Variability Representations in Class Models: An Empirical Assessment

Daniel Strüber
Radboud University Nijmegen
d.strueber@cs.ru.nl

Anthony Anjorin
IAV Automotive Engineering
anthony.anjorin@iav.de

Thorsten Berger
Chalmers | University of Gothenburg
thorsten.berger@chalmers.se

ABSTRACT

Owing to the ever-growing need for customization, software systems often exist in many different variants. To avoid the need to maintain many different copies of the same model, developers of modeling languages and tools have recently started to provide representations for such variant-rich systems, notably *variability mechanisms* that support the implementation of differences between model variants. Available mechanisms either follow the *annotative* or the *compositional* paradigm, each of them having unique benefits and drawbacks. Language and tool designers select the used variability mechanism often solely based on intuition. A better empirical understanding of the comprehension of variability mechanisms would help them in improving support for effective modeling.

In this paper, we present an empirical assessment of annotative and compositional variability mechanisms for class models. We report and discuss findings from an experiment with 73 participants, in which we studied the impact of two selected variability mechanisms during model comprehension tasks. We find that, compared to the baseline of listing all model variants separately, the annotative technique did not affect developer performance. Use of the compositional mechanism correlated with impaired performance. For two out of three considered tasks, the annotative mechanism is preferred to the compositional one and the baseline. We present actionable recommendations concerning support of flexible, tasks-specific solutions, and the transfer of established best practices from the code domain to models.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Software product lines**.

KEYWORDS

variability mechanisms, model-based software product lines

ACM Reference Format:
Daniel Strüber, Anthony Anjorin, and Thorsten Berger. 2020. Variability Representations in Class Models: An Empirical Assessment. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Montreal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3365438.3410935

MODELS '20, October 18–23, 2020, Montreal, QC, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410935

1 INTRODUCTION

Variant-rich systems can offer companies major strategic advantages, such as the ability to deliver tailor-made software products to their customers. Still, when developing a variant-rich system, severe challenges may arise during maintenance, evolution, and analysis, especially when variants are developed in the naive *clone-and-own* approach, that is, by copying and modifying them [62]. The typical solution to these challenges is to manage variability by using dedicated *variability representations*, capturing the differences between the variants [81]. An important type of variability representation are *variability mechanisms*, which are used to avoid duplication and to promote reuse when implementing variability in assets such as code, models, and requirements documents. Over more than three decades, researchers have developed a plethora of variability mechanisms, albeit mostly for source code [8, 15, 82].

As companies begin to streamline their development workflows for building variant-rich systems, they recognize a need for variability management in all key development artifacts, including models. The use of models is manifold, ranging from sketches of the system design, to system blueprints used for verification and code generation. The car industry is particularly outspoken on their need for model-level variability mechanisms. For example, General Motors named support for variation in UML models as a major requirement [37], and Volkswagen reports large numbers of complex, cloned variants of Simulink models in their projects [67]. Beyond automotive, the need for model-level variability has been documented for power electronics, aerospace, railway technology, traffic control, imaging, and chip modeling [16].

Recognizing this need, researchers have started building variability mechanisms for models. Variability mechanisms are now available both for UML [9, 25, 69] and DSMLs [7, 42, 77]. Building on these results, researchers have started to address advanced problems such as the migration of a set of "cloned-and-owned" model variants to a given mechanism [11, 54, 64, 67, 88], and efficient analysis of large sets of model variants [22, 28, 59]. Adoption in several industrial DSMLs has demonstrated the general feasibility of model-level variability mechanisms in practice [80].

While variability mechanisms for source code are reasonably well understood [15, 55, 82], language and tool designers are offered little guidance on selecting the most effective variability mechanism for their purposes. In fact, there is a lack of evidence to support the preference of one mechanism over the other. In line with previous studies on code-level mechanisms [35, 36, 49, 55], we argue that *comprehensibility* is a decisive factor for the efficiency of a variability mechanism—for any maintenance and evolution activity (e.g. bugfixing, feature implementations), the developers first need to understand the existing system. A better empirical understanding

of the comprehension of variability mechanisms could support the development of more effective modeling languages and tools.

To this end, we present an empirical study of variability representations in models. In a fully randomized experiment performed with 73 participants with relevant background, we studied how the choice of variability mechanism affects performance during model comprehension tasks. We consider two selected variability mechanisms that are representative for the two main types distinguished in the literature [47]: *Annotative* mechanisms maintain an integrated, annotated representation of all variants. Examples include preprocessor macros [72] (for code) and model templates [25] (for models). Annotative mechanisms are conceptually simple, but can impair understandability since they clutter model or code elements with variability information [55, 72]. *Compositional* mechanisms allow to compose a set of smaller sub-models to form a larger model. Examples include *feature-oriented programming* [10] (for code) and *model refinement* [34] (for models). Compositional mechanisms are appealing as they establish a clear separation of concerns, but they involve a composition step which might be cognitively challenging. We aimed to shed light on the impact of these inherent trade-offs.

As a baseline for comparison, we consider a third solution of listing model variants individually, which we call the "enumerative mechanism." As a particularly simple solution, this supports a richer comparison of the considered mechanisms. Despite its use in practice [80], this solution is problematic as a standalone representation. We discuss the implications of our baseline solution in Sect. 5.4.

Our focus is on class models, a ubiquitous model type: Class models are the most frequently used part of UML [52, 60], due to their role in system design and analysis. In code generation contexts, they are used to generate data management components (e.g., large parts of enterprise web and mobile apps can be generated from class models [50, 57, 84]), object-oriented code in roundtrip engineering scenarios [17], and ample MDE tooling in modeling platforms, such as EMF [74]. In aeronautics, class models are used in AIXM, a massively used information exchange format [32]. Furthermore, they are representative for a wide array of visual languages based on the graph paradigm, such as ER diagrams and DFDs.

We make the following contributions:

- We present a quantitative analysis of correctness, the completion time, and subjective assessments of our participants for six model comprehension tasks.
- We present a qualitative analysis of participant responses, adding rationale to explain the observed results.
- Based on our synthesized findings, we propose recommendations for language and tools developers.
- A replication package [12] that includes our experimental material, anonymized responses, and analysis scripts.

We present the first study on variability mechanisms for models. While earlier studies have investigated the comprehensibility of code variability mechanisms (see Sect. 7), their generalizability to models is unclear. Code usually has a tree-like structure and is expressed in textual notations. Modeling languages support the structuring of models in a graph-like manner and usually have graphical notations. Since different representations are known to affect performance during decision-making tasks [85], specifically, software engineering tasks [3, 51], we argue that the comprehensibility of model variability mechanisms requires a dedicated investigation.

2 BACKGROUND

There has been a recent surge of interest in dedicated variability mechanisms for models. Lifting the related distinction from code-level mechanisms, two main types are distinguished: Annotative mechanisms represent variability with an annotated integrated representation of all variants. Mechanism in this category are *model templates* [25, 42, 79] and *union models* [5]. Compositional mechanisms represent variability by composing variants from smaller sub-models. Available approaches mostly differ in their model fragment syntax and composition strategy. Examples are *delta modeling* [21], *model superimposition* [9], and *model refinement* [34].

To illustrate the role of both types of mechanisms in industry, we refer to a recent survey of variability support in 23 DSMLs [80]. They describe four strategies being used: First, a model represents one variant (9 languages); second, elements are reused across models by referencing (10 languages); third, multi-level modeling is used for capturing variability (1 language); fourth, elements have so-called presence conditions (explained shortly, 3 languages). The first strategy is considered as a baseline in our experiments. The second and third one are compositional, as they spread differences between variants across several smaller models. The fourth one is annotative.

We selected the two variability mechanisms for our experiment based on the following criteria: (M1) *The mechanism has a graphical syntax.* (M2) *The mechanism is supported by available tools.* (M3) *The mechanism has been described in scientific literature.* The rationale for M1 was to study variability mechanisms in the widespread graphical representation of class diagrams. Support by available literature (M2) and tools (M3) may contribute to the transfer of existing research results to industrial practice, and allows practitioners to test the mechanisms in an available prototype.

Based on these criteria, as annotative mechanism, we identified *model templates* (implemented in FeatureMapper [42], SuperMod [69], and Henshin [77]). For compositional, we identified two existing approaches fulfilling the criteria: Delta modeling (implemented by DeltaEcore [70] and SiPL [61]) and model refinements (implemented by eMoflon [7]). We decided to consider model refinements, as they implement the compositional paradigm in the most straightforward way (delta modeling supports deletions, which increases its expressiveness, but requires a more complex syntax and semantics).

Example. We illustrate the specific variability mechanisms used in our experiments with a simple example, inspired by Schaefer [66]. The same example was also used in the experiment to introduce the variability mechanisms to the participants.

The example represents a simple cash desk system that exists in three similar, but different variants. Figure 1 depicts the individual variants using separate class diagrams: Variant `var1` consists of a `CashDesk` with a `KeyBoard` and a `Display`. Variant `var2` has additionally exactly one `CardReader` connected to the `CashDesk`. Variant `var3` replaces the `Keyboard` with a `Scanner` and makes the `CardReader` optional (multiplicity `0..1` instead of 1).

The depicted representation of listing variants individually is used as a baseline in our experiments, referred to as the "enumerative mechanism." This solution is frequently applied in practice

[80], where it leads to severe maintenance drawbacks. For example, a bug found in one of the variants must be fixed in all variants separately. The goal of the variability mechanisms presented below is to simplify working with such similar, but distinct variants.

Annotative. The annotative mechanism considered in our experiments is *model templates* [25]. Like annotative mechanisms in general, it combines all variants into a single representation with annotations. The left-hand side of Fig. 2 shows a model template for our example: a class diagram that represents the three variants of the cash desk system. Parts of the class diagram are annotated with *presence conditions*, stating the variants in which the part occurs. In general, a presence condition is a list of configuration options (disjunction). For example, the presence condition «`var1,var2`» indicates that the annotated part is present when either the configuration option `var1` or `var2` is selected. The absence of a presence condition denotes that the part is contained in all variants.

Colors are used in the following way: Elements (classes and associations) with a black outline occur in all variants, elements with a grey outline occur in two or more variants, elements with a colored outline belong to precisely one variant, whose annotation is also depicted with the same color. The use of colors to distinguish elements goes back to the original paper that introduced model templates [25]. Colors may be crucial for comprehensibility. In the case of code-level variability mechanisms, Siegmund et al. [35] found that colors support understanding of annotative variability. We are interested to study if this finding also applies to models.

Individual variants are obtained from the combined representation as follows: The user sets one of the configuration options as active. The corresponding class model is produced by removing all those elements whose presence condition does not contain the configuration option. For example, selecting the configuration option `var1` leads to the model variant `var1` in Fig. 1.

Compositional. The compositional mechanism we considered is *refinement* [7]. Like all compositional variants, refinement provides (i) a means of decomposing variants into smaller building blocks, and (ii) a means of *merging* building blocks to form complete variants. This allows for a sharing and reuse of common parts in different variants. The building blocks are visually shown as a network, as depicted in the right-hand side of Fig. 2. Commonalities of `var1` and `var3`, as well as `var2` and `var3` have been extracted into separate "super" class diagrams. These diagrams have a dashed border as they only represent commonalities and are "abstract" in the sense that they are not complete variants. Composition of diagrams is denoted using an inheritance arrow, e.g., `var2` is formed by combining `var1`, the elements specified in `var2`, and the elements in the common super class of `var3` and `var2`. As the example demonstrates, multiple super class diagrams (see `var3`), and transitive composition (see `var2`) are possible.

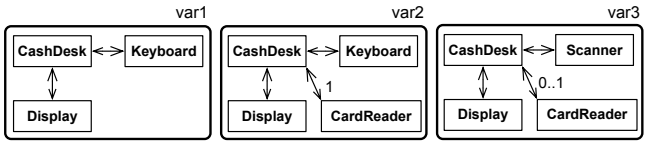

Figure 1: Three variants of a cash desk system

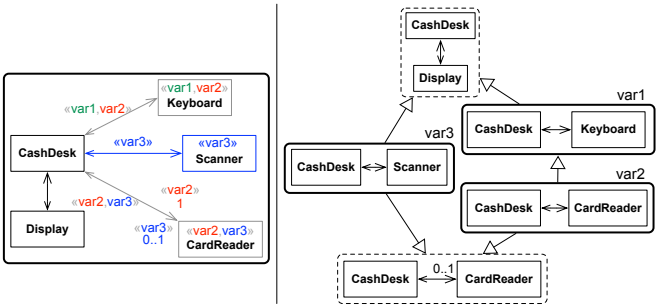

Figure 2: Annotative and compositional variability

Deriving individual variants is a two-step process. First, a *union* of the contents of the variant and all its transitive parents is computed; this results in a single, flat class diagram (with no parents). Second, a *merge* operator is used to combine elements that should be the same. For class diagrams, this operator combines all elements with the same name. The merge operator also defines how to resolve conflicts: for class diagrams, a common subtype must exist for nodes to be merged, and multiplicities of merged edges are combined by taking the maximum of lower bounds and minimum of upper bounds. For example, when the variant `var1` is selected, it is merged with its parent (top class diagram with dashed lines). Building the union of both class diagrams and merging the cash desk elements leads to the model variant `var1` in Fig. 1.

3 PREPARATORY STUDY

As preparatory study, we conducted an experiment to shape the design of our materials and tasks. The goal was to assess the suitability of our experimental tasks and to derive potential improvements of the setup. The study was performed on a population of 28 students (disjoint from the population of our experiment). The students were familiar with class models, the model type used in the experiment.

The tasks considered were *bug-finding tasks*, a typical task type for assessing the usefulness of visual representations [55, 76]. Participants were handed a textual requirement specification, together with design models implementing the requirements with one of the given variability mechanisms. The design models contained a number of deviations from the textual requirements (bugs), which the participants were asked to identify. We also asked the participants to suggest potential improvements to the experiment using a textual input form.

To obtain meaningful example system, our examples were obtained from the existing literature. The first example represented a phone product line with phones being conditionally capable of making incoming and outgoing calls [9]. The second example represented a project management system with managers, employees, and tasks [33]. Students obtained a virtual instruction sheet and a link to an explanation video for the used variability mechanisms.

From this preliminary study we made three main conclusions: First, example models with 3 to 4 classes, and 3 or 4 variants each are too simple to demonstrate a difference between both mechanisms. This conjuncture is supported by one of the participants' recommendation to *"create [more] complicated examples with 6 or 7 classes and not so easy ones,"* provided via the textual feedback form.

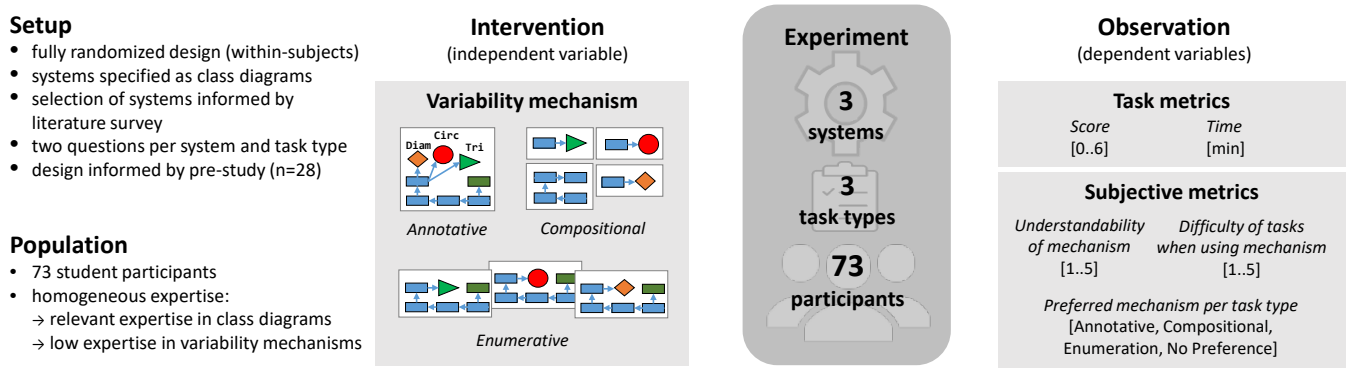

Figure 3: Methodology overview

Second, despite our efforts to provide clear requirements, a participant asked us to be "more specific and less ambiguous with the requirement specifications." Ambiguity is an inherent risk to experimental validity since its effect is hard to quantify (it is unclear how many participants assume a different understanding than intended). Another, recurrent comment was that reading the descriptions was tiring, threatening the completion rate. Therefore, we decided to switch the nature of the used tasks in the main experiment to comprehension tasks that do not rely on additional artifacts.

Third, a provided instruction video was viewed as redundant, as it showed only information that is available on the instruction sheet. In the final experiment, we decided to omit the explanation video.

4 METHODOLOGY

The goal of our experiment was to study the effect of variability representations on model comprehension. Figure 3 provides a high-level overview of our methodology. Using experimental material from three subject systems, we asked 73 participants to perform three kinds of comprehension tasks. We varied the used variability mechanism during the tasks (independent variable) and recorded task metrics, subjective metrics, and quantitative feedback (dependent variables). In this section, we describe the research questions, participants, tasks, metrics, and questions of our experiment. Our experimental materials and result data are publicly available [12].

4.1 Goal and Research Questions

We formulated and investigated the following research questions:

> **RQ1** *To what extent do variability representations impact the efficiency of model comprehension?*

We studied the effect of annotative and compositional representations on the ability to solve model comprehension tasks correctly and quickly.

> **RQ2** *How are variability representations perceived during comprehension tasks?*

We studied the understandability and perceived difficulty to complete model comprehension tasks, based on subjective assessments.

> **RQ3** *Which variability representations do the participants prefer during comprehension tasks?*

We elicited qualitative and quantitative data regarding the participants' subjective preferences, by asking them to name a preferred representation and explain their choices.

4.2 Subject Systems and Material

Our final experiment comprised a number of tasks based on certain subject systems. To select the systems, we specified a set of criteria that a subject system would need to fulfill: (C1) *The system has been introduced in previous literature.* (C2) *The system comprises several variants.* (C3) *The system has not been introduced in a context related to a particular variability mechanism.* The rationale of these criteria was to select systems that represent real variability, rather than making up artificial examples on the spot. Moreover, we wanted to avoid bias in favor of one of the considered variability mechanism types.

Three subject systems were identified, based on their familiarity to the authors (*convenience sampling* [89]): *Simulink*, *Project Management*, and *Phone*. For the former two systems, we were aware of several available variants in the literature. To systematically identify available variants, we performed database searches in Google Scholar, IEEExplore, and ACM's Digital Library, with the search strings "Project Management meta-model" and "Simulink meta-model". The considered variants of *Phone* correspond to the feature model from the original paper.

Simulink is a block-based modeling language that is widely applied in the design of embedded and cyber-physical systems. The absence of an official specification has given rise to the emergence of various variants. We obtained six publications that included a Simulink meta-model [6, 41, 44, 53, 71, 78]. These meta-models are relevant for our study, since meta-modeling is one of the prime use cases of class models, enabling the development of custom-tailored DSMLs [73] and language families [24].

The *Project Management (PM)* product line represents a family of software systems for project management, with concepts such as projects, activities, tasks, persons, and roles. This example is inspired by the availability of various similar, but different class models for project management software, which we identified in our literature search [18, 23, 31, 38, 39, 43].

The *Phone* product line, introduced by Benavides et al. [14], represents a family of software systems for mobiles phones with various hardware functionalities, such as different cameras and displays. As

an example that was originally introduced for variability analysis in software systems, it is suitable for studying variability mechanisms.

We developed three class model representations for each system: an enumerative, an annotative, and a compositional one (cf. Sect. 2). For each system, the starting point was the enumerative representation, consisting of several class model variants. The other representations were derived manually as follows: the annotative one by following Fahrenberg et al.'s merge procedure [33]; the compositional one by identifying reusable *features* (e.g., *keyboard*) and encapsulating them as fragments (e.g., *var1* in the example). Since this strategy relies on subjective decisions (of what to consider as a feature), a separate expert researcher was consulted for quality assurance. Two of the authors verified the correctness of all produced models by manually checking that "flattening" the produced annotative and compositional solutions leads the enumerative solutions again. The enumerative version of the *Simulink* scenario contains eight classes and six variants in total. The *PM* scenario includes nine classes and five variants. The *Phone* scenario comprises 14 classes and six variants in total. The given numbers of classes refer to the total number of all classes as shown the annotative representation; each individual variant contains a smaller number of classes.

4.3 Experimental Design

We applied a *cross-over trial*, a variant of the within-subject design [45], in which all participants are sequentially exposed to each treatment. The treatments in our case are the use of an annotative, compositional, or enumerative representation during comprehension tasks. Annotative and compositional are the variability mechanisms under study, while the enumerative representation acts as a baseline, representing the case in which no dedicated variability mechanism is available. The main benefit of the chosen design is its efficiency in supporting statistically valid conclusions for a given number of participants. The design also reduces the influence of confounding factors, such as participant expertise, because each participant serves as their own control.

A main threat to this kind of study design are learning effects: during the experiments, participants might transfer experience gained by solving one task to other tasks. We mitigated this threat by using the *latin square design* [55, 56]. All participants were randomly distributed across three equally sized groups, of which each followed one of three paths through the questionnaire:

- *Enumerative*→*Annotative*→*Compositional* (path 1),
- *Compositional*→*Enumerative*→*Annotative* (path 2), and
- *Annotative*→*Compositional*→*Enumerative* (path 3).

Following the latin square design, to avoid bias related to the complexity of the considered systems, the order of systems was fixed between paths: *Phone*→*PM*→*Simulink* (see Sect. 3). We discuss further threats and mitigation strategies in Sect. 6.

4.4 Tasks Types and Questions

The design of our experimental tasks was informed by the experiences from our pre-study (see Sect. 3). We used comprehension tasks, in which the participants were asked to answer comprehension questions regarding provided models (equipped with variability mechanisms). Models were presented on paper printouts, which is less realistic than presenting them in a tool, but avoids many confounding factors (e.g., different screen sizes, maturity issues of tools). We chose three task types to capture whether the participants understood the variability present in the systems based on the used variability mechanisms. For quality assurance (i.e., ensuring that the provided instructions and examples are clear), we performed a trial run with 3 participants. Below, we explain the task types, and argue for their representativeness during realistic comprehension scenarios.

Task type 1: *trace elements to variants*. The participants were asked to identify variants that include specified classes and relationships. Such a task is representative of the feature location problem, in which parts of a given code base or model implementing a specified feature are to be identified. In absence of fully reliable automated techniques, feature location is often performed manually [30]. Participants were asked the following two questions for the *Simulink* system, and similar question pairs for the other two systems:

> (1) How many variants have both the classes "InPort" and "OutPort"?
> (2) Are there any variants that do not have the class "Port"? If yes, which variants?

Task type 2: *compare two variants*. The participants were asked to compare two variants, either by naming their differences or by comparing them in terms of a specified quantity, such as number of classes. Such a task is performed when understanding the nuances of how two closely related variants differ. Participants were asked the following two questions for the *PM* system, and similar questions for the other two systems:

> (3) How do variants var1 and var2 differ?
> (4) Which of the two variants var3 and var5 has more associations?

Task type 3: *compare all variants*. The participants were asked to identify model elements that appear in a maximal or a minimal subset of all variants. Such a task is typically performed when trying to understand a full variant space. Participants were asked the following two questions for all systems:

> (5) Which class is required by all variants? List all such classes if there are more than one.
> (6) Which class is required by only one variant? List all such classes if there are more than one.

Task metrics: To address RQ1, we collected two *tasks metrics*: *correctness score* and *completion time*. The correctness score of a particular answer is either 1 (correct), 0.5 (partially correct), or 0 (incorrect), as obtained by a manual assessment of the answer. Answers were deemed as partially correct if they contained some correct aspects, but not all required aspects or some incorrect ones. For example, considering question 3, if the variants differ in two classes, and the answer consists of one of them, the answer is partially correct. The assessment was checked and agreed on by two authors. For each of the three treatments, we elicited the total *completion time* (in minutes) for completing all tasks. The elicitation was performed by asking the participants to enter the current time at the end of each page in the questionnaire.

Subjective assessment: To address RQ2 and RQ3, we collected three *subjective metrics* and additional textual feedback at the end of the experiment. For RQ2, we asked the participants to assess the

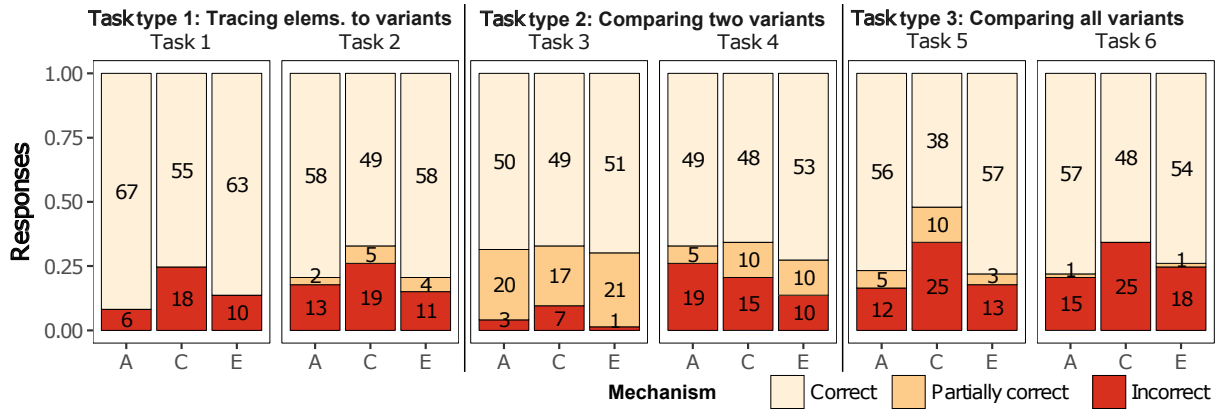

Figure 4: Correctness scores.

Table 1: Correctness scores of our participants. Scores are between 0 and 2 per task type.

	Annotative			Compositional			Enumerative		
Task type	**Mean**	**Median**	**Sd.dev**	**Mean**	**Median**	**Sd.dev**	**Mean**	**Median**	**Sd.dev**
1: Tracing elements to variants	1.7/2	2.0/2	0.6	1.5/2	2.0/2	0.7	1.7/2	2.0/2	0.6
2: Comparing two variants	1.5/2	1.5/2	0.6	1.5/2	1.5/2	0.6	1.6/2	1.5/2	0.4
3: Comparing all variants	1.6/2	2.0/2	0.7	1.2/2	1.0/2	0.7	1.5/2	2.0/2	0.7
Total	4.8/6	5.0/6	1.3	4.2/6	4.5/6	1.4	4.9/6	5.0/6	1.1

understandability of each mechanism and the *difficulty* of addressing each task type using each mechanism. For RQ3, we asked them to specify a *preferred mechanism* per task type. To complement this quantitative information with qualitative data, we also asked our participants to elaborate by asking them to explain their choice of preferred mechanism. We used the following questions:

(S1) How easy did you find it to understand each mechanism?
(S2) How difficult was it to answer the questions on "Finding classes and relationships in variants" (Questions 1 and 2) for each mechanism?
(S3) How difficult was it to answer the questions on "Comparing two variants" (Questions 3 and 4) for each mechanism?
(S4) How difficult was it to answer the questions on "Comparing all variants" (Questions 5 and 6) for each mechanism?
(S5) Which mechanism do you prefer for each of the three task types?
(S6) Can you explain your subjective preferences (intuitively)?

Following the common practice for subjective responses, we captured the answers to S1–4 on one five-point Likert scale for each mechanism. The answer to S5 was specified by selecting one of the literals Annotative, Compositional, Enumerative, None for each of the task types. To collect qualitative data in S6, we asked the participants to enter their answer into a free-form text field.

For hypothesis testing, we used the Wilcoxon signed-rank test [87] which we applied to the task and subjective metrics, following recommendations according to which this test can be applied to Likert-type data [29]. We used the standard significance threshold of 0.05. Two measurements involve multiple comparisons (correctness, difficulty; each for 3 different task types). For these metrics, we apply the Bonferroni correction [1], yielding a corrected significance threshold of 0.017, obtained by dividing 0.05 by 3. We assessed effect size using the A_{12} score, following Vargha and Delaney's original interpretation [83]: $A_{12} \approx 0.56$ = *small*; $A_{12} \approx 0.64$ = *medium*; and $A_{12} \approx 0.71$ = *large*. All tests were executed with R.

4.5 Participants

We performed the experiment with 73 participants. The participants were recruited from four undergraduate and graduate courses at German universities. Our rationale for recruiting students is their suitability as stand-ins for practitioners: students can perform involving unfamiliar software engineering tools equally well as practitioners [65]. The baseline modeling technology in our experiments is class models. The students were, therefore, recruited from courses with completed previous lectures and homework assignments on class models. Before the experiment, it was pointed out that participation in the experiment was entirely voluntary, and data would be stored anonymously. To encourage participation, a gift card raffle was offered as a prize to interested participants.

We asked the students to self-assess their expertise in three relevant categories using five-point Likert scales: class diagrams (the baseline technology of our experiments), programming (to argue for the representativeness of our findings), and the considered variability mechanisms (the experimental treatment). In line with our strategy to recruit students familiar with class models, students expressed an average level of expertise, amounting to 3.47 (mean) ± 0.60 (standard deviation). The self-reported programming expertise of 3.62 ± 0.74 was comparable, justifying generalizations of our findings to programmers with average experience. In contrast, the self-reported expertise in variability mechanisms was considerably lower, amounting to 1.73 ±0.87 in annotative mechanisms, 1.86 ±1.03 in compositional mechanisms, and 1.87 ±0.93 in enumerative

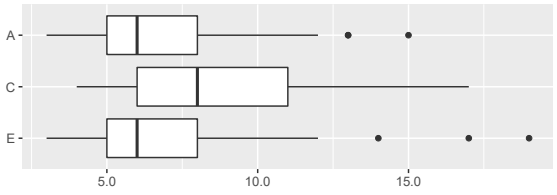

Figure 5: Completion times (in minutes).

mechanisms. The homogenous experience in the considered mechanisms is beneficial for the validity of our findings, by countering a possible threat related to different previous knowledge.

5 RESULTS

We now present the results from our experiment, focusing on the aggregated results over all three systems. We also briefly discuss the results per system, providing more details in our appendix [12].

5.1 RQ1: Efficiency

We determined efficiency by considering the ability of our participants to correctly and rapidly complete the presented tasks. Considering correctness, Table 1 and Fig. 4 summarize the scores of all participants for solving all tasks. The participants generally performed equally well with the annotative and the enumerative mechanisms: on the individual task types, the achieved mean scores amount to 1.7, 1.5, 1.6 for annotative versus 1.7, 1.5, 1.6 for enumerative. In contrast, the use of the compositional mechanism leads to a noticeable drop in mean performance to 1.5, 1.5 and 1.2. Hypothesis testing shows that the difference for compositional to other types is significant for task types 1 and 3. For type 1, we find p=0.01 for the comparison to annotative, with a medium-ranged effect size of A_{12}=0.62 (p=0.02 for the comparison to enumerative, surpassing the corrected threshold). For type 3, we find p<0.01 when comparing compositional to both annotative and enumerative, with medium effect sizes (A_{12}=0.66 and 0.64, respectively). We did not find significant differences between the mechanisms for type 2. Annotative and enumerative do not differ significantly in any considered case. The results per system [12] are generally consistent with the aggregated results; however, the differences are more pronounced in the largest system, Simulink, and smaller in the other systems.

Considering rapidness, Table 2 and Fig. 5 provide an overview of the completion times of all participants for solving all tasks. According to this data, the participants were fastest on average when using the annotative mechanism (mean completion time: 6.6 minutes). Enumerative led to the second-lowest completion times (7.1 minutes). Compositional comes last in this comparison (8.8 minutes). The difference between annotative and enumerative is not significant (p=0.12). In contrast, the differences between compositional and both annotative and enumerative are highly significant with p<0.001. The effect size is large when comparing compositional and annotative (A_{12}=0.71) and medium to large for compositional to enumerative (A_{12}=0.67). The differences of completion times per system [12] are fully consistent with those for the aggregated times.

Table 2: Completion times (in minutes) of our participants.

Mechanism	Min	Mean	Median	Max	Sd.dev
Annotative	3	6.6	6	15	2.6
Compositional	4	8.8	8	17	3.2
Enumerative	3	7.1	6	19	3.1

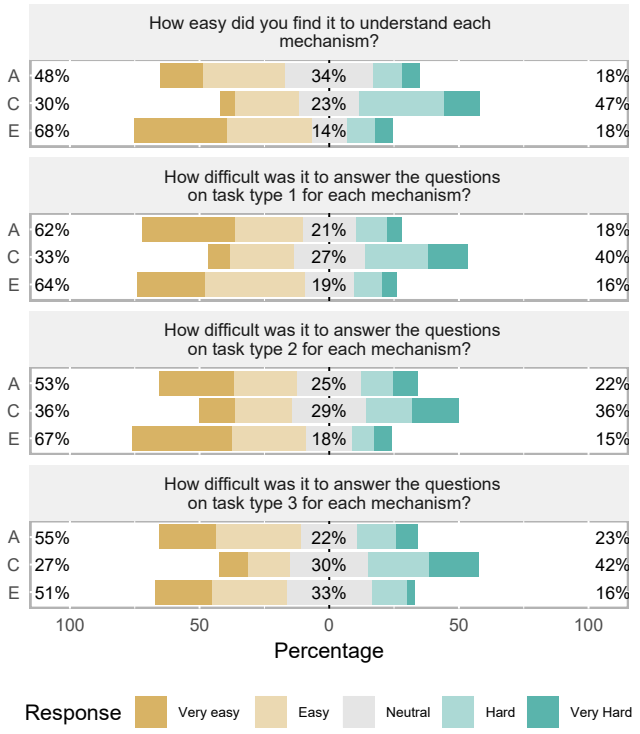

Figure 6: Subjective perceptions.

> Compared to the enumerative mechanism, the participants were equally efficient when using the annotative one. Using the compositional mechanism, they took longer to complete the tasks and made more mistakes, specifically when a good overview of all variants and the ability to trace elements to variants was required.

5.2 RQ2: Subjective Perception

We determined the subjective perception of all variability mechanisms by asking the participants relevant questions on a 5-point Likert scale. Table 3 and Fig. 6 give an overview of the results.

In the understandability rating, with a mean rating of 2.2, enumerative was considered easier than annotative (mean: 2.6), and compositional (mean: 3.2). We observe statistical significance as follows: for annotative vs. enumerative, p=0.006 with A_{12}=0.61 (small to medium effect); for annotative vs. compositional, p=0.004 with A_{12}=0.66 (medium effect); for compositional vs. enumerative, p≤0.001, with A_{12}=0.73 (large effect).

The difficulty rating is fully consistent with the objective task metrics (RQ1). Comparing the annotative and the enumerative mechanism, the given mean ratings are approximately equal, amounting to 2.5, 2.5, 2.5 for annotative, and 2.5, 2.2, 2.5 for enumerative. We do not find statistical significance for this comparison, the lowest observed p-value being for task type 2 (p=0.8). In contrast, the mean ratings for compositional of 3.1, 3.0 and 3.2 are much higher.

Table 3: Participant perception. Scores from a 1..5 Likert scale; lower value indicates better understandability/less difficulty

		Annotative			Compositional			Enumerative		
Quality		**Mean**	**Median**	**Sd.dev**	**Mean**	**Median**	**Sd.dev**	**Mean**	**Median**	**Sd.dev**
Understandability		2.6/5	3/5	1.1	3.2/5	3/5	1.1	2.2/5	2/5	1.2
Difficulty	Task type 1	2.3/5	2/5	1.2	3.1/5	3/5	1.2	2.3/5	2/5	1.1
	Task type 2	2.5/5	2/5	1.3	3.0/5	3/5	1.3	2.2/5	2/5	1.2
	Task type 3	2.5/5	2/5	1.2	3.2/5	3/5	1.3	2.5/5	2/5	1.1

In all comparisons of compositional to another mechanism, we find significance. In all cases but one (task 2, annotative vs. compositional: p=0.03; A_{12}=0.62), the p-value is below 0.003 and the effect size between A_{12}=0.65 and 0.69, indicating a medium to large effect.

The participants found it equally difficult to perform the comprehension tasks with enumerative and annotative. Likewise, both annotative and enumerative outperform compositional. Enumerative was found more understandable than annotative, which itself was found more understandable than compositional.

5.3 RQ3: Preference

To study subjective preference, we collected a combination of quantitative and qualitative data.

Table 4 and Fig. 7 provide an overview of our quantitative data: the percentages of selected answers when asked to specify a preferred variability mechanism per task type. Interestingly, we find that the preferences vary strongly between the tasks. Annotative is preferred by most participants for task types 1 and 3, albeit with only a moderate to slight difference to the enumerative mechanism: 50.7% vs. 34.2% for type 1, and 43.8% vs. 42.5% for type 3. In contrast, the enumerative mechanism is preferred with a large margin for task type 2, comparing two variants (which are explicitly present in the enumerative representation). Compositional comes in last in all comparisons, with percentages between 12.3% and 15.1%. Participants generally expressed a preference; the *no-preference* option was only selected in 1.4% of all cases. Intuitively, the preference for enumerative for type 2 is not surprising: in a comparison between two variants, explicitly representing the variants seems beneficial. Based on the preference of annotative for type 1 and 3, we hypothesize that this representation is suitable for tasks that require a good overview of all variants and the ability to trace elements to variants.

To obtain additional insights, we asked the participants to explain their preferences in a textual form. Based on a manual assessment performed on the answers, we give an overview of recurring aspects deemed as relevant by the participants.

Inherent trade-offs. Our quantitative results indicate that the preferred mechanism highly depends on the task at hand. Several answers explicitly address the trade-offs inherent to the means of displaying information in each mechanism: *"Each mechanism provides good readability for a specific kind of information but trades off readability regarding other aspects, e.g. the Enumerative mechanism makes it very easy to compare two specific variants but finding similarities and differences between all variants is hard."* This finding is aligned with the *cognitive fit* theory [85], according to which the performance during decision-making tasks depends on the suitability of the information being emphasized in the used representation, and that required for the task at hand. It highlights the need for flexible, purpose-tailored visualizations, as provided by paradigms such as virtual separation of concerns [46], and projectional editing [13].

Efficient use of space. Among the three considered representations, the annotative one is the most compact one, since it avoids showing any elements redundantly. Participants found the resulting compactness convenient: *"In the Annotative one you had all the information asked on the first look; comparing was easy since the different vars were all in the same diagram"*. Conversely, inefficient use of space was addressed as a disadvantage in the other mechanisms: *"[in compositional] with taking a brief look on to it it's nearly undoable to get the whole context in this overview"* and *"The enumerative variant is the easiest, but uses a lot of space"*.

Use of color. A design choice was to show the feature names in the annotative representation in distinguished colors, based on previous recommendations for code-level mechanisms [35]. Doing so balances out a disadvantage of annotative representations: the use of labels increases information density and visual crowding [86], thus affecting readability. In line with these findings, participants noted that it was *"easier to compare classes in Annotative because of colors"*, and that *"the colouring of annotative diagrams make [task type 1] really easy"*. We propose related recommendations in Sect. 5.4.

Scalability. In their assessment, several participants extrapolated from the considered case to more complex ones. *"[In enumerative,] although you need more models/space, you can see everything relatively easy. However, if you have maybe like 20 variants, enumerative is probably not the way to go!" [sic], "last the Enumerative, the 6*

Table 4: Preference distributions of our participants.

Task type	Ann	Comp	Enum	None
1: Tracing el.s to variants	50.7%	13.7%	34.2%	1.4%
2: Comparing two variants	26.0%	15.1%	57.5%	1.4%
3: Comparing all variants	43.8%	12.3%	42.5%	1.4%

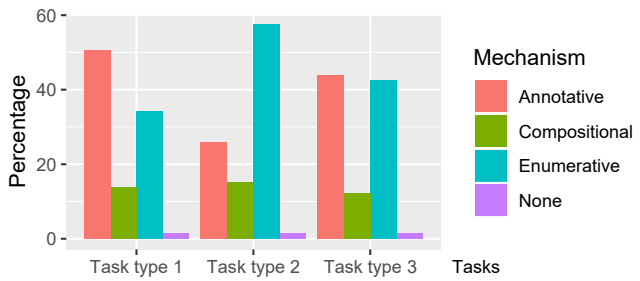

Figure 7: Preference distributions of our participants.

variants were okay but when there are even more it is to much" [sic], and finally *"The problem with the Compositional was [the] bigger and more complex it gets, it is harder to understand in a short time"*.

> The participants preferred different mechanisms for different tasks. Annotative was preferred for comparing all variants, and for tracing elements to variants. Enumerative was largely preferred for comparing two variants to each other. A contributing factor to preferring annotative is its higher compactness; some inherent disadvantages could be balanced out by the use of colors.

5.4 Discussion and Recommendations

The objective and subjective differences between variability mechanisms observed in our study can be considered by tool and language developers for improving user experience, an important prerequisite for MDE adoption [2]. We discuss our findings in the light of derived recommendations.

Provide flexible, task-oriented representations. We find that there is no globally preferable variability mechanism—indeed, the "best" mechanism may depend on the task to be performed. Tool and language developers can support user performance and satisfaction by providing multiple representations, tailored to the task at hand. We propose to consider a spectrum of solutions, each trading off the desirable qualities *flexibility* and *simplicity*: As the most simple, but least flexible solution, one can augment a given representation with task-specific, read-only views, e.g., given an annotative representation, generate individual enumerated variants (or a subset thereof, see below). A second, more advanced solution is to make these additional representations editable, which offers more flexibility, but gives rise to a new instance of the well-known view-update problem [26]—-the particular challenge here is to deal with the implications of layout changes. The third, most advanced solution is *projectional editing* [13], in which developers interact with freely customizable representations of an underlying structure. Projectional editing offers the highest degree of flexibility, but poses a learning threshold to users for adapting to a new editing paradigm.

Support the simple solution, for appropriate use-cases. Our participants preferred the simple enumerative solution for a subset of tasks. While being commonly applied in practice (e.g., in 9 out of 23 cases studied by Tolvanen et al. [80]), this solution is inherently problematic: In small to moderate product lines, organizations struggle with the propagation of changes between cloned variants [67]. In large product lines, considering a distinct model for each of thousands of variants is simply infeasible. Instead, we suggest to address use-cases that involve a clearly defined subset of variants: In staged configuration processes [27, 68], such subsets are derived by incrementally reducing the variant space, thus obtaining partial configurations of the system. Variability viewpoints [48], which are applied at companies like Daimler, reduce the variant space based on the perspective of a specific stakeholder. To address theses use cases, we suggest to provide support for selecting and interacting with a subset of enumerated variants, while using a proper variability mechanism for maintaining the overall system.

Use colors, and use them carefully. In line with the existing literature [35], we find that colors can be helpful for mitigating the drawbacks of annotative techniques. However, relying on colors in an unchecked way is undesirable due to the prevalence of color-blindness. Up to 8% of males and 0.5% of females of Northern European descent are affected by red-green color blindness [19]. A recommendation for language and tool designers is to avoid representations that solely rely on color, and to use dedicated color-blindness simulators such as Sim Daltonism (https://michelf.ca/projects/sim-daltonism/) to check their tools.

6 THREATS TO VALIDITY

We discuss the threats to validity of our study, following the recommendations by Wohlin et al. [89].

External validity. Our experiment focuses on class models, a ubiquitous model type. We discuss representativeness and practical relevance in MDE contexts in Sect. 1. While studying a wider selection from the modeling language design space is left to future work, the qualitative data presented in Sect. 5.3 is by no means specific to class models, and yields a promising outlook on generalizability.

Another issue is the generalizability of our results to larger systems, specifically, systems with more variants and model elements. Since the number of variants grows exponentially with the number of features, the enumerative representation will eventually be outperformed by the other representations. We discuss possible roles for the enumerative representation in larger systems in Sect. 5.3. To avoid researcher bias, we selected systems that were not used before with a specific variability mechanism. Studying comprehension in larger models is desirable, but has some principle limitations with regard to the amount of information that participants can be exposed to in the scope of an experiment (participant fatigue).

Student participants can be representative stand-ins for practitioners in experiments that involve new development methods [65]. Specifically, our participants had limited prior experience with variability mechanisms. While considering a broader spectrum of experience levels would be worthwhile, we arguably focus on a critical population: In a given organization, consider the onboarding of a new team member with a similar experience level to our participants. Poor comprehension would pose a major hurdle to becoming productive, and, therefore, pose a risk for the organization.

Internal validity. Within-subject designs help to elicit a representative number of data points to support statistically valid conclusions. We addressed their drawbacks as follows: To address learning effects, we applied counterbalancing. Between the different groups, we distributed the order of variability mechanisms equally, while keeping the system and task order constant. To balance the assignment of participants to classes, we randomized the assignment.

Conclusion validity. We operationalized comprehensibility with comprehension tasks, arguing for the importance of the considered tasks in Sect. 4. The choice of tasks was informed by our pre-study, in which we encountered trade-offs regarding participant fatigue and confounding factors when using more demanding tasks (Sect. 3). Since we find significant performance differences between mechanisms, the difficulty level of our tasks seems appropriate; however, other tasks might exist (e.g. understanding a single feature and its context), and task completion could also be facilitated if users are supported by specialized tools (e.g. query engines). Generally, systematic knowledge on the design space of model comprehension

tasks would help to maximize realism in comprehension experiments, but such knowledge is currently lacking.

Our setup did not involve tools, representing an unavoidable trade-off: While having the participants use a tool environment would have been more realistic, it would have lead to confounding factors related to usability obstacles and participants' familiarity with the tool. Extending our research to consider the effect of tools on model comprehension is an important avenue for future work.

Colors were only used in the annotative representation, where their usefulness (for distinguishing elements from different variants) seems more obvious than in the compositional one (where such elements are already distinguished by being contained in different modules). A follow-up study for studying the impact of colors in different representations might provide additional insight.

Construct validity. Subjective measures are generally less reliable than objective ones. However, previous findings suggest that they are correlated with objective performance measurements [40]. In fact, we find an agreement between the subjective and objective measurements performed in our experiments.

7 RELATED WORK

Annotative vs. compositional. Annotative approaches are traditionally seen as inherently problematic. Spencer [72], for example, argues that #ifdef usage in C as a means to cope with variability is harmful, leading to convoluted, unreadable, and unmaintainable code (the infamous *"#ifdef hell"*). Spencer appeals to basic principles of good software engineering: explicit interfaces, information hiding, and encapsulation. Kästner et al. [47] argue that compositional approaches tend to promise advantages, which, however, only become manifest under rather specific assumptions. They emphasize that only empirical research can provide conclusive evidence.

Aleixo et al. [4] compare both mechanism types in the context of *Software Process Line* engineering, i.e., applying concepts and tools from SPL engineering to software processes. They compare two established tools: EPF Composer, which uses a compositional mechanism, and GenArch-P, which uses an annotative one. Similar to our conclusions, they report that the annotative mechanism performs better, especially with regards to a criterion the authors call *adoption*, i.e., how much knowledge is required to initially apply the mechanism.

Empirical studies of variability mechanisms. Krüger et al. [49] present a comparative experimental study of two variability mechanisms: decomposition into classes, and annotations of code sections. They find that annotations have a positive effect on program comprehension, while the decomposition approach shows no significant improvement and, in some cases, a negative effect. While these findings are in line with ours, this study focuses on Java programs, and compares the considered mechanisms to a different baseline, pure OO code without any traces of variants. In our case, we considered the frequent case in industry of copied and reused model variants.

Fenske et al. [36] present an empirical study based on revision histories from eight open-source systems, in which they study the effect of #ifdef preprocessors to maintainability. The analyze maintainability in terms of change frequency, which is known to be correlated with error-proneness and change effort. In contrast to the traditional belief, they find that a negative effect of #ifdefs to maintainability cannot be confirmed.

Feigenspan et al. [35] study the potential of background colors as an aid to support program comprehension of source code with #ifdef preprocessors. In three controlled experiments with varying tasks and program sizes, they find that background colors contributed to better program comprehension and were preferred by the participants. We base the use of color in our experiments on these findings, and confirm them for the previously unconsidered case of a model-level variability mechanism.

Empirical studies of model comprehension. Labunets et al. [51] study graphical and tabular models representations in security risk assessment. In two experiments, they find that participants prefer both representations to a similar degree, but perform significantly better when using the tabular one. The authors build on cognitive fit theory [85] to explain their findings: tables represent the data in a more suitable way for the considered task. Like we do, this study supports the need for task-tailored representations.

Nugroho [58] studies the effect of *level of detail* (LoD) on model comprehension. In an experimental evaluation with students, the author finds that a more detailed representation contributes to improved model understanding. Ramadan et al. [63] find a positive effect to comprehension of security and privacy aspects when graphical annotations are included in the considered models. Our results are in line with these findings, since the annotative mechanism includes the names of the associated variants as one point of additional information.

Acreţoaie et al. [3] empirically assess three model transformation languages with regard to comprehensibility. They consider a textual language and two graphical ones, one of which uses stereotype annotations to specify change actions in UML diagrams. They observe best completion times and lowest cognitive load when using the graphical language with annotations, and best correctness when using the textual language. Studying this trade-off further, by studying variability mechanisms in graphical and textual representations, would be an interesting extension of our work.

8 CONCLUSION

We presented the results of a controlled experiment, in which we studied the effect of the two variability mechanisms, representative for the two fundamental mechanism types—annotative and compositional ones—on model comprehensibility. We conducted the study with 73 student participants with relevant background knowledge. For the scope of models in the size of our examples and similar tasks, we present and discuss the following main conclusions:

- The annotative mechanism does not affect comprehensibility for any task.
- The compositional mechanism can impair comprehensibility in tasks that require a good overview of all variants.
- The preferred variability mechanism depends on the task at hand.

We present several recommendations to language and tool developers. We consider a spectrum of solutions to maintain multiple task-tailored representations, especially in contexts of large systems where maintaining a separate model per variant is infeasible. We endorse the recommendation to use colors for improving comprehension in annotative variability, and discuss its limitations.

We envision three directions of future work. First, we want to understand the effect of tools to model comprehension. Second, we wish to systematically explore the space of typical tasks during model comprehension. Additional experiments would allow us to come up with a catalog of task-specific recommendations for variability mechanism use. Third, we are interested in broadening the scope of our experiments to take different modeling languages into account, including textual ones, which represent a middle ground between traditional programming languages and graphical modeling languages, and transformation languages, for which many different reuse mechanisms have recently been developed [20, 75].

Acknowledgement. We thank the reviewers and the participants from our pre- and main experiment.

REFERENCES

[1] Hervé Abdi. 2007. Bonferroni and Šidák corrections for multiple comparisons. *Encyclopedia of measurement and statistics* 3 (2007), 103–107.

[2] Silvia Abrahão, Francis Bourdeleau, Betty Cheng, Sahar Kokaly, Richard Paige, Harald Stöerrle, and Jon Whittle. 2017. User experience for model-driven engineering: Challenges and future directions. In *MODELS*. IEEE, 229–236.

[3] Vlad Acreţoaie, Harald Störrle, and Daniel Strüber. 2018. VMTL: a language for end-user model transformation. *Software & Systems Modeling* 17, 4 (2018), 1139–1167.

[4] Fellipe Araújo Aleixo, Marília Aranha Freire, Daniel Alencar da Costa, Edmilson Campos Neto, and Uirá Kulesza. 2012. A Comparative Study of Compositional and Annotative Modelling Approaches for Software Process Lines. In *SBES*. 51–60.

[5] Sanaa Alwidian and Daniel Amyot. 2019. Union Models: Support for Efficient Reasoning about Model Families over Space and Time. In *SAM*. Springer, 200–218.

[6] Carsten Amelunxen, Elodie Legros, Andy Schürr, and Ingo Stürmer. 2007. Checking and enforcement of modeling guidelines with graph transformations. In *AGTIVE*. Springer, 313–328.

[7] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. 2014. Modularizing Triple Graph Grammars Using Rule Refinement. In *FASE*. 340–354.

[8] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. Feature-Oriented Software Product Lines: Concepts and Implementation. (2013).

[9] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. 2009. Model superimposition in software product lines. In *ICMT*. Springer, 4–19.

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

[11] Wesley KG Assunção, Silvia R Vergilio, and Roberto E Lopez-Herrejon. 2017. Discovering software architectures with search-based merge of UML model variants. In *ICSR*. Springer, 95–111.

[12] The authors. 2020. Online appendix to "Variability Representations in Class Models: An Empirical Assessment". https://figshare.com/projects/Online_Appendix_to_Variability_Representations_in_Class_Models_An_Empirical_Assessment_/86207.

[13] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *ICSE*. IEEE.

[14] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information systems* 35, 6 (2010), 615–636.

[15] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She. 2014. Variability mechanisms in software ecosystems. *Information and Software Technology* 56, 11 (2014), 1520–1535.

[16] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. 2020. The State of Adoption and the Challenges of Systematic Variability Management in Industry. *Empirical Software Engineering* 25 (2020), 1755–1797. Issue 3.

[17] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. 2010. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 173–174.

[18] Daniel Antonio Callegari and Ricardo Melo Bastos. 2007. Project management and software development processes: integrating RUP and PMBOK. In *ICSEM*. IEEE, 1–8.

[19] Xin Bei V Chan, Shi Min S Goh, and Ngiap Chuan Tan. 2014. Subjects with colour vision deficiency in the community: what do primary care physicians need to know? *Asia Pacific Family Medicine* 13, 1 (2014), 10.

[20] Marsha Chechik, Michalis Famelis, Rick Salay, and Daniel Strüber. 2016. Perspectives of model transformation reuse. In *iFM*. Springer, 28–44.

[21] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. 2010. Abstract delta modeling. *CW Reports* (2010).

[22] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE*. ACM, 335–344.

[23] Alberto Colombo, Ernesto Damiani, Fulvio Frati, Sergio Oltolina, Karl Reed, and Gabriele Ruffatti. 2008. The use of a meta-model to support multi-project process measurement. In *APSEC*. IEEE, 503–510.

[24] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, et al. 2018. Concern-oriented language development (cold): Fostering reuse in language engineering. *Computer Languages, Systems & Structures* 54 (2018), 139–155.

[25] Krzysztof Czarnecki and Michał Antkiewicz. 2005. Mapping features to models: A template approach based on superimposed variants. In *GPCE*. Springer, 422–437.

[26] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. 2009. Bidirectional transformations: A cross-discipline perspective. In *ICMT*. Springer, 260–283.

[27] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Staged configuration through specialization and multilevel configuration of feature models. *Software process: improvement and practice* 10, 2 (2005), 143–169.

[28] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying feature-based model templates against well-formedness OCL constraints. In *GPCE*. ACM, 211–220.

[29] Joost CF De Winter and Dimitra Dodou. 2010. Five-point Likert items: t test versus Mann-Whitney-Wilcoxon. *Practical Assessment, Research & Evaluation* 15, 11 (2010), 1–12.

[30] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.

[31] Abir El Yamami, Souad Ahriz, Khalifa Mansouri, Mohammed Qbadou, and El Hossein Illousamen. 2017. Representing IT projects risk management best practices as a metamodel. *Engineering, Technology & Applied Science Research* 7, 5 (2017), 2062–2067.

[32] EUROCONTROL and Federal Aviation Administration. 2019. AIXM 5.1.1 Specification. http://aixm.aero/page/aixm-51-specification.

[33] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wąsowski. 2014. Sound merging and differencing for class diagrams. In *FASE*. Springer, 63–78.

[34] David Faitelson and Shmuel S. Tyszberowicz. 2017. UML diagram refinement (focusing on class- and use case diagrams). In *ICSE*. 735–745.

[35] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. 2013. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering* 18, 4 (2013), 699–745.

[36] Wolfram Fenske, Sandro Schulze, and Gunter Saake. 2017. How preprocessor annotations (do not) affect maintainability: a case study on change-proneness. In *GPCE*, Vol. 52. 77–90.

[37] Rick Flores, Charles Krueger, and Paul Clements. 2012. Mega-Scale Product Line Engineering at General Motors. In *Proc. SPLC*.

[38] Chris Gane. 2001. Process management: integrating project management and development. In *New directions in project management*. Auerbach Publications, 81–96.

[39] Ivan Garcia, Carla Pacheco, Magdalena Arcilla, and Neira Sanchez. 2016. Project Management in Small-Sized Software Enterprises: A Metamodeling-Based Approach. In *Trends and Applications in Software Engineering*. Springer, 3–13.

[40] Daniel Gopher and Rolf Braune. 1984. On the psychophysics of workload: Why bother with subjective measures? *Human Factors* 26, 5 (1984), 519–532.

[41] Peng Guo, Yahui Li, Peng Li, Shuai Liu, and Dongya Sun. 2014. A UML Model to Simulink Model Transformation Method in the Design of Embedded Software. In *ICCIS*. IEEE, 583–587.

[42] Florian Heidenreich, Jan Kopcsek, and Christian Wende. 2008. FeatureMapper: mapping features to models. In *ICSE-Companion*. ACM, 943–944.

[43] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa, and Dominique Rieu. 2014. A model-driven approach for embedded system prototyping and design. In *ISRSP*. IEEE, 23–29.

[44] Ákos Horváth, István Ráth, and Rodrigo Rizzi Starr. 2015. Massif-the love child of Matlab Simulink and Eclipse. *EclipseCon NA* (2015).

[45] Byron Jones and Michael G Kenward. 2003. *Design and analysis of cross-over trials*. Chapman and Hall/CRC.

[46] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *ICSE*. ACM, 311–320.

[47] Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The road to feature modularity?. In *SPLC Companion*. 5.

[48] Tobias Kaufmann, Christian Manz, and Thorsten Weyer. 2014. Extending the SPES Modeling Framework for Supporting Role-specific Variant Management in the Engineering Process of Embedded Software. In *SE*. 77–86.

[49] Jacob Krüger, Gul Calikli, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. *FSE*.

[50] Vinay Kulkarni, R Venkatesh, and Sreedhar Reddy. 2002. Generating enterprise applications from models. In *OOIS*. Springer, 270–279.

[51] Katsiaryna Labunets, Fabio Massacci, Federica Paci, Sabrina Marczak, and Flávio Moreira de Oliveira. 2017. Model comprehension for security risk assessment: an empirical comparison of tabular vs. graphical representations. *Empirical Software Engineering* 22, 6 (2017), 3017–3056.

[52] Philip Langer, Tanja Mayerhofer, Manuel Wimmer, and Gerti Kappel. 2014. On the usage of UML: Initial results of analyzing open UML models. *Modellierung* (2014).

[53] Elodie Legros, Wilhelm Schäfer, Andy Schürr, and Ingo Stürmer. 2007. 14 MATE-A Model Analysis and Transformation Environment for MATLAB Simulink. In *Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*. Springer, 323–328.

[54] Jabier Martinez, Tewfik Ziadi, Tegawende F Bissyande, Jacques Klein, and Yves Le Traon. 2015. Automating the extraction of model-based software product lines from model variants. In *ASE*. IEEE, 396–406.

[55] Jean Melo, Claus Brabrand, and Andrzej Wasowski. 2016. How does the degree of variability affect bug finding?. In *ICSE*. 679–690.

[56] Douglas C Montgomery. 2017. *Design and analysis of experiments*. John Wiley & Sons.

[57] Nathalie Moreno, Piero Fraternali, and Antonio Vallecillo. 2007. WebML modelling in UML. *IET software* 1, 3 (2007), 67–80.

[58] Ariadi Nugroho. 2009. Level of detail in UML models and its impact on model comprehension: A controlled experiment. *Information and Software Technology* 51, 12 (2009), 1670–1685.

[59] Sven Peldszus, Daniel Strüber, and Jan Jürjens. 2018. Model-based security analysis of feature-oriented software product lines. In *GPCE*. ACM, 93–106.

[60] Marian Petre. 2013. UML in practice. In *ICSE*. IEEE, 722–731.

[61] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL–A Delta-Based Modeling Framework for Software Product Line Engineering. In *ASE*. IEEE, 852–857.

[62] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[63] Qusai Ramadan, Daniel Strüber, Mattia Salnitri, Jan Jürjens, Volker Riediger, and Steffen Staab. 2020. A semi-automated BPMN-based framework for detecting conflicts between security, data-minimization, and fairness requirements. *Software and Systems Modeling* (2020), 1–37.

[64] Julia Rubin and Marsha Chechik. 2013. N-way model merging. In *FSE*. ACM, 301–311.

[65] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *ICSE*. 666–676.

[66] Ina Schaefer. 2010. Variability Modelling for Model-Driven Development of Software Product Lines. In *VaMoS*. 85–92.

[67] Alexander Schlie, David Wille, Sandro Schulze, Loek Cleophas, and Ina Schaefer. 2017. Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and Its Evaluation. In *SPLC*. 215–224.

[68] Julia Schroeter, Peter Mucha, Marcel Muth, Kay Jugel, and Malte Lochau. 2012. Dynamic configuration management of cloud-based applications. In *SPLC*. 171–178.

[69] Felix Schwägerl, Thomas Buchmann, and Bernhard Westfechtel. 2015. SuperMod—A model-driven tool that combines version control and software product line engineering. In *ICSOFT*, Vol. 2. IEEE, 1–14.

[70] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Deltaecore-A model-based delta language generation framework. *Modellierung 2014* (2014).

[71] Hyun Seung Son, Woo Yeol Kim, Robert YoungChul Kim, and Hang-Gi Min. 2012. Metamodel design for model transformation from Simulink to ECML in cyber physical systems. In *Computer Applications for Graphics, Grid Computing, and Industrial Environment*. Springer, 56–60.

[72] Henry Spencer and Geoff Collyer. 1992. #ifdef Considered Harmful, or Portability Experience with C News. In *USENIX*.

[73] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. 2006. *Model-driven software development: technology, engineering, management*. John Wiley & Sons.

[74] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[75] Daniel Strüber and Anthony Anjorin. 2016. Comparing Reuse Mechanisms for Model Transformation Languages: Design for an Empirical Study.. In *HuFaMo@ MoDELS*. 27–32.

[76] Daniel Strüber, Jacob Krüger Mukelabai Mukelabai, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *SPLC*. 26:1–26:12.

[77] Daniel Strüber and Stefan Schulz. 2016. A tool environment for managing families of model transformation rules. In *ICGT*. Springer, 89–101.

[78] Ingo Stürmer and Dietrich Travkin. 2007. Automated transformation of MATLAB simulink and stateflow models. In *OMER*. 57–62.

[79] Gabriele Taentzer, Rick Salay, Daniel Strüber, and Marsha Chechik. 2017. Transformations of software product lines: A generalizing framework based on category theory. In *MODELS*. IEEE, 101–111.

[80] Juha-Pekka Tolvanen and Steven Kelly. 2019. How Domain-specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases. In *SPLC*. 24:1–24:9.

[81] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.

[82] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. 2001. On the notion of variability in software product lines. In *ICSA*. IEEE, 45–54.

[83] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[84] Steffen Vaupel, Gabriele Taentzer, Jan Peer Harries, Raphael Stroh, René Gerlach, and Michael Guckert. 2014. Model-driven development of mobile applications allowing role-driven variants. In *MODELS*. Springer, 1–17.

[85] Iris Vessey. 1991. Cognitive fit: A theory-based analysis of the graphs versus tables literature. *Decision Sciences* 22, 2 (1991), 219–240.

[86] David Whitney and Dennis M Levi. 2011. Visual crowding: A fundamental limit on conscious perception and object recognition. *Trends in cognitive sciences* 15, 4 (2011), 160–168.

[87] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[88] David Wille, Tobias Runge, Christoph Seidl, and Sandro Schulze. 2017. Extractive software product line engineering using model-based delta module generation. In *VaMoS*. ACM, 36–43.

[89] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.