Bridging the Gap between Academia and Industry: Transforming the Universal Variability Language to pure::variants and Back

Dario Romano
LIT CPS Lab
Johannes Kepler University Linz
Linz, Austria
dario.romano@jku.at

Kevin Feichtinger
LIT CPS Lab
Johannes Kepler University Linz
Linz, Austria
kevin.feichtinger@jku.at

Danilo Beuche
pure-systems GmbH
Madgeburg, Germany
danilo.beuche@pure-systems.com

Uwe Ryssel
pure-systems GmbH
Madgeburg, Germany
uwe.ryssel@pure-systems.com

Rick Rabiser
CDL VaSiCS, LIT CPS Lab
Johannes Kepler University Linz
Linz, Austria
rick.rabiser@jku.at

ABSTRACT
In the last 30 years, many variability modeling approaches have been developed and new ones are still developed regularly. Most of them are only described in academic papers, only few come with tool support. The sheer plethora of approaches, all differing in terms of scope and expressiveness, makes it difficult to assess their properties, experiment with them and find the right approach for a specific use case. Implementing transformations between variability modeling approaches or importers/exporters for tools can help, but are hard to realize without information loss. In this paper, we describe how we derived and implemented transformations between the academically developed Universal Variability Language and the commercially developed pure::variants tool, with as little information loss as possible. Our approach can also be used to optimize constraints, e.g., reduce their number without an effect on the configuration space, using particular capabilities pure::variants provides. Also, via an existing variability model transformation approach, which uses UVL as a pivot language, we enable the transformation of FeatureIDE feature models, DOPLER decision models, and Orthogonal Variability Models into/from pure::variants and back. With our approach, we work towards bridging the gap between academic and industrial variability modeling tools and enable experiments with the different capabilities these tools provide.

CCS CONCEPTS
• Software and its engineering → Software product lines.

KEYWORDS
Software product lines, variability modeling, variability model transformations, variability modeling tools.

1 INTRODUCTION
In Software Product Lines (SPLs) variability models are used to explicitly capture the commonalities and variabilities of a set of software-intensive systems [7]. In the last 30 years, several variability modeling approaches have been developed [2, 7, 11, 21, 35, 37], which all have benefits and drawbacks. Most approaches are either based on the Feature-Oriented Domain Analysis (FODA) approach by Kang et al. [25] (feature modeling) or on the Synthesis method [12] (decision modeling) [13]. Other approaches include Orthogonal Variability Models (OVM) [33], UML-based approaches [22] or textual variability modeling approaches [4]. Also, industry and open source communities developed approaches, e.g., the KConfig language to configure the Linux Kernel [39] and the Component Definition Language (CDL) [8] to manage the variability of the eCos [43] operating system. Despite multiple attempts to unify variability modeling approaches, e.g., the Common Variability Language (CVL) [23], the Universal Variability Language (UVL) [40] or ISO-26558 [24], no standard variability modeling approach exists.

Even within one type of approach, multiple variants exist, i.e., not only one feature modeling approach exists [37, 38]. For instance, in feature modeling one can use the academic approaches/tools FeatureIDE [30], FAMA [5] or UVL [40], or the commercially developed approach used in the tool pure::variants [34]. For researchers and practitioners planning to utilize a variability modeling approach, it is challenging to pick the right approach and tool for their needs. As a result, both industry and academia frequently develop new approaches instead of building on existing ones [18].

Most existing variability modeling approaches have been shown to be applicable and useful for a certain domain and use case. Few provide well maintained tool support [2], which hinders their adoption by industry [9, 29]. Researchers investigated the transformation of variability models to increase the interoperability among
tools [36] or to enable automated analysis [3, 14] and experiments with different approaches [17]. For that purpose, we develop the variability artifact transformation approach TRAVART [18], which allows engineers to transform FeatureIDE feature models [30], DOPLER decision models [15] and OVM [33] between each other, using UVL [40] as a pivot language. UVL is developed by the MODEVAR community (https://modevar.github.io/) and represents a first step towards a unified variability modeling approach. Additionally, existing variability modeling tools like FeatureIDE [30] and pure::variants [10] support the import/export of other variability modeling approaches. The pure::variants approach was originally presented in 2008 and since then developed and extended to become a commercially successful variability modeling tool [10].

Most existing transformations only work for academic variability modeling approaches and often lose information during transformation [19]. Therefore, in this paper, we present how we derived and developed transformations between pure::variants [34] and UVL [40]. We derived a mapping table between the two approaches in close cooperation between researchers and pure-systems experts and identified expected information loss. Based on the mapping table we defined and implemented the necessary transformations [17], based on earlier work [18]. Via TRAVART and its pivot UVL model, we now enable transformations from all the supported approaches to and from pure::variants. Additionally, our approach can also optimize the set of constraints during transformation.

With our work, we aim to bridge the gap between academic and industrial variability modeling approaches and bring their tools closer together. We allow academic models to be imported into pure::variants and industrial variability models into academic tools to experiment with the capabilities the different tools provide. We evaluate our transformation approach by transforming a set of variability models of UVL and pure::variants models into each other. We verify the transformed models by manually inspecting the models and comparing their configuration spaces. This is done by using a sampler that creates a set of valid and invalid configurations in the original model. These samples are then programmatically verified in the models generated by the roundtrips. It is important to mention that for bigger models the configuration spaces may quite fied in the models generated by the roundtrips. It is important to mention that for bigger models the configuration spaces may quite fied in the models generated by the roundtrips. It is important to mention that for bigger models the configuration spaces may quite large. Thus, we define and implemented the necessary transformations [17], based on earlier work [18]. Via TRAVART and its pivot UVL model, we now enable transformations from all the supported approaches to and from pure::variants. Additionally, our approach can also optimize the set of constraints during transformation.

In the remainder of the paper, we discuss background on UVL, pure::variants, TRAVART and the information loss occurring in transformations in Section 2. We then present a mapping between the two languages and outline expected information loss due to language differences in Section 3. Section 4 describes the transformation algorithms and how we optimize and merge constraints during transformations. Section 5 presents the evaluation of our transformations. Section 6 presents related work. We conclude the paper and outline future work in Section 7.

2 BACKGROUND

We use the Weatherstation example from pure::variants as a running example to explain key concepts of pure::variants [34] and UVL feature models [40]. Figure 1 depicts the Weatherstation feature model in the pure::variants tool (left) and a UVL feature model in its textual representation (right). A feature in a feature model can have various semantics [6, 13]. In the Weatherstation example a feature represents a technical property, i.e., which sensors included, appearance properties, i.e., which languages should be supported and behavioral aspects, i.e., which warnings should be given. Features are either mandatory (e.g., Languages) or optional (e.g., Warnings). Hierarchy is a key concept in feature modeling. In pure::variants and UVL, features are organized in a tree structure, creating parent-child relationships, i.e., when a child feature is selected also its parent feature must be selected. A set of multiple child features can be organized in groups. A group then specifies a cardinality. For instance, in both languages a group can be an OR group (i.e., 1:n relationship, either the language Turkish or Dutch must be selected or both), an alternative group (i.e., 1:1 relationship; exactly one of the languages English and German must be selected), or a mutex selection (i.e., n:m relationship; of the three sensor types Temperature, WindSpeed and AirPressure at least 2 must be selected). Both languages allow constraints to narrow the configuration space. UVL supports propositional formula constraints (e.g., Heat implies Temperature). In pure::variants two types of constraints can be defined. Relations (e.g., Heat requires Temperature) or pvSCL constraints (not visible in Figure 1). Both languages provide additional concepts, which we explain with among a mapping table in Section 3.

Information loss is a major problem during transformations, due to concept and expressability differences in the approaches. As a result, information will be lost when transforming between approaches. In a recent study [19], we investigated the information loss between three variability modeling approaches. We identified four types of information loss, i.e., no information loss, structural loss, semantic loss, and configurability loss. No information loss happens when a modeling element can be transformed into an equivalent modeling element of another language. For instance, both languages in our study use features as unit of variability. A structural loss occurs when during transformation modeling elements...
are added to the models. For instance, constraints in the original model, which are split into multiple ones in the transformed models. A semantic loss happens when one language supports modeling constructs, which are not supported in another language, but these constructs do not change the configuration space. For instance, not each feature modeling approach does supports hidden features [38]. A configurability loss occurs when modeling elements are not supported in another language and the configuration space changes. For instance, the constraints language expressiveness differences. We use the classes to discuss the expected information loss of our transformations in detail using a mapping table in Section 3.

In our work, we build on TRAVART [18]. TRAVART is a variability artifact transformation approach, which enables engineers to transform variability models of different types, e.g., FeatureIDE feature models [30], UVL feature models [40], DOPLER decision models [15], Orthogonal Variability Models (OVM) [33]. The approach builds on three main components [17]: Variability meta-models of the supported variability modeling approaches, either provided as actual meta-models using EMF or descriptions in academic papers. Based on these meta-models one can define transformation operations between the two target variability approach and the UVL [40] feature modeling approach. TRAVART uses UVL [40] as a pivot language to avoid implementing n:m transformations for each approach [28]. The defined transformation operations are then implemented for the concrete new variability approach and UVL [40] in concrete transformation algorithms. In this work, we define a mapping table between the two feature modeling approaches pure::variants [34] and UVL [40]. Based on the mapping we specify transformation operations and implement them in transformation algorithms. We describe the transformations in Section 4.

3 COMPARING UVL AND PURE::VARIANTS

In Table 1 we provide an overview of the key elements of the two modeling approaches and information losses that occur when transforming between them. We employ a color coding scheme to show different types of losses [19] during the transformation, i.e., no information loss (green), structural loss (blue), semantic loss (yellow), and configurability loss (red). Both approaches organize features in a tree structure. Features in UVL can have the properties abstract and hidden. While pure::variants does not have a similar concept, these properties generate no difference in the configuration space of models created with the two approaches. When performing a roundtrip starting from UVL, these properties will be translated to attributes and reconstructed from there. Other than in UVL, features on the pure::variants side can contain a lot of metadata. This includes the possibility to give a feature display names in multiple languages, while keeping a unique identifier. Features may also have descriptions and each of their attributes can also have an additional description. Pure::variants can model solution space artifacts and their variability [34]. Because we want to focus on the variability of features, we consider this aspect of pure::variants out-of-scope for the current paper.

Both approaches allow attributes per-feature. These can be translated in both directions with almost no restrictions. Pure::variants also has flags and a type for attributes. These aspects cannot be translated to UVL, and cannot be generated when importing from UVL. The type and flags have no impact on the configuration space, however, and we therefore consider them only a semantic loss.

UVL uses groups to limit choices between sets of (sub-)features. Pure::variants allows for similar configurability, but provides it as Variation Type on a per-feature level. As a result, pure::variants does not allow the definition of multiple distinct groups of the same type under one feature, which can cause a configuration loss. Although this could be circumvented by hiding additional groups behind proxy-features it is currently not implemented in our transformation approach.

Both approaches employ constraints to manage complex cross-tree feature dependencies. Constraints in UVL are listed in a separate section at the end of the model, and are written as propositional logic formulas, with features being the literals. For a configuration to be valid, all constraints must hold. When using the pure::variants model, constraints are not separate entities. Constraints always belong to exactly one feature. Furthermore, pure::variants employs two different methods of defining constraints. The first method is called relations, which is a pre-defined set of easily comprehensible relations between features. For most relation-types pure::variants prompts the user with an error, should relations of these types be unfilled. We call these hard relations. Some other relation-types like recommends or discourages do only prompt the user with a warning. We refer to these relations as soft relations. All available relation-types can be found in Table 9.2 of the pure::variants user guide [34]. The other option to write constraints is writing them

### Table 1: Mapping UVL [40] concepts to pure::variants [30] (upper part) and vice versa (lower part).

<table>
<thead>
<tr>
<th>UVL Model</th>
<th>pure::variants Model</th>
<th>Roundtrip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
<td>Visible Name</td>
<td>as attribute</td>
</tr>
<tr>
<td></td>
<td>Mandatory</td>
<td>Mandatory Feature</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
<td>Optional Feature</td>
</tr>
<tr>
<td>Attributes</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Numeric</td>
<td>Numeric</td>
</tr>
<tr>
<td></td>
<td>Vector</td>
<td>Vector</td>
</tr>
<tr>
<td>Group</td>
<td>Or</td>
<td>Or</td>
</tr>
<tr>
<td></td>
<td>Alternative</td>
<td>Alternative</td>
</tr>
<tr>
<td></td>
<td>Constraints</td>
<td>Constraints</td>
</tr>
<tr>
<td></td>
<td>Descriptions</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td>Metadata</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td>Equivalence</td>
<td>Equivalence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pure::variants Model</th>
<th>UVL Model</th>
<th>Roundtrip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
<td>Visible Name</td>
<td>as attribute</td>
</tr>
<tr>
<td></td>
<td>Mandatory</td>
<td>Mandatory Feature</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
<td>Optional Feature</td>
</tr>
<tr>
<td>Attributes</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Numeric</td>
<td>Numeric</td>
</tr>
<tr>
<td></td>
<td>Vector</td>
<td>Vector</td>
</tr>
<tr>
<td>Group</td>
<td>Or</td>
<td>Or</td>
</tr>
<tr>
<td></td>
<td>Alternative</td>
<td>Alternative</td>
</tr>
<tr>
<td>Constraint</td>
<td>Hard Relation</td>
<td>Constraint</td>
</tr>
<tr>
<td></td>
<td>Soft Relation</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

#### Constraints

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>No Information Loss</th>
<th>Structural Loss</th>
<th>Semantic Loss</th>
<th>Configurability Loss</th>
</tr>
</thead>
</table>

[34]
in a domain-specific language called pvSCL. Constraints in this language can be very complex, and even involve computations. The UVL constraint language allows to write logic terms using the operators not, and, or, implies and equals. Parenthesis can also be used in UVL constraints. Because all these operators are also available in pvSCL constraints, we consider the entire UVL constraint language a subset of pvSCL. The translation of UVL constraints to pure::variants will be discussed in further detail in Section 4. Considering that in many cases we have to re-structure constraints to match them to relations, importing constraints to pure::variants generally entails structural loss. If the translation to constraints is completely skipped, and everything is directly transformed into pvSCL constraints instead, it is possible to have a completely lossless roundtrip. When starting the roundtrip from pure::variants, the transformations are more restrictive. On the one hand, hard relations can in all cases be translated to constraints in UVL. Soft relations on the other hand cannot be translated, because there is no equivalent concept in UVL. While pure::variants has an option to treat soft relations as hard relations, they are not meant to alter the configuration space. Therefore we consider this a semantic loss. Since pvSCL has more ways to describe constraints than the UVL constraint language does, we consider translating pvSCL constraints written in pure::variants to UVL to introduce a potential configurability loss.

4 TRANSFORMATION

In this section we describe in detail the methods employed to transform between UVL and pure::variants considering the previously mentioned differences between the two approaches. Here we focus on the key algorithms and the optimizations performed, which merge constraints into each other to reduce their overall amount, potentially making the model less complicated. We provide an overview of all algorithms in an online appendix 1.

4.1 From UVL to pure::variants

When transforming from UVL to pure::variants, the feature tree is transformed recursively as shown in Algorithm 1. By traversing the tree, and moving the cardinality from groups to feature attributes, a feature tree can be quickly generated. We try to map constraints to pure::variants relations. Therefore, we analyze constraints in detail and perform a number of mandatory and optional transformations to allow for the best possible translation. We derive relations for constraints based on Table 9.2 in the pure::systems user handbook [34], showing logic formulas that are represented by these relations. When importing constraints from a UVL model, we need to analyze the structure of these constraints, and find ways to re-structure them such that they can be represented by the relations listed in this table. First thing to note is that all relations have a feature that owns said relation – represented by D in the table. The remaining features in the relations are targets, and they are always connected by the same logic operator ∧ or ∨ for a given relation. There is also only very limited use of negations in relations, as only conflicts and conflicts/any include negations for the right sides of their implications. Relations only provide a single Implies or Equals top-level logic operator. This is important to keep in mind, as constraints in UVL can have a more complex structure with multiple implications and equivalences, parenthesis, negations and any top-level logic operator.

![Algorithm 1: Recursively create pure::variants tree](https://doi.org/10.5281/zenodo.6642520)

To enable transformation of complex logic constraints into the rigid relation types, we employ multiple strategies of transforming these relations. In a first step, we check if there is exactly one implication or equivalence, and if it is the top level logic operator. If this is the case, we analyze the constraint in increasingly finer detail to determine which method of translation is suitable. The different outcomes of this analysis are (a) it can be directly translated into a relation, (b) it can be split into multiple constraints that are then separately analyzed again, (c) it can be logically de- and re-constructed, (d) or it needs to be modeled as a pvSCL constraint.

When trying to analyze a constraint, we first identify the top-level logic operator. If it is either an implication or an equivalence, we investigate both the right and left side of it to determine if the structure can be directly translated into a relation. This is usually the case if either side has exactly one feature, and all the literals on the other side are connected by the same logic operator.

$$((A \land B) \lor (A \lor C) \Rightarrow D) = \begin{cases} A \land B \Rightarrow D \\ A \lor C \Rightarrow D \end{cases} \quad (1)$$

$$A \land B \Rightarrow C \land D = \begin{cases} A \land B \Rightarrow C \\ A \land B \Rightarrow D \end{cases} \quad (2)$$

If the constraint is more complex, as shown in Algorithm 2, we attempt to split up the constraint into multiple logically equivalent constraints, and try to find a structure that can be translated to a relation, potentially splitting those new sub-constraints even further. To split the constraints, we use Formulas 1 and 2.

If terms or literals on the left side are connected by a logical or, it means that satisfying any of the terms leads to fulfilling the constraint. In that case, the constraint can be split by the logical or into multiple constraints, keeping the right side identical for all new constraints. Moreover, terms that are on the right side, and are connected by a logical and can be split as well. As the input on the left side remains the same for all new constraints, all split...
Algorithm 2 Translate a constraint that has more than one literal on both sides of an `implies`/`equivalence` top-level operator.

```plaintext
1: function complexLeft(complexRight(c))
2: if `c.left().contains()` then
3:     cnfRight ← c.right().cnf()
4:     if cnfRight.contains(()) then
5:         for term ∈ cnfRight do
6:             if c.topLevelOperator == then
7:                 translateConstraints(c.left() ⇒ term))
8:         else
9:             translateConstraints(c.left() ⇒ term))
10:         end if
11:     end for
12: else
13:     if optimize then
14:         if reconstructRelation(c.cnf()) then
15:             createPvSCL(c)
16:         end if
17:     else
18:         createPvSCL(c)
19:     end if
20: end if
21: else
22:     if `c.left().contains()` then
23:         for literal ∈ c.left().literals() do
24:             if c.topLevelOperator == then
25:                 translateConstraints(literal ⇒ c.right()))
26:         else
27:             translateConstraints(literal ⇒ c.right()))
28:         end if
29:     end for
30: else
31:     cnfRight ← c.right().cnf()
32:     if cnfRight.topLevelOperator == then
33:         for term ∈ cnfRight do
34:             if c.topLevelOperator == then
35:                 translateConstraints(c.left() ⇒ term))
36:         else
37:             translateConstraints(c.left() ⇒ term))
38:         end if
39:     end for
40: end if
41: end if
42: end function
```

The base of this reconstruction is the alternative form of the logic implication, as shown in Formula 3. It allows us to get back from a CNF to the basic form of these three structures of relation-types, which all require an implication. For the `requires` and `requiredForAll` we can identify the single standout literal as the owner of the newly formed relation. For the `conflicts` case, it is sufficient to pick an arbitrary owner. It is important to note that `conflicts` is a bidirectional relation-type, which means that having all targets active implies that the owner is disabled. Fortunately, we do not have to create an additional relation, because pure::variants handles this bidirectionality automatically if it finds a `conflicts` constraint. If even this approach fails, we finally resort to converting the constraint in its pre-deconstructed form into a `pvSCL` constraint.

The above methods are only used in that order, if the constraints already has an implication or equivalence as top-level logic operator, and only contains negations, logical `or` and logical `and` on both sides of it. If the constraint shows the presence of multiple implications, equivalences, a negation, logical `or` or logical `and` as top-level logic operator, we directly resort to the de- and re-construction method with `pvSCL` as a fallback.

The relations that have been identified by the previously described methods are collected in a list to be subjected to an optional optimization algorithm. This optimization step is only performed across multiple relations for a single given owner feature. For example, if for a given owner feature, multiple `requiresAll` relations exist, it is possible to group the targets of all `requiresAll` relations into a single relation of the same type. This grouping is possible for the relation types `requiresAll`, `requiredFor`, `equalsAll` and `conflictsAny`. We call these mergeable relations.

For the relation types `requires`, `requiredForAll`, `equalsAny`, and `conflicts` the merging of targets is not possible. In a special corner case, where only one target is present, these non-mergeable relations can be merged to create a new, or be merged into a pre-existing, mergeable relation. In this case we merge `requires` relations to become a `requiresAll`, multiple `requiredForAll` to become a `requiredFor`, multiple `equalsAny` to become an `equalsAll` and multiple `conflicts` to become a `conflictsAny` relation. After performing these optimizations, or after skipping them, the relations are assigned to their owner features in the tree. This step concludes the transformation process.

### 4.2 From pure::variants to UVL

When transforming a pure::variants model to UVL we again traverse the tree recursively as demonstrated in Algorithm 3. Pure::variants defines cardinality on a feature-level, instead of a group-level like UVL. To deal with this, we have to look at children of a given feature, before we can create its corresponding groups. During this recursive descent, relations are translated into their logically equivalent form and collected into a list. Additionally, all `pvSCL` constraints are checked if they can be translated into a form compatible with propositional logic constraints in UVL, and stored with the translated relations. This check is performed by a simple pattern matching algorithm.

After all constraints have been collected, the user can perform an optional optimization on them. The changes performed on the constraints are described through the Formulas 1 and 2 applied in reverse order. This means we look for either identical terms on the left or right of a single implication or equivalence. If identical left constraints will be fulfilled if the conditions on the left are met. This means they are also implicitly connected by logical `and`. After splitting these constraints, the resulting constraints are themselves recursively analyzed again, until a relation-type structure is reached. In case no further splitting is possible, and no relation could be generated from the given constraint, we try to de- and reconstruct the constraint. In a first step, we create the Conjunctive Normal Form (CNF) of the (sub-)constraint. We identified three cases in which we can reconstruct relations from this CNF form. The cases are either that (a) exactly one literal is positive as in Formula 4, or (b) all literals are negated as demonstrated in Formula 5, (c) exactly one literal is negated as in Formula 6.

\[
(A \Rightarrow B) = (\overline{A} \lor B) \tag{3}
\]

\[
(\overline{A} \lor \overline{B} \lor C) \iff (A \land B \Rightarrow C) = C \text{ requiredForAll } B \land C \tag{4}
\]

\[
(\overline{A} \lor \overline{B} \lor \overline{C}) \iff (A \land B \Rightarrow \overline{C}) = C \text{ conflicts } B \land C \tag{5}
\]

\[
(\overline{A} \lor B \lor C) \iff (A \Rightarrow B \lor C) = A \text{ requires } B \lor C \tag{6}
\]
side terms are found for constraints with the same top-level logic operator, the right terms are connected via a logical and to form a new constraint with the original left side. The original constraints are removed. When identical right side terms are found for a number of constraints with the same top-level logic operator, the terms on the left side are accumulated and connected with a logical or. An example of applying these rules can be seen in Formulas 7 and 8.

\[
\begin{align*}
A \land B &\Rightarrow C \lor D \\
X \land Y &\Rightarrow C \lor D \\
A \land B &\Rightarrow C \lor D \\
A \land B &\Rightarrow (C \lor D) \land (X \lor Y)
\end{align*}
\]

Since these rules are only applicable to formulas that have a single implication or equivalence, all other types of constraints are exempt from this step. Considering we have created optional optimizations in both transformation paths, in case of a roundtrip we have implemented four different paths that are configurable.

4.3 Implementation

Often, the syntax of a constraint can vastly change how easy its implications can be understood by a human. It can also change how easy it is to maintain a given constraint in case of a change. Because our optimizations can perform a number of consecutive transformations in case of a roundtrip, the constraints and their structure can look very different from the original model. This is why we chose to make these optimizations optional. In case maintaining the structure of the constraints is more important to the user than having an overall smaller amount of constraints, they can opt out. It is also easier for a user to trace translated constraints if they are not subjected to multiple de- and reconstruction steps.

On the other hand, a user might be interested in finding a more concise way to phrase a number of simple constraints. That is a case where a roundtrip with all optimizations turned on, may vastly decrease the model size and overall complexity of the model. An example of this can be seen in the DellLaptopNotebook and BusyBox optimized roundtrip metrics in Table 2. We chose to integrate our approach as plugins for pure::variants, because it allows us easier access to the data structures used by pure::variants and because it provides ease of use for existing users of the program.

5 EVALUATION

We performed a feasibility study of our transformations between pure::variants models [34] and UVL models [40] and investigated the information lost during the transformations and the impact of our optimizations on the resulting transformed models. We defined the following three research questions (RQs). We performed roundtrips for a number of models, each native to their respective variability modeling approach. Starting the roundtrip from pure::variants yields UVL models as intermediate models, and vice versa. For each model, we report the number of features and constraints/relations for both the optimized and unoptimized transformation. To get more insight into the significance of the two separate steps of the roundtrip transformation, we also included metrics for the intermediate models. We also report the found information losses. Table 2 shows the results of our study.

**RQ1. Does our approach correctly transform pure::variants models into UVL models and vice versa?** In an experiment, we transformed six selected UVL models [40] and four pure::variants models [34] of varying size (in terms of number of features) and complexity (in terms of number of constraints and relations) into each other and verified their configuration spaces. We selected the variability models from online repositories\(^2\) and the examples delivered with the evaluation version of pure::variants.\(^3\) We used the configuration sampling [27, 42] capabilities of TRAVART [18] and sampled a representative set of valid and invalid configurations of the original model and verified that these configurations are also valid/invalid in the roundtrip models. We repeated this experiment in the other direction as well, i.e., sampling the roundtrip model and verified the sampled valid/invalid configurations to be valid/invalid in the original model. We report the results of our experiment in Section 5.1.

\(^2\)https://github.com/Universal-Variability-Language/uvl-models

Table 2: Information loss classes found in the roundtrip transformed variability models and the roundtrip model quality.

<table>
<thead>
<tr>
<th>Original Model</th>
<th>Intermediate Model Optimized</th>
<th>Intermediate Model Unoptimized</th>
<th>Roundtrip Metrics Optimized</th>
<th>Roundtrip Metrics Unoptimized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Features</td>
<td>#Constraints</td>
<td>#Features</td>
<td>#Constraints</td>
</tr>
<tr>
<td>UVL pure::variants</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BerkeleyDB</td>
<td>76</td>
<td>20</td>
<td>76</td>
<td>21</td>
</tr>
<tr>
<td>BusyBox</td>
<td>631</td>
<td>681</td>
<td>631</td>
<td>549</td>
</tr>
<tr>
<td>DellLaptopNotebook</td>
<td>47</td>
<td>105</td>
<td>47</td>
<td>21</td>
</tr>
<tr>
<td>Server</td>
<td>10</td>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>263</td>
<td>72</td>
<td>263</td>
<td>72</td>
</tr>
<tr>
<td>WikiMatrix</td>
<td>21</td>
<td>14</td>
<td>21</td>
<td>17</td>
</tr>
<tr>
<td>Windows 7</td>
<td>70</td>
<td>0</td>
<td>70</td>
<td>0</td>
</tr>
<tr>
<td>pure::variants UVL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>#Features</td>
<td>#Relations</td>
<td>#pSCL</td>
<td>#Relations</td>
</tr>
<tr>
<td></td>
<td>#Constraints</td>
<td></td>
<td>#Features</td>
<td>#Constraints</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coffee</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>WeatherStation</td>
<td>15</td>
<td>2</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Wagon</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

RQ2. How often does the transformation between pure::variants models into UVL models and vice versa lose information? We inspected the transformed models (intermediate and roundtrip models), manually and by using a model sampler, to verify the mapping table (cf. Table 1) and count their frequency of occurrence. Specifically, we reused the metrics presented in earlier work [19], to assess the lost information in the resulting models. We report our results in Section 5.2.

RQ3. What is the impact of the optimizations on the transformed variability models? We repeated the experiment outlined for RQ1, but during transformation we let our approach optimize the models in both directions (import/export from pure::variants). We again verified the resulting optimized variability models using configurations sampling and compared the resulting optimized models with their respective unoptimized counterpart from RQ1. We discuss the findings of our comparison in Section 5.3.

5.1 Transformations (RQ1)

Our transformations showed that each feature in pure::variants/UVL resulted in a feature in UVL/pure::variants. Also, for each constraint/relation a relation/constraint was created. Despite changing the structure of the models, the impact of the transformations is moderate. For both, pure::variants and UVL models, the number of features remained identical and only for some models (e.g., UVL models BerkeleyDB or WikiMatrix) the number of constraints slightly changed during the transformation (cf. Table 2). The configuration space of the UVL models could be preserved. For instance, we sampled 171 valid and 1503 invalid configurations for the Ubuntu feature model, which we could verify for the intermediate and roundtrip model. Also, from the roundtrip Ubuntu feature model, we could verify 166 valid and 1454 invalid configurations to be valid/invalid in the original UVL feature model. We found similar results in the other UVL models. For pure::variants we verified the model properties, including the possible variations, which remained identical for all four models. For instance, in the WeatherStation example we preserved the original 264 variations through the UVL transformation in the roundtrip model. Unfortunately, in the documentation of pure::variants [34] it is revealed that for the properties calculations restrictions and relations are ignored. As the models are smaller than the UVL models we verified them via manual inspection and verified that the feature tree, cardinalities and relations are preserved. For the models Coffee and Wagon we found differences in the pvSCL constraints (cf. Section 5.2).

The results of our feasibility study show that the (roundtrip) transformation of UVL models to pure::variants preserves the configuration space without a major transformation impact. Also, for the transformation of pure::variants models into UVL and back, we verified that the transformation is feasible and most concepts are transformed correctly. In our (small) sample we could transform two models completely and preserve all features, cardinalities and relations. For two others the expressiveness of pvSCL exceeded the capabilities of propositional logic in UVL.

5.2 Information Loss (RQ2)

For the transformation from UVL models to pure::variants we could only find structural losses. These losses were mostly introduced
by propositional logic constraints modeled with conjunctions and disjunctions, which then via relations were transformed into constraints using implies/equivalence operators. Additionally, our optimizing algorithms deconstruct or restructure constraints in the resulting transformed models, which also creates a \textit{structural loss}. These losses were expected from the mapping in Table 1.

When transforming pure::variants models into UVL models we lose more information. Regarding \textit{semantic loss}, we found that several attributes, e.g., display names and descriptions of features or constraints possible in pure::variants, could not be transformed into UVL and therefore got lost in the roundtrip transformation back to pure::variants. However, the configurability of the models remained equal for two of the pure::variants test models. In two other models (cf. \textit{Coffee} and \textit{Wagon}) we also found a \textit{configurability loss}. As expected by the mapping table (cf. Table 1), pvSCL constraints are more powerful than the propositional logic constraints possible with UVL. In these two models pvSCL constraints are used to perform computations and/or to perform constraints on attributes. Both are not supported by UVL and therefore get lost.

Our initial analysis of the UVL/pure::variants transformations showed the transformations conform to the expected information loss outlined in Table 1. Nevertheless, UVL models could be transformed into pure::variants without any \textit{configurability loss}. The total number of losses increased with the number of constraints.

### 5.3 Optimization Impact (RQ3)

During transformations which optimized the intermediate as well as the roundtrip model, the complexity, i.e., the number of constraints/relations could be reduced, without changing the configuration space of the models. For instance, for the feature model DellLaptopNotebook our optimizing strategy reduced the number of pvSCL constraints to 0 compared to 84 in the unoptimized one. Using configuration sampling we could verify 41 valid and 316 invalid configurations of the original feature model are also valid/invalid in the roundtrip model. For the other direction we sampled 33 valid and 136 invalid configurations in the roundtrip model and verified them to be valid/invalid in the original model. We found similar results for all other UVL models. Our optimizations did not impact the pure::variants models during their roundtrip transformation, as these models were too small (too few relations/constraints) to trigger any optimizations.

Our initial results of the feasibility study suggest that optimizing constraints can reduce the complexity of a variability model, without losing configurability. However, further research must be conducted towards other variability modeling approaches and the traceability and understandability of the resulting set of constraints.

### 5.4 Threats to Validity

We selected the UVL models and pure::variants models ourselves. Additionally, only for UVL models we had access to a variety of models. We carefully selected the models of both sides to have models of varying size and complexity, without introducing a bias towards UVL models. We still think our initial results provide evidence regarding the practical feasibility of UVL to pure::variants transformations. We plan to further test our evaluation with more and more complex models. Also, we plan to use TRAVART to demonstrate the import/export of other variability modeling approaches such as DOPLER decision models [15] or OVM [33] into/from pure::variants.

### 6 RELATED WORK

With this work, we increase the interoperability among different variability modeling tools [2], which has been investigated before. Either by directly transforming two variability modeling approaches to each other, e.g., Roos Frantz et al. [36] transformed feature models to OVM models, or by the development of import/export capabilities for concrete tools. For instance, FeatureIDE [30] supports importing and exporting from other feature modeling tools such as S.P.L.O.T [31], or approaches such as UVL [41]. Galindo et al. [20] worked towards a common configuration tool interface for multiple (types of) variability models. For custom-developed variability artifacts as Product Comparison Matrices [32] approaches have also been developed [1, 16]. Knüppel et al. [26] developed an algorithm that eliminates complex cross-tree constraints in feature models to ease the import of models into tools without the necessary capabilities. All these works highlight the need for academic as well as commercial variability modeling tools to import/export a variety of different (industrial) variability modeling approaches. Our work bridges the gap between academically developed and commercially developed variability modeling tools.

### 7 CONCLUSION AND FUTURE WORK

In this paper, we describe how we connected the academically developed UVL with the commercially developed pure::variants variant management tool. We outlined how we mapped UVL feature model concepts to the feature modeling approach used in pure::variants and analyzed the information lost during the transformation between the two languages. We implemented the outlined transformations as import/export capabilities of pure::variants and showed in a feasibility study that, despite some losses, a transformation between these languages is possible. Specifically, we showed that UVL models can be imported and exported into/from pure::variants without any \textit{configurability loss}. Also, we showed that, despite the exceeding expressiveness of pvSCL, pure::variants models can be exported into UVL models with minimal information loss. Additionally, we implemented optimization capabilities, which reduce the complexity of the resulting models. We investigated the impact of constraint optimizations during the transformations and showed that we reached the same configuration space with fewer constraints.

In future work, we aim to further investigate optimizing the transformations and also integrate the capabilities into our variability artifact transformation approach TRAVART. In a subsequent user study, we plan to investigate the readability and traceability of certain optimized constraints during the transformation from pure::variants to other variability modeling approaches.

### ACKNOWLEDGMENTS

The financial support by the Christian Doppler Research Association, the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.
REFERENCES


