Extractive Software Product Line Engineering Using Model-Based Delta Module Generation

David Wille[1], Tobias Runge[1], Christoph Seidl[1], Sandro Schulze[2]

[1]TU Braunschweig, Germany
{d.wille, tobias.runge, c.seidl}@tu-bs.de

[2]Otto-von-Guericke-Universität Magdeburg, Germany
sandro.schulze@iti.cs.uni-magdeburg.de

ABSTRACT

To satisfy demand for customized products, companies commonly apply so-called clone-and-own strategies by copying functionality from existing products and modifying it to create product variants that have to be developed, maintained, and evolved in isolation. In previous work, we introduced a variability mining technique to identify variability information (commonalities and differences) in block-based model variants (e.g., MATLAB/Simulink models), which can be used to guide manual transition from clone-and-own to managed reuse of a software product line (SPL). In this paper, we present a procedure that uses the extracted variability information to generate a transformational delta-oriented SPL fully automatically. We generate a delta language specifically tailored to transforming models in the analyzed modeling language and utilize it to generate delta modules expressing variation of the SPL's implementation artifacts. The procedure seamlessly integrates with our variability mining technique and allows to fully adopt a managed reuse strategy (i.e., generation of products from a single code base) without manual overhead. We show the feasibility of the procedure by applying it to state chart and MATLAB/Simulink model variants from two industrial case studies.

CCS Concepts

•Software and its engineering → Software product lines;

Keywords

extractive product line engineering, delta modeling, variability mining, model-based, clone-and-own

1. INTRODUCTION

Industry faces the challenge of satisfying a growing demand for customized products. Thus, companies allow configuration of their products. For example, car manufacturers allow customization of their cars with additional optional functionality (e.g., driver assistance systems). However, development of the corresponding software *variants* is often realized without proper reuse strategies. For example, existing functionality is copied and modified to changed requirements (e.g., adding functionality). While these so-called *clone-and-own* approaches save money in short-term, they introduce new risks to the development process as the copied functionality is not maintained in a single code-base and the copies evolve independently [1]. For instance, errors have to be fixed manually and in isolation in all variants. Thus, maintenance becomes a time-consuming and costly task.

In previous work, we introduced *family mining*, a reverse-engineering technique enabling semi-automatic identification of variability information (i.e., common and varying parts) for related block-based model variants (e.g., *MATLAB/Simulink*[1] models or state charts) [24, 25, 26]. This information can provide guidance for developers during a manual transition to a *software product line (SPL)*. In literature, different approaches exist to transition variants from clone-and-own to SPLs in general [16, 17] and SPLs using the *common variability language (CVL)* in particular [2, 3, 12, 27]. However, so far, no approach exists to *automatically* transition from clone-and-own variant creation to a *delta-oriented SPL* and only support for conscious encoding of variability between edited model variants in such an SPL exists [14]. Delta modeling is a transformational variability mechanism that allows flexible and modular design of SPLs and uses operations in a *delta language* specifically tailored for the underlying modeling language to transform between product variants [22]. For example, DeltaSimulink is specifically designed to modify *MATLAB/Simulink* models [5]. Although delta-oriented SPLs represent an efficient reuse strategy, adopting this technique for existing variants involves large manual effort and, thus, is a costly task. A major challenge is the creation of a suitable delta language as knowledge about proper language design is needed. Furthermore, variability between existing variants has to be encoded carefully as incorrect modeling might compromise the created SPL (e.g., erroneous variants might be generated).

We present our approach that solves these challenges by enabling language-independent and automatic delta language generation based on the results of our variability mining. The generated delta-language is used to correctly encode the identified variability in delta modules for automatic transition of cloned variants to an SPL. We apply our family mining [24, 25, 26] to extract relations between variants. Using this information, we make the following contributions:

[1]http://www.mathworks.com/products/simulink/

VaMoS '17, February 01-03, 2017, Eindhoven, Netherlands

DOI: http://dx.doi.org/10.1145/3023956.3023957

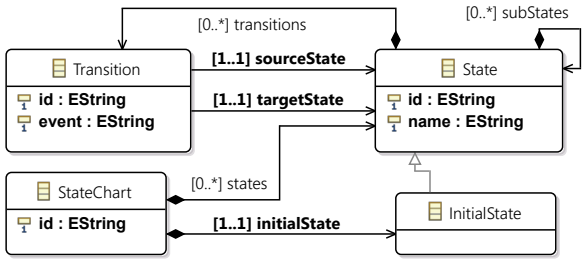


Figure 1: Meta-model used for the running example

- We generate a delta language for the block-based language analyzed during family mining.
- We generate reusable artifacts using the created delta language to allow generation of all SPL variants.
- We evaluate the feasibility of our approach using *IBM Rational Rhasody*[2] state charts from an SPL and *MATLAB/Simulink* models from an industrial case study.

This paper is structured as follows: Section 2 provides background on variability mining, SPLs, and delta modeling. Section 3 introduces our approach to generate a concrete delta language and respective delta modules used to store the extracted variability. Section 4 reports on a feasibility case study. Section 5 discusses related work and Section 6 concludes with an outlook to future work.

2. BACKGROUND

In Figure 1, we show the meta-model for the three state chart variants of an air-condition system in Figure 2a – 2c, which we use as a running example. Variant `V1` realizes a manually controlled cooling system with an additional fan consisting of the states `Off`, `Manual Cool`, and `Fan` with corresponding transitions. Variant `V3` realizes a similar cooling system but does not provide a fan. Variant `V2` realizes an automatically controlled cooling system without a fan consisting of the states `turned off` and `Auto Cool` with corresponding transitions. The variants are realized using hierarchical states to encapsulate their concrete functionality in the hierarchy (not displayed due to space limitations). Using this mechanism, the states `Manual Cool` and `Auto Cool` realize similar yet differing functionality to control the air-condition system. In addition, although their names differ, the states `turned off` and `Off` both realize the same functionality to switch off the system.

2.1 Software Product Lines

Software product lines (SPLs) allow structured reuse of shared and varying implementation artifacts across a *family* of related product *variants* [15]. In contrast to clone-and-own approaches, variability is realized through proper engineering in a single code base. Thus, SPLs ease maintenance of shared functionality and allow its efficient reuse across variants by using suitable generators for automatic variant derivation. *Features* represent high-level configuration options (e.g., a driver assistance system for cars) and map to reusable implementation artifacts (i.e., concrete code realizing a feature) [6]. During generation of a variant, the artifacts associated with the selected features are added to the generated product. *Extractive SPL engineering* creates an SPL from a set of related variants that were previously developed in isolation [8]. By analyzing the common and varying parts of such variants, extractive SPL engineering allows to encode the identified variability in reusable artifacts. However, this task is still a largely manual task as often no documentation of the relations between variants exists and the variants have to be compared manually.

[2]http://www.ibm.com/software/awdtools/rhapsody/

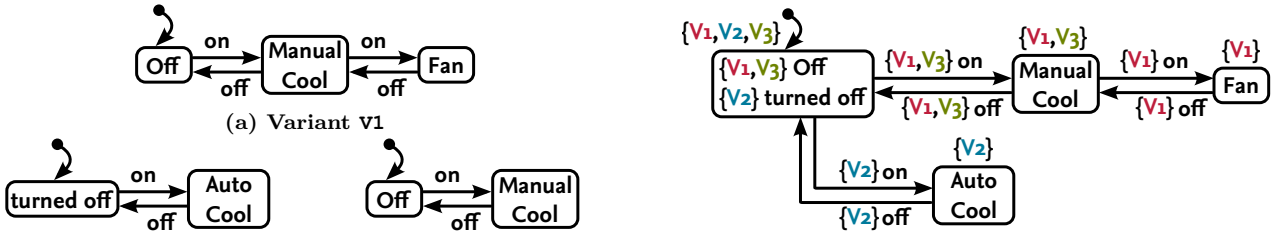
2.2 Family Mining

Our family mining from previous work allows to identify necessary variability information for extractive SPL engineering from a set of related block-based models [24, 25, 26]. During the *Compare Phase*, we identify possible relations between the models by iterating through the models and following the data-flow or execution-flow between model elements from the execution start (e.g., the initial states in Figure 2) to the end. To determine which model elements are equivalent, alternative, optional, or entirely unrelated, we calculate, for each comparison between two model elements, a *similarity value* according to a *metric*. The metric ranks the impact of the elements' properties on their overall similarity. The algorithm creates results which are not necessarily unambiguous because each element from a model might be compared to multiple elements from another model. Thus, we execute the *Match Phase* to identify distinct relations between the models (i.e., one-to-one relations for the model elements) by considering the similarity value of ambiguous elements and selecting the pairs of elements with the highest similarity. During the *Merge Phase*, these results are merged into a *150% model*. This model stores all artifacts from the compared and merged products together with explicit information on their variability and the containing products. We distinguish *mandatory* artifacts (i.e., common to all products), *alternative* artifacts (i.e., mutually exclusive between products), and *optional* artifacts (i.e., only contained in particular products). The created 150% model is used as a reference for comparison with the next product.

In Figure 2d, we show an exemplary 150% model for the comparison of the three air-condition system state chart variants from Figure 2a – 2c. For reasons of clarity, we limited the annotations to information needed by our proposed generation process and only show annotations `V1`, `V2`, and `V3` in curly brackets to annotate containing variants. We can see that `V1` and `V3` share the `Manual Cool` state and that `V1` contains the `Fan` state and `V2` the `Auto Cool` state. In addition, the algorithm identified differences for the mandatory initial state as it realizes the same functionality in all variants but has differing names (i.e., `Off` in `V1` and `V3` and `turned off` in `V2`). Due to space limitations, we do not show variability in the hierarchical states. The generated 150% model allows detailed analysis of the identified variability. However, it does not represent a complete SPL as significant manual effort still is needed to encode the variability in artifacts allowing automatic variant derivation.

2.3 Delta Modeling

Delta modeling (DM) is an approach to realize SPLs by applying transformational rules to derive new variants conforming to a user's product configuration [22]. DM allows definition of *delta modules* in a *delta language* specifically tailored to transform model instances realized in a certain modeling language. These delta modules store *delta operation calls* to the *delta operations* (i.e., transformation rules) defined by the corresponding language. These operations al-



(a) Variant V1

(b) Variant V2

(c) Variant V3

(d) 150% model comprising V1, V2, and V3

Figure 2: State charts modeling three air-condition system variants and the corresponding 150% model

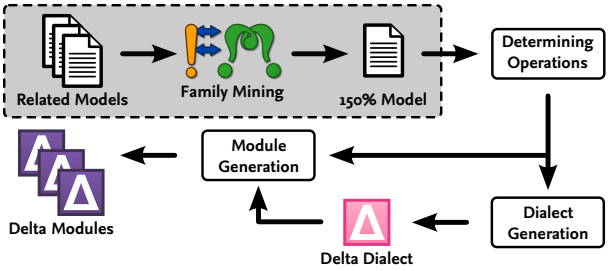


Figure 3: The workflow for the delta generation

low us to *add*, *remove*, and *modify* realization artifacts of an existing *core variant* (i.e., a complete variant implementing a valid feature configuration). For example, it is possible to add a new state to a state chart, to modify its properties, and to remove unneeded transitions. Due to its flexible nature (e.g., additional variability can be added by introducing new delta modules) and its applicability for extractive SPL engineering [22], we decided to apply DM for our SPL generation from results created during family mining.

DeltaEcore[3] is a completely model-based tool suite allowing developers to easily define and use delta languages for EMOF[4]-based languages [23]. The framework consists of the *common base delta language*, which can be used to define a delta language by providing a *delta dialect* for a specific programming/modeling language.

The common base delta language provides basic delta operations: While *set* and *unset* operations assign a new value to a single-valued reference or replace the current reference with the default value, *add* and *remove* operations add a reference to a many-valued reference or remove it, respectively. In addition, an operation to *modify* object attributes exists.

Delta dialects define a set of delta operations to alter concrete model instances realized in a certain modeling language (cf. the state chart delta dialect in Listing 1 generated by our approach). These operations have a *signature*, which differs from classic programming languages: Each operation can be identified unambiguously by the used delta operation type from the common base delta language (e.g., *modify*), the altered reference or attribute, and the operation's parameter types. Thus, the operation name only represents a user-specified label used during delta module definition.

The user-defined delta dialect can be used to generate a *concrete delta language* using the operations provided by the common base delta language. Besides the base operations to create new delta languages for model-based languages, DeltaEcore provides editor support for such languages, feature modeling support and variant derivation using specified delta modules. Thus, DeltaEcore provides all necessary facilities for SPL engineering from variability language creation and variability modeling to variant generation.

Our family mining is based on the *Eclipse Modeling Framework (EMF)*[5] meta-model *Ecore* and, thus, also supports model-based languages defined using an EMOF-based notation. Using DeltaEcore to generate reusable SPL artifacts enables us to make use of the provided facilities.

3. DELTA GENERATION APPROACH

In this section, we describe the details of our model-based approach to create an SPL from the variability information identified during family mining of related variants. We apply DM to model the SPL's variability and generate delta modules corresponding to the identified variability. To ease usage for modeling languages without previously created delta language, we also provide automatic generation of delta dialects that can directly be applied during delta module creation.

In Figure 3, we show the workflow of the generation process. First, our family mining is executed on a set of related model variants to create a 150% model of the identified variability. As this algorithm is not part of the paper's contribution, we marked this part with a gray box. For more details on the actual algorithm, we refer to [24, 25, 26]. In theory, this mining can be replaced by other approaches as long as they create 150% models conforming with variability annotations created by our family mining. We process these annotations to determine and collect the needed delta operations (cf. Section 3.1). This information is used to generate a suitable delta dialect (cf. Section 3.2). Afterwards, the delta language derived from the generated dialect together with the common base delta language provides delta operations used for the delta module generation (cf. Section 3.3).

3.1 Determining Transformation Operations

To generate delta modules for the created SPL, the executed family mining algorithm has to provide all variability information in form of annotations (e.g., `EAnnotations`) to the elements of the 150% model. In this section, we describe the algorithm to determine needed transformation operations to realize variable functionality between variants.

As delta modeling is a transformational approach, it transforms a core variant `CV` into new variants (cf. Section 2.3). The `CV` can be chosen freely by the user to generate SPLs with different cores. The algorithm identifies, for elements from the 150% model, whether they have to be *added*, *removed*, or *modified* to transform the selected `CV` to a trans-

[3]http://www.deltaecore.org/

[4]EMOF (Essential MOF) is part of the Meta-Object Facility (MOF) standard by the Object Management Group (OMG), see http://omg.org/mof

[5]https://www.eclipse.org/modeling/emf/

formed variant TV. To generate corresponding decisions, the process expects as input the extracted 150% model together with the identifiers for core variant CV and the transformed variant TV whose delta operations should be generated. Thus, it is important to note that the selected CV highly influences the determined set of needed operations and the subsequent steps generating the delta language (cf. Section 3.2) and corresponding delta modules (cf. Section 3.3). For example, when selecting a CV containing only the shared elements of all other variants the need for *remove* operations might not be identified as only elements need to be *added* to the TVs.

The algorithm analyzes, for each element in the 150% model, how it has to be represented in the delta modules by processing the elements' information about models containing the element. No action is required when the element *is not* contained in CV and TV or if the element *is* contained in CV and TV but *no differences* were identified between the elements of both variants. However, in all other constellations, a transformation is required to realize the changes between both variants. We consider the following cases to determine a suitable operation to apply: In case the element is contained in CV but not in TV, we have to *unset/remove* it from the transformed variant. In case the element is *not* contained in CV but in TV, we have to *set/add* the element to TV. For both cases, we distinguish modifications of single-valued and multi-valued references as these require differing operations in the meta-model representation (cf. Section 2.3). For single-valued references, the algorithm returns *set* or *unset* decisions and for multi-valued references *add* or *remove* decisions. In our example, the algorithm only returns *add* or *remove* operations, as all references, except for the `initialState`, `sourceState`, and `targetState` references, are multi-valued (cf. Figure 1). In case differences between elements exist (e.g., the differing `name` of state `Off` / `turned off` in Figure 2d), a *modify* operation is needed.

In Table 1, we present the decisions created by the algorithm for our running example in Figure 2d with `CV = V1`. As we can see, the family mining correctly identified that the internal functionality of the initial states `Off` and `turned off` does not differ. However, during a transformation of `CV = V1` to `TV = V2` they are still regarded as *not* identical because the name in CV has to be *modified* to `turned off`. In addition, the states `Manual Cool` and `Fan` are *removed* and the state `Auto Cool` is *added* as they are stored in the multi-valued reference `states` in `StateChart` (cf. Figure 1). For space reasons, we do not discuss the decisions for contents of the hierarchical states and transitions. For a complete transformation between `CV = V1` and `TV = V2` or `TV = V3` they have to be generated in a similar manner.

3.2 Creating the Delta Language

Using the described algorithm to determine delta operations, we are now able to generate a delta dialect for the used meta-model and the selected CV. We execute the algorithm to determine needed operations (cf. Section 3.1) for each TV and generate the delta operations for the returned decisions. The combination of the delta operation type (e.g., *add* or *remove*) and the modified reference (e.g., the `states` reference of `StateChart`) uniquely describes an operation independently of its name. All generated delta operations are stored in a set to prevent generating the same declaration multiple times. For each needed operation, we generate delta operations from references by identifying the referencing classes (e.g., the `StateChart` class) and creating operations to alter (e.g., to *add* or *remove* an element) the given reference (e.g., the `states` reference of `StateChart`). The operations' names are automatically generated in a similar manner by concatenating the action (e.g., *add* state) with the modified reference (e.g., the `states` reference) and its referencing class (e.g., the `StateChart` class).

```
deltaDialect {
  configuration:
    metaModel: <http://www.tu-braunschweig.de/isf/
        familymining/stateoriented>;

  deltaOperations:
    addOperation addStateToStatesOfStateChart(
        State value, StateChart [states] element);
    removeOperation removeStateFromStatesOfStateChart(
        State value, StateChart [states] element);
    modifyOperation modifyNameOfState(String value,
        State [name] element);
    // ...
}
```

Listing 1: The generated delta dialect (excerpt)

```
delta "V1->V2"
  dialect <http://www.tu-braunschweig.de/isf/
      familymining/stateoriented>
  modifies <V1.statechart> {
    removeStateFromStatesOfStateChart(<manual_cool>,
        <state_chart>);
    removeStateFromStatesOfStateChart(<fan>,
        <state_chart>);

    State s = new State(id: "auto_cool",
        name: "Auto Cool");
    addStateToStatesOfStateChart(s, <state_chart>);

    modifyNameOfState("turned off", <off>);
    // ...
}
```

Listing 2: The generated delta module (excerpt)

In Listing 1, we show an excerpt of the delta dialect generated for the 150% model in Figure 2d and its meta-model in Figure 1 using the decisions from Table 1. As we can see, the automatically generated names give a precise idea of the operations' functionality. However, users might not agree with theses names and might prefer more expressive or concise names (e.g., `addState` instead of `addStateToStatesOfStateChart`). Thus, we consider the generated delta dialect as a basis for a custom delta language, where names can be edited easily to meet the user's preferences.

3.3 Generating the Delta Modules

Using the delta dialect derived from the 150% model, we are now able to generate the corresponding delta language and use it during delta module generation. Instead of using the generated delta dialect, users can also manually define their delta dialect and provide it for the delta module generation. In this case, all needed operations have to be identified and created manually. To generate delta modules for the compared variants from an input 150% model, we execute, for each variant TV, the algorithm to determine needed operations (cf. Section 3.1). The resulting decisions are used to identify the delta operations needed to encode the identified variability using the meta-model's delta dialect. During this operation look-up process, we use our operation's signature to identify a distinct operation: First, we filter all operations from the meta-model's delta dialect according to the processed decision and only

TV	Element	Determination Process	Action
V2	Off / turned off	Decide $\xrightarrow{in\ CV}$ in CV $\xrightarrow{\substack{in\ TV\ \&\\ !identical}}$ Modify	Change name attribute from Off to turned off
V2	Manual Cool	Decide $\xrightarrow{in\ CV}$ in CV $\xrightarrow{!in\ TV}$ Remove	Remove state Manual Cool
V2,V3	Fan	Decide $\xrightarrow{in\ CV}$ in CV $\xrightarrow{!in\ TV}$ Remove	Remove state Fan
V2	Auto Cool	Decide $\xrightarrow{!in\ CV}$!in CV $\xrightarrow{in\ TV}$ Add	Add state Auto Cool

Table 1: Decisions for the 150% model in Figure 2d with CV = V1 and TV = V2 / V3

consider matching operations. For example, when looking for the addStateToStatesOfStateChart operation in Listing 1, we filter out all *set*, *unset*, *remove*, and *modify* operations and only search all possible *add* operations. Afterwards, we check for each of the remaining operations whether its declared parameters conform with the information that should be processed by the operation (i.e., in our example that a State should be added to the states references of a StateChart). When identifying a matching operation, we create a corresponding operation call and store it in a delta module. Afterwards, we continue with the next decision. For added elements (e.g., the State in Listing 1), we also generate corresponding constructor calls. The generated delta module header contains information on the used delta dialect and a reference to the modified core variant. In Listing 2, we show a delta module for the 150% model in Figure 2d to transform CV = V1 to TV = V2. Due to space limitations, we did not include the delta operation calls for transitions and contents of hierarchical states, which have to be generated for a complete transformation.

Our approach generates model-based delta dialects and delta modules using corresponding meta-models. Thus, we realize a completely model-based workflow starting with the family mining [24, 25, 26] and ending with the delta dialect and delta module generation. The generated dialects and modules can be printed to textual files (cf. Listing 1 and Listing 2). Using the described approach, transferring a set of related product variants from ad-hoc reuse strategies (e.g., through clone-and-own) to an SPL is not a costly manual task anymore. Instead, by applying our approach, developers can fully automatically transfer these variants to a delta-oriented SPL with managed reuse across variants.

4. CASE STUDY

We applied the presented approach to *IBM Rational Rhapsody* state chart variants from an existing SPL and to *MATLAB/Simulink* models from an industrial case study. We evaluated the following research questions:

RQ1 – Language Independence: *Is the proposed generation technique capable of generating delta languages and delta modules for different modeling languages?*

RQ2 – Correctness: *Are the generated delta languages and delta modules capable of correctly transforming a core variant into the corresponding target variants?*

4.1 Case Study Subjects

For our evaluation, we use two case studies with industrial background modeling software from the automotive domain.

Driver Assistance System (DAS): We had access to real world *MATLAB/Simulink* models from the *SPES_XT*[6] project realizing the *driver assistance system (DAS)* of a car. These *MATLAB/Simulink* models comprise the self-contained features *EmergencyBreak*, *FollowToStop*, *SpeedLimiter*, *CruiseControl*, and *Distronic* with two dependencies. First, the *FollowToStop* feature requires either the *CruiseControl* or the *Distronic* feature. Second, the *Distronic* feature always requires the *CruiseControl* feature. Thus, the set of features can be combined to create 18 variants of different size. For our evaluation, we concentrate on eight product combinations with an increasing complexity. Starting with small differences between the variants, we increase the number of differing features until a large distance exists and the variants realize mostly different functionality. This allows us to analyze the correctness of our generation for scenarios with an increasing complexity.

Body Comfort System (BCS): We use a set of *IBM Rational Rhapsody* state chart variants from the *Body Comfort System (BCS)* case study. The BCS was realized as an SPL by decomposing a real world automotive software system into 27 reusable features using best practices in SPL design [11]. These features encapsulate the functionality of the BCS (e.g., the alarm system or the central locking system) and allow derivation of 11,616 valid variants. During our evaluation, we concentrate on 18 product variants (P0 – P17) that were derived from the BCS using a pairwise sampling algorithm and, thus, build representative variants for the SPL [11]. For these variants, we execute our approach comparing P1 – P17 with P0, which represents the core of the BCS SPL. By applying the generated results, we evaluate whether our approach is capable of correctly transforming this core variant into the different products.

In Table 2, we present statistics for both selected case studies. Table 2a contains information on the number of contained classes, references, and attributes of the meta-models created to process the case study models during family mining. As we can see, the meta-model used for variants of the *DAS* case study is less complex than the *BCS* case study meta-model and contains a lower number of classes, references and attributes. In addition, Table 2b depicts information on the average number of model elements used in the models of the considered case studies. We distinguish elements realizing model hierarchy (subsystems in *MATLAB/Simulink* and regions in state charts), normal model elements (blocks in *MATLAB/Simulink* and states in state charts), and connections between these elements (connectors in *MATLAB/Simulink* and transitions in state charts). As we can see, the *DAS* models are larger than the *BCS* models and contain more elements on average.

4.2 Methodology

In Table 2, we can see that the meta-models and corresponding model instances for the selected case studies differ. Thus, selecting these two case studies with different proper-

[6]http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html

Models	Classes	References		Attributes	
		Single	*Multi*	*Single*	*Multi*
DAS	3	3	4	3	0
BCS	22	6	9	10	4

(a) Structure of the meta-models used for the case studies

Models	Subsystems / Regions	Blocks / States	Connectors / Transitions
DAS	78.4	730.8	762.4
BCS	40.2	21.9	65.6

(b) Average number of elements in the used models

Table 2: Statistics for the models and their corresponding meta-models used during our case study

ties (i.e., differing complexity of meta-models and modeled variants) allows us to analyze our proposed approach regarding different aspects. First, we evaluate the language independence of our approach to generate delta languages and delta modules for differing languages without additional implementation effort by the user and, second, we evaluate the correctness of the corresponding results.

All selected cases of our case study were executed using the approach presented in Section 3 with our family mining algorithm from [24, 25, 26]. For each case, we generated a delta dialect for the respective notation's meta-model, derived the corresponding delta language, and generated the needed delta module. Afterwards, we evaluated the generated artifacts regarding our research questions by applying them to the selected core variant using DeltaEcore. The resulting variants were compared manually with the corresponding variants that served as an input to the family mining algorithm. Thus, the input variants served as ground truth for our delta module generation as they allow direct correctness analysis of the results. All selected combinations were executed by a single developer using the described setup on a laptop with a 2.7 GHz Intel i7 processor and 12 GB RAM. Each combination was executed 10 times to reduce the influence of inaccurate runtime measuring.

4.3 Results & Discussion

Next, we report on our results and discuss them with respect to our research questions. The average runtime during the case study was 944 ms for the *BCS* (mining: 886 ms, delta generation: 57 ms) and 2341 ms for the *DAS* (mining: 1484 ms, delta generation: 857 ms). During the delta generation the algorithms generated on average 279 delta operation calls for the *BCS* (add: 126, remove: 14, set: 119, unset: 19, modify: 1) and 1564 delta operation calls for the *DAS* (add: 712, remove: 66, set: 716, unset: 66, modify: 4).
RQ1 – Language Independence: During the execution of our case study, we were able to successfully generate delta dialects for the meta-models used to store the models from the *DAS* and *BCS* case studies. We were able to derive corresponding delta languages and the generated dialects. In the following step, we used the corresponding delta language to automatically generate delta modules from the results of our family mining approach. During a manual analysis of the generated delta languages, we identified that their delta operations are correct (i.e., their execution triggers the expected transformations) and complete (i.e., all needed operations are generated). In addition, as the approach did not need additional implementation effort to generate these delta languages and corresponding delta modules for our case studies with differing languages, we argue that it is language-independent. Thus, we can answer *RQ1* positively.
RQ2 – Correctness: During the comparison of the variants derived from the generated delta modules with the ground truth, we identified that not all of the generated results were correct. While, for the *BCS* case study, all results were correct, only about 95% of the *DAS* cases were correct. After investigating the reasons for the incorrect results, we identified a minor bug in the *Merge Phase* of the family mining implementation. This bug was related to the merging of connectors and we identified that our implementation did not handle certain rarely occurring cases with optional connectors. Corresponding situations did not occur with previous case studies used to evaluate the implementation and, thus, we did not encounter them before. After fixing these problems, we executed the generation again and revalidated the results. This time, we encountered no problems and, thus, identified the results of our delta generation to be correct with respect to the ground truth. Thus, it is important to note that our approach can only be as good as the provided variability information. As long as the input 150% models contain valid information, the generation can create valid delta modules. Overall, we can answer *RQ2* positively as the generated delta modules correctly transform the selected core variants into the expected variants.

4.4 Threats to Validity

Threats to validity for our case study are mostly related to the selected models as they are limited to two case studies. Thus, the created results might not be representative for other languages and case studies. However, we selected models from two case studies with different modeling paradigms (i.e., data-flow-oriented for *MATLAB/Simulink* vs. state-oriented for state charts) to be confident that our delta generation approach is applicable for a wider range of models. Furthermore, the selected case studies differ in complexity and size of contained models and the corresponding meta-model used during family mining. In addition, we implemented our delta generation technique using synthetic examples and only afterwards applied the approach to the selected case studies. Thus, the created results are not overfitting for a single case study and show that our approach is capable of handling differing settings. Another threat to validity is that both meta-models used in the cases studies were created by authors of the paper. Thus, a similar modeling style might be used. However, as the meta-models were created according to best practices in model-driven development, we are confident that their design is close to solutions of other developers using similar guidelines.

In previous work, we manually evaluated the corresponding results to be correct in a sense that they conformed with the expectations of experienced SPL developers [25, 26]. Nevertheless, other developers might have a differing idea of such results. Although this is a threat to validity, we demonstrated with our presented approach that we are capable of processing the variability information identified during family mining to create delta modules allowing derivation of all initial input variants. Thus, we have successfully executed extractive generation of SPLs using the proposed approach.

5. RELATED WORK

In this section, we discuss related work on delta language generation and on extracting SPL implementation artifacts.

Delta Language Generation: Different approaches exist to generate transformational languages to manage variability. For instance, approaches exist to extend textual grammars using custom delta languages [4] and to apply general purpose transformation languages to realize variability management [28]. In addition, approaches exist to generate domain specific model transformation languages from the concrete syntax of domain specific languages (DSLs) without considering variability information [18]. In contrast, we use the model-based facilities provided by DeltaEcore and do not use general purpose transformation techniques. Thus, we can build our delta dialect generation on the metamodel representation of any language (e.g., textual or graphical) [23]. In addition, DeltaEcore's facilities allow type checking and editor support for the generated language with integration into the common variant derivation of DeltaEcore [23]. Overall, we not only derive a delta dialect with tooling from our family mining results but also generate an SPL with delta modules using the created dialect.

Extracting SPL Implementation Artifacts: Different approaches exist that are closely related to our work. For instance, Pietsch et al. calculate the differences between two models after editing a copy of a single model to realize a new variant and encode the variability in form of delta modules [14]. The authors assume that their approach is used in a scenario, where the user consciously decides to formalize variability between variants while editing them. In contrast, our approach transfers a complete family of existing variants to an SPL were the variability was previously realized in an unmanaged way (e.g., using clone-and-own approaches).

In addition, approaches exist to generate CVL-based SPLs from model variants [2, 3, 12, 27]. Zhang et al. introduce the tool CVL Compare to formalize identified commonalities and differences of variants in an SPL [27]. In addition, Font et al. extract variability information from compared models by identifying larger model patterns [3]. Furthermore, Font et al. introduce an approach to include the domain engineers' knowledge to identify variability information meeting their expectations [2]. These approaches store the identified variability in CVL models, which do not allow in-depth analysis of the family's variability. In contrast, we first create a merged 150% model allowing domain experts to analyze the identified variability prior generating SPL artifacts.

Martinez et al. identify and visualize features with directly assigned artifacts describing the variability of model variants [13]. The authors extend their work by generating CVL artifacts from 150% models storing the feature artifacts [12]. While the authors use a 150% model when deriving variants using the CVL artifacts, we are much more flexible and derive variants from an existing valid variant by using not only removals but also modifications and additions. This allows easy extension of the identified 150% model with new variants and generation of corresponding delta modules.

Ryssel et al. allow automatic generation of block-based models using elements from language-independent libraries storing blocks with compatible data ports [21]. To ease the transition from existing models to a generative solution based on their language-independent format, Ryssel et al. describe an approach to identify and store variation points from block-based models [19] and to derive corresponding feature models [20]. While the authors focus on providing generative reuse of models from existing model libraries, we provide variability information from complete products and variant generation from reusable delta modules.

Rubin et al. describe the formal foundations for merging products into an SPL [16]. Furthermore, the authors propose an approach to simultaneously compare and merge a potentially arbitrary number of model variants [17]. Unlike our approach, the authors only merge variants into an SPL and do not provide facilities to derive the compared variants.

Linsbauer et al. concentrate on dependencies between features and their implementation artifacts to identify feature interaction [10]. Although using traces for their evaluation to regenerate the analyzed variants, the authors do not show how to manage these traces for variant derivation in an SPL.

Lee et al. identify reusable parts from a set of functionally related products, which are developed independently by different groups of developers and, thus, have differing realization artifacts [9]. In contrast to our approach, these differing artifacts cannot be used to automatically derive SPL artifacts and manual reimplementation might be needed.

Kästner et al. describe an approach to transform back and forth between physically separated features (i.e., modules) and virtually separated features (i.e., 150% models) in source code [7]. While we identify variability information prior transforming the corresponding 150% models to modules, the authors assume variability information to be already available to transform between the notations.

6. CONCLUSION

In this paper, we presented a completely model-based approach to apply extractive software product line engineering to a set of related models that were realized by clone-and-own approaches. Our approach seamlessly integrates with our variability mining from previous work [24, 25, 26] and provides language-independent and automatic facilities to generate delta languages for different modeling languages. The approach uses the newly generated delta language to correctly encode the variability between the variants in delta modules. Using models from two case studies with industrial context, we showed the feasibility of our approach. Overall, our approach overcomes the challenges that previously hindered manual adoption of delta-oriented SPLs. Most importantly, the previously error-prone encoding of variability between variants is replaced by automatic generation facilities to correctly encode the variability in delta modules.

In future work, we plan to evaluate our approach with additional case studies from different languages. Furthermore, we currently work on identifying coherent variable functionality in form of features encapsulating related functionality.

7. ACKNOWLEDGMENTS

This work was partially supported by the European Commission within the project HyVar (grant agreement H2020-644298).

8. REFERENCES

[1] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. "An Exploratory Study of Cloning in Industrial Software Product Lines". In: *European Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 25–34.

[2] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. "Building Software Product Lines from Conceptualized Model Patterns". In: *Intl. Software Product Line Conference (SPLC)*. ACM, 2015, pp. 46–55.

[3] J. Font, M. Ballarín, Ø. Haugen, and C. Cetina. "Automating the Variability Formalization of a Model Family by Means of Common Variability Language". In: *Intl. Software Product Line Conference (SPLC)*. ACM, 2015, pp. 411–418.

[4] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, I. Schaefer, and C. Schulze. "Systematic Synthesis of Delta Modeling Languages". In: *Software Tools for Technology Transfer* 17.5 (2015), pp. 601–626.

[5] A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer. "First-class Variability Modeling in Matlab/Simulink". In: *Intl. Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, 2013, 4:1–4:8.

[6] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Carnegie-Mellon University, 1990.

[7] C. Kästner, S. Apel, and M. Kuhlemann. "A Model of Refactoring Physically and Virtually Separated Features". In: *Intl. Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 2009, pp. 157–166.

[8] C. W. Krueger. "Easing the Transition to Software Mass Customization". In: *Intl. Workshop on Software Product-Family Engineering*. Springer, 2002, pp. 282–293.

[9] H. Lee, H. Choi, K. C. Kang, D. Kim, and Z. Lee. "Experience Report on Using a Domain Model-Based Extractive Approach to Software Product Line Asset Development". In: *Intl. Conf. on Software Reuse (ICSR)*. Springer, 2009, pp. 137–149.

[10] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. "Variability extraction and modeling for product variants". In: *Software & Systems Modeling* (2016), pp. 1–21.

[11] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer. *Delta-oriented Software Product Line Test Models – The Body Comfort System Case Study*. Tech. rep. 2012-07. Technische Universität Braunschweig, 2012.

[12] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. l. Traon. "Automating the Extraction of Model-Based Software Product Lines from Model Variants". In: *Intl. Conf. on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 396–406.

[13] J. Martinez, T. Ziadi, J. Klein, and Y. le Traon. "Identifying and Visualising Commonality and Variability in Model Variants". In: *European Conf. on Modeling Foundations and Applications (ECMFA)*. Vol. 8569. LNCS. Springer, 2014, pp. 117–131.

[14] C. Pietsch, T. Kehrer, U. Kelter, D. Reuling, and M. Ohrndorf. "SiPL – A Delta-Based Modeling Framework for Software Product Line Engineering". In: *Intl. Conf. on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 852–857.

[15] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[16] J. Rubin and M. Chechik. "Combining Related Products into Product Lines". In: *Intl. Conf. on Fundamental Approaches to Software Engineering (FASE)*. Vol. 7212. LNCS. Springer, 2012, pp. 285–300.

[17] J. Rubin and M. Chechik. "N-way Model Merging". In: *European Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 301–311.

[18] B. Rumpe and I. Weisemöller. "A Domain Specific Transformation Language". In: *Intl. Workshop on Models and Evolution (ME)*. 2011.

[19] U. Ryssel, J. Ploennigs, and K. Kabitzsch. "Automatic library migration for the generation of hardware-in-the-loop models". In: *Science of Computer Programming* 77.2 (2012), pp. 83–95.

[20] U. Ryssel, J. Ploennigs, and K. Kabitzsch. "Automatic Variation-point Identification in Function-block-based Models". In: *Intl. Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 2010, pp. 23–32.

[21] U. Ryssel, J. Ploennigs, K. Kabitzsch, and M. Folie. "Generative Design of Hardware-in-the-Loop Models". In: *Intl. Workshop on Automatic Program Generation for Embedded Systems (APGES)*. ACM, 2007.

[22] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *Software Product Lines: Going Beyond*. Vol. 6287. LNCS. Springer, 2010, pp. 77–91.

[23] C. Seidl, I. Schaefer, and U. Aßmann. "DeltaEcore – A Model-Based Delta Language Generation Framework". In: *Modellierung*. 2014, pp. 81–96.

[24] D. Wille. "Managing Lots of Models: The FaMine Approach". In: *Intl. Symposium Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 817–819.

[25] D. Wille, S. Schulze, and I. Schaefer. "Variability Mining of State Charts". In: *Intl. Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2016, pp. 63–73.

[26] D. Wille, S. Schulze, C. Seidl, and I. Schaefer. "Custom-Tailored Variability Mining for Block-Based Languages". In: *Intl. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE, 2016, pp. 271–282.

[27] X. Zhang, Ø. Haugen, and B. Møller-Pedersen. "Model Comparison to Synthesize a Model-Driven Software Product Line". In: *Intl. Software Product Line Conference (SPLC)*. IEEE, 2011, pp. 90–99.

[28] S. Zschaler, P. Sánchez, J. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, and U. Kulesza. "VML* – A Family of Languages for Variability Management in Software Product Lines". In: *Intl. Conf. on Software Language Engineering (SLE)*. Springer, 2010, pp. 82–102.