated with presence conditions and the right side shows the feature model.

uct variants produced is in the low tens of thousands. GM is re-engineering its variability tooling to use the commercial product line tool Gears by BigLever Software[1] [14]. To help manage the complexity, product lines will be decomposed into modules corresponding to the natural divisions in the automotive system architecture to produce a hierarchical product line. For example, the subsystems dealing with entertainment, climate control, etc. will have their own product lines, and these will be merged into parent product lines to represent the variability for an entire vehicle.

Case Study Product Line. We applied a transformation on a realistic product line exemplar (as opposed to the actual product line used in GM) due to reasons of confidentiality. We started with publicly available models [1] and built an exemplar model conforming to the GM metamodel in Fig. 2 and consisting of six features and 201 elements. With the help of our industrial partners, we validated that our exemplar is realistic in terms of its structure and size. Since our goal is to do transformation lifting, the product line we produced is *annotative* [10,18,26]. We formally review the definition of the annotative product line approach below.

Definition 1 (Product Line) *A product line P consists of the following parts: (1) A* feature model *that consists of a set of features and the constraints between them; (2) a* domain model *consisting of a set of model elements; and, (3) a mapping from the feature model to the domain model that assigns to each element of the domain model a propositional formula called its* presence condition *expressed in terms of features. We call any selection of features that satisfy the constraints in the feature model to be a* configuration *and the corresponding set of domain elements with presence conditions that evaluate to $True$ given these features is called a* product*. We denote the set of all configurations of P by $\mathsf{Conf}(P)$.*

Note that the Gears product lines used at GM are annotative but use a slightly different terminology than in Def. 1. Fig. 1 shows a fragment of the exemplar product line to illustrate the components of an annotative product line. It

[1] www.biglever.com

shows three of the six features: feature *F2* representing Adaptive Cruise Control (ACC), *F8* representing Anti-lock Braking System (ABS), and *F3* representing Smart Control (SC), an integrated system for assisted driving. The relevant fragment of the feature model is shown on the right of the figure and the solid bar connecting the three features expresses the constraint that the features are mutually exclusive.

The domain model is a class diagram showing the architectural elements. The *BodyControl PhysicalNode* runs *Partitions* such as the *HumanMachineInterface*. The *HumanMachineInterface Partition* contains the *Display Module* which runs multiple *ExecFrames* at the same or different rates. The *De_ActivateACC* ExecFrame allows controlling the ACC feature by invoking Services for variable updates (e.g., *TurnACCon* and *TurnACCoff Services*). The *BrakingControl PhysicalNode* runs the *SituationManagement Partition*. The *SituationManagement Partition* contains the *ABScontroller Module* which runs the *De_activateABS ExecFrame*. The *De_activateABS ExecFrame* provides the *TurnABSoff* and *SetABSstate* Services to control the ABS feature. The *De_activateABS ExecFrame* provides a *Service* (i.e., *TurnABSoff*) that is required by the *De_ActivateACC ExecFrame*, and the two *ExecFrames* require a common *Service* (i.e., *TurnACCoff*).

The presence conditions mapping the features to the elements of the domain model are shown directly annotating the architecture elements. For example, the element *BodyControl* has the presence condition *F2 or F3*. Configuring the product line to produce a particular product involves selecting the features that should be in the product and then using these features with the presence conditions to extract the domain elements that should be in the product. For example, assume that we want to configure the product that has only feature *F2*. In this case, the product will contain the element *BodyControl* because its presence condition says that it is present when the product contains feature *F2* or if it contains *F3*. However, it will not contain element *SetABState* because its presence condition is *F8 or F3*.

3 Migrating GM Models to AUTOSAR

Previously, we reported on an industrial transformation that maps between subsets of a legacy metamodel for General Motors (GM) and the AUTOSAR metamodel [31]. This GmToAutosar transformation manipulated subsets of the metamodels that represent the deployment and interaction of software components. We summarize the source and target metamodels of the GmToAutosar transformation and its implementation in DSLTrans. More details on the source and target metamodels can be found in [31].

The GM Metamodel. Fig. 2 shows the subset of the GM metamodel manipulated by our transformation in [31]. A *PhysicalNode* may contain multiple *Partitions* (i.e., processing units). Multiple *Modules* can be deployed on a single *Partition*. A *Module* is an atomic, deployable, and reusable software element and can contain multiple *ExecFrames*. An *ExecFrame*, i.e., an execution frame,

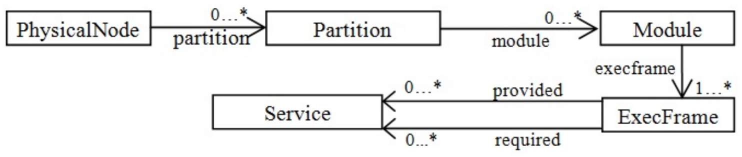


Fig. 2. Subset of the source GM metamodel used by our transformation in [31].

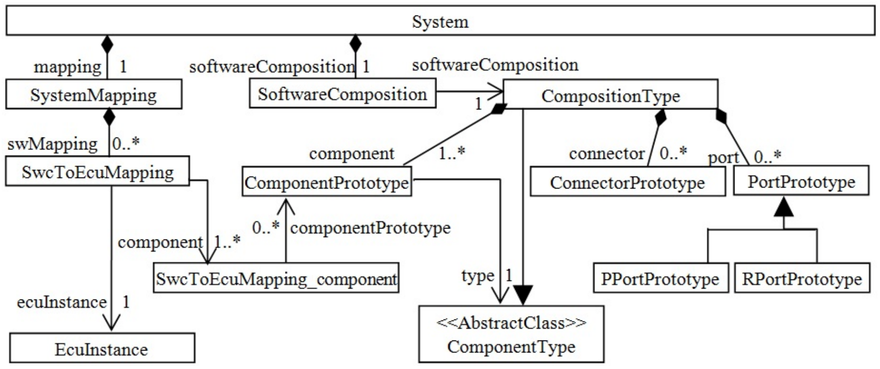


Fig. 3. Subset of the target AUTOSAR metamodel used by our transformation in [31].

is the basic unit for software scheduling. It contains behavior-encapsulating entities, and is responsible for providing/requiring *Services* to/from these behavior-encapsulating entities.

The AUTOSAR Metamodel. In AUTOSAR, an Electronic Control Unit (ECU) is a physical unit on which software is deployed. Fig. 3 shows the subset of the AUTOSAR metamodel [2] used by our transformation. In AUTOSAR, the ECU configuration is modeled using a *System* that aggregates *SoftwareComposition* and *SystemMapping*. *SoftwareComposition* points to *CompositionType* which eliminates any nested software components in a *SoftwareComposition*. *SoftwareComposition* models the architecture of the software components (i.e., *ComponentPrototypes*) deployed on an ECU and their ports (i.e., *PPortPrototype*/ *RPortPrototype* for providing/ requiring data and services). Each *ComponentPrototype* has a type that refers to its container *CompositionType*.

SystemMapping binds software components to ECUs using *SwcToEcuMappings*. *SwcToEcuMappings* assign *SwcToEcuMapping_components* to an *EcuInstance*. *SwcToEcuMapping_components*, in turn, refer to *ComponentPrototypes*.

The GmToAutosar Transformation. Although originally implemented in ATL [31], the GmToAutosar transformation was later reimplemented in DSLTrans for the purpose of a study where several of its properties where automatically verified [30]. This allowed us to increase our confidence in the correctness of the transformation. Table 1 summarizes the rules in each transformation layer of the GmToAutosar transformation after reimplementing it in DSLTrans, and the input/output types that are mapped/generated by each rule. For example, rule MapPhysNode2FiveElements in Layer 1 maps a *PhysicalNode* element in the

Layer	Rule Name	Input Types	Output Types
1	MapPhysNode2FiveElements	PhysicalNode, Partition, Module	System, SystemMapping, SoftwareComposition, CompositionType, EcuInstance
	MapPartition	PhysicalNode, Partition, Module	SwcToEcuMapping
	MapModule	PhysicalNode, Partition, Module	SwCompToEcuMapping_component, ComponentPrototype
2	MapConnPhysNode2Partition	PhysicalNode, Partition	SystemMapping, EcuInstance, SwcToEcuMapping
	MapConnPartition2Module	PhysicalNode, Partition, Module	CompositionType, ComponentPrototype, SwcToEcuMapping, SwCompToEcuMapping_component
3	CreatePPortPrototype	PhysicalNode, Partition, Module, ExecFrame, Service	CompositionType, PPortPrototype
	CreateRPortPrototype	PhysicalNode, Partition, Module, ExecFrame, Service	CompositionType, RPortPrototype

Table 1. The rules in each layer of the GmToAutosar transformation, and their input and output types.

input model to five elements in the output model (i.e., *System, SystemMapping, SoftwareComposition, CompositionType*, and *EcuInstance* elements). A detailed explanation of the mapping rules and the reimplementation of the transformation in DSLTrans can be found in [31,30]. DSLTrans and the notion of rule layers is described in Sec. 4.1.

4 Lifting GmToAutosar

4.1 Background: DSLTrans

DSLTrans is an out-place, graph-based and rule-based model transformation engine that has two important properties enforced by construction: all its computations are both *terminating* and *confluent* [6]. Besides their obvious importance in practice, these two properties were instrumental in the implementation of a verification technique for pre- / post-condition properties that can be shown to hold for all executions of a given DSLTrans model transformation, independently of the provided input model [21,22,30].

Model transformations are expressed in DSLTrans as sets of graph rewriting rules, having the classical left- and right-hand sides and, optionally, negative application conditions. The scheduling of model transformation rules in DSLTrans is based on the concept of *layer*. Each layer contains a set of model transformation rules that execute independently from each other. Layers are organized sequentially and the output model that results from executing a given layer is passed to the next layer in the sequence. A DSLTrans rule can match over the elements of the input model of the transformation (that remains unchanged throughout the

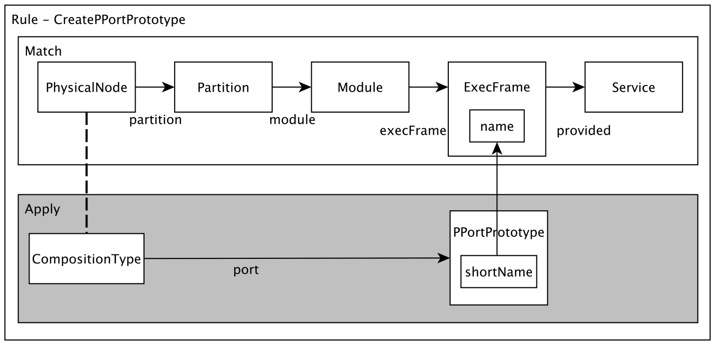


Fig. 4. The *CreatePPortPrototype* rule in the GmToAutosar DSLTrans transformation.

entire execution of the transformation) but also over elements that have been generated so far in the output model. The independence of the execution of rules belonging to the same layer is enforced by allowing matching over the output of rules from previous layer but not over the output of rules of the current layer. Matching over elements of the output model of a transformation is achieved using a DSLTrans construct called *backward links*. Backward links allow matching over traces between elements in the input the output models of the transformation. These traces are explicitly built by the DSLTrans transformation engine during rule execution.

For example, we depict in Fig. 4 the *CreatePPortPrototype* rule in the GmToAutosar DSLTrans transformation, previously introduced in Table 1. The rule is comprised of a *match* and an *apply* part, corresponding to the usual left- and right-hand sides in graph rewriting. When a rule is applied, the graph in the match part of the rule is looked for in the transformation's input model, together with the match classes in the apply part of the rule that are connected to *backward links*. An example of a *backward link* can be observed in Fig. 4, connecting the *CompositionType* and the *PhysicalNode* match classes. During the rewrite part of rule application, the instances of classes in the apply part of the rule that are not connected to backward links, together with their adjacent relations, are created in the output model. In the example in Fig. 4, the *CreatePPortPrototype* rule creates a *PPortPrototype* object and a *port* relation per matching site found. Note that the vertical arrow between the *shortName* attribute of *PPortPrototype* and the *name* attribute of *ExecFrame* implies that the value of attribute *name* is copied from its matching site to the *shortName* attribute of the*PPortPrototype* instance created by the rule.

In addition to the constructs presented in the example in Fig. 4, DSLTrans has several others: *existential matching* which allows selecting only one result when a match class of a rule matches an input model, *indirect links* which allow transitive matching over containment relations in the input model, and *negative application conditions* which allow to specify conditions under which a rule should not match, as usual. The GmToAutosar transformation does not make use of these constructs, and thus we leave the problem of lifting them for future work.

4.2 Lifting DSLTrans for GmToAutosar

Lifting of Production Rules. When executing a DSLTrans transformation, the basic operation (called here a *"production"*) is the application of a individual rule at a particular matching site site. The definition and theoretical foundation of lifting for productions are given in [27]. Below, we describe how they apply in the case of GmToAutosar using the model fragment in Fig. 1 and the *CreatePPortPrototype* rule in Fig. 4.

When a DSLTrans rule R is lifted, we denote it by $R^{\uparrow}$. Intuitively, the meaning of a $R^{\uparrow}$-production is that it should result in a product line with the same products as we would get by applying R to all the products of the original product line at the same site. Because of this, we do not expect a $R^{\uparrow}$-production to affect the set of allowable feature combinations in the product line. Formally:

Definition 2 (Correctness of lifting a production) *Let a rule R and a product line P, and an application site c be given. $R^{\uparrow}$ is a* correct lifting *of R iff (1) if $P \overset{R^{\uparrow}|c}{\Longrightarrow} P'$ then $\mathsf{Conf}(P') = \mathsf{Conf}(P)$, and (2) for all configurations $\mathsf{Conf}(P)$, $M \overset{R|c}{\Longrightarrow} M'$, where M can be derived from P and M' from P' under the same configuration.*

An algorithm for applying lifted rules at a specific site is given in [27], along with a proof of production correctness that is consistent with the above definition. In brief, given a matching site and a lifted rule, the algorithm performs the following steps: (a) use a SAT solver to check whether the rule is applicable to at least one product at that site, (b) modify the domain model of the product line, and (c) modify the presence conditions of the changed domain model so the rule effect only occurs in applicable products.

For example, consider the match c={*BodyControl*, *HumanMachineInterface*, *Display*, *De_ActivateACC*, *TurnABSoff*, *BodyControlCT*} in the fragment in Fig. 1. In this match, we assume that an element named *BodyControlCT* of type *CompositionType* and its corresponding backward link have been previously created by the rule *MapPhysNode2FiveElements* (see Table 1) and therefore have the presence condition $F2 \vee F3$. To apply the rule *CreatePPortPrototype*$^{\uparrow}$ to c, we first need check whether all of c is fully present in at least one product. We do so by checking whether the formula $\Phi_{apply} = (F2 \vee F3) \wedge (F8 \vee F3)$ is satisfiable. Φ_{apply} is constructed by conjoining the presence conditions of all the domain elements in the matching site c. According to the general lifting algorithm in [27], the construction of Φ_{apply} for arbitrary graph transformation rules is more complex; however, rules in GmToAutosar do not use Negative

Application Conditions and do not cause the deletion of any domain element. Therefore, the construction of Φ_{apply} follows the pattern we described for all rules in GmToAutosar$^{\uparrow}$.

Because Φ_{apply} is satisfiable, *CreatePPortPrototype*$^{\uparrow}$ is applicable at c. Therefore, the rule creates a new element called *De_ActivateACC* of type *PPortPrototype*, a link of type *port* connecting it to *BodyControlCT*, as well as the appropriate backward links. Finally, all created elements are assigned Φ_{apply} as their presence condition. In other words, the added presence conditions ensure that the new elements will only be part of products for which the rule is applicable. By construction, this production satisfies the correctness condition in Def. 2. Thus, according to the proofs in [27], the lifting of productions preserves confluence and termination.

Lifting the Transformation. We define the notion of global correctness for GmToAutosar$^{\uparrow}$ to mean that, given an input product line of GM models, it should produce a product line of AUTOSAR models that would be the same as if we had applied GmToAutosar to each GM model individually:

Definition 3 (Global Correctness of GmToAutosar$^{\uparrow}$) *The transformation GmToAutosar$^{\uparrow}$ is* correct *iff for any input product line P, it produces a product line P' such that (a)* $\mathsf{Conf}(P) = \mathsf{Conf}(P')$*, and (b) for all configurations* $\mathsf{Conf}(P)$*, $M' = GmToAutosar(M)$, where M and M' can be derived from P and P', respectively, under the same configuration.*

In order to lift GmToAutosar, we use the DSLTrans engine to perform the identification of matching sites and scheduling of individual productions, and use the lifting algorithm in [27] to lift individual productions, as described above. Since each production is correct with respect to Def. 2, then, by transitivity, the lifted version GmToAutosar$^{\uparrow}$ is globally correct. Also by transitivity, since the lifting of individual productions preserves confluence and termination, it is confluent and terminating, like GmToAutosar. Because of global correctness, and because it preserves confluence and termination, GmToAutosar$^{\uparrow}$ also preserves the results of the verification of pre- and post-condition properties using the techniques in [21,22,30]. In other words, GmToAutosar$^{\uparrow}$ satisfies the same set of pre- and post-condition properties as GmToAutosar.

Implementation. Adapting the DSLTrans engine for GmToAutosar$^{\uparrow}$ did not require changing the existing codebase. Instead, we had to write code to extend it in two ways: (a) Reading and writing presence conditions, expressed as Comma-Separated Values (CSV) and attached to EMF [16] models. (b) Interfacing with the API of the Z3 SMT solver [12], used for checking the satisfiability of Φ_{apply}. These changes required an addition of less than 300 lines of code to an existing codebase of 9250 lines.

5 Applying the Lifted Transformation GmToAutosar$^{\uparrow}$

The aim of this case study is to investigate the feasibility of applying industrial-grade transformations to product lines via lifting [27]. We thus lifted GmToAutosar and applied it to various input product lines with the goal to answer the following research questions:

RQ1: Does GmToAutosar$^\uparrow$ scale to industrial-sized SPLs?
RQ2: How sensitive is it to the complexity of the product line?

To answer RQ1, we generated realistic product lines, based on input from our industrial partners. We then applied GmToAutosar$^\uparrow$ to them and measured two variables: (a) total runtime, and (b) complexity of presence conditions of the output. We used the clause-to-variable ratio as a measure of the complexity of presence conditions because it is a well-known metric for evaluating the complexity of queries to SAT solvers. To answer RQ2, we varied the size of the generated product lines in terms of the size of the domain model and the number of features in the feature model.

Setup. Due to limitations of publication of sensitive industrial data, we opted to use a *realistic* rather than *real* product lines, constructed as follows:
1) Using publicly available examples [1], we created the exemplar product line described in Sec. 2. As described earlier, its domain model consists of 201 elements and its feature model has 6 features. 50% of domain model elements in the model had a single feature presence condition, whereas the presence conditions of the other 50% consisted of conjunctive clauses of 2-3 features. The overall product line was validated with input from our industrial partners.
2) We consulted our industrial partners regarding the characteristics of a typical product line. We were given the following parameters for a typical product line of DOORS requirements: (a) domain model size is 400 elements, (b) the number of feature variables is 25, (c) 1/8th of elements are variation points, (d) an average clause-to-variable ratio of the presence conditions is $^2/_{25} = 0.08$, i.e. an average presence condition consists of 2 clauses containing any of the 25 feature variables.
3) We used the exemplar model built in step 1 as a seed to create product lines of varying sizes for the model and the set of features, i.e., varying parameters (a) and (b) from step 2 while keeping parameters (c) and (d) constant. Therefore, models of increasing sizes were obtained by cloning the exemplar domain model to create models of 200, 400, 800, 1600 and 3200 elements. To obtain product lines with different numbers of feature variables, we cloned the feature model of the exemplar, creating feature models with 6, 12, 24, 48, and 96 features. The product line with 400 elements and 24 features corresponds to the parameters reported by our industrial partners in the previous step. Each variation point was assigned a randomly generated presence condition based on the presence conditions of the exemplar.

We executed the experiments on a computer with Intel Core i7-2600 3.40GHz $\times$4 cores (8 logical) and 8GB RAM, running Ubuntu-64.
Results. Fig. 5(a) shows the observed runtimes of applying GmToAutosar$^\uparrow$ to product lines with domain models of increasing size. One line is plotted for each feature set size. For comparison, we also include the runtime of applying GmToAutosar to models (not product lines) of different sizes. Fig. 5(b) shows the clause-to-variable ratio of output product lines for inputs of varying size of

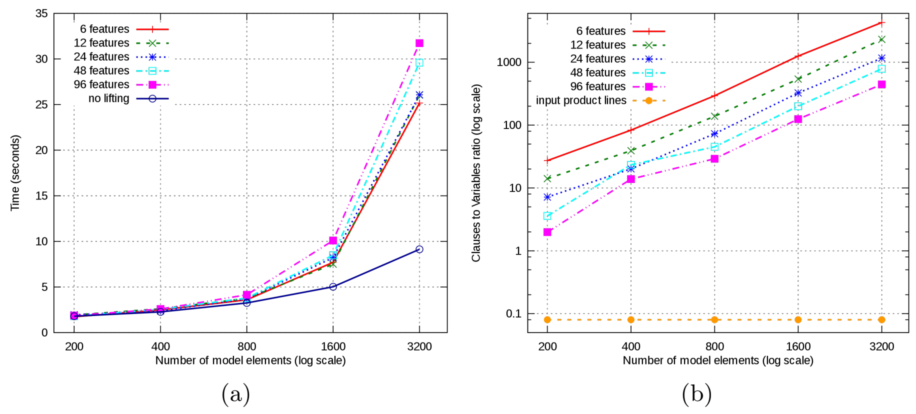


Fig. 5. (a) Observed increase in running time. (b) Observed increase in the size of presence conditions.

domain model. One line is plotted for each feature set size. For comparison, we also include the clause-to-variable ratio of the input product line.

With respect to RQ2, we note that runtime grows exponentially with the size of the domain model, while product lines with larger feature sets take longer to transform. The size of presence conditions also grows exponentially with increasing domain model sizes, and is two to three orders of magnitude larger than the input. Applying GmToAutosar$^{\uparrow}$ to product lines with smaller size of the feature set results in a larger increase to the clause-to-variable ratio. With regard to the sensitivity of GmToAutosar$^{\uparrow}$ to size of the domain model, we observe that runtime follows the expected pattern of exponential increase. Since the non-lifted version also grows exponentially, we conclude that this exponential increase is not solely due to the use of a SAT solver but also due to the inherent complexity of graph-rewriting-based model transformations. With regard to the sensitivity of GmToAutosar$^{\uparrow}$ to the size of presence conditions, we again observe an expected pattern of exponential increase. However, the increase is orders of magnitude large which is explained by the fact that our current implementation of GmToAutosar$^{\uparrow}$ does not perform any propositional simplification.

With respect to RQ1, we observe that for sizes of domain model and feature set that correspond to the description of real GM product lines, the observed runtime of GmToAutosar$^{\uparrow}$ is 3.59 seconds, compared to 3.25 for GmToAutosar. These differences in runtime indicate that GmToAutosar$^{\uparrow}$ scales well in terms of runtime. On the other hand we observe that the clause-to-variable ratio increased from 0.08 to 293.53, meaning that the output presence conditions contained a very large number of clauses. This points to the need to further optimize the DSLTrans engine, taking care to strike a balance between runtime and propositional simplification. Additionally, we note that the observed clause-to-variable ratio is not close to 4.26, which is considered to be the hardest for automated SAT solving [24].

Threats to Validity. There are two main threats to validity: First, the seed model was constructed using non-GM data, but rather publicly available automotive examples. Second, product lines of different sizes of domain model and feature set were artificially constructed by cloning the seed model. Both these issues stem from the fact that we could not access to real product lines due to limitations to publication of sensitive industrial data. To mitigate the first concern, we asked industrial partners to validate that our exemplar is realistic in terms of structure and size. To mitigate the second concern, we ensured that our cloning process resulted in product lines that had characteristics that were consistent with the parameters given by our industrial partners (number of variation points, average clause-to-variable ratio, shape of the presence conditions).

6 Lessons Learned and Discussion

The goal of this case study was to study the tractability of transformation lifting for industrial-grade transformations. In this section, we reflect on the experience of lifting GmToAutosar and describe the lessons learned from it.

We note that applying GmToAutosar to product lines fulfils a real industrial need to migrate legacy product lines to a new format. This validates the basic premise of our theory that lifting transformations for product lines is an industrially relevant endeavour. The observed results in Sec. 5 indicate that using GmToAutosar$^{\uparrow}$ is tractable for industrial-sized product lines, even if some additional optimization is required. It thus adds more evidence to the evaluation results obtained using experimentation with random inputs in [27]. This strengthens the claim that transformation lifting scales to real-world models.

A claimed benefit of transformation lifting is that transformations do not need to be rewritten specifically for product lines. Instead, what is required is the lifting of the transformation engine. This case study did not contradict this claim: we were able to migrate legacy GM product lines to AUTOSAR without having to rewrite the GmToAutosar transformation for product lines. Instead, we lifted the DSLTrans engine.

In [27], lifting was implemented using the Henshin graph transformation engine [5]. Specifically, we implemented lifting for graph transformations while *using* some capabilities of Henshin (e.g., matching) as a black box. However, lifting GmToAutosar required *adapting* part of the underlying transformation engine (DSLTrans) itself. The reason why this was possible was because the DSLTrans language is (a) based on graph-rewriting and (b) uses graph rewriting productions as atomic operations. It is thus possible to lift the entire engine by lifting just these atomic operations while leaving the rest of the matching and scheduling untouched. On the other hand, since GmToAutosar does not make use of certain more advanced language constructs in DSLTrans (e.g., indirect links), we were only required to make very targeted interventions to the DSLTrans engine. Lifting DSLTrans for arbitrary transformations will require more extensive changes. For some language features, most notably, existential matching, this also requires rethinking parts of the lifting algorithm from [27].

7 Related Work

There is extensive work on adapting software engineering techniques to product lines in order to avoid having to explicitly manipulate individual products [32]. Lifting has been applied to model checking [8], type checking [19], testing [20], etc. Our work fits in this category, focusing on lifting transformations.

The combination of product lines and model transformations has been extensively studied from the perspective of using transformations for configuring and refining product lines [10,15,17,11], and merging products and feature models [3,9,26], A theory of product line refinement along with a classification of commonly used refinement approaches is presented in [7]. Transformation lifting differs from these works because it is about adapting existing product-level transformations to the level of entire product lines, as opposed to creating transformations specifically for product lines.

Variant-preserving refactoring, aimed to improve the structure of source code, is presented in [28], for feature-oriented product lines [4]. This is accomplished by extending conventional refactoring with feature-oriented programming. Our lifting approach focuses on *annotative, model-based* product lines instead, and is not limited to structural improvement.

Approaches to product line evolution [23,29] focus on scenarios such as merging and splitting product lines, and changing the feature set or the domain model. The aim is usually to create templates for manually evolving the product line in a safe way. Our approach is to automatically evolve product lines by lifting product-level translation transformations, such as GmToAutosar. Safety is thus ensured by reasoning about the properties of the transformation at the product level [21,22,30].

8 Conclusion and Future Work

In this paper, we presented an empirical case study where we lifted GmToAutosar, a transformation that migrates GM legacy models to AUTOSAR, so that it can be used to transform product lines as opposed to individual products. Lifting required us to adapt the execution engine of DSLTrans, the model transformation language in which GmToAutosar is written. We experimented with the lifted transformation GmToAutosar$^{\uparrow}$, using realistic product lines of various sizes to study the effect of lifting to the execution time and the complexity of the resulting product line. The observations confirm our theory that lifted model transformations can be applied to industrial-grade product lines. However, more optimization is required in order to strike a balance between keeping the runtime low and avoiding the growth of the size of presence conditions. Our experience with lifting GmToAutosar indicates that lifting is feasible for transformation languages like DSLTrans, where individual productions can be lifted while reusing the engine for matching and scheduling. However, lifting the full range of language features (not used in GmToAutosar) requires rethinking our lifting method. In the future, we intend to lift the entire DSLTrans engine, to take into account its full range of advanced language features such as existential matching and transitive link matching. We also intend to leverage the experience of lifting an entire model transformation language to apply our approach to more complex and powerful transformation languages.

References

1. Automotive Simulink Examples, http://www.mathworks.com/help/simulink/examples.html#d0e477.
2. AUTOSAR Consortium. AUTOSAR System Template, http://{AUTOSAR}.org/index.php?p=3&up=1&uup=3&uuup=3&uuuup=0&uuuuup=0/{AUTOSAR}_{TPS}_{S}ystem{T}emplate.pdf, 2007.
3. M. Acher, Ph. Collet, Ph. Lahire, and R. France. Comparing Approaches to Implement Feature Model Composition. In *Proc. of ECMFA'10*, pages 3–19, 2010.
4. S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *J. of Object Technology*, 8(5):49–84, 2009.
5. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G Taentzer. "Henshin: advanced concepts and tools for in-place EMF model transformations". In *Proc. of MODELS'10*, pages 121–135, 2010.
6. B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *Proc. of SLE'10*, pages 296–305. Springer, 2010.
7. P. Borba, L. Teixeira, and R. Gheyi. A Theory of Software Product Line Refinement. *J. of Theoretical CS*, 455:2–30, 2012.
8. A. Classen, P. Heymans, P.Y. Schobbens, A. Legay, and J.F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proc. of ICSE'10*, pages 335–344, 2010.
9. A. Classen, P. Heymans, T. T. Tun, and B. Nuseibeh. Towards Safer Composition. In *Proc. of ICSE'2009, Companion Volume*, pages 227–230, 2009.
10. K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. of GPCE'05*, pages 422–437, 2005.
11. K. Czarnecki, S. Helsen, and U. Eisenecher. Staged Configuration Using Feature Models. In *Proc. of SPLC'04*, pages 266–283, 2004.
12. Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS'08*, LNCS, pages 337–340, 2008.
13. M. Famelis, R. Salay, and M. Chechik. Partial Models: Towards Modeling and Reasoning with Uncertainty. In *Proc. of ICSE'12*, pages 573–583, 2012.
14. R. Flores, Ch. Krueger, and P. Clements. Second-Generation Product Line Engineering: A Case Study at General Motors. In R. Capilla, J. Bosch, and K.C. Kang, editors, *Systems and Software Variability Management*, pages 223–250. Springer, 2013.
15. Kelly Garcés, Carlos Parra, Hugo Arboleda, Andrés Yie, and Rubby Casallas. Variability Management in a Model-Driven Software Product Line. *Revista Avances en Sistemas e Informática*, 4(2):3–12, 2007.
16. R. Gronback. *Eclipse Modeling Project.* Addison Wesley, 2009.

17. Ø Haugen, B Moller-Pedersen, Jon Oldevik, Gøran K Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Proc. of SPLC'08*, pages 139–148, 2008.
18. C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. of McGPLE Workshop at GPCE'08*, pages 35–40, 2008.
19. C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM TOSEM*, 21(3):14, 2012.
20. C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-aware Testing. In *Proc. of FOSD'12*, pages 1–8, 2012.
21. L. Lúcio, B. Barroca, and V. Amaral. A Technique for Automatic Validation of Model Transformations. In *Proc. of MoDELS'10*, pages 136–150. Springer, 2010.
22. L. Lúcio, B. J. Oakes, and H. Vangheluwe. A Technique for Symbolically Verifying Properties of Graph-Based Model Transformations. Technical Report SOCS-TR-2014.1, McGill University, 2014. http://msdl.cs.mcgill.ca/people/levi/30_publications/files/A_Technique_\for_Symbolically_Verifying_Properties_of_Model_Transf.pdf.
23. L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the Safe Evolution of Software Product Lines. *ACM SIGPLAN Notices*, 47(3):33–42, 2011.
24. E. Nudelman, K. Leyton-Brown, H. Hoos, A. Devkar, and Y. Shoham. Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In *Proc of CP'2004*, pages 438–452. 2004.
25. A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software Engineering for Automotive Systems: A Roadmap. In *Proc. of FOSE '07*, pages 55–71, 2007.
26. J. Rubin and M. Chechik. Combining Related Products into Product Lines. In *Proc. of FASE'12*, pages 285–300, 2012.
27. R. Salay, M. Famelis, J. Rubin, A. Di Sandro, and M. Chechik. Lifting Model Transformations to Product Lines. In *Proc. of ICSE'14*, pages 117–128, 2014.
28. S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *Proc. of VAMOS'12*, pages 73–81, 2012.
29. C. Seidl, F. Heidenreich, and U. Aßmann. Co-Evolution of Models and Feature Mapping in Software Product Lines. In *Proc. of SPLC'12*, pages 76–85, 2012.
30. G.M.K. Selim, L. Lúcio, J. Cordy, J. Juergen, and B.J. Oakes. Specification and Verification of Graph-Based Model Transformation Properties. In *Proc. of ICGT'14*, pages 113–129, 2014.
31. G.M.K. Selim, Sh. Wang, J.R. Cordy, and J. Dingel. Model Transformations for Migrating Legacy Models: An Industrial Case Study. In *Proc. of ECMFA'12*, pages 90–101. Springer, 2012.
32. T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis Strategies for Software Product Lines. *School of Computer Science, University of Magdeburg, Tech. Rep. FIN-004-2012*, 2012.