Variability Mining of State Charts

David Wille
TU Braunschweig, Germany
d.wille@tu-bs.de

Sandro Schulze
TU Hamburg-Harburg, Germany
sandro.schulze@tuhh.de

Ina Schaefer
TU Braunschweig, Germany
i.schaefer@tu-bs.de

Abstract

Companies commonly use state charts to reduce the complexity of software development. To create variants with slightly different functionality from existing products, it is common practice to copy the corresponding state charts and modify them to changed requirements. Even though these so-called clone-and-own approaches save money in the short-term, they introduce severe risks for software evolution and product quality in the long term as the relation between the software variants is lost so that all products have to be maintained separately. In previous work, we introduced variability mining algorithms to identify the relations between related MATLAB/Simulink model variants regarding their common and varying parts. In this paper, we adapt these algorithms for state charts by applying guidelines from previous work to make them available for developers to better understand the relations between a set of state chart variants. Using this knowledge, maintenance of related variants can be improved and migration from clone-and-own based single variant development to more elaborate reuse strategies is possible to increase maintainability and the overall product quality. We demonstrate the feasibility of variability mining for state charts by means of a case study with models of realistic size.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software

Keywords variability mining, state charts, block-based language, clone-and-own

1. Introduction

State charts, originally introduced by Harel [13], increasingly gained importance for developing complex software systems (e.g., in the automotive domain). For instance, in early phases of the development process, state charts are used in the *UML specification*[1] to describe the internal behavior of components on an abstract level before using this specification to develop and test the actual functionality. In later phases of the development process, state charts are used to implement the concrete behavior of components and generate executable code for these descriptions.

While state charts allow companies to save money when developing individual software systems, creating *software families* consisting of multiple *variants* is still a time consuming and complex task. For instance, car manufacturers allow customers to create customized variants of their cars by providing configuration options (e.g., additional driver assistance systems can be added). To reduce the effort needed to create such software variants, copying existing state charts that realize the software and modifying them to changed requirements is common practice. While this so-called *clone-and-own* approach is an efficient means to create related yet slightly differing variants, problems arise in the long-term as the relation between the created variants is rarely documented and possible errors propagate from the initially cloned variant to new variants [9]. As a result, fixing identified errors in all variants becomes a time-consuming task because developers have to manually compare all variants to identify affected parts in all of them.

One way to overcome the clone-and-own related problems is to introduce managed reuse to the family of product variants. By identifying *variability information* (i.e., the variants' common and varying parts) from related software variants, developers are able to better understand the relations between the software variants. Using this information, the overall maintenance can be improved as changes between variants can be propagated more easily and existing functionality can be reused in a managed way [6]. In literature, a number of different algorithms exist to merge state charts and annotate the source variant of the merged elements [12, 21, 22, 24, 26]. However, most of these approaches do not natively support comparison of states modeling parallel execution [12, 24, 26] or have to translate them to non-parallel structures to allow their merging [21]. Such a translation presents the resulting merges in a non-intuitive presentation and might confuse developers analyzing the results. Most importantly, the approaches do not an-

[1] `http://www.omg.org/spec/UML/`

FOSD'16, October 30, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4647-4/16/10...
http://dx.doi.org/10.1145/3001867.3001875

notate explicit variability information between elements on a fine grained level (e.g., which states are alternatives to each other) [12, 21, 22, 24, 26], only annotate from which variants the elements originate [21, 22, 24, 26], or only identify the variability on a coarse grained level [24, 26] (i.e., complete features with sets of elements are identified). Thus, no fine-grained variability information is available to support developers during maintenance. In previous work, we successfully demonstrated a mining technique to automatically analyze and reverse engineer such variability information from *The Mathworks MATLAB/Simulink*[2] models [38, 39]. However, despite their common graph-based structure with nodes (i.e., blocks or states), which are linked with edges (i.e., connectors or transitions), these languages differ in underlying concepts. Thus, we presented guidelines to adapt our variability mining for languages with differing concepts [40].

In this paper, we apply these guidelines to adapt our existing mining algorithms for a large set of state chart notations. In particular, we make the following contributions:

- We analyze a large set of state chart notations with industrial relevance to consider corresponding notation-specific language elements for the adaption.
- We adapt our family mining for these notations to identify fine grained variability and discuss the extensions needed to consider all available information.
- We demonstrate the applicability of our family mining algorithms by means of a case study with realistic industrial-scale state charts.

This paper is structured as follows: Section 2 provides background on state charts and our existing family mining algorithms from previous work. Section 3 describes all necessary extensions to adapt the algorithms for state charts. Section 4 provides a case study to demonstrate applicability of family mining to state charts. Section 5 discusses related work and Section 6 concludes with an outlook to future work.

2. Background

In Section 2.1, we describe all language elements that need to be considered when comparing state charts. In Section 2.2, we describe our existing family mining algorithms.

2.1 State Charts

A large number of differing state chart notations exists in academia [34] and industry (e.g., *The Mathworks Stateflow*[3] or state charts in *IBM Rational Rhapsody*[4]). We base our work on the initial notation defined by Harel [13] but also consider language specific elements from other notations (cf. Section 3.1). State charts (cf. Figure 1) model the execution states of a system and allow to process data in these *states* (e.g., state `X` in Figure 1) by executing *code* defined in a general purpose programming language (e.g., C or JAVA).

[2] `http://www.mathworks.com/products/simulink/`

[3] `http://www.mathworks.com/products/stateflow/`

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

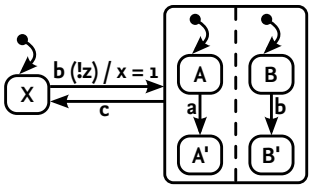
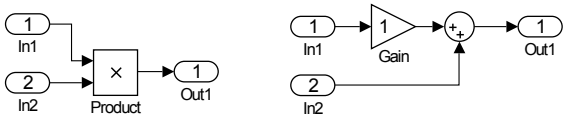


Figure 1: State chart example



(a) Model 1 (M1) **(b)** Model 2 (M2)

Figure 2: Exemplary *MATLAB/Simulink* model variants

Transitions between these states can define actions to process data. According to Harel [13], transitions define labels $\alpha(P)/S$ consisting of an *event* α triggering the execution of the corresponding transition (i.e., the code S) when its *condition* P is fulfilled. For instance, in Figure 1, the code `x = 1` is executed when the event `b` occurs in state `X` and the condition `!z` is fulfilled.

To provide a higher degree of abstraction, state charts allow to define hierarchical elements in order to solve a concrete problem by starting on an abstract level and refining it with each added hierarchy level. For this purpose, state charts allow to define *hierarchical states*. In addition, they provide *parallel states* to model parallel execution. For example, after leaving state `X` in Figure 1, a parallel state is entered. Here, the *regions* separated by dashed lines are executed concurrently. Consequently, the state of the current execution is defined by one state from the left side and one from the right side. State charts only allow to define a *single* point for the execution start (i.e., an *initial state*) in each hierarchy level or parallel region, which is indicated by a black dot with a transition to a state (i.e., state `X` in Figure 1).

2.2 Family Mining

Our family mining algorithm [38, 39] aims at reverse engineering fine grained *variability* information between implementation artifacts (e.g., states) of product variants. This includes information on parts of the software family that are *mandatory* (i.e., common to all variants), *alternative* (i.e., mutually exclusive), or *optional* (i.e., only contained in particular variants). The identified variability is represented in so-called *150% models* storing all implementation artifacts of the software family together with their variability. Analyzing this information helps developers to understand the relations between compared variants and, thus, helps to improve the maintainability of software families. Furthermore, it builds a suitable basis to migrate from ad-hoc variant creation to more reliable and sophisticated reuse strategies within a *software product line (SPL)* by mapping artifacts to *features* [6]. Using these features as configuration options,

different variants from the family can be generated. Using the *MATLAB/Simulink* models from Figure 2, we explain our existing family mining technique.

2.2.1 Compare Phase

Our existing family mining algorithm [38, 40] compares two *MATLAB/Simulink* models starting at the *start blocks* (i.e., blocks introducing data to models, such as the *Inports* `In1` and `In2` in Figure 2). Following the data-flow (i.e., the connectors), we only compare blocks that are part of the same *stages*. A stage contains only blocks, which are separated by the same number of blocks in relation to the start blocks (e.g., in Figure 2 the *Product* block is compared with the *Gain* and *Sum* blocks). Each comparison is represented by a *compare element* storing the elements under comparison and a *similarity value*. These similarity values are calculated using a *metric* [39] to assign different *weights* to the attributes of blocks. Such weights are adjustable to the users' requirements and allow to rank the attributes' influence on the overall functionality (e.g., the blocks' names have a lower impact compared to their functions).

2.2.2 Match Phase

The created list of possible compare elements might contain ambiguities as model elements often are considered in multiple compare elements. For instance, four compare elements (i.e., $[M1.In1, M2.In1]$, $[M1.In1, M2.In2]$, $[M1.In2, M2.In1]$, and $[M1.In2, M2.In2]$) are created during the comparison of the *Inport* blocks in Figure 2. Thus, our family mining algorithm [38, 40] identifies distinct matches by selecting the compare elements with the highest similarity in order to rule out ambiguous elements. In certain situations, compare elements with the same similarity value exist and, thus, the algorithm cannot directly identify a distinct match. To implicitly solve such conflicts by matching other compare elements first, we sort corresponding compare elements to the end of all possible compare elements. In case this solution does not solve the conflict, we either use an user-defined automatic algorithm or allow manual selection.

2.2.3 Merge Phase

After identifying distinct matches between the compared models, the resulting list of compare elements can be used to create a merged 150% model. To classify the identified variability during the merging, our family mining algorithm [38, 40] uses the following adjustable mapping function $rel(CE_i)$ to analyze the similarity value sim_{CE_i} for each compare element CE_i. We identified these thresholds by comparing the impact of differing properties on their similarity according to the experience of domain engineers [39].

$$rel(CE_i) \leftarrow \begin{cases} mandatory & sim_{CE_i} \geq 0.95 \\ alternative & 0 < sim_{CE_i} < 0.95 \\ optional & sim_{CE_i} = 0 \end{cases}$$

For mandatory compare elements, we allow small deviations between the compared blocks (e.g., their names have changed) and, thus, consider all blocks with a similarity of 95% as mandatory. Blocks without any matching partner result in a similarity of 0% and are regarded as optional elements as they are only contained in one of the models. The remaining compare elements store alternative blocks because major differences were identified between them (e.g., their function differs). All compare elements are merged into a 150% model containing all blocks from the compared models together with information on their identified variability and the models from which they originate. These models can be used to create a graphical representation of the product family or for the comparison with further models.

3. Adapting Family Mining for State Charts

Our current family mining algorithms for *MATLAB/Simulink* rely on model-based techniques and, thus, our guidelines in [40] request a meta-model representation of the analyzed language. Thus, the first step during the adaption is to create a meta-model representation for state charts (cf. Section 3.1). Following our guidelines in [40], we define a metric to allow the comparability of state chart elements by assigning weights to their properties and to calculate corresponding similarity values (cf. Section 3.2). Using the results from these two steps, we follow the guidelines in [40] to adapt the algorithms for the three family mining phases (i.e., *Compare*, *Match*, and *Merge*) and identify the variability information in related state charts (cf. Section 3.3). We directly compare the adaptions for state charts with the existing realization for *MATLAB/Simulink* to explicitly point out their differences. After executing these steps, we are able to identify variability information between related state chart variants and can use it to improve their maintenance.

3.1 Unified Meta-Model for State Charts

Creating a meta-model representation for state charts allows to reduce the analyzed information to relevant language elements (e.g., *states* and *transitions*) and neglect irrelevant information (e.g., the position or color of states) [40]. We decided to analyze a large set of different state chart notations to create a meta-model capable of representing different notations and, thus, allowing to adapt family mining not only for a single state chart notation but a larger set of notations. As clone-and-own related maintenance problems mostly arise in industrial contexts, we mainly focused on notations commonly used in industry (*The Mathworks Stateflow*, *ETAS ASCET*[5], *IBM Rational Rhapsody*, and *Esterel Technologies SCADE Suite*[6]) and the *UML specification*. This way, we are able to adapt our mining algorithms for state chart notations with industrial relevance.

[5] `http://www.etas.com/ascet/`

[6] `http://esterel-technologies.com/products/scade-suite/`

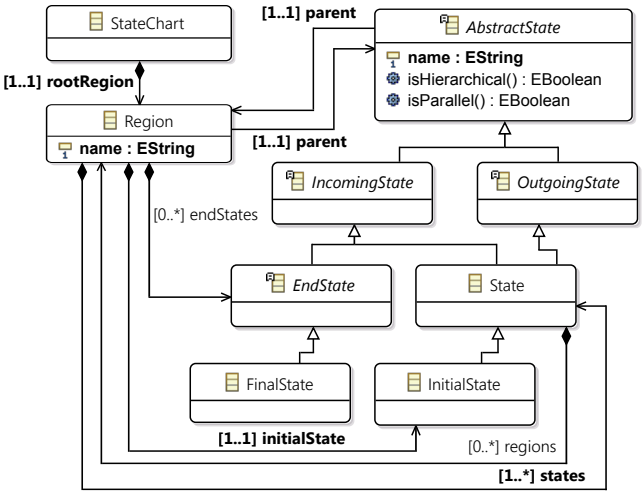


Figure 3: Excerpt from the state chart meta-model

Our analysis of these state chart notations and their documentation resulted in a set of state chart elements, which we modeled in a corresponding meta-model. We identified that state charts consist of a *root region* representing the highest hierarchy level. Such regions represent containers for states in the different hierarchy levels of state charts. We identified three different types of states, which can be contained in regions. Regular states can contain further regions allowing to model hierarchical states (i.e., states containing exactly one region) and parallel states (i.e., states containing more than one region). As initial states allow incoming and outgoing transitions (cf. state `X` in Figure 1), they extend regular states and define the execution start. Deterministic execution is guaranteed by only allowing a single initial state per region. In contrast, end states only accept incoming transitions and terminate the execution. Each regular state can contain actions, which are normally defined by program code (e.g., JAVA or C) and can be executed on different occasions (e.g., when entering or leaving a state). Transitions allow to link states and define transition labels, which describe events triggering the execution of a transition under certain conditions resulting in the execution of actions represented by program code. In our created meta-model, all states, regions, transitions, transition labels and the different action types were modeled using meta-classes. All properties, such as the states' names and contents of transition labels, are modeled using attributes. An excerpt of the resulting meta-model showing only the states and regions can be found in Figure 3. We leave a complete evaluation of the meta-model's capabilities to store models in the considered languages for future work. However, all preliminary tests with small examples in the different notations were positive.

3.2 Variability Metric for State Charts

The created state chart meta-model allows to define metrics to calculate the similarity of compared state chart elements [40]. Similar to blocks in *MATLAB/Simulink*, states represent an important part of the functionality in state charts and, thus, have to be considered in a corresponding metric. Connectors in *MATLAB/Simulink* do not contain information relevant for comparisons as their names are optional and no other relevant information exists. In contrast, we consider transitions for state chart metrics as they highly influence the execution with their events, conditions, and actions. While hierarchy in *MATLAB/Simulink* is modeled by adding blocks inside blocks, the hierarchy in state charts is modeled using additional container elements (i.e., regions). Furthermore, parallel executions can be modeled by adding multiple regions to a state. For hierarchical *MATLAB/Simulink* blocks, we calculate the similarity by comparing their interfaces and names. Their functional similarity is identified by calculating the average of their sub block similarity values. We follow a similar approach for regions and calculate the average of their sub state and sub transition similarity values.

In Table 1, we show a ranking for a selected set of state and transition properties together with the corresponding weights for the evaluation in Section 4. For each property, we analyzed their influence on the execution of state charts together with their impact on the similarity of the compared elements. For both model elements, we distinguish between *static* and *dynamic* portions, which we separated by a line. While, static properties (e.g., the states' names) do not influence the execution of state charts, dynamic properties have an influence on the execution (e.g., the actions triggered in an executed state). We weighted these portions with a ratio of 0.5 / 0.5 for states and 0.2 / 0.8 for transitions, respectively. This allows us to give the dynamic parts of transitions a strong weight as their events and actions have a high influence on the functionality of state charts.

We identified the names to have a very low impact on the similarity of states and transitions as they might change between product variants. Start states and end states define the start and end of the execution and, thus, influence the execution. However, as they only select the start and end point and do not actively influence the behavior for the complete execution, we considered them as static properties with a low influence on the similarity. For our evaluation, end states are the only property with a weight of 0.0 as they are not part of our case study subjects (cf. Section 4). To consider end states in other models, this weight has to be changed to a value > 0. The behavior of parallel states is defined by their sub elements and, thus, we ranked the property whether a state is a parallel state with very low impact on the similarity. However, we consider the sub elements when comparing the regions contained in parallel states. The hierarchy distance d_h of two states is calculated by identifying how many hierarchy levels exist between them (e.g., for states on level one and two $d_h = 2-1 = 1$). We use $e^{-|d_h|}$ to calculate an exponential decay of d_h on the similarity of two compared states (i.e., larger distances between two states result in lower similarities). The neighbor property can be compared to the in-

Property	Influences execution	Impact on similarity	weight
name	–	very low	0.2
start state	(✓)	low	0.2
end state	(✓)	low	0.0
parallel state	(✓)	very low	0.1
hierarchy distance	–	low	0.25
neighbors	(✓)	low	0.25
dependent on events	✓	very high	0.4
triggered actions	✓	very high	0.4
events triggering change	✓	low	0.2

(a) Ranking of state properties

Property	Influences execution	Impact on similarity	weight
name	–	very low	1.0
events	✓	high	0.3
conditions	✓	high	0.3
actions	✓	very high	0.4

(b) Ranking of transition properties

Table 1: Ranking for selected state and transition properties

terfaces in family mining for *MATLAB/Simulink* [39] and, thus, compares the predecessors and successors of the states according to the neighbors' names and actions. The dynamic portion of states is defined by three properties. States are dependent on certain events and, thus, are only executed when these events occur. In turn, they trigger actions during execution. Both these properties have a high impact on the states' similarity as they highly influence the state charts' behavior. Leaving a state is only possible if defined events are triggered. As other states are dependent on these events, they are considered during other comparisons and, thus, we assigned a lower weight to events triggering state changes.

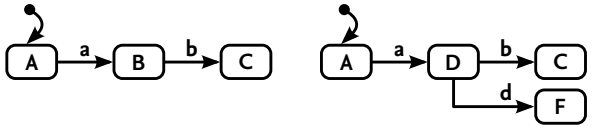
Transitions only have the name as a single static property. Similar to states, it has a very low impact on similarity. The dynamic part of transitions consists of events, conditions and actions. Events and conditions both have a high impact on similarity as they trigger the execution of transitions (i.e., events) or check preconditions to only execute them in defined cases. We ranked the influence of actions very high as they trigger new events and, thus, have a high impact on the similarity of transitions. In addition to the ranking of properties in Table 1, we rank the influence of sub states and sub transitions for compared regions with a ratio of 0.5 / 0.5 when calculating their similarity values. Table 1 only contains the most common properties of states and transitions in the analyzed notations (cf. Section 3.1).

3.3 Variability Analysis for State Charts

The created meta-model representation for the analyzed state chart notations and the corresponding metric are used to adapt our existing mining algorithms [38]. After executing these steps, we are able to identify variability between related state charts. We use the state charts from Figure 4 to explain the adaption of these algorithms for state charts.

3.3.1 Adapted Compare Phase

The first step during family mining is to create possible compare elements for the state charts in comparison [40].



(a) State Chart 1 (SC1) **(b)** State Chart 2 (SC2)

Figure 4: Exemplary state charts

After selecting two models, we compare their regions on the highest hierarchy level (i.e., the state charts' root regions) by identifying their initial states (i.e., in Figure 4 both `A` states). Starting from these states, we separate the compared state charts into stages. Similar to *MATLAB/Simulink*, a stage for states only contains states that have the same distance (i.e., the same number of previous transitions) relative to the initial states. In contrast to family mining for *MATLAB/Simulink*, we consider the transitions between the states as they contain relevant information for the analysis (cf. Section 3.2). Thus, we also identify stages for transitions, with each stage only containing transitions separated by the same number of previous transitions in relation to the initial states. We identify the state stages $(SC1[A], SC2[A])$, $(SC1[B], SC2[D])$, $(SC1[C], SC2[C, F])$ and transition stages $(SC1[a], SC2[a])$, $(SC1[b], SC2[b, d])$ for our example in Figure 1.

After identifying the stages for states and transitions, we create compare elements by comparing each stage from one model with the corresponding stage from the other model. For each comparison, we create a corresponding compare element similar to the comparisons for *MATLAB/Simulink* models (cf. Section 2.2.1). In contrast to *MATLAB/Simulink* models, state charts contain explicit elements to encapsulate elements contained in the hierarchy (i.e., regions). To consider these elements during comparison of models, we also create corresponding compare elements storing the comparison results for their sub states and sub transitions together with a similarity value averaging their calculated similarity. We distinguish between three scenarios. The first scenario

compares two states without any hierarchy or parallel execution and is handled as described. The second scenario compares one state without any hierarchy or parallel execution with a hierarchical or parallel state. In this case, we only compare all static properties of the states together with transitions the states are dependent on (i.e., incoming transitions triggering their execution) and transitions triggering a change of state (i.e., outgoing transitions triggering a state change). The third scenario compares two states, which either contain hierarchy or define parallel execution. For this scenario, all comparison steps are executed similar to scenario two with the only difference that compare elements for the regions contained in the states are created by comparing each region from one model with each region from the other model. These comparisons are executed by recursively triggering the comparison of regions (i.e., the same as for the root regions). For scenarios two and three, we do not compare the states' triggered actions as the functionality of hierarchical or parallel states is defined by their corresponding sub elements and not actions defined by source code.

3.3.2 Adapted Match Phase

After comparing the state charts and creating corresponding compare elements, we have to identify distinct matches for them [40]. Similar to our existing family mining, the created compare elements might be ambiguous as the model elements are contained in multiple compare elements. When comparing the created compare elements for state charts with the ones created for *MATLAB/Simulink*, we see that they contain the same information (i.e., two compared model elements and a corresponding similarity value). To distinctively match relations between *MATLAB/Simulink* models (cf. Section 2.2.2) the concrete contents of compare elements are irrelevant for our algorithm. It solely operates on the compare elements independent of the contained model elements and only considers the resulting similarity value to identify distinct matches. Thus, we can apply this matching algorithm without any modifications to the compare elements created for states, transitions, and regions.

3.3.3 Adapted Merge Phase

After identifying distinct matches, the corresponding variability has to be merged into a single model to create a 150% model of the analyzed product family [40]. To realize this merging process for state charts, we follow an approach similar to family mining for *MATLAB/Simulink* models (cf. Section 2.2.3) and use the same relation $rel(CE_i)$ to identify the similarity between compared state chart elements. The main difference between merging for *MATLAB/Simulink* and state charts is the number of elements that have to be considered. While the merging for *MATLAB/Simulink* only considers the merging of blocks and afterwards creates correct connections based on these results, the merging for state charts has to consider states, regions, and transitions. Starting with the *root region*, we first merge the corresponding sub states and only afterwards merge all transitions. When merging two states, we recursively merge their sub regions in order to consider all sub functionality. Here, we apply an additional strategy to split up certain compare elements for regions.

To explain such scenarios, we consider two states `A` and `B` realizing parallel execution with regions `W`, `X1`, and `Y` for state `A` and regions `W`, `X2`, and `Z` for state `B`. Given that `X1` and `X2` are variants of each other, our algorithm would identify the two `W` regions as mandatory elements and the regions `X1` and `X2` as alternatives. Due to the combinatoric comparison of regions (cf. Section 3.3.1), the algorithm would select the compare element containing the regions `Y` and `Z` and merge them as alternatives. Given that the names `Y` and `Z` indicate that these regions realize a completely unrelated functionality, the algorithms should not identify them as alternatives (i.e., mutually exclusive) but as optional elements. To solve such situations, we compare the names of alternative regions (e.g., using the Levenshtein distance [16]) and, in case that they are not similar enough (we used a threshold of 80% similarity), we regard the regions as optional. Despite the simplicity of the approach, it identified and fixed incorrect matchings for models from the SPL analyzed during our evaluation (i.e., we used the SPL's variability information to compare it with our results). Here, it is important to keep in mind that this solution will obviously fail if the names of related regions differ too much (e.g., after a major name refactoring between variants). In future work, we plan to further investigate this issue using additional case studies.

4. Evaluation

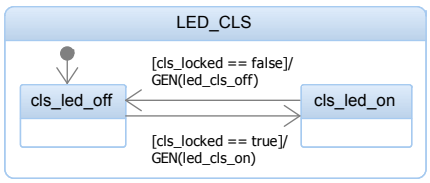
We applied the adapted mining algorithms (cf. Section 3) to state charts from an SPL case study and evaluated the following research questions to analyze the feasibility.

RQ1: *Is the adapted family mining algorithm capable of correctly identifying variability information between related state charts with regard to the variability modeled in the used SPL case study?*

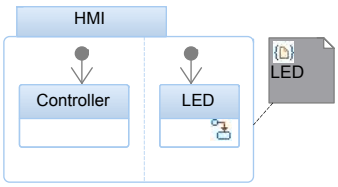
RQ2: *Does the adapted family mining algorithm for state charts scale for compared models with a large set of model elements?*

4.1 Setup & Methodology

The *Body Comfort System (BCS)* case study was realized as an SPL by decomposing a real world automotive software system into reusable *features* using best practices in SPL design [18]. All functionality of the BCS was realized using state charts in *IBM Rational Rhapsody*. The features encapsulate parts of implementation artifacts to realize specific functionality in products and can be composed to derive different product variants. The resulting BCS SPL consists of 27 features and allows to derive 11,616 valid product variants. The state charts from the case study include initial states, regular states, parallel states, hierarchical states,



(a) Contents of the hierarchical state `LED CLS`



(b) Contents of the parallel state `HMI`

Figure 5: Exemplary states from the BCS case study

and transitions with events, conditions, and actions. In Figure 5, we can see the contents of two exemplary states from the 150% model for the BCS SPL. Figure 5a shows the hierarchical state `LED CLS` modeling an LED for the *central locking system (CLS)* to show whether the car is locked. Figure 5b shows the parallel state `HMI` modeling the *human machine interface (HMI)* of the car consisting of two regions with the sub states `Controller` and `LED`. The gray flag `LED` indicates that the region containing the `LED` state is only part of a specific variant when the `LED` feature is present.

Although the BCS only contains a subset of all language elements available for state charts, it allows to evaluate if the adapted family mining algorithms for state charts are able to correctly identify variability information corresponding to the variability modeled in the BCS SPL. In addition, we are able to evaluate the correct comparison of states and transitions and the comparison of multiple hierarchical containers (i.e., regions), which are the major extensions to the existing algorithms. During our evaluation, we concentrate on the 18 product variants (`P0` – `P17`), which were derived from the BCS in [18] and contain between 91 and 283 elements (i.e., states, regions, and transitions). In theory, our approach allows to compare n variants with each other (dependent on memory limitations of the used hardware). However, we concentrated on pairwise combinations during our evaluation as our main goal was to evaluate whether the identified variability is correct in a sense that it conforms with the BCS SPL variability model. Thus, we manually compared the generated results with the variability modeled in the BCS SPL. As the BCS SPL was realized using SPL best practices, this comparison allows us to validate that our algorithms provide good results (i.e., the identified variability conforms with such best practices). With a growing number of compared state charts, a manual analysis of the results would be infeasible. However, the results for all comparisons of more than two models were identified to be correct. From the executed 27 pairwise comparisons, the first 17 compare the product variants `P1` – `P17` with product variant `P0`, which represents the *core* of the BCS SPL. This core is the common basis of all other product variants and, thus, comparing it with the other product variants allows us to evaluate whether the variability in relation to the core is identified correctly. Here, all additional features extending the core have to be identified as optional parts with regard to the BCS SPL. All other combinations compare product variants containing different variations of the same feature. For instance, `P1` and `P11` contain alternative variations of the `CLS` feature. By applying our adapted family mining algorithms to these cases, we evaluate whether these alternatives are identified correctly. The selected combinations compare between 184 and 468 model elements (i.e., states, regions, and transitions). All comparisons were executed by a single developer using the metric weights from Table 1 in Section 3.2 on a laptop with a 2.7 GHz Intel i7 processor and 12 GB RAM. Each comparison between two variants was executed 10 times to reduce the influence of inaccurate runtime measuring.

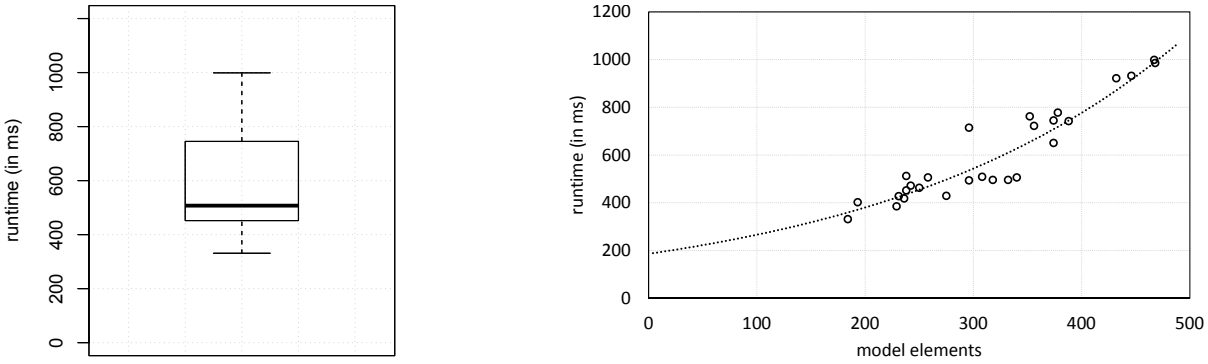
4.2 Results & Discussion

Next, we report the results of our evaluation regarding different criteria (i.e., correctness, runtime, and scalability) and discuss them to answer our research questions.

Correctness After manually analyzing the results generated by our adapted family mining algorithm for the selected 27 combinations, we identified that 20 results were 100% correct with regard to the variability modeled in the BCS SPL. The remaining 7 results were not 100% correct as the algorithms identified certain regions as alternatives although they were modeled as optional features. For example, the region `EM_heating`, which models the heating functionality for the exterior mirror, was matched in certain cases with the region `RCK_SF`, which models the remote control key safety function. In Section 3.3.3, we already discussed such scenarios and presented a possible solution. After realizing the proposed solution (i.e., splitting alternative compare elements for regions into optional parts when the regions' names differ too much), we were able to solve the incorrect matchings and, thus, identified the results for all selected 27 combinations to correspond with the variability in the BCS SPL.

Runtime During the evaluation, we measured the runtime of our adapted family mining algorithms. In Figure 6a, we present a box plot of the runtime for all 27 combinations. For the executed cases, the runtime ranges from 331 ms (184 compared model elements) to 999 ms (467 compared model elements). Consequently, the algorithms are able to correctly identify the variability for the selected cases in a reasonable time outperforming any manual analysis of the same models.

Scalability To evaluate the scalability of our adapted family mining algorithms, we set the runtime for each combination in relation to the corresponding number of compared



(a) Box plot for the runtime

(b) Runtime in relation to number of model elements

Figure 6: Graphical evaluation of the results

model elements (i.e., the states, regions, and transitions that were compared). In Figure 6b, we present the resulting scatter plot together with a trend line to visualize the runtime increase with a growing number of compared model elements. We can see that the data points are scattered more or less evenly around the trend line and 80% of all markers are in an interval of ± 100 ms around the trend line. Overall, the created trend line gives the impression of an exponential trend. However, the considered window with only up to 468 compared model elements is too small to give a precise answer and we will further investigate the scalability with larger case studies in future work. Nevertheless, we argue that the identified runtime is acceptable since it is below one second. Furthermore, we argue that even runtimes in the range of hours should be acceptable when using the approach to transition a large set of model variants to an SPL as this step is only executed once during SPL creation.

Manually analyzing the generated variability information from the comparisons during our evaluation, we were able to show that the automatically generated results by our family mining algorithms are correct with regard to the variability modeled in the BCS SPL. Thus, the variability information is identified correctly and we can give a positive answer to *RQ1*. In addition, we identified that our algorithms seem to follow an exponential trend for models with a growing number of model elements. However, the analyzed subjects have not enough elements to give a precise answer for *RQ2* and we will further investigate the scalability in future work.

4.3 Threats to Validity

During the case study, a single developer manually evaluated the results of our family mining algorithms for state charts and compared them to manually created variability information. Other researchers or developers might question the correctness of the identified variability as their intuition might differ. Furthermore, our approach for family mining of state charts is realized using metrics to compare related state charts. Metrics use heuristic weights and settings, which are highly dependent on human intuition and experience of the implementing developer. Consequently, metrics might not always conform with the results expected by other developers. However, the developer analyzing the results and implementing the metrics has several years of experience with SPLs and already conducted research on mining variability [38–40]. In addition, we have created our metrics with caution and only after carefully analyzing the notations of state charts. Consequently, the results should be at least close to the intuition of domain experts.

Our case study is limited to the state charts of a single case study and, thus, we show that the family mining algorithms and the corresponding implementation support these particular models. However, as this case study contains a large set of common language elements for state charts, our algorithms should also be capable of handling models from other case studies using the same elements. Despite the large set of commonly used language elements, the BCS case study does not use all language elements common to state charts (e.g., final states or state actions). As a consequence, we were not able to evaluate our algorithms for these missing elements. However, as such elements can be compared similar to other elements, we implemented our algorithms allowing to compare them correspondingly. Thus, our algorithms should be capable of handling these elements.

Although using models from an SPL to evaluate a variability mining approach can be seen as a threat to validity, we think it shows that our approach can identify variability conforming with SPL best practices. Besides, this provides us with a ground truth of the contained variability and we do not have to manually identify variability information that might be raised to question. In addition, the BCS SPL also contains minor differences between the implementations of features (e.g., to realize proper interaction with additionally selected features). Thus, our approach shows that it is capable of identifying differences that might be regarded as accidental differences introduced in clone-and-own scenarios.

The sizes of the state charts contained in the BCS SPL case study are limited and the contained elements range from 91 to 283. Consequently, we can only have a limited prediction about the scalability of our approach for state charts

with increasing elements. Thus, we will further investigate the scalability with larger case studies in future work.

5. Related Work

Identifying variability information from related product variants has been extensively investigated for different languages [32]. Algorithms exist to detect cloned (i.e., common) parts of related models. Using graph-based algorithms, it is possible to identify syntactic clones [8, 17, 23], semantic clones [1], and clones with minor changes (i.e., near-miss clones) [23]. Other algorithms are able to identify cloned model parts by analyzing textual representations of models [2]. Alalfi et al. cluster identified clones [3] and infer that remaining non-cloned parts can be regarded as variable parts [4]. However, as their approach is limited to *MATLAB/Simulink* models, it is not applicable for state charts.

Identifying only cloned parts in models is not sufficient to identify the complete variability information as their differences also have to be considered. Many tools identifying the differences between models are integrated in commercial tools (e.g., *SimDiff*[7]). In addition, further differentiation algorithms exist in literature [7, 14, 35, 41]. However, lacking the information on common parts these algorithms do not identify complete variability information.

After comparing related state charts, merging algorithms are needed to create a merged view on the identified variability. Different algorithms exist to merge models in different contexts [5, 20, 30, 33, 37]. Neglecting information on the variability between merged models, these algorithms are not directly applicable for our purposes. In addition, algorithms exist to merge models and annotate the source of merged elements [12, 24, 26] or to visualize variability between models [19]. Lacking fine-grained variability information (i.e., whether elements are mandatory, alternative, or optional), these approaches are not suitable for our goals.

The approach by Nejati et al. is fairly similar to our approach and also uses heuristics (e.g., metrics) and similar algorithms to compare and match elements from compared state charts [21, 22]. However, their approach does not merge fine grained variability information (e.g., which elements are alternatives to each other) into the created model and only annotates the models containing the element. This limits the use of the identified information to generating the compared variants. In contrast, we allow developers a detailed and fine grained analysis of the variability between compared variants. While the variability identification approach for *MATLAB/Simulink* models by Ryssel et al. is similar to ours, the authors do not aim at storing the information in form of 150% models [29]. Their goal is to store the identified information in form of library elements [27]. Font et al. concentrate in their work on identifying variability information in families of models and storing it by means of the *Common Variability Language (CVL)* [11]. In similar work, Font et al. incorporate the developer's domain knowledge to create model patterns allowing to identify variable model parts, which represent the domain experts expectations [10]. While these approaches store the identified variability using the CVL, we concentrate on creating 150% models explicitly modeling all variability information (e.g., the artifacts' source models). Using *abstract syntax trees (AST)*, Klatt et al. identify the variability between related source code artifacts [15]. Although, the approach by Klatt et al. and our family mining approach both operate on a graph-based representation of the analyzed languages, AST-based algorithms are limited to the underlying data-structure. In contrast, we showed the applicability of our approach to different block-based languages [40].

While the most part of variability mining approaches in literature uses pairwise approaches, Rubin et al. describe an n-way algorithm for merging a potentially arbitrary number of *UML* model variants at the same time [25]. While we showed that our family mining algorithm can be successfully adapted to different languages (i.e., *MATLAB/Simulink* and state charts), such an evaluation still has to be executed for the n-way approach by Rubin et al.

In addition, different approaches exist to identify variability on a much higher level in form of feature models or CVL models from differing input artifacts. For example, approaches exist to create such models from natural-language requirements [36], product maps [28, 31], or existing products [42]. While these approaches focus on high level descriptions, our family mining analyzes concrete implementations to create 150% models.

6. Conclusion and Future Work

In this paper, we successfully presented the adaption of our existing variability mining algorithms [38] for state charts using the guidelines in [40]. The identified variability information allows managed reuse of existing functionality without abolishing variant creation using clone-and-own approaches. By means of a case study, we showed that the identified variability for related state charts corresponds to variability modeled in software product lines. In future work, we plan to realize additional algorithms to detect insertions of states between existing states and hierarchy shifts of coherent functionality (i.e., elements encapsulated in hierarchical states to have a higher degree of abstraction). Furthermore, we plan to involve the developers domain knowledge to reduce the number of executed comparisons by comparing only regions that are related according to them. In the course of these improvements, we plan to further evaluate the scalability of our approach with larger case studies.

Acknowledgments

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

[7] `http://www.ensoftcorp.com/simdiff/`

References

[1] B. Al-Batran, B. Schätz, and B. Hummel. Semantic Clone Detection for Model-Based Development of Embedded Systems. In *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 6981 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 2011.

[2] M. Alalfi, J. Cordy, T. Dean, M. Stephan, and A. Stevenson. Models are code too: Near-miss clone detection for Simulink models. In *Proc. of the Intl. Conference on Software Maintenance (ICSM)*, pages 295–304. IEEE, 2012.

[3] M. Alalfi, J. Cordy, and T. Dean. Analysis and Clustering of Model Clones: An Automotive Industrial Experience. In *Proc. of the Intl. Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, CSMR-WCRE '14, pages 375–378. IEEE, 2014.

[4] M. Alalfi, E. Rapos, A. Stevenson, M. Stephan, T. Dean, and J. Cordy. Semi-automatic Identification and Representation of Subsystem Variability in Simulink Models. In *Proc. of the Intl. Conference on Software Maintenance and Evolution (ICSME)*, ICSME '14, pages 486–490. IEEE, 2014.

[5] M. Alanen and I. Porres. Difference and Union of Models. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863, pages 2–17. Springer, 2003.

[6] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer. Flexible Product Line Engineering with a Virtual Platform. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*, pages 532–535. ACM, 2014.

[7] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proc. of the Intl. Conference on Management of Data (MOD)*, SIGMOD '96, pages 493–504. ACM, 1996.

[8] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone Detection in Automotive Model-based Development. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*, pages 603–612. ACM, 2008.

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

[10] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. Building Software Product Lines from Conceptualized Model Patterns. In *Proc. of the Intl. Software Product Line Conference (SPLC)*, pages 46–55. ACM, 2015.

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

[12] H. Frank and J. Eder. Towards an Automatic Integration of Statecharts. In *Proc. of the Intl. Conference on Conceptual Modeling (ER)*, volume 1728, pages 430–445. Springer, 1999.

[13] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[14] U. Kelter, J. Wehren, and J. Niere. A Generic Difference Algorithm for UML Models. *Software Engineering*, 64(105-116):4–9, 2005.

[15] B. Klatt, M. Küster, and K. Krogmann. A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies. In *Proc. of the Intl. Workshop on Reverse Variability Engineering (REVE)*, pages 1–8. ACM, 2013.

[16] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10 (8):707–710, 1966.

[17] Z. Liang, Y. Cheng, and J. Chen. A Novel Optimized Path-Based Algorithm for Model Clone Detection. *Journal of Software*, 9(7):1810–1817, 2014.

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

[19] J. Martinez, T. Ziadi, J. Klein, and Y. le Traon. Identifying and Visualising Commonality and Variability in Model Variants. In *Proc. of the European Conference on Modeling Foundations and Applications (ECMFA)*, volume 8569 of *Lecture Notes in Computer Science*, pages 117–131. Springer, 2014.

[20] A. Mehra, J. Grundy, and J. Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*, pages 204–213. ACM, 2005.

[21] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*, pages 54–64. IEEE, 2007.

[22] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and Merging of Variant Feature Specifications. *IEEE Transactions on Software Engineering (TSE)*, 38 (6):1355–1375, 2012.

[23] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and Accurate Clone Detection in Graph-based Models. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*, pages 276–286. IEEE, 2009.

[24] J. Rubin and M. Chechik. Combining Related Products into Product Lines. In *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 7212, pages 285–300. Springer, 2012.

[25] J. Rubin and M. Chechik. *Domain Engineering: Product Lines, Languages, and Conceptual Models*, chapter A Survey of Feature Location Techniques, pages 29–58. Springer, 2013.

[26] J. Rubin and M. Chechik. N-way Model Merging. In *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 301–311. ACM, 2013.

[27] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic Variation-point Identification in Function-block-based Models. In *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–32. ACM, 2010.

[28] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Extraction of Feature Models from Formal Contexts. In *Proc. of the Intl. Software Product Line Conference (SPLC)*, pages 4:1–4:8. ACM, 2011.

[29] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic library migration for the generation of hardware-in-the-loop models. *Science of Computer Programming*, 77(2):83–95, 2012.

[30] M. Sabetzadeh and S. Easterbrook. Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach. In *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*, pages 12–21. IEEE, 2003.

[31] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse Engineering Feature Models. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*, pages 461–470. IEEE, 2011.

[32] M. Stephan and J. R. Cordy. A Survey of Methods and Applications of Model Comparison. Technical Report 582, School of Computing, Queen's University, Kingston, Ontario, Canada, 2011.

[33] S. Uchitel and M. Chechik. Merging Partial Behavioural Models. In *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*, pages 43–52. ACM, 2004.

[34] M. von der Beeck. A Comparison of Statecharts Variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863, pages 128–148. Springer, 1994.

[35] S. Wenzel and U. Kelter. Model-Driven Design Pattern Detection Using Difference Calculation. In *Workshop on Design Patterns Detection for Reverse Engineering (DPD4RE)*. IEEE, 2006.

[36] N. Weston, R. Chitchyan, and A. Rashid. A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements. In *Proc. of the Intl. Software Product Line Conference (SPLC)*, pages 211–220. ACM, 2009.

[37] J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*, pages 314–323. ACM, 2000.

[38] D. Wille. Managing Lots of Models: The FaMine Approach. In *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*, pages 817–819. ACM, 2014.

[39] D. Wille, S. Holthusen, S. Schulze, and I. Schaefer. Interface Variability in Family Model Mining. In *Proc. of the Intl. Workshop on Model-Driven Approaches in Software Product Line Engineering (MAPLE)*, pages 44–51. ACM, 2013.

[40] D. Wille, S. Schulze, C. Seidl, and I. Schaefer. Custom-Tailored Variability Mining for Block-Based Languages. In *Proc. of the Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 271–282. IEEE, 2016.

[41] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*, ASE '05, pages 54–65. ACM, 2005.

[42] X. Zhang, Ø. Haugen, and B. Møller-Pedersen. Model Comparison to Synthesize a Model-Driven Software Product Line. In *Proc. of the Intl. Software Product Line Conference (SPLC)*, pages 90–99. IEEE, 2011.